

PRAKTIKUM

Q-Learning mit Boltzmann-Quanten-Maschinen

Nicolas Tamino Kraus, Thomas Gabor, and Claudia Linnhoff-Popien

Ludwig-Maximilians-Universität München

Veröffentlicht 9 September 2021

Abstract

Wir wollen untersuchen, wie Quanten-Annealing für Reinforcement-Learning genutzt werden kann. Dafür nutzen wir Q-Learning mit einer ϵ -Greedy-Strategie. Die Q-Funktion approximieren wir dabei mit einer Boltzmann-Maschine. Dabei repräsentiert der sichtbare Layer den Zustand und die Aktionen. Die Aktivierung des nicht-sichtbaren Layers können wir mit einem Quanten Annealer berechnen, bzw klassisch simulieren. Als Umgebung für unseren Agenten nutzen wir den "zugefrorenen See", ähnlich zu der bekannten Problemstellung von OpenAI mit kleinen Modifizierungen. Es zeigt sich, dass eine einfache Q-Tabelle mit Abstand die besten Ergebnisse liefert. Wobei eine Quanten-Boltzmann Maschine komplexere Zusammenhänge modellieren kann und sehr ähnliche Ergebnisse liefert.

Keywords: Boltzmann-Maschinen, Reinforcement-Learning, Quanten Annealing

1. Einleitung

Reinforcement Learning ist ein sehr spannender Bereich innerhalb von maschinellem Lernen, da hier wenig Vorarbeit vom Menschen geleistet werden muss. Wir brauchen keine Trainingsdaten manuell zu erstellen, sondern der Algorithmus erstellt seine Trainingsdaten selbst. Gleichzeitig stoßen klassische Rechner hier auch sehr schnell an ihre Grenzen, da die Datensätze sehr groß werden können. Daher ist eine spannende und offene Frage, ob zukünftige Quanten-Computer auch in diesem Bereich irgendwann Vorteile gegenüber klassischen Verfahren haben könnten.

Wir wollen hier eine Methode anschauen, wie wir Quanten-Annealing für eine Deep-Boltzmann-Maschine anwenden können, die Ergebnisse untersuchen und mit klassischen Ansätzen vergleichen.

Als Problem betrachten wir einen Agenten, der eine Frisbee auf einem zugefrorenen See finden muss. Es gibt jedoch Löcher im Eis. Der Agent startet auf einer Startposition und kann sich nun wie auf einem Schachbrett Feld für Feld über den See bewegen. Fällt er dabei in ein Loch, hat er verloren und bekommt 0 Punkte. Findet er den richtigen Weg zur Frisbee, hat er gewonnen und bekommt 1 Punkt. Wir versuchen also den kürzesten Weg vom der Startposition zur Frisbee zu finden, ohne dabei in ein Loch zu fallen.

Ein bekanntes Problem dabei ist, dass wir einen guten Ausgleich zwischen Erkunden und bekannte Informationen verwenden finden wollen. Allgemeiner betrachtet, weiß der Agent nicht, wie viele Frisbees es gibt und wie viele Punkte er in dem Spiel bekommen kann. Daher will er möglichst viele Felder und Wege erkunden, bevor er sich einen konkreten Weg aussucht, den er bei jedem Spiel geht. Gleichzeitig will der Agent aber auch schnell die richtige Strategie finden, was konkret heißt, dass der

Computer weniger Rechenzeit verbraucht, bis er den schnellsten Weg gefunden hat. Dieses Dilemma lösen wir mit der ϵ -Greedy-Strategie.

Um die beste Aktion von einem gegebenen Feld zu finden, müssen wir die Aktionen bewerten. Die einfachste Möglichkeit dafür ist eine sogenannte Q-Tabelle, die für jedes Feld jedem Zug einen Wert zuordnet.

Eine mächtigere aber auch komplexere Möglichkeit ist, die Aktionen mit einer Q-Funktion zu bewerten, die durch ein neuronales Netz realisiert wird. Dafür können wir beliebige neuronale Netze auswählen und mit bekannten Optimierungsmethoden trainieren. Eine Boltzmann-Maschine ist dabei besonders interessant, da die Struktur einer Deep-Boltzmann-Maschine sehr ähnlich zu der Struktur von einem Ising Model in einem Quanten-Annealer ist. So können wir die Aktivierung des nicht-sichtbaren Layers auf dem DWave oder alternativ mit simuliertem Quanten-Annealing berechnen.

2. Problemstellung

2.1 Umgebung

Als erstes wollen wir genau definieren, wie unsere Umgebung/das Spielfeld aussieht und welche Regeln es gibt. Wir haben also ein Spielfeld ähnlich wie ein Schachbrett mit verschiedenen Feldern. Der Agent wird am Anfang auf ein Startfeld gesetzt, was sich in der Regel in einer der Ecken befindet. Von diesem Startfeld kann er nun in eine der vier Richtungen (links, unten, rechts, oben) einen Zug machen und sich ein Feld bewegen. Falls er sich mit diesem Zug vom Spielfeld bewegen würde, passiert nichts und er bleibt auf seinem Feld. Ist das Feld auf das er zieht ein Loch, wird das Spiel beendet und der Agent hat verloren. Das wird dadurch symbolisiert, dass er 0 Punkte bekommt. Ist das Feld auf das er zieht das Ziel, dann wird das Spiel auch beendet und der Agent

bekommt einen Punkt als Belohnung. Falls das Feld leer ist, bewegt sich der Agent auf das Feld und der Agent kann einen weiteren Zug machen.

