Facharbeit
Langfassung
im Schuljahr 2024/2025

Graphische Umsetzung des Würfelspiels Kniffel und Einbindung einer Künstlichen Intelligenz

von
Hardardt, Jonas

geschrieben im Informatik Leistungskurs
unter der Betreuung von Volker Wagner

dem Europa-Gymnasium in Wörth vorgelegt am
13.05.2024

1 Kurzfassung

Diese Arbeit beschäftigt sich mit der Implementierung und graphischen Umsetzung des Würfelspiels Kniffel. Des Weiteren wird versucht, unter Anwendung und Entwicklung einer Künstlichen Intelligenz, dem Computer eine möglichst gute Strategie für das Spielen von Kniffel beizubringen. Die Annäherung an eine möglichst gute Strategie mithilfe einer Künstlichen Intelligenz (KI) wird aufgezeigt.
Auch wenn sich die Aufgabe der Entwicklung einer KI für ein, auf den ersten Blick sehr einfaches und naives, Würfelspiel wie Kniffel, sehr einfach erscheint, so bemerkt man sehr schnell, dass selbst ein so einfaches Spiel schon sehr komplex ist. So kann man sich während des Spiels in einem von der vielen Zustände befinden. Berechnet man diese Anzahl der Zustände, so stellt man fest, dass unter der Annahme, dass es ungefähr 100 verschiedene Punktestände gibt und man seine Strategie und Risikobereitschaft auch abhängig von den Punktzahlen der Mitspieler macht, so erhält man folgende Anzahl an Zuständen:

$$S(n) = (2^{13} \cdot 100)^n \cdot 252 \cdot 3 \tag{1}$$

mit n als Anzahl der Spieler, mit dem ersten Faktor als alle möglichen noch offenen Kategoriekombinationen, mit dem zweiten Faktor als alle möglichen unterschiedlichen Punktzahlen in diesen Kategorien, dem dritten Faktor als alle möglichen Würfelergebnisse der fünf Würfel und mit dem letzten Faktor, als mögliche verbleibende Anzahl an Restwürfen, da man bis zu dreimal würfeln darf.

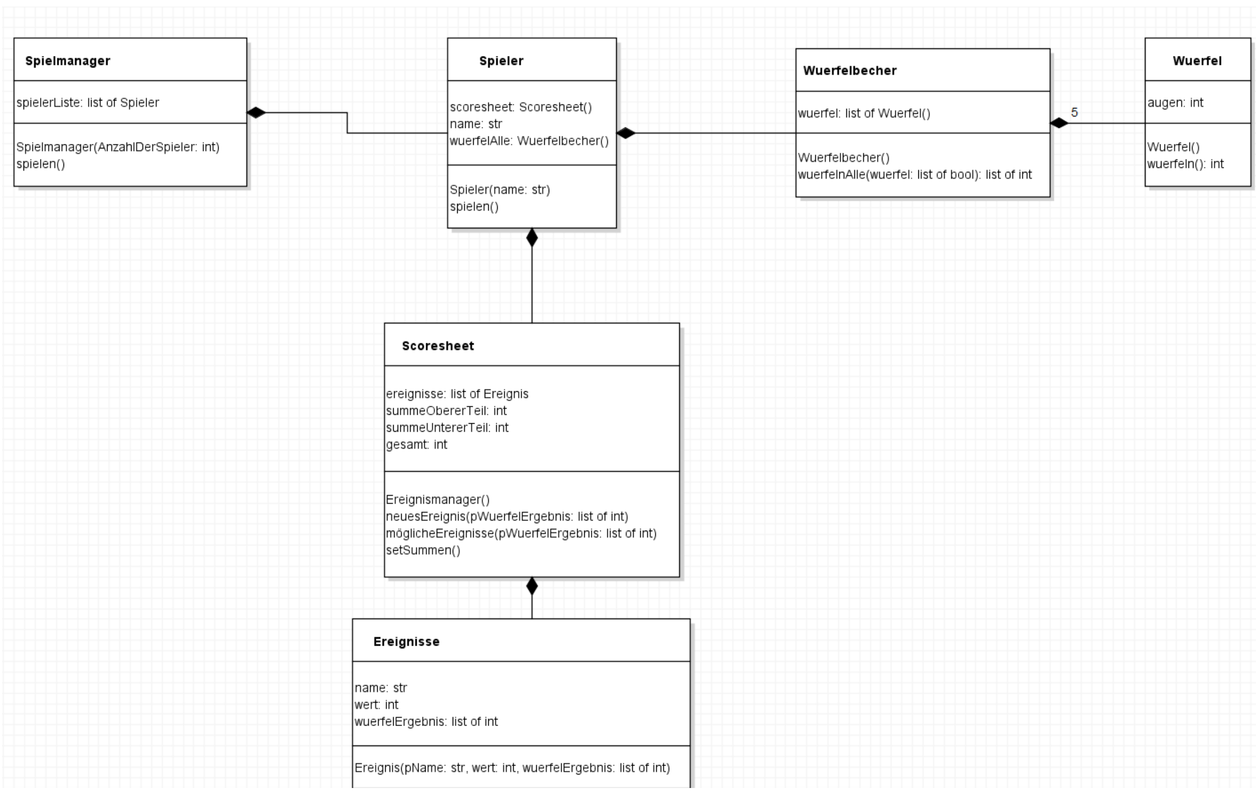
Die Anzahl an möglichen Zuständen beläuft sich schon nur bei zwei Spielern auf ungefähr $5 \cdot 10^{14}$ Zustände. Daher ist es mir in meiner Arbeit nicht gelungen, eine höhere Punktzahl als 160 zu erreichen, wobei diese Punktzahl unter dem Durchschnitt von 245 Punkten liegt. Dennoch lernt die KI Zusammenhänge, performt stabil und es ist eine positive Entwicklung sichtbar.
Aufgrund dieser hohen Anzahl an Zuständen, habe ich die Betrachtung der Spielstände der anderen Spieler vorerst nicht beachtet, da bereits ohne diese Betrachtung eine hohe Anzahl an Zuständen vorhanden ist und die Anzahl an Iterationen aufgrund der Rechenleistung beschränkt ist.
Abschließend hebt die Arbeit die Probleme und Schwierigkeiten hervor und zeigt mögliche Problemlösungen und Verbesserungsmöglichkeiten für die Zukunft.

Inhaltsverzeichnis

2 Implementierung des Spiels Kniffel

2.1 Umsetzung in Python

Im Rahmen der Unterrichtseinheit *Objektorientiertes Programmieren* bekamen wir Schüler die Möglichkeit, ein Spiel unserer Wahl objektorientiert umzusetzen und zu implementieren. Ich habe mich für Kniffel entschieden und dieses implementiert. Dazu habe ich mir folgende Klassen überlegt:



(a)

Abbildung 1

Der Spielmanager kümmert sich um die Verwaltung der Spieler. Die Klasse Spieler verwaltet die Würfel eines Spielers sowie sein Kniffelblockblatt, das Scoresheet. Das Scoresheet verwaltet alle möglichen Ereignisse und die erreichten Punktzahlen. Der Würfelbecher verwaltet alle Würfel und kümmert sich um das Werfen aller oder ausgewählter Würfel.

2.2 Graphik

Die Graphik wurde mit der Pythonbibliothek tkinter umgesetzt. Zu Beginn des Spiels werden die Namen der Spieler abgefragt und es folgen zwei Fenster. Das linke ist für das Kategoriezuweisen und Würfeln mittels Buttons zuständig. Auf der linken Seite wird der Kniffelblock angezeigt. Außerdem werden Buttons ausgegraut, wenn Kategorien schon genommen wurde und man sieht auf dem Kniffelblock eine Vorschau, welche Kategorie welche Punktzahl bringen wird, abhängig vom aktuellen Würfelergebnis.

3 Künstliche Intelligenz

3.1 Was ist eine Künstliche Intelligenz?

Spätestens nach dem Launch von ChatGPT hat jeder schon ein Mal von einer Künstlichen Intelligenz gehört. Doch was ist eine Künstliche Intelligenz (KI)? Eine KI beschreibt menschliche Fähigkeiten einer Maschine um Probleme lösen zu können, wie beispielsweise das Lernen[1]. So kann eine Maschine mit einer KI auf ihre Umwelt reagieren und sich dieser anpassen.

3.2 Machine Learning

Um Probleme lösen zu können und um sich seiner Umwelt anpassen zu können, eignet sich die Verwendung von Machine Learning. Bei Machine Learning oder Maschinelles Lernen (ML) handelt es sich um ein Teilgebiet der Künstlichen Intelligenz. Machine Learning ist eine Möglichkeit, Systemen Lernen beizubringen und dadurch ihre Performance, in Bezug zu einer Problemstellung, verbessern zu können. Dies erreichen sie, indem sie Beziehungen und Muster selbstständig erkennen können[2, S. 685]. Dabei wurde der Computer nicht explizit für das Problem programmiert, sondern lernt mit Hilfe von Daten[3]. Möchte man ein Problem in der Informatik mithilfe einer Künstlichen Intelligenz und Machine Learning lösen, so muss man sich für eine der drei Machine Learning Modelle entscheiden.

