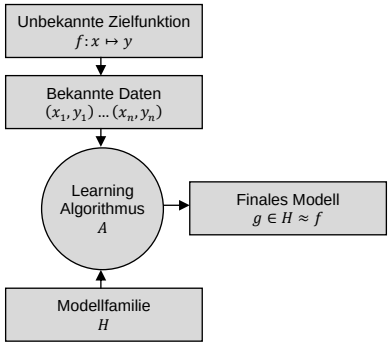
AI Applications

Grundbegriffe

Komponenten von Machine Learning

Beim Machine Learning geht es darum, eine unbekannte, «reale» Zielfunktion mithilfe von bekannten Daten über eine Modellfunktion zu approximieren.



⇨ Der Learning-Algorithmus beinhaltet Loss & Optimizer.
⇨ Die Modellfamilie wird auch als «Hypothesis Set» bezeichnet.

Artificial Neural Networks

Artificial Neurons

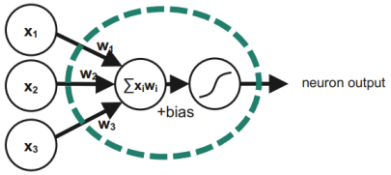
Ein Artificial Neuron ist eine «einfache» Recheneinheit bestehend aus:

- Einem Inputvektor $\vec{x} = [x_1, x_2 \dots]$.
- Einem Gewichtevektor $\vec{w} = [w_1, w_2 \dots]$.
- Ein Bias / Intercept b.
- Eine nicht-lineare Aktvierungsfunktion φ.

Jedes Neuron berechnet das Skalarprodukt von $\vec{x}$ und $\vec{w}$, addiert b und sendet den Wert durch die Funktion φ.

$$\text{AN}_{\text{Output}} = \varphi(\vec{x} \cdot \vec{w} + b)$$

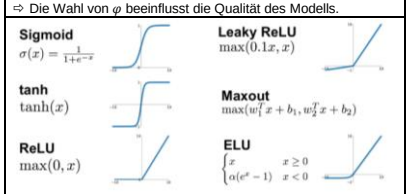
⇨ Das b und $\vec{w} = [w_1, w_2 \dots]$ sind die trainierbaren Parameter.



Aktivierungsfunktion

Die Aktivierungsfunktion φ ist eine beliebige nicht-lineare mathematische Funktion, welche man «frei» wählen kann.

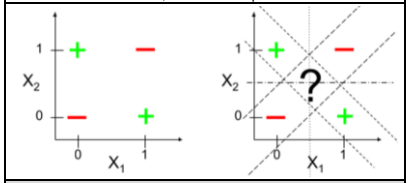
⇨ Die Wahl von φ beeinflusst die Qualität des Models.



Grenzen von Neuronen

Sieht man sich die Aktivierungsfunktionen genauer an, so fällt auf, dass ein einzelnes Neuron nur linear separierbare (binäre) Muster erkennen kann.

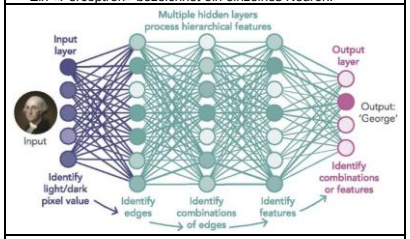
⇨ Vergleiche «Sigmoid-Funktion» mit der linearen Regression.
⇨ Dies gilt auch bei Daten in höheren Dimensionen.
⇨ XOR ist ein bekanntes, nicht linear separierbares Problem.



Artificial Neural Networks (ANN)

Die Probleme eines einzelnen Neurons lassen sich lösen, indem wir mehrere, «hintereinander-gestapelte» Neuronen verwenden, sogenannte «Layers».

⇨ Wir sprechen dann auch von einem «Multi Layer Perceptron»
⇨ Ein «Perceptron» bezeichnet ein einzelnes Neuron.



Das Gebiet von Machine Learning wird aktuell von komplexen ANNs mit Millionen von Parametern dominiert. Wir sprechen daher oft auch von sogenannten «Deep Learning» Architekturen.

⇨ Terminologie: Ein ANN implementiert eine Modellfamilie.
⇨ Einfachere Modelle sind oftmals weniger ausdrucksstark.

Backpropagation

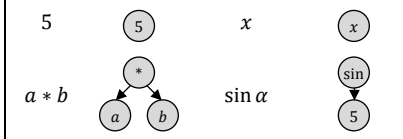
Expression Trees

Ein «Expression Tree» ist eine Datenstruktur für die Darstellung und Berechnung von mathematischen Ausdrücken.

⇨ Die Implementation davon ist für dieses Modul nicht relevant.

Grundbausteine

Ein «Expression Tree» besteht immer aus den gleichen Grundbausteinen:

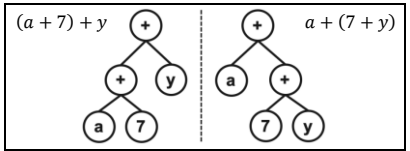


⇨ Wobei die «*» Regel auch für +, −, ÷ und ^ gilt.

Erklärungsbeispiel

Wir können einen Expression Tree für die Formel «$a + 7 + y$» zeichnen.

⇨ Wie wir sehen werden, sind Expression Trees nicht eindeutig.



Backpropagation

Der «Backpropagation»-Algorithmus ist eine effiziente Methode für die Berechnung der Gradienten einer Funktion.

⇨ Die meisten «Optimizer» in ML sind Gradienten-basiert.
⇨ Ohne Backpropagation gäbe es also kein Machine Learning.

Erklärungsbeispiel

Betrachten wir ein Modell in der Form:

$$\hat{y} = ax^2 + bx + c$$

Wir verwenden nun den MSE als Loss L und SGD als unser Optimizer.

$$\text{MSE} = (\hat{y} - y)^2 = (ax^2 + bx + c - y)^2$$
$$\text{SGD} = [a, b, c]_t - \alpha * \left[\frac{\delta L}{\delta a}, \frac{\delta L}{\delta b}, \frac{\delta L}{\delta c}\right]_t$$

Wie wir sehen, müssen wir in jedem Optimierungsschritt den Gradienten der Loss-Funktion L bestimmen.

⇨ Dazu müssen wir also jeweils alle Ableitungen berechnen.

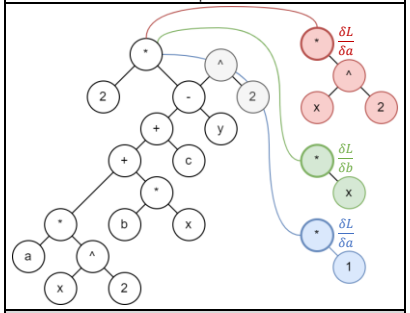
Aufbau des Expression Trees

Rechnen wir die einzelnen Ableitungen im Gradienten aus, so erhalten wir:

$$\frac{\delta L}{\delta a} = 2 * (ax^2 + bx + c - y) * x^2$$
$$\frac{\delta L}{\delta b} = 2 * (ax^2 + bx + c - y) * x$$
$$\frac{\delta L}{\delta b} = 2 * (ax^2 + bx + c - y) * 1$$

Wir erkennen also, dass alle Ableitungen einen identischen Teil beinhalten.

⇨ Wir können dies in einem Expression Tree darstellen.



Anwendung von Backpropagation

Wir können diesen Baum nun mithilfe von Backpropagation auswerten.

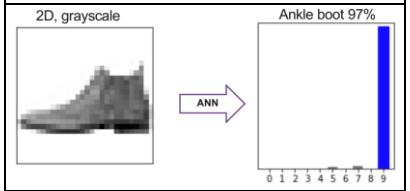
1. **Forward-Pass:** Als erstes Werten wir von «unten nach oben» alle Teilbäume bis zum benötigten Endresultat aus.
2. **Backward-Pass**: Anschliessen gehen wir «zurück» und passen die Parameter an.

⇨ Backpropagation basiert dabei auf «Dynamic Programming».

Convolutional Neural Networks (CNN)

Anwendungsgebiet

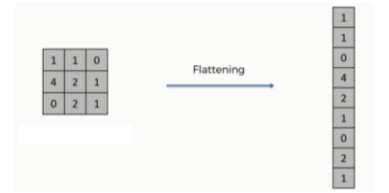
Die Bildverarbeitung ist eines der bekanntesten Einsatzgebiete von Neuralen Netzwerken. ANNs ermöglichen es z.B. beliebige Bilder zu klassifizieren.



Probleme von ANNs

Um ein Bild $\mathbb{R}^{n \times m}$ in einem «normalen» ANN verwenden zu können, müssen wir es in einen Vektor $\mathbb{R}^{n*m}$ umwandeln.

⇨ Diesen Prozess nennt man «Flattening».



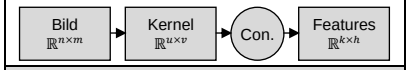
Besonders bei Bildern gehen bei diesem Prozess aber die «Muster» der Daten wie Formen und Strukturen verloren.

⇨ Ein ANN kann diese Muster nur mit Mühe wiederherstellen.

Convolutional Feature Detectors (CFD)

Um dieses Problem zu lösen, versucht man anstelle von «Flattening» die Formen und Strukturen eines Bildes mithilfe von «Convolution» zu erkennen.

⇨ Diese Idee ist wie so oft von unserem Hirn inspiriert.
⇨ Die erkannten Strukturen sind dann unsere «Features».



Funktionsweise

Die Convolution erkennt die Strukturen eines Bildes, indem mehrere «Kernel» über das Bild «geschoben» werden. An jeder Position werden dann die Features berechnet und in einer Map gespeichert.

⇨ Kernel $\mathbb{R}^{u \times v}$ und Bias b sind die trainierbaren Parameter.
⇨ Mehrere Layer können auch detaillierte Features erkennen.

Feature Maps

Wurden die Kernel sinnvoll trainiert, so lassen sich aus den «Feature Maps» die Muster der Daten ableiten.

⇨ Eine Feature Map zeigt also die «Stärke» eines Features an.
⇨ Wir können pro Layer auch mehrere Feature Maps erstellen.

Berechnungen Kernel

$R_1 = 308$ $R_2 = -498$ $R_3 = 164$

$$R = \sum_i \sum_j I_{ij} * K_{ij}$$

⇨ Die Summe der Produkte der einzelnen Komponenten.
⇨ R = Resultat vom Kernel, I = Inputdaten, K = Kernel

Berechnungen Feature Map

$F_1 = 308 - 498 + 164 + 1 = -25$

$$F = \varphi(\sum_i R_i + b)$$

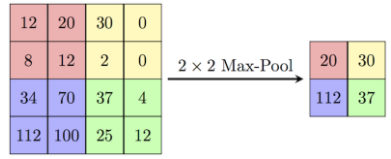
⇨ Nun werden die Kernels verschoben und erneut berechnet.
⇨ Normalerweise wenden wir hier die Aktivierungsfunktion φ an.
⇨ F = Feature an einer bestimmten Position

Max / Average Pooling

In vielen Fällen ist es sinnvoll, die berechneten Daten in einer Feature Map weiter zu reduzieren. Wir können dafür das sogenannte «Pooling» verwenden:

- Max-Pooling: Nimm den grössten Wert in einem definierten $n \times m$ Bereich.
- Average-Pooling: Nimm den Durchschnitt aller Werte in einem $n \times m$ Bereich.

⇨ Andere Pooling-Mechanismen existieren, sind aber selten.
⇨ Wir können so die Anzahl trainierbaren Parameter reduzieren.



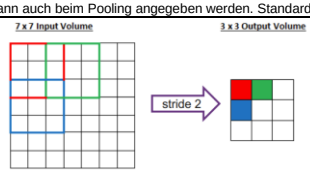
Trotz der Datenreduktion bleibt die Performance des Modells meistens gleich wie bei einem Modell ohne «Pooling».

⇨ Wir können also komplexere ANNs mit weniger Daten bauen.

Strides

Der sogenannte «Stride» definiert die Distanz, mit welcher ein Kernel über das Bild «geschoben» wird.

⇨ Kann auch beim Pooling angegeben werden. Standard ist 1.



Vorteile von CFDs

Die CFDs haben diese Vorteile:

- Translation Invariance: Einzelne Features können überall im Bild erkannt werden.

- Geteilte Gewichte: Durch teilen der Kernels verringert sich der Berechnungsaufwand.
- Parallelisierbar: Die Berechnung der Kernel kann komplett parallelisiert werden.

⇨ «Translation Invariant» bedeutet «Positionsunabhäng»

Anwendungsbeispiele

«Alex Net»

Alex Net gilt als eine der einflussreichsten Arbeiten im Bereich von «Computer Vision» und hat seit seiner Vorstellung im Jahr 2012 zahlreiche wissenschaftliche Arbeiten angeregt.

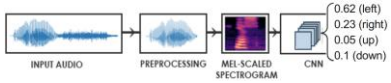
- Art des ANNs: CNN
- Anzahl zu klassifizierende Bilder: 1.2 Mil.
- Anzahl der Klassen: 1000
- Anzahl der Parameter: 60 Mil.
- Anzahl der Neuronen: 650'000
- Top-1 Error Rate: 37.5%
- Top-5 Error Rate: 17.0%

⇨ Ein Bild zum Aufbau befindet sich im Anhang.
⇨ Für die Prüfung muss man evtl. den Aufbau verstehen.

