Skript zu Modellierung und Simulation II (SoSe 2019)

Christian B. Mendl

8. August 2019

Inhaltsverzeichnis

Kapitel 8

Convolutional Neural Networks (CNNs)

8.1 Grundlagen

Referenz: http://neuralnetworksanddeeplearning.com/chap6.html

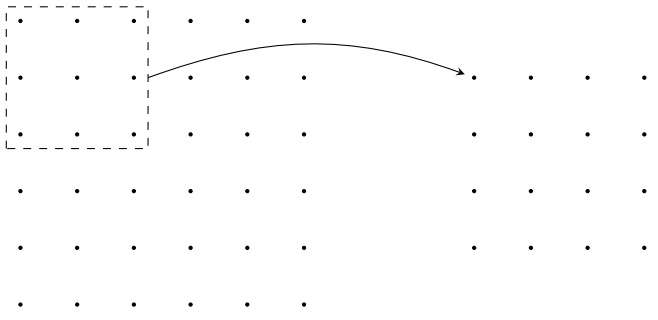
Bisher: $28^2 = 784$ Pixel der MNIST-Bilder im Eingabevektor aneinandergereiht, "fully connected layers" (d.h. jede Zelle ist gleichwertig mit allen Zellen der vorherigen Schicht verbunden) $\rightsquigarrow$ Verlust der räumlichen Struktur (2D-Nachbarschaftsbeziehung der Pixel)

Ansatz zur Berücksichtigung der räumlichen Struktur: **convolutional neural networks (CNNs)**

Bausteine: lokale rezeptive Felder, uniforme Gewichte und Bias-Werte, Pooling (optional)

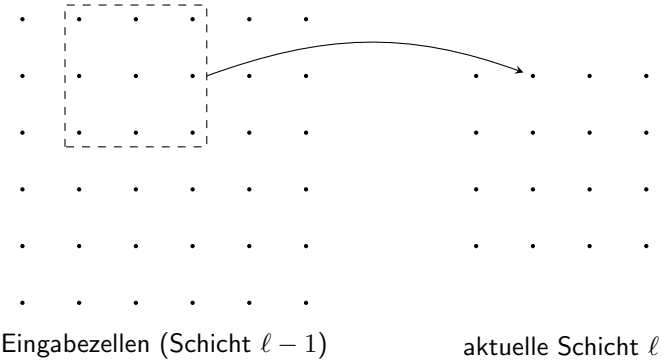
8.1.1 Lokale rezeptive Felder

Idee: Eingabe eines jeden Neurons in Schicht ℓ ist lokaler Ausschnitt der Eingabe ("rezeptives Feld") aus vorheriger Schicht $\ell - 1$, z.B. für einen 3×3 Ausschnitt:



Eingabezellen (Schicht $\ell - 1$) aktuelle Schicht ℓ

Verschiebung jeweils um "**stride length**" Pixel in horizontale oder vertikale Richtung, typisch ist stride length $= 1$ Pixel:



Für 6×6 Bilder und ein 3×3 rezeptives Feld $\rightsquigarrow 4 \times 4$ Zellen in erster versteckten Schicht, falls das rezeptive Feld nicht über Bildrand hinauslaufen soll.

8.1.2 Uniforme Gewichte und Bias-Werte

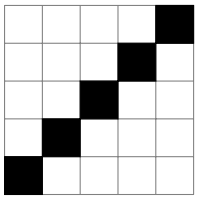
Idee: Verwende dieselbe Gewichtsmatrix und Bias-Werte für jedes Neuron ($\rightsquigarrow$ Translationsinvarianz): für ein $K \times K$ rezeptives Feld lautet die Ausgabe des (j_x, j_y)-ten Neurons in Schicht ℓ:

$$a^{\ell}_{j_x,j_y} = \sigma\left(\sum_{k_x=0}^{K-1}\sum_{k_y=0}^{K-1} w^{\ell}_{k_x,k_y} a^{\ell-1}_{j_x+k_x,j_y+k_y} + b^{\ell}\right), \tag{8.1}$$

d.h. eine Faltung[1] (englisch convolution) mit Faltungskern w^{ℓ} in zwei Dimensionen, gefolgt von der Aktivierungsfunktion σ. Im Gegensatz zu "fully connected layers" hängen $w^{\ell}_{k_x,k_y}$ und b^{ℓ} nicht von (j_x, j_y) ab.

Interpretation der Operation (8.1): "**feature map**", detektiert ein "Feature " der Eingabe, z.B. schräge Kante (für 5×5 Filter):

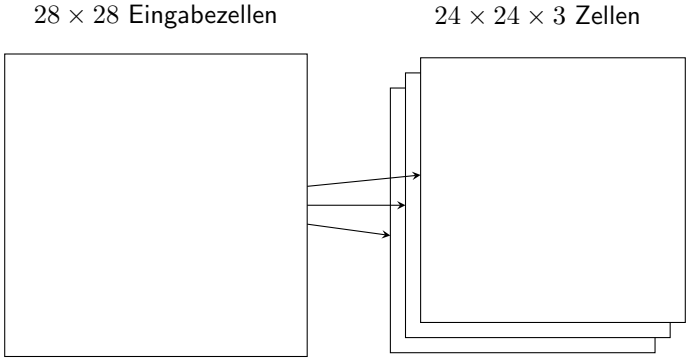
$$w^{\ell} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} =$$



Für Bildklassifizierung im Allgemeinen ist somit zu erwarten, dass eine einzelne "feature map" nicht ausreicht, sondern mehrere feature maps benötigt werden, mit jeweils eigenem Faltungskern und Bias-Wert.

Beispiel für entsprechende Netzwerk-Topologie mit 3 Features und 5×5 Faltungskern:

[1] Mathematisch würde man die Operation in (8.1) eigentlich als Kreuzkorrelation bezeichnen; diese ist aber äquivalent zu einer Faltung nach Uminterpretation des Faltungskerns $w^{\ell}_{k_x,k_y} \to w^{\ell}_{-k_x,-k_y}$.



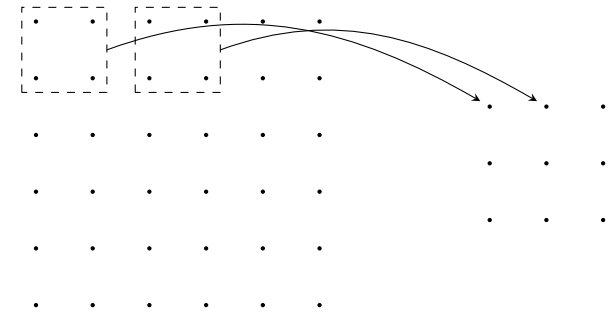
Anzahl Parameter für $K \times K$ Faltungskern: $\#\text{features} \cdot (K^2 + 1)$, eingesetzt für 30 Features und 5×5 Kern: 520

Vergleich mit bisheriger "fully connected" Topologie mit 30 Zellen in erster versteckter Schicht und 28×28 Eingabebild: Anzahl Parameter für Gewichtsmatrix und Bias-Vektor: $28^2 \cdot 30 + 30 = 23550 \gg 520$, d.h. eine Convolution-Schicht ist viel ökonomischer

8.1.3 Pooling

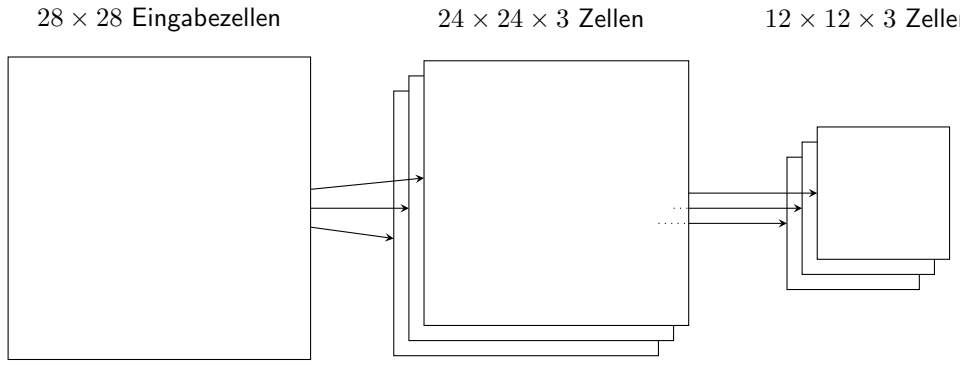
Idee: "Kondensiere" die Information der Convolutional Layer, "coarse graining" der räumlichen Auflösung (etwa bei Bildklassifikation: räumliche Information "wo im Bild" spielt bei Ausgabe keine Rolle mehr)

z.B. 2×2 max-Pooling: Maximum vier Werte in 2×2-Region
L2-Pooling: ℓ^2-Norm der vier Werte in 2×2-Region



Eine Pooling Layer reduziert also die Anzahl der Zellen in der nächsten Schicht, im Beispiel um den Faktor $2 \cdot 2 = 4$.

Somit: Topologie insgesamt für Convolution und Pooling, für obiges Beispiel:



8.2 Backpropagation für Convolutional Layers

Erinnerung: Operation einer Convolutional Layer in Gleichung (8.1):

$$a_j^\ell = \sigma\left(\sum_k w_k^\ell \, a_{j+k}^{\ell-1} + b^\ell\right),$$

wobei $j = (j_x, j_y) \in \mathbb{Z}^2$ und $k = (k_x, k_y) \in \mathbb{Z}^2$ zweidimensionale Indizes (analog für 3D, 4D, ...), und σ eine (nichtlineare) Aktivierungsfunktion; Summationsgrenzen von k sind genau die Dimensionen des Faltungskerns

Wie bisher: Kostenfunktion C hängt von Ausgabe a^L der letzten Schicht ab

Zur Vereinfachung der Notation lasse Schichtindex ℓ im Folgenden weg, bezeichne Eingabe mit $x = a^{\ell-1}$, Ausgabe mit $a = a^\ell$

Weitere Vereinfachung: fasse Aktivierungsfunktion σ als eigene Schicht auf $\rightsquigarrow$ Convolutional Layer berechnet lediglich linearen Anteil:

$$a_j = \sum_k w_k \, x_{j+k} + b \tag{8.2}$$

Kettenregel für Ableitungen $\rightsquigarrow$

$$\frac{\partial C}{\partial w_k} = \sum_j \frac{\partial C}{\partial a_j}\frac{\partial a_j}{\partial w_k} = \sum_j \frac{\partial C}{\partial a_j} x_{j+k}, \tag{8.3a}$$

$$\frac{\partial C}{\partial b} = \sum_j \frac{\partial C}{\partial a_j}\frac{\partial a_j}{\partial b} = \sum_j \frac{\partial C}{\partial a_j}, \tag{8.3b}$$

$$\frac{\partial C}{\partial x_m} = \sum_j \frac{\partial C}{\partial a_j}\frac{\partial a_j}{\partial x_m} = \sum_j \frac{\partial C}{\partial a_j} w_{m-j} = \sum_k \frac{\partial C}{\partial a_{m-k}} w_k. \tag{8.3c}$$

Für das vorletzte Gleichheitszeichen haben wir ausgenutzt, dass in $\partial a_j / \partial x_m = \partial(\sum_k w_k \, x_{j+k} + b)/\partial x_m$ nur der Summand für $j + k = m$, also $k = m - j$, übrig bleibt. Das letzte Gleichheitszeichen ist lediglich ein Umbenennen des Summationsindex.

Laut Gl. (8.3c) lässt sich die Ableitung bezüglich der Eingabe x wiederum als Faltungsoperation (zwischen $\partial C/\partial a_j$ und dem Faltungskern w) interpretieren; im Vergleich zu (8.2) ist jetzt der Faltungskern Punkt-gespiegelt

Entsprechender Pseudo-Code für Forward-Pass:

1: **function** CONV_FORWARD(x, w, b)

2: $a \leftarrow 0$ ▷ als Matrix mit Dimension der Ausgabe
3: **for** $i \leftarrow 0, 1, \ldots$ **do**
4: **for** $j \leftarrow 0, 1, \ldots$ **do**
5: $a[i, j] \leftarrow \mathrm{sum}\,(w * x[i : i + \mathit{width}, j : j + \mathit{height}]) + b$
6: $\mathit{cache} \leftarrow (x, w, b)$
7: **return** a, *cache*

wobei *width* und *height* die Dimensionen des Filters w sind und die Multiplikation komponentenweise verstanden wird[2]; *cache* sammelt die für die Backpropagation benötigten Variablen

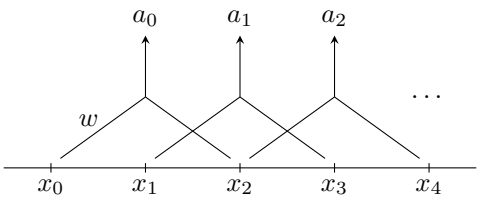
Der entsprechende Pseudo-Code für die Backpropagation verwendet die folgenden zusätzlichen Variablen mit Einträgen

$$\underbrace{\mathit{da}[i,j] = \frac{\partial C}{\partial a_{i,j}}}_{\text{vorgegeben}}, \quad \underbrace{\mathit{dx}[i,j] = \frac{\partial C}{\partial x_{i,j}}, \quad \mathit{dw}[k_1,k_2] = \frac{\partial C}{\partial w_k}, \quad \mathit{db} = \frac{\partial C}{\partial b}}_{\text{soll berechnet werden}}$$

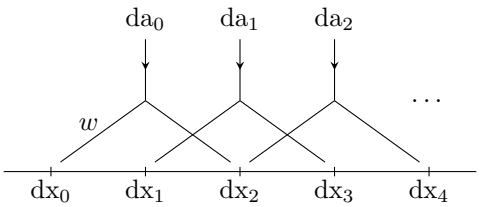
1: **function** CONV_BACKWARD(*da*, *cache*)
2: $(x, w, b) \leftarrow \mathit{cache}$
3: $\mathit{dx} \leftarrow 0$ ▷ als Matrix mit selber Dimension wie x
4: $\mathit{dw} \leftarrow 0$ ▷ als Matrix mit selber Dimension wie w
5: **for** $i \leftarrow 0, 1, \ldots$ **do**
6: **for** $j \leftarrow 0, 1, \ldots$ **do**
7: $\mathit{dw} \leftarrow \mathit{dw} + \mathit{da}[i, j] * x[i : i + \mathit{width}, j : j + \mathit{height}]$ ▷ Gl. (8.3a)
8: $\mathit{dx}[i : i + \mathit{width}, j : j + \mathit{height}]$
$\leftarrow \mathit{dx}[i : i + \mathit{width}, j : j + \mathit{height}] + \mathit{da}[i, j] * w$ ▷ Gl. (8.3c)
9: $\mathit{db} \leftarrow \mathrm{sum}(\mathit{da})$ ▷ Gl. (8.3b)
10: **return** $(\mathit{dx}, \mathit{dw}, \mathit{db})$

Schematische Darstellung in einer Dimension:

forward



backward



[2] In der Praxis sind die Dimensionen Höhe und Breite per Konvention oft vertauscht, d.h. die entsprechende Zeile im obigen Code für den Forward-Pass lautet dann $a[j, i] \leftarrow \mathrm{sum}\,(w * x[j : j + \mathit{height}, i : i + \mathit{width}]) + b$. Bei Matrizen entspricht der erste Index ja den Zeilen, und lässt sich somit als "Höhenindex" interpretieren.

