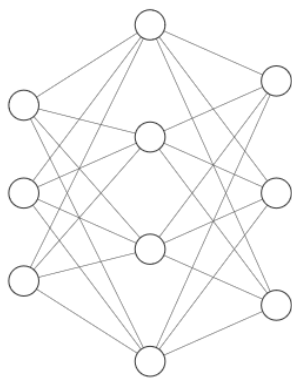


# Programmieren von Spiele-KI mit neuronalen Netzen



**VS**

O		X
X	X	O
O		

Livio D'Agostini

Maturaarbeit

Betreuung: Joachim Kambor

Kantonsschule Zug

2019

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Theorie Teil 1: Künstliche Neuronale Netze</b>	<b>3</b>
2.1	Ziel des <i>machine learning</i> im Allgemeinen . . . . .	3
2.2	Ein Netz definieren . . . . .	5
2.3	Fehlerrückführung - Backpropagation . . . . .	8
<b>3</b>	<b>Theorie Teil 2: Deep Reinforcement Learning</b>	<b>10</b>
3.1	Markov Decision Process (MDP) . . . . .	10
3.2	Q-Learning . . . . .	11
3.3	Tabulares Q-Learning (=tQL) . . . . .	12
3.4	Deep Q-Learning . . . . .	13
<b>4</b>	<b>Methoden</b>	<b>14</b>
4.1	Tic-Tac-Toe . . . . .	14
4.2	Tabulares Q-Learning . . . . .	15
4.3	Deep Q-Learning . . . . .	16
<b>5</b>	<b>Resultate</b>	<b>18</b>
5.1	Vergrößerung auf 4x4 . . . . .	18
5.2	Verbesserungsvorschläge . . . . .	19
<b>6</b>	<b>Fazit</b>	<b>20</b>

# 1 Einleitung

In den letzten fünf bis zehn Jahren hat sich in einem Unterbereich der Informatik, dem der künstlichen Intelligenz, viel verändert. Der heute erreichte Forschungsstand ermöglicht Resultate, die erste Anwendungen in der Industrie haben, zum Beispiel in der Bilderkennung, der Musik- und Spracherkennung oder dem autonomen Fahren. [14]

An dieser Stelle muss erwähnt werden, dass sich nicht das ganze Gebiet namens KI weiterentwickelt hat, sondern in der Tat nur ein kleiner Teil namens *machine learning* und in diesem wiederum nur der Teil des *deep learning*. Deep learning, zu welchem die mathematischen Grundlagen im nächsten Kapitel beschrieben werden, ist der Grund hinter der ganzen medialen wie wissenschaftlichen Aufregung, die sich seit 2012 eingestellt hat. [20]

Es sollte deshalb nicht verwundern, dass man sich für dieses Thema interessieren kann und die Maturaarbeit deshalb im Bereich des deep learning schreiben will. Meine Zielsetzung war es, eine kleine auf deep learning basierende KI schreiben, die ein einfaches Spiel spielen kann. Für Letzteres bot sich Tic-Tac-Toe an (welches in meiner Arbeit und meinem Code immer wieder als TTT abgekürzt wird). Wie sich während der Arbeitsphase bald zeigte, reichen die Methoden des deep learning noch nicht aus, um eine Spiele-KI zu machen, es wird zusätzlich noch reinforcement learning bzw. deep reinforcement learning benötigt. Beide Bereiche verfügen über eine solche immense Breite von verschiedenen Methoden, so dass eine vollständige Behandlung der Themen unmöglich war. In der Tat wird in dieser Arbeit im Bereich des deep learning wie reinforcement learning jeweils nur auf die einfacheren Methoden eingegangen. Es zeigt sich jedoch, dass diese für ein Spiel wie TTT ausreichen.

Informationen zum Thema habe ich zum grössten Teil im Internet gefunden. Es ist allerdings auch ein Buch zu nennen: *Neuronale Netze selbst programmieren* von Tariq Rashid [17] hatte ich gelesen, bevor ich mich zu dieser Arbeit entschloss. Es konnte mich überzeugen, dass deep learning keine Hexerei darstellt und dass eine Arbeit in diesem Bereich durchaus machbar ist.

## 2 Theorie Teil 1: Künstliche Neuronale Netze

Künstliche neuronale Netze sind ein zentraler Bestandteil dieser Arbeit. Es ist deshalb unerlässlich, sich mit ihnen genauer auseinanderzusetzen. In diesem Kapitel werden die grundlegenden Konzepte hinter diesen Netzen erklärt, allerdings reichen diese noch nicht aus, um eine Spiele-KI zu programmieren, was erst im zweiten Teil der Theorie gezeigt wird.

### 2.1 Ziel des *machine learning* im Allgemeinen

Heutzutage gibt es viele Bereiche in der Wissenschaft, wo der Computer zur Anwendung kommt: Als Simulationswerkzeug in der Physik, der Biologie, der Chemie oder der Geologie, als Sammler, Speicherer und Auswerter von Datenmengen in der Statistik und in den Sozialwissenschaften. Auffallend ist, dass die Programmierer dabei immer bis ins Detail wissen, was genau sie implementieren müssen: Der Physiker zum Beispiel kennt die Formeln, die seine Simulation benötigt, in und auswendig.

In unserer Welt existieren aber auch Problemstellungen, zu welchen selbst den schlauesten Köpfen keinen Algorithmus und keine Formel einfällt. Einer der Klassiker ist: Der Computer soll sich ein Bild ansehen und es einer Klasse wie Katze, Flugzeug oder Fussgängerstreifen zuordnen.

Genau diese interessanten Probleme will das machine learning lösen: Solche, bei denen die Menschheit die Lösung noch nicht kennt.

Neuronale Netze muss man, gleich wie Algorithmen, als mathematische Funktion auffassen. Das bedeutet, dass sie ein Etwas in ein anderes Etwas überführen, zum Beispiel ein  $x$  in ein  $y$  nach der Regel einer Funktion:

$$y = f(x) | z.B. : f(x) = x^2$$

$x$  ist so etwas wie ein Input,  $y$  gleicht analog einem Output. Dieses  $f$  ist das, was wir suchen. Es ist der Zusammenhang zwischen Input und Output. Nach obiger Problemstellung:

$$Bildklasse = unbekannteFunktion(Bildchen)$$

Wie gesagt soll das unbekannte  $f$  gefunden werden. Machine learning löst dieses Problem mit folgendem Ablauf:

1. Wir suchen eine allgemeine Funktion (das *neural network*), und setzen es für die Funktion ein, die wir nicht kennen, obwohl die beiden Funktionen noch Nichts miteinander zu tun haben. Das neurale Netz enthält anpassbare Parameter, die es uns ermöglichen, jede beliebige Funktion zu modellieren.
2. Mit einem grossem Satz von Datensets gehen wir nun ans Werk: Jeder Schritt besteht aus einem Input zum Netz, wobei die Funktion ausgerechnet wird. An-

schliessend vergleicht man die Ausgabe mit einer Musterlösung, um zu erkennen, wie richtig das Netz momentan ist. Mithilfe von einer Fehlerfunktion und des Backpropagation-Algorithmus lassen sich die Parameter des Netzes ein wenig verändern, so dass diese in jedem Fall einen Schritt in die richtige Richtung tun. Wenn das Netz mit immer mehr Beispielen angepasst wird, dann wird es sich immer genauer an die unbekannte Funktion annähern. Schlussendlich hoffen wir, dass der Unterschied zwischen der unbekannten Funktion und unseres networks, unserer Approximation, so klein wie möglich wird: In diesem Fall IST das Neuronale Netz gleich der unbekannten Funktion, und so haben wir diese gefunden!