Voice Recognition

Grundsätzlich können wir auch die Spracherkennung als ein einfaches Klassifizierungsproblem betrachten.

⇨ Welches Wort (= Klasse) wurde im Input gesagt?



Wenn wir den Audio-Input nun in ein Spektrogramm (≈ Audiobild) umwandeln, so können wir diese Klassifizierung auch mit einem CNN lösen.

Deep Learning Architectures

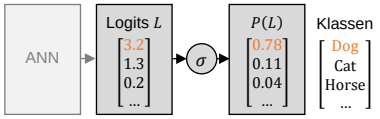
Werkzeuge & Techniken

Softmax-Funktion

Besonders bei der Klassifizierung ist oftmals eine Wahrscheinlichkeit als Output eines ANNs gewünscht. Dies lässt sich mit der Softmax-Funktion erreichen:

$$\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

⇨ Wobei $\vec{x}$ die Logits (numerische Outputs) des ANNs sind.
⇨ Die Softmax-Funktion ist ableitbar (s. Backpropagation).

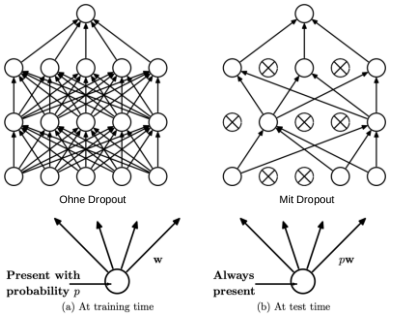


⇨ Merke: Die Summe über $P(L)$ muss immer 1 sein.

Dropout

Die Regularisierungsmethode «Dropout» versucht, Overfitting zu reduzieren, indem es beim Trainieren eines ANNs «zufällig» eine bestimmte Anzahl Neuronen deaktiviert.

⇨ Die Anzahl deaktivierte Neuronen nennt man Dropout Rate.
⇨ Dropout ähnelt von der Idee her den Ensemble Methoden.

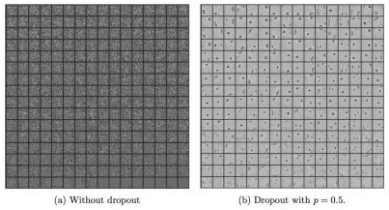


⇨ Die Dropout Rate p ist ein Hyperparameter.
⇨ Dropout wird nur beim Training verwendet (s. Bild).

Co-Adaptation

Ohne Dropout kann eine Co-Adaptation zwischen den einzelnen Feature Maps auftreten. Das bedeutet, dass die Features nur noch in der Menge und nicht mehr allein sinnvolle Resultate liefern.

⇨ Dropout kann diese Co-Adaptation aufbrechen.
⇨ Dies ist der Hauptgrund für die Verwendung von Dropout.



Data Augmentation

Alle ML-Algorithmen (z.B. CNNs) können ausschliesslich mit Daten arbeiten, welche den Trainingsdaten ähneln.

«We cannot expect a network to do better than what it was trained for.»

⇨ CNNs können z.B. keine Rotationen o. Skalierung erkennen.

Probleme

Besonders bei grossen ANNs werden viele Daten für das Trainieren der zahlreichen Parameter benötigt. Wir stossen dabei aber auf einige Schwierigkeiten:

- Daten zu sammeln & annotieren ist teuer.
- Die Datenmengen sind oftmals zu klein.
- Nicht alle Daten sind für ANNs geeignet.

Datenmanipulation

Mit «Data Augmentation» wollen wir dieses Problem lösen, indem wir aus den vorhandenen Daten neue generieren.

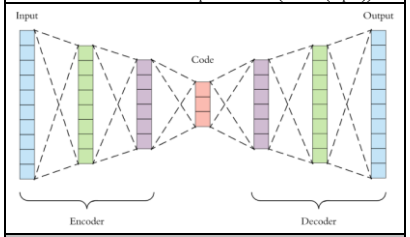
⇨ Die Idee ist, dass die Modelle so besser generalisieren.

Autoencoder

Anwendungsgebiet

Autoencoder haben das Ziel, die eigenen Inputdaten über ein ANN zu reproduzieren. Somit lassen sich z.B. «Denoising»-Aufgaben lösen.

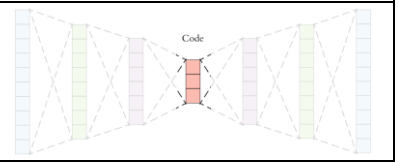
⇨ «Denoising»: Unregelmässigkeiten in z.B. Bildern entfernen
⇨ Ziel des Autoencoders ist: $\text{Input} \approx \text{Decode}(\text{Encode}(\text{Input}))$



Sinn & Zweck

Bei Autoencoder zwingen wir das Netzwerk dazu, so viele Information wie nur möglich durch einen «Bottleneck»-Layer zu übertragen. Der Autoencoder muss also eine enorm kompakte Repräsentation des Inputs (einen «Code») lernen.

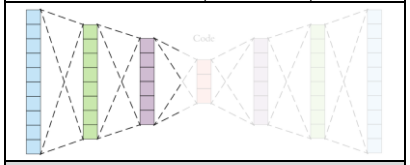
⇨ Der Code ist unser «Feature Vector» im «Latent Space».
⇨ Im «Latent Space» sind ähnliche Elemente nahe beieinander.
⇨ Autoencoder sind also mehr als eine Identitätsfunktion.



Anwendungsgebiet Pretraining

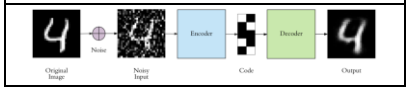
Da Autoencoders keine Labels benötigen, können wir damit Modelle ohne Labels trainieren und dann die gelernten Features (den Encoder) in einem anderen ANN wiederverwenden.

⇨ Wir nennen dieses Konzept «Transfer Learning»
⇨ Autoencoder sind also «Unsupervised / Self-Supervised»



Anwendungsgebiet Denoising

Der Code vom «Bottleneck»-Layer ist eine Repräsentation der wichtigsten Features über alle Inputs hinweg. Da einzelne, isolierte Aspekte («Noise») nicht Teil vom Code sind, können wir diese mit einem Autoencoder entfernen.

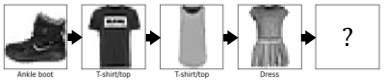


Recursive Neural Networks (RNN)

Anwendungsgebiet

Durch die Verwendung von RNNs lassen sich sequenzielle Daten, wie z.B. Aktienkurse, Texteingaben oder Wetterbilder, sinnvoll verarbeiten.

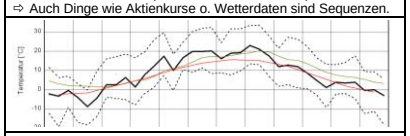
⇨ Sinnvoll bedeutet, dass die Sequenzmuster beachtet werden.
⇨ z.B. Kleidervorhersage basierend auf den letzten n Tagen.



Sequenzielle Daten

Bis anhin sind wir bei unseren ANNs immer von «statischen» Daten ausgegangen. Tatsächlich handelt es sich aber bei den meisten Dingen in der realen Welt um «Sequenzen» von Daten.

⇨ Wörter = Zeichensequenzen, Texte = Wortsequenzen, etc.
⇨ Auch Dinge wie Aktienkurse o. Wetterdaten sind Sequenzen.



Oftmals können wir anhand der Sequenz mehr über die Daten schliessen.

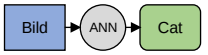
⇨ z.B. «Ich lese aktuell eine …… Zusammenfassung.»

Unterscheidung von ANNs

Mit dieser Idee können wir nun unsere ANNs basierend auf den Ein- und Ausgabedaten in **4** Kategorien unterteilen.

1. One-to-One

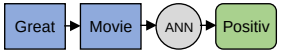
Die Applikation nimmt einen Input und erstellt daraus einen Output.



⇨ Die Daten sind wie bisher «statisch». (z.B. Bildklassifizierung)

2. Many-to-One

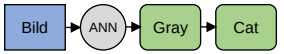
Die Applikation nimmt eine Sequenz von Inputdaten und erstellt einen Output.



⇨ z.B. Sentiment-Analyse von einem Text

3. One-to-Many

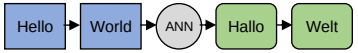
Die Applikation nimmt einen Input und erstellt eine Sequenz von Outputdaten.



⇨ z.B. Automatische Beschriftung von Bildern

4. Many-to-Many

Die Applikation nimmt eine Sequenz von Inputdaten und erstellt daraus eine Sequenz von Outputdaten.



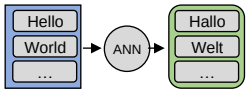
⇨ z.B. Automatische Übersetzung von Texten

Probleme mit ANNs und CNNs

ANNs

Könnte man nicht einfach die Inputsequenz in einen Vektor codieren und durch ein reguläres ANN senden?

⇨ Wir codieren eine Sequenz einfach in ein One-to-One ANN.
⇨ Warum also überhaupt diese Unterteilungen?

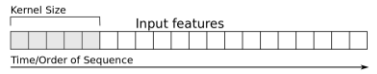


Das Problem hierbei ist, dass normale ANNs die Reihenfolge der Eingabedaten nicht beachten. Jedes Element im Vektor wird also unabhängig von allen anderen Elementen betrachtet.

⇨ Genau diese Abhängigkeiten wollen wir aber miteinbeziehen.
⇨ ANNs sind daher nicht geeignet für unsere Problemstellung.

CNNs

Könnte man nicht einfach eine Sequenz mit n Elementen als Bild $\mathbb{R}^{n \times 1}$ interpretieren und ein CNN darauf anwenden?



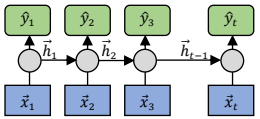
Tatsächlich können wir mit den CFD wiederkehrende Muster in Sequenzen erkennen. Jedoch können auch CNNs keine Informationen über die Reihenfolge der Eingabedaten speichern.

⇨ CNNs können also Muster wie «Gut» o. «Schlecht» erkennen.
⇨ Wir können so z.B. Textsequenzen mit CNNs klassifizieren.
⇨ Für andere Aufgaben sind CNNs aber ungeeignet.

Recursive Neural Networks (RNN)

Funktionsweise

Wir können die Probleme von ANNs und CNNs lösen, indem wir die Eingabedaten nacheinander in ein RNN einfügen. In jedem Schritt t speichern wir nun einen Teil der Berechnungen h_t ab, welchen wir dann im nächsten Schritt wiederverwenden können.



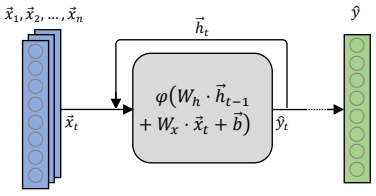
⇨ Beachte, dass es sich bei $\hat{y}$ auch um einen Vektor handelt.

Unfolding

In der obigen Darstellung haben wir das RNN «entfalten», also über die Sequenz her ausgespannt. In dieser Form können wir das Netzwerk auch als ein reguläres ANN mit vielen Layers betrachten. Oft werden RNNs aber als sogenannte «Memory Cells» dargestellt.

⇨ Wir nennen dieses «Entfalten» auf Englisch «Unfolding».
⇨ «Unfolding» wird auch beim Trainieren von RNNs angewendet.

Memory Cell



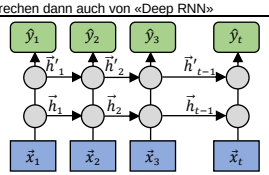
⇨ W_h, W_x und $\vec{b}$ sind die trainierbaren Parameter.
⇨ Diese 3 Parameter werden von allen Elementen geteilt.
⇨ φ ist eine beliebige, nicht-lineare Aktvierungsfunktion.

Varianten von RNNs

Stacked RNNs

Wie mit den «Layers» können wir auch mehrere Memory Cells von RNNs hintereinanderstellen und trainieren.

⇨ Wir sprechen dann auch von «Deep RNN»

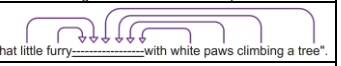


⇨ Der Output von einem RNN wird zum Input des nächsten.
⇨ Das Endresultat aller RNNs befindet sich dann in $\hat{y}$.

Bidirectional RNNs

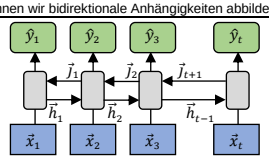
In vielen Anwendungsfällen kann es vorkommen, dass ein Element eines RNN von der ganzen Sequenz abhängt.

⇨ Die Spracherkennung ist ein dominantes Beispiel dafür.



Bidirektionale RNNs kombinieren nun RNNs die «vorwärts» durch die Sequenz laufen mit RNNs die «rückwärts» durch die Sequenzen laufen.