3.3 Supervised Learning

Supervised Learning hat die Aufgabe herauszufinden wie Ein- und Ausgabe zusammenhängen, um dann die Ausgabe vorhersagen zu können[4]. Hierfür werden die Daten in Trainings- und Testdaten aufgeteilt und müssen davor manuell gelabelt werden, dass heißt die zu einer Eingabe passende Ausgabe definieren. Daher nennt man dieses Lernen auch Überwachtes Lernen[3], es findet beispielsweise bei der Bilderkennung Anwendung.

3.4 Unsupervised Learning

Beim Unsupervised Learning versucht man im Kontrast zum vorherigen Modell, keine Ausgaben vorherzusagen, sondern vielmehr eine Struktur in den Eingaben zu finden. Es muss daher niemand vorher die Daten generieren beziehungsweise definieren und es ist daher unüberwacht[3, 5].

3.5 Reinforcement Learning

Im Reinforcement Learning gibt es einen Agenten, der lernt mit seiner Umgebung zu interagieren. Von dieser Umgebung erhält er den *Zustand/state s* und kann *Aktion/action a* ausführen. Für diese Aktionen beziehungsweise Zustände erhält der Agent eine entsprechende *Belohnung/reward r*. Da sich das Verhalten des Agenten abhängig von dem Reward verändert nennt man dieses Lernen, Bestärkendes Lernen[6, S. 1-4]. Der große Vorteil hierbei ist, dass sich diese Art des Lernens besonders für Spiele eignet, da es dort sehr viele unterschiedliche Zustände und Aktionen gibt und ein Labeling, welches für Supervised Learning nötig ist, sehr umständlich wäre. Vielmehr liegt es in der Natur der Spiele, dass man für bestimmte Aktionen und Zustände Belohnungen erhält. Auch durch die direkte Interaktion des Agenten mit seiner

Umwelt, ist die Umsetzung und der Aufwand deutlich geringer als bei den anderen beiden Ansätzen und macht es überhaupt erst möglich, eine KI für Spiele zu entwickeln.

3.6 Q-Learning

3.6.1 Einführung

Q-Learning ist ein möglicher Algorithmus, um modellfreies Reinforcement Learning umzusetzen. Dabei interagiert der Agent mit der Umgebung und kann in einem bestimmten Zustand s eine Aktion a ausprobieren. Führt er eine Aktion a aus, so erhält dieser eine Belohnung r oder Strafe, welche eine negative Belohnung darstellt[7, S. 279]. Das Ziel des Agenten ist es, eine Strategie zu finden, die die erwarteten erhaltenen Belohnungen maximiert[8, S. 27] [7, S. 279], und das nicht nur kurzfristig, sondern eben auch langfristig.

3.6.2 Bestimmung des Q-Werts

Um diese Strategie zu erhalten, müssen wir die zu erwartenden Belohnungen betrachten. Diese Gesamtsumme der zu erwartenden Belohnung ist jedoch nicht einfach nur die Summe der folgenden Belohnungen, sondern die in Zukunft erhaltenen Belohnungen werden mit einem Faktor γ rabattiert und dadurch berücksichtigt. Durch diese rabattierten Belohnungen kann man erreichen, dass Belohnungen für spätere Zustände und Aktionen schon bei früheren Entscheidungsfragen berücksichtigt werden, auch wenn sie einen geringeren Einfluss haben, als direkt erhaltene Belohnungen [7, S. 280].
Diese Summe aus zu erwartenden Belohnungen wird im Folgenden *Rendite R* genannt. Die Rendite ist, wie bereits erklärt, nicht einfach die Summe der in Zukunft zu erwartenden Belohnungen:

$$R_t = r_t + r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T = \sum_{n=0}^{\infty} r_{t+n} \tag{2}$$

zum Zeitpunkt t, $r_t \in R$ und dem letzten Zeitpunkt T ($T = \infty$)

Sondern die Rendite berechnet sich aus der Summe der rabattierten Belohnung, wobei weiter in der Zukunft liegende Belohnungen stärker rabattiert werden:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \gamma^4 r_{t+4} + \cdots + r_T = \sum_{n=0}^{\infty} \gamma^n r_{t+n} \tag{3}$$

mit Rabattfaktor $\gamma \in (0;1]$ [9]

Bisher können wir nur die erwartete Rendite berechnen, aber können noch nichts über einen Q-Wert aussagen. Bei Markow-Entscheidungsproblemen(MDP) gibt es eine endliche Anzahl an Zuständen und Aktionen und die Wahrscheinlichkeit in einen nächsten Zustand zu kommen, hängt nur von dem aktuellen Zustand ab. Bei solchen MDP lässt sich die Wertefunktion nach der Bellman Gleichung aufstellen[10, 11, 12]:

$$V_\pi(s) = E_\pi\{r_t + \gamma V(s_{t+1}\} \tag{4}$$

Diese Formel berechnet einen Wert V der abhängig von der genutzten Strategie π und dem Zustand s ist.

Hier fällt auf, dass der zweite Summand ebenfalls durch diese Gleichung bestimmt werden kann und dass somit diese Berechnung rekursiv ist. Da der erste Summand die direkte Belohnung ist und man somit immer die direkte Belohnung mit der rabattierten Summe des nächsten Zustands addiert, kann man den gesamten rechten Term mit der Gleichung (2) ersetzen[11, 12]. Es folgt daher:

$$V_\pi(s) = E_\pi\{r_t + \gamma V(s_{t+1}\} = E_\pi\{R_t\} = E_\pi\{\sum_{n=0}^{\infty} \gamma^n r_{t+n}\} \tag{5}$$

Im Moment können wir ausschließlich für einen Zustand s einen Wert berechnen, jedoch nicht den Wert in einem bestimmten Zustand s eine Aktion a auszuwählen. Betrachten wir nun den Q-Wert, so stellen wir fest, dass man diesen ähnlich wie den Zustands-Wert $V(s)$ berechnen kann:

$$Q_\pi(s,a) = E_\pi\{r_t + \gamma max_{a'} Q(s',a')\} \tag{6}$$

Gleichung (6) berechnet daher den Q-Wert, indem es zu dem rabattierten maximal möglichen Q-Werts des nächsten Zustand und der besten Aktion in diesem neuen Zustand $\gamma max_{a'} Q(s',a')$ die direkt erhaltene Belohnung r_t für die Aktion a addiert[11, 13].
Möchten wir nun die beste Strategie π finden, so müssen wir immer nur die Aktion a wählen, die in einem Zustand s den höchsten Q-Wert hat [11, 13]:

$$\pi^{opt}(s) = argmax_a Q(s,a)[11, S.46] \tag{7}$$

3.6.3 Q-Tabellen

Eine recht einfache Möglichkeit Q-Learning umzusetzen, ist die Q-Werte in einer Q-Tabelle zu speichern, in der alle möglichen Zustände zeilenweise und alle möglichen Aktionen spaltenweise aufgetragen sind. Der Q-Wert in der jeweiligen Zelle beschreibt daher den Q-Wert des in dieser Zeile stehenden Zustands s und der in dieser Spalte stehenden Aktion a. Vorteilhaft hierbei ist, dass die Implementierung vergleichsweise leicht erscheint, jedoch nur eine geringe Anzahl an möglichen Zuständen und Aktionen vorherrschen kann. Sonst kann die Q-Tabelle sehr groß geraten und nur noch schlecht gespeichert werden.
Zu Beginn des Lernens ist die Tabelle mit Nullen initialisiert und die erste Aktion wird daher zufällig gewählt. Ändern sich die Werte in dieser Tabelle, so besteht die Gefahr, dass der Agent, der mit der Umgebung interagiert, immer die gleichen Aktionen wählt, da diese einen Q-Wert > 0 besitzen. Es werden dadurch Explorationsstrategien nötig, um sicherzustellen, dass weiterhin neue Aktionen ausprobiert werden[13, 14].
Eine mögliche Explorationsstrategie ist die ε-greedy Strategie bei der, mit einer Wahrscheinlich ε eine zufällige Aktion ausgewählt wird und mit einer Wahrscheinlichkeit von $1 - \varepsilon$ die Aktion mit dem höchsten Q-Wert ausgewählt wird[15].
Um die Q-Werte in der Tabelle zu aktualisieren, kann folgende Formel verwendet werden:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r_t + \gamma max_{a'} Q(s',a') - Q(s,a)) \; [16, S.249] \tag{8}$$

wobei $\alpha \in (0;1]$ die Lernrate darstellt, um starke Schwankungen zu vermeiden und r_t die direkte Belohnung darstellt.

Der neue Q-Wert berechnet sich eben dadurch, dass zum aktuellen Q-Wert (erster Summand) ein Wert, um die Lernrate α verkleinert, erhöht oder erniedrigt wird, je nachdem, ob die Belohnung positiv oder negativ war. Dieser Wert wird durch die Differenz aus der Summe des maximalen Q-Werts des nächsten Zustands und der erhaltenen Belohnung und dem aktuellen Q-Wert gebildet.