Unten sehen wir beispielhaft zwei Spielfelder. In den Grafiken unten steht S für Start, Z für Ziel und L für Loch:

S			
	L		L
			L
L			Z

Figure 1. 4x4 Feld

Z				
		L		
		L		S

Figure 2. 3x5 Feld

2.2 Varianten der Umgebung

Unsere Variante ist eine etwas abgewandelte Version von der "Frozen-Lake" Umgebung die aus der Library Gym von OpenAi stammt. In der Originalversion gibt es allerdings Unterschiede: Zum einen wird der Agent zurück aufs Startfeld gesetzt, wenn er das Spielfeld verlässt. Und zum anderen ist das Feld in den Standardeinstellungen "rutschig". Das heißt wählt der Agent für den nächsten Zug z.B. rechts aus, gibt es eine bestimmte Wahrscheinlichkeit, dass der Agent auch wirklich nach

rechts geht. Ansonsten macht er einen zufälligen Zug. Beide Bedingungen machen das Spiel schwerer.

In (Crawford et al., 2019), gibt es dagegen noch eine fünfte Aktion "auf der Stelle bleiben". Außerdem endet das Spiel nach einer festen Anzahl von Zügen und der Agent hat z.B. die Möglichkeit eine Belohnung öfter zu bekommen, indem er auf dem Zielfeld mehrere Züge bleibt. Die Löcher werden dann durch "Bestrafungen" ersetzt und der Agent kann über diese Felder gehen. Dies ist also eine einfachere Version, da der Agent leichter zum Ziel kommt und die Belohnungen öfter einsammelt und damit öfter lernen kann. Außerdem werden zusätzlich Wände eingeführt, also im Prinzip Felder auf die der Agent nicht gehen kann. Die Grenzen des Spielfeldes sind auch Wände, dadurch reduzieren sich an den Randfeldern auch die Anzahl der möglichen Züge.

Wir haben uns für unsere Variante entschieden, da wir so mit durch den Ähnlichkeit mit OpenAi bessere Vergleichbarkeit zu bekannten Methoden haben. Gleichzeitig haben wir die Variante vom "Frozen-Lake" etwas vereinfacht, da eine komplexe Boltzmann-Maschine sich beim lernen etwas schwerer tut, als eine einfache Q-Tabelle wie sie bei OpenAi verwendet wird. Dafür kann eine Boltzmann-Maschine komplexere Zusammenhänge zwischen Zuständen und Aktionen erkennen. Dies hilft uns für diese Umgebung nicht, man sollte dies jedoch im Hinterkopf behalten, wenn man die Resultate vergleicht.

2.3 Darstellung der Umgebung

In Folgenden werden wir die Position auf dem Spielfeld als Zustand und die Züge als Aktionen bezeichnen. Wir haben damit für jedes Feld einen möglichen Zustand. Für den menschlichen Verstand ist es vorteilhaft sich das Feld als Matrix vorzustellen mit Reihen und Spalten. Dann werden alle Einträge der Matrix auf 0 gesetzt und das Feld auf dem wir stehen, wird auf 1 gesetzt. Um die Zustände aber besser in einem Algorithmus verwenden zu können, verwandelt wir ihn in einen einzelnen Vektor, wo wiederum jeder Eintrag für ein Feld steht.

In gleichem Sinne haben wir einen Vektor für die Aktionen, wobei der erste Eintrag für links, der zweite für unten, der dritte für rechts und der vierte für oben steht.

Die Umgebung wird auch durch eine Matrix dargestellt, wobei eine 1 für das Ziel steht und eine -1 für ein Loch. Die restlichen Einträge sind 0.

2.4 Der Agent

Der Agent ist unsere Intelligenz, die sich über das Spielfeld bewegt. Ein Agent hat eine Methode wie er lernt, die beim Erstellen des Agenten ausgewählt wird. Mit dieser Methode kann er einen Zug auswählen und er kann aus einem gemachten Zug lernen. Er kann also abwechselnd Züge machen und daraus lernen, bis er hoffentlich nach genügend Spielen gelernt hat, welche Aktionen in welchem Zustand am besten ist. Dies nennt man die

Police des Agenten. Am Anfang ist sie bei den meisten Methoden zufällig und wird dann durch das Lernen Stück für Stück verbessert.

Betrachten wir als Beispiel eine bestmögliche Police für unser 4x4 Beispiel von oben.

↓	→	↓	←
↓	L	↓	L
→	↓	↓	L
L	→	→	Z

Figure 3. Beispiel einer optimalen Police

Um diesen Lernvorgang sinnvoll zu gestalten nutzen wir die ϵ -Greedy-Strategie. Wir spielen insgesamt 10000 Spiele wobei ϵ zu Beginn auf 1 gesetzt wird. Jetzt wird eine Aktion zufällig gewählt. Nach jedem Spiel wird ϵ etwas gesenkt, um dann mit einer Wahrscheinlichkeit von ϵ eine zufällige Aktion zu wählen und sonst den besten Zug, den der Agent anhand seiner Police auswählt. Nach 90% der Spiele ist ϵ bei 0.01 angekommen, das heißt wir wählen mit 99% Wahrscheinlichkeit einen Zug nach der Police. So können wir bei den letzten 1000 Spielen beobachten, ob unser Agent die richtige Police gelernt hat.