⇨ So können wir bidirektionale Anhängigkeiten abbilden.



$$\vec{h}_t = \varphi(W_h \cdot \vec{h}_{t-1} + W_{hx} \cdot \vec{x}_t + \vec{b}_h)$$
$$\vec{j}_t = \varphi(W_j \cdot \vec{j}_{t+1} + W_{jx} \cdot \vec{x}_t + \vec{b}_j)$$
$$\hat{y}_t = func(\vec{h}_t, \vec{j}_t)$$

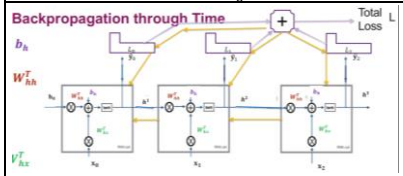
⇨ Wobei $func$ eine für dieses Modul nicht relevante Funktion ist.
⇨ Beachte, dass Gewichte und Bias **nicht** geteilt werden.

Trainieren von RNNs

Wir können RNNs wie gewohnt mithilfe von «Backpropagation» trainieren.

1. Initialisiere die Gewichte
2. **Forward-Pass:** Laufe «vorwärts» über alle Elemente der Sequenz und berechne für jedes Element den jeweiligen Loss.
3. **Backward-Pass:** Laufe nun «rückwärts» über alle Elemente der Sequenz und passe für alle Elemente die Gewichte an.
4. Wiederhole, bis der Loss minimal wird.

⇨ Wir verwenden dazu das «Unfolding».



⇨ Unfolding + Backprop. = «Backpropagation through Time»

Loss

Die Loss-Funktion hängt wie so oft vom jeweiligen Problem ab. Oft werden aber diese Loss-Funktionen verwenden:

- Binary Cross Entropy (Binärklassifizierung)
- Categorical Cross Entropy (n-Klassifizierung)
- Root Mean Square Error (Regression)

Optimizer

In den meisten Fällen verwenden wir einen Gradienten-basierten Optimizer wie z.B. «Stochastic Gradient Descent».

Probleme von RNNs

Vanishing & Exploding Gradient

Beim «Backward-Pass» vom Backpropagation-Algorithmus kann es vorkommen, dass die Gradienten der Loss-Funktion nach einigen Iterationen enorm klein oder enorm gross werden.

⇨ Wir nennen dies «Vanishing / Exploding Gradient Problem»
⇨ Alle Gradienten-basierten Algorithmen haben dieses Problem.
⇨ Wegen dem «Unfolding» ist es bei RNN besonders prävalent.

Vanishing Gradient

Probleme von Vanishing Gradient sind:

- Die Gewichte ändern sich nur noch minimal.
- Das Modell wird nicht mehr verbessert.
- Besonders «ältere» Elemente der Sequenz verlieren ihre Wichtigkeit.

⇨ Das Modell lernt nur noch kurzfristige Zusammenhänge.

Exploding Gradient

Probleme von Exploding Gradient sind:

- Die Gewichte ändern sich enorm.
- Das Modell wird sehr instabil.
- Die Werte können den gültigen Wertebereich übersteigen. («Value Overflow»)

Lösungen

Tatsächlich gibt es zahlreiche Ansätze, wie sich das Problem lösen lässt:

- Aktivierungsfunktionen wie «ReLU» oder «Leaky ReLU» verhindern das Problem.
- Mit «Gradient Clipping» können wir die Gradienten in einem definierten Bereich halten.
- Eine gute Wahl der Initialparameter wirkt sich ebenfalls positiv auf das Problem aus.

⇨ Alle diese Ansätze können für RNN verwendet werden.
⇨ Die nachfolgenden Ansätze sind aber besonders relevant.

Skip Connections

Anstatt wie bisher für die Berechnung von $\hat{y}_t$ und h_t den Wert h_{t-1} zu verwenden, können wir auch einen beliebigen wert an der Stelle h_{t-d} nehmen.

⇨ Wir können so dem «Vanishing Gradient» entgegenwirken.
⇨ Wir nennen das d auch den «Delay».

Gated RNN

Anwendungsgebiet

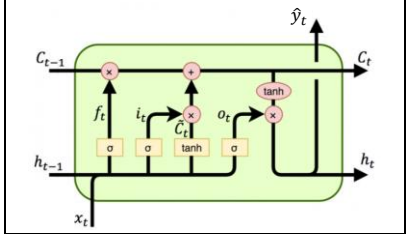
Gated RNNs sind eine Erweiterung der normalen RNNs. Sie sind in der Lage, auch langfristige Zusammenhänge innerhalb einer Sequenz zu merken.

⇨ Sie basieren auf der Idee von «Leaky Units»

Long Short-Term Memory (LSTM)

LSTMs funktionieren ähnlich wie normale RNNs, besitzen aber einen komplexeren Aufbau der «Memory Cell» und einen zusätzlichen «Cell State».

⇨ LSTMs können so auch langfristige Zusammenhänge lernen.



Gates

Die «Gates» erlauben es einem LSTM, Informationen zu merken und bei Bedarf auch wieder zu vergessen. LSTMs bestehen dabei aus **4** Gates.

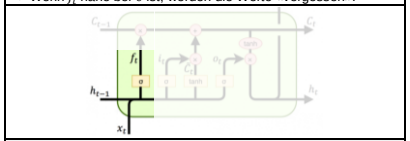
⇨ LSTMs lernen also nur die wichtigen Informationen.
⇨ Im Grunde beinhalten Gates lediglich eine Sigmoidfunktion.

1. Forget Gate

Das «Forget Gate» vergisst irrelevante Informationen aus den vorherigen Teilschritten. Wir berechnen also:

$$f_t = \sigma(W_{fh} \cdot \vec{h}_{t-1} + W_{fx} \cdot \vec{x}_t + \vec{b}_f)$$

⇨ Wobei σ die Sigmoidfunktion ist.
⇨ Wenn f_t nahe bei 0 ist, werden die Werte «vergessen».

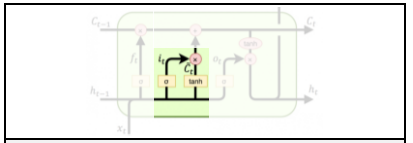


2. Compute Gate

Das «Compute Gate» berechnet den neuen «Cell State». Die Sigmoidfunktion soll dabei die relevanten und irrelevanten Informationen «aussortieren».

$$i_t = \sigma(W_{ih} \cdot \vec{h}_{t-1} + W_{ix} \cdot \vec{x}_t + \vec{b}_i)$$
$$\tilde{C}_t = \tanh(W_{ch} \cdot \vec{h}_{t-1} + W_{cx} \cdot \vec{x}_t + \vec{b}_c)$$

⇨ Wird auch «Input Gate» oder «Store Gate» genannt.

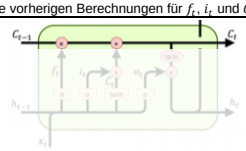


3. Update Gate

Das «Update Gate» verbinden den alten mit dem neuen «Cell State». Dieses Resultat wird anschliessen an die nächste «Cell» weitergereicht.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

⇨ Siehe die vorherigen Berechnungen für f_t, i_t und $\tilde{C}_t$.

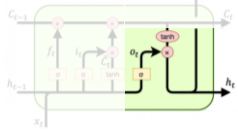


4. Output Gate

Das «Output Gate» berechnet nun den finalen Output der aktuellen «Cell».

$$o_t = \sigma(W_{oh} \cdot \vec{h}_{t-1} + W_{ox} \cdot \vec{x}_t + \vec{b}_o)$$
$$\hat{y}_t = \vec{h}_t = o_t * \tanh(C_t)$$

⇨ Siehe die vorherigen Berechnungen für C_t.



Varianten von LSTMs

Die Grundidee von LSTMs lässt sich in verschiedenen Varianten finden:

- Peephole Variation
- Coupled Gate Variation
- Depth Gated / Clockwork RNNs

Probleme mit LSTMs

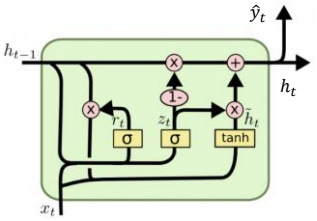
LSTMs haben enorm viele trainierbare Parameter, etwa 4-mal so viele wie wir in normalen RNNs finden.

⇨ Dieses Problem lässt sich leider nur schwer lösen.
⇨ Architekturen wie «GRU» sollen diese Anzahl aber verringern.

Gated Recurrent Units (GRU)

GRUs funktionieren vom Prinzip her ähnlich wie LSTMs, benötigen aber weniger trainierbare Parameter.

⇨ GRUs besitzen ausserdem keinen «Cell State»
⇨ Somit sind alle Parameter von «aussen» Sichtbar.



Gates

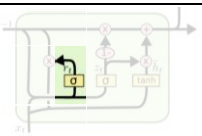
GRUs bestehen aus **3** Gates.

⇨ Sie haben also 3-Mal so viele Parameter wie RNNs.

1. Reset Gate

Das «Reset Gate» soll die kurzfristigen Zusammenhänge merken. Das Resultat dieses Gates gibt an, welche Informationen vergessen werden sollen.

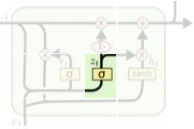
$$r_t = \sigma(W_{rh} \cdot \vec{h}_{t-1} + W_{rx} \cdot \vec{x}_t + \vec{b}_r)$$



2. Update Gate

Das «Update Gate» soll die langfristigen Zusammenhänge merken.

$$z_t = \sigma(W_{zh} \cdot \vec{h}_{t-1} + W_{zx} \cdot \vec{x}_t + \vec{b}_z)$$

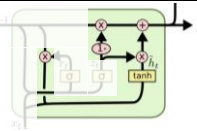


3. Output Gate

Der finale Output der aktuellen «Cell» wird nun wiefolgt berechnet:

$$\tilde{h}_t = \tanh(r_t * (W_{\tilde{h}h} \cdot \vec{h}_{t-1}) + W_{\tilde{h}x} \cdot \vec{x}_t + \vec{b}_{\tilde{h}})$$
$$\hat{y}_t = \vec{h}_t = (1 - z_t) * \vec{h}_{t-1} + z_t * \tilde{h}_t$$

⇨ Siehe die vorherigen Berechnungen für r_t und z_t.
⇨ Der «Reset» r_t wird also direkt auf h_{t-1} angewandt.



LSTM vs. GRU

Die beiden Arten von RNN unterscheiden sich grundsätzlich in ihrem Aufbau und der Komplexität. Es gilt:

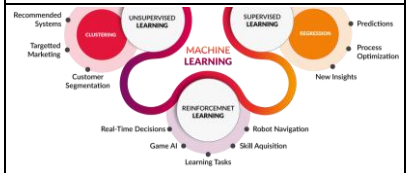
- **GRU:** Sind einfacher u. kompakter, wodurch das Training schneller wird. Sie können aber nicht so lange Sequenzen lernen.
- **LSTM:** Sind komplexer u. schwerfälliger, können aber auch lange Sequenzen lernen. Das Training dauert normalerweise länger.

⇨ GRUs haben auch ein Gate weniger als LSTM.
⇨ Grob gilt: Wenige Daten = GRU, Viele Daten = LSTM.
⇨ Wie immer hängt dies aber von der Anwendung ab.

Reinforcement Learning (RL)

Grundkonzepte

Reinforcement Learning bildet neben Supervised und Unsupervised-Learning eine eigene Klasse von AI-Algorithmen basierend auf dem Konzept von «Trial-and-Error» Learning.



Was bedeutet «Learning»?

Generell betrachtet beschreibt «Lernen» einen Prozess der Aneignung von neuen Informationen. Für Menschen beinhaltet dies insbesondere das Ändern des Verhaltens basierend auf Erfahrungen.

⇨ Wir sprechen in diesem Fall von «Behavioral Learning».
⇨ Wir können diese Idee auch in Machine Learning anwenden.

Grundidee

In RL lässt man ein «Agent» mit einem «Environment» interagieren. Das Ziel vom «Agent» ist es dabei, einen bestimmten «Reward» zu maximieren.

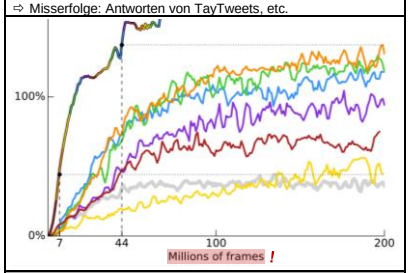
⇨ Wir verwenden dabei das «Trial-and-Error» Prinzip.
⇨ Wir lernen also **nicht** von vorgesammelten Daten.
⇨ Stattdessen «generiert» ein Agent seine eigenen Daten.

Schwierigkeiten

Auch wenn sich Reinforcement Learning konzeptionell gut anhört, müssen wir dabei folgende Punkte beachten:

- Sample Efficiency: RL-Algorithmen brauchen Millionen von Iterationen, tausende Simulationsstunden und enorme Rechenleistung.
- Reward Definition: Die Definition vom Reward kann schwieriger sein als man denkt.

⇨ Erfolgsgeschichten: AlphaGo, Gran Turismo Sophy, etc.
⇨ Misserfolge: Antworten von TayTweets, etc.