3.6.4 Deep Q-Networks

Bei den Q-Tabellen ist vor allem die Anzahl an Zuständen und Aktionen der limitierende Faktor. Deswegen kann man statt diesen Tabellen auch Neuronale Netze, beziehungsweise Deep Q-Networks (DQN) verwenden. Gerade bei einer großen Anzahl an Zuständen sind diese deutlich effizienter und ermöglichen so erst, die Verarbeitung einer deutlich größeren Anzahl an Daten. Ziel dieses DQN ist es für einen Zustand s die Q-Werte für die möglichen Aktionen $a \in A(s)$ auszugeben[13, 14].

$$f(s) = [Q(s,a_1), Q(s,a_2), Q(s,a_3), Q(s,a_4), \dots] \tag{9}$$

mit f als Neuronales Netz, welches einen Zustand s als Eingabe erhält und Q-Werte für mögliche Aktionen $a \in A(s)$ ausgibt.

Auch hier ist die Verteilung der Werte zu Beginn zufällig, doch mit der Zeit werden einzelne Q-Werte angepasst, sodass die zugehörigen Aktionen präferiert werden. Deswegen ist es auch hier nötig eine Explorationsstrategie zu verwenden. Die Verwendung der ε-greedy-Strategie ist auch hier möglich, jedoch kann man auch zu einer anderen greifen. Die Boltzmann Exploration stellt hierbei eine Alternative dar. Bei dieser wird eine Wahrscheinlichkeitsverteilung basierend auf den Q-Werten, beziehungsweise auf Basis der zu erwartenden Belohnungen, erstellt, wobei Aktionen mit einem höheren Q-Wert auch eine höhere Wahrscheinlichkeit haben, gewählt zu werden. Des Weiteren gibt es einen Parameter T $]0;\infty[$, die Boltzmann-Temperatur. Wird T sehr hoch gewählt, so sind die Wahrscheinlichkeiten der Aktionen nahe zu gleich, wird T jedoch klein gewählt, so haben die Aktionen mit einem höherem Q-Wert auch eine höhere Wahrscheinlichkeit, ausgewählt zu werden[17, 18].
Um Deep Q-Networks zu trainieren, können wir nicht einfach Gleichung (8) verwenden, da wir den größten Q-Wert des nächsten Zustands nicht kennen. Um Stabilität im Lernprozess zu gewährleisten, benötigen wir nicht nur ein DQN, sondern zwei. Wir haben das Ziel das erste DQN, das Q Neural Network (QNN) zu trainieren und benötigen hierzu noch ein Target Neural Network (TNN). Dieses Target Network benutzen wir dazu, uns den größten Q-Wert des nächsten Zustands zu generieren. Zusammen mit der direkt erhaltenen Belohnung, bildet der größte Q-Wert des nächsten Zustands den Target Q-Wert für den aktuellen Zustand und die gewählte Aktion. Um nun Back-Propagation betreiben zu können, also die Gewichte des QNN anzupassen, benötigen wir den sogenannten Loss (Verlust). Dieser Loss berechnet sich aus dem Mean-Square-Error, also dem Quadrat aus der Differenz vom vorhergesagten Q-Wert des QNN, dem Target Q-Wert und dem erhaltenen Reaward[19]:

$$Loss = MSE(Q_{QNN}(s_t,a), r_t + \gamma * max_{a'} Q_{TNN}(s_{t+1},a')) = (Q_{QNN}(s_t,a) - (r_t + \gamma * max_{a'} Q_{TNN}(s_{t+1},a')))^2 \tag{10}$$

Durch das Quadrat wird erreicht, dass sowohl positive als auch negative Fehler gleichermaßen bestraft werden. Außerdem werden Fehler > 1 durch das Quadrieren deutlich stärker bestraft als kleinere Fehler < 1.
Um noch mehr Stabilität zu gewährleisten, betreiben wir Experience Replay, dass heißt wir speichern uns schon besuchte Zustände, sowie die ausgeführte Aktion, den folgenden Zustand und die erhaltene Belohnung. Nun geben wir im Trainingsprozess nicht nur die neueste Aktion und Zustand in das QNN hinein, sondern wählen auch noch andere zufällig aus dem Speicher aus und geben diese ebenfalls in den Trainingsprozess. Wir erreichen dadurch, dass unser QNN sich dabei nicht nur ausschließlich an die neuen Datensätze anpasst, sondern auch weiterhin

noch auf die früheren Daten möglichst genau reagieren kann. Damit dieser Speicher nicht ewig groß wird, legt man eine Obergrenze fest. Wird diese überschritten, so werden Datensätze gelöscht.
Bisher haben wir nur das QNN angepasst und das TNN nicht verändert. Dies führt dazu, dass dessen Vorhersagen nach einigen Iterationen womöglich nicht mehr richtig sind. Deshalb kopiert man nach einer gewissen Anzahl an Iterationen das QNN und diese Kopie wird dann das neue TNN. Man adaptiert allerdings das TNN nicht bei jedem Trainingsschritt bei dem man das QNN updatet, da man eine gewisse Kontinuität bei den Vorhersagen der Q-Werte des TNN verfolgt[19].

3.7 Policy Deep Neural Networks

3.7.1 Einleitung

Anstatt eine Strategie indirekt mit Q Neural Networks zu finden, indem man die Aktionen, die die höchsten Q-Werte haben, wählt, kann man auch direkt ein Deep Neural Network (DNN) darauf trainieren, die beste Strategie zu lernen. Dass heißt wir geben den aktuellen Zustand s_t hinein und bekommen eine Wahrscheinlichkeitsverteilung über alle möglichen Aktionen:

$$f(s_t) = [P(s_t, a_1), P(s_t, a_2), P(s_t, a_3), \dots] \tag{11}$$

abhängig vom Zeitpunkt t, vom Zustand s und der Aktion a, erhält man für jede Aktion die Wahrschinlichkeit P.

Eine Aktion wählen wir nun, indem wir mithilfe der Wahrscheinlichkeitsverteilung randomisiert auswählen. Dadurch ist hier keine Explorationsstrategie nötig, da jede Aktion jederzeit ausgewählt werden kann, da nur die Wahrscheinlichkeit der vorraussichtlich besseren Aktionen steigt, jedoch die anderen Aktionen weiterhin auswählbar sind. Deren Wahrscheinlichkeit der schlechteren Aktionen geht gegen Null, wird jedoch nicht Null und daher werden diese auch noch selten ausgewählt. Die Wahrscheinlichkeit der besten Aktion in einem Zustand geht gegen Eins.
Die Auswahl der Aktionen in einem bestimmten Zustand ist also nicht deterministisch, sondern bleibt randomisiert.

3.7.2 Policy Gradient Optimierung

Jetzt, da wir die Strategiefunktion, beziehungsweise die Wahrscheinlichkeitsfunktion mithilfe eines Neuronalen Netzes darstellen, stellt sich natürlich die Frage wie wir dieses trainieren und dadurch eben anpassen. Das Problem ist nun, dass wir keinen Target Wert mehr haben, mit welchem man vergleichen könnte und sich diesem durch die Differenz zwischen IST Wert und Target/SOLL Wert annähern könnte. Deshalb müssen wir die erwartete Rendite anders definieren:

$$R_{\sum,\theta} = \sum_{s \in S} \mu(s) \sum_{a \in A(s)} \pi_\theta(s,a) Q(s,a) [20,\ 21] \tag{12}$$

mit θ als Parameter bzw. Gewichte des Neuronalen Netzes

Diese Gleichung beschreibt die zu erwartende Gesamtrendite abhängig von den Parametern des Neuronalen Netzes. So beschreibt die erste Summe die Summe der Wahrscheinlichkeiten

sich in gewissen möglichen Zustand s zu befinden und die zweite Summe die Wahrscheinlichkeiten Aktionen in diesem Zustand zu wählen, multipliziert mit dem zugehörigen Q-Wert.
Möchten wir nun θ verändern, sodass sich die Rendite maximiert, so müssen wir den Gradienten, Ableitung einer Funktion mit mehr als einer Variable, berechnen:

$$\nabla_\theta R_{\sum,\theta} = \sum_{s\in S} \mu(s) \sum_{a\in A(s)} Q(s,a)\nabla_\theta \pi_\theta(s,a)[20,\ 21] \tag{13}$$

Wir erweitern nun mit $\frac{\pi_\theta(s,a)}{\pi_\theta(s,a)}$

$$\nabla_\theta R_{\sum,\theta} = \sum_{s\in S} \mu(s) \sum_{a\in A(s)} \pi_\theta(s,a) Q(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)}[20,\ 21] \tag{14}$$

mit $\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)}$ als Ableitung von $\nabla_\theta log(\pi_\theta(s,a))$

$$\nabla_\theta R_{\sum,\theta} = \sum_{s\in S} \mu(s) \sum_{a\in A(s)} \pi_\theta(s,a) Q(s,a)\nabla_\theta log(\pi_\theta(s,a))[20,\ 21] \tag{15}$$

$$= E[Q(s,a)\nabla_\theta log(\pi_\theta(s,a))][20,\ 21] \tag{16}$$

Sodass wir dann die Aktualisierung von θ wie folgt berechnen können:

$$\theta_{neu} = \theta_{alt} * \alpha \nabla_\theta R_{\sum,\theta}[20,\ 21] \tag{17}$$

mit θ als Parameter des Neuronalen Netzes, der Lernrate α und der erwarteten Rendite $R_{\sum,\theta}$. Der Gradient stellt dabei die Richtung dar, in welche θ verändert werden muss, um die Rendite zu maximieren.
Aktualisieren wir mit dieser Formel das Neuronale Netz, so erhalten wir eine optimale Strategie beziehungsweise Policy.