3. Nun kann man das trainierte Netz für Anwendungszwecke nutzen.

Im Folgenden werden die einzelnen Schritte genauer beschrieben.

## 2.2 Ein Netz definieren

Es gibt zwei Punkte, die eine beliebige Funktion erfüllen muss, so dass sie sich für machine learning eignet:

- Sie muss so viele verschiedene Funktionen annähern können, wie nur irgendwie möglich ist. Deshalb muss sie anpassbare Parameter enthalten, durch welche sie sich verändern kann.
- Es muss klar sein, wie die oben genannten Parameter angepasst werden können. Die einzige Informationsquelle, auf welche eine Update-Regel zurückgreifen darf, sind die Daten, mit denen das Netz trainiert wird; Das sind viele Paare der Art (Input, korrekter Output).

Neuronale Netze jeglicher Ausprägung erfüllen diese Punkte. Um das Vorgehen im *machine learning* zu erklären, wird hier nur das sogenannte MLP/dense-layer/fully-connected-neural network beschrieben. Es ist das am wenigsten Komplizierte der Netze. Es ist im Wesentlichen aus miteinander verbundenen Perzeptronen aufgebaut; Dabei handelt es sich um eine Abstrahierung der echten biologischen Neuronen[22]. Ein künstliches Neuron, auch Perzeptron genannt, sieht folgendermassen aus:

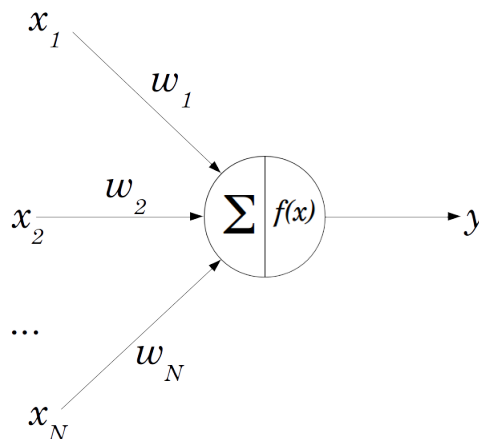


Abbildung 2: Ein Perzeptron[11]

Ein Perzeptron darf beliebig viele Inputs haben, das heisst beliebig viele Signale dürfen eintreffen. Alle Inputs sind gewichtet: Jeder Input-Wert wird mit einem Gewicht-Wert multipliziert. Das Gewicht ist ein Wert in  $\mathbb{R}$ . Diese Massnahme simuliert die Signalstärke des sendenden Neurons. Gleichzeitig erzeugt dies die Möglichkeit, dass ein Signal dämpfend/inhibitorisch wirken kann. Alle gewichteten Inputs werden zusammengezählt und bilden damit das Signal. Dieses wird an eine Aktivierungsfunktion weitergegeben. Diese simuliert, nach dem Vorbild eines echten Neurons, die Entscheidung, ob das Perzeptron das Signal weitergibt oder nicht. Die Ausgabe dieser wird zum Output des Perzeptrons. Die Aktivierungsfunktion ist aber nur selten eine

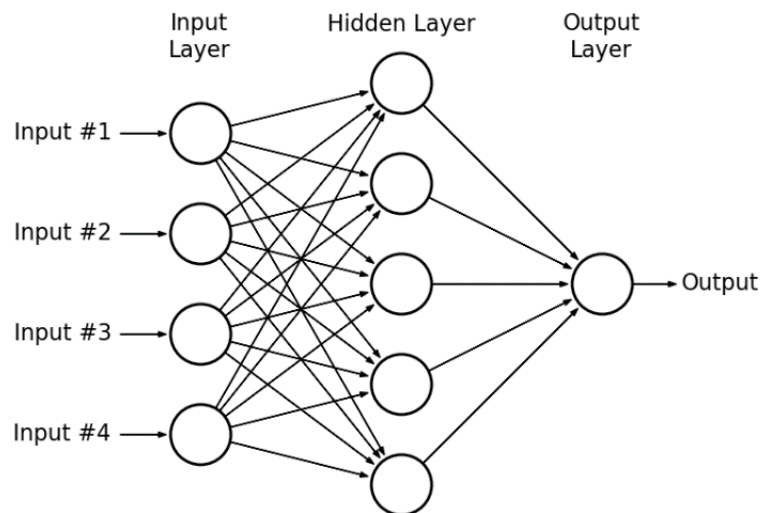


Abbildung 3: Ein MLP[24]

Stufenfunktion, so wie sie nach dem biologischen Vorbild sein sollte. Nun kann man mehrere dieser Perzeptrons hintereinander oder nebeneinander stellen. In den meisten Fällen sind die Perzeptronen schichtenhaft angeordnet, wobei alle Perzeptronen der vorherigen mit denen der nachfolgenden Schicht verbunden sind. Die erste Schicht nimmt den Input des Netzes an, die Letzte repräsentiert die Ausgabe. Dadurch entstehen MLPs (multilayer-perceptrons), wie in Abbildung 3 dargestellt. Natürlich steht es jedem offen, sein Netzwerk von Perzeptronen (innerhalb von Netzen meistens *Knoten* genannt, wenn überhaupt) auf beliebig komplizierte Art zu modellieren, jedoch ist es fraglich, wie viel besser ein spezielles Netz sein würde verglichen mit einem Normalen, welches einem Schichtenaufbau folgt. Zwischen den Schichten liegen die Gewichte, je eines für jede Verbindung. Als Aktivierungsfunktionen wurden historisch Sigmoid- und Tangens(hyperbolicus)funktionen verwendet[6], welche eine Art S-Form haben. Der heutige Standard heisst ReLU und ist wie folgt definiert:

$$ReLU(x) = \max(0, x)$$

D.h., alle negativen Werte werden auf null gesetzt, alle andern bleiben gleich. Weshalb ReLU so gut funktioniert, ist nicht vollständig geklärt, jedoch hat sich ReLU in der Praxis durchgesetzt. Intuitiv darf man sich vorstellen, dass bestimmte Knoten im MLP einen ganz bestimmten Sinn relativ zum ganzen Netz besitzen, z.B. ein Netz, dass Bilder zuordnen soll, muss evtl. an einer Stelle entscheiden können, ob ein bestimmtes Merkmal im Input vorhanden ist. ReLU, so stellt man sich vor, hilft, Werte, die zu einem bestimmten Merkmal korrespondieren, auf null zu setzen, wenn das Merkmal in einem bestimmten Bild nicht vorhanden ist[18]. Ein weiterer Vorteil des Schichtenaufbaus liegt darin, dass alle Gewichte in eine Matrix geschoben werden können. Zum Beispiel liegen alle Gewichte, die für Verbindungen zum ersten Knoten zuständig sind, in der ersten Zeile der Matrix. Hat man jetzt einen Vektor von Inputs,

so lassen sich die Inputs für Knoten der zweiten Schicht mit einer Matrixmultiplikation berechnen:

$$in2 = \text{dotp}([weights], in1)$$

Dank dieser Formel ist erkennbar, dass Neuronale Netze nicht unbedingt als Netze oder als Systeme von Verbindungen gesehen werden müssen. Eigentlich wird ein Vektor weitergegeben, der auf dem Weg wiederholt mit einer Matrix von Gewichten multipliziert und elementweise einer Aktivierungsfunktion übergeben wird. Die Anfangs beschriebenen Perzeptronen sind nicht mehr sichtbar.