Grundbausteine

Grundsätzlich besteht Reinforcement Learning immer aus **2** Elementen:

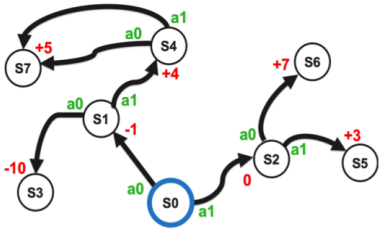
- Environment: Beinhaltet die States, Actions, Rewards und Transitions.
- Agent: Versucht ein RL-Problem durch Interaktionen mit dem Environment zu lösen.

Markov Decision Process (MDP)

Wir können das Konzept des Environments auch in einem «Markov Decision Process» formalisieren. Ein MDP ist ein generelles, mathematisches Framework bestehend aus **4** Elementen:

1. Eine Menge von States $S = \{S_0, S_1 \dots S_n\}$
2. Eine Menge von Actions $A = \{a_0, a_1 \dots a_m\}$
3. Die Rewards $R \subset S \times \mathbb{R}$
4. Die Transitions P oder $T \subset S \times A \times S$

⇨ MDPs werden auch in vielen anderen Bereichen verwendet.
⇨ Wir können ein MDP auch in einem Diagramm darstellen.



⇨ In diesem Falls sind S_3, S_5, S_6 und S_7 die «Terminal-States».
⇨ Terminal-States beenden eine sogenannte «Episode».
⇨ Wir erhalten den Reward **bei der Ankunft** in einem State.

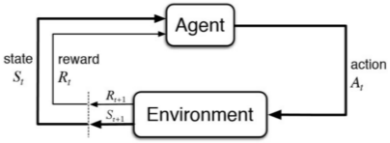
Funktionsweise

Bei Reinforcement Learning interagiert ein Agent nun mit unserem definierten Environment. Das bedeutet konkret:

1. Der Agent nimmt eine Action A_t.
2. Er sieht eine Veränderung vom State S_{t+1}.
3. Er erhält einen Reward R_{t+1}

Dieser Prozess wird nun unendlich lange oder bis zum Erreichen eines Terminal-States wiederholt.

⇨ Der Agent will dabei den Reward maximieren (= hedonisch).
⇨ Um dies zu erreichen, muss er ein optimales Verhalten lernen.



⇨ Jede Action hat (potenziell) einen Einfluss auf das Environment.
⇨ Das Environment wechselt also in einen neuen State S_{t+1}.

Policy π

Das Verhalten eines Agents wird vollständig durch seine Policy π definiert. Die Policy ist eine Funktion, welche States S und Actions A auf Wahrscheinlichkeiten abbildet.

$$\pi = \begin{cases} S \times A \to [0,1] \\ (s,a) \mapsto P(a_t = a \mid s_t = s) \end{cases}$$

⇨ Wir schreiben z.B.: $\pi(s = S_5, a = a_{links}) = 0.76$
⇨ «Die Wahrscheinlichkeit in S_5 nach a_{links} zu gehen ist 76%.»

Ziel des Agents

Ein RL-Agent versucht also immer, eine optimale Policy π^* zu finden, welche den totalen Reward maximiert.

⇨ Wir können die Policy also als «Modell» interpretieren.

Temporal Difference Learning (TDL)

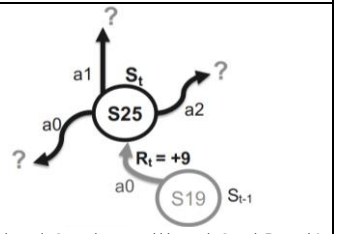
Ausgangslage

Nehmen wir an, wir haben ein Environment vollständig anhand des MDP spezifiziert. Wir können nun die einzelnen Elemente in einer Tabelle darstellen.

⇨ Lese: Von S_{25} nehme ich a_0 und lande in S_{94} mit Reward 4.
⇨ Diese Tabelle ist deterministisch (Mit a_0 lande ich immer in S_{94})
⇨ Oft sind solche Tabellen stochastisch (60% in S_{94}, 40% in S_{67})

Sicht des Agents

In den meisten Fällen kennt der Agent die Definitionen des Environments nicht, sondern muss diese durch Interaktionen mit dem Environment herausfinden.



⇨ Lese: Ich war in S_{19}, nahm a_0 und bin nun in S_{25} mit Reward 9.
⇨ Der Agent muss sich nun für die nächste Aktion entscheiden.
⇨ Nach und nach lernt der Agent die besten Aktionen.

State Value Function $V(s)$

Die State Value Function $V(s)$ gibt uns den erwarteten, totalen Reward eines States unter Verwendung von Policy π an. Wir schreiben dies als:

$$V(s) = \mathrm{E}_{\tau\sim\pi}(R \mid s)$$

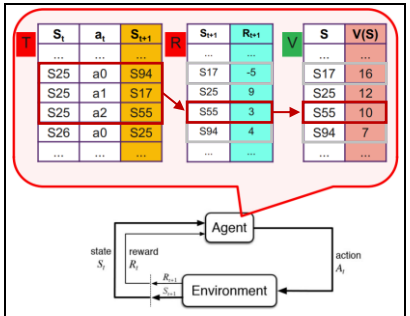
Wir können $V(s)$ verwenden, um die Qualität einer Policy π zu evaluieren und somit die optimale Policy π^* zu finden.

⇨ Ausgeschrieben gilt also: $V(S_t) = \mathrm{E}(R_{t+1} + R_{t+2} + \cdots)$
⇨ Denke: Der totale Reward, wenn ich von S_t der Policy π folge.
⇨ Anmerkung: Die Notation kann zum Teil etwas abweichen.

Anwendungsbeispiel

Nehmen wir an, dass der Agent die Werte T, R und $V(s)$ kennt. Wir können nun anhand der State Value Function $V(s)$ die optimale Aktion ablesen.

⇨ Im Normalfall müssen T, R und $V(s)$ zuerst gelernt werden.



⇨ Lese: In S_{25} wähle Aktion a_2 da $V(S_{55}) + R_{55}$ am grössten ist.

Linearität von Erwartungswerten

Das «E» in der $V(s)$-Formel beschreibt einen Erwartungswert, also einen Wert, der einer bestimmten Wahrscheinlichkeit unterliegt. Allgemein gilt dabei:

$$\mathrm{E}(X + Y + Z) = \mathrm{E}(X) + \mathrm{E}(Y) + \mathrm{E}(Z)$$

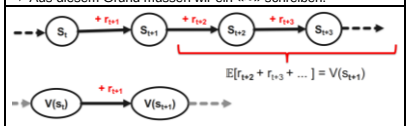
Wir können also daraus schliessen:

$$V(S_t) = \mathrm{E}(R_{t+1}) + \mathrm{E}(R_{t+2} + \cdots)$$

Woraus nun ausserdem gilt:

$$V(S_t) \approx R_{t+1} + V(S_{t+1})$$

⇨ Achtung: Der Erwartungswert ist nur eine «Erwartung».
⇨ Der reale Wert kann abweichen, es gilt also $\mathrm{E}(R_{t+1}) \neq R_{t+1}$.
⇨ Aus diesem Grund müssen wir ein «≈» schreiben.



Differenz δ

Wir können den Unterschied zwischen realem und erwartetem Wert mit einer Differenz δ ausdrücken:

$$\mathrm{E}(R_{t+1}) = R_{t+1} \pm \delta$$

Reward Prediction Error (RPE)

Basierend auf diesen Überlegungen können wir nun unsere Formel mit der Differenz δ_t ergänzen:

$$V(S_t) + \boldsymbol{\delta_t} = R_{t+1} + V(S_{t+1})$$

Diese Differenz δ_t bezeichnen wir nun auch als den Reward Prediction Error:

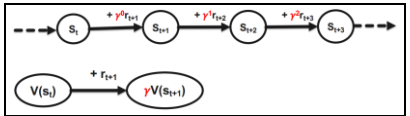
$$\mathrm{RPE}: \boldsymbol{\delta_t} = R_{t+1} + V(S_{t+1}) - V(S_t)$$

⇨ Wir initialisieren $V(s)$ also mit «inkorrekten» Werten (z.B. 0)
⇨ Unser RL-Algorithmus soll nun die Differenz δ minimieren.
⇨ Konvergieren die Werte von $V(s)$, wird δ im Durchschnitt 0.

Discount Factor γ

Wir können in der Formel oben erkennen, dass wir alle zukünftigen Rewards grundsätzlich gleichbehandeln. Oftmals macht es jedoch Sinn, dass wir nähere Rewards den entfernteren vorziehen.

⇨ Hätte ich lieber 10.- jetzt oder 50.- in 100 Jahren?
⇨ Lange Zeitspannen kommen immer mit einem Risiko.
⇨ Denke: «Lebe ich in 100 Jahren noch?»



Der Discount Factor implementiert diese Idee des «Vernachlässigens» durch einen konstanten Wert γ. Daraus folgt:

$$V(S_t) = \mathrm{E}(\gamma * R_{t+1} + \gamma * R_{t+2} + \cdots)$$

Woraus wir nun ableiten:

$$V(S_t) + \delta_t = R_{t+1} + \boldsymbol{\gamma} * V(S_{t+1})$$

$$\mathrm{RPE}: \delta_t = R_{t+1} + \boldsymbol{\gamma} * V(S_{t+1}) - V(S_t)$$

⇨ Setzen wir $\gamma = 1$, so erhalten wir die ursprüngliche Formel.
⇨ Anstatt vom Reward R sprechen wir dann oft vom «Return G».

Temporal Difference

Die rechte Seite der Gleichung nennen wir auch die «Temporal Difference».

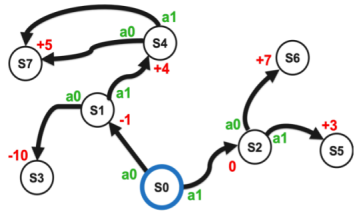
$$\delta_t = R_{t+1} + \gamma * V(S_{t+1}) - V(S_t)$$

⇨ Daher kommt der Name «Temporal Difference Learning»

Rechenbeispiel

Berechne den Wert $V(S_0)$, wobei gilt $\tau = S_0 \to S_1 \to S_4 \to S_7$ und $\gamma = 0.5$

⇨ Das τ bezeichnet die «Trajectory», also den definierten Pfad.
⇨ Da der Pfad vollständig definiert ist, können wir die Erwartungswerte komplett ausrechnen. Daher gilt $\delta = 0$.



Grundformel: $V(S_t) = R_{t+1} + \gamma * V(S_{t+1})$
Idee: Berechnung von «Hinten» nach «Vorne»

$$V(S_7) = 0$$
$$V(S_4) = 5 + 0.5 * 0 = 5$$
$$V(S_1) = 4 + 0.5 * 5 = 6.5$$
$$V(S_0) = -1 + 0.5 * 6.5 = \underline{2.25}$$

⇨ Bei Terminal-States gilt immer $V(s) = 0$.

Policy Evaluation

Um nun die «realen» Werte für $V(s)$ und somit anschliessend eine optimale Policy π^* zu finden, wenden wir nun iterativ einen Optimierungsschritt an:

$$\mathrm{RPE} = R_{t+1} + \gamma * V(S_{t+1}) - V(S_t)$$

$$V(S_t) \longleftarrow V(S_t) + \alpha * \mathrm{RPE}$$

⇨ Wobei wir die Differenz δ_t hier direkt als RPE bezeichnen.
⇨ Der Hyperparameter α ist die «Learning Rate».

Pseudocode

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

Anmerkung

In diesem Optimierungsschritt lernen wir noch keine optimale Policy π^*, sondern berechnen lediglich die Werte $V(s)$ für eine gegebene Policy π.

⇨ Wir «evaluieren» also eine gegebene Policy π.
⇨ Aus diesem Grund nennen wir es auch «Policy Evaluation».

Policy Improvement

Ausgangslage

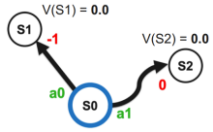
Bisher haben wir gesehen, wie wir die Qualität einer Policy π ermitteln können. Wie finden wir nun aber die optimale Policy π^* mit dem grössten Reward?

⇨ Das ist schliesslich unser Ziel in Reinforcement Learning.

Exploration-Exploitation Dilemma

Am Anfang von RL kennt unser Agent die Rewards im Environment noch nicht. Wir müssen diese zuerst herausfinden.

⇨ Wir müssen also zuerst die $V(s)$ Werte ermitteln.
⇨ Im Normalfall müssen wir sogar die Transitions T lernen.



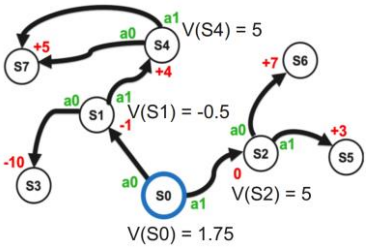
Nach einer bestimmten Zeit soll der RL-Agent aber damit beginnen, dem vielversprechendsten Pfad zu folgen. Er soll sein Wissen also ausnutzen.