3.8 Actor Critic

3.8.1 Diskussion

Q-Learning und Policy Networks haben beide ihre Vor- und Nachteile. So ist ein großer Nachteil des Q-Learning, dass die Auswahl der Aktion deterministisch ist, dass heißt, dass in gleichen Zuständen die gleiche Aktion ausgewählt wird. Daher ist dort im Trainingsprozess eine Explorationsstrategie von Nöten. Außerdem wird die Strategie nur indirekt bestimmt, indem die Aktion mit dem höchsten Q-Wert genommen wird und deshalb ist der Trainingsprozess eher ineffizient und langsam. Aber vorteilhaft ist, dass gerade bei einer großen Anzahl an Zuständen, der Q-Wert schnell und effizient berechnet werden kann.
Policy Networks haben den großen Vorteil, dass direkt eine Policy also Strategie gelernt wird. Daher ist diese Form des Reinforcement Learnings effizienter und schneller. Der Nachteil ist jedoch, dass sich die gesamt Rendite schwer berechnen lässt, beziehungsweise erst am Ende einer Episode.

3.8.2 Vorteile

Daher kombiniert man beide Ansätze und kommt zum Actor Critic Verfahren. Hierbei ist der Actor das Netzwerk das handelt und Aktionen wählt. Hierbei handelt es sich um ein Neuronales Netz welches Policy-basierend ist, wie Policy Gradient. Der Critic/Kritiker bewertet diese Aktion. Dieser ist ein Werte basierendes Neuronale Netzwerk aus dem Q-Learning Bereich, wie zum Beispiel Deep Q-Network. Der Kritiker bewertet also eine Aktion mit einem Q-Wert, welcher benutzt werden kann, um den Actor zu trainieren. Dadurch kann der Lernprozess deutlich verkürzt werden.

3.8.3 Trainieren

Wie trainiert man nun Actor und Critic?
Für das Netzwerk des Actors haben wir folgende Formel hergeleitet:

$$\theta_{neu} = \theta_{alt} + \alpha \nabla_\theta R_{\sum,\theta}[20,\ 21] \tag{18}$$

Jedoch ist die Gesamtrendite $R_{\sum,\theta}$ schlecht zu bestimmen, da wir dafür bis an das Ende einer Episode warten müssten, um diese nach Gleichung (3) zu berechnen. Diese Gesamtrendite können wir jedoch auch einfach von unserem Q-Network schätzen lassen. Man kann nun die Parameter des Actors mit der Loss-Funktion 10 updaten, sodass die Formel folgendermaßen aussieht:

$$\theta_{neu} = \theta_{alt} + \alpha(r_t + \gamma * max_{a'} Q_{TNN}(s_{t+1}, a') - Q_{QNN}(s_t, a))[20] \tag{19}$$

Den Critic aktualisiere ich wie im Kapitel *Deep Q-Network* beschrieben.

4 Umsetzung der Künstlichen Intelligenz

4.1 Konfiguration des Neuronalen Netzes

Alle im Verlauf verwendeten Neuronalen Netze sind gleich konfiguriert[1]. Die Anzahl der Layer beträgt drei, wobei es einen Eingabe-, einen Hidden- und einen Ausgangslayer gibt. Daher wird das Neuronale Netz als Deep Neural Network klassifiziert. Die Anzahl der Eingangsneuronen wird durch die Anzahl an Eingabeparametern festgelegt. So besitzt dieses 19 Eingangsneuronen, da die Würfelaugen, die Anzahl an noch möglichen Restwürfen und die noch offenen und bereits genommenen Kategorien als Eingabeparameter festgelegt wurden. Alle Eingabewerte müssen auf einen Bereich zwischen Null und Eins normalisiert werden, sodass die Augenzahl der Würfel durch zehn geteilt werden muss und die Kategorien auf Null oder Eins gesetzt werden müssen, je nach dem ob diese schon ausgefüllt wurden (repräsentiert durch Eins) oder eben noch nicht (repräsentiert durch Null). Die Anzahl der Neuronen im Hiddenlayer sollte zwischen der Anzahl an Neuronen der Ein- und Ausgangsneuronen liegen, weshalb die Anzahl 15 gewählt wurde. Die Anzahl der Ausgangsneuronen entspricht der Anzahl an möglichen Aktionen, in diesem Fall 13, da 13 unterschiedliche Kategorien ausgewählt werden können. Außerdem muss die Lernrate α festgelegt werden. Eine Lernrate von 10^{-3} brachte mir die beste und stabilste Lernkurve. Der Rabattfaktor γ muss ebenfalls beim Policy Gradient Learning festgelegt werden und nimmt hier von 0.8 zu Beginn des Trainings kontinuierlich bis 0.5 am Ende des Trainings ab.

[1] Quellcodezeilen 1-56

4.2 1. Versuch

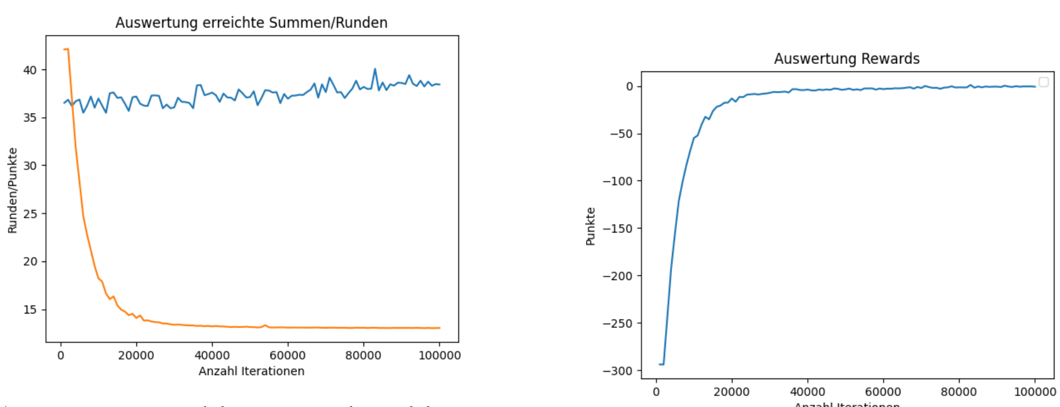
4.2.1 Einleitung

Für die Kniffel KI bietet sich die Umsetzung der Policy Deep Neural Networks an, da hierbei direkt die bestmögliche Strategie versucht wird, zu lernen. Es ist auch keine Explorationsstrategie von Nöten und im Gegensatz zu anderen Reinforcement Learning Methoden, wie Q-Learning, benötigt man hier nur ein Neuronales Netzwerk. Dies steigert die Effizienz des Lernprozesses erheblich, da das Trainieren eines Neuronalen Netzwerkes sehr rechen- und zeitintensiv ist.

4.2.2 Training

In einem ersten Versuch ist das Ziel der KI beizubringen, dass bereits ausgefüllte Ereignisse, dass heißt Kategorien bei denen bereits auf dem Kniffelblock eine Punktzahl eingetragen ist, nicht mehr genommen werden. Wurde bereits die Kategorie *Full House* genommen, so versteht die KI zu Beginn nicht, dass diese Kategorie nicht noch ein zweites Mal genommen werden kann. Dafür werden Aktionen bei denen bereits erledigte Kategorien erneut genommen werden mit dem Reward -10 bestraft. Des Weiteren werden Kategorien mit -5 bestraft, wenn die Punktzahl in dieser Kategorie Null beträgt, da hier ein Ereignis genommen wurde, das sehr wahrscheinlich völlig ungünstig bei den aktuellen Würfelergebnissen ist[2]. Auch wenn es in seltenen Fällen vorkommen kann, dass es sinnvoller ist eine Kategorie zu nehmen, auf die eine Punktzahl von Null folgt, also diese Kategorie zu streichen, ist die Anzahl dieser Fälle sehr gering und wird daher in diesem ersten Schritt nicht beachtet. Außerdem wird in diesem Stadium nur einmal pro Runde gewürfelt und es werden nicht gezielt Würfel behalten, um die Restlichen nochmals zu würfeln[3].