Modifikation für mehrere features:

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

Im Forward-Pass:

for $f \leftarrow 0, 1, \ldots$ **do**
  for $i \leftarrow 0, 1, \ldots$ **do**
    for $j \leftarrow 0, 1, \ldots$ **do**
      $a[f, i, j] \leftarrow \mathrm{sum}\,(w[f] * x[i : i + \mathit{width}, j : j + \mathit{height}]) + b[f]$

Backpropagation analog

Modifikation für allgemein "stride length": ersetze

$$x[i : i + \mathit{width}, j : j + \mathit{height}]$$

durch

$$x[i * \mathit{stride} : i * \mathit{stride} + \mathit{width}, j * \mathit{stride} : j * \mathit{stride} + \mathit{height}]$$

8.3 Allgemeine Architektur von Convolutional Neural Networks

Benötige zusätzliche Dimension für Farbbilder als Eingabe: 3 Farbkanäle RGB (rot, grün, blau) $\rightsquigarrow$ Eingabe ist 3D Volumen:

$$C \times H \times W \qquad (\text{channel} \times \text{height} \times \text{width})$$

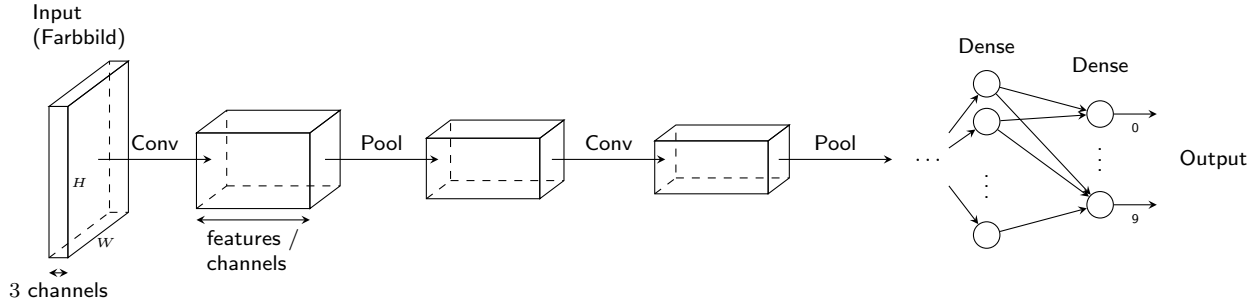
Die "hidden layers" sind ebenfalls dreidimensional, denn die "features" bilden eine zusätzliche Dimension.

Somit allgemein Operation einer convolutional layer:

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

wobei wie bisher j und k zweidimensionale (räumliche) Indizes sind.

Typische Architektur eines CNN:



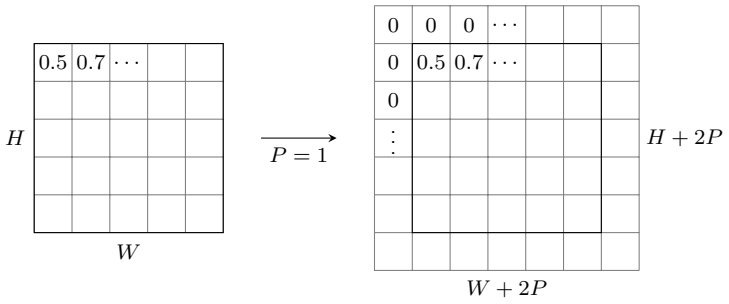
Bemerkungen:

- Pooling-Schichten werden oft ersetzt durch *stride* > 1 in der vorausgehenden convolutional layer
- Abfolge der verschiedenen Schicht-Typen, Anzahl features und weitere Parameter des Netzwerks sind je nach Anwendung unterschiedlich; oft heuristisches Ausprobieren

8.3.1 Zero Padding

Oft üblich: "padding" der Eingabe mit Nullen entlang Bild-Dimensionen; entsprechender Parameter P: Breite des Randes aus Nullen

Beispiel:



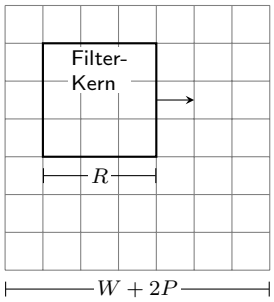
z.B. bei Faltung mit 3×3 Filter und $P = 1$, oder Faltung mit 5×5 Filter und $P = 2$: erhalte Ausgabe mit ursprünglicher Breite W und Höhe H (jeweils für *stride* $= 1$)

8.3.2 Ausgabedimension einer convolutional layer

Hier nur räumliche Bild-Dimension (Höhe und Breite); Anzahl der channels bzw. features sind separate Größen und werden hier nicht betrachtet

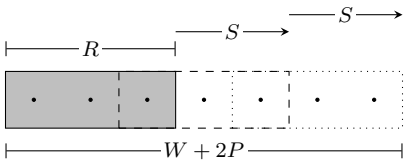
Entlang der x-Richtung (y analog):

W	Breite des Eingabe-Bilds
R	Breite des Filter-Kerns
P	Padding
S	stride length



Breite nach Padding: $W + 2P$

Breite der Ausgabe: Illustration für $R = 3$ und $S = 2$:



$$R + nS = W + 2P$$

wobei n: "wie oft weiterschieben"; somit:

$$n = \frac{1}{S}(W + 2P - R),$$

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

Insgesamt: Breite der Ausgabe:

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

8.4 Optimierte Implementierung der Faltungsoperation

Ziel: Laufzeitoptimierung der Berechnung einer convolutional layer

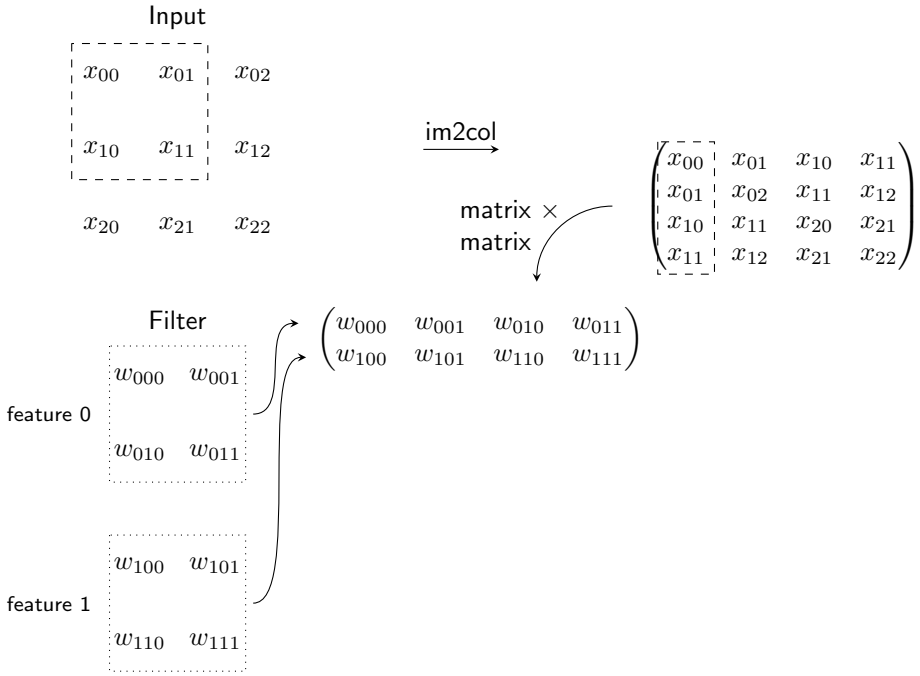
$$a_{f,j} = \sum_{c,k} w_{f,c,k}\, x_{c,j+k} + b_f$$

Naheliegende Idee: mittels FFT (siehe Hausaufgabe A15), aber:

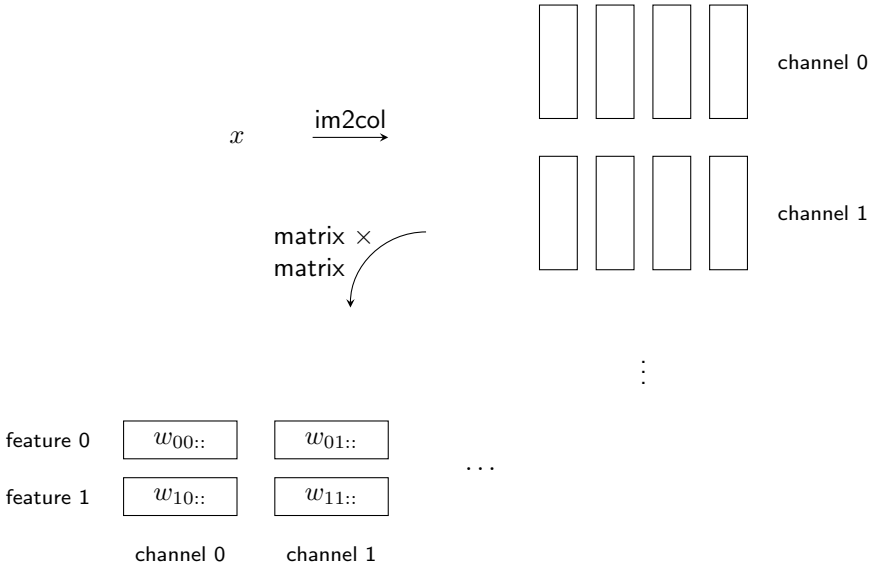
- Kann räumliche Lokalität der Filter nicht ausnutzen, denn Faltung mittels FFT benötigt Vektoren gleicher Dimension, d.h. müsste Filter-Kern "zero-padden", so dass er dieselbe Dimension wie das Eingabebild hat
- *stride* ≥ 2 technisch kompliziert umzusetzen mittels FFT

Stattdessen: Schreibe Faltungsoperation als Matrix-Matrix-Multiplikation mittels **im2col**-Operation um und verwende hochoptimierte BLAS-Funktion GEMM ("general matrix multiplication")

Illustration für $H = W = 3$ (Höhe und Breite des Eingabebilds) mit $C = 1$ channels, 2×2 Filter und $F = 2$ features:



Für mehrere channels:



Dimensionen allgemein:

Input

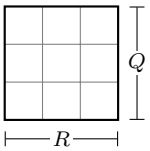
$$N \times C \times H \times W \xrightarrow{\text{im2col}} C\,Q\,R \times N\,W_{\text{out}}\,H_{\text{out}}$$

(mini-batch × channel × height × width)

Filter

$$F \times C \times Q \times R \longrightarrow F \times C\,Q\,R$$

(feature × channel × height × width)



Bemerkung: Praktische Umsetzung auf dedizierter Hardware (z.B. auf Graphikkarten): Ausgabematrix der im2col-Operation typischerweise nicht als ganze Matrix aufgestellt (würde sehr viel Speicher benötigen), sondern sequentielles Aufstellen der gerade benötigten Untermatrizen

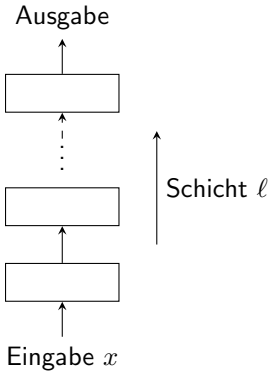
8.5 Anwendungsbeispiel: Artistic Style Transfer

Kapitel 9

Recurrent Neural Networks (RNNs)

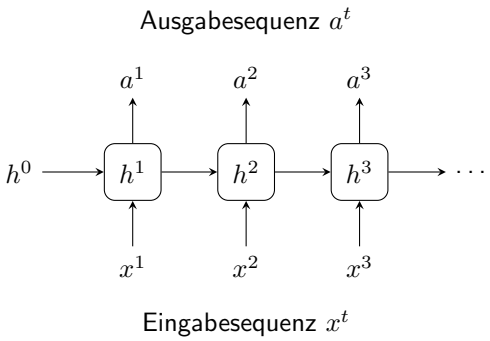
9.1 Grundlegende Architektur

Bisher: Feedforward-Netze

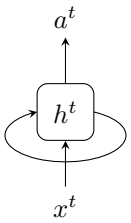


Problem: nicht angepasst an (zeitliche) Sequenz als Eingabe, z.B. Wörter in einem zu übersetzenden Text, Buchstaben bei Wortvervollständigung, Video-Streams, ...

RNNs: modifizierte Architektur:



Kurzschreibweise:



wobei h^t: **hidden state** zum Zeitpunkt t

 : selbe Zelle (interne Gewichtsmatrizen, Bias-Werte) für verschiedene Zeitschritte, nur Ein- und Ausgabedaten ändern sich

Aktueller hidden state hängt von vorherigem hidden state und aktueller Eingabe ab:

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

Anschauliche Interpretation: "zusammengefasste Information" über Input-Sequenz bis zum aktuellen Zeitpunkt t

Basisvariante eines RNN ("Vanilla RNN" bzw. Elman-RNN):

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

wobei $\tanh$ komponentenweise angewendet wird; Wh^{t-1} und Ux^t sind Matrix-Vektor-Multiplikationen.

