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 Reinforcment-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 Layern 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-Table 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

Reinforment 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 Dilema lösen wir in unserem Beispiel 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 oder einer Allgemeinen-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 Quaten-Annealing brechnen.

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 Spieldeld 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:

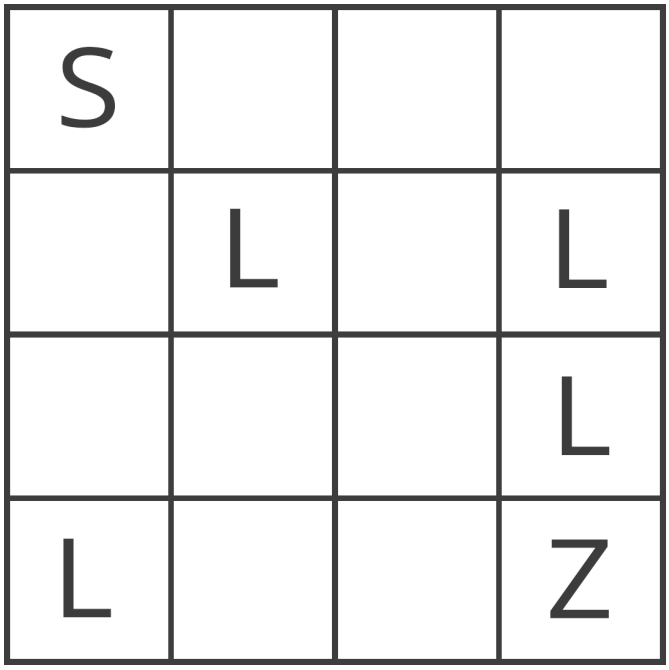

Figure 1. 4x4 Feld

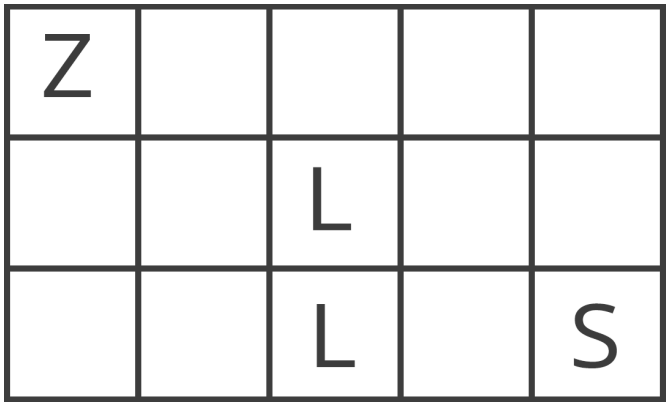

Figure 2. 3x5 Fekd

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 drüber 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 auf 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.

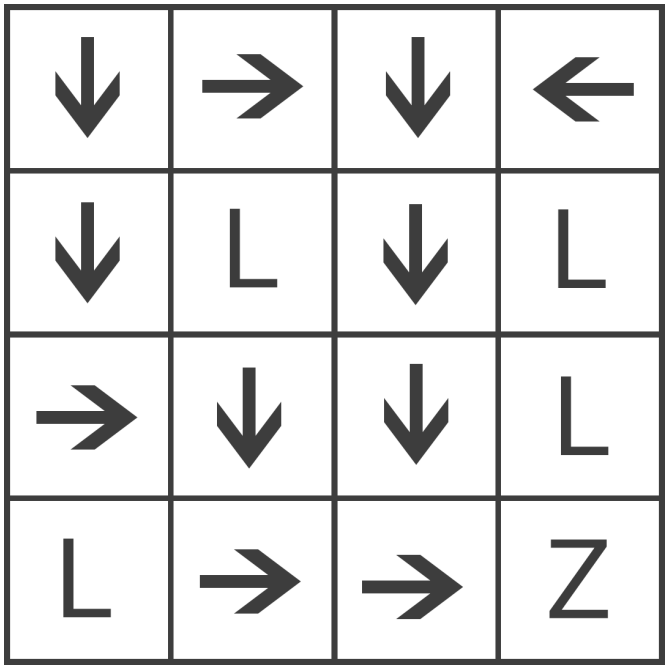

Figure 3. Beispiel einer optimalen Police

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 seine Police auswählt. Nach 90% der Spiele ist ϵ bei 0.01 angekommen, dass 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.

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 Menge von Aktionen ist
- $\mathbb{P}(s' \in S | s \in S, a \in A)$ gibt für jede Kombination aus einem Zustand s und eine Aktion a, die Wahrscheinlich an, mit der wir in Zustand s' landen
- $r(s, a)$ ist eine Funktion in die rellen Zahlen, welche uns für einen Zustand s und eine Aktion a die Belohung ausgibt
- $\gamma \in [0, 1)$ ist ein Diskontierungsfaktor für Belohungen 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 den 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 Belohung an. Also die Belohnung, die vorraussichtlich 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)

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

und damit können wir die optimale Police ausdrücken mit:

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

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):

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

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

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

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 geupdatet 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 aktuallisiert, während die Zustände und Aktionen beobachtet und ausgeführt werden:

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

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 6, dann haben wir 16 States 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-len 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 kaum 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

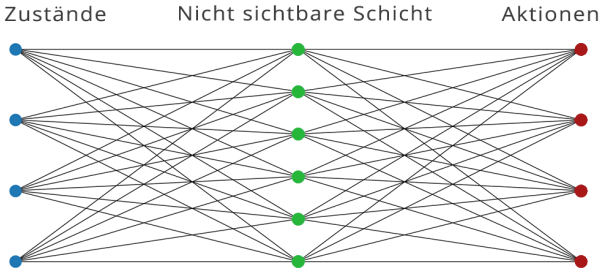

Figure 4. Einfaches euronales Netz

Die naheliegenste 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 Beschränkte Boltzmann Maschine

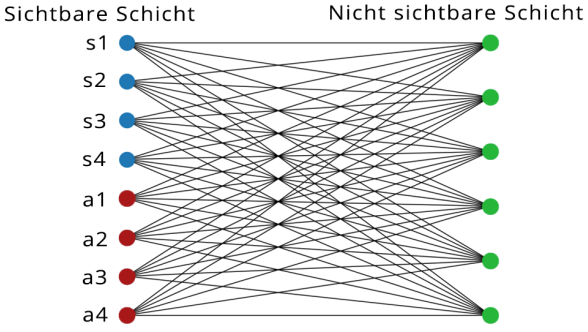

Figure 5. Beschränkte Boltzmann Maschine

3.4 Generelle Boltzmann Maschine

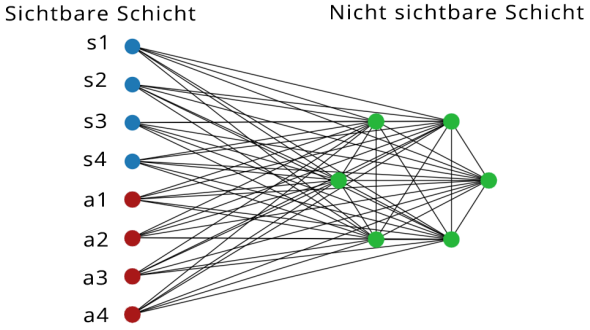

Figure 6. Generelle Boltzmann Maschine

4. RESULTS

. . .

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