4.2.3 Auswertung



(a) orange: Anzahl an Runden, blau erreichte Summen/gesamt Punktzahlen

(b) blau: erhaltene Rewards

Abbildung 2

Die Abbildung 2 zeigt die Entwicklung im Trainingsprozess der KI. So zeigt die x-Achse der Abbildung 2a) die Anzahl an Trainingsiterationen, also wie viele Runden bereits trainiert

[2]Quellcodezeilen 59-157
[3]Quellcodezeilen 158-201

wurden und die y-Achse die in der jeweiligen benötigten Runden um einmal alle Kategorien auf dem Kniffelblockblatt auszufüllen beziehungweise die pro Runde erreichte Punktzahl. Die Abbildung 2b) stammt vom gleichen Trainingsprozess, zeigt auf der x-Achse die Anzahl an Trainingsiterationen und auf der y-Achse die Punktzahl des gesamten Rewards einer Runde.
Man sieht sehr deutlich, dass die Anzahl an benötigten Runden und Rewards zusammenhängen. So ist zu Beginn des Trainings die Anzahl an benötigten Runden mit knapp 45 sehr hoch, da ein Kniffelspiel in 13 Runden erledigt sein sollte. Da diese falsche Entscheidungen und die damit verbundene hohe Anzahl an Runden sehr stark bestraft werden, ist auch der Reward mit mehr als -300 sehr schlecht. Man sieht jedoch, dass die KI zu lernen beginnt und die benötigte Anzahl an Runden innerhalb von nur 20.000 Iterationen beginnt, gegen die ideale Anzahl von 13 zu konvergieren. Im gleichen Abschnitt verbessert sich im gleichen Maß auch der erhaltene Reward. Dieser steigt stark an und beträgt nun nicht mehr -300, sondern konvergiert gegen Null.
Die erreichte Punktzahl ist mit knapp 40 Punkten sehr wenig und unterdurchschnittlich. Das liegt aber daran, dass in der Berechnung der Rewards noch nicht berücksichtigt wird, wie gut die genommene Aktion ist. So wird lediglich der Reward -5 gegeben, wenn die Punktzahl der genommenen Kategorie Null beträgt, da hier sehr wahrscheinlich die Entscheidung falsch war. Deswegen ist auch keine Entwicklung im Verlauf der Iterationen zu sehen, weder bei der erreichten Punktzahl noch bei den erhaltenen Rewards.

4.3 2. Versuch

4.3.1 Einleitung

In einem zweiten Versuch war das Ziel, der KI eine Strategie bei zu bringen, die die vorherige KI in ihrer im Spiel erreichten Punktzahl deutlich übersteigt.

4.3.2 Umsetzung

Eine erste Überlegung war, das bereits in 4.1 trainierte Neuronale Netz zu übernehmen, mit dem Ziel deren Performance zu verbessern, da diese ja bereits gelernt hat, schon genommene Kategorien kein zweites Mal zu nehmen. Dieser Versuch scheiterte jedoch, da das Neuronale Netz nun schon zu sehr eingestellt war und nicht einfach auf die neuen Anforderungen adaptiert werden konnte. So wurde das Training sehr instabil, die Anzahl der Runden erhöhte sich wieder und die Punktzahl und der Reward insgesamt verbesserte sich nicht wirklich. Deshalb wird diesem zweiten Entwurf eine KI komplett neu trainiert und es wird nicht die Vorangegangene weiter verwendet.
Damit die KI eine passable Performance erhält, muss diese einer bestimmten Strategie folgen. Die einfachste und schnellste Idee ist hierbei eine greedy-Strategie. Das heißt es wird der KI beigebracht, die Kategorie auszuwählen, die erstens noch nicht ausgewählt wurde und die zweitens mit dem aktuellen Würfelergebnis die höchste Punktzahl erzielt.
Man muss jedoch darauf achten, dass die KI nicht sehr früh schon Chance nimmt, nur weil deren Punktzahl sehr hoch ist. Aufgrund dessen wird Chance nur mit der halben Punktzahl bewertet und nur wenn die Hälfte dieser Punktzahl noch immer größer ist als alle anderen möglichen noch offenen Kategorien.
In diesem Modell wird aufgrund der greedy-Strategie jedoch nicht beachtet, dass es in manchen Fällen sinnvoller sein kann Kategorien zu nehmen, die vielleicht jetzt nicht die maximale

Punktzahl erbringen, aber später höhere noch zu erreichende Punktzahlen zu ermöglichen. So ist es in manchen Fällen sicher sinnvoller den Kniffel zu streichen, auch wenn das eine Punktzahl von Null in dieser Kategorie zur Folge hat, um in später eine höhere Gesamtpunktzahl zu erreichen.
Da bisher noch nicht Würfel ausgewählt wurden um diese bis zu zwei weiteren Male zu würfeln, muss diese sinnvolle Auswahl auch noch umgesetzt werden. Dies verspricht nochmals höhere Punktzahlen in einzelnen Kategorien und für die KI eindeutigere Zuweisungen zu Kategorien, da es in Kapitel 4.1 sehr sehr selten vor kam, dass eine große Straße oder ein Kniffel gewürfelt wurde. Dadurch dass nun häufiger auch solche im ersten Wurf selten auftretende Würfelergebnisse vorkommen, kann die KI nun auch lernen, diese Würfe den Kategorien sinnvoll zuzuordnen.

4.3.3 Auswählen der Würfel zum erneuten Würfeln

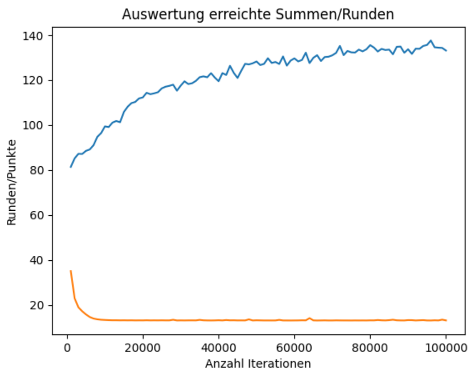
Wie in Kapitel 4.2.2 angemerkt, ist es sinnvoll das geschickte Auswählen der Würfel zum erneuten Würfeln umzusetzen, da dies die Performance der KI nochmals deutlich verbessert. Da das Auswählen der Würfel von vielen Faktoren, wie zum Beispiel bereits genommenen Kategorien, aktuell gewürfelte Würfel, sowie der Anzahl an noch verbleibenden Restwürfen, abhängt, wurde diese Auswahl nicht mithilfe einer KI umgesetzt, sondern algorithmisch. Das Trainieren der KI wäre aufgrund der großen Anzahl an Faktoren und Abhängigkeiten besonders rechen- und zeitintensiv. Insbesondere die hohe Rechenintensität und fehlenden Ressourcen meinerseits verhindern den Einsatz einer KI.
Daher ist die Auswahl der Würfel zum erneuten Würfeln algorithmisch umgesetzt. Genauer wird bei jedem Wurf der Erwartungswert für jede Kategorie berechnet. Dieser wird berechnet, indem zunächst die Wahrscheinlichkeit die Bedingungen einer bestimmten Kategorie zu erfüllen, ausgehend vom vorliegenden Würfelergebnis und den pro Kategorie am sinnvollsten zu behaltenen Würfeln, berechnet wird. Diese Wahrscheinlichkeit multipliziert mit dem daraus resultierenden zu erwartenden Wert. Die pro Kategorie am sinnvollsten zu behaltenen Würfeln ergeben sich aus einer allgemeinen Strategie basierend auf Wahrscheinlichkeiten für Kniffel [22]. Nun werden die zu behaltenden Würfel ausgewählt, die den höchsten Erwartungswert versprechen.

4.3.4 Trainieren

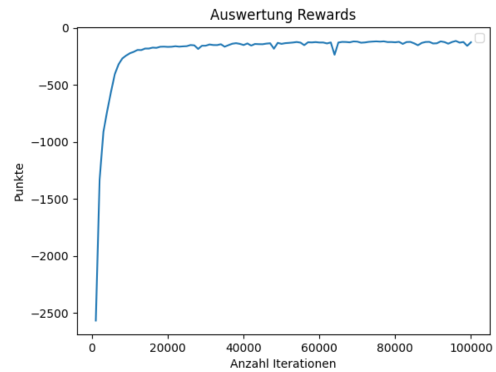
Nun muss die KI nur noch trainiert werden[4]. Hierfür muss bekanntermaßen ein Reward berechnet werden. Doch wie setzt sich dieser zusammen?
Nimmt die KI eine Kategorie die schon einmal in dieser Kniffelrunde genommen wurde, so erhält sie einen Reward von *-100*. Ansonsten wird die Auswahl der KI mit der Auswahl der greedy-Strategie verglichen und die Differenz zwischen dem maximalen in dieser Runde zu erreichenden Punktzahl der greedy-Strategie und der erhaltenen Punktzahl der tatsächlich genommen Kategorie, ergibt den Reward. Nimmt die KI die Kategorie die auch die greedy-Strategie nimmt so erhält sie dementsprechend den Reward von *0* [5]. Es macht keinen Unterschied, ob ein Reward von *0* oder einer in Höhe der tatsächlich erhaltenen Punktzahl der genommenen Kategorie gegeben wird, da das Prinzip der Rewardmaximierung gleich bleibt und sich nur das Maximum verschiebt.