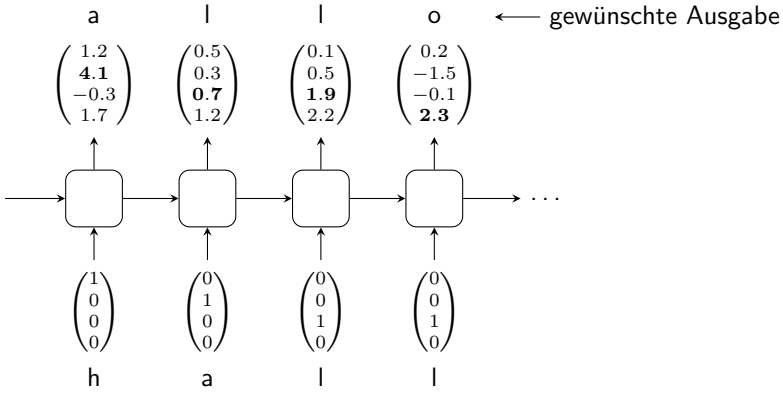
Ausgabe:

$$a^t = Vh^t$$

Beispielanwendung: Wortvervollständigung

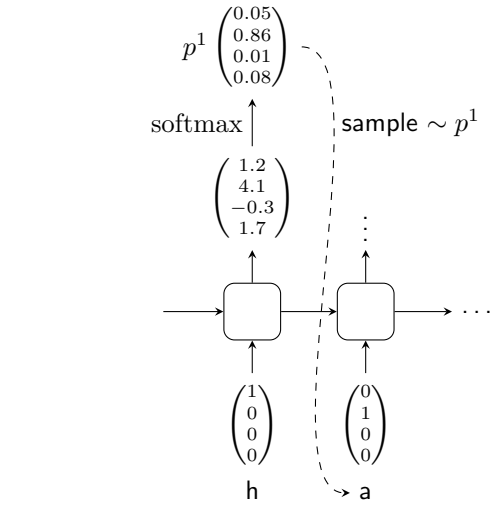
Vokabular: $\{\mathsf{h}, \mathsf{a}, \mathsf{l}, \mathsf{o}\}$

Trainieren:



In der Ausgabesequenz sollte der jeweils hervorgehobene Eintrag (entsprechend dem gewünschten Ausgabebuchstaben) größer sein als die anderen Einträge. Beim Ausgabevektor a^1 wurde das bereits erreicht (da 4.1 dominiert), aber beim Ausgabevektor a^2 noch nicht (da $0.7 < 1.2$).

Testen bzw. Auswerten: die Softmax-Funktion überführt die Einträge der Ausgabevektoren in eine Wahrscheinlichkeitsverteilung: $p^t = \mathrm{softmax}(a^t)$; anhand dieser wird anschließend der nächste Buchstabe ausgewürfelt:

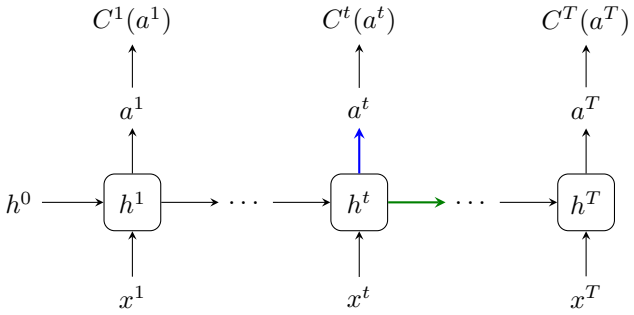


Somit: Netzwerk generiert Buchstabensequenz, die als Wort interpretiert werden kann

9.2 Backpropagation Through Time (BPTT)

Mathematik: gewöhnliche Berechnung des Gradienten (mittels Kettenregel)

zeitliche Sequenz bis zu einem maximalen Zeitpunkt T ausgeschrieben (für zeitlich unbeschränkte Sequenzen teilt man diese in disjunkte Blöcke der Länge T ein):



Erinnerung Vanilla-RNN:

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

Volle Kostenfunktion:

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

Gradient:

$$\frac{\partial C}{\partial h_k^t} = \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}} \frac{\partial h_{k'}^{t+1}}{\partial h_k^t} + \sum_j \frac{\partial C}{\partial a_j^t} \frac{\partial a_j^t}{\partial h_k^t}$$
$$= \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}} \tanh'(Wh^t + Ux^{t+1} + b)_{k'} W_{k'k} + \sum_j \frac{\partial C}{\partial a_j^t} V_{jk}$$

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

$$\tanh'(Wh^t + Ux^{t+1} + b)_{k'} = 1 - \left(h_{k'}^{t+1}\right)^2$$

Einsetzen liefert:

$$\frac{\partial C}{\partial h_k^t} = \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}} \left(1 - \left(h_{k'}^{t+1}\right)^2\right) W_{k'k} + \sum_j \frac{\partial C}{\partial a_j^t} V_{jk}$$

(Die erste Summe entfällt für $t = T$.)

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

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

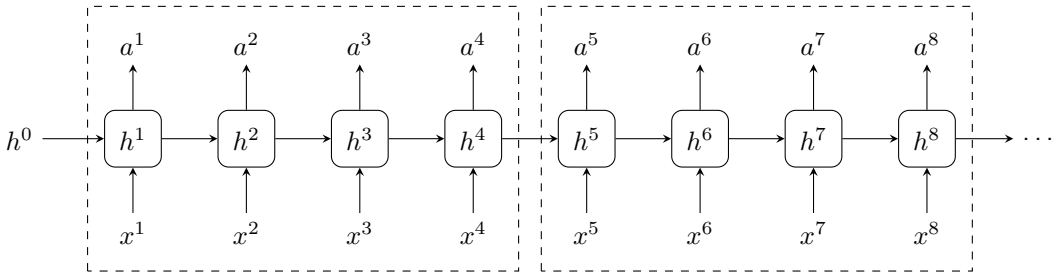
$$\frac{\partial C}{\partial V_{jk}} = \sum_{t=1}^{T} \frac{\partial C}{\partial a_j^t} \frac{\partial a_j^t}{\partial V_{jk}^t} = \sum_{t=1}^{T} \frac{\partial C}{\partial a_j^t} h_k^t$$

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

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

Bemerkung: BPTT in der Praxis: Anwenden auf Sequenz-Blöcke ("truncated BPTT")

Beispiel für Blöcke der Länge 4:



"hidden state" wird von einem Block zum nächsten weiterverwendet

9.3 Deep Recurrent Neural Networks

9.4 Long Short-Term Memory Networks (LSTM Networks)

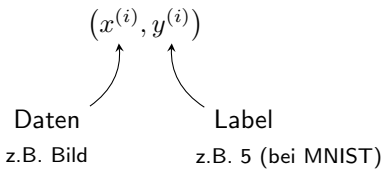
9.5 Anwendungsbeispiel: Computergenerierte Bildbeschriftung

Kapitel 10

Generative Modelle

10.1 Überblick und Motivation

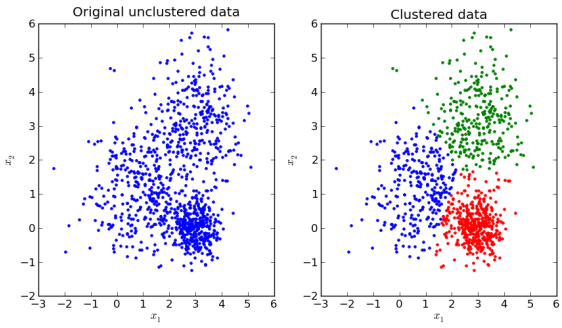
Bisher: Supervised Learning, erfordert "Labeling" der Daten:



Unsupervised Learning: finde Struktur von Daten ohne Labels; kann anschließend oft zur *Generierung* von neuen Daten (mit ähnlicher Struktur) verwendet werden

Beispiele:

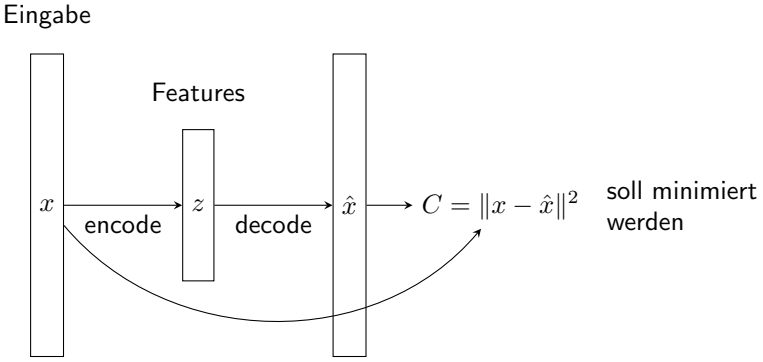
- k-means clustering:



Quelle: https://i.stack.imgur.com/cIDB3.png

  Algorithmus findet Cluster selbständig

- Autoregressive Modelle: z.B. RNNs zur Textgenerierung Buchstabe für Buchstabe oder Wort für Wort
- Autoencoder (automatisches "Komprimieren" von Daten, Reduktion auf essentielle "Features")

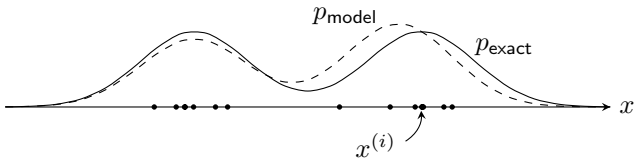


z hat viel kleinere Dimension als x

Beispiel Gesichtserkennung: z könnte charakteristische Eigenschaften wie Augenabstand, Größe des Mundes etc. speichern, während x das Gesicht als RGB-Bild enthält.

Funktionen "encode" und "decode" sind jeweils durch neuronales Netzwerk realisiert, durch Trainieren optimiert

- Density estimation: Finde bzw. approximiere unbekannte Dichteverteilung p_{exact} gegeben Samples $x^{(i)} \sim p_{\text{exact}}$, $i = 1, 2, \ldots$ ($x^{(i)} \in \mathbb{R}^n$, Normierung $\int_{\mathbb{R}^n} p_{\text{exact}}(x)\,\mathrm{d}x = 1$):



Algorithmus liefert Approximation p_{model}

Alternative Variante: Algorithmus soll neue Samples gemäß p_{model} generieren können (ohne p_{model} explizit aufzustellen) $\rightsquigarrow$ siehe vor allem "generative adversarial networks" (GANs)

Anwendung: für $x^{(i)}$: Bilder von Personen bzw. von Gesichtern: generiere (künstliche) Bilder von nicht-existierenden Personen! https://thispersondoesnotexist.com (generiert neues Gesicht "on the fly", anderes Gesicht bei jedem Aufruf der Seite)

10.2 Autoregressive Modelle

Idee: kann jede Wahrscheinlichkeitsdichte $p(x)$ (mit $x \in \mathbb{R}^n$) faktorisieren als Produkt bedingter Wahrscheinlichkeiten:

$$p(x) = \prod_{i=1}^{n} p(x_i | x_{i-1}, \ldots, x_1)$$

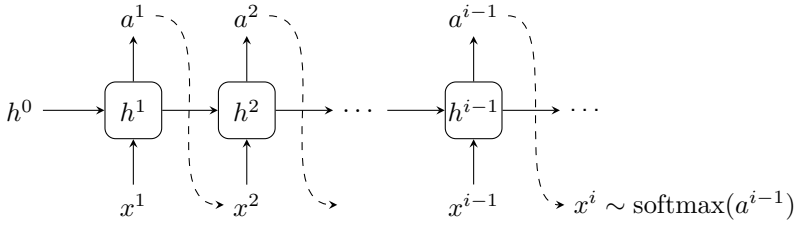
wobei $p(x_i|x_{i-1}, \ldots, x_1)$: bedingte Wahrscheinlichkeitsverteilung, d.h. Dichteverteilung der i-ten Komponente gegeben bisherige Komponenten $x_{i-1}, \ldots, x_1$ (insbesondere $\int p(x_i|x_{i-1}, \ldots, x_1)\,\mathrm{d}x_i = 1$)

Umgekehrt: $p(x)$ festgelegt durch bedingte Wahrscheinlichkeiten $\rightsquigarrow$ autoregressives Modell (Ansatz für $p(x_i|x_{i-1}, \ldots, x_1)$, z.B. mittels eines neuronalen Netzwerks)

Gebräuchlich sind zwei Varianten:

10.2.1 RNN als autoregressives Modell

(siehe Beispiel in Abschnitt 9.1)



$p(x_i|x_{i-1}, \ldots, x_1) = \mathrm{softmax}(a^{i-1})$

Beispiel: Buchstabe-für-Buchstabe Generierung eines Textes

10.2.2 Maskierung

Netzwerk realisiert Abbildung $\mathbb{R}^n \to \mathbb{R}^n$, Abhängigkeit lediglich von vorherigen Einträgen wird durch Maskierung (Nullsetzen) bestimmter Einträge der Gewichtsmatrizen realisiert

Beispiel: Standard Feedforward-Netzwerk, zwei "dense layers" (Eingabe $x \in \mathbb{R}^n$, Ausgabe $\tilde{x} \in \mathbb{R}^n$):

$$x \xrightarrow{\text{dense}} a^1 \xrightarrow{\text{dense}} \tilde{x}$$

$$a^1 = \sigma\big(W^1 \cdot x + b^1\big), \quad \text{d.h.} \quad a^1_j = \sigma\left(\sum_{k=1}^{n} W^1_{jk} x_k + b^1_j\right) \ \forall\ j$$
$$\tilde{x} = \sigma\big(W^2 \cdot a^1 + b^2\big)$$

Abstrakt: $\tilde{x} = f(x)$

Anforderung:

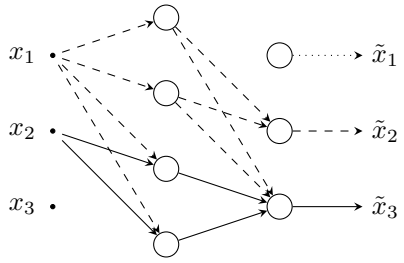
- $\tilde{x}_1$ muss unabhängig von Eingabe x sein
- $\tilde{x}_2$ darf nur von x_1 abhängen
- $\tilde{x}_3$ darf nur von x_1 und x_2 abhängen
- ...

Umsetzung: zu jedem versteckten Neuron j wähle $m_j \in \{1, \ldots, n-1\}$, definiere zwei Masken-Matrizen M^1 und M^2 mit Einträgen:

$$M^1_{jk} = \begin{cases} 1 & k \le m_j \\ 0 & \text{sonst} \end{cases}, \qquad M^2_{kj} = \begin{cases} 1 & k > m_j \\ 0 & \text{sonst} \end{cases}$$

und modifiziere Berechnung des Netzwerks (wobei $\odot$ komponentenweise Multiplikation):

$$a^1 = \sigma\big((W^1 \odot M^1) \cdot x + b^1\big)$$
$$\tilde{x} = \sigma\big((W^2 \odot M^2) \cdot a^1 + b^2\big)$$



Wegen M^1 hängt a_j^1 nur von $x_1, \ldots, x_{m_j}$ ab.

$\tilde{x}_{\tilde{k}}$ kann nur von x_k abhängen, falls $M^2_{\tilde{k}j} M^1_{jk} \neq 0$ für mindestens ein j:

$$x_k \xrightarrow{M^1_{jk}} a_j^1 \xrightarrow{M^2_{\tilde{k}j}} \tilde{x}_{\tilde{k}}$$

Für $\tilde{k} \leq k$: $M^2_{\tilde{k}j} M^1_{jk} = 1_{\{\tilde{k} > m_j\}} 1_{\{m_j \geq k\}} = 0$, also: gewünschte Unabhängigkeit erfüllt!