Die Architektur eines Netzes definiert den *forward pass*. Damit ist das Berechnen der Ausgabe abhängig einer Eingabe gemeint. Im Fall vom obigen MLP:

$$\text{forward\_pass}(in0) = \text{ReLU}(\text{dotp}([weights1], \text{ReLU}(\text{dotp}([weights0], in0))))$$

Auf den ersten Input  $in0$  wird nach Konvention meistens keine Aktivierungsfunktion angewendet[17]. Die Gewichte sind entscheidend: Sie sind die anpassbaren Parameter. Der totale Zustand der Gewichte entscheidet über das Verhalten des Netzes. In der Theorie ist bewiesen, dass ein MLP von genügender Grösse und richtig gewählten Gewichten jede erdenkliche (kontinuierliche) Funktion annähern kann[12]. Natürlich weiss man nicht, was die korrekten Wahlen für die Gewichte sind. Um trotzdem Gewichte sinnvoll setzen zu können, hat man Backpropagation (im Deutschen als Fehlerückführung bekannt) erfunden. Benötigt werden viele Datenpaare mit der Form:

$$(Input, \text{verlangterOutput})$$

Zuerst wird das NN initialisiert, das heisst alle Gewichte erhalten einen zufälligen Wert. Nun werden die Gewichte schrittweise verbessert, in jedem Schritt durchläuft man die folgende Prozedur: Man nimmt ein Datenpaar und macht einen forward pass mit dem Input. Das erzeugt eine Ausgabe, die insbesondere zu Beginn des Trainings weit vom verlangten Output liegt. Als nächstes benötigt man eine Methode, die uns sagt, wie falsch die Ausgabe verglichen mit dem Verlangten ist und abgeleitet davon wie unrichtig das NN ist. Die Funktion, die dies tut, heisst Fehlerfunktion oder *loss function*. Meistens benutzt man den L1 oder den L2 loss[4]:

$$L1(out, \text{verlangt}) = \text{sum}(out - \text{verlangt})$$

$$L2(out, \text{verlangt}) = \text{sum}((out - \text{verlangt})^2)$$

Das  $\text{sum}()$  wird nur benötigt, um den loss als eine Zahl darzustellen. Für Backpropagation wird der intakte loss-Vektor benötigt. Bis jetzt kann man eine Aussage darüber machen, wie falsch das NN insgesamt ist, man interessiert sich aber nur für Aussagen über jedes einzelne Gewicht. Im Folgenden wird versucht, die Backpropagation



verständlich zu erklären. Einiges Wissen stammt dabei von Brilliant.org[9].

## 2.3 Fehlerrückführung - Backpropagation

Verfügt man über ein Neuronales Netz  $nn$ , ein Trainingsbeispiel  $t = (x, y)$  und eine loss-Funktion  $E$ , dann kann man eine Version von  $nn$  erzeugen, die Parameter benutzt, durch welche  $E$  kleiner wird, also besser wird. Die Idee ist, dass  $E$  nach den (allen) Parametern abgeleitet werden muss. Die Bedeutung der Ableitung ist gross: Sie sagt, wie sich ganz  $E$  verändert, wenn die Parameter angepasst werden. Addiert man die Ableitung zum Parameter, dann wird  $E$  grösser, deshalb addiert man die negative Ableitung, was  $E$  verkleinert. Ein Beispiel:

$$f(x) = -3x + 42, \frac{df}{dx} = -3$$

Setzt man  $x$  auf 15, dann ist  $f(15) = -3$ . Addiert man  $f'(x)$  zu  $x$ , so wird  $x$  zu 12, und  $f(12) = 6$ , also grösser als  $-3$ . Jedes Gewicht kann dann mit seiner partiellen Ableitung verbessert werden:

$$w_{ij}^k = w_{ij}^k - \alpha \frac{dE}{dw_{ij}^k}$$

Das ist die Update-Regel des Gewichts, welches die  $(k-1)$ -te Schicht mit der  $k$ -ten Schicht verbindet, nämlich von Knoten  $i$  in der vorherigen zu Knoten  $j$  in der  $k$ -ten Schicht (siehe Diagramm MLP). Das  $\alpha$  oben ist ein arbiträrer Wert, der später als Lernrate bezeichnet wird. Meistens ist dies ein Wert in  $]0, 1]$  und erlaubt ein verfeinertes Anpassen der Parameter. Der schwierigste Teil von Backpropagation ist jedoch das Ausrechnen der Ableitung selbst, weil die Netzstrukturen von  $nn$  sehr kompliziert werden können. Im Fall eines MLPs lässt sich dies relativ einfach nachvollziehen.

Die Ableitung  $dE/dw_{ij}^k$  lässt sich aufspalten zu:

$$\frac{dE}{da_j^k} \frac{da_j^k}{dw_{ij}^k}$$

$a_j^k$  ist dabei die Summe aller Inputs zum Knoten  $j$  in der  $k$ -ten Schicht bevor die Aktivierungsfunktion  $act$  wie Sigmoid oder ReLU angewendet wird.  $\frac{da_j^k}{dw_{ij}^k}$  ist nun einfach zu lösen:  $a_j^k = \sum_{I=0}^n (w_{Ij}^k o_I)$ ;  $o_I$  ist die Ausgabe vom  $I$ -ten Knoten in der vorherigen Schicht. Weil man nach einem bestimmten  $w_{ij}^k$  ableitet, fallen alle ausser folgender Summand weg:  $w_{ij}^k o_i$ ; d.h. wenn  $I = i$ . Somit ist die Ableitung von  $\frac{da_j^k}{dw_{ij}^k} = o_i \cdot \frac{dE}{da_j^k}$  ist mühsamer, denn  $a_j^k$  kann in jeder Schicht sein. Wüsste man jedoch  $\frac{dE}{da_i^{k+1}}$ , dann hätte man  $\sum (\frac{dE}{da_i^{k+1}} \frac{da_i^{k+1}}{da_j^k})$ ;  $a_i^{k+1}/a_j^k$  ergibt  $w^{k+1} act'(a_j^k)$ . Dadurch wird die Ableitung nach  $a_j^k$  abhängig von allen  $a_i^{k+1}$  aus der nachfolgenden Schicht gemacht. Wichtig ist, dass  $\frac{dE}{da_i^{k+1}}$ , die Knoteninputs zur letzten Schicht, berechnet werden können, was