[4] Quellcodezeilen 369-452
[5] Quellcodezeilen 205-365

4.3.5 Auswertung



(a) orange: Anzahl an Runden, blau erreichte Summen/gesamt Punktzahlen



(b) blau: erhaltene Rewards

Abbildung 3

Auch hier sieht man zu Beginn, dass die erhaltenen Rewards sehr schlecht sind und bis zu -2500 betragen. Das hängt damit zusammen, dass auch die Anzahl der Runden mit knapp 40 sehr hoch ist, da eine komplette Spielrunde Kniffel nur 13 Runden haben sollte. Beginnt die KI durch die Rewardmaximierung zu lernen, solche bereits genommenen Kategorien kein zweites Mal zu nehmen, so sieht man, dass in nur circa 10.000 Iterationen sich der Reward um etwa 2.000 verbessert und die Anzahl der benötigten Runden gegen 13 konvergiert. Die kleinen Unregelmäßigkeiten bei den Rewards bei circa 50.000 und 60.000 Iterationen lassen sich durch einen kleinen plötzlichen Anstieg an benötigten Runden erklären, die nur von kurzer Dauer sind.

Betrachtet man die erreichte Summen beziehungsweise die erreichten Gesamtpunktzahlen eines kompletten Spiels Kniffel an, so ist erkennbar, dass bereits die Punktzahl zu Beginn mit ungefähr 80 Punkten schon mehr als doppelt so hoch ist, wie bei der vorherigen KI in 4.1. Des Weiteren steigt die Punktzahl fast kontinuierlich mit kleineren Unregelmäßigkeiten an, sodass die Gesamtpunktzahl in der Spitze gegen Ende des Spiels zwischen 130 und 140 liegt. Dies entspricht einer Steigerung um etwa 200% im Vergleich zur vorherig entwickelten KI. Zwar liegt die maximal Gesamtpunktzahl noch deutlich unter dem Mittelwert von Kniffel mit 245 Punkten [22], aber es ist eine deutliche Steigerung erkennbar. Die Steigerung der erreichten Punktzahlen ist auch unmittelbar mit dem erkennbaren leichten Anstieg des Rewards zusammenhängend.

4.4 3. Versuch

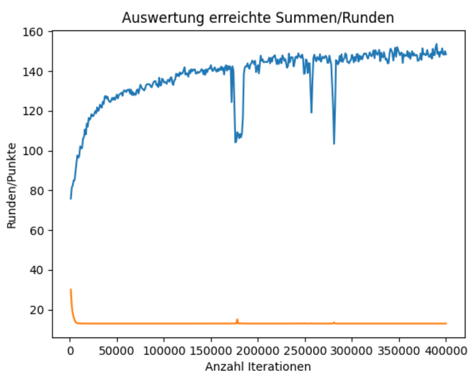
4.4.1 Einleitung

In diesem letzten Versuch war das Ziel die Performance der KI nochmals zu verbessern. Um dies zu erreichen, haben nun zwei KI Modelle gegeneinander gespielt. Dies hat den Zweck, dass die KI durch Gewinne gegen die andere KI dazu ermutigt werden soll, auch andere Aktionen auszuführen und andere Kategorien auszuwählen.

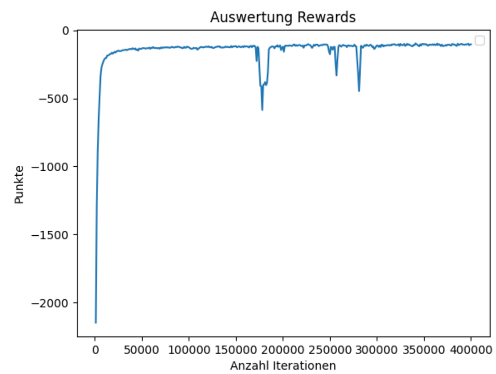
4.4.2 Training

Konkret gibt es zwei Neuronale Netze, die nach dem in 4.2 erklärten Konzept nach jeder Runde trainiert werden[6]. Ferner wird nach jeder kompletten Kniffelrunde ein Sieger zwischen den beiden KIs ermittelt. Der Sieger erhält für jeden Reward seiner Aktionen fünf Bonuspunkte und der Verlierer bekommt diese Punkte von seinen Rewards abgezogen[7]. Dadurch wird das Gewinnen und Verlieren in der Rewardberechnung berücksichtigt und die KI lernt Aktionen, die eher zum Ziel führen, auszuwählen.

4.4.3 Auswertung

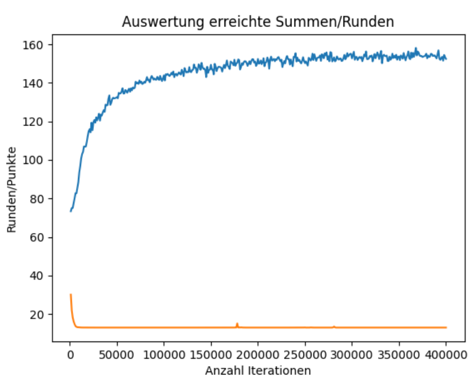


(a) orange: Anzahl an Runden, blau erreichte Summen/gesamt Punktzahlen

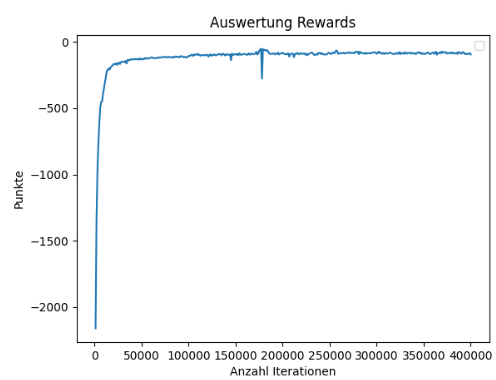


(b) blau: erhaltene Rewards

Abbildung 4



(a) orange: Anzahl an Runden, blau erreichte Summen/gesamt Punktzahlen



(b) blau: erhaltene Rewards

Abbildung 5

Abbildungen 4 und 5 stellen jeweils die Entwicklung einer KI dar. Es lässt sich einen großen Unterschied erkennen. Während die KI in Abbildung 4 zwar kontinuierlich aber mit großen

[6]Quellcodezeilen 205-365
[7]Quellcodezeilen 456-604

Ausreißern lernt, lernt die KI in Abbildung 5 sehr kontinuierlich und ohne auffällige Ausreißer. Wie in den beiden vorherigen KIs, lernen diese beiden KI auch zu Beginn, bereits genommene Kategorien nicht noch ein zweites Mal zu nehmen. Deshalb steigen auch hier die Rewards in den ersten Iterationen stark an und die Anzahl an benötigten Runden konvergiert gegen 13. Auffällig ist in beiden KI Modellen, dass die erreichte Punktzahl pro Spiel in einem ersten Abschnitt von 50.000 Iterationen stark ansteigt von 80 auf 140 Punkte. An dieses starke Wachstum schließt sich ein deutlich schwächeres Wachstum an, dass schlussendlich zu einer maximal Punktzahl pro Spiel von 160 Punkten bei beiden KIs führt. Es konnte so, durch das Gegeneinanderspielen der KIs nochmals eine Steigerung von 20 Punkten herbeigeführt werden. In der KI in Abbildung 4 kann man besonders im Abschnitt von 150.000 bis 300.000 eine gewisse Instabilität erkennen. So bricht dort die Punktzahl und der Reward teilweise um 40 Punkte ein, da erstens die KI deutlich schlechtere Entscheidungen trifft und daher schlechtere Rewards bekommt, aber diese zweitens in diesem Abschnitt auch gegen die KI in Abbildung 5 verliert und so noch zusätzlichen negativen Reward bekommt.
Aber insgesamt konnte die Perfomance am Ende der Iterationen bei beiden KIs nochmal um gut 20 Punkte gesteigert werden.

5 Probleme und Verbesserungsmöglichkeiten

Auch wenn mit den entwickelten KIs schon einige Fortschritte gemacht werden konnte, so gibt es immer noch Probleme, die eine stärkere Performance verhindern.
KI Berechnung und das Feed Forward, also das Benutzen eines Neuronalen Netzes, sowie die Backpropagation, also das Trainings des Neuronalen Netzes, sind sehr rechenintensive und fordernde Aufgaben. Fehlende Rechenleistung meinerseits hat zu einem erhöhten Zeitaufwand und zahlreichen Abstürzen geführt. Daher betrug die Maximalanzahl an Iterationen auch nur 400.000. Mit einer höheren Anzahl an Iterationen ist sicher auch eine Erhöhung der Performance möglich, wie die Graphen aus Kapitel 4 zeigen.
Möglicherweise helfen auch andere KI Modelle wie das Actor-Critic Modell um stabileres Lernen zu etablieren und stärkere Performance zu trainieren. Dieses ist jedoch aufwändiger und besonders rechenintensiv, da hier drei unterschiedliche Neuronale Netze benötigt werden. Jedoch wäre dies eine sehr gute Möglichkeit, die KI hinsichtlich ihrer Performance zu verbessern.
Möglich ist auch, dass die gewählte Lernrate α und der Rabattfaktor γ schlecht gewählt sind. In meinen Trainings haben sich diese jedoch als die besten etabliert.
Eine Erhöhung der Anzahl der Hiddenlayer im Neuronalen Netzwerk stellt ebenfalls eine Möglichkeit der Erhöhung des Potentials der KI dar, bringt einen jedoch noch größeren Rechenaufwand mit sich.
Außerdem ist die gewählte greedy-Strategie sicher nicht die Beste, um die Aktionen der KI zu bewerten. So erreichte diese im Test eine durchschnittliche Punktzahl von etwa 210, was etwas unter der durchschnittlichen Punktzahl von 245 liegt. So werden mit dieser Strategie sinvolle Streichungen des Kniffels beispielsweise nicht berücksichtigt. Dieses Problem versuchte ich zumindest teilweise durch das Gegeneinanderspielen der KIs zu lösen, da hier nicht mehr einzig und allein mit der greedy-Strategie bewertet wurde, sondern eben das Gewinnen eine höhere Relevanz bekommen hat. Trotzdem wäre es noch besser, die KIs noch freier gegen sich oder andere schon fortschrittlichere KIs spielen zulassen, um die Exploration weiter zu vergrößern und neue Möglichkeiten auszuprobieren. Des Weiteren müsste man, um eine möglichst perfekte Strategie zu erhalten, auch die Spielstände der anderen Mitspieler mitgeben. Dies erhöht die

Anzahl an möglichen Zuständen wie in Kapitel 1 gezeigt erheblich.