Bemerkung: erhalte Fixpunkt $f(x) = x$ durch sukzessives Einsetzen (wobei $\star$ beliebiger Eintrag):

$$x_1 = f(\star, \ldots, \star)_1\,, \quad x_2 = f(x_1, \star, \ldots, \star)_2\,, \quad x_3 = f(x_1, x_2, \star, \ldots, \star)_3\,, \quad \ldots$$

10.2.3 Anwendung: PixelRNN und PixelCNN

A. van den Oord, N. Kalchbrenner, K. Kavukcuoglu. *Pixel Recurrent Neural Networks* (2016) [OKK16]
(beide Varianten: RNN und Convolutional-Layers mit Maskierung)

Generiere Bild Pixel-für-Pixel $\rightsquigarrow$ muss Reihenfolge der Pixel festlegen (nicht kanonisch), z.B.

x_0	x_1	x_2	x_3
x_4	x_5	$\cdots$	

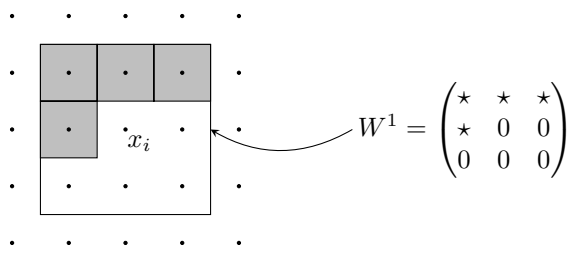
Farbbilder: Farbkanäle (RGB) als separate Einträge in x

PixelRNN: analog zur Textgenerierung Buchstabe-für-Buchstabe

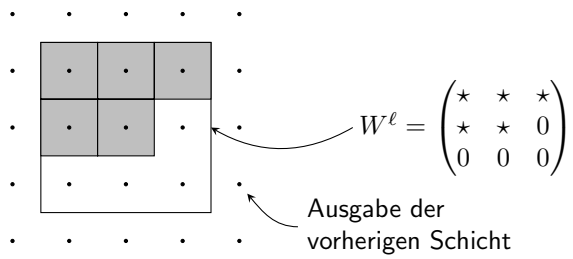
PixelCNN: Mehrere Convolutional Layers mit Maskierung, Ausgabe hat selbe Dimension wie Eingabe (insbesondere kein Pooling):

1. Erste Convolutional Layer ($\ell = 1$): 7×7 Filter

   Maskierung veranschaulicht für 3×3 Filter (7×7 analog):



2. Weitere Convolutional Layer ($\ell \geq 2$): 3×3 Filter



Somit: räumlicher Abhängigkeitsbereich eines Ausgabe-Pixels beschränkt sich auf "vorherige" Eingabe-Pixel

Einträge der Filter-Kerne W^ℓ, $\ell = 1, 2, \ldots L$ und Bias-Vektoren werden (mittels Backpropagation) optimiert, wobei gewünschtes Ausgabebild gleich dem Eingabebild gesetzt wird

Anwendung: Bildvervollständigung: Eingabe ist räumlich abgeschnittenes (oder verdecktes) Bild, Netzwerk generiert vollständiges Bild

10.3 Variational Autoencoders (VAEs)

10.3.1 Grundlagen der stochastischen Beschreibung

(siehe auch A10)

p Wahrscheinlichkeitsdichte auf $\mathbb{R}^n$, Normierung $\int_{\mathbb{R}^n} p(x)\,\mathrm{d}x = 1$; Erwartungswert einer Funktion f:

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)p(x)\,\mathrm{d}x$$

Entropie:

$$H(p) = -\int p(x)\log(p(x))\,\mathrm{d}x = \mathbb{E}_{x\sim p}[-\log(p(x))]$$

wobei $\log$ den natürlichen Logarithmus bezeichnet

Kullback-Leibler-Divergenz zweier Wahrscheinlichkeitsdichten p und q:

$$D_{\mathsf{KL}}(p \parallel q) = \int p(x)\log\left(\frac{p(x)}{q(x)}\right)\mathrm{d}x$$

Eigenschaften: $D_{\mathsf{KL}}(p \parallel q) \geq 0$, $D_{\mathsf{KL}}(p \parallel q) = 0 \Leftrightarrow p = q$

Cross-entropy (Kreuzentropie):

$$H(p, q) = -\int p(x)\log(q(x))\,\mathrm{d}x$$

Somit:

$$H(p, q) = H(p) + D_{\mathsf{KL}}(p \parallel q)$$

Maximum likelihood estimation: Ziel: approximiere (unbekannte) W-Dichte p_{exact} durch p_{model}, minimiere hierzu die KL-Divergenz zwischen p_{exact} und p_{model}:

$$\min_{p_{\mathsf{model}}} D_{\mathsf{KL}}(p_{\mathsf{exact}} \parallel p_{\mathsf{model}})$$

Äquivalent dazu, wegen $D_{\mathsf{KL}}(p_{\mathsf{exact}} \parallel p_{\mathsf{model}}) = H(p_{\mathsf{exact}}, p_{\mathsf{model}}) - H(p_{\mathsf{exact}})$:

$$\min_{p_{\mathsf{model}}} H(p_{\mathsf{exact}}, p_{\mathsf{model}})$$

Aber: kenne p_{exact} in der Praxis gar nicht, sondern nur Samples $x^{(1)}, x^{(2)}, \dots \sim p_{\mathsf{exact}} \rightsquigarrow$ ersetze p_{exact} durch empirische Verteilung (für N Samples):

$$p_{\mathsf{empir}}(x) = \frac{1}{N} \sum_{i=1}^{N} \delta\big(x - x^{(i)}\big)$$

Einsetzen in Cross-entropy: $H(p_{\mathsf{empir}}, p_{\mathsf{model}}) = \mathcal{L}$ mit der **negative log-likelihood**

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \log\Big(p_{\mathsf{model}}\big(x^{(i)}\big)\Big)$$

Minimieren von $\mathcal{L}$ bezüglich p_{model} wird als *maximum likelihood estimation* bezeichnet.

Gemeinsame und bedingte Wahrscheinlichkeitsdichte:

$$p(x, y) = p(x|y)p(y),$$

wobei $p(x|y)$ die *bedingte Wahrscheinlichkeitsdichte* von x gegeben y ist, mit Normierung $\int p(x|y)\,\mathrm{d}x = 1$; vergleiche mit Stochastik $\mathbb{P}(A \cap B) = \mathbb{P}(A|B)\,\mathbb{P}(B)$

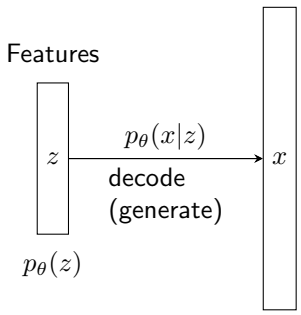
10.3.2 Auto-encoding variational Bayes

Diederik P. Kingma, Max Welling. *Auto-encoding variational Bayes* (2014) [KW14]

Gegeben: Samples $x^{(i)}$, $i = 1, 2, \dots$ (z.B. Bilder von Gesichtern), entsprechende Dichteverteilung soll möglichst gut durch p_{model} beschrieben werden

Fasse Parameter von p_{model} in Variable θ zusammen, Notation: p_θ; Ziel: geeignete Parameter θ

Modellansatz:



"Bausteine" des Modells:

- versteckte Zufallsvariable z mit "einfacher" Wahrscheinlichkeitsverteilung $p_\theta(z)$, z.B. mehrdimensionale Normalverteilung
  (Intuition bei Beschreibung von Gesichtern: übergeordnete Eigenschaften, z.B. erste Komponente z_1 entspricht Haarfarbe, zweite Komponente z_2 entspricht fröhlich / traurig, z_3 entspricht Ausrichtung, etwa von schräg links betrachtet, ...)
- bedingte Wahrscheinlichkeit $p_\theta(x|z)$, kann z.B. spezifiziert sein als $x \sim \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z))$ (Normalverteilung mit von θ und z abhängigen Parametern)

Somit: gemeinsame Wahrscheinlichkeitsdichte $p_\theta(x,z) = p_\theta(x|z)p_\theta(z)$ ebenfalls festgelegt, sowie Wahrscheinlichkeitsdichte für x:

$$p_\theta(x) \equiv \int p_\theta(x,z)\,\mathrm{d}z = \int p_\theta(x|z)p_\theta(z)\,\mathrm{d}z$$

Problem: Integration in der Praxis nicht analytisch bzw. effizient durchführbar, aber möchte $\log p_\theta(x^{(i)})$, $i = 1, 2, \ldots$ auswerten (für maximum likelihood estimation) und nach θ ableiten (für Gradientenverfahren)

Auch bedingte Wahrscheinlichkeitsdichte von z gegeben x nicht handhabbar:

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)} \qquad \text{(benötige } p_\theta(x)\text{)}$$

(vgl. mit Satz von Bayes $\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$)

Intuition: $p_\theta(z|x)$ modelliert Enkodierung, z.B. Haarfarbe gegeben Pixel-Bild eines Gesichts

Ausweg: Erweitere Modell um weitere (einfach auszuwertende) bedingte Wahrscheinlichkeitsdichte $q_\phi(z|x)$, so dass

$$q_\phi(z|x) \approx p_\theta(z|x)$$

Parameter θ und ϕ werden zusammen (gleichzeitig) optimiert. Trick: definiere *evidence lower bound* (ELBO)

$$\mathcal{L}^{\mathsf{ELBO}}_{\theta,\phi}(x) = \mathbb{E}_{z\sim q_\phi(z|x)}\big[\log p_\theta(x,z) - \log q_\phi(z|x)\big] = \int \left(\log p_\theta(x,z) - \log q_\phi(z|x)\right) q_\phi(z|x)\,\mathrm{d}z$$

Umformen (Hausaufgabe) $\rightsquigarrow$

$$-\mathcal{L}^{\mathsf{ELBO}}_{\theta,\phi}(x) + \log p_\theta(x) = D_{\mathsf{KL}}\left(q_\phi(z|x) \parallel p_\theta(z|x)\right) \geq 0,$$

also:

$$-\log p_\theta(x) \leq -\mathcal{L}^{\mathsf{ELBO}}_{\theta,\phi}(x)$$

Interpretation: $\mathcal{L}^{\mathsf{ELBO}}_{\theta,\phi}(x)$ ist variationelle Schranke an $\log p_\theta(x)$, Schranke wird umso schärfer, je näher $q_\phi(z|x)$ an $p_\theta(z|x)$ liegt (da dann der KL-Term $\to 0$)

Erinnerung maximum likelihood estimation: minimiere $-\frac{1}{N}\sum_{i=1}^N \log(p_\theta(x^{(i)}))$ bezüglich θ, hier ersetzt durch:

$$\min_{\theta,\phi} \frac{(-1)}{N} \sum_{i=1}^N \mathcal{L}^{\mathsf{ELBO}}_{\theta,\phi}\big(x^{(i)}\big)$$

Vorgehen zur Minimierung: starte von alternativer Darstellung (durch Umformen):

$$\mathcal{L}^{\mathsf{ELBO}}_{\theta,\phi}(x) = \underbrace{-D_{\mathsf{KL}}\left(q_\phi(z|x) \parallel p_\theta(z)\right)}_{\text{analytisch (für Normalverteilungen)}} + \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\big[\log p_\theta(x|z)\big]}_{\text{Reparameterization Trick}}$$

- $D_{\mathsf{KL}}\left(q_\phi(z|x) \parallel p_\theta(z)\right)$ für Spezialfall $q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \Sigma_\phi(x))$ und $p_\theta(z) = \mathcal{N}(z; 0, I)$ (mehrdimensionale Normalverteilungen): explizite Rechnung führt auf

  $$D_{\mathsf{KL}}\left(q_\phi(z|x) \parallel p_\theta(z)\right) = -\frac{1}{2}\left(k + \log|\Sigma_\phi(x)| - \|\mu_\phi(x)\|^2 - \mathrm{Tr}\left[\Sigma_\phi(x)\right]\right)$$

  wobei k die Dimension von z ist, $|\Sigma_\phi(x)|$ die Determinante der Kovarianzmatrix bezeichnet und $\|\cdot\|$ die euklidische Norm. Determinante und Spur lassen sich durch die (nicht-negativen) Eigenwerte von $\Sigma_\phi(x)$ ausdrücken, bezeichnet mit $\sigma_{\phi,j}(x)^2$, $j = 1, \ldots, k$:

  $$D_{\mathsf{KL}}\left(q_\phi(z|x) \parallel p_\theta(z)\right) = -\frac{1}{2}\sum_{j=1}^k \left(1 + \log\sigma_{\phi,j}(x)^2 - \mu_{\phi,j}(x)^2 - \sigma_{\phi,j}(x)^2\right)$$

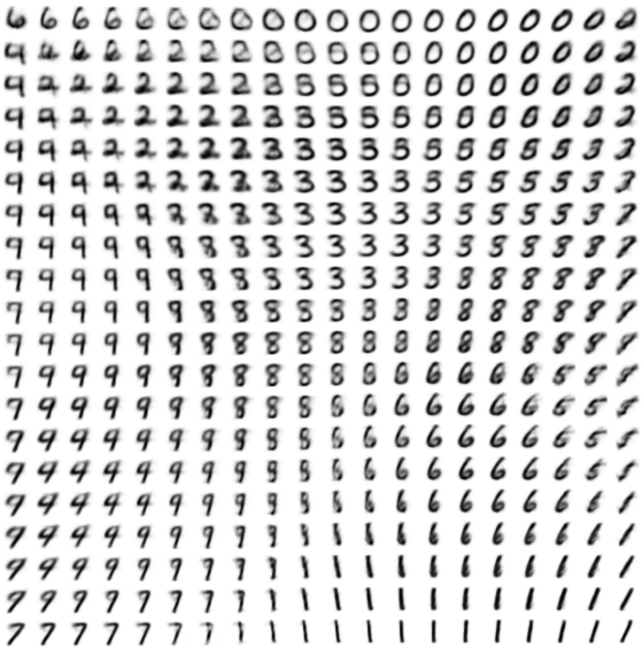
- Reparameterization Trick (siehe T11) angewendet auf $z \sim q_\phi(z|x)$, zur Bestimmung des Gradienten bezüglich ϕ: reparametrisiere die Wahrscheinlichkeitsdichte $q_\phi(z|x)$ als

$$z = g_\phi(\epsilon, x), \quad \epsilon \sim \tilde{p}(\epsilon),$$

  wobei die Hilfsvariable ϵ einer separaten (von ϕ und x unabhängigen) Wahrscheinlichkeitsdichte $\tilde{p}(\epsilon)$ folgt