⇨ Doch wann soll dieser Wechsel stattfinden?
⇨ Wir müssen Erkunden und Ausnutzen irgendwie balancieren.
⇨ Dies nennen wir das «Exploration-Exploitation Dilemma».

Random Policy Explore

Die «Random Policy» wählt die nächste Aktion jeweils **zufällig**. Durch dieses Verhalten erkundet die Random Policy immer das gesamte Environment.

⇨ Wir können so also die $V(s)$ Werte sinnvoll approximieren.

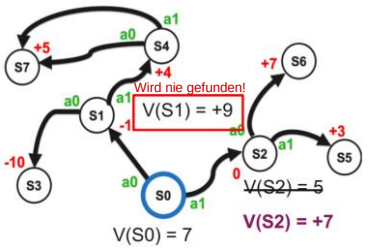


$V(S_2)_R = 0.5 * 7 + 0.5 * 3 = 5$ $V(S_4)_R = 0.5 * 5 + 0.5 * 5 = 5$
$V(S_1)_R = 0.5 * -10 + 0.5 * (4 + V(S_4)_R) = -0.5$
$V(S_0)_R = 0.5 * (-1 + V(S_1)_R) + 0.5 * (0 + V(S_2)_R) = 1.75$

Greedy Policy Exploit

Die «Greedy Policy» wählt immer die Aktion, welche den **grössten** Reward bringt. Durch dieses Verhalten maximiert die Policy den Reward, findet aber keine potenziell besseren Pfade mehr.

⇨ Wir könnten z.B. zuerst Random und dann Greedy verwenden.



$V(S_1)_G = V(S_1)_R$ $V(S_2)_G = 1 * 7 + 0 * 3$
$V(S_0)_G = 0 * (-1 + V(S_1)_G) + 1 * (0 + V(S_2)_G) = 7$

⇨ Dieses Diagramm basiert auf dem vorherigen.
⇨ Da $V(S_1)_R - 1 < V(S_2)_R + 0$ wird der Pfad nicht mehr erkundet.
⇨ Somit ändert sich $V(S_1)_G$ nicht mehr, obwohl $V(S_1)_G > 7$ wäre.

Epsilon-Greedy Policy (EGP)

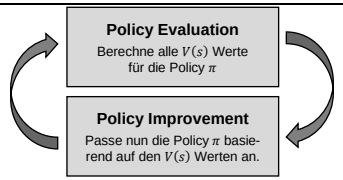
Um nun die beste Kombination von «Random» und «Greedy» Policy zu bekommen, müssen wir die beiden irgendwie balancieren. Die EGP erreicht dies durch einen neuen Parameter ε.

- Mit Wahrscheinlichkeit $1 - \varepsilon$: Greedy Action
- Mit Wahrscheinlichkeit ε: Random Action

⇨ Die Wahl des Hyperparameters ε ist dabei offen.
⇨ Das «ε» kann z.B. fix sein oder sich über die Zeit verändern.
⇨ Oft sieht man $\varepsilon = 0.1$ oder $\varepsilon \in [0.05, 0.95]$ (von 0.95 nach 0.05)

Policy Iteration

Wir können nun die Idee von «Policy Evaluation» und «Policy Improvement» miteinander kombinieren.



⇨ Die Policy Evaluation ist fertig, wenn die Werte konvergieren.
⇨ Die Policy Improvement soll weiterhin erkunden (z.B. EGP)
⇨ Natürlich sind hier auch Varianten möglich.

Anmerkung

Der Begriff «Policy Improvement» ist in diesem Fall etwas irreführend, da die Policy an sich nicht «verbessert» wird.

⇨ Wir wählen bloss die Aktion und verbessern so die $V(s)$ Werte.
⇨ Am Ende verwenden wir **immer** die Greedy-Policy.
⇨ Wir werden aber noch andere Algorithmen kennen lernen.

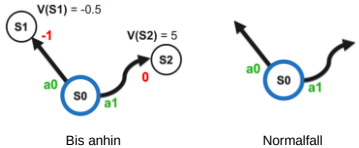
Q-Learning / SARSA

Ausgangslage

Wir haben nun gesehen, wie wir die $V(s)$ Werte für das Finden der optimalen Policy π^* verwenden können. Jedoch haben wir dabei ein wenig gemogelt:

- Wir haben die «beste» Aktion immer basierend auf dem Reward R_{t+1} und dem Wert $V(S_{t+1})$ ausgewählt.
- Normallerweise kennen wir aber nur den State S_t, den Reward R_t und die Actions A_t.

⇨ Allgemein kennt der Agent die nächsten States S_{t+n} nicht.
⇨ Denke z.B. AlphaGo: Der nächste State hängt vom Zug des Agents **und** vom Zug des Menschen ab.



Bis anhin

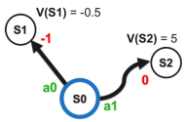
Normalfall

State-Action Values $Q(s,a)$

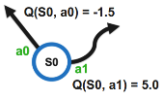
Die State-Action Values (Q-Values) lösen dieses Problem, indem sie den zu erwartenden Reward nicht den States, sondern den Actions zuordnen.

$$Q(s,a) = \mathrm{E}_{\tau\sim\pi}(R \mid s,a)$$

⇨ Nehme ich in S_1 die Action a_0, erwarte ich Reward $Q(S_1, a_0)$.
⇨ Q-Values sind wie die $V(s)$ Werte von der Policy π abhängig.



Wähle Aktion a_1 da $V(S_2) + R_2 > V(S_1) + R_1$



Wähle Aktion a_1 da $Q(S_0, a_1) > Q(S_0, a_0)$

Policy Evaluation

Grundsätzlich gilt dabei die exakt selbe Formel wie bei den $V(s)$ Werten:

$$Q(S_t, a_t) + \delta_t = R_{t+1} + \gamma * Q(S_{t+1}, a_{t+1})$$

Woraus wir nun den RPE sowie den Optimierungsschritt ableiten können:

$$\mathrm{RPE} = R_{t+1} + \gamma * Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t)$$
$$Q(S_t, a_t) \leftarrow Q(S_t, a_t) + \alpha * \mathrm{RPE}$$

Evaluationsmethoden

Es gibt verschiedene Methoden für die Policy Evaluation mit Q-Values. Die Grundidee ist, dass wir im State S_t eine Aktion a_t nehmen und dann im nächsten Schritt S_{t+1} die alten Werte anpassen.

⇨ Im Modul haben wir SARSA und Q-Learning angeschaut.

SARSA

Diese Methode funktioniert wiefolgt:

1. Wähle in S_t eine Aktion a_t
2. Führe die Aktion aus und lande in S_{t+1}
3. Wähle in S_{t+1} eine Aktion a_{t+1}
4. Passe nun die Werte von S_t mittels des Werts $Q(S_{t+1}, a_{t+1})$ und R_{t+1} an.

⇨ Wiederhole dies nun iterativ bis zum Ende der Evaluation.
⇨ SARSA steht dabei für «State-Action-Reward-State-Action»
⇨ Die gewählte Aktion a_{t+1} kommt in Q vor. («On-Policy»)

Pseudocode

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

⇨ Wobei $S' = S_{t+1}$ und $A' = a_{t+1}$ gilt.

Q-Learning

Diese Methode funktioniert wiefolgt:

1. Wähle in S_t eine Aktion a_t
2. Führe die Aktion aus und lande in S_{t+1}
3. Wähle in S_{t+1} eine Aktion a_{t+1}
4. Ermittle in S_{t+1} nun die «beste» Aktion a^*_{t+1}
5. Passe nun die Werte von S_t mittels des Werts $Q(S_{t+1}, a^*_{t+1})$ und R_{t+1} an.

⇨ Wiederhole dies nun iterativ bis zum Ende der Evaluation.
⇨ Die beste Aktion a^*_{t+1} ist die Aktion mit dem grössten Q Wert.
⇨ Die gewählte Aktion a_{t+1} kommt nicht in Q vor. («Off-Policy»)

Pseudocode

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\textit{terminal}, \cdot) =$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

⇨ Wobei $S' = S_{t+1}$ und $A' = a_{t+1}$ gilt.

Deep Reinforcement Learning (DRL)

Ausgangslage

Probleme von Q-Values

Denken wir an komplexe Videospiele (z.B. Dota), so stossen wir mit den Q-Values schnell an die Grenzen:

- Videospiele haben enorm viele States S.
- Videospiele haben viele Actions A.
- Wir haben also enorm viele Q-Values Q.
- Wir müssen alle diese Q-Values lernen!

⇨ Die Q-Values skalieren also enorm schlecht.
⇨ Wir können dies in einer Tabelle visualisieren.

	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
1	0.068	0.226	0.160	0.699	0.267	0.657	0.563	0.261	0.796	0.953
2	0.268	0.832	0.184	0.918	0.637	0.718	0.383	0.501	0.731	0.634
3	0.935	0.331	0.965	0.282	0.006	0.336	0.400	0.467	0.956	0.104
4	0.471	0.942	0.501	0.241	0.526	0.075	0.931	0.952	0.953	0.116
5	0.123	0.036	0.029	0.933	0.088	0.986	0.977	0.326	0.624	0.292

⇨ Was wäre mit 1'000 Actions und 1'000'000 States?

Q-Values haben ausserdem ein Problem mit der Generalisierung. Oftmals können wir nämlich in «ähnlichen» States auch ähnliche Aktionen anwenden.

⇨ z.B. Ist die Veränderung von einem Pixel meist nicht relevant.
⇨ Die Aktion mit Pixel A_{Rot} und Pixel $A_{Grün}$ ist also dieselbe.
⇨ Q-Values können diesen Zusammenhang aber nicht erlernen.

Lösungsidee

Wie wir bereits wissen, sind Q-Values nichts Weiteres als eine Funktion.

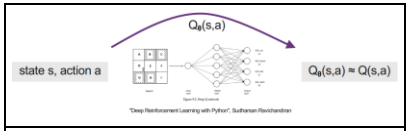
⇨ Die Funktion bildet States und Actions auf einen Reward ab.

Aus der Mathematik wissen wir, dass wir jede Funktion irgendwie approximieren können. Wieso also nicht auch diese?

Deep Reinforcement Learning

In Deep Reinforcement Learning kombinieren wir Methoden aus dem Deep Learning (wie ANNs) mit RL. Wir können so völlig neue Problemstellungen lösen.

⇨ Wir verwenden z.B. CNNs, um die Q-Values zu approximieren.



Da wir in diesem Fall die Q-Values mithilfe von Deep Learning approximieren, sprechen wir auch von einem sogenannten Deep Q-Network (DQN).

Continuous-Action Environments

Ausgangslage

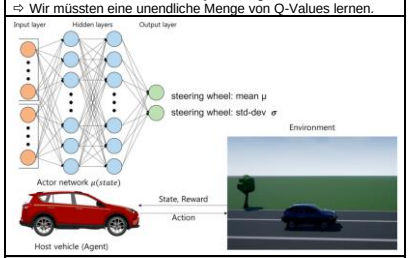
Bis anhin haben wir mit unseren Algorithmen jeweils die $V(s)$ / $Q(s,a)$ Werte gelernt und daraus die optimale Policy π^* abgeleitet. Wir wollen nun Algorithmen anschauen, welche direkt die Policy π verbessern.

⇨ Wir lernen also keine V-Values oder Q-Values mehr.
⇨ Somit können wir auch mit kontinuierlichen Aktionen arbeiten.

Continuous-Actions

In vielen Anwendungsfällen kann es vorkommen, dass ein Agent aus einer «unendlichen» Menge von Actions auswählen kann. Wir reden dann von sogenannten kontinuierlichen Aktionen.

⇨ z.B. Drehe das Lenkrad um 1, 0.1, 0.01, 0.001, ... Grad.
⇨ Wie würden wir das mit z.B. Q-Learning anstellen?
⇨ Wir müssten eine unendliche Menge von Q-Values lernen.

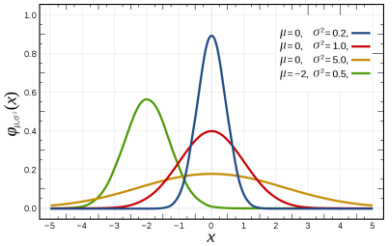


Gaussian Policy (GP)

Eine Gaussian Policy ist eine Policy, bei der die Actions A basierend auf einer Normalverteilung ausgewählt werden.

$$\pi = \begin{cases} S \times A \to [0,1] \\ (s,a) \mapsto P(a) \sim \mathcal{N}(\mu_s, \sigma_s) \end{cases}$$

⇨ Bei kontinuierlichen Aktionen nehmen wir eine GP an.
⇨ Lese: «Action a is sampled from a gaussian distribution»
⇨ Wir erkennen, dass jeder State S eigene Werte μ_s und σ_s hat.



⇨ Eine hohe Varianz σ bedeutet «explore», eine tiefe «exploit».
⇨ Erkennbar ist dies durch die «Spitze» der Verteilung.