6 Fazit

Obwohl es noch einige Probleme und Verbesserungsmöglichkeiten gibt, wie in Kapitel 5 erklärt, liefern die KIs schon teilweise gute Performances ab. Besonders die letzte entwickelte KI in 4.4 liegt zwar mit 160 Punkten noch unter dem Durchschnitt von 245 Punkten, hat sich diesem aber schon deutlich angenähert. Des Weiteren ist ein klarer Trend in der Entwicklung der KIs erkennbar und so performt meine letzte KI deutlich besser als die erste. Auch ist mir eine gute objektorientierte Implementation des Spiels Kniffel gelungen und es gibt die Möglichkeit, das Spiel mit einer graphischen Benutzeroberfläche zu spielen.
Auch wenn die KI noch nicht auf einem sehr guten Level spielt, so ist die Leistung in einem guten Bereich. Es ist nicht verwunderlich, dass die Performance meiner KI mit 100.000 bis 400.000 Iterationen nicht perfekt ist, wenn man sich die Anzahl an möglichen Zuständen anschaut, die in Kapitel 1 berechnet wurde und bei einem Spiel mit zwei Spielern schon etwa $5 \cdot 10^{14}$ beträgt. Die vorgenommenen Abstrahierungen, durch die fehlende Betrachtung der Punktzahlen der Mitspieler und der algorithmischen Umsetzung zur Entscheidung der neu zu würfelnden Würfel, reduzieren zwar die Anzahl an möglichen Zuständen, verhindern aber dadurch auch eine perfekte Strategie zum Spielen von Kniffel, da dazu alle Informationen nötig sind. Ziel dieser Arbeit war es nicht, die beste KI zu entwickeln, sondern ein erstes Mal in Berührung mit dem großen Bereich der Künstlichen Intelligent zu kommen und erste Erfahrungen beim Umsetzen von KI Lernmodellen, wie dem Reinforcement Learning und der Policy Gradient Methode, zu sammeln. Dieses Ziel habe ich mit dieser Arbeit erreicht.

7 Literaturverzeichnis

Literatur

[1] J. McCarthy, “WHAT IS ARTIFICIAL INTELLIGENCE?” *Computer Science Department Stanford University Stanford, CA 94305*, pp. 2–8, 2007 Nov 12, 2:05 a.m. [Online]. Verfügbar unter: https://www-formal.stanford.edu/jmc/whatisai.pdf

[2] C. Janiesch, P. Zschech, and K. Heinrich, “Machine learning and deep learning,” *Electronic Markets*, vol. 31, no. 3, pp. 685–695, 8. Apr 2021.

[3] M. Batta, “Machine Learning Algorithms - A Review,” *International Journal of Science and Research*, vol. 9, no. 1, pp. 381–386, 1 2020. [Online]. Verfügbar unter: https://www.ijsr.net/getabstract.php?paperid=ART20203995

[4] P. Cunningham, M. Cord, and S. J. Delany, *Supervised Learning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 21–22. [Online]. Verfügbar unter: https://doi.org/10.1007/978-3-540-75171-7_2

[5] K. Tyagi, C. Rane, R. Sriram, and M. Manry, “Chapter 3 - Unsupervised learning,” in *Artificial Intelligence and Machine Learning for EDGE Computing*, R. Pandey, S. K. Khatri, N. kumar Singh, and P. Verma, Eds. Academic Press, 2022, pp. 33–52. [Online]. Verfügbar unter: https://www.sciencedirect.com/science/article/pii/B9780128240540000125

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, 2nd ed., ser. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018.

[7] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, 01. May 1992.

[8] C. J. C. H. Watkins, “Learning from delayed rewards,” 1989.

[9] R. S. Sutton and A. G. Barto, “The reinforcement learning problem,” *Reinforcement learning: An introduction*, pp. 51–85, Jan. 1998, course notes from the book, chapter 3.

[10] H. Hamdy, “Bellman Equation (1): Understanding the Recursive Nature of the Bellman Equation in Mathematics,” [Online], 4. Jul 2023, [Zugegriffen am 12.03.2023]. [Online]. Verfügbar unter: https://medium.com/@hosamedwee/bellman-equation-1-understanding-the-recursive-nature-of-the-bellman-equation-in-mathematics-

[11] TU Chemnitz, “The Reinforcement Learning Problem,” [Online], [Zugegriffen am 12.03.2023]. [Online]. Verfügbar unter: https://www.tu-chemnitz.de//informatik/KI/scripts/ws0910/ml09_3.pdf

[12] A. Singh, “Reinforcement Learning: Bellman Equation and Optimality (Part 2),” [Online], 31 Aug. 2019, [Zugegriffen am 12.03.2023]. [Online]. Verfügbar unter: https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3

[13] The Avenga Team, “Real-world applications of Q-learning: a gentle introduction,” [Online), 30. Nov. 2022, [Zugegriffen am 12.03.2024]. [Online]. Verfügbar unter: https://www.avenga.com/magazine/q-learning-applications/?region=de

[14] K. Doshi, “Reinforcement Learning Explained Visually (Part 4): Q Learning, step-by-step,” [Online], 28 Nov. 2020, [Zugegriffen am 12.03.2023]. [Online]. Verfügbar unter: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-4-q-learning-step-by-step-b65efb731d3e

[15] M. Wunder, M. L. Littman, and M. Babes, “Classes of multiagent q-learning dynamics with epsilon-greedy exploration,” in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 1167–1174. [Online]. Verfügbar unter: http://engr.case.edu/ray_soumya/mlrg/2010ClassesofMultiagentQ-learningDynamicswithe-greedyExploration.pdf

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement Learning: A Survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 01. May 1996. [Online]. Verfügbar unter: https://www.jair.org/index.php/jair/article/view/10166/24110

[17] P. Boekhoven, “Entwicklung eines Reinforcement LearningFrameworks auf Basis eines Agentensystems,” *Bachelorarbeit eingereicht im Rahmen der Bachelorprüfungim Studiengang Angewandte Informatikam Department Informatikder Fakultät Technik und Informatikder Hochschule für Angewandte Wissenschaften Hamburg*, pp. S. 14–15, 27. März 2011. [Online]. Verfügbar unter: https://reposit.haw-hamburg.de/bitstream/20.500.12738/5359/1/Arbeit.pdf

[18] M. Semmler, “Exploration in Deep Reinforcement Learning,” *Bachelor-Thesis von Markus Semmler aus Rüsselsheim TU Darmstadt*, pp. S. 7–8, 23. Okt. 2017. [Online]. Verfügbar unter: https://www.ias.informatik.tu-darmstadt.de/uploads/Theses/Abschlussarbeiten/markus_semmler_bsc.pdf

[19] K. Doshi, “Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step,” [Online], 19. Dez. 2020, [Zugegriffen am 13.03.2024]. [Online]. Verfügbar unter: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b

[20] S. L. Brunton, “Overview of Deep Reinforcement Learning Methods,” [Online, Video], 21. Jan. 2022, [Zugegriffen am 13.03.2024].