Konkrete Wahl der W-Dichten in [KW14]: $p_\theta(z) = \mathcal{N}(z; 0, I)$ und $p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \mathrm{diag}(\sigma_\theta(z)^2))$, wobei die Vektoren $\mu_\theta(z)$ und $\sigma_\theta(z)^2$ die Ausgabe eines (relativ einfachen) neuronalen Netzwerks mit Parametern θ (Gewichtsmatrizen und Bias-Vektoren) sind; analog $q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \mathrm{diag}(\sigma_\phi(x)^2))$

Anwendung auf MNIST für $z \in \mathbb{R}^2$ (also zweidimensionale "Features"):



Quelle: [KW14]

x- und y-Achse entsprechen den Komponenten z_1 und z_2

Anwendung auf Gesichter, ebenfalls für $z \in \mathbb{R}^2$:

Quelle: [KW14]

x-Achse: Betrachtungswinkel, y-Achse: wie fröhlich bzw. traurig

10.4 Generative Adversarial Networks (GANs)

Idee zurückgehend auf Jürgen Schmidhuber: "artificial curiosity" (1990)

10.4.1 Prinzip und mathematische Beschreibung

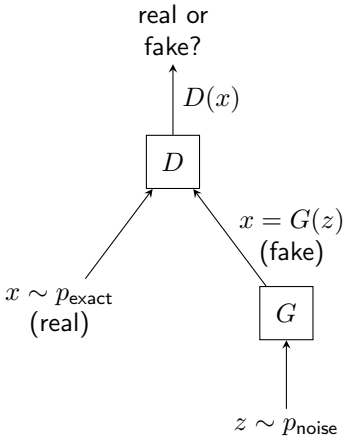
Ian J. Goodfellow et al. *Generative adversarial networks* (2014) [Goo+14]

Wie bisher: gegeben Samples $x^{(i)} \sim p_{\text{exact}}$, $i = 0, 1, \ldots$, möchte weitere Samples generieren

Jetzt: kein explizites Aufstellen von p_{model}

Ansatz bei GANs: zwei Gegenspieler:
"*Generator*" G: generiere möglichst glaubwürdige Samples
"*Diskriminator*" D: unterscheide echte von generierten Samples

G möchte D täuschen, D möchte G entlarven



$D(x) \in [0, 1]$: Wahrscheinlichkeit, dass x "echt" (real) ist

G erzeugt Realisierung aus Zufallsvektor z; Intuition: z enthält z.B. bei Gesichtsgenerierung abstrakte Charakterisierungen wie Haarfarbe, Brille vorhanden?, ..., aber a priori Interpretation von z nicht festgelegt

Mathematisch: Optimierung von G und D mittels Min-Max-Formulierung

$$\min_G \max_D V(D, G)$$

mit

$$V(D,G) = \mathbb{E}_{x\sim p_{\text{exact}}}[\log(\underbrace{D(x)}_{\text{"real"}})] + \mathbb{E}_{z\sim p_{\text{noise}}}[\log(1 - \underbrace{D(G(z))}_{\text{"fake"}})]$$

Somit:
Diskriminator: möglichst $D(x) = 1$ für x "echt", aber $D(G(z)) = 0$
Generator: möglichst $D(G(z)) = 1$ (täusche Diskriminator)

p_g bezeichne die von der Transformation $x = G(z)$ induzierte Wahrscheinlichkeitsdichte, d.h.

$$x \sim p_g \quad \text{für} \quad x = G(z), \quad z \sim p_{\text{noise}}$$

Spezialfall $G : \mathbb{R}^n \to \mathbb{R}^n$ Diffeomorphismus: für $A \subset \mathbb{R}^n$ (messbar) gilt

$$\mathbb{P}(x \in A) = \mathbb{P}\left(z \in G^{-1}(A)\right) = \int_{G^{-1}(A)} p_{\text{noise}}(z)\,\mathrm{d}z, \quad \text{andererseits}$$

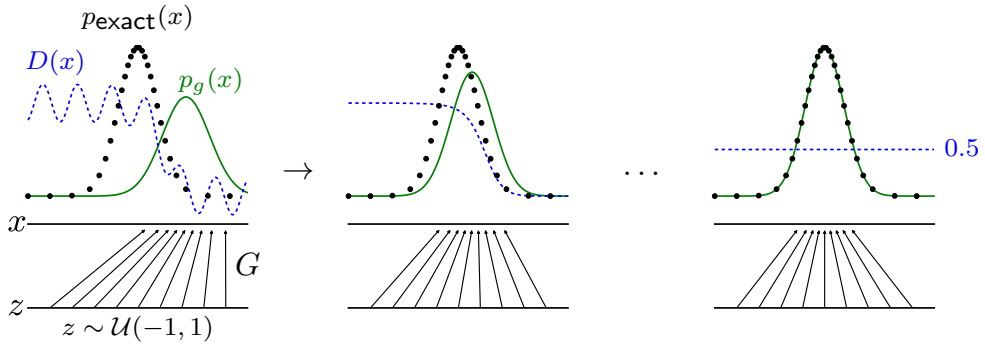
$$\mathbb{P}(x \in A) = \int_A p_g(x)\,\mathrm{d}x = \int_{G^{-1}(A)} p_g(G(z)) \cdot \underbrace{|DG(z)|}_{\text{Jacobi-Det.}}\,\mathrm{d}z,$$

somit, da A beliebig:

$$p_g(G(z)) \cdot |DG(z)| = p_{\text{noise}}(z) \quad \forall z \in \mathbb{R}^n$$

(Transformation von Wahrscheinlichkeitsdichten)

Schema in 1D (wobei $\mathcal{U}(a, b)$: Gleichverteilung im Intervall $[a, b]$):



Quelle: [Goo+14]

Proposition 1 ([Goo+14]). *Für vorgegebenen Generator G lautet der optimale Diskriminator $D^*_G = \operatorname{argmax}_D V(D, G)$:*

$$D^*_G(x) = \frac{p_{\text{exact}}(x)}{p_{\text{exact}}(x) + p_g(x)}.$$

Beweis.

$$\begin{aligned} V(D, G) &= \mathbb{E}_{x \sim p_{\text{exact}}}[\log(D(x))] + \mathbb{E}_{x \sim p_g}[\log(1 - D(x))] \\ &= \int p_{\text{exact}}(x)\ \log(D(x)) + p_g(x)\ \log(1 - D(x))\, \mathrm{d}x. \end{aligned}$$

Punktweise Maximierung des Integranden bezüglich $D(x)$: Abbildung $d \mapsto a \log(d) + b \log(1 - d)$ nimmt Maximum im Intervall $(0, 1)$ an der Stelle $\frac{a}{a+b}$ an (kurzes Nachrechnen). □

Bemerkungen:

- In der Praxis p_g nicht explizit bekannt (aber hilfreich für analytische Rechnungen)
- Alternative Interpretation der Diskriminator-Optimierung: D minimiert Cross-entropy Kostenfunktion (siehe A10)

  $$C^{\text{cross}}_{x,y} = -\Big[y \log\big(a^L\big) + \big(1 - y\big) \log\big(1 - a^L\big)\Big]$$

  (mit $a^L = D(x)$) für Trainingspaare $(x^{(i)}, y^{(i)})$, wobei die $x^{(i)}$ alternierend von p_{exact} und p_g gesampelt werden, und

  $$y^{(i)} = \begin{cases} 1, & \text{für } x^{(i)} \sim p_{\text{exact}} \text{ (real)} \\ 0, & \text{für } x^{(i)} \sim p_g \text{ (fake)} \end{cases}$$

Einsetzen des optimalen Diskriminators in Min-Max-Formel $\rightsquigarrow$

$$\min_G \max_D V(D, G) = \min_G \underbrace{V(D^*_G, G)}_{=: C(G)}$$

$$\begin{aligned} C(G) &= \int p_{\text{exact}}(x)\ \log\left[\frac{p_{\text{exact}}(x)}{p_{\text{exact}}(x) + p_g(x)}\right] + p_g(x)\ \log\left[\frac{p_g(x)}{p_{\text{exact}}(x) + p_g(x)}\right] \mathrm{d}x \\ &= \int p_{\text{exact}}(x)\ \log\left[\frac{p_{\text{exact}}(x)}{\frac{1}{2}\left(p_{\text{exact}}(x) + p_g(x)\right)}\right] + p_g(x)\ \log\left[\frac{p_g(x)}{\frac{1}{2}\left(p_{\text{exact}}(x) + p_g(x)\right)}\right] \mathrm{d}x - \log 4 \\ &= \underbrace{D_{\text{KL}}\left(p_{\text{exact}} \parallel \tfrac{1}{2}(p_{\text{exact}} + p_g)\right) + D_{\text{KL}}\left(p_g \parallel \tfrac{1}{2}(p_{\text{exact}} + p_g)\right)}_{2 D_{\text{JSD}}(p_{\text{exact}} \| p_g)} - \log 4 \end{aligned}$$

mit der **Jensen-Shannon Divergenz**

$$D_{\mathsf{JSD}}(p \parallel q) = \tfrac{1}{2} D_{\mathsf{KL}}\left(p \parallel \tfrac{1}{2}(p+q)\right) + \tfrac{1}{2} D_{\mathsf{KL}}\left(q \parallel \tfrac{1}{2}(p+q)\right)$$

Eigenschaften von $D_{\mathsf{JSD}}(p \parallel q)$: symmetrisch in p, q, $D_{\mathsf{JSD}}(p \parallel q) \geq 0$, $D_{\mathsf{JSD}}(p \parallel q) = 0 \Leftrightarrow p = q$

Hieraus folgt direkt:

Satz 2 ([Goo+14]). *Das globale Minimum von $C(G)$ beträgt $-\log 4$ und wird genau für $p_g = p_{exact}$ angenommen.*

Erinnerung: p_g ist die von G induzierte Wahrscheinlichkeitsdichte, d.h. optimaler Generator erzeugt Realisierungen, die von "echten" Samples $x \sim p_{\mathsf{exact}}$ statistisch ununterscheidbar sind. In diesem Fall

$$D_G^*(x) = \frac{1}{2},$$

also 50% Wahrscheinlichkeit, dass Realisierung x "echt" ist.

Umsetzung von GANs: D und G sind künstliche neuronale Netze mit Parametern θ_d bzw. θ_g, werden simultan mittels Backpropagation anhand des Min-Max-Problems optimiert $\rightsquigarrow$ ein Schritt des Gradientenverfahrens mit Lernrate η (einfachste Version, in der Praxis verbesserte Gradientenverfahren):

$$\theta_d \to \theta_d + \eta \nabla_{\theta_d} V(D_{\theta_d}, G_{\theta_g}),$$
$$\theta_g \to \theta_g - \eta \nabla_{\theta_g} V(D_{\theta_d}, G_{\theta_g})$$

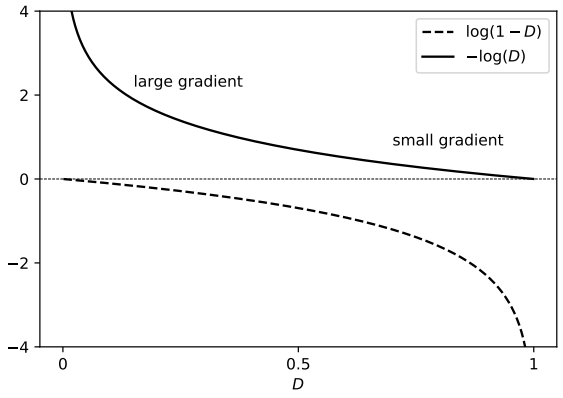
Beobachtung:

$$\nabla_{\theta_g} V(D_{\theta_d}, G_{\theta_g}) = \underbrace{\nabla_{\theta_g} \mathbb{E}_{x \sim p_{\mathsf{exact}}}[\log(D_{\theta_d}(x))]}_{=0} + \nabla_{\theta_g} \mathbb{E}_{z \sim p_{\mathsf{noise}}}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))],$$

in der Praxis: zweiter Term beim Gradientenschritt für den *Generator* ersetzt durch

$$\nabla_{\theta_g} \mathbb{E}_{z \sim p_{\mathsf{noise}}}[-\log(D_{\theta_d}(G_{\theta_g}(z)))]$$

Motivation: möglichst (betragsmäßig) großer Gradient im Fall $D(G(z)) \approx 0$, d.h. falls G nicht gut funktioniert:



10.4.2 Anwendung zur künstlichen Bildgenerierung: ProGAN und StyleGAN

Referenz ProGAN: T. Karras, T. Aila, S. Laine, J. Lehtinen. *Progressive growing of GANs for improved quality, stability, and variation.* (2018) [Kar+18]

Referenz StyleGAN: T. Karras, S. Laine, T. Aila. *A style-based generator architecture for generative adversarial networks.* (2019) [KLA19]

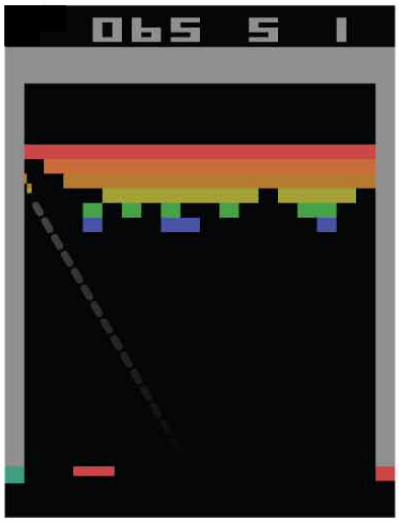
Kapitel 11

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" besteht aus anderen Verkehrsteilnehmern

Toy-Model: Atari 2600 Computerspiele (1977 – 1982): Pac-Man, Pong, Breakout, ...: verwende einzelne Bildschirm-Frames (Bildschirmausgabe) und Punktestand eines Computerspiels als Eingabe, Programm soll optimale Steuerung selbst herausfinden



"Breakout"
Quelle: [Mni+15]

V. Mnih, K. Kavukcuoglu, D. Silver, ..., D. Hassabis (Google DeepMind). *Human-level control through deep reinforcement learning.* Nature (2015) [Mni+15]

Selbe Programm-Architektur auf 49 verschiedene Atari-Spiele angewendet

$\rightsquigarrow$ (ca. 1 Jahr später): AlphaGo Programm:
D. Silver, A. Huang, ..., D. Hassabis (Google DeepMind). *Mastering the game of Go with deep neural networks and tree search.* Nature (2016) [Sil+16]

Weiter verbesserte Version:
AlphaGo Zero: D. Silver, ..., D. Hassabis (Google DeepMind). *Mastering the game of Go without human knowledge.* Nature (2017) [Sil+17]

Im Gegensatz zu AlphaGo verwendet AlphaGo Zero keine Datenbank mit menschlichen Spielen, sondern "trainiert", indem es gegen sich selbst spielt. Lediglich die Spielregeln sind vorgegeben.