Die ϵ -Greedy-Strategie hat den Vorteil, wirklich einen Roboter zu simulieren, der auf unbekannten Gelände unterwegs ist und aus seinen Aktionen lernt. Allerdings stößt sie schnell an ihre Grenzen, nämlich genau dann, wenn am Anfang vom Start aus durch zufällige Aktionen das Ziel fast nie gefunden wird. Tatsächlich passiert dies in unserer Version mit den Löchern die das Spiel beenden sehr schnell. Daher haben wir zwei Alternativen: Zum einen können wir verschiedene Startpositionen wählen, die erst nah am Ziel sind und sich dann weiter weg bewegen. Zum anderen können wir geordnet von jedem Feld jeden Zug machen und jeweils daraus lernen. Letzteres funktioniert natürlich viel besser, entspricht aber dann nicht mehr einem echten Spiel. Dafür können wir hier schauen, wie robust die Verfahren sind, wenn die Zustandsräume sehr groß werden.

2.5 Markov-Entscheidungsprozess

Ein Markov-Entscheidungsprozess ist ein zeitlich diskreter, stochastischer Steuerungsprozess. Hier soll ein Agent Entscheidungen treffen, wonach sich der Zustand mit einer bestimmten Wahrscheinlichkeit ändert.

Konkret ist ein Markov-Entscheidungsprozess ein Tupel $(S, A, \mathbb{P}, r, \gamma, \pi)$, wobei:

- S eine endliche Menge von Zuständen ist
- A eine endliche Menge von Aktionen ist
- $\mathbb{P}(s' \in S | s \in S, a \in A)$ gibt für jede Kombination aus einem Zustand s und einer Aktion a , die Wahrscheinlichkeit an, mit der wir in Zustand s' landen
- $r(s, a)$ ist eine Funktion in die reellen Zahlen, welche uns für einen Zustand s und eine Aktion a die Belohnung ausgibt
- $\gamma \in [0, 1)$ ist ein Diskontierungsfaktor für Belohnungen in folgenden Zeitschritten
- π ist die Police, die uns für jeden Zustand angibt, welchen Zug wir als nächstes ausführen

Unsere Umgebung die wir weiter oben definiert haben führt uns also zu einem Markov-Entscheidungsprozess. In unseren Beispielen verwenden wir größtenteils eine Wahrscheinlichkeit von 100% in dem naheliegenden State s' zu landen ($\mathbb{P}(s' \in S | s \in S, a \in A) = 1$) und 0% sonst ($\mathbb{P}(s' \in S | s \in S, a \in A) = 0$).

$r(s, a)$ ist in unserem Beispiel immer 0, es sei denn wir landen auf dem Zielfeld, dann gilt $r(s, a) = 1$.

γ können wir selbst wählen und wir werden hier später verschiedene Werte testen. π wollen wir optimieren, also ein optimales π^* finden.

2.6 Value-Funktion

Die Value-Funktion gibt für einen Zustand s und eine Police π die zu erwartende Belohnung an. Also die Belohnung, die voraussichtlich in diesem und allen weiteren Zeitschritten gesammelt werden kann. Hier benötigen wir den Diskontierungsfaktor γ , damit der erwartete Gewinn auf jeden Fall endlich bleibt (Sallans & Hinton, 2004)

$$V(\pi, s) = \mathbb{E} \left[\sum_{i=0}^{\infty} \gamma^i r(\Pi_i^s, \pi(\Pi_i^s)) \right] \quad (1)$$

und damit können wir die optimale Police ausdrücken mit:

$$\pi^*(s) = \operatorname{argmax}_{\pi} V(\pi, s) \quad (2)$$

Hier gilt es jedoch zu berücksichtigen, dass in unserem Beispiel, das Spiel endet, wenn wir in ein Loch fallen oder das Ziel erreichen. Wir müssen also die Value-Funktion dementsprechend anpassen. Die Summe geht dann nicht bis ∞ , sondern nur bis zu dem Zug, in dem das Spiel endet.

2.7 Value-Iteration und Q-Funktion

Die Value-Funktion können wir mit der Value-Iteration von Bellmann weiter aufteilen (Bellman, 1956):

$$\begin{aligned} V(\pi, s) &= \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r(\Pi_i^s, \pi(\Pi_i^s))\right] \\ &= \mathbb{E}[r(s, \pi(s))] + \gamma \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^{i-1} r(\Pi_i^s, \pi(\Pi_i^s))\right] \\ &= \mathbb{E}[r(s, \pi(s))] + \gamma \mathbb{E}\left[\sum_{s'} \mathbb{P}(s'|s, \pi(s)) V(\pi, s')\right] \end{aligned}$$

Und dies führt uns gleich zu der Q-Funktion, die zusätzlich eine konkrete Aktion bewerten soll:

$$Q(\pi, s, a) = \mathbb{E}[r(s, a)] + \gamma \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^{i-1} r(\Pi_i^s, \pi(\Pi_i^s))\right] \quad (3)$$