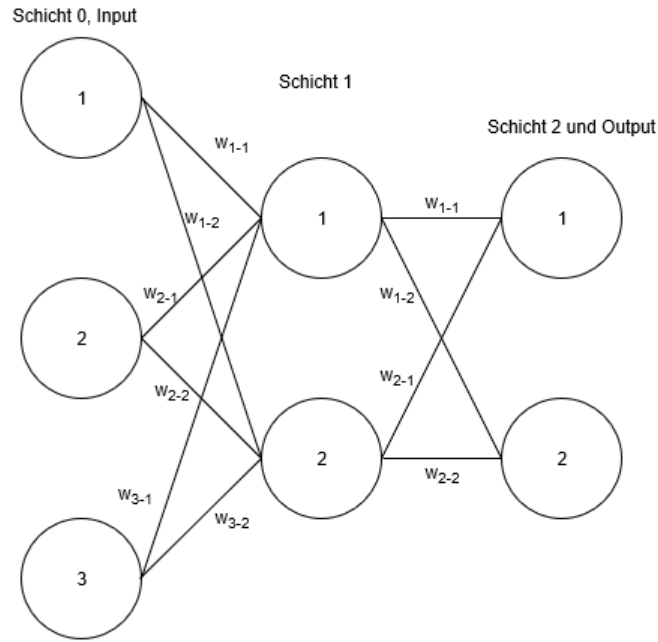


Abbildung 4: Ein MLP mit angeschriebenen Gewichten[13]

der Fall ist:  $\frac{dE}{da^{last}} = 2(o^{last} - y)act'(a^{last})$  für alle  $a^{last}$  in der letzten Schicht (mit entsprechendem Index). Zusammengesetzt ergibt das:

$$\frac{dE}{dw_{ij}^k} = act'(a_j^k) o_i^{k-1} \sum (w_{jl}^{k+1} \frac{dE}{da_l^{k+1}})$$

mit  $\frac{dE}{da_j^k} = w^{k+1} act'(a_j^k)$  für die vorherige Schicht. Nun werden die Schichten von hinten nach vorne durchlaufen, damit die notwendigen  $\frac{dE}{da_l^{k+1}}$  fortlaufend berechnet werden können. Dank der Backpropagation ist es möglich, ein Neuronales Netz zu jeder loss-Funktion zu optimieren - bis zu einem gewissen Grad natürlich.

## 3 Theorie Teil 2: Deep Reinforcement Learning

Ein herkömmliches Neuronales Netz, wie zum Beispiel ein CNN, reicht noch nicht aus, um eine KI zu erstellen, die in der Umgebung eines Spiels richtige Entscheidungen treffen kann. Weshalb? Die Netze, die bisher besprochen wurden, gehören zum Teilgebiet des *Supervised Learning* (etwa: Überwachtes Lernen). Bei diesem Konzept verfügt man immer über DatenSETS, welche in der Form ( Trainingsdaten, Korrekte Antwort ) verfügbar sind. Will man ein CNN dazu bringen, Bilder zu einer bestimmten Klasse zuzuordnen, so benötigt man Datensets der Art ( Bild, Klasse ). Glücklicherweise existieren Datenbanken zu diesem Zweck, zum Beispiel das ImageNet[8]. Wichtig ist, dass man erkennt, dass diese Datensets durch Menschenhand erstellt wurden. Grundsätzlich liesse sich dies auch für Spiele machen: Im Fall von Schach könnte man ein Expertenteam von Profischachspielern anstellen, die Unmengen von Stellungen durchgehen würden, um bei jeder entscheiden würden, was am Besten zu spielen sei; Auch könnte ein herkömmliches Schachprogramm diese Aufgabe übernehmen. Die Sache hat jedoch einen Haken: Ein Neuronales Netz wird immer versuchen, die Vorgabe, so gut wie es möglich ist, nachzuahmen. Deshalb kann es niemals besser als die Lösung werden. Ausserdem hat Niemand bewiesen, dass die Angaben der Profispieler oder des Schachprogramms mit Sicherheit korrekt wären (angesichts der Tatsache, dass AlphaZero besser als Stockfish spielt[15], muss man das wohl annehmen). Aus diesem Gründen wird klar, dass klassisches Supervised Learning nur beschränkten Erfolg als Spiele-KI haben kann. Wäre es denn möglich, die KI zu bauen, die nicht solche Lösungen oder Labels benötigt?

Genau mit dieser Frage beschäftigt sich (Deep) Reinforcement Learning. Im Folgenden werden einige Methoden und Konzepte aus diesem Bereich behandelt.

### 3.1 Markov Decision Process (MDP)

Um besser verstehen zu können, welche Eigenschaften eine Spiele-KI haben muss, führt man ein mathematisch formales Konzept ein, welches die Grundeigenschaften eines Spiels verkörpert. Damit sind einfache Tatsachen gemeint, wie zum Beispiel dass das Spiel in eine neue Stellung übergeht, nachdem der Spieler einen Zug ausgeführt hat. Man nennt das System einen Markov Decision Process, benannt nach dem russischen Mathematiker Andrej Markov, der es formuliert hat[21].

Die Beziehung zwischen diesen Variablen und Funktionen ist im obigen Bild dargestellt: Es gibt einen *agent*, der Spieler/der Handelnde, und das Environment, die Umgebung, bei uns das Spiel. Das MDP ist ein zeitlicher Ablauf: Es startet mit einer bestimmten Start-Stellung und durchläuft dann Zeitschritte (entweder unendlich lang oder endet beim Erreichen einer bestimmten End-Stellung). Bei jedem Zeitschritt erhält der Agent vom Environment die neue Stellung  $s$  und die Belohnung  $r$  betreffend den vorherigen Zeitschritt (und evtl. die verfügbaren Aktionen aus  $A$ ), er wählt ein

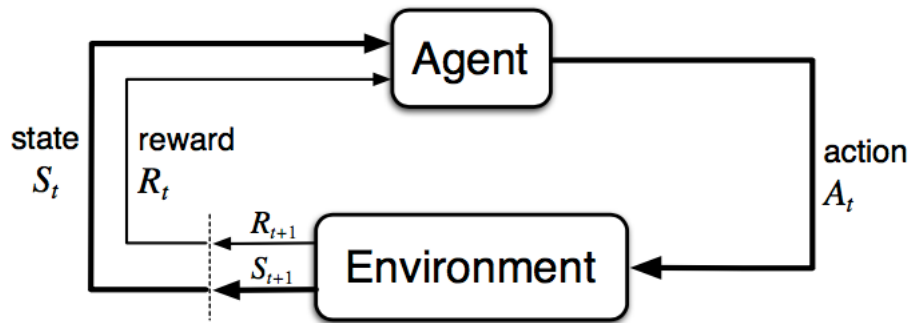


Abbildung 5: Diagramm zu MDP[2]

a aus, schickt es zurück ans Environment und nun beginnt der nächste Zeitschritt...  
Man definiert:

$S$  und  $s$ ; wobei  $S$  für die Menge aller möglichen Stellungen steht, und  $s$  für ein Element aus dieser Menge

$A(s)$  und  $a$ ; wobei  $A(s)$  die Menge aller in einer Stellung erlaubten oder möglichen Aktionen (= Züge) steht und wiederum  $a$  als Element aus  $A(s)$

$T(s, a) \rightarrow s'$ ; eine Funktion, die ein  $s'$  berechnet abhängig der Stellung  $s$  und der Aktion  $a$ , die sie erhalten hat - es sei angemerkt, dass es sich dabei um eine Verteilung handelt,  $s$  und  $a$  bestimmen lediglich, welches  $s'$  wahrscheinlicher ist

$R(s, a) \rightarrow r$ ; eine Funktion, die eine Belohnung  $r$  für das  $a$  berechnet, das in  $s$  genommen wurde - kann ebenfalls aus einer Verteilung stammen

Das Ziel ist es nun, den Agenten so steuern, dass  $r$  über das gesamte Spiel hinweg gesehen maximal ist.

### 3.2 Q-Learning

Der totale noch erreichbare *reward* aus einer Position  $s$  ist definiert als  $G$ :

$$G(s) = \sum_t^{t_{end}} \gamma^t r_t$$

Es wird also einfach jegliche Belohnung zusammengezählt, die im weiteren Spielverlauf noch dazukommt.  $\gamma$  ist dabei ein Faktor, der im Bereich  $[0, 1]$  liegt und Belohnungen, die noch in weiter Ferne liegen, abschwächt. Der Grund zu seiner Existenz liegt in ein paar mathematischen Annehmlichkeiten[siehe 19, Value Function]. Nun stellt man sich eine Funktion vor, die den *expected future reward* in Abhängigkeit vom momentanen Spielstand und einer möglichen Aktion weiss:

$$Q(s, a) = \text{Erwartungswert}(G(s)|s, a)$$

Die Funktion erhält den Buchstaben Q, was evtl. für *Quality* steht.  $Q(s, a)$  ist praktisch: Weiss man sie, so kann man direkt ein sinnvolles Verhalten für den agent ableiten: Man kann alle möglichen Züge nach ihrem Q-Wert abfragen und schliesslich denjenigen mit dem höchsten auswählen. Das ist als *greedy-policy* bekannt. Es ist nur selten möglich, eine Q-Funktion einfach so zu besitzen. Meistens muss sie approximiert werden. Dazu hilft die Bellman-Gleichung, die die Rekursivität der Q-Funktion deutlich macht.

$$Q(s, a) = \text{Erwartung}(r_t + \gamma \cdot Q(s_{t+1}, a_{t+1}) | s, a)$$

Die Summe aus G wird vereinfacht, indem alle ausser der erste Summand durch  $\gamma \cdot Q(s_{t+1}, a_{t+1})$  zusammengefasst wird.  $s_{t+1}$  ist dabei einer der Schwachpunkte. Die Funktion  $T(s, a)$  ist probabilistisch -  $s_{t+1}$  kann nicht mit aller Sicherheit bestimmt werden. Jedoch genau deshalb ist die Q-Funktion ein Erwartungswert: Mit bestimmten Wahrscheinlichkeiten gelangt man in eine Position  $s_{t+1}$ . Jedes s trägt aber implizit einen Wert mit sich, nämlich der der besten Aktion in  $s_{t+1}$ . Der implizite Wert eines jeden s ist dadurch:

$$s = \sum_{\text{alles } t+1} \max(Q(s_{t+1}, a)) P(s_{t+1} | s, a)$$

Jetzt will man wissen, wie man seine Q-Funktion verbessert. Bellman liefert dazu:

$$Q^*(s, a) = \text{Erwartung}(r_t + \gamma \cdot \max(Q^*(s_{t+1}, a_{t+1}) | s, a))$$

Im Fall der perfekten Q-Funktion muss diese Gleichung für alle s und a erfüllt sein.

### 3.3 Tabulares Q-Learning (=tQL)

Die Q-Funktion wird in tQL als 2d-Liste/Tabelle repräsentiert, wo s und a die Achsen sind. Trainingsdaten werden durch Spiel-gegen-sich-selbst (self-play) erzeugt. Das heisst, dass ein Spiel simuliert wird, bei dem immer die momentane Q-Liste nach dem besten Zug abgefragt wird, bis das Spiel zuende geht. Bei jedem Zeitschritt werden s, a, r und s' in eine Liste abgespeichert. Das sind alle nötigen Informationen, um ein Update in der Q-Tabelle durchführen zu können:

$$Q[s][a] = Q[s][a](1 - \alpha) + \alpha(r + \gamma \cdot \max(Q[s']))$$

$\alpha$  ist die Lernrate; Dadurch wird der neue Eintrag von  $Q[s][a]$  aus Anteilen aus dem alten und aus dem neuen Wert zusammengesetzt, die von alpha gewichtet werden. Meistens werden Updates nicht sofort ausgeführt, sondern zu einem späteren Zeitpunkt. Die [s,a,r,s']-Listen werden in einen Speicher aufgenommen, der vor der

Trainingsphase zufällig vermischt wird. Das führt zu einer Art experience replay.[19] TQL hat allerdings grosse Schwächen: Q könnte je nach Problemstellung fast unendlich gross werden: Wenn s Teile enthält, die kontinuierlich sind, d.h. ein Wert ist eine Zahl aus  $\mathbb{R}$ , ist es hoffnungslos, eine Liste aller Möglichkeiten zu führen.

### 3.4 Deep Q-Learning

Bei Deep Q-Learning(=DQN) erinnert man sich an die Tatsache, das die Q-Funktion, die bei tQL als Tabelle im Hintergrund geführt wird, auch Funktion angesehen werden kann, die durch ein neuronales Netz approximiert wird.

$$DQN(s, a, \phi) - > r$$

wobei phi für alle Parameter des Netzes steht. Vorteile sind sofort ersichtlich:

- DQN kann auf einem kontinuierlichen state-space angewendet werden.
- DQN kann states bewerten, die zuvor noch nie durch Spiel-gegen-sich-selbst erreicht wurden.
- Der Speicherverbrauch bleibt konstant.

Die Gewinnung der Trainingsdaten verläuft gleich wie bei tQL: [s,a,r,s']-Listen werden abgespeichert. Die Optimierung des Netzes wird durch folgende loss-Funktion vorgenommen:

$$loss = (DQN(s, a) - (r + y \cdot \max(DQN(sn))))^2$$

Es handelt sich dabei um einen MSE-loss zwischen  $DQN(s, a)$  und  $(r + y \cdot \max(DQN(sn)))$ . Dadurch wird die Differenz der beiden Terme minimiert, so dass das NN sich immer näher an die Bellman-Gleichung annähert.