11.1 Markov Decision Processes (MDPs)

Formale Beschreibung: Ein Agent und Umgebung befinden sich zum Zeitpunkt $t = 0, 1, 2, \ldots$ im **Zustand** $s_t \in \mathcal{S}$ (wobei $\mathcal{S}$ ein endlicher Zustandsraum); der Agent wählt eine **Aktion** $a_t \in \mathcal{A}$ aus $\rightsquigarrow$ das System geht in den nächsten Zustand $s_{t+1} \in \mathcal{S}$ über, mit Übergangswahrscheinlichkeit

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

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

Zu jedem Zeitpunkt erhält der Agent einen **reward** (Belohnung):

$$r_t = R(s_t)$$

(kann positiv, negativ oder 0 sein)

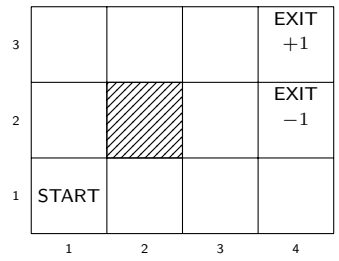
Ziel ist die Maximierung der kumulativen, (möglicherweise) zeitlich abgeschwächten Belohnung (cumulative discounted reward), der sogenannten **utility** (Nutzen):

$$U(s_0, s_1, \ldots) = \sum_{t=0}^{\infty} \gamma^t R(s_t), \quad 0 < \gamma \leq 1$$

wobei γ^t ein Gewichtungsfaktor ist; für $\gamma < 1$: Belohnung zu späteren Zeitpunkten (weit in der Zukunft) weniger wichtig

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

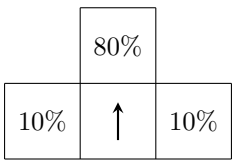
Modellbeispiel: 4×3-Spielfeld mit einer Spielfigur (schraffiertes Feld $(2, 2)$ darf nicht betreten werden):



s_t: aktuelle Position der Spielfigur

Mögliche Aktionen: $\uparrow$, $\downarrow$, $\leftarrow$, $\rightarrow$: "versuche, ein Feld nach oben/unten/links/rechts zu gehen"

Übergangswahrscheinlichkeit:



Formal:

$$\begin{aligned}
&\mathbb{P}\left(s_{t+1} = s_t + \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right) | s_t, a_t = \uparrow\right) = 0.8,\\
&\mathbb{P}\left(s_{t+1} = s_t + \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right) | s_t, a_t = \uparrow\right) = 0.1,\\
&\mathbb{P}\left(s_{t+1} = s_t + \left(\begin{smallmatrix}-1\\0\end{smallmatrix}\right) | s_t, a_t = \uparrow\right) = 0.1
\end{aligned}$$

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

"Die Wahrscheinlichkeit, dass sich der Agent tatsächlich in die intendierte Richtung bewegt, ist 80%; mit jeweils 10% Wahrscheinlichkeit bewegt er sich in eine der dazu senkrechten Richtungen."

(Falls der Agent auf ein unzulässiges Feld ziehen würde, verbleibt er auf dem aktuellen Feld.)

Spiel endet nach Betreten eines der Exit-Felder.

Reward: $+1$ für finales Feld $(4,3)$, -1 für finales Feld $(4,2)$, -0.04 für jedes andere Feld (Agent soll motiviert sein, Ausgang möglichst schnell zu erreichen)

Wie sieht Lösung des Problems aus?

Verhalten des Agenten durch "**policy**" (Verhaltensregel) $\pi : \mathcal{S} \to \mathcal{A}$ spezifiziert:

$$a_t = \pi(s_t)$$

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

"Qualität" von π quantifiziert durch *erwarteten Nutzen* (für Startzustand s_0):

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

wobei $\mathbb{E}$ den stochastischen Erwartungswert bezeichnet und $s_1, s_2, \ldots$ als Zufallsvariablen aufgefasst werden

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

Der erwartete Nutzen $U^\pi(s)$ wird auch als **value function** (Bewertungsfunktion) des Zustands s bezeichnet.

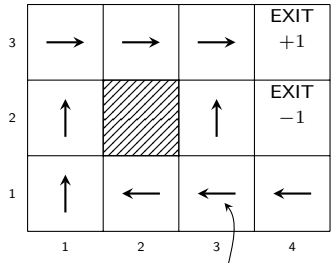
Optimale policy:

$$\pi^* = \operatorname*{argmax}_\pi U^\pi(s_0)$$

π^* hängt nicht von s_0 ab! Intuition: Falls Trajektorie $s_0, s_1, \ldots$ zum Zeitpunkt t einen vorgegebenen Zustand $\tilde{s}_0$ erreicht, d.h. $s_t = \tilde{s}_0$, dann muss die optimale Verhaltensregel ab t die gleiche sein wie bei Startzustand $\tilde{s}_0$, denn verbleibender erwarteter Nutzen hat selbstähnliche Form:

$$\mathbb{E}\left[\sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'}) \Big| \pi, s_t = \tilde{s}_0\right] = \gamma^t \, \mathbb{E}\left[\sum_{\Delta t=0}^{\infty} \gamma^{\Delta t} R(s_{t+\Delta t}) \Big| \pi, s_t = \tilde{s}_0\right] = \gamma^t \, U^\pi(\tilde{s}_0).$$

Beispiel (für $\gamma = 1$):



nehme Umweg, anstatt zu riskieren, in $(4, 2)$ zu landen

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

Beispiel (selbe Parameter):

3	0.812	0.868	0.918	EXIT +1
2	0.762		0.660	EXIT −1
1	0.705	0.655	0.611	0.388
	1	2	3	4

Man kann umgekehrt auch π^* aus U erhalten:

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s, a) U(s')$$

"Wähle Aktion, die den erwarteten Nutzen maximiert."

11.1.1 Bellman-Gleichung für die Bewertungsfunktion

Intuition: erwarteter Nutzen eines Zustands s ist gleich dem aktuellen reward und erwartetem Nutzen des nächsten Zustands, bei optimalem Verhalten des Agenten

Daraus erhält man die *Bellman-Gleichung* für die Bewertungsfunktion:

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

(eindeutig lösbar für $\gamma < 1$, siehe Tutoraufgabe Blatt 12)

11.1.2 Value-Iteration-Algorithmus

Naheliegende Idee zur (numerischen) Lösung der Bellman-Gleichung: Iteration:

$$U_{i+1}(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s, a) U_i(s') \quad \forall s \in \mathcal{S}, \quad i = 0, 1, \ldots$$

Abstrakt: Bellman-Schritt:

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

B ist Kontraktion (für $\gamma < 1$) bezüglich der Maximums-Norm:

$$\|B(U_i) - B(U_i')\| \leq \gamma \|U_i - U_i'\| \quad \text{mit} \quad \|U_i\| = \max_{s \in \mathcal{S}} |U_i(s)|,$$

Lösung $U = U^{\pi^*}$ ist eindeutiger Fixpunkt: $B(U) = U$

Somit:

$$\|U_i - U\| \leq \gamma \|U_{i-1} - U\| \leq \gamma^2 \|U_{i-2} - U\| \leq \cdots \leq \gamma^i \|U_0 - U\|,$$

exponentielle Konvergenz

Abschätzung von $\|U_i - U\|$, ohne U zu kennen:

$$\begin{aligned}
\|U_i - U\| &= \|U_i - U_{i+1} + U_{i+1} - U\| \\
&\leq \|U_{i+1} - U_i\| + \|U_{i+1} - U\| \\
&\leq \gamma \|U_i - U_{i-1}\| + \|U_{i+1} - U\| \\
&\leq \gamma \|U_i - U_{i-1}\| + \gamma \|U_{i+1} - U_i\| + \|U_{i+2} - U\| \quad \text{(Anwenden auf } i+1\text{)} \\
&\leq \gamma \|U_i - U_{i-1}\| + \gamma^2 \|U_i - U_{i-1}\| + \|U_{i+2} - U\| \leq \dots \\
&\leq \|U_i - U_{i-1}\| \cdot \sum_{j=1}^{\infty} \gamma^j = \|U_i - U_{i-1}\| \, \frac{\gamma}{1-\gamma}
\end{aligned}$$

Somit: falls

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

dann ist $\|U_i - U\| < \epsilon$ (Abbruchkriterium für Iteration)

11.1.3 Policy-Iteration-Algorithmus

Idee: "beste" policy gemäß U_i, also

$$\pi_i(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a) U_i(s'),$$

stimmt möglicherweise schon mit optimaler policy π^* überein, obwohl U_i noch von U^{π^*} abweicht.

$\rightsquigarrow$ *Policy-Iteration-Algorithmus*:

Require: Anfängliche policy π_0 (z.B. zufällig ausgewürfelt)

for $i \leftarrow 0, 1, 2, \dots$ **do**

(a) policy-Auswertung: berechne

$$U_i(s) = U^{\pi_i}(s) \quad \forall s \in \mathcal{S}$$

(erwarteter Nutzen, falls sich der Agent gemäß π_i verhält)

(b) policy-Verbesserung: verwende U_i zur Bestimmung einer neuen policy π_{i+1}:

$$\pi_{i+1}(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a) U_i(s') \quad \forall s \in \mathcal{S}$$

Abbruch, falls optimale policy gefunden, d.h. falls $\pi_{i+1} = \pi_i$ bzw. (äquivalent dazu) $U_{i+1} = U_i$ (Begründung: in diesem Fall ist U_i auch eindeutiger Fixpunkt der Value-Iteration)

Vorteil der Policy-Iteration (im Vergleich zur Value-Iteration): (a) ist *lineares* Gleichungssystem für U_i:

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

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

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

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

setze $U_i = U_{i,k}$

Effizienzsteigerung durch Anwendung des Algorithmus auf Teilmenge aller Zustände (Idee: brauche keine genaue Bewertung für Zustände, die vermieden werden sollen)

11.1.4 Q-value function

Bisher: Bewertungsfunktion $U^\pi(s)$ für Zustand s

Idee bei **Q-value function**: bewerte Zustands-Aktions-Tupel (s, a):

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

"erwarteter Nutzen, wenn der Agent im Zustand s die Aktion a wählt und ab dann der policy π folgt"

Bemerkung: $Q^\pi(s, a)$ wird auch als **action-value function** bezeichnet.

Optimale Q-value function für optimale policy:

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

Vorteil: optimale Bewertungsfunktion U und optimale policy π^* folgen direkt aus Q^*:

$$U(s) = \max_a Q^*(s, a),$$
$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s, a)$$

Benötige hierfür keine Kenntnis der Übergangswahrscheinlichkeit $\mathbb{P}(s'|s, a)$, die Beschreibung ist "**model free**" (modellfrei).

11.1.5 Bellman-Gleichung für die Q-value function und Q-value-Iteration

Analog zur Bellman-Gleichung für U:

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

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

Entsprechende Q-value-Iteration:

$$Q_{i+1}(s, a) = R(s) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) \max_{a' \in \mathcal{A}} Q_i(s', a') \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, \quad i = 0, 1, \ldots$$

11.2 Klassischer Q-learning-Algorithmus

C. J. Watkins, P. Dayan (1992) [WD92]

Wie bisher: Iteration $Q_i \to Q_{i+1}$, aber anstatt (unbekannter) Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$ verwende tatsächlich beobachteten Übergang $s_t \to s_{t+1}$ in Simulationen der Umgebung (Episoden)

Update der Q-value function an der Stelle $(s,a) = (s_t, a_t)$ (sonst unverändert):

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

mit Lernrate η_i, wobei $0 < \eta_i < 1$

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

Watkins und Dayan beweisen Konvergenz $Q_i \xrightarrow{i\to\infty} Q^*$ unter milden Voraussetzungen, im Limes unendlich vieler Episoden, so dass jedes mögliche Tupel (s,a) unendlich oft auftritt.

11.3 Deep Reinforcement Learning: Motivation und Überblick

Problem der bisherigen iterativen "dynamic programming" (DP) Algorithmen (Value-Iteration, Policy-Iteration, Q-value-Iteration):

1. Erfordert Speichern der Werte von $U(s)$, $\pi(s)$ bzw. $Q(s,a)$ für alle möglichen Zustände $s \in \mathcal{S}$ (und Aktionen $a \in \mathcal{A}$), nur durchführbar für relativ kleinen Zustandsraum

   Bei Brettspielen:
   $$|\mathcal{S}| \approx b^d,$$
   wobei
   b: mittlere Anzahl erlaubter Züge pro Brettposition
   d: Spieltiefe, d.h. typische Anzahl an Zügen pro Spiel

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

   $\rightsquigarrow$ kann sämtliche mögliche Brettpositionen in der Praxis nicht aufzählen

2. Setzt Kenntnis der Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$ voraus, aber in der Praxis oft unbekannt bzw. schwer abzuschätzen (z.B. autonomes Fahren: Wahrscheinlichkeit für Verkehrssituation im nächsten Zeitschritt?)

   Stattdessen: konzeptionelle Vorstellung beim Reinforcement Learning: Agent soll sich in (anfangs) völlig unbekannter Umgebung zurechtfinden, d.h. ohne Kenntnis der Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$ und Reward-Funktion $R(s)$ ("**model free**")

Ansatz Deep Reinforcement Learning:

1. Approximiere U, π bzw. Q durch neuronale Netzwerke; "deep" bezieht sich auf Netzwerktiefe
2. Anstatt explizit mit Übergangswahrscheinlichkeiten zu rechnen, führe viele Simulationen des Systems durch (d.h. Realisierungen $(s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ des MDPs); Agent "lernt" während dieser Simulationen

   Kompromiss zwischen "**exploration**" und "**exploitation**": probiere bisher nicht oder selten verwendete Aktion aus (auch wenn nicht optimal) um Erfahrung zu sammeln, oder wähle Aktion die vermutlich den Nutzen maximiert?