So können wir die Bellmann Optimalitätsgleichung mit $Q^*(s, a) = \max_{\pi} Q(\pi, s, a)$ aufstellen:

$$Q^*(s, a) = \mathbb{E}[r(s, a)] + \gamma \sum_{s'} \mathbb{P}(s'|s, a) Q^*(s', a') \quad (4)$$

Mit Hilfe dieser Gleichung können wir eine Folge Q_k aufstellen die mit einem beliebigen Startwert Q_0 startet und dann iterativ durch die folgende Regel aktualisiert werden kann:

$$Q_{k+1}(s, a) = \mathbb{E}[r(s, a)] + \gamma \sum_{s'} \mathbb{P}(s'|s, a) Q_k(s', a') \quad (5)$$

Daraus ergibt sich der SARSA Algorithmus. Hier wird die Q-Funktion iterativ aktualisiert, während die Zustände und Aktionen beobachtet und ausgeführt werden:

$$Q(s_n, a_n) \leftarrow \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) \quad (6)$$

Hier ist α die Lernrate und n steht für die n -te Aktion die ausgeführt wird.

3. Die Methoden

3.1 Q-Tabelle

Die einfachste Möglichkeit den SARSA Algorithmus umzusetzen ist die sogenannte Q-Tabelle. Betrachten wir dafür unser Beispiel der Police aus Figure 13, dann haben wir 16 Zustände und 4 Aktionen. Damit ist unsere Q-Tabelle eine 16x4 Matrix. Wir können uns nun für einen State i die i -te Reihe in unserer Q-Tabelle anschauen und mit dem argmax dieser Reihe die beste Aktion auswählen.

Am Anfang können wir bei dieser Methode einfach eine leere Tabelle verwenden, die nur mit 0-ten gefüllt

ist. Dann können wir in jedem Schritt mit dem ϵ -greedy Algorithmus eine Aktion ausführen und nach jeder Aktion mit SARSA (6) die Q-Tabelle aktualisieren. Beachte hier, dass $Q(s_{n+1}, a_{n+1})$ immer null wird, falls wir bei diesem Zug in einem Loch oder am Ziel landen, weil dann keine weiteren Züge stattfinden können.

Diese einfache Methode funktioniert hier sehr gut, da es keine wirklichen Zusammenhänge zwischen Zuständen gibt. Insbesondere können wir hier nicht in zwei verschiedenen Zuständen gleichzeitig sein. Es gibt wirklich nur pro Feld einen Zustand. Bei ähnlichen Vektoren die mit neuronalen Netzen bearbeitet werden, kann ein binärer Vektor mit 16 Einträgen 2^{16} Zustände ausdrücken und nicht nur 16.

3.2 Ein einfaches neuronales Netz

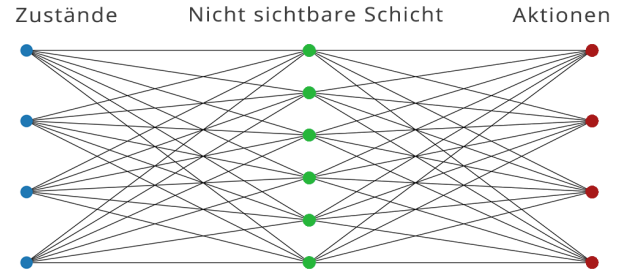


Figure 4. Einfaches neuronales Netz

Die naheliegende Möglichkeit ein neuronales Netz zu verwenden, ist die Funktion $Q(s, a)$ mit einem neuronalen Netz zu approximieren. Hierfür benutzen wir einfach einen versteckten Layer von Neuronen und realisieren dies mit PyTorch. Diese Layer werden linear verbunden und mit der Relu-Aktivierungsfunktion versehen. Wir können dann den Fehler direkt mit der rechten Seite von 6 berechnen und an das Netz übergeben. Dazu benutzen wir den Adams-Optimierer.

3.3 Generelle Boltzmann Maschine

Eine Boltzmann Maschine ist ein ungerichtetes probabilistisches graphisches Modell (Sallans & Hinton, 2004). Die Ecken des Graphen sind binäre Variablen und können die Werte 1 und 0 annehmen. Sie sind üblicherweise in sichtbare (v) und nicht-sichtbare (h) Variablen aufgeteilt. Die gewichteten Kanten sind paarweise symmetrische Interaktionen. Die Gewichte bestimmen die "Energie" von den Variablen. Allgemein können diese Gewichte zwischen allen Variablen auftauchen.

$$E(v, h) = - \sum_{k,i} w_{k,i} v_i h_k - \sum_{i < j} w_{ij} v_i v_j - \sum_{k < m} w_{km} h_k h_m \quad (7)$$

wobei i und j Indices über die sichtbaren Variablen und k und m Indices über die nicht-sichtbaren Variablen sind. Wir wollen eine Energie-Funktion in Abhängigkeit der sichtbaren Variablen aufstellen:

$$F(v) = \sum_h \mathbb{P}(h|v) E(v, h) + \sum_h \mathbb{P}(h|v) \log(\mathbb{P}(h|v)) \quad (8)$$

Dabei steht $\mathbb{P}(h|v)$ für die Wahrscheinlichkeit, dass die Variable $h = 1$ ist, mit dem gewählten Variablen v . Um die erste Summe zu minimieren, werden möglichst viele Variablen auf 1 gesetzt, wo $E(v, h)$ niedrig bzw. negativ ist und um die zweite Summe zu minimieren, soll die Wahrscheinlichkeitsverteilung von \mathbb{P} eine hohe Entropie haben, das heißt, die Wahrscheinlichkeit soll entweder 1 oder 0 sein.