## 4 Methoden

### 4.1 Tic-Tac-Toe

Es folgt ein Einblick in das TTT-Programm, welches zum Spielen und Trainieren notwendig war. Die Basis liegt in einem 2D numpy-array, welches direkt das Spielfeld repräsentiert. Alle Felder werden mit -1 initialisiert. Zieht der erste Spieler, dann wird am entsprechenden Ort eine 0 gesetzt, für den zweiten eine 1. Nach jedem Zug eines Spielers muss überprüft werden, ob das Spiel geendet hat. Bei Spielende wird der Sieger bestimmt. Letzteres ist einfach, denn ein Zug kann nur gewinnen oder Unentschieden herbeiführen, aber nicht verlieren. Ein Remis kann nur entstehen, wenn das ganze Brett gefüllt worden ist, aber keiner der beiden Spieler gewonnen hat. Weil das TTT-Programm erweiterbar sein soll, d.h. auch ein grösseres Spielfeld verarbeiten können soll, ist es nicht ganz offensichtlich, wie man überprüft, ob das Spiel geendet hat. Zum Beispiel könnte die Regel sein auf einem 10x10 Spielfeld nach fünf hintereinanderfolgenden  $x$  suchen. Um auf einem  $n \times m$  Feld nach  $k$  hintereinanderfolgenden gleichen Zeichen zu suchen, musste ein einfacher Algorithmus entworfen werden, der nach jedem Zug überprüft, ob das Spiel geendet hat. Es wird ausgenutzt, dass im Falle eines Siegs der letzte Zug entscheidend war, d.h. dass das zuletzt gesetzte Feld Teil der mindestens  $k$ -langen Linie ist. Deshalb geht der Algorithmus vom Feld des letzten Zugs aus und sucht sternförmig nach allen acht Richtungen nach Feldern, die gleich ausgefüllt sind wie das Anfangsfeld. Schlussendlich muss beachtet werden, dass die sich gegenüberstehenden Richtungen (z.B. links oben und rechts unten) in ihrer Länge zusammengezählt werden müssen.

```
Code:
foundinlineX = 1
origin = pos[:] # position vom letzten zug
while pos[0] != 0:
    pos[0] -= 1
    if self.field[pos[0], pos[1]] == who:
        foundinlineX += 1
    else:
        break
    if foundinlineX == self.numtwin:
        return True
pos = origin[:]
while pos[0] < self.field.shape[0] - 1:
    pos[0] += 1
    if self.field[pos[0], pos[1]] == who:
        foundinlineX += 1
    else:
        break
    if foundinlineX == self.numtwin:
```

```
return True
```

Listing 1: Code, der nach Lösungen in der Horizontale sucht

## 4.2 Tabulares Q-Learning

Für die Tabelle habe ich ein Python Standard-dictionary benutzt. Keys sind jeweils die string-Repräsentationen der entsprechenden Positionen aus dem TTT-Programm. Values sind jeweils ein numpy-array mit so vielen Einträgen, wie es actions gibt, also nxm. Sie sind auf null initialisiert.

Die Replay-Liste wird nun angelegt, indem effektiv eine arbiträre Anzahl Spiele durchlaufen wird. Mit der Q-Tabelle werden für beide Seiten die Züge gespielt, denen der höchste Wert zugeordnet ist; jedoch muss mit einer bestimmten Wahrscheinlichkeit (in meinen Trainingsläufen meistens im Bereich 50-60%) ein zufälliger Zug ausgeführt werden, weil sonst immer das gleiche Spiel gespielt würde, und das würde zu Nichts führen. Ebenfalls wichtig ist, dass nur legale Züge durchgelassen werden. Dies kann der Q-Learning-Algorithmus nicht selbst übernehmen, sondern muss extern gemacht werden.

```
def query(self, state):
    if str(state) not in self.Q:
        self.Q[str(state)] = np.zeros(self.sz)
    if random.uniform(0,1) < self.random_prob:
        return np.random.choice(TTTGame.get_legal(state))

    res = self.Q[str(state)]
    besta = -1
    bestr = -1000
    for lgldx in TTTGame.get_legal(state):
        if res[lgldx] > bestr:
            bestr = res[lgldx]
            besta = lgldx
    return besta
```

Listing 2: Abfrage-Funktion

Die Replays werden zufällig vermischt und anschliessend zum Training benutzt. Meistens werden zwischen 1000 bis 5000 Replays gespielt und trainiert. Bei tQL gestaltet sich das Verbessern relativ einfach:

```
def train(self):
    for replay in self.replay_save:
        s,a,r,sn = replay
        self.Q[str(s)][a] = self.Q[str(s)][a] * \
            (1 - self.lr) + self.lr * (r + self.decay * \
```



```
self.choose_from_legal(sn, self.Q[str(sn)])
```

Listing 3: Trainingsfunktion

### 4.3 Deep Q-Learning

Für DQN werden viele der oben genannten Konzepte und effektiv Funktionen übernommen. Die Q-Tabelle wird durch ein Neuronales Netzwerk ersetzt, welches mit TF.keras implementiert ist.

```
net = tfk.models.Sequential()
net.add(tfk.layers.Dense(50, input_shape=(9*3),
                           activation=tfk.activations.relu))
net.add(tfk.layers.Dense(9))
net.compile(optimizer=tfk.optimizers.SGD(lr=0.1),
            loss=tfk.losses.MSE,
            metrics=[tfk.metrics.MAE])
```

Listing 4: Beispiel für das Erstellen eines Netzwerks mit zwei dense(aka normalen)-Schichten

Mit den Methoden `net.predict` und `net.fit` lässt sich das Netz abfragen bzw. trainieren. Gleich wie beim Tabellenansatz muss die Ausgabe des Netzwerks mit einer Maske der noch möglichen Züge verglichen werden. Die `train`-Funktion ändert am Stärksten, weil die `[s,a,r,s']`-Listen für die `net.fit`-Funktion aufbereitet werden müssen.

DQN hat einen Input, der dreimal so gross ist wie das TTT-Feld. Das liegt an verwendeten one-hot encoding. Dabei wird die TTT-Position mit z.B. 3x3 Feldern, die jeweils einen Wert von -1,0 oder 1 besitzen, so umgeschrieben, dass jedes Feld mit einem Vektor der Länge 3 ersetzt wird. Diese Vektoren enthalten nur 0 und genau eine 1. Der Index, bei dem die 1 steht, hat dann die ursprüngliche Bedeutung. So steht der Index 0 für -1, 1 für 0 und 2 für 1.

```
[[ -1, -1, 0],
 [ -1, 0, 1],
 [ 1, 0, -1]]
wird zu:
[ 1, 0, 0],
[ 1, 0, 0],
[ 0, 1, 0],
....
[ 0, 0, 1],
[ 0, 1, 0],
[ 1, 0, 0]
```

One-hot encoding ergibt Sinn im Fall von TTT, weil die Bedeutung der Werte -1,0,1 für leer, Spieler0, Spieler1 gewissermassen nicht kontinuierlich/linear ist. Wäre es so, würde das heissen, dass ein grösserer Input für eine proportional grössere Bedeutung steht. Aber ein Wert von 12 hat keine Bedeutung in TTT. Ein NN will aber, dass der Input eine lineare Bedeutung hat. Das liegt hauptsächlich daran, dass das Netzwerk intern die Werte nur multipliziert und addiert (insbesondere wenn ReLU benutzt wird). Bei one-hot encoding erschafft man deshalb eine Dimension für jeden diskreten Wert.