11.4 Deep Q-learning

Referenzen:

V. Mnih, K. Kavukcuoglu, D. Silver, ..., D. Hassabis (Google DeepMind). *Human-level control through deep reinforcement learning*. Nature (2015) [Mni+15] (Atari 2600 Computerspiele)

V. Mnih, et al. (Google DeepMind). *Asynchronous methods for deep reinforcement learning* (2016) [Mni+16]

Ziel: Algorithmus zur (approximativen) Lösung der Bellmann-Gleichung

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

Ansatz:

$$Q^*(s,a) \approx Q(s,a;\theta),$$

wobei $Q(s,a;\theta)$ ein neuronales Netzwerk (Q-Netzwerk) mit Parametern θ (Gewichte und Bias-Vektoren); iterative Optimierung der Parameter: $\theta_{i-1} \to \theta_i$, $i = 1, 2, \ldots$

Idee: Approximiere

$$R(s) + \gamma \max_{a'} Q^*(s',a') \approx R(s) + \gamma \max_{a'} Q(s',a';\theta_i^-) =: Y_i^-(s')$$

mit θ_i^-: Referenzparameter aus früherer Iteration, und minimiere die quadratische Kostenfunktion

$$L_i(\theta_i) = \mathbb{E}_{s,a}\big[\big(\underbrace{\mathbb{E}_{s'}[Y_i^-(s')|s,a]}_{\text{rechte Seite von Bellmann}} - Q(s,a;\theta_i)\big)^2\big], \quad s' \sim \mathbb{P}(\cdot|s,a)$$

Erwartungswert $\mathbb{E}_{s,a}[\ldots]$: Zustand $s \in \mathcal{S}$ und Aktion $a \in \mathcal{A}$ als Zufallsvariablen aufgefasst; entsprechende Wahrscheinlichkeitsverteilung noch offen gelassen

Allgemein gilt für eine Zufallsvariable Y (mit Var: Varianz): $\mathbb{E}[Y]^2 = \mathbb{E}[Y^2] - \mathrm{Var}(Y)$, somit:

$$\big(\mathbb{E}_{s'}[Y_i^-(s')|s,a] - Q(s,a;\theta_i)\big)^2 = \mathbb{E}_{s'}\big[\,\big(Y_i^-(s') - Q(s,a;\theta_i)\big)^2\,|s,a\big] - \mathrm{Var}_{s'}(Y_i^-(s')|s,a)$$

und

$$L_i(\theta_i) = \mathbb{E}_{s,a,s'}\Big[\big(Y_i^-(s') - Q(s,a;\theta_i)\big)^2\Big] - \underbrace{\mathbb{E}_{s,a}\big[\mathrm{Var}_{s'}(Y_i^-(s')|s,a)\big]}_{\text{unabhängig von } \theta_i},$$

also: letzter Term kann für die Minimierung von L_i weggelassen werden

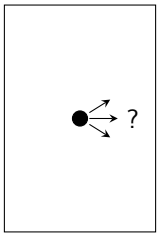
Gradient von L_i:

$$\begin{aligned}\nabla_{\theta_i} L_i(\theta_i) &= \mathbb{E}_{s,a,s'}\big[\big(Y_i^-(s') - Q(s,a;\theta_i)\big)\nabla_{\theta_i} Q(s,a;\theta_i)\big] \\ &= \mathbb{E}_{s,a,s'}\Big[\Big(R(s) + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\Big)\,\nabla_{\theta_i} Q(s,a;\theta_i)\Big]\end{aligned}$$

$\mathbb{E}_{s,a,s'}[\ldots]$ in der Praxis meist nicht explizit auswertbar (zu großer Zustandsraum); stattdessen: Approximation durch Sampling (Simulation von vielen Spieldurchläufen, siehe unten)

Bemerkung: klassischer Q-learning-Algorithmus ist Spezialfall: einzelnes Sample (s,a,s') und $\theta_i^- = \theta_{i-1}$

Bei Atari-Spielen: Zustandsraum $\mathcal{S}$?
Naive Idee: Zustand s_t ist aktueller Bildschirm-Frame (Einzelbild), aber: verletzt Markov-Eigenschaft: kann z.B. Flugrichtung eines Balls aus Einzelbild nicht erkennen, benötige vorausgehende Bildschirm-Frames



Setze stattdessen

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

wobei $x_{t'}$: Bildschirm-Frame zum Zeitpunkt t', $a_{t'}$: gewählte Aktion zum Zeitpunkt t'

Insbesondere Markov-Eigenschaft (per Konstruktion) erfüllt: vorherige Zustände $s_1, \ldots, s_{t-1}$ liefern keine Zusatzinformation bei bekanntem s_t

Vereinfachung für praktische Umsetzung mittels preprocessing-Funktion ϕ:

- Original: Bildschirm-Dimension 210×160, 128 Farben $\rightsquigarrow$ konvertiere zu Graustufen-Bildern (Helligkeit) und skaliere auf 84×84 Pixel
- Q-Netzwerk erhält nicht gesamte Spielhistorie (d.h. alle vorausgehenden Bildschirm-Frames) als Eingabe, sondern nur die letzten m Frames (konkret $m = 4$)

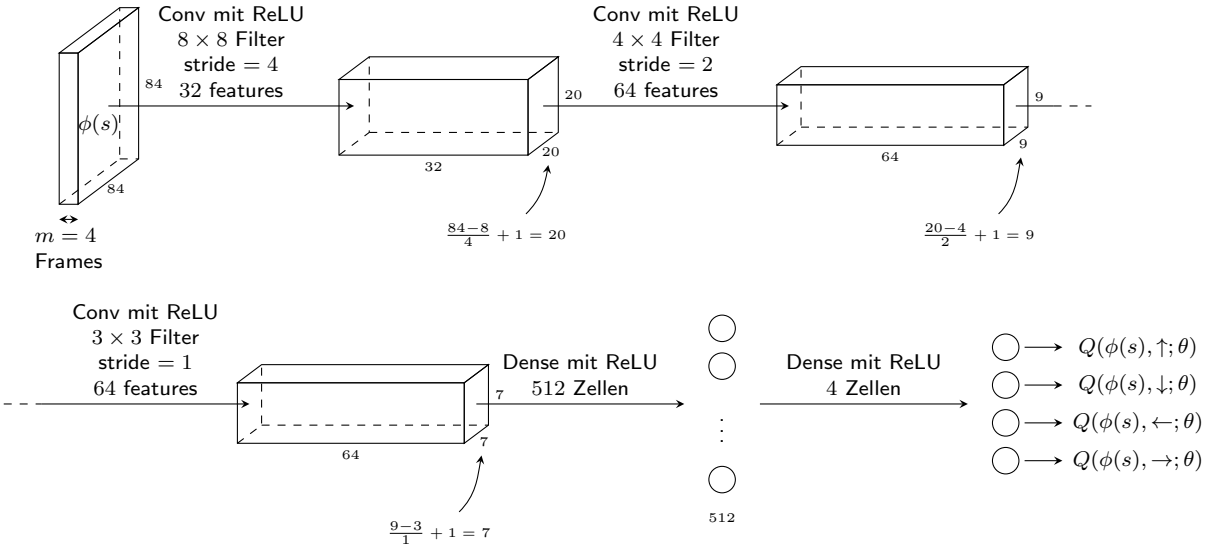
Insgesamt:

Spielhistorie aus allen bisherigen
210×160 Frames und entsprechende Aktionen

$\downarrow \phi$

letzte $m = 4$ Frames als
84×84-Graustufenbilder

Konkrete Q-Netzwerk Architektur bei Atari-Spielen: CNN mit Eingabe $\phi(s)$, bestehend aus drei convolutional und zwei dense layers:



(Anzahl möglicher Aktionen je nach Spiel unterschiedlich)

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

11.4.1 Training mit Experience Replay

(Bemerkung: verwendet im Nature 2015 paper, mittlerweile verbesserte Varianten entwickelt, siehe 11.4.2)

Ziel: optimiere Q-value function mittels stochastic gradient descent (bzw. Varianten hiervon) angewendet auf Netzwerk-Parameter θ; noch festzulegen: Approximation des Erwartungswerts $\mathbb{E}_{s,a,s'}[\dots]$, Wahl von θ_i^-

Allgemein unterscheidet man zwischen

- **on-policy training**: die zu optimierende policy wird auch verwendet, um Aktionen während der Spieldurchläufe (Simulationen) zu wählen, d.h. zur Generierung von Trainings-Samples (hier policy festgelegt durch $\operatorname{argmax}_a Q(\phi(s), a; \theta)$)
- **off-policy training**: die zu optimierende policy ist i.A. verschieden von der policy, die zum Wählen von Aktionen während der Spieldurchläufe (Simulationen) verwendet wird

Möglichkeiten zur "exploration" in beiden Fällen:

- on-policy: anstatt deterministischer Aktion liefert policy eine Wahrscheinlichkeitsverteilung über mögliche Aktionen; anfangs nahe an Gleichverteilung (exploration), im Laufe des Trainings Übergang zu deterministischer Funktion, d.h. eine einzelne Aktion hat Wahrscheinlichkeit 1
- off-policy: exploration mittels der policy zum Wählen von Aktionen während der Spieldurchläufe; oft gebräuchlich: ϵ-greedy strategy

Definition ϵ-**greedy strategy**:

- wähle gleichverteilt zufällige Aktion mit Wahrscheinlichkeit ϵ (exploration)
- wähle (gemäß aktuellem Wissen) beste Aktion mit Wahrscheinlichkeit $1 - \epsilon$, d.h. folge der zu optimierenden policy (exploitation)

Konkret bei Atari-Spielen: on-policy training eines künstlichen neuronalen Netzwerks problematisch bzw. instabil wegen unerwünschtem Feedback und künstlichem "Aufschaukeln" der berechneten Q-value function: aktuelles Verhaltensmuster (z.B. "Bewege Balken nach links" bei Breakout) beeinflusst weitere Samples ("Balken auf linker Seite")

Stattdessen: zweifache Umsetzung von off-policy training: ϵ-greedy strategy und "experience replay"

Definition **experience replay**: ein *replay memory* speichert Zustandsübergänge $(\phi(s_t), a_t, r_t, \phi(s_{t+1}))$ aus früheren Spieldurchläufen ab; diese werden zum Trainieren der Q-value function verwendet (anstatt aktuellem Übergang im aktuellen Spiel): "Lerne aus früherer Erfahrung"

Konkrete bei Atari-Spielen: 10^6 Einträge im replay memory

Weiter Innovation: Stabilisiere Zielfunktion Y_i^- durch "zeitliches Einfrieren":

Erinnerung:

$$Y_i^-(s') = R(s) + \gamma \max_{a'} Q(s', a'; \theta_i^-)$$

ist Referenzwert für Kostenfunktion (entspricht tatsächlichem Label beim Trainieren eines Netzwerks zur Bildklassifizierung), wobei i: einzelner Schritt im Gradientenverfahren, θ_i^-: Referenzparameter aus früherer Iteration

Problem bei "naiver" Wahl $\theta_i^- = \theta_{i-1}$: instabil, kann zu künstlichem Aufschaukeln der berechneten Q-value function führen

Abhilfe: anstatt Parameter des vorherigen Schritts des Gradientenverfahrens zu verwenden, lasse Parameter θ_i^- des Referenz-Netzwerks für $C = 10000$ Schritte unverändert

Algorithmus: Deep Q-learning with experience replay

1: Initialisiere (anfangs leeres) replay memory D
2: Initialisiere Q-Netzwerk mit zufälligen Gewichten θ, setze $\theta^- = \theta$
3: **for** episode $\leftarrow 0, 1, \ldots, M$ **do**
4: Initialisiere Zustand $s_1 = (x_1)$ (erster Frame), führe Preprocessing $\varphi_1 = \phi(s_1)$ durch
5: **for** $t \leftarrow 1, 2, \ldots, T$ **do**
6: ϵ-greedy stategy: wähle zufällige Aktion a_t mit Wahrscheinlichkeit ϵ, ansonsten

$$a_t = \operatorname*{argmax}_a Q(\varphi_t, a; \theta)$$

(beste Aktion gemäß aktuellem Q-Netzwerk)
7: Führe Aktion a_t aus $\rightsquigarrow$ nächster Bildschirm-Frame x_{t+1} und reward $r_t = R(s_t)$
8: Setze $s_{t+1} = (s_t, a_t, x_{t+1})$, führe Preprocessing $\varphi_{t+1} = \phi(s_{t+1})$ durch
9: Speichere Übergang $(\varphi_t, a_t, r_t, \varphi_{t+1})$ im replay memory D ab
10: Wähle (zufällig) mini-batch aus Übergängen $(\varphi_j, a_j, r_j, \varphi_{j+1})$ aus D ▷ (realisiert $\mathbb{E}_{s,a,s'}[\ldots]$ in der Formel für Kostenfunktion)
11: Setze

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

12: Gradientenschritt für Kostenfunktion $(y_j - Q(\varphi_j, a_j; \theta))^2$ angewendet auf θ
13: Nach jeweils C Schritten: setze $\theta^- = \theta$

Bemerkungen:

- Bei Atari-Spielen: anfangs $\epsilon = 1$ (Agent spielt völlig zufällig), schrittweiser Übergang innerhalb 10^6 Bildschirm-Frames zu $\epsilon = 0.1$, danach fixiert auf $\epsilon = 0.1$
- Algorithmus funktioniert, obwohl die Aktionen oft erst nach vielen Zeitschritten zu einer Belohnung führen ("Planung" implizit in Q-value function enthalten)

11.4.2 Asynchrones Training

Idee: ersetze replay memory durch mehrere parallel laufende Spiele (insbesondere fällt Speicherbedarf für replay memory weg)

Mehrere Agenten mit jeweils eigener Instanz der Umgebung (d.h. laufendem Spiel) spielen parallel, aber optimieren eine gemeinsame Q-value function $\rightsquigarrow$ "Aufschaukeln" wie bei on-policy training wird vermieden (Samples werden simultan durch die laufenden Spiele generiert, ausgeglichener)

Algorithmus: Asynchronous Q-learning Pseudocode für einzelnen Agenten;
verwendet globale Parameter θ und θ^- für Q-Netzwerk (anfangs $\theta = \theta^-$) und globalen Zeitschritt-Zähler T (anfangs $T = 0$)

1: Initialisierung:
- individueller Zeitschritt-Zähler $t \leftarrow 0$
- individuelle akkumulierte Gradienten $\mathrm{d}\theta \leftarrow 0$
- Startzustand s

repeat
  Wähle Aktion a gemäß ϵ-greedy strategy und $Q(s, a; \theta)$
  Führe Aktion a aus $\rightsquigarrow$ erhalte nächsten Zustand s' und reward r
  Setze

$$y = \begin{cases} r, & \text{falls } s' \text{ Endzustand} \\ r + \gamma \max_{a'} Q(s', a'; \vartheta^-), & \text{sonst} \end{cases}$$

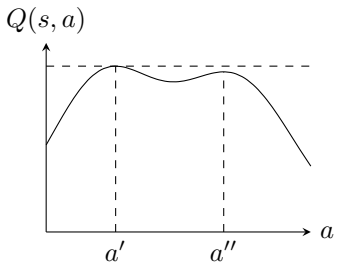
  Akkumuliere Gradienten: $\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \frac{\partial}{\partial \theta} \left(y - Q(s, a; \theta)\right)^2$
  $s \leftarrow s'$
  $T \leftarrow T + 1$ und $t \leftarrow t + 1$
  if $T \mod I_{\text{target}} = 0$ **then**
    $\theta^- \rightarrow \theta$ ▷ aktualisiere Zielnetzwerk
  if $t \mod I_{\text{AsyncUpdate}} = 0$ oder s Endzustand **then**
    Asynchrones Update von θ mittels Gradienten $\mathrm{d}\theta$
    $\mathrm{d}\theta \leftarrow 0$
until $T > T_{\max}$

11.5 Policy Gradient Methods

Idee: bestimme optimale policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ *direkt*, d.h. im Allgemeinen ohne value function $U(s)$ oder Q-value function $Q(s, a)$ zu berechnen

Motivation:

- optimale Q-value function $Q(s, a)$ kann sehr kompliziert bzw. schwer zu beschreiben sein, vor allem bei hochdimensionalem Zustandsraum
  Beispiel: Steuerung eines Roboters: Zustand bestehend aus Winkeln aller Gelenke, Sensordaten, ...; Aktionen: Steuerung von Aktuatoren, ...; oft naheliegende Verhaltensregel in bestimmter Situation relativ einfach (z.B. "schließe Hand"), benötige "Umweg" über $Q(s, a)$ nicht
- Bestimmung einer policy ausgehend von Bewertungsfunktion U benötigt Übergangswahrscheinlichkeit: $\pi(s) = \operatorname{argmax}_a \sum_{s'} \mathbb{P}(s'|s, a) U(s')$, aber $\mathbb{P}(s'|s, a)$ oft nicht explizit verfügbar
- policy function besser angepasst an kontinuierlichen Zustands- und Aktionsraum: kann z.B. $\pi : \mathcal{S} \rightarrow \mathcal{A}$ stückweise stetig oder stetig differenzierbar fordern
  Vergleich mit Q-value function $Q(s, a)$ und entsprechender policy $\pi(s) = \operatorname{argmax}_a Q(s, a)$: kann sensitiv bezüglich kleiner Änderungen von Q sein, falls $Q(s, a') \approx Q(s, a'')$ mit $a' = \operatorname{argmax}_a Q(s, a)$:



Ansatz für policy function: π_θ, wobei $\theta \in \mathbb{R}^m$ zu optimierende Parameter, z.B. Gewichte und Bias-Vektoren eines ANNs

Verwende (wie bisher) den erwarteten Nutzen zur Bewertung einer policy function, mit vorgegebenem

Anfangszustand $\hat{s}_0$:

$$J(\theta) := U^{\pi_\theta}(\hat{s}_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \Big| s_0 = \hat{s}_0, \pi_\theta\right]$$

Ziel: $\theta^* = \operatorname{argmax}_\theta J(\theta)$

11.5.1 Policy Gradient Theorem

Verallgemeinerung im Folgenden: policy function gibt Wahrscheinlichkeitsverteilung über Aktionen an: $\pi(a|s)$ (anstatt $a = \pi(s)$); deterministische policy function ist Spezialfall

Beobachtung: allgemein gilt

$$U^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s,a) \quad \forall s \in \mathcal{S}$$

und

$$Q^\pi(s,a) = R(s) + \gamma \sum_{s'} \mathbb{P}(s'|s,a) U^\pi(s') \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

Somit, zusammen mit Produktregel:

$$\begin{aligned} \nabla_\theta U^{\pi_\theta}(s) &= \sum_{a \in \mathcal{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta Q^{\pi_\theta}(s,a) \right) \\ &= \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \gamma \sum_{s',a} \pi_\theta(a|s) \mathbb{P}(s'|s,a) \nabla_\theta U^{\pi_\theta}(s') \\ &= \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \gamma \sum_{s'} \mathbb{P}_\theta(s'|s) \nabla_\theta U^{\pi_\theta}(s'), \end{aligned} \tag{11.1}$$

wobei

$$\mathbb{P}_\theta(s'|s) = \sum_a \pi_\theta(a|s) \mathbb{P}(s'|s,a)$$

die Übergangswahrscheinlichkeit $s \to s'$ unter Verwendung der policy π_θ bezeichnet

Gl. (11.1) liefert rekursive Beziehung für $\nabla_\theta U^{\pi_\theta}$; wiederholtes Einsetzen $\rightsquigarrow$

$$\nabla J(\theta) = \nabla_\theta U^{\pi_\theta}(\hat{s}_0) = \sum_{t=0}^{\infty} \sum_{s \in \mathcal{S}} \mathbb{P}_\theta(\hat{s}_0 \to s \text{ in } t \text{ Schritten}) \gamma^t \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$$

Betrachte einzelnen Term:

$$\begin{aligned} \gamma^t \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) &= \gamma^t \sum_a \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a|s) \\ &= \mathbb{E}_a\left[\gamma^t Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a|s) \big| \pi_\theta\right] \\ &= \mathbb{E}\left[\sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'}) \nabla_\theta \log \pi_\theta(a_t|s_t) \big| s_t = s, \pi_\theta\right], \end{aligned}$$

wobei sich der letzte Erwartungswert auf Trajektorien $(s_t, a_t, s_{t+1}, a_{t+1}, \dots)$ ab Zeitpunkt t bezieht

Insgesamt:

$$\nabla J(\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \left(\sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'}) \right) \nabla_\theta \log \pi_\theta(a_t|s_t) \big| \pi_\theta\right]$$

Interpretation der Optimierung von $J(\theta)$ mittels Gradientenverfahren: falls verbleibender "cumulative discounted reward" $\sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'})$ ab Zeitpunkt t positiv, dann dann erhöhe die Wahrscheinlichkeiten $\pi(a_t|s_t)$ der gewählten Aktionen

11.5.2 REINFORCE-Algorithmus

Direkt basierend auf Formel für $\nabla J(\theta)$: Gradientenverfahren mit Lernrate η

Zurückgehend auf: R. J. Williams. *Simple statistical gradient-following algorithms for connectionist reinforcement learning* (1992) [Wil92]

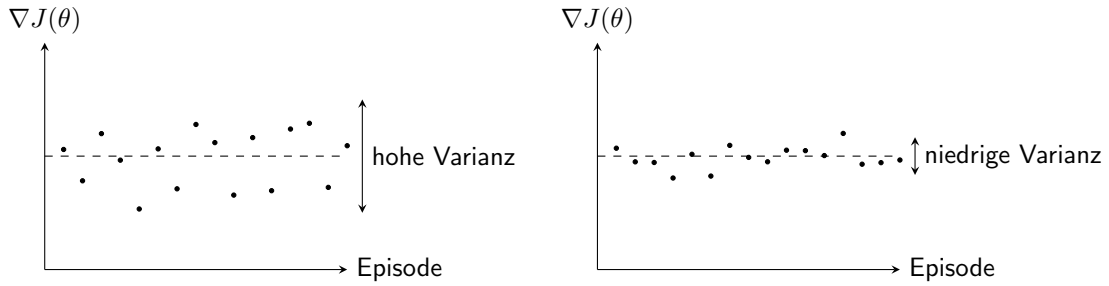
1: Initialisiere anfängliche Parameter θ
2: **for** episode $\leftarrow 0, 1, \ldots$ **do**
3: Führe Simulation mit policy π_θ durch, liefert Trajektorie $(s_0, a_0, r_0, \ldots, s_T, a_T, r_T)$
4: **for** $t \leftarrow 0, 1, \ldots, T$ **do**
5: $G_t \leftarrow \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
6: $\theta \leftarrow \theta + \eta \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$ ▷ Gradientenschritt

Bemerkungen:

- Varianten des Algorithmus ohne Faktor γ^t in Zeile 6
- Gradientenschritt erst "a posteriori", d.h. nach Beendigung eines Spiels möglich

11.5.3 Actor-Critic Methode

Motivation: reduziere Varianz beim Abschätzen von $\nabla J(\theta)$ mittels Sampling (d.h. Simulation vieler Spiele zur Abschätzung des Erwartungswerts $\nabla J(\theta) = \mathbb{E}[\ldots|\pi_\theta]$)



kleine Varianz vorteilhafter $\rightsquigarrow$ genauer, bzw. benötige weniger Samples

Idee: in Herleitung des Policy Gradient Theorems:

$$\nabla J(\theta) = \sum_{t=0}^{\infty} \sum_{s \in \mathcal{S}} \mathbb{P}_\theta(\hat{s}_0 \to s \text{ in } t \text{ Schritten}) \gamma^t \sum_a Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s) :$$

ersetze $\sum_a Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s)$ durch

$$\sum_a \left(Q^{\pi_\theta}(s,a) - b(s)\right) \nabla_\theta \pi_\theta(a|s)$$

mit "baseline"-Funktion $b : \mathcal{S} \to \mathbb{R}$: beliebige (von Parametern θ unabhängige) Funktion

Wert des Ausdrucks bleibt unverändert, da

$$\sum_a b(s) \nabla_\theta \pi_\theta(a|s) = b(s) \nabla_\theta \underbrace{\sum_a \pi_\theta(a|s)}_{=1} = 0$$

Führt auf verallgemeinerte Formel für $\nabla J(\theta)$:

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\left(\left(\sum_{t'=t}^{\infty}\gamma^{t'-t}R(s_{t'})\right)-b(s_t)\right)\nabla_\theta\log\pi_\theta(a_t|s_t)\Big|\pi_\theta\right]\\ &= \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\Big(Q^{\pi_\theta}(s_t,a_t)-b(s_t)\Big)\nabla_\theta\log\pi_\theta(a_t|s_t)\Big|\pi_\theta\right]\end{aligned}$$

Hierbei haben wir den verbleibenden "cumulative reward" mit $Q^{\pi_\theta}(s_t,a_t)$ identifiziert, da der Erwartungswert $\mathbb{E}[\ldots|\pi_\theta]$ gegeben policy π_θ gebildet wird

$\sum_{t'=t}^{\infty}\gamma^{t'-t}R(s_{t'})-b(s_t)$: Abweichung des beobachteten "cumulative reward" von baseline $\rightsquigarrow$ im Gradientenverfahren: erhöhe oder erniedrige Wahrscheinlichkeiten der gewählten Aktionen je nachdem, ob *Abweichung* positiv oder negativ

Naheliegende Wahl von $b(s)$: Bewertungsfunktion $U_\phi(s)$ mit (von θ unabhängigen) Parametern ϕ

Abweichnung bezeichnet mit **advantage**:

$$A_t = Q^{\pi_\theta}(s_t,a_t) - U_\phi(s_t)$$

Führt auf **actor-critic**-Methode:

- actor: policy function π_θ
- critic: Bewertung der gewählten Aktionen mittels A_t

Parameter θ und ϕ werden gleichzeitig optimiert

Beobachtung: für optimale policy $\pi_\theta = \pi^*$ und optimale Bewertungsfunktion $U_\phi = U^*$: $A_t = 0\ \forall t$, da $Q^*(s_t,\pi^*(s_t)) = U^*(s_t) \rightsquigarrow$ naheliegende Kostenfunktion zur Optimierung von ϕ:

$$\sum_{t=0}^{\infty}\frac{1}{2}(A_t)^2$$

Pseudocode (gegeben Lernraten η und $\tilde{\eta}$):

1: Initialisiere anfängliche Parameter θ und ϕ
2: **for** episode $\leftarrow 0,1,\ldots$ **do**
3: Führe Simulation mit policy π_θ durch, liefert Trajektorie $(s_0,a_0,r_0,\ldots,s_T,a_T,r_T)$ ▷ T: Spielende-Zeitpunkt
4: **for** $t \leftarrow 0,1,\ldots,T$ **do**
5: $G_t \leftarrow \sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}$
6: $A_t \leftarrow G_t - U_\phi(s_t)$
7: $\theta \leftarrow \theta + \eta\gamma^t A_t\nabla_\theta\log\pi_\theta(a_t|s_t)$ ▷ Gradientenschritt für θ
8: $\phi \leftarrow \phi + \tilde{\eta}A_t\nabla_\phi U_\phi(s_t)$ ▷ Gradientenschritt für ϕ

Bemerkung: in der Praxis asynchrone Varianten mit mehreren parallel laufenden Agenten (analog zu asynchronem Q-learning) gebräuchlich, siehe A3C ("asynchronous advantage actor-critic") [Mni+16]

11.6 Anwendungsbeispiel: AlphaGo Zero

Literatur

[Goo+14] I. J. Goodfellow u. a. „Generative adversarial networks“. In: *Advances in Neural Information Processing Systems 27 (NIPS 2014)* (2014). URL: https://arxiv.org/abs/1406.2661.

[Kar+18] T. Karras u. a. „Progressive growing of GANs for improved quality, stability, and variation“. In: *Sixth International Conference on Learning Representations (ICLR)* (2018). URL: https://arxiv.org/abs/1710.10196.

[KLA19] T. Karras, S. Laine und T. Aila. „A style-based generator architecture for generative adversarial networks“. In: *2019 Conference on Computer Vision and Pattern Recognition* (2019). URL: https://arxiv.org/abs/1812.04948.

[KW14] D. P. Kingma und M. Welling. „Auto-encoding variational Bayes“. In: *Proceedings of the 2nd International Conference on Learning Representations (ICLR)* (2014). URL: https://arxiv.org/abs/1312.6114.

[Mni+15] V. Mnih u. a. „Human-level control through deep reinforcement learning“. In: *Nature* 518 (2015), S. 529. DOI: 10.1038/nature14236.

[Mni+16] V. Mnih u. a. „Asynchronous methods for deep reinforcement learning“. In: *Proceedings of The 33rd International Conference on Machine Learning*. Bd. 48. Proceedings of Machine Learning Research. 2016, S. 1928–1937. URL: https://arxiv.org/abs/1602.01783.

[OKK16] A. van den Oord, N. Kalchbrenner und K. Kavukcuoglu. „Pixel recurrent neural networks“. In: *ICML'16 Proceedings of the 33rd International Conference on International Conference on Machine Learning*. Bd. 48. 2016, S. 1747–1756. URL: https://arxiv.org/abs/1601.06759.

[Sil+16] D. Silver u. a. „Mastering the game of Go with deep neural networks and tree search“. In: *Nature* 529 (2016), S. 484. DOI: 10.1038/nature16961.

[Sil+17] D. Silver u. a. „Mastering the game of Go without human knowledge“. In: *Nature* 550 (2017), S. 354. DOI: 10.1038/nature24270.

[WD92] C. J. Watkins und P. Dayan. „Technical note: Q-learning“. In: *Machine Learning* 8 (1992), S. 279–292. DOI: 10.1023/A:1022676722315.

[Wil92] R. J. Williams. „Simple statistical gradient-following algorithms for connectionist reinforcement learning“. In: *Machine Learning* 8 (1992), S. 229–256. DOI: 10.1007/BF00992696.

Index