Das Ziel unseres RL-Algorithmus ist es nun, die optimalen Werte μ und σ zu finden, welche einen bestimmten Reward maximieren. Wir schreiben daher auch:

$$\pi_\theta = \begin{cases} S \times A \to [0,1] \\ (s,a) \mapsto P(a) \sim \mathcal{N}(\mu_{s,\theta}, \sigma_{s,\theta}) \end{cases}$$

⇨ Wobei θ der trainierbare Parameter ist.
⇨ Oftmals können wir auch σ fixieren und nur μ lernen.

Policy Gradient

Die Policy Gradient Methode ermöglicht es uns nun, den optimalen Parameter θ zu finden. Sie verwendet dazu verschiedene mathematische Konzepte.

1. Objective Function

Die Objective Function J beschreibt den zu erwartenden totalen Reward einer Policy bei definiertem Parameter θ.

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[R(\tau)]$$

⇨ Diesen Wert wollen wir also maximieren.
⇨ Dazu müssen wir den optimalen Parameter θ finden.

2. Gradient Ascent

Wir können nun dank der $J(\theta)$ Funktion den optimalen Parameter θ mithilfe der «Gradient Ascent» Methode finden.

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

⇨ Achtung: «Gradient **Ascent**», nicht «Gradient **Descent**».
⇨ $J(\theta)$ ist ein Erwartungswert. Wie finden wir den Gradienten?
⇨ Wir verwenden dazu das «Policy Gradient Theorem».

3. Policy Gradient Theorem

Dieses Theorem erlaubt es uns, den Gradienten der $J(\theta)$ Funktion zu bestimmen, obwohl es sich dabei um einen Erwartungswert handelt. Es gilt:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)]$$

⇨ Die Herleitung davon ist für dieses Modul nicht relevant.

4. Monte-Carlo Approximation

Um nun die «Gradient Ascent» Methode verwenden zu können, müssen wir nun konkrete Werte für den Gradienten der $J(\theta)$ Funktion bestimmen.

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

Da $\nabla_\theta J(\theta)$ aber weiterhin Erwartungswerte beinhaltet, müssen wir diese irgendwie umwandeln. Wir verwenden dazu die Monte-Carlo Approximation.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \, R(\tau) \right]$$

Als Vergleich dazu:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)]$$

⇨ Die M.C.-Approximation findet konkrete Werte für $\nabla_\theta J(\theta)$, indem sie zufällig N Trajectories τ der Länge T auswertet.
⇨ Dies entspricht also wieder der Idee von «Trial-and-Error».

REINFORCE

Der «REINFORCE» Algorithmus gehört zu den einfachsten Policy Gradient Methoden. Er verwendet die Monte-Carlo Approximation.

Funktionsweise

1. Initialize the network parameter θ with random values
2. Generate N trajectories $\{\tau^i\}_{i=1}^N$ following the policy π_θ
3. Compute the return of the trajectory $R(\tau)$
4. Compute the gradients

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \, R(\tau) \right]$$

5. Update the network parameter as $\theta = \theta + \alpha \nabla_\theta J(\theta)$
6. Repeat *steps 2* to *5* for several iterations

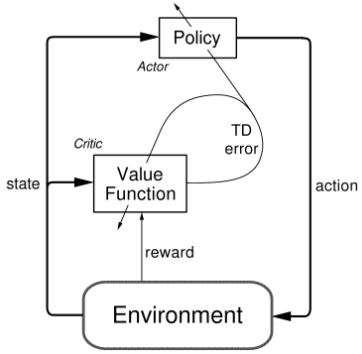
Actor-Critic

Ausgangslage

Bei Actor-Critic verbinden wir nun die Funktionsweise von $V(s)$ / $Q(s,a)$ Werten mit der Idee von Policy Gradient.

- Actor: Der Actor lernt eine optimale Policy, indem er fortlaufend Aktionen auswählt.
- Critic: Der Critic lernt die $V(s)$ / $Q(s,a)$ Werte und evaluiert die Aktionen des Actors, indem er den erhalten Reward mit dem erwartetem Reward vergleicht.

⇨ Wichtig: Im Gegensatz zu Policy Gradient wird der Actor fortlaufend verbessert und wertet nicht zuerst eine Trajectory aus.



Funktionsweise

1. Initialize the actor network parameter θ and the critic network parameter ϕ
2. For N number of episodes, repeat *step 3*
3. For each step in the episode, that is, for $t = 0, \ldots, T\text{-}1$:
   1. Select an action using the policy, $a_t \sim \pi_\theta(s_t)$
   2. Take the action a_t in the state s_t, observe the reward r, and move to the next state s_t'
   3. Compute the policy gradients:
   $$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(a_t|s_t)\left(r + \gamma V_\phi(s_t') - V_\phi(s_t)\right)$$
   4. Update the actor network parameter θ using gradient ascent:
   $$\theta = \theta + \alpha \nabla_\theta J(\theta)$$
   5. Compute the loss of the critic network:
   $$J(\phi) = r + \gamma V_\phi(s_t') - V_\phi(s_t)$$
   6. Compute gradients $\nabla_\phi J(\phi)$ and update the critic network parameter ϕ using gradient descent:
   $$\phi = \phi - \alpha \nabla_\phi J(\phi)$$

Berechnungen in Keras

Definitionen:

- In: Eingabedaten im Format (h, w, d)
- $\mathrm{P_T}$: Anzahl der trainierbaren Parameter
- Out: Ausgabedaten im Format (h, w, d)

Convolution

```
layers.Conv2D(
   filters = k, kernel_size = (n,m),
   strides = (u,v), activation='…')
```

$$\mathrm{P_T} = (n * m * d_{\mathrm{In}} + 1) * k$$

$$\mathrm{Out} = \left(\frac{h_{\mathrm{In}} - n}{u} + 1, \frac{w_{\mathrm{In}} - m}{v} + 1, k\right)$$

⇨ Standardwerte: $u = v = 1$

Max / Average Pooling

```
layers.MaxPooling2D(
   pool_size = (n,m), strides = (u,v))
```

$$\mathrm{P_T} = 0$$

$$\mathrm{Out} = \left(\frac{h_{\mathrm{In}} - n}{u} + 1, \frac{w_{\mathrm{In}} - m}{v} + 1, d_{\mathrm{In}}\right)$$

⇨ Standardwerte: $n = m = 2$, $u = n$, $v = m$
⇨ Gleiches gilt auch für `layers.AveragePooling2D(…)`

Flatten

```
layers.Flatten()
```

$$\mathrm{P_T} = 0$$

$$\mathrm{Out} = (h_{\mathrm{In}} * w_{\mathrm{In}} * d_{\mathrm{In}}, 1, 1)$$

Dense

```
layers.Dense( units = k )
```

$$\mathrm{P_T} = \mathrm{k} * (h_{\mathrm{In}} + 1)$$

$$\mathrm{Out} = (k, 1, 1)$$

Dense

```
layers.SimpleRNN( units = k )
```

$$\mathrm{P_T} = \mathrm{k} * (k + w_{\mathrm{In}} + 1)$$

$$\mathrm{Out} = (k, 1, 1)$$

⇨ Beachte: h_{In} = Anzahl Zeitschritte, w_{In} = Anzahl Features

Components of real-world AI projects

Data !
- not enough / too much data
- biased data
- need for data curation and "data automation"

Time/Budget constraints

Reproducibility (which model was trained with which data?)

Monitoring Is the "in-the-field" performance as good as expected?

Versioning (which model is deployed/updated on which product?)

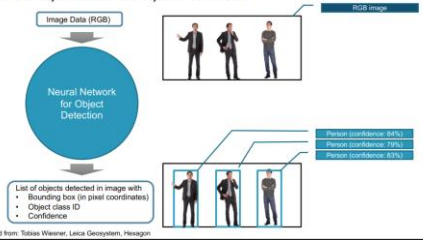
... etc.

CI / CD
Continuous Integration (build/test)
Continuous Delivery/ Deployment (main/stage/prod . Do we want this?)

➢ How do we "get things into production" ?

AI-based scene understanding

Neural networks for object detection and object re-identification



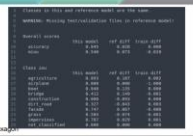
Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Training pipeline for Neural Networks

Post-processing and Testing

Model Evaluation
- Acceptance Testing and Performance Evaluation
- **Reproducibility**: versioning and documentation of neural networks, test suites and test results

Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring



Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Data Analysis & Curation (Mit lightly.ai)

- Consider camera data for training a "self driving car"

Projecting all images down to a 2D embedding:
- Data is clustered (e.g. day/night)
- More data collected during day
- Insights?
  - Collect data in "missing areas"
  - Balance Data / Remove redundant data (Car companies deal with Terabytes)

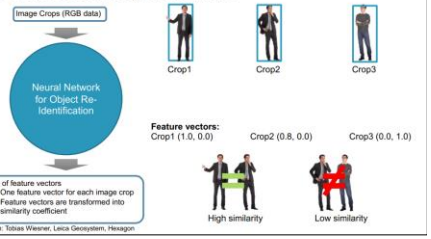
Learning Objectives

- Real-world machine-learning projects involve many different tasks and technologies. You learn the basic concepts of Machine Learning Operations (ML-OPS)

Figure: https://ml-ops.org

AI-based scene understanding

Neural networks for object detection and object re-identification



Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Training pipeline for Neural Networks

Deployment

Model Deployment
- Manage deployment and rollout of different networks in the market
- Track and document model versions in the market

Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring

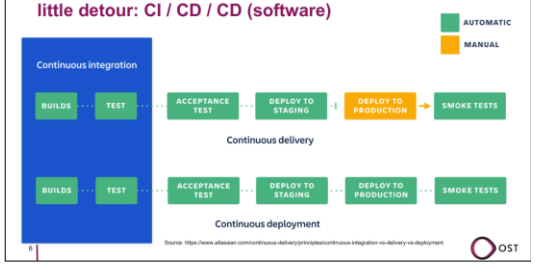
Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring

Lightly:
- Uses **self-supervised learning & active-learning**
- Learns **which images are similar** and thus, less relevant

As a result:
- ML models **get better** through better data
- Dataset size and costs decrease **by up to 90%**

little detour: CI / CD / CD (software)



Source: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Training pipeline for Neural Networks

Data Collection

Data collection
- Provided by customer via cloud
- Generated using computer models
- Collected by our company experts

Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring

ML data management:
- Data Ingestion
- Data storage
  - Versioning
  - Database structure
  - Metadata management
  - Documentation

Data Ingestion: Ein breiter Begriff, der das Erhalten oder Importieren von Daten für die Speicherung oder Verarbeitung beschreibt.

Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Training pipeline for Neural Networks

"after sales":
"in action", the ANN will perform differently!
When used in thousands of different context, even rare/unexpected event happen!

Model Monitoring
- Monitor performance (quality) of Neural Networks in the market
- Identify problematic cases where models fail
- Collect customer feedback and trigger Re-Training process

Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring

Rare / Unexpected Events: Wir haben z.B. unser Modell nur bei normalen Wetterbedingungen trainiert und getestet. In einem Gebiet mit vielen Stürmen funktioniert unser Produkt daher evtl. nicht.

Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

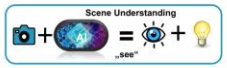
Data Versioning:

- https://dvc.org
  - Data on Git
  - metric tracking
  - reproducibility
  - Integrate into the ML Pipeline
  - ...

My lesson learned:
In practice, there are many unexpected problems! Do not underestimate the costs of a AI Project !

The goal: Scene understanding. Use AI to snalyse Sensordata. Enable new product features and workflows

Object Detection + Object Tracking + Object Re-Identification



AI on the edge enables **Autonomous Workflows**
Example on the video: Prism Carrier Tracking. The person carrying a prism is hidden behind by an obstacle. It is reidentified when re-appearing later.

Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Training pipeline for Neural Networks

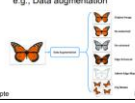
Prepare training data

Data preparation
- Data augmentation
- Data sampling (select subset of relevant data for re-training)

Data Analysis: Wie gut passen die Daten auf unser Problem? Haben wir zu wenige oder «biased» Daten? (s. «lightly.ai»)

Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring

Standard pre-processing:
e.g., Data augmentation



Data analysis (variance of information)

Slides adapte Hexagon

Take away messages

Data Engineering competences are essential for Machine Learning

Competences
Understand general workflows and data flows with all specific requirements

Data Engineers

Technical skillset:
Operate and maintain processing ressources, network infrastructure

DevOps

Data Analysts, ML Engineers

In-depth knowledge
Detailed knowledge of underlying algorithm and processes

Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

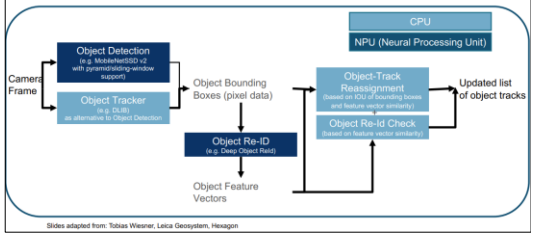
Learning Objectives

- Tensorflow.js is a javascript library that brings machine learning to the web browser. You learn how tow work with tensorflow.js. In particular, you will be able to:
  - define a model in tf.js
  - train and run (inference) a model in your browser
  - train a model using Keras, export the model, and load the model in tf.js

5/23/22 11:17 AM

AI-based scene understanding

Tracking by detection framework



Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Training pipeline for Neural Networks

Training process

Training & Optimization
- Training infrastructure
- Consistent versioning of input and output
- Model optimization (size, inference time)

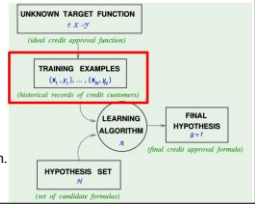
Data Collection | Data Preparation | Model (Re-) Training | Model Evaluation | Model Deployment | Model Monitoring

Slides adapted from: Tobias Wiesner, Leica Geosystem, Hexagon

Possible problems with data

- biased data
  (when the training samples are not representative of "all" data. see next slide)
- too many similar/**redundant** samples
  (a road with a car in front of us)
- **missing** the (interesting) **edge cases**
  (a pedestrian coming "out of nothing")
- bad Data **quality**
  (camera picts on a rainy day)

Biased Data: Wir haben z.B. eine AI, die Strassenschilder erkennt. Wir trainieren sie aber nur mit europäischen Strassenschildern. Wir haben somit mit «Biased Data».