Der Code ist auf Github unter <https://github.com/ldzgch/TTTlearning> verfügbar.

## 5 Resultate

tQL ist sehr erfolgreich nach genügend langer Trainingszeit: Fast immer kann die KI ein Unentschieden forcieren. Mache ich als Spieler einen Fehler, so kann sie diesen zum Gewinn ausnutzen. Jedoch ist es immer möglich, dass ein paar Stellungen existieren, die während der Trainingsphase zu wenig oft besucht wurden und folglich fehlerhaft bewertet werden. Abschliessende Sicherheit zu erlangen ist deshalb schwierig.

Auch DQN hat wenig Probleme, TTT zu lernen. Ein einfaches Dense-layer network mit ReLU-Aktivierungsfunktionen und folgenden Schichtgrössen:

$$27- > 100- > 100- > 100- > 9| \text{ inklusive Input und Output}$$

genügt, um gleiche Performance wie tQL zu erreichen.

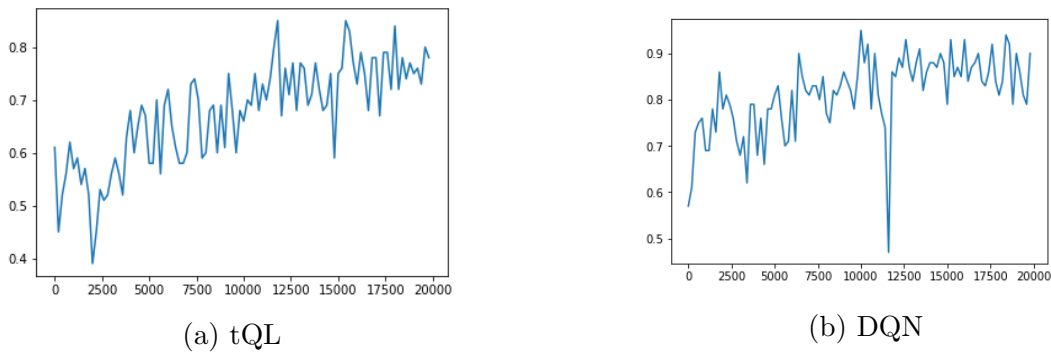


Abbildung 6: Gewinnchancen von tQL bzw. DQN während der Trainingsphase  
x-Achse: Anzahl trainierte Beispiele  
y-Achse: Gewinnchance als Wert zwischen 0 und 1

In Abbildung 6 sind die Leistungen beider Programme in der Trainingsphase während den ersten 20'000 Beispielen zu sehen. Der Gegner hat jeweils 100% zufällig gespielt. Wenn die Anzahl Trainingsbeispiele von Hundertausenden zu Millionen übergeht, erreichen die Programme annäherungsweise menschliche Leistung.

### 5.1 Vergrößerung auf 4x4

4x4 hat viel mehr Stellungen. Nicht im Sinne eines kontinuierlichen state-space, sondern in der Anzahl von diskret verschiedenen Stellungen. Q-Learning hat mit der Vergrößerung des Feldes enorm Mühe. Eine sehr grobe Abschätzung der Anzahl Stellungen:

$$3 \times 3 : 3^9$$

$$4 \times 4 : 3^{16}$$

Das bedeutet eine Vergrößerung um den Faktor 2187. Natürlich wird diese Zahl abgeschwächt durch die Natur der Regeln von TTT, jedoch reicht diese Zahl aus, um

erklären zu können, das ein Algorithmus wie Q-Learning, der stark vom wiederholten Durchlauf verschiedener Spielverläufe abhängig ist, nicht grossen Erfolg erzielt. tQL ist sehr stark davon betroffen: Ich habe mein Programm einige Zeit trainieren lassen, so dass total im Bereich von 15 Millionen Beispielen trainiert wurden und die Spielstärke bleibt ziemlich eingeschränkt. Ein menschlicher Spieler, der sich einigermaßen konzentriert, spielt besser. Daneben wird das Q-Dictionary wie erwartet riesig gross. Das Training habe ich nach einer gewissen Zeit abgebrochen; Die Anzahl der erfassten Positionen war in diesem Moment bei zirka 1.7 Millionen (bereit viel mehr als die 5478[23] vom Standard-TTT) und `numpy.save` erzeugt ein `save-file` von etwa 476 MB. Deep Q-Learning mit einem einfachen MLP/Dense-Layer-Netzwerk ist bereits um Längen besser, wie aus meinen Versuchen hervorgeht. Jedoch ist es immer noch schwach. Es ist zu vermuten, dass Q-Learning nicht der optimale Lösungsansatz für Tic-Tac-Toe ist, weil TTT von Natur aus viele Stellungen besitzt. Das Problem von Q-Learning ist, dass alle möglichen Positionen mehrmals durchlaufen werden müssen und dass dabei verschiedene Züge gespielt werden. Daraus schliesse ich, dass Q-Learning besser für Problemstellungen gedacht ist, die wenige states enthalten, mit einer Anzahl von  $s$  im Bereich von 100 bis 1000.

## 5.2 Verbesserungsvorschläge

Weil die Performance der Programme auf 4x4 so schlecht ist, sollen hier ein paar Ideen genannt werden, die vielversprechend klingen. Wäre noch mehr Zeit übrig geblieben (z.B. zwei Monate), so hätten sie durchaus getestet werden können.

Es soll noch einmal klargestellt werden: In dieser Arbeit wurden die Gebiete *deep learning* und *reinforcement learning* nur oberflächlich, d.h. in ihren Grundkonzepten, behandelt. Jedoch sind während der Recherche zu beiden Gebieten interessante Techniken gefunden worden, von welchen einige hier genannt werden sollen:

- *deep learning*

Das neuronale Netzwerk könnte kompliziertere Architekturen bessere Resultate erzielen:

- Mit der Verwendung von Convolutional-Schichten[5], RNN-Schichten[3] oder ResNets[7]
- Mit der Verwendung anderer Aktivierungsfunktionen wie leaky ReLU oder Tangens[6]

- *reinforcement learning*

Q-Learning ist nicht die einzige Methode, weitere Konzepte sind:

- Policy Gradient[19]
- REINFORCE-Algorithmus[19]

## 6 Fazit

Wenn ich die Projektvereinbarung durchsehe, die ich vor dreivierteljahren abgegeben habe und mit dem vergleiche, was ich tatsächlich als Programm umsetzen konnte, so ist klar, dass ich mir damals zu viel vorgenommen habe. Damals hatte ich die Idee, nach erfolgreichem Bezwingen von Tic-Tac-Toe mit Schach weiterzufahren. Es hat sich aber gezeigt, dass deep learning zuweilen eine mühsame Angelegenheit sein kann, wenn das Programm eben nicht funktioniert. In solch einer Situation fragt man sich, wo der Fehler liegt; Diesen zu finden ist allerdings schwieriger als bei einem normalen Algorithmus, weil das Verhalten meines Programms teilweise nur indirekt von meinem Code abhängt - das Programm lernt ja.