Interessanterweise ist es recht einfach die Gewichte zu optimieren:

$$\frac{\partial F(v)}{\partial w_{ik}} = -v_i \langle h_k \rangle_{\mathbb{P}(h|v)} \quad (9)$$

Dies liegt daran, dass die die Verteilung $\mathbb{P}(h|v)$ für $F(v)$ minimal ist, sodass die Ableitung von $F(v)$ in Abhängigkeit zu dieser Verteilung 0 ist. Für Details dazu siehe (Sallans & Hinton, 2004) Appendix A.

Wir können jetzt mit der Funktion F und der Optimierung der Gewichte eine Q -Funktion für einen Markov-Entscheidungsprozess mit einer Boltzmann Maschine darstellen. Die Zustände und Aktionen werden dabei zu einem Vektor vereinigt und bilden die sichtbaren Variablen. Es gilt:

$$F(v) = -Q(v) \quad (10)$$

3.4 Beschränkte Boltzmann Maschine

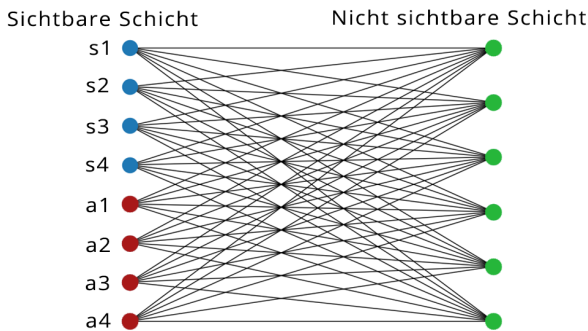


Figure 5. Beschränkte Boltzmann Maschine

Eine beschränkte Boltzmann Maschine hat nur Gewichte zwischen sichtbaren und nicht-sichtbaren Variablen. So entsteht ein bipartiter Graph. Ansonsten bleibt alles gleich wie in (8) und (9). Die Energie $E(v, h)$ reduziert sich dann auf:

$$E(v, h) = - \sum_{i,k} w_{i,k} v_i h_k \quad (11)$$

$Q(s, a)$ können wir dann wie folgt zusammenfassen:

$$\begin{aligned} Q(s, a) &= \sum_h \mathbb{P}(h|s, a) E(s, a, h) - \sum_h \mathbb{P}(h|s, a) \log(\mathbb{P}(h|s, a)) \\ &= - \sum_{k,i} w_{i,k} s_i \langle h_k \rangle - \sum_{k,j} w_{j,k} a_j \langle h_k \rangle \\ &\quad - \sum_k \langle h_k \rangle \log(\langle h_k \rangle) + (1 - \langle h_k \rangle) \log((1 - \langle h_k \rangle)) \end{aligned}$$

wobei $\langle h_k \rangle$ mit der sigmoid Funktion berechnet wird:

$$\langle h_k \rangle = \sigma\left(\sum_i w_{i,k} v_i\right)$$

Dazu leiten wir uns aus (6) und (9) die Regeln her, um nach jedem Schritt zu lernen:

$$w_{sh} + = \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) s \langle h \rangle$$

$$w_{ah} + = \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) a \langle h \rangle$$

3.5 Tiefe Boltzmann Maschine

Die tiefe Boltzmann Maschine unterscheidet sich von der beschränkten Boltzmann Maschine dadurch, dass hier zusätzliche Verbindungen zwischen den Variablen der versteckten Schichten sind.

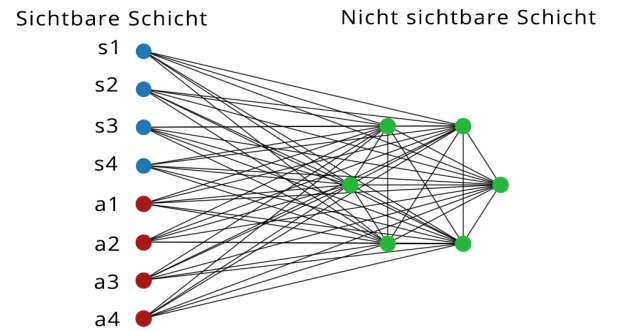


Figure 6. Tiefe Boltzmann Maschine

Allgemein kann man eine tiefe Boltzmann Maschine auch mit mehreren versteckten Schichten aufbauen. Dann wandern die sichtbare Schicht mit den Aktionen auf die rechte Seite und die sichtbaren Schichten beeinflussen nur die benachbarten Variablen in den nicht-sichtbaren Schichten. Diese Variante wollen wir hier nicht betrachten.

Die Q -Funktion erweitert sich jetzt hier im Vergleich zur beschränkten Boltzmann Maschine um einen weiteren Term:

$$Q(s, a) = - \sum_{k,i} w_{i,k} s_i \langle h_k \rangle - \sum_{k,j} w_{j,k} a_j \langle h_k \rangle - \sum_{k,m} w_{km} \langle h_k h_m \rangle \quad (12)$$

$$- \sum_k \langle h_k \rangle \log(\langle h_k \rangle) + (1 - \langle h_k \rangle) \log((1 - \langle h_k \rangle)) \quad (13)$$

und auch die Update-Regel bekommt einen weiteren Teil:

$$w_{sh} + = \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) s \langle h \rangle$$

$$w_{ah} + = \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) a \langle h \rangle$$

$$w_{hh'} + = \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) \langle hh' \rangle$$

Jetzt kommt der spannende Teil. Die freie Energie der tiefen Boltzmann Maschine kann aus (7) abgeleitet wie folgt berechnet werden:

$$E_v(h) = - \sum_{k,i} w_{k,i} v_i h_k - \sum_{k < m} w_{km} h_k h_m \quad (14)$$

Dabei sind die sichtbaren Schichten v jetzt feste Parameter und nur noch h sind freie Variablen. Da $\langle h \rangle$ binäre Variablen sind und wir die Gesamtenergie minimieren wollen, handelt es sich hier also um ein quadratisches, binäres Optimierungsproblem kurz QUBO. Dieses können wir mit Hilfe von Annealing lösen oder auch mit Hilfe von einem Quantencomputer mit QAOA oder mit adiabatischen Quantencomputern. Wir interessieren uns vor allem für Quanten-Annealing, einer Unterform von adiabatischen Quantencomputern, welches von DWave erforscht wird. Wir haben dabei einen vollvernetzten Graphen mit so vielen Variablen, wie wir Variablen in der versteckten Schicht haben. Hier kann der DWave bekanntlich auch Probleme mit z.B. 16 Variablen gut lösen. Da wir aber in unseren Tests für einen Testlauf meistens 10 000 Spiele auf dem Frozen Lake spielen wollen und für jeden einzelnen Zug 8 QUBO's lösen müssen, verwenden wir hier simuliertes Quanten Annealing.

3.6 Quanten Annealing

Wir können ein Optimierungsproblem als ein System beschreiben, sodass die minimale Energie des Systems der Lösung von unserem Optimierungssystem entspricht. Für das Quanten Annealing können wir hier das Ising Problem verwenden. Ein QUBO ist allerdings äquivalent dazu und wir können ein QUBO in ein Ising Problem umwandeln. So ein Ising Problem stellen wir dann mit einem Hamiltonian dar, welcher die Energie des Systems beschreibt.

Nach dem Adiabatischen Theorem gilt: Wenn sich das System im Grundzustand des ersten Hamiltonians

befindet und sich dann adiabatisch langsam genug zeitlich verändert, wird es sich danach im Grundzustand des zweiten Hamiltonians befinden.

Der erste Hamiltonian H_0 ist dabei das sogenannte Transverse Feld, dessen Energieminimum eine Superposition von allen Qubits ist. Der zweite Hamiltonian H_1 ist dabei ein von uns aufgestellter Hamiltonian, dessen Energieminimum der Lösung von unserem Optimierungsproblem entspricht.

$$H_0 = - \sum_i \sigma_i^x \quad (15)$$

$$H_1 = \sum_{i,j} J_{ij} \sigma_i^z \sigma_j^z + \sum_i h_i \sigma_i^z \quad (16)$$

Wir können dann einen zeitabhängigen Hamiltonian angeben, der eine Mischung vom ersten und zweiten Hamiltonian darstellt:

$$H(t) = (1 - t)H_0 + tH_1, t \in [0, 1] \quad (17)$$

Wenn wir diesen zeitlichen Prozess, den wir Annealing nennen, langsam genug durchführen, kommen wir am Schluss bei $t = 1$ sicher in der minimalen Energie des H_1 Hamiltonians an und haben damit unser Optimierungsproblem gelöst. Wie langsam dieser Prozess sein muss, hängt dabei vom minimalen Unterschied (Gap) des niedrigsten Energie Niveaus zum ersten angeregten Energieniveau ab.

Wir benutzen Quanten Annealing oder simuliertes Quanten Annealing um eine Lösung von (14) zu finden. Allerdings haben wir hier eine Besonderheit, wir wollen nicht unbedingt immer die beste Lösung finden, denn dann wären unsere versteckten Variablen immer entweder 0 oder 1, wie die binären Variablen in der Lösung vom QUBO. Stattdessen führen wir mehrere Durchläufe vom Annealing Prozess durch und wählen dabei die Annealing Zeit so kurz, das wir verschiedene Lösungen finden. So können wir den Durchschnitt der Lösungen berechnen und bekommen so weichere Aktivierungen für unsere versteckten Variablen, die auch zwischen 0 und 1 liegen können.

Wir können dann (12) durch die durchschnittliche Energie ersetzen, die wir in unserem QUBO Problem durch den Hamiltonian ausgerechnet haben. Diese bekommen wir vom simulierten Annealing (wir benutzen dafür eine Library von DWave) zurück und können diese gleich einsetzen.

Wie wir dann auch in unseren Experimenten sehen können, brauchen wir (13) für diese Version nicht mehr, da wir mit den Parametern im simulierten Annealing schon steuern können, wir sehr die Variablen richtung 0 oder 1 tendieren. Damit wird die Zeile einfach überflüssig und wir sehen auch in den Experimenten, dass dieser Term dafür sorgt, dass wir die Lösungen langsamer finden.

4. RESULTS

4.1 Prinzipien des Testens

Wir wollen hier festlegen, wie wir testen, was wir testen und wie die Ergebnisse dargestellt und wie sie zu bewerten sind.

Als erstes wollen wir bei jeder Methode die verschiedenen Parameter und ihre Auswirkungen auf die Lösungen testen. Dabei beschränken wir uns hauptsächlich auf ein 3x3 Feld mit der ϵ -Greedy-Methode, da wir hier weniger Rechenzeit brauchen und mit jeder Methode richtige Lösungen finden können. Wenn wir bereits gute Parameter einer Methode gefunden haben, ist es auch interessant kompliziertere Spielfelder auszuprobieren. Bei sehr großen Spielfeldern stößt irgendwann die ϵ -Greedy-Methode an ihre Grenzen, da wir hier mit einer zufälligen Police am Anfang nicht mehr das Ziel finden können.

S		L
		L
L		Z

Figure 7. 3x3 Feld

Hier gibt es zwei Möglichkeiten, die erste ist, dass wir unser Startfeld in die Nähe des Ziels setzen und so lernen. Dann können wir ohne das Netz zurückzusetzen, das Startfeld weiter vom Ziel entfernen und von dort weiter lernen. So können wir das Startfeld in mehreren Schritten zur eigentlichen Startposition bewegen und haben so eine Chance, die richtige Lösung zu finden. Interessant ist hier, wie stabil die Methoden mit den bis jetzt gelernten Policen umgehen und etwas dazulernen können, ohne alles zu verlernen.

Die zweite Möglichkeit ist, komplett von der ϵ -Greedy-Methode abzusehen und eine Schleife über alle Felder zu machen und für jedes Feld jede Aktion zu testen und daraus zu lernen. Das funktioniert besser, braucht aber für große Spielfelder sehr viel Rechenzeit, bis wir die richtige Lösung gefunden haben.

Die Resultate lassen wir durch einen Graphen anzeigen. Bei der ϵ -Greedy-Methode stehen auf der y-Achse die durchschnittlichen Punkte von 100 Spielen. Bei der Schleife über alle Felder dagegen, betrachten wir, von wie vielen Feldern wir die richtige Lösung finden können und zeichnen dies auf der y-Achse auf.

4.2 Q-Tabelle Resultate

Die Q-Tabelle findet immer eine passende Lösung, wenn bei der ϵ -Greedy-Methode zumindest ab und zu eine Lösung gefunden werden kann.

Der blaue Graph ist hier übrigens ϵ . In den meisten Vergleichen, wenn wir verschiedene Parameter testen wollen, lassen wir ϵ weg. Die ϵ Kurve sieht aber immer gleich aus, es sei denn, wenn wir verschiedene Startfelder nacheinander wählen. Dann startet ϵ jedes Mal wieder bei 1.