ML in the web browser, a few comments:

- Browsers provide a platform-independent runtime. That means, you can deploy your machine learning model virtually anywhere (mobile, edge, ...).
- Performance is not sooo bad. WebGL can access a local GPU.
- The web client can easily be updated when the models are hosted and update on the web.
- Training on the web? It's advertised by some. I am not sure... depends on the use case (for example: adjust a pretrained model to the local parameters: camera, user gestures, ...)
- Can also be used server-side (Node.js)
- Remember: no python -> no numpy and other python libraries!

5/23/22 11:17 AM

Use case: transfer learning (tutorial. in class)

- Transfer Learning: https://codelabs.developers.google.com/tensorflowjs-transfer-learning-teachable-machine#0
- **What exactly is transfer learning?**

https://codelabs.developers.google.com/tensorflowjs-transfer-learning-teachable-machine#2

- This idea can be generalized:
  - Train a model on ***some*** task -> Learn useful features.
    Example: Image - Autoencoder to learn visual features
  - Then train a model on a different task using those features.
- **Transfer learning is particularly attractive when:**
  - **You can learn useful features using an unsupervised/self-supervised task**
  - **You have only a few labels/data for your actual task**

Use case: export / import your own model

- during exercise session:
  - Define and train a model on a server backend (well. today you use your laptop)
  - Export the model (for example in JSON)
  - Import the model in a client (e.g. a tensorflow.js web application)

Saving a Keras model:

```
model = ...  # Get model (Sequential, Functional Model, or Model subclass)
model.save('path/to/location')
```

Loading the model back:

```
from tensorflow import keras
model = keras.models.load_model('path/to/location')
```

```
const model = tf.sequential({
  layers: [tf.layers.dense({ inputShape: [1], units: 1 })]
});

model.compile({ optimizer: "sgd", loss: "meanSquaredError" });

const xs = tf.tensor([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0]);
const ys = tf.tensor([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0]);

await model.fit(xs, ys, { epochs: 1000 });

model.predict(tf.tensor([10.0])).print();
```

RNN

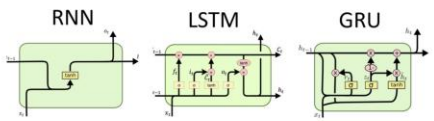
- RNN is a neural network that consists of a hidden state h and an (optional) output y operating on a variable-length sequence $\mathbf{x} = (x_1, \ldots, x_{T_x})$. At each time step t, the hidden state h_t is updated by $h_t = f(h_{t-1}, x_t)$
- f is a non-linear activation function. It may be as simple as an element-wise logistic sigmoid function (SimpleRNN) and even complexer a long short-term memory (LSTM)
- An RNN can learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence.
  - The output at each timestep t is the conditional probability $p(x_t \mid x_{t-1}, \ldots, x_1)$.

$$p(x_{t,j} = 1 \mid x_{t-1}, \ldots, x_1) = \frac{\exp(\mathbf{w}_j \mathbf{h}_{[t]})}{\sum_{j'=1}^{K} \exp(\mathbf{w}_{j'} \mathbf{h}_{[t]})},$$

Dense layer with softmax

Long term dependencies

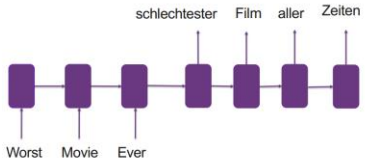
- LSTM and GRU improved control of information flow
- Adding more complexity (forget gate, reset gate, update gate , cell state etc.)



Application of RNN

Applications

- Many-to-Many : Machine Translation



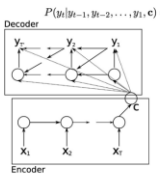
Simple Seq2Seq

Cho'2014

Denke: Die Wahrscheinlichkeit für «Hallo Welt», wenn zuvor «Hello World» kam.

- Proposal: A novel neural network architecture that learns to *encode* a variable-length sequence into a fixed-length vector representation and to *decode* a given fixed-length vector representation back into a variable-length sequence.
- From a probabilistic perspective, it is a general method to learn the conditional distribution over a variable-length sequence conditioned on another sequence $p\left(y_{1}, \ldots, y_{T_y} \mid x_{1}, \ldots, x_{T_x}\right)$, where T_y and T_x can be different.

$$P(y_t|y_{t-1}, y_{t-2}, \ldots, y_1, \mathbf{c}) = g\left(\mathbf{h}_{\langle t \rangle}, y_{t-1}, \mathbf{c}\right)$$



$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, y_{t-1}, \mathbf{c}\right)$$

$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, x_t\right),$$

Cho, 2014
Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Simple Seq2Seq

Cho'2014

Denke: Encoder h_t, Decoder s_t

The decoder is RNN which is trained to *generate* the output sequence by predicting the next symbol y_t given the hidden state h_t. Both y_t and h_t are conditioned on y_{t-1} and summary c of the input sequence. The hidden state of the decoder at time t is computed by,

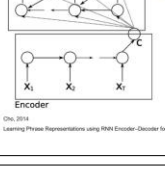
$$s_t = f(s_{t-1}, y_{t-1}, c)$$

The conditional probability of the next symbol is

$$P(y_t|y_{t-1}, y_{t-2}, \ldots, y_1, c) = g(s_t, y_{t-1}, c).$$

For given activation functions f and g (the latter must produce valid probabilities, e.g. with a softmax).

$$P(y_t|y_{t-1}, y_{t-2}, \ldots, y_1, c) = g(s_t, y_{t-1}, c)$$



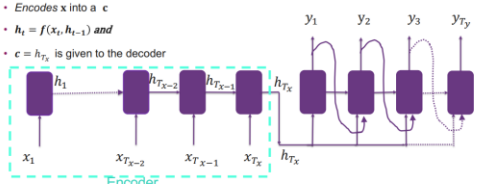
$$s_t = f(s_{t-1}, y_{t-1}, c)$$

$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, x_t\right),$$

Cho, 2014
Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

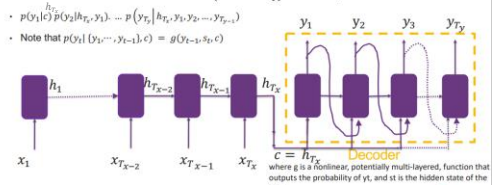
Basic/Simple encoder-decoder Unrolled

- An encoder reads the input sequence $\mathbf{x} = (x_1, x_2, .., x_n)$
- *Encodes* **x** into a **c**
- $\boldsymbol{h_t = f(x_t, h_{t-1})}$ ***and***
- $\boldsymbol{c = h_{T_x}}$ is given to the decoder



Basic/Simple encoder-decoder Unrolled

- *The probability over the translation* $\mathbf{y} = \{y_1, y_2, .., y_{T_y}\}$ and $\mathbf{c} = \mathbf{h}_{T_x}$ *is multiplication of*
  - $p(y_1|\mathbf{c})\, p(y_2|h_{T_x}, y_1), \ldots\, p\left(y_{T_y}|h_{T_x}, y_1, y_2, \ldots, y_{T_{y-1}}\right)$
  - Note that $p(y_t|\{y_1, \cdots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$



where g is a nonlinear, potentially multi-layered, function that outputs the probability of yt, and st is the hidden state of the RNN.

Training Seq2Seq

- The two components of the *RNN Encoder–Decoder* are jointly trained to maximize the conditional log-likelihood
- $\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_{\theta}(\mathbf{y}_n|\mathbf{x}_n)$
- where θ is the set of the model parameters and each $(\mathbf{x}_n, \mathbf{y}_n)$ is an (input sequence, output sequence) pair from the training set.
- As the output of the decoder, starting from the input, is differentiable, a gradient-based algorithm can be used to estimate the model parameters.

Limitations of single encoding $c = h_{T_x}$

- Encoding Bottleneck
- Sequential -> slow
- Long term dependencies
- Loss of information

DreamLand

- Continous stream to decoder
- Parallalizable (not one input sequentially at a time)
- Long sequences / memory

ATTENTION!

- The author Kenneth Anderson hailing from a Scottish family settled in India and his love for the denizens of the XXXX jungle led him to game XXXX and XXXX real-life adventure stories.
- When predicting a token, if not all the input tokens are relevant, the model should attends only to parts of the input sequence that are relevant to the current prediction. This is achieved by treating the context variable as an output of attention.
- Attend to the most important parts of the input
  - Identify the important parts
  - h_{T_x}achieves that to a certain extent but it summatrizes the context across all timesteps..
  - As we said, the RNN and even LSTMs/GRUs will forget the word "author" given a very long sequences.

Bahdanau et al (2015)

How about using all the hidden states $h_1, h_2, \ldots, h_{T_x}$

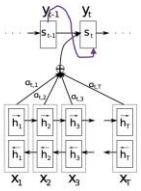
- So far we had,

$$p\left(y_i \mid \mathbf{x}, y_1, y_2, \ldots, y_{i-1}\right) = p\left(y_i \mid \boldsymbol{c} = \boldsymbol{h}_{T_x}, y_1, y_2, \ldots, y_{i-1}\right) = g(h_i, y_{t-1}, c)$$

Bahdanau et al (2015) came up with a simple but elegant idea.
They suggested that not only can all the input words be taken into account in the context vector, **but relative importance** should also be given to each one of them.
Whenever the model generates a sentence, it searches for a set of positions in the encoder hidden states where the most relevant information is available. This idea is called 'Attention'.

$$p(y_i|\mathbf{x}, y_1, y_2, y_3, \ldots, y_{i-1}) = p\left(y_i|\, \boldsymbol{c_i}, y_1, y_2, \ldots, y_{i-1}\right) = g(c_i, y_{i-1}, h_i)$$

Bahdanau2014



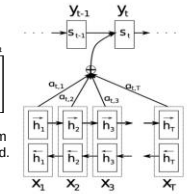
- $p(y_t|\{y_1, \cdots, y_{t-1}\}, \mathbf{x}) = g(y_{t-1}, s_t, c_t)$
- We see that $s_t = f(s_{t-1}, y_{t-1}, c_t)$
- Well, what is c_t?
- c_t makes the model give particular ATTENTION to certain hidden states when decoding each word.
- It lets the model itself 'learn' which words to give ATTENTION to and which ones to ignore during translation of each word at the decoder.

ATTENTION

- $c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$

$$\begin{bmatrix}\alpha_i\\0.7\\0.2\\\vdots\\0.1\end{bmatrix} \Rightarrow c_i = 0.7 * \begin{bmatrix}h_1\\4\\2\\\vdots\\5\end{bmatrix} + 0.2 * \begin{bmatrix}h_2\\2\\6\\\vdots\\1\end{bmatrix} + \cdots + 0.1 * \begin{bmatrix}h_n\\5\\1\\\vdots\\7\end{bmatrix}$$

Dank α_i weiss ich, welche «Hidden States» vom Encoder für die Berechnung von y_i relevant sind.



- How do we compute α_{ij}? Using a softmax
- $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$ and $\mathrm{e}_{ij} = \mathrm{a}(\mathrm{s}_{i-1}, \mathrm{h}_j)$
- a is an *alignment model* which scores how well the inputs around position *j* and the output at position i match.
- The score is based on the RNN hidden state s_{i-1} (just before emitting y_i) and h_j of the input sentence.

The probability α_{ij}, or its associated energy e_{ij}, reflects the importance of the annotation $\boldsymbol{h_j}$ with respect to the previous hidden state $\boldsymbol{s_{i-1}}$ in deciding the next state $\boldsymbol{s_i}$ and generating $\boldsymbol{y_i}$.

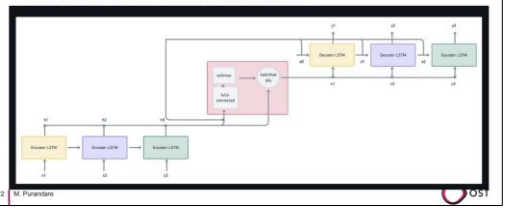
- Intuitively, this implements a mechanism of ATTENTION in the decoder. The decoder decides parts of the source sentence to pay ATTENTION to.