Trotzdem hat mir die Arbeit an diesem Projekt Spass gemacht. Das liegt zum Einen daran, dass verschiedene Phasen zum Teil recht abenteuerlich ausgefallen sind: Zu Anfang, also vor gut einem Jahr wusste ich noch sehr wenig über Neuronale Netzwerke und es war nicht klar, wie viel ich erreichen würde und was für einen Kantonsschüler in diesem Bereich überhaupt möglich ist.

Das Suchen und Verstehen von Informationen war ebenfalls eine Herausforderung. Dass mit Ausnahme des Buches[17], welches ich vor Beginn mit meiner Arbeit gelesen habe, fast alle Informationen im Internet in Englisch geschrieben sind, hat mich weniger gestört. Schwieriger war es, dass benötigte mathematische Verständnis hervorzubringen. Oft musste ich eine Textstelle mehrmals lesen oder an mehreren Tagen wieder anschauen, bis ich den Sinn einer Methode oder Formel begriffen habe.

Spannend war deshalb auch meine Selbstverantwortlichkeit für das Verständnis des Themas. Nur ich alleine kann helfen, mich vom Unverständnis zu befreien.

Selbstverständlich habe ich viel Neues im Bereich des *machine learning* gelernt. Zuweilen dachte ich verärgert, weshalb ich mich selbst genau in diese Ecke des Deep Reinforcement Learning stecken musste, als ich mich für mein Projekt entschieden habe; Viele spannende Phänomene wie Neural Style Transfer oder Deep Dream sind mir während der Arbeitsphase begegnet. Ich kann mir gut vorstellen, in Zukunft weitere *deep learning*-basierte KI zu programmieren - ganz einfach, weil es spannend ist.

Ich hoffe, dass ich im Leser Interesse für neuronale Netze wecken konnte - Das meinige ist mit Sicherheit vorhanden.

# Literatur

- [1] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. URL: <https://arxiv.org/abs/1712.01815>.
- [2] billy-inn. *Notes on Reinforcement Learning (1): Finite Markov Decision Processes*. 2016. URL: <http://billy-inn.github.io/blog/2016/10/05/notes-on-reinforcement-learning-1-finite-markov-decision-processes/>.
- [3] Stanford University School of Engineering. *Lecture 10 — Recurrent Neural Networks*. 2017. URL: <https://youtu.be/6niqTuYFZLQ> (besucht am 31.01.2019).
- [4] Stanford University School of Engineering. *Lecture 2 — Image Classification*. 2017. URL: <https://youtu.be/0oUX-n0EjG0?t=1499> (besucht am 30.01.2019).
- [5] Stanford University School of Engineering. *Lecture 5 — Convolutional Neural Networks*. 2017. URL: <https://youtu.be/bNb2fEVKeEo> (besucht am 31.01.2019).
- [6] Stanford University School of Engineering. *Lecture 6 — Training Neural Networks I*. 2017. URL: <https://youtu.be/wEoyxE0GP2M?t=1596> (besucht am 30.01.2019).
- [7] Stanford University School of Engineering. *Lecture 9 — CNN Architectures*. 2017. URL: <https://youtu.be/DA0cjicFr1Y> (besucht am 31.01.2019).
- [8] *IMAGENET*. URL: <http://www.image-net.org/> (besucht am 30.01.2019).
- [9] Christopher Williams et al. John McGonagle George Shaikouski. *Backpropagation*. URL: <https://brilliant.org/wiki/backpropagation/> (besucht am 30.01.2019).
- [10] Alexander Lenail. *NN-SVG*. URL: <http://alexlenail.me/NN-SVG/index.html> (besucht am 30.01.2019).
- [11] Philippe Lucidarme. *Simplest perceptron update rules demonstration*. 2017. URL: <https://www.lucidarme.me/simplest-perceptron-update-rules-demonstration/>.
- [12] Daniel McNeela. *The Universal Approximation Theorem for Neural Networks*. 2017. URL: [http://mcneela.github.io/machine\\_learning/2017/03/21/Universal-Approximation-Theorem.html](http://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html) (besucht am 30.01.2019).
- [13] *Mithilfe von Draw.io selbstgezeichnete Diagramme*. URL: [draw.io](http://draw.io).
- [14] Vartul Mittal. *Top 15 Deep Learning applications that will rule the world in 2018 and beyond*. 2017. URL: <https://medium.com/@vratulmittal/top-15-deep-learning-applications-that-will-rule-the-world%20in-2018-and-beyond-7c6130c43b01>.

- [15] pete. *AlphaZero gewinnt erneut ein 1000-Partien-Duell gegen Stockfish*. 2018. URL: <https://www.chess.com/de/news/view/alphazero-gewinnt-erneut-ein-1000-partien-duell-gegen-stockfish> (besucht am 31.01.2019).
- [16] pixabay.com. URL: <https://pixabay.com/de/tic-tac-toe-spiel-quadratspielen-150614/> (besucht am 30.01.2019).
- [17] Tariq Rashid. *Neuronale Netze selbst programmieren*. 2017.
- [18] Tinymind.com. *ReLU*. URL: <https://www.tinymind.com/learn/terms/relu> (besucht am 30.01.2019).
- [19] Lilian Weng. *A (Long) Peek into Reinforcement Learning*. 2018. URL: [<https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html>] (besucht am 29.01.2019).
- [20] Wikipedia. *Deep Learning*. 2019. URL: [https://en.wikipedia.org/wiki/Deep\\_learning#Deep\\_learning\\_revolution](https://en.wikipedia.org/wiki/Deep_learning#Deep_learning_revolution).
- [21] Wikipedia. *Markov Decision Process*. URL: [https://en.wikipedia.org/wiki/Markov\\_decision\\_process](https://en.wikipedia.org/wiki/Markov_decision_process) (besucht am 08.09.2018).
- [22] Wikipedia. *Perceptron*. 2019. URL: <https://en.wikipedia.org/wiki/Perceptron#History>.
- [23] Wikipedia. *Tic-Tac-Toe*. URL: [https://de.wikipedia.org/wiki/Tic-Tac-Toe#Strategie\\_und\\_Taktik](https://de.wikipedia.org/wiki/Tic-Tac-Toe#Strategie_und_Taktik) (besucht am 31.01.2019).
- [24] Mohamed Zahran. *A hypothetical example of Multilayer Perceptron Network*. 2015. URL: [https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network\\_fig4\\_303875065](https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065).