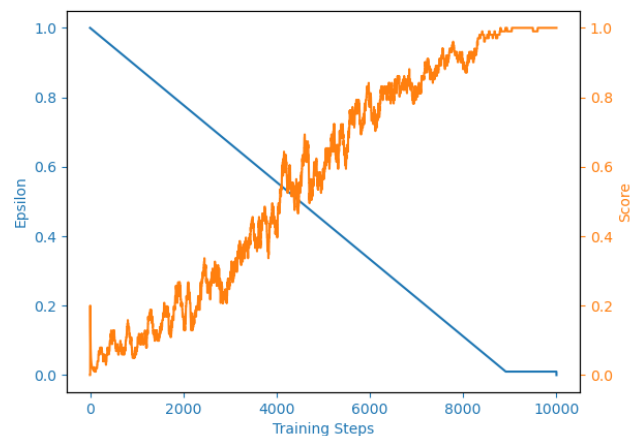


Figure 8. 3x3

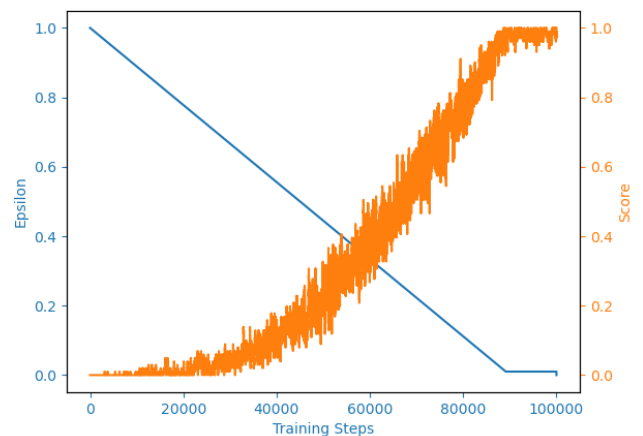


Figure 9. 5x5

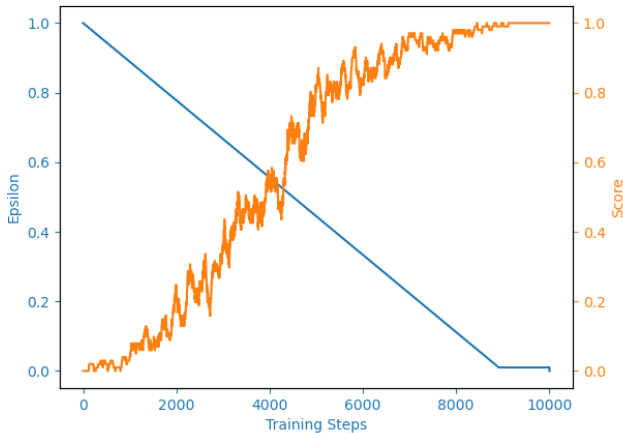


Figure 10. 8x8

Das 5x5 Feld ist sehr voll mit Löchern, daher sorgen hier geringe ϵ immer noch für einige falsche Züge und damit für niedrigere Punkte. Allgemein sind die Ergebnisse alle vergleichsweise sehr gut. Parameter testen lohnt sich hier nicht, da wir nur die Lernrate benötigen und diese quasi keine Rolle spielt. Z.B. eine Verdopplung der Lernrate sorgt einfach nur dafür, dass sich alle Einträge in der Q-Tabelle verdoppeln, was zu exakt dem gleichen Ergebnis führt. Wir können im weiteren die Ergebnisse der Q-Tabelle als Referenz für eine sehr gute Methode hernehmen.

4.3 Einfaches neuronales Netz Resultate

Hier wollen wir als erstes testen, wie viele Neuronen wir in der versteckten Schicht brauchen, um überhaupt richtige Ergebnisse zu erzielen. Dies testen wir in dem kleinsten 2x2 Spielfeld.

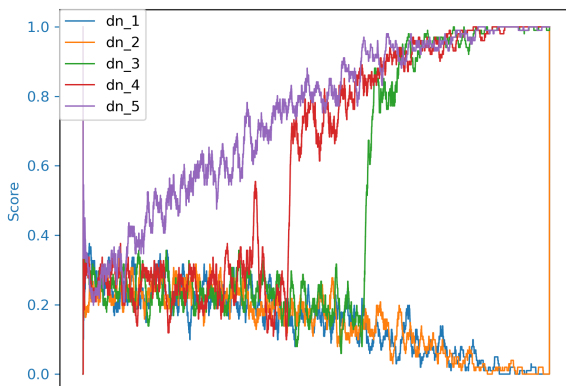


Figure 11. Tiefe Boltzmann Maschine

Wir sehen also, wir brauchen mindestens 3 Neuronen, um auf die richtige Lösung zu kommen. Das ist

interessant, da wir 4 Aktionen und 4 Zustände haben. Ab 5 Neuronen gibt es keine klaren Unterschiede mehr bei dem kleinen Spielfeld.

Beim 3x3 Feld brauchen wir gleich deutlich mehr Neuronen. Zwischen 8 und 64 Neuronen bekommen wir richtige Ergebnisse.

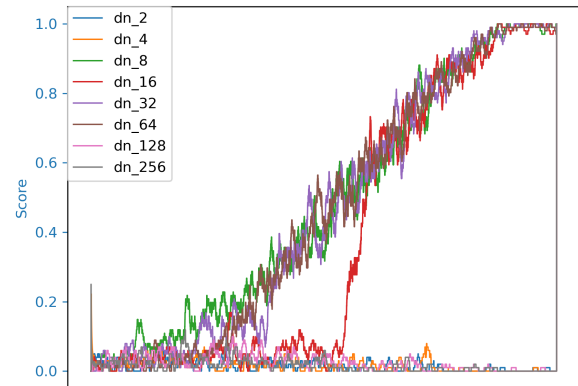


Figure 12. Tiefe Boltzmann Maschine

Als nächstes testen wir mit 16 Neuronen weiter und vergleichen verschiedene Werte für γ .

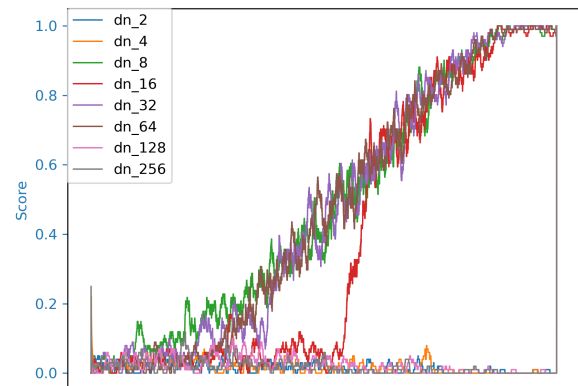


Figure 13. Tiefe Boltzmann Maschine

Acknowledgement

Wir bedanken uns bei allen Unterstützern!

References

- Bellman, R. 1956, Proceedings of the National Academy of Sciences, 42, 767
- Crawford, D., Levit, A., Ghadermarzy, N., Oberoi, J. S., & Ronagh, P. 2019, Reinforcement Learning Using Quantum Boltzmann Machines, arXiv:1612.05695
- Sallans, B., & Hinton, G. E. 2004, J. Mach. Learn. Res., 5, 1063