Step by step attention

- Firstly, the input sequence x1,x2,x3 is given to the encoder LSTM. The vectors h1,h2,h3 are computed by the encoders from the given input sequence. These vectors are the inputs given to the attention mechanism. This is followed by the decoder inputting the first state vector s0, which is also given as an input to the attention mechanism. We now have s0 and h1,h2,h3 as inputs.
- The attention mechanism mode (depicted in a red box) accepts the inputs and passes them through a fully-connected network and a softmax activation function, which generates the "attention weights".
- The weighted sum of the encoder's output vectors is then computed, resulting in a context vector c1. Here, the vectors are scaled according to the attention weights.
- It's now the decoder's job to process the state and context vectors to generate the output vector y1.
- The decoder also produces the consequent state vector s1, which is again given to the attention mechanism model along with the encoder's outputs.
- This produces the weighted sum, resulting in the context vector c2.
- This process continues until all the decoders have generated the output vectors y1,y2,y3

What is e_{ij}

- $\boldsymbol{e_{ij}}$ is the output score of a feedforward neural network described by the function **a** that attempts to capture the alignment between input at **j** and output at **i**.

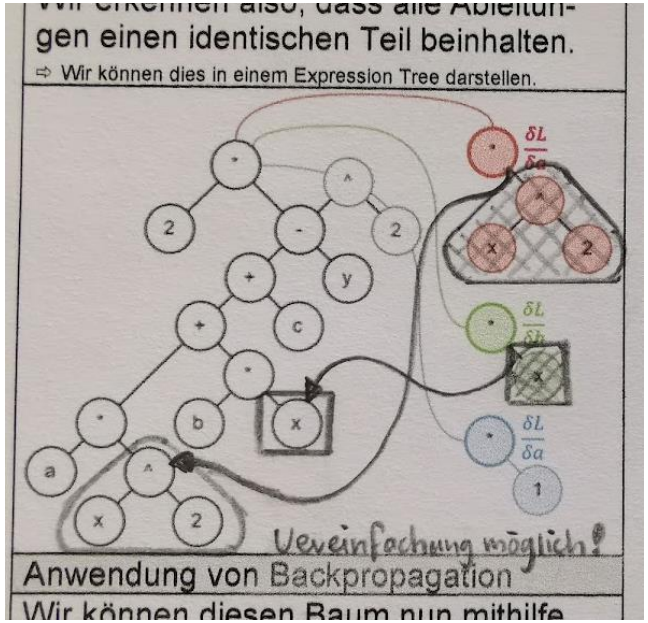


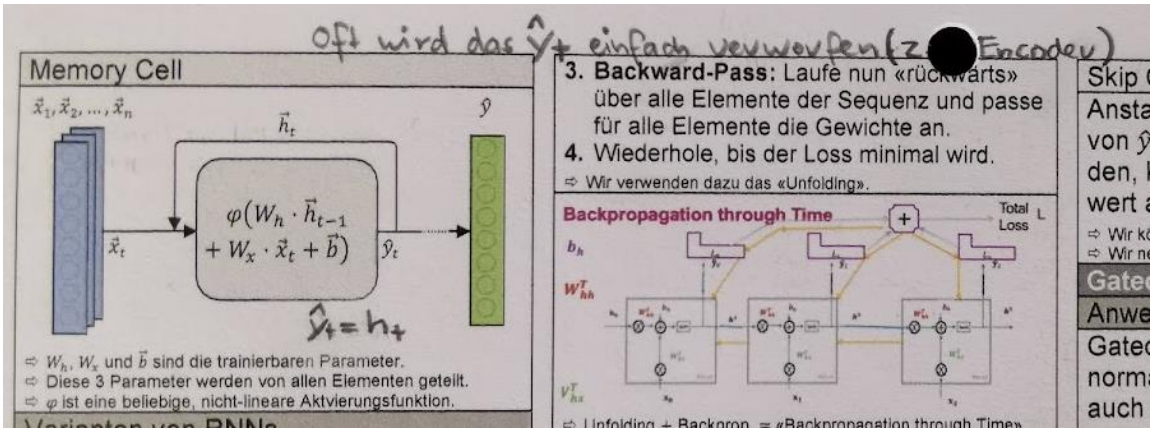
Global and local attention

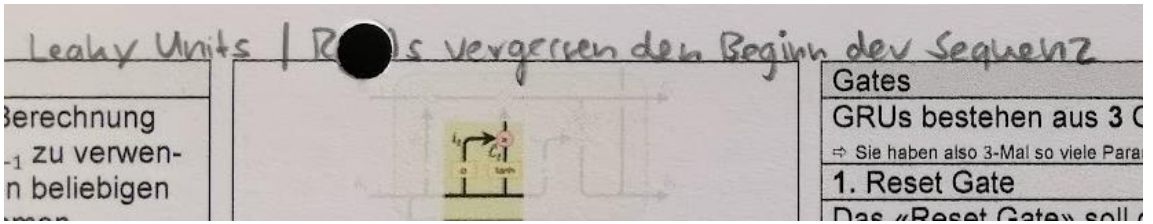
- **Effective Approaches to Attention-based Neural Machine Translation**
  - **Minh-Thang Luong Hieu Pham Christopher D. Manning**
- This paper examines two simple and effective classes of attention mechanism:
  - a *global* approach which always attends to all source words and
  - a *local* one that only looks at a subset of source words at a time.

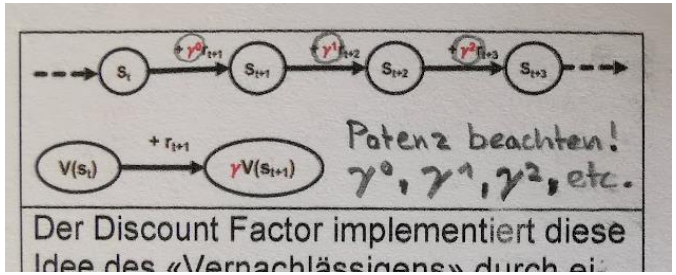
Denke: Global = Gesamtes Buch
Lokal = Aktueller Abschnitt

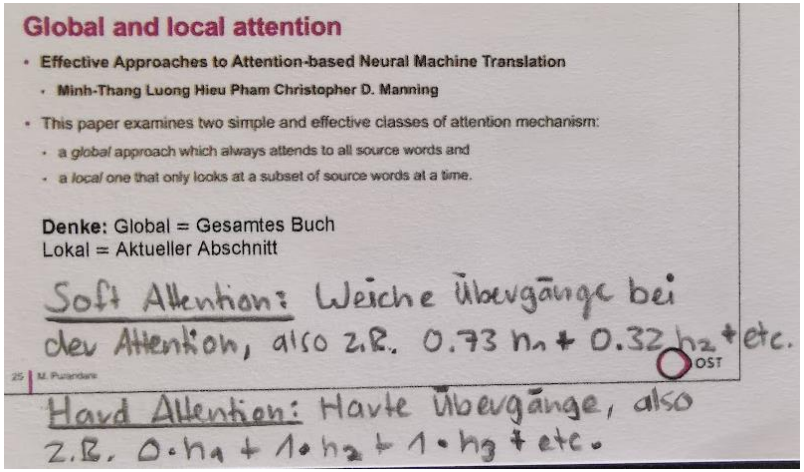
Anhang / Nachtrag

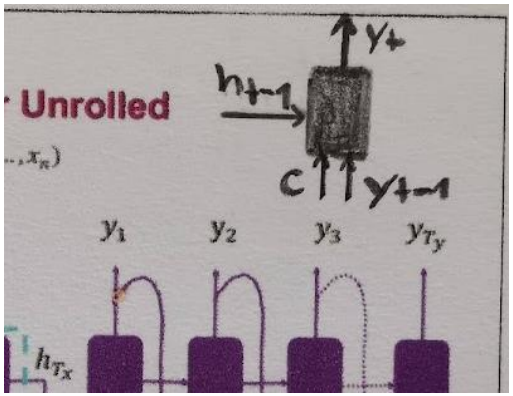


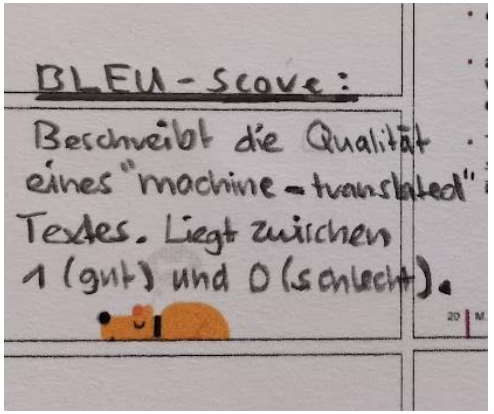












$P_T = k * (h_{In} + 1)$

$Out = (k, 1, 1)$

~~Dense~~ RNN

`layers.SimpleRNN( units = k )`

$P_T = k * (k + w_{In} + 1)$ LSTM = $P_T \cdot 4$

$Out = (k, 1, 1)$ GRU = $P_T \cdot 3$ (bias 2x)

⇨ Beachte: h_{In} = Anzahl Zeitschritte, w_{In} = Anzahl Features

- Ungerade Zahlen immer abrunden.
- Höhe und Breite eines Bildes haben keinen Einfluss auf die P_T der Conv2D-Layer, dafür aber auf die Dence-Layer.

- Model-based RL: Der Agent hat Zugriff auf das "World-Model", also die Parameter T und R, entweder durch lernen oder einfach so. Der Agent kann also "planen".

Model-free RL: Der Agent hat keinen Zugriff auf das "Model". Er muss also z.B. die Q-Values lernen. T und R werden nicht gelernt.

Experience Replay: Beschreibt den Prozess des Abspeicherns der Erfahrungen im Replay Buffer um dann mit diesen Werten zu lernen.

Keras multi layer CNN

```
model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

Rescaling (scale, offset=0.0, **kwargs):
- Rescales input vlaues to a new range

Conv2D (filters, kernel_size, padding, activation):
- Convolution slides over a 2D "surface"
- Image is 3D 3rd demension colors RGB
- **filters**: number of filters that convolutional layers will learn from. Also defines number of output filters.
- **kernel_size**: specifying height and width of the 2D convolution window.
- **strides**: specifies "step" of convolution & height/width of the input volume
- **padding**: `"same"` = no padding, `"valid"` = padding with zeros
- **activation**: Activation function to use.

MaxPooling2D (pool_size, strides, padding):
- Downsamples input along ist spatial dimensions by taking maximum value over an input window for each channel of input.
- Window shifted by `"strides"` along dimesion
- **pool_size**: win. size over which to take max

Flatten:
- Flattens input
- Does not affect the batch size

Dense (units, activation):
- Reduziert parameter
- Implements operation: `output = activation(dot(input, kernel) + bias)`

- **Saturating:**
- tahn activation function
- Even for very small or very large input, hte ouput range is within [-1, 1]

- **Non-Saturating:**
- Relu

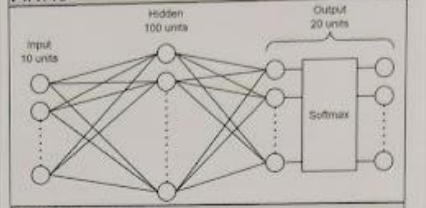
For large input, the output is large (unbounded)

A3C: Mehrere "Worker" helfen dabei, das Netzwerk zu verbessern. Dies erlaubt uns, das Training besser zu parallelisieren.

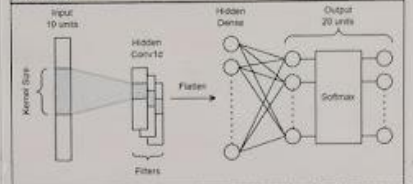
Window of Intevest: Beschreibt das "Ansichtsfenster" beim Pooling. Die Größe entspricht der "Pool-Size".

Value has short term memory span of 10 days. The valet should predict which dress to make ready the next day. Master has 20 dresses in total. These are two examples using ANNs and CNNS.

ANNs



CNNs



DQN (Deep Q Network)

Approximieren von Q-Value für alle Actions gegeben vom Input State. DQN nutzen, um Regression zu machen.

Replay Buffer

- Speichert Erfahrung vom Agent über einige Episoden als Tupels (S,A,R,S')
- Tupels werden nach und nach hinzugefügt beim Iterieren mit der Umgebung

1. Initialize Replay Buffer *D*
2. For each episode perform:
   a. Make transition, that is perform an action a in the state *s*, move to next state *s'*, and receive reward *r*
   b. Store transition information in relay buffer *D*

Loss Function

- MSE als Loss für Regression
- Netzwerk wird trainiert, indem MSE zwischen «Target» Q-Value und «Predicted» Q-Value minimiert wird
- Substituting equation in precending equation

$Q * (s, a) = r + ymaxQ * (s', a') - Q_0(s, a)$

Target Network

- Soll das Training stabiler machen
- Kopie von Neural Network wird behalten und genutzt für $Q(s', a')$
- Parameter des Target Networks werden **nicht** trainiert, aber periodisch synchronisiert mit denen des Haupt-Q-Networks