[21] C. Yoon, “Deriving Policy Gradients and Implementing REINFORCE,” [Online], 30. Dez. 2018, [Zugegriffen am 13.03.2023]. [Online]. Verfügbar unter: https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

[22] W. Brefeld, “Kniffel - Wahrscheinlichkeiten und Punktzahlen bei optimaler Strategie,” 1998, [Zugegriffen am 03.05.2023]. [Online]. Verfügbar unter: https://brefeld.hier-im-netz.de/kniffel.html

8 Anhang

8.1 Quellcode

```python
#4.1
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable


class PolicyNet(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, learning_rate=6e-4):
        super().__init__()

        self.num_actions = num_actions
        self.linear1 = nn.Linear(num_inputs, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, num_actions)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def forward(self, state):
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = F.softmax(self.linear3(x), dim=0)
        return x

    def get_action(self, state):
        state = torch.Tensor(np.array(state, dtype = float))
        probs = self.forward(Variable(state))
        highest_prob_action = np.random.choice(self.num_actions, p=np.squeeze(probs.detach().numpy()))
        log_prob = torch.log(probs.squeeze(0)[highest_prob_action])
        return highest_prob_action, log_prob

    def update_policy(self, rewards, log_probs, gamma):
        discounted_rewards = []

        for t in range(len(rewards)):
            Gt = 0
            pw = 0
            for r in rewards[t:]:
                Gt = Gt + gamma**pw * r
                pw = pw + 1
            discounted_rewards.append(Gt)

        discounted_rewards = torch.tensor(discounted_rewards)
        if len(discounted_rewards) > 1:
            discounted_rewards = (discounted_rewards - discounted_rewards.mean()) / (discounted_rewards.std() + 1e-9)

        policy_gradient = []
        for log_prob, Gt in zip(log_probs, discounted_rewards):
            policy_gradient.append(-log_prob * Gt)
```

```

        self.optimizer.zero_grad()
        policy_gradient = torch.stack(policy_gradient).sum()
        policy_gradient.backward()
        self.optimizer.step()

#4.2
import numpy as np

class KniffelEnv(object):
    def __init__(self):
        self._action_spec = dict(
            dtype=np.int64, minimum=0, maximum=12, name='action')
        self._observation_spec = dict(shape=[21,1], dtype=np.int64, minimum=0, name='observation')
        self.state = []
        self.wuerfel = np.random.randint(1, 6, size = 5)
        self.listeGemacht = np.zeros(13, dtype = int)
        self.listePunkte = np.zeros(13, dtype = int)
        self.summen = np.zeros(3, dtype = int)
        self.episode_ended = False
        self.current_time_step = None

    def action_spec(self):
        return self._action_spec

    def observation_spec(self):
        return self._observation_spec

    def getListeGemacht(self):
        return self.listeGemacht

    def normalisiereWuerfel(self, wuerfel):
        wuerfelN = np.array([0,0,0,0,0], dtype='float')
        for i in range(5):
            wuerfelN[i] = wuerfel[i]/10
        return wuerfelN

    def reset(self):
        self.wuerfel = np.random.randint(1, 6, size = 5)
        self.listeGemacht = np.zeros(13, dtype = int)
        self.listePunkte = np.zeros(13, dtype = int)
        self.summen = np.zeros(3, dtype = int)
        self.state = np.concatenate((self.normalisiereWuerfel(self.wuerfel), self.listeGemacht))
        self.episode_ended = False
        self.current_time_step = dict(oberservation = [np.concatenate((self.normalisiereWuerfel(self.wuerfel), self.listeGemacht, self.summen))], dtype=np.int64)
        return self.state

    def step(self, aktion):
        if self.listeGemacht[aktion] == 0:
            self.listeGemacht[aktion] = 1
            if aktion >= 0 and aktion <= 5:
                reward = np.count_nonzero(self.wuerfel == aktion + 1) * (aktion +
```

```
1)
                self.summen[0] += reward
            else:
                gewuerfelteZahlen = []
                for i in range(1, 7):
                    gewuerfelteZahlen += [np.count_nonzero(self.wuerfel == i)]
                if aktion == 6:
                    if 3 in gewuerfelteZahlen or 4 in gewuerfelteZahlen or 5 in gewuerfelteZahlen:
                        reward = self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4]
                    else:
                        reward = 0
                elif aktion == 7:
                    if 4 in gewuerfelteZahlen or 5 in gewuerfelteZahlen:
                        reward = self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4]
                    else:
                        reward = 0
                elif aktion == 8:
                    if 2 in gewuerfelteZahlen and 3 in gewuerfelteZahlen:
                        reward = 25
                    else:
                        reward = 0
                elif aktion == 9:
                    if (1 in self.wuerfel and 2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel) or (2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel) or (3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel and 6 in self.wuerfel):
                        reward = 30
                    else:
                        reward = 0
                elif aktion == 10:
                    if (1 in self.wuerfel and 2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel) or (2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel and 6 in self.wuerfel):
                        reward = 40
                    else:
                        reward = 0
                elif aktion == 11:
                    if 5 in gewuerfelteZahlen:
                        reward = 50
                    else:
                        reward = 0
                elif aktion == 12:
                    reward = (self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4])/2
                self.summen[1] += reward
            self.listePunkte[aktion] = reward
            self.summen[2] = self.summen[0] + self.summen[1]
            if self.summen[0] >= 63:
                self.summen[2] += 35
        else:
            reward = -10
        if reward == 0:
```

```
            reward = -5
        self.wuerfel = np.random.randint(1, 6, size = 5)
        self.state = np.concatenate((self.normalisiereWuerfel(self.wuerfel), self.listeGemacht))
        self.current_time_step = dict(state = np.concatenate((self.normalisiereWuerfel(self.wuerfel), self.listeGemacht, self.summen)), reward = reward, dtype=np.int64)
        self.done = False
        if np.count_nonzero(self.listeGemacht) == 13:
            self.done = True
        return self.state, reward, self.done, self.summen[-1]


from KniffelInt import KniffelEnv
from PolicyGradient import PolicyNet
import numpy as np
import torch

def trainAgent(numEpisodes):
    env = KniffelEnv()
    net = PolicyNet(18, 13, 18)
    steps = []
    collectRewards = []
    collectSums = []

    for episode in range(1, numEpisodes + 1):
        gammaBeg = 0.2
        gammaEnd = 0.8
        state = env.reset()
        logProbs = []
        rewards = []
        step = 0
        done = False
        iteration = 500
        if episode % iteration == 0:
            print(episode, np.mean(collectRewards[-iteration:]), np.mean(collectSums[-iteration:]), np.mean(steps[-iteration:]))
        while not done:
            action, logProb = net.get_action(state)
            nextState, reward, done, sum = env.step(action)
            logProbs += [logProb]
            rewards += [reward]
            step += 1
            state = nextState
        gamma = round((gammaEnd - gammaBeg)* episode/numEpisodes + gammaBeg, 2)
        net.update_policy(rewards, logProbs, gamma)
        steps += [step]
        collectRewards += [np.sum(rewards)]
        collectSums += [sum]
    print(steps)
    print(collectRewards)
    return net


net = trainAgent(200000)
torch.save(net.state_dict(), 'policyNetDict1.pth')
```

```
#4.3/4.4
import numpy as np

class KniffelEnv(object):
    def __init__(self):
        self.state = []
        self.neugewuerfelt = 0
        self.wuerfel = np.random.randint(1, 6, size = 5)
        self.listeGemacht = np.zeros(13, dtype = int)
        self.listePunkte = np.zeros(13, dtype = int)
        self.summen = np.zeros(3, dtype = int)
        self.episode_ended = False
        self.current_time_step = None


    def getListeGemacht(self):
        return self.listeGemacht

    def normalisiereWuerfel(self, wuerfel):
        wuerfelN = np.array([0,0,0,0,0], dtype='float')
        for i in range(5):
            wuerfelN[i] = wuerfel[i]/10
        return wuerfelN

    def reset(self):
        self.wuerfel = np.random.randint(1, 6, size = 5)
        self.listeGemacht = np.zeros(13, dtype = int)
        self.listePunkte = np.zeros(13, dtype = int)
        self.summen = np.zeros(3, dtype = int)
        self.state = np.concatenate(([self.neugewuerfelt], self.normalisiereWuerfel(self.wuerfel), self.listeGemacht))
        self.episode_ended = False
        self.neugewuerfelt = 0
        self.current_time_step = dict(oberservation = [np.concatenate((self.normalisiereWuerfel(self.wuerfel), self.listeGemacht, self.summen))], dtype=np.int64)
        return self.state

    def getPunktzahl(self, aktion, gewuerfelteZahlen):
        if self.listeGemacht[aktion] == 0:
            if aktion >= 0 and aktion <= 5:
                punkte = np.count_nonzero(self.wuerfel == aktion + 1) * (aktion + 1)
            else:
                if aktion == 6:
                    if 3 in gewuerfelteZahlen or 4 in gewuerfelteZahlen or 5 in gewuerfelteZahlen:
                        punkte = self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4]
                    else:
                        punkte = 0
                elif aktion == 7:
                    if 4 in gewuerfelteZahlen or 5 in gewuerfelteZahlen:
                        punkte = self.wuerfel[0] + self.wuerfel[1] + self.wuerfel
```

```
[2] + self.wuerfel[3] + self.wuerfel[4]
                    else:
                        punkte = 0
                elif aktion == 8:
                    if 2 in gewuerfelteZahlen and 3 in gewuerfelteZahlen:
                        punkte = 25
                    else:
                        punkte = 0
                elif aktion == 9:
                    if (1 in self.wuerfel and 2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel) or (2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel) or (3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel and 6 in self.wuerfel):
                        punkte = 30
                    else:
                        punkte = 0
                elif aktion == 10:
                    if (1 in self.wuerfel and 2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel) or (2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel and 6 in self.wuerfel):
                        punkte = 40
                    else:
                        punkte = 0
                elif aktion == 11:
                    if 5 in gewuerfelteZahlen:
                        punkte = 50
                    else:
                        punkte = 0
                elif aktion == 12:
                    punkte = (self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4])//2
        else:
            punkte = -10
        return punkte

    def getPunkteFuerJedeAktion(self):
        gewuerfelteZahlen = []
        for i in range(1, 7):
            gewuerfelteZahlen += [np.count_nonzero(self.wuerfel == i)]
        punktzahlen = []
        for aktion in range(13):
            punktzahlen += