

## PRAKTIKUM

# Q-Learning mit Boltzmann-Quanten-Maschinen

Nicolas Tamino Kraus

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  $\epsilon$ -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. Wir vergleichen diese Quanten Boltzmann Maschine anschließend mit einer einfachen Q-Tabelle, mit einem einfachen neuronalen Netz von PyTorch und mit einer Beschränkten Boltzmann Maschine.

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

### 1. Einleitung

Reinforcement Learning ist ein sehr spannender Bereich innerhalb von maschinellem Lernen, da hier wenig vorarbeiten 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  $\epsilon$ -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 Quaten-Annealing berechnen.

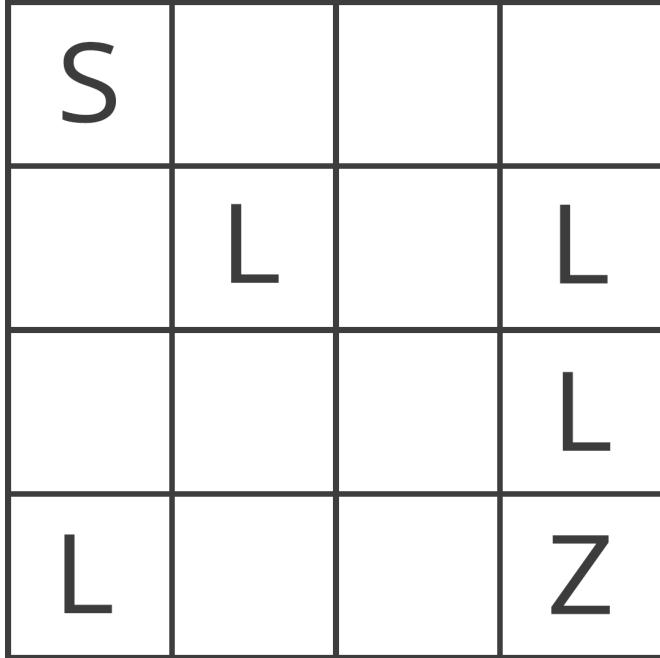
### 2. Problemstellung

#### 2.1 Umgebung

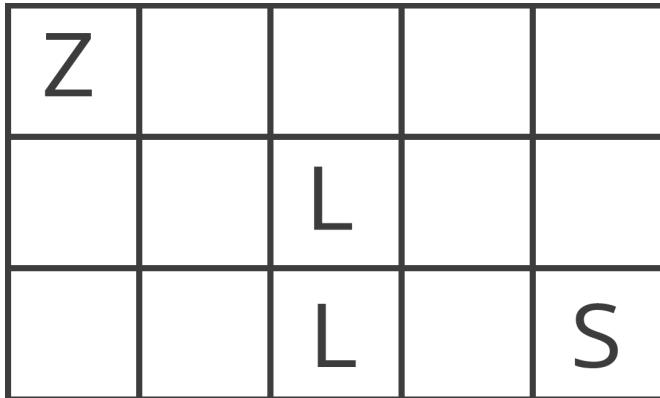
Als erstes wollen wir genau definieren, wie unsere Umgebung/das Spielfeld aussieht und welche Regeln es gibt. Wir haben also ein Spielfeld ähnlich wie ein Schachbrett mit verschiedenen Feldern. Der Agent wird am Anfang auf ein Startfeld gesetzt, was sich in der Regel in einer der Ecken befindet. Von 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:



**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 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  $\epsilon$ -Greedy-Strategie. Wir spielen insgesamt 10000 Spiele wobei  $\epsilon$  zu Beginn auf 1 gesetzt wird. Jetzt wird eine Aktion zufällig gewählt. Nach jedem Spiel wird  $\epsilon$  etwas gesenkt, um dann mit einer Wahrscheinlichkeit von  $\epsilon$  eine zufällige Aktion zu wählen und sonst den besten Zug, den der Agent anhand seiner Police auswählt. Nach 90% der Spiele ist  $\epsilon$  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.

Die  $\epsilon$ -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
- $\pi$  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  $r(s, a) = 1$ .

$\gamma$  können wir selbst wählen und wir werden hier später verschiedene Werte testen.  $\pi$  wollen wir optimieren, also ein optimales  $\pi^*$  finden.

## 2.6 Value-Funktion

Die Value-Funktion gibt für einen Zustand s und eine Police  $\pi$  die zu erwartende Belohnung an. Also die Belohnung, die vorraussichtlich in diesem und allen weiteren Zeitschritten gesammelt werden kann. Hier benötigen wir den Diskontierungsfaktor  $\gamma$ , 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  $\infty$ , 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):

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

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

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

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:

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

Hier ist  $\alpha$  die Lernrate und  $n$  steht für die  $n$ -te Aktion die ausgeführt wird.

## 3. Die Methoden

### 3.1 Q-Tabelle

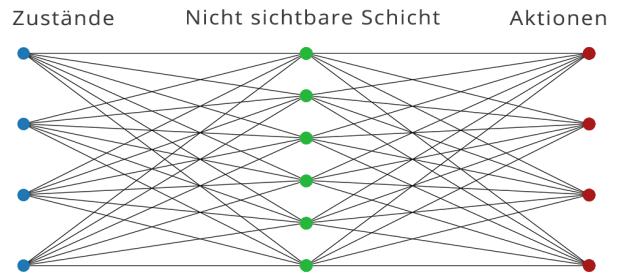
Die einfachste Möglichkeit den SARSA Algorithmus umzusetzen ist die sogenannte Q-Tabelle. Betrachten wir dafür unser Beispiel der Police aus Figure 6, dann haben wir 16 Zustände und 4 Aktionen. Damit ist unsere Q-Tabelle eine  $16 \times 4$  Matrix. Wir können uns nun für einen State  $i$  die  $i$ -te 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  $\epsilon$ -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 naheliegenste Möglichkeit ein neuronales Netz zu verwenden, ist die Funktion  $Q(s, a)$  mit einem neuronalen Netz zu approximieren. Hierfür benutzen wir einfach einen verdeckten 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:

$$F(v) = \sum_h \mathbb{P}(h|v)E(v, h) + \frac{1}{\beta} \sum_h \mathbb{P}(h|v)\log(\mathbb{P}(h|v)) \quad (8)$$

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.  $\beta$  ist ein Parameter, den man wählen kann. Durch ihn kann man die Wirkung der zweiten Summe abschwächen.

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 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

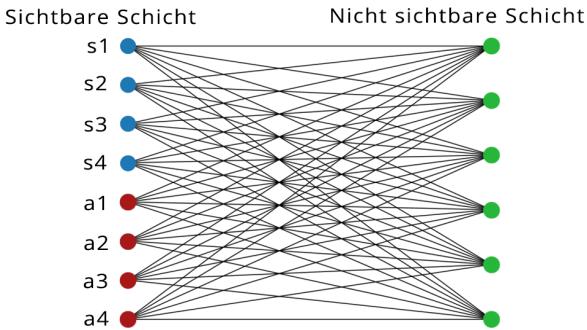


Figure 5. Beschränkte Boltzmann Maschine

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:

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

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

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

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

$$\begin{aligned} w_{sh}^+ &= \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n))s_i \langle h \rangle \\ w_{ah}^+ &= \alpha(r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n))a_i \langle h \rangle \end{aligned}$$

#### 3.5 Tiefe Boltzmann Maschine Resultate mit Quanten Annealing

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

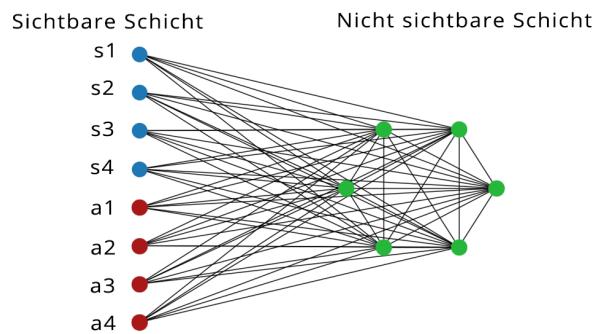


Figure 6. Tiefe Boltzmann Maschine

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:

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

$$- \frac{1}{\beta} \sum_k \langle h_k \rangle \log(\langle h_k \rangle) + (1 - \langle h_k \rangle) \log((1 - \langle h_k \rangle)) \quad (13)$$

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 Neuronen 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.

Eine naheliegende Idee wäre jetzt (12) mit der durchschnittlichen Energie zu 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. Allerdings ist die durchschnittliche Energie nicht identisch mit der Energie, die wir aus den versteckten Variablen  $h$  berechnen. Der Grund dafür sind die quadratischen Terme im QUBO. Dazu ein kleines Beispiel:

Sei ein QUBO gegeben, durch zwei Variablen die mit einer Kante verbunden sind. Diese Kante hat den Wert -1. Führen wir jetzt zwei Annealing Durchläufe aus, bekommen wir zum Beispiel die Ergebnisse (1,1) und (0,0). Damit berechnen wir die Aktivierung von den versteckten Variablen mit:

$$\frac{(1, 1) + (0, 0)}{2} = (0.5, 0.5) \quad (15)$$

Es ergibt sich mit (12) ohne die Entropy die Energie:

$$0.5 * 0.5 * -1 = -0.25 \quad (16)$$

Im Vergleich dazu, hat das erste Ergebnis (1,1) die Energie -1 und das zweite Ergebnis (0,0) die Energie 0 und damit wäre die durchschnittliche Energie -0.5.

Das heißt wir bekommen mit der durchschnittlichen Energie unserer Messergebnisse ein ähnliches aber leicht verschiedenes Ergebniss zu (12), welches die freie Energie der Boltzmann Maschine beschreibt. In verschiedenen Tests zeigte sich auch, dass der Algorithmus mit der durchschnittlichen Energie aus den Messergebnissen ähnliche Ergebnisse liefert, jedoch ein wenig schlechter als mit der direkten Berechnung durch (12). Wir wollen daher die Variante mit der durchschnittlichen Energie im weiteren nicht näher betrachten.

### 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.

$$H_0 = - \sum_i \sigma_i^x \quad (17)$$

$$H_1 = \sum_{i,j} J_{ij} \sigma_i^z \sigma_j^z + \sum_i h_i \sigma_i^z \quad (18)$$

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

$$H(t) = (1-t)H_0 + tH_1, t \in [0, 1] \quad (19)$$

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.

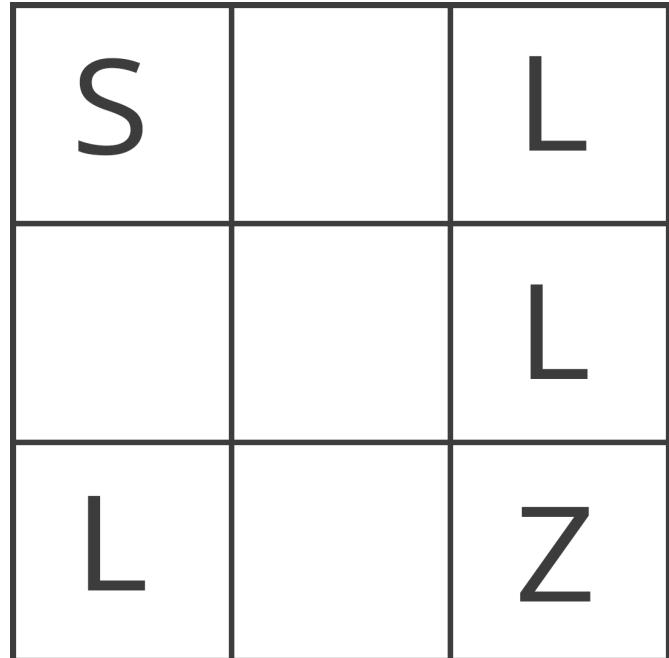
Wie wir dann auch in unseren Experimenten sehen können, brauchen wir (13) für diese Version nicht mehr, da wir mit den Parametern im simulierten Annealing schon steuern können, wir sehr die Variablen richtung 0 oder 1 tendieren. Damit wird die Zeile einfach überflüssig und wir sehen auch in den Experimenten,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

## 4. Resultate

### 4.1 Prinzipien des Testens

Wir wollen hier festlegen, wie wir testen, was wir testen, 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  $\epsilon$ -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  $\epsilon$ -Greedy-Methode an ihre Grenzen, da wir hier mit einer zufälligen Police am Anfang nicht mehr das Ziel finden können.



**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 Policien umgehen und etwas dazulernen können, ohne altes zu verlernen.

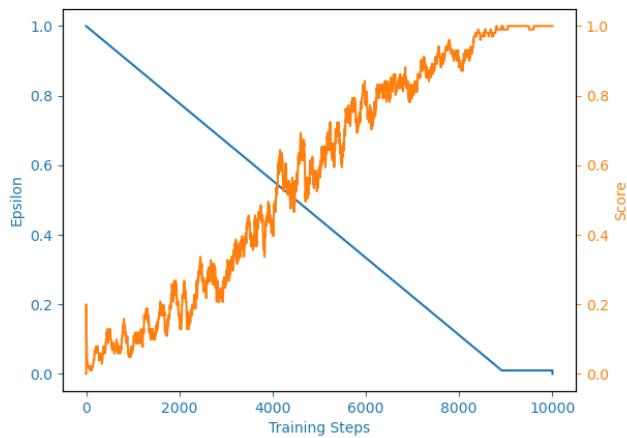
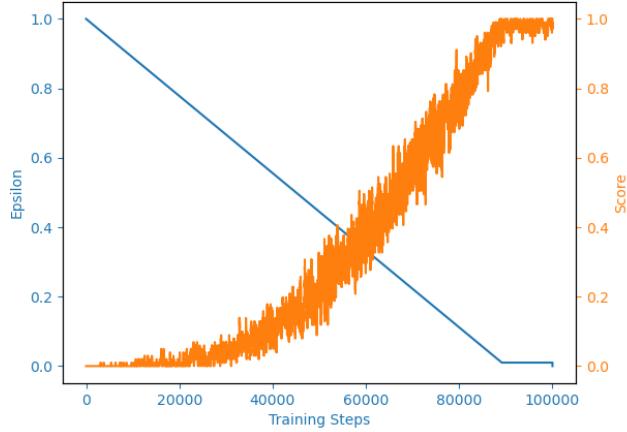
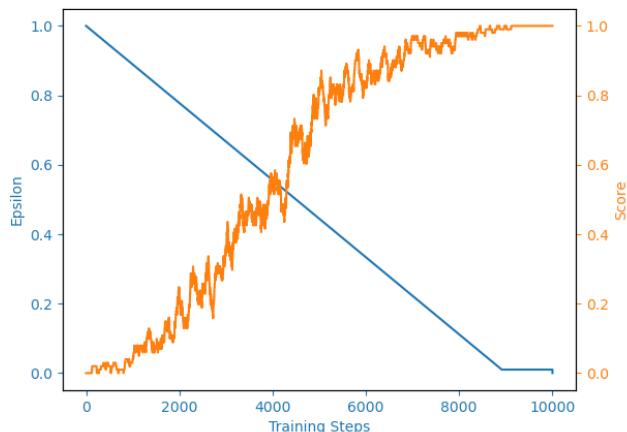
Die zweite Möglichkeit ist, komplett von der  $\epsilon$ -Greedy-Methode abzusehen und eine Schleife über alle Felder zu machen und für jedes Feld jede Aktion zu testen und daraus zu lernen. Das funktioniert besser, braucht aber für große Spielfelder sehr viel Rechenzeit, bis wir die richtige Lösung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

Die Resultate lassen wir durch einen Graphen anzeigen. Bei der  $\epsilon$ -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 durch zufällige Aktionen zumindestens einmal die Lösung gefunden werden kann.

Testen wir erst mit Standardparametern. Der blaue Graph ist hier übrigens  $\epsilon$ . In den meisten Vergleichen, wenn wir verschiedene Parameter testen wollen, lassen wir  $\epsilon$  weg. Die  $\epsilon$  Kurve sieht aber immer gleich aus, es sei denn, wenn wir verschiedene Startfelder nacheinander wählen.

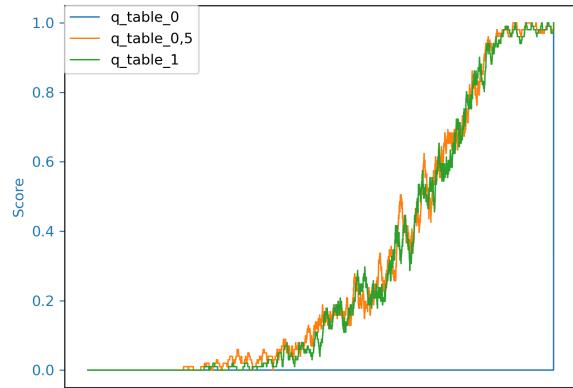
**Figure 8.** 3x3 Q-Tabelle**Figure 9.** 5x5 Q-Tabelle**Figure 10.** 8x8 Q-Tabelle

Das 5x5 Feld ist sehr voll mit Löchern, daher sorgen

hier geringe  $\epsilon$  immer noch für einige falsche Züge und damit für niedrigere Punkte.

Allerdings fällt auf, dass wir bei dem 5x5 Feld schon die Anzahl der Spiele deutlich erhöhen müssen, um überhaupt das Ziel zu finden. Eine elegantere Alternative ist, die Q-Tabelle nicht mit 0 zu initialisieren, sondern z.B. mit 0.5. Es passiert dann Folgendes: Sobald der Agent eine Aktion macht, die keine Belohnung bekommt, wird der Wert der Q-Tabelle reduziert. Daher macht der Agent beim nächsten Mal eine andere Aktion. So werden vor allem weit entfernte Ecken des Spielfeldes erreicht.

Im Folgenden die Ergebnisse mit den Initialwerten 0, 0.5 und 1:

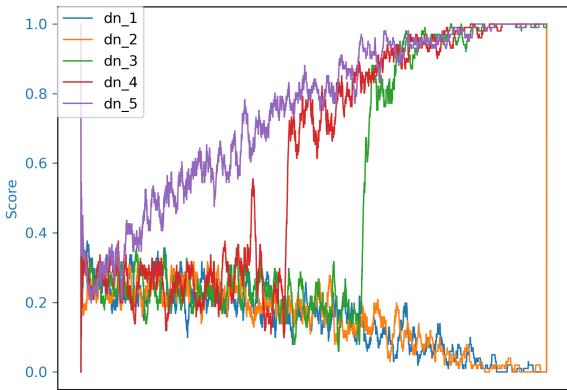
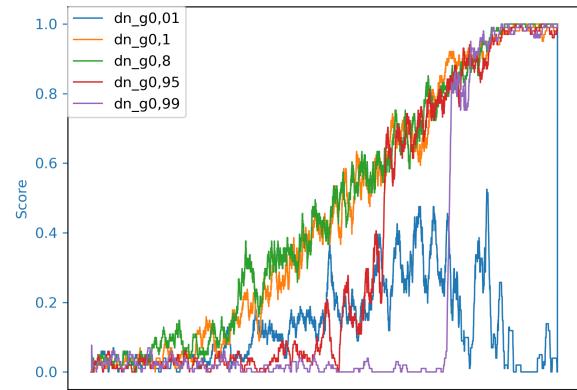
**Figure 11.** 5x5 Q-Tabelle

Während der Agent mit dem Initialwert 0 das Ziel nie findet und garnicht anfangen kann zu lernen, finden die beiden anderen Versionen die Lösung extrem schnell.

Übrigens wird für die Grafik bei jeder Methode am Schluss der Wert 1 in die Scores eingefügt. Das führt dazu, dass immer die richtige Scala (0 bis 1) auf der y-Achse zu sehen ist. Dadurch entsteht in der letzten Grafik die farbige blaue Linie rechts im Bild.

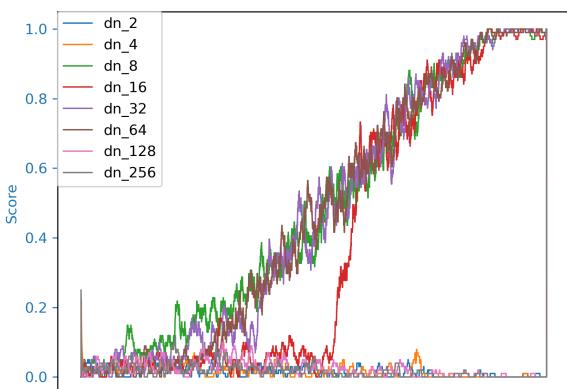
#### 4.3 Einfaches neuronales Netz Resultate

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 12.** 2x2 Neuronales Netz: Neuronen**Figure 14.** 3x3 Neuronales Netz:  $\gamma$ 

Wir brauchen also mindestens 3 Neuronen, um auf die richtige Lösung zu kommen. Das ist interessant, da wir 4 Aktionen und 4 Zustände haben. Wir können das Problem also mit weniger Neuronen als möglichen Aktionen lösen. Ab 5 Neuronen gibt es keine klaren Unterschiede mehr bei dem kleinen Spielfeld.

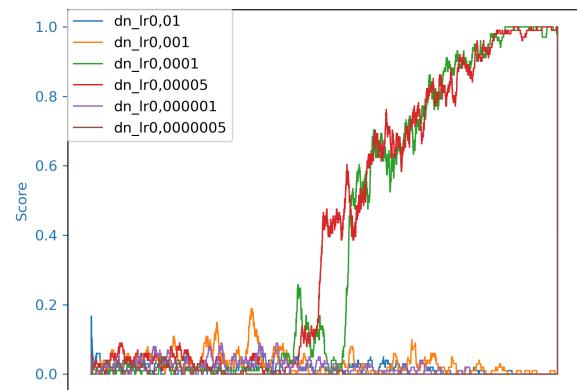
Beim 3x3 Feld brauchen wir gleich deutlich mehr Neuronen. Zwischen 8 und 64 Neuronen bekommen wir richtige Ergebnisse.

**Figure 13.** 3x3 Neuronales Netz: Neuronen

Als nächstes testen wir mit 16 Neuronen weiter und vergleichen verschiedene Werte für  $\gamma$ .

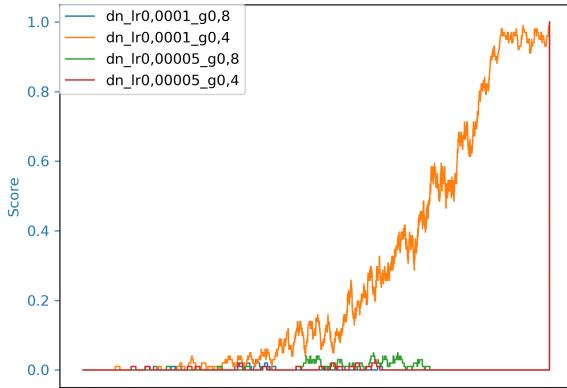
Am besten haben hier die Werte 0,1 und 0,8 für  $\gamma$  funktioniert. Wir bleiben also im weiteren bei dem Wert 0,8. Dieser wird auch häufig in der Literatur verwendet. Behalten wir aber im Auge, dass eventuell niedrigere Werte besser funktionieren könnten.

Zuletzt wollen wir die Lernrate testen. Eine hohe Lernrate kann dazu führen, dass das Netz hin und her schwankt und sich nicht einpendelt. Eine zu niedrige Lernrate kann dazu führen, dass die 10000 Spiele nicht ausreichen, um das richtige Ergebnis zu lernen.

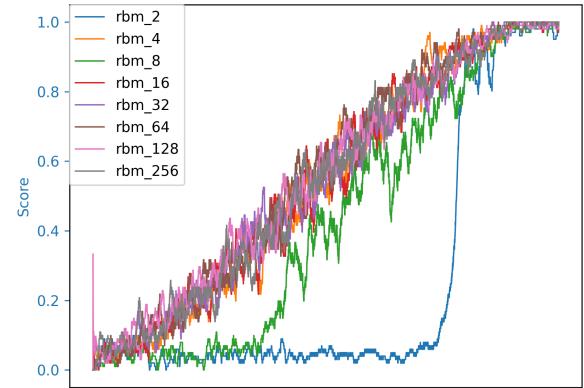
**Figure 15.** 3x3 Neuronales Netz: Lernrate

Hier sieht man also, dass das Netz wirklich sehr empfindlich auf die Lernrate reagiert. Ein Wert von 0,00005 scheint in jedem Fall gut zu sein. Aber genau wie bei  $\gamma$  kann man hier noch andere Werte in der Nähe testen.

Als letztes wollen wir das 4x4 Feld testen (Figure 1). Dazu kombinieren wir verschiedene gute Parameter aus den vorigen Testläufen.



**Figure 16.** 4x4 Neuronales Netz: Gamma und Lernrate

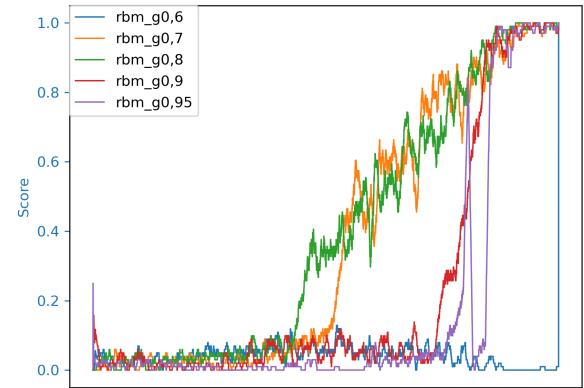


**Figure 17.** 3x3 Beschränkte Boltzmann maschine: Neuronen

Hier schneidet der Agent mit  $\gamma = 0,4$  und einer Lernrate von 0,0001 am besten ab. Meine erste Vermutung war hier, dass es sich um einen Zufall handelt und es davon abhängt, wie oft der Agent in den ersten 2000-3000 Spielen zufällig das Ziel findet. Durch eine Wiederholung des Experiments konnten wir das jedoch ausschließen. Mit anderen Parametern konnte zwar manchmal auch die beste Police gelernt werden, allerdings waren die gerade genannten Parameter immer am schnellsten.

Wir sehen hier also, je mehr Neuronen umso besser. Das ist schon ein erster spannender Unterschied zu dem einfachen neuronalen Netz, wo zu vielen Neuronen schlecht waren. Allerdings führen viele Neuronen hier gleich zu einem deutlichen Anstieg an Rechenzeit. Daher wollen wir im Folgenden 16 Neuronen für das 3x3 Feld verwenden und nur bei größeren Feldern auf mehr Neuronen umsteigen.

Als nächstes wollen wir verschiedene Werte für  $\gamma$  testen.



**Figure 18.** 3x3 Beschränkte Boltzmann maschine:  $\gamma$

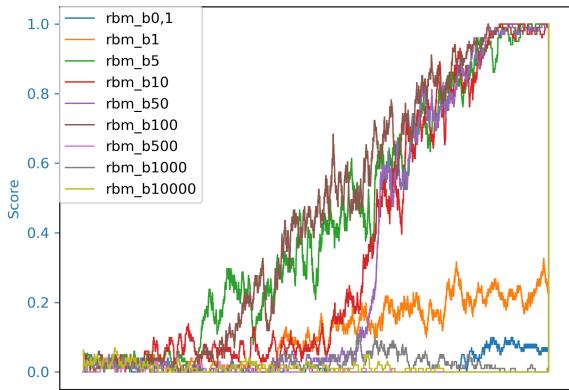
#### 4.4 Beschränkte Boltzmann Maschine

Bei der beschränkten Boltzmann Maschine haben wir 2 weitere Parameter. Zum einen den Wert  $\beta$ , der die Entropy abschwächt und zum anderen initialisieren wir die Gewichte am Anfang zufällig mit dem Erwartungswert 0 und einer Standardabweichung. Die Standardabweichung hat tatsächlich großen Einfluss auf das Ergebniss, da eine hohe Standardabweichung dazu führt, dass der Agent auf verschiedenen leeren Feldern sehr hohe Belohnungen oder Bestrafungen erwartet. Diese Vorannahmen müssen dann erst wieder verlernt werden, was zu einem langsameren Lernverhalten führt.

Als erstes testen wir wieder, wie viele versteckte Neuronen wir brauchen:

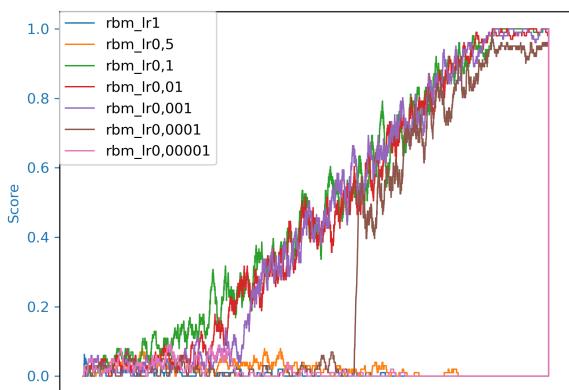
Die besten Ergebnisse bekommen wir also mit einem  $\gamma$ -Wert von 0,7 oder 0,8. Daher bleiben wir im weiteren wie in (Crawford et al., 2019) bei 0,8.

Wir wollen nun  $\beta$  vergleichen. Mit  $\beta$  schwächen wir die zweite Summe in (8) ab. Dadurch können wir steuern, wie stark die Werte der versteckten Neuronen richtung 0 oder 1 gesteuert werden.

**Figure 19.** 3x3 Beschränkte Boltzmann maschine:  $\beta$ 

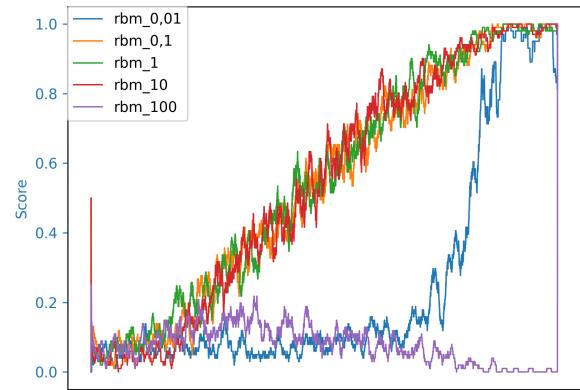
Wir sehen hier, dass bei Werten zwischen 5 und 100 richtige Polices gefunden werden. Es scheint also sinnvoll zu sein, diesen Entropy-Faktor stark abzuschwächen, aber nicht ganz wegzulassen. Wir verwenden im Folgenden einen mittleren Wert von 20.

Nun schauen wir uns die Lernrate an:

**Figure 20.** 3x3 Beschränkte Boltzmann maschine: Lernrate

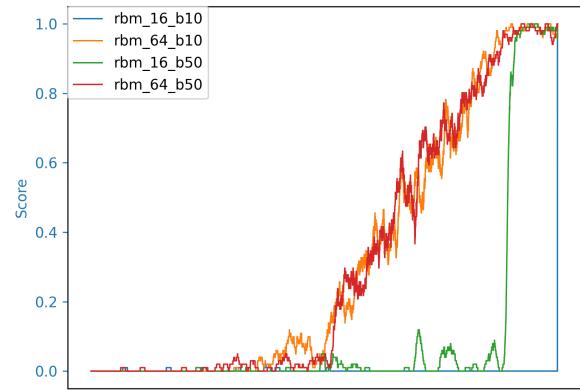
Hier sehen wir gute Ergebnisse zwischen 0,1 und 0,001. Daher nehmen wir auch hier den mittleren Wert von 0,01 für folgende Experimente.

Als letztes testen wir die Standardabweichung für die Initialisierung der Gewichte.

**Figure 21.** 3x3 Beschränkte Boltzmann maschine: Standardabweichung

Wir sehen also, dass zu große oder zu kleine Werte zu schlechteren Ergebnissen führen und verwenden genau wie in (Crawford et al., 2019) die übliche Standardabweichung von 1.

Da wir bei den Feldern aus (Figure 1) und (Figure 2) noch sehr gute Ergebnisse bekommen und sogar verschiedene Parameter alle zuverlässig das richtige Ergebnis liefern, wollen wir hier gleich zu dem 5x5 Spielfeld übergehen. Hier testen wir sowohl mit 16 als auch mit 64 Neuronen und mit  $\beta$  von 10 und 50:

**Figure 22.** 5x5 Beschränkte Boltzmann maschine: Neuronen und  $\beta$ 

Hier sehen wir zum ersten Mal, wie 16 Neuronen nicht mehr ausreichen. Der  $\beta$ -Wert dageben scheint in einer recht großen Spannweite gut zu funktionieren. Wobei anscheinend der Wert 50 sogar besser ist, da hier selbst mit 16 Neuronen noch die richtige Police gefunden wurde.

#### 4.5 Tiefe Boltzmann Maschine Resultate mit Quanten Annealing

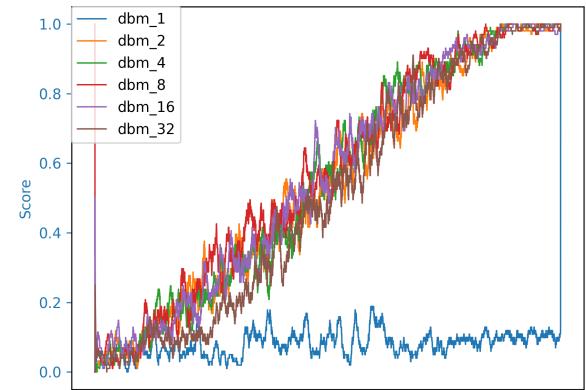
Die Neuheit bei diesem Algorithmus ist vor allem, dass wir die Aktivierung der versteckten Neuronen mit simuliertem Quanten Annealing berechnen. Daher kommen hier 3 neue Parameter hinzu: Anzahl der Annealing-Durchläufe, Anzahl der Schritte beim Annealing und die Annealing Temperatur, die aus einem Startwert und einem Endwert besteht.

Die Anzahl der Annealing-Durchläufe und die Anzahl der Schritte beim Annealing sollten möglichst hoch gewählt werden, da wir so genauere Ergebnisse bekommen. Gleichzeitig sind diese beiden Parameter für einen starken Anstieg der Rechenzeit verantwortlich. Daher setzen wir beide für unsere Testdurchläufe für die anderen Parameter auf 100. So bekommen wir auch noch in akzeptabler Zeit Ergebnisse. Später für die Vergleiche zwischen den verschiedenen Algorithmen können wir noch höhere Werte nehmen. Um diese beiden Werte zu testen, kann man 100 mal die Aktivierung der versteckten Neuronen berechnen und aus diesen 100 Vektoren die Standardabweichung berechnen. Diese liegt zwischen 0.01 und 0.03 mit den gerade genannten Parametern.

Die Annealing Temperatur sollte so gewählt werden, dass wir gute Ergebnisse bekommen, aber nicht immer das Richtige. Wenn wir immer das bestmögliche Ergebniss berechnen würden, wären unsere versteckten Neuronen immer 1 oder 0 und damit hätten wir keine sauberen Übergänge zwischen den Lernphasen sondern "Brüche" wenn die versteckten Neuronen den Zustand wechseln. Dies führt zu schlechteren Ergebnissen. Wir starten hier mit dem Startwert 2 und dem Endwert 20, wie auch in (Crawford et al., 2019).

Dafür können wir in diesem Verfahren  $\beta$  weglassen, da wir die Entropy über die Annealing Temperatur steuern können. Konkreter gesagt, wenn wir die Temperatur sehr gut wählen, bekommen wir immer das optimale Ergebniss und das wiederum führt zu einer maximalen Entropy. Diese Vermutung werden wir später noch mit Tests bestätigen.

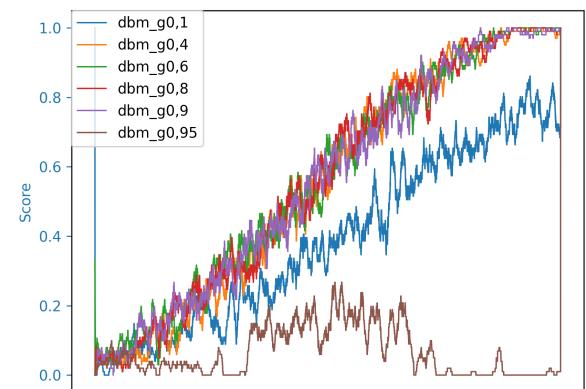
Starten wir unsere Tests mit der Anzahl an Neuronen:



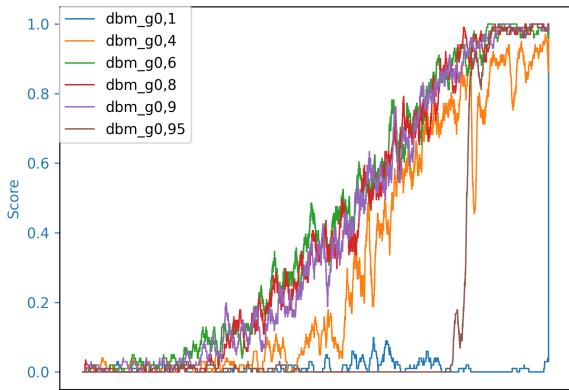
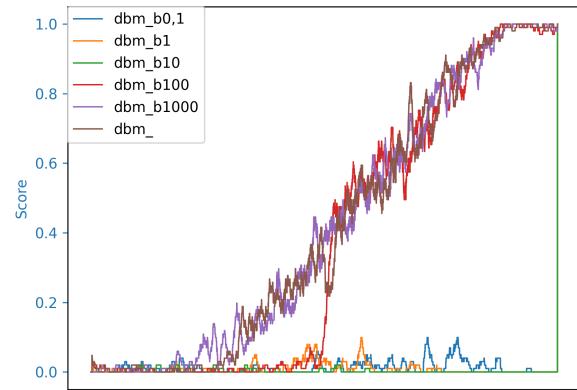
**Figure 23.** 3x3 Tiefe Boltzmann maschine: Neuronen

Hier zeigt sich bereits die Stärke dieses Verfahrens. Während wir beim einfachen neuronalen Netz mindestens 8 Neuronen gebraucht haben, brauchen wir hier nur 2. Die beschränkte Boltzmann Maschine konnte dieses Beispiel zwar auch mit 2 Neuronen lösen, fand die richtige Police allerdings sehr spät. Nach oben scheint es hier keine Grenze zu geben, obwohl die Version mit 32 Neuronen minimal langsamer reagiert. Eine höhere Anzahl von Neuronen führt hier bereits zu sehr langen Rechenzeiten, weswegen wir diese hier nicht testen.

Als nächstes schauen wir uns wieder  $\gamma$  an. Da die Ergebnisse hier nur für extreme Randwerte schlechter werden, testen wir es auch gleich noch auf dem 4x4 Spielfeld:



**Figure 24.** 3x3 Tiefe Boltzmann maschine: $\gamma$

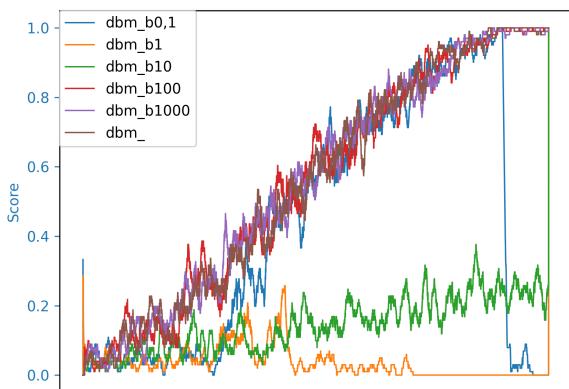
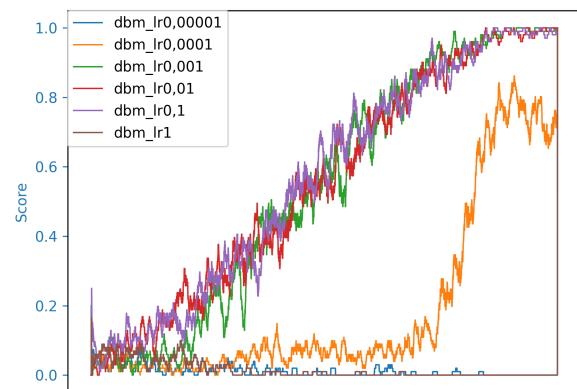
**Figure 25.** 4x4 Tiefe Boltzmann maschine:  $\gamma$ **Figure 27.** 4x4 Tiefe Boltzmann maschine:  $\beta$ 

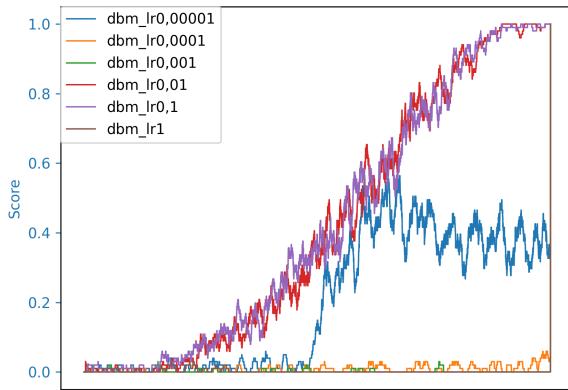
Hier sind die Ergebnisse ähnlich wie bei der beschränkten Boltzmann Maschine. Zwischen 0,6 und 0,9 sind die Resultate gut. Daher bleiben wir hier wie auch in (Crawford et al., 2019) bei  $\gamma = 0.8$ . Wir können hier im Hinterkopf behalten, dass größere Spielfelder eventuell mit höheren  $\gamma$  Werten besser funktionieren. Anschaulich macht das Sinn, da der erwartete Gewinn über eine längere Kette von Aktionen bis zum Startfeld erhalten bleiben soll. Man kann diesen Effekt bei dem Vergleich zwischen dem 3x3 und dem 4x4 Feld auch erkennen.

Als nächstes testen wir  $\beta$ . Wie auch zuvor schon erwähnt, gibt es hier eine Variante  $dbm\_$ , welche den Entropy Term komplett weglässt.

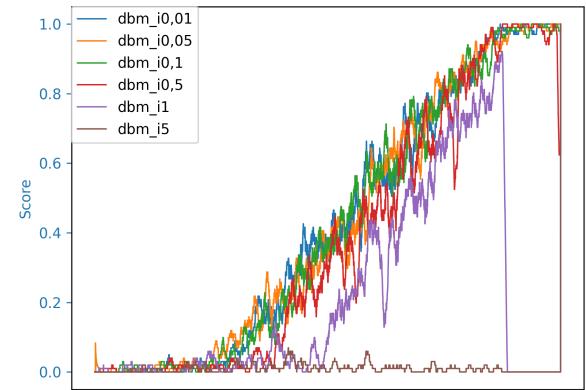
Wir sehen also hier, je höher der Wert für  $\beta$  umso besser die Ergebnisse.  $\beta$  ist dabei der Faktor, durch den der Entropy-Term geteilt wird. Also ein sehr hoher Wert für  $\beta$  reduziert den Einfluss von diesem Term. Werte wie 1000 sind dabei so groß, dass der Term quasi keinen Einfluss mehr hat. Man sieht hier auch, dass die Variante  $dbm\_$  ohne die Entropy vergleichbare Ergebnisse erzielt wie sehr hohe  $\beta$  Werte. Infolgedessen können wir ab hier den Entropy-Term komplett weglassen.

Als nächstes testen wir die Lernrate:

**Figure 26.** 3x3 Tiefe Boltzmann maschine:  $\beta$ **Figure 28.** 3x3 Tiefe Boltzmann maschine:  $\beta$



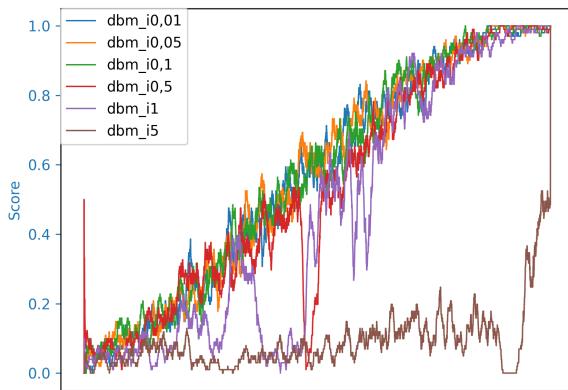
**Figure 29.** 4x4 Tiefe Boltzmann maschine:  $\beta$



**Figure 31.** 4x4 Tiefe Boltzmann maschine:  $\beta$

Am besten schneiden hier die Werte 0,1 und 0,01 ab. Wir entscheiden uns daher im weiteren einen mittleren Wert von 0,05 zu benutzen.

Schauen wir uns nun die Standardabweichung an, mit der wir die Gewichte initialisieren:



**Figure 30.** 3x3 Tiefe Boltzmann maschine:  $\beta$

Hier scheinen vor allem kleine Werte gut zu funktionieren. Wir wählen im Folgenden 0,05, da dieser Wert gute Ergebnisse liefert.

Als letztes wollen wir noch die Annealing Temperatur testen. Dies gestaltet sich von allem Parametern am schwierigsten. Zum einen müssen wir hier drei Werte im Auge behalten: Den Startwert, den Endwert und das Verhältnis von beiden. Zum anderen liefern hier viele Werte gute Ergebnisse. Daher schauen wir uns gleich das 5x5 Spielfeld an. Hier ist es aber so schwierig, überhaupt das Ziel zu finden, dass die Ergebnisse ein wenig vom Zufall abhängen. Daher wollen wir hier keine Graphen von einzelnen Durchläufen zeigen, sondern die Azahl der richtige Polisen die mit den drei oben genannten Werten gefunden wurden, in einer Tabelle aufzeigen:

Startwert	Richtige Polisen
0,001	0 / 1
0,01	4 / 7
0,1	5 / 13
1	6 / 12
10	11 / 13
100	5 / 8
1000	2 / 3

Endwert	Richtige Policen
0,2	0 / 1
0,5	0 / 2
1	0 / 6
2	1 / 2
5	0 / 2
10	4 / 5
20	2 / 2
50	2 / 3
100	7 / 9
200	1 / 1
500	1 / 2
1000	7 / 10
5000	1 / 2
10 000	6 / 9
100 000	1 / 1

Verhältnis	Richtige Policen
1 zu 2	4 / 6
1 zu 5	3 / 7
1 zu 10	7 / 13
1 zu 50	1 / 4
1 zu 100	8 / 12
1 zu 1000	2 / 5
1 zu 10000	6 / 7
1 zu 100000	1 / 2
1 zu 1000000	1 / 1

Insgesamt haben wir also 57 Strategien getestet und dabei 33 mal die richtige Police gefunden. Das entspricht 58%. Allerdings ist es garnicht einfach, die Ergebnisse zu interpretieren. Wir testen also am besten mehrere Varianten von guten Parametern mit mehreren Durchläufen.

Bei den Startwerten haben wir die besten Ergebnisse für 10. Bei den Endwerten scheinen tendenziell größere Werte besser zu funktionieren. Auch bei dem Verhältnis, sind größere Verhältnisse in der Regel besser.

Wir wollen daher einen Testlauf mit dem Startwert 10 und dem besten Verhältnis 1 zu 10000 machen. So kommen wir auf die Parameter [10, 100000]. Einen zweiten Testlauf wollen wir mit dem besten Endwert machen. Dieser ist nicht ganz klar, wir wählen hier 100, da dieser Wert öfter getestet wurde. Was auch bei den Tests oft richtige Policen lieferte war [1, 100]. Wählen wir also diese Werte. Und zuletzt wollen wir noch die Werte aus (Crawford et al., 2019) mit [2, 20] testen.

Der Gewinner von diesen drei Vergleichen ist [2, 20] mit 27 von 30 richtigen Policen. Platz 2 ist [10, 100000] mit 24 von 30. Und Letzter ist [1, 100] mit 20 von 30. Beachte aber das auch alle drei über den 58% liegen,

die wir in den gemischten Parametern erreicht haben.

Hier wäre es sicherlich noch spannend weiter zu testen. Vor allem fällt hier auf, das teilweise sehr kleine Verhältnisse wie bei [2, 20] gut funktionieren. Dann sind größere Verhältnisse wie 1 zu 1000 plötzlich schlecht und 1 zu 10000 funktioniert wieder gut. Hier scheint es keine lineare Abhängigkeit zu geben.

Fassen wir nun die besten Ergebnisse von allen Methoden zusammen und testen dann die Methoden gegeneinander.

#### 4.6 Die besten Parameter

Bevor wir die Ergebnisse von allen Methoden im Vergleich zeigen, hier eine Auflistung der besten Parameter aus den bisherigen Experimenten. Rbm steht für beschränkte Boltzmann Maschine und Qbm für die tiefe Boltzmann Maschine mit Quanten Annealing:

Parameter	Q-Tabelle	Einfaches Netz	Rbm	Qbm
Lernrate	0.1	0.0001	0.01	0.05
Neuronen	-	16 - 64	64 - $\infty$	8 - 32
$\gamma$	0.8	0.4	0.8	0.8
$\beta$	-	-	50	-
Initialwert	0.5	-	0	0
Std	-	-	1	0.05
Ann. runs	-	-	-	100 - $\infty$
Ann. sweeps	-	-	-	100 - $\infty$
Ann. Temp	-	-	-	[2, 20]

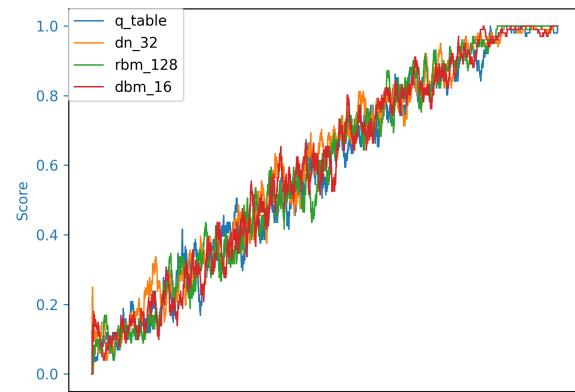


Figure 33. 3x3 Alle Methoden

#### 4.7 Vergleich der Methoden gegeneinander

Wir untersuchen als erstes 8 verschiedene Spielfelder in aufsteigender Größe und testen jeweils alle 4 Methoden. Die Spielfelder haben dabei die Größen 2x2, 3x3, 3x5, 4x4, 5x5, 6x6, 8x8 und 8x8. Dabei sind die beiden 8x8 Spielfelder nur noch mit wenigen Löchern versehen.

Bei der Anzahl an Neuronen haben wir hier noch etwas erhöht, da tendenziell bei größeren Spielfeldern mehr Neuronen gebraucht werden. Hier kommt das einfache Netz mit 32 Neuronen daher, die beschränkte Boltzmann Maschine mit 128 und die Quanten Boltzmann Maschine mit 16. Die Annealing Runs und Sweeps werden auf 200 gesetzt, für eine höhere Genauigkeit. Höhere Werte fürs Annealing sind leider aufgrund von der Rechenzeit auf meinem PC nicht möglich. Alle anderen Werte die benutzt wurden können 1 zu 1 auf der oben genannten Tabelle abgelesen werden.

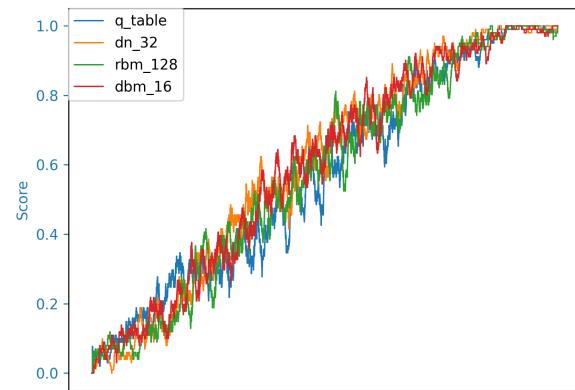


Figure 34. 3x5 Alle Methoden

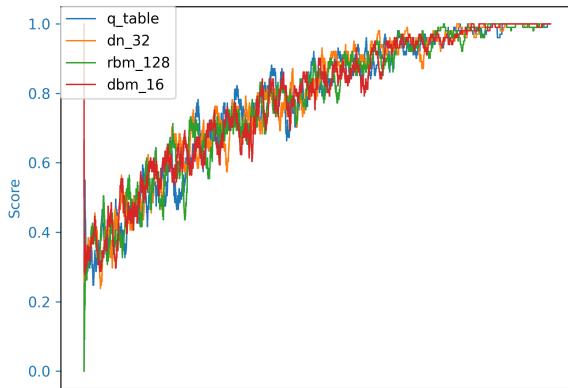


Figure 32. 2x2 Alle Methoden

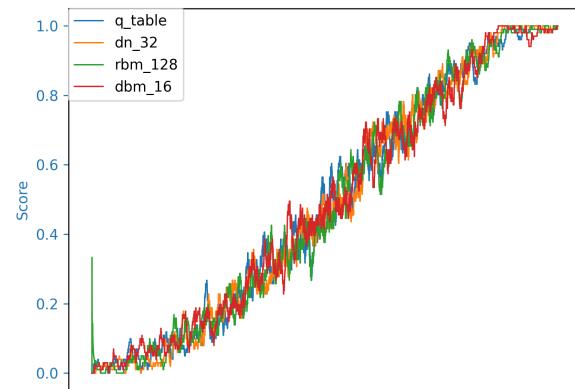
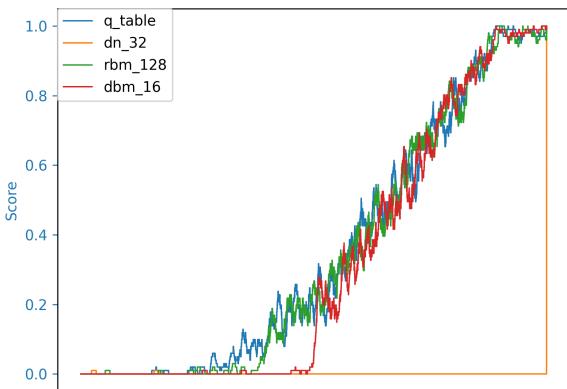
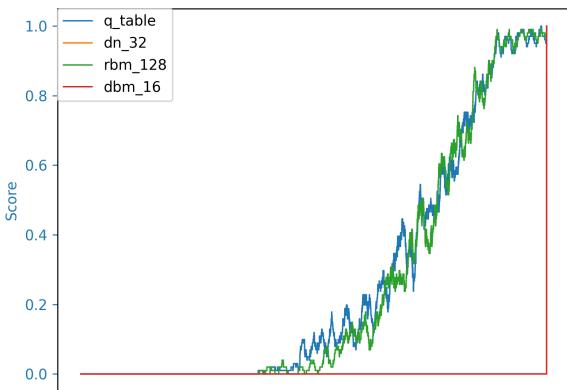


Figure 35. 4x4 Alle Methoden

Bis hierher funktionieren alle Methoden also perfekt.

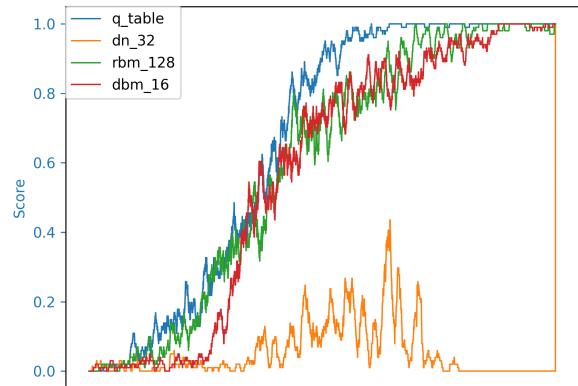
**Figure 36.** 5x5 Alle Methoden

Hier wird es also zum ersten Mal interessant. Das einfache Netz findet hier nicht mehr die Lösung, während alle anderen gut abschneiden. Man sieht bei der Quanten Boltzmann Maschine hier einen leicht verzögerten Start im Vergleich zu den anderen Netzen.

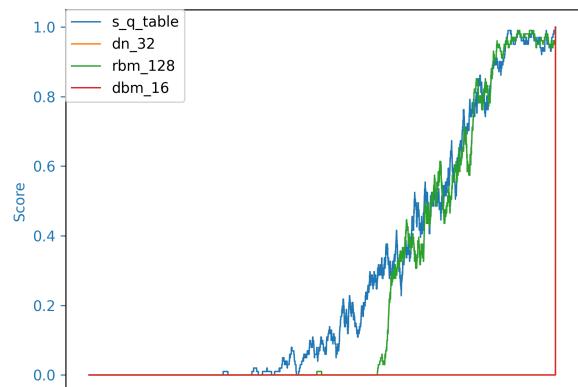
**Figure 37.** 6x6 Alle Methoden

Dieses Beispiel ist bereits so kompliziert, dass es quasi unmöglich ist, mit zufälligen Zügen das Ziel zu finden. Überraschenderweise ist hier die beschränkte Boltzmann Maschine sehr gut darin, automatisch in unbekannte Gebiete zu wandern. Eine mögliche Erklärung dafür ist, dass bei vier Aktionen pro Feld die Wahrscheinlichkeit hoch ist, dass mindestens eine davon und damit auch das gesamte Feld einen positiven Gewinn verspricht. Durch mehrmaliges Ausführen der Aktion, wandern alle Werte gegen 0 und damit haben Felder die noch nicht oft besucht wurden, einen höheren zu erwartenden Gewinn. Warum dies bei der Quanten Boltzmann Maschine nicht funktioniert ist nicht ganz klar. Zum einen könnte dies an der niedrigeren Initialisierung der Gewichte liegen. Zum

anderen an der Ungenauigkeit bei der Aktivierung der versteckten Neuronen die durch das Annealing entsteht.

**Figure 38.** 8x8 einfach Alle Methoden

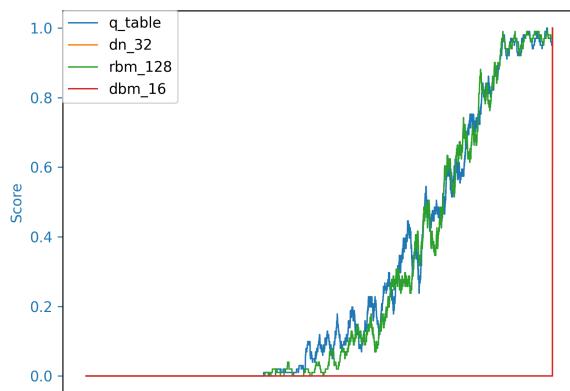
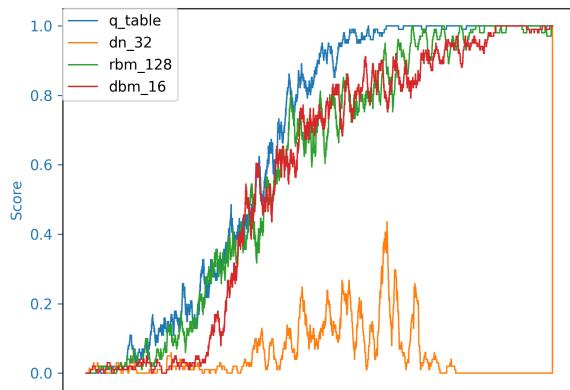
Das 8x8 Feld hat nur 3 Löcher und dadurch werden hier immer richtige Lösungen gefunden. Das einfache Netz mit nur 32 Neuronen scheint allerdings nicht in der Lage zu sein, so viele Felder abzubilden.

**Figure 39.** 8x8 schwer Alle Methoden

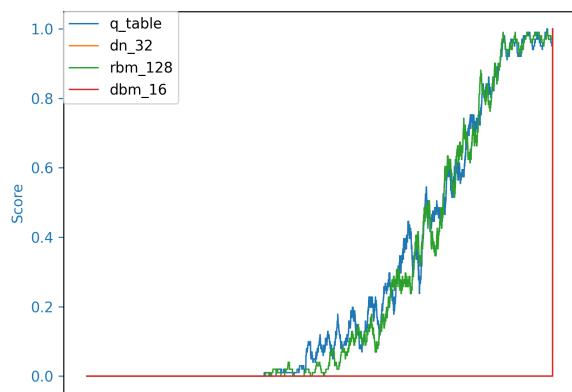
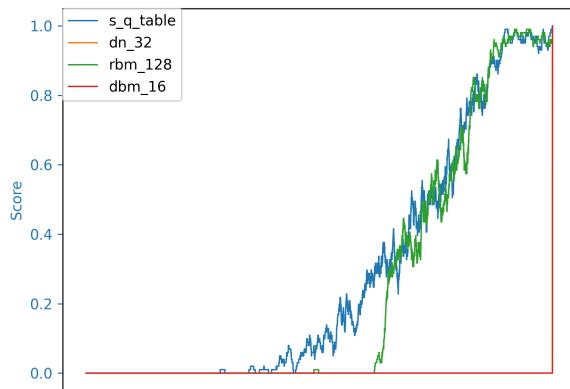
Hier sehen wir wieder ein ähnliches Verhalten wie beim 6x6 Feld.

Als nächstes wollen wir das  $\epsilon$ -Greedy-Verfahren verfeinern, in dem wir von verschiedenen Startpositionen starten, die sich nach und nach vom Ziel weg bewegen.

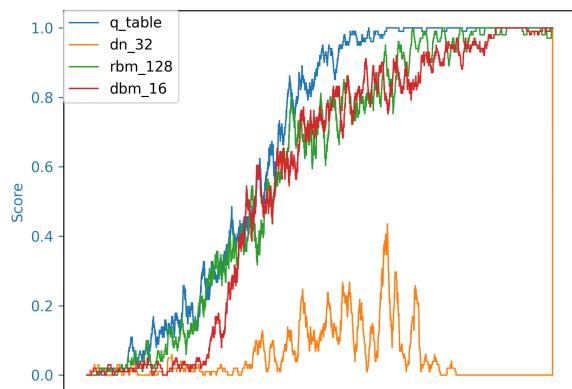
Da hier sehr gute Ergebnisse erwartet werden, prüfen wir nur die schwersten Felder (6x6 und 8x8) und bauen zusätzlich ein 11x11 Feld, welches einem Labyrinth entspricht mit nur einem möglichen Weg zum Ziel.

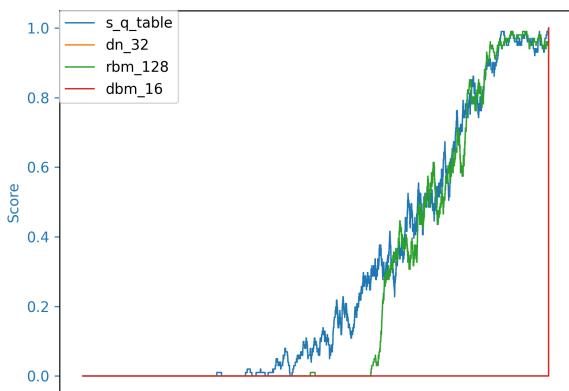
**Figure 40.** 6x6 Alle Methoden**Figure 41.** 8x8 einfach Alle Methoden

großen Feldern sind. Dafür gehen wir mit einer Schleife über alle Felder und führen von da jeweils jede Aktion einmal aus und lernen daraus. Hier findet also jede Methode zum Ziel und hat so die Möglichkeit, die richtige Police zu finden. Wir starten dann alle 10 Durchläufe ein Spiel von jedem Feld und messen so, von wie vielen Feldern das Ziel erreicht wird. Dies ist unser Score. Insgesamt gehen wir 1000 mal über alle Felder.

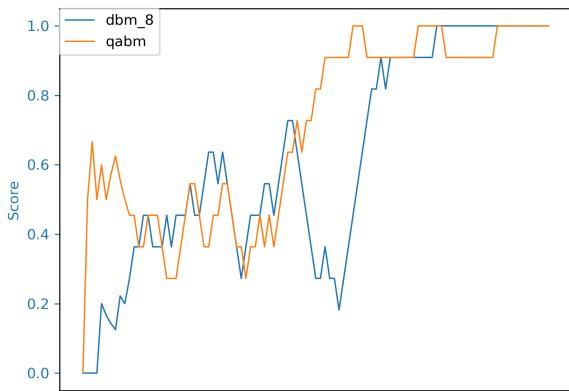
**Figure 43.** 6x6 Alle Methoden**Figure 42.** 11x11 schwer Alle Methoden

Nun wollen wir schauen, wie stabil die Netze bei

**Figure 44.** 8x8 einfach Alle Methoden

**Figure 45.** 11x11 schwer Alle Methoden

Als letztes wollen wir noch schauen, ob dieses Verfahren auch mit einem echten Quanten Annealer von DWave funktionieren würde. Da wir begrenzte Rechenzeit haben, testen wir nur das 2x2 Spielfeld und reduzieren die Anzahl der Spiele auf 100. Wir wählen dazu die gleichen Parameter wie beim simulierten Annealing. Nur die Anzahl der Annealing Durchläufe setzen wir auf 50. Die Annealing Zeit lassen wir bei  $20\mu\text{s}$  wie in der Voreinstellung. Dieses Ergebnis vergleichen wir mit dem simulierten Annealing.

**Figure 46.** 11x11 schwer Alle Methoden

Wir sehen hier, beide Algorithmen finden bei dem kleinen Spielfeld schon nach 100 Spielen die richtige Police. Dieses Verfahren scheint also mit einem echten Quantencomputer zu funktionieren. Ob dies auch bei schwierigeren Beispielen funktioniert, müsste noch getestet werden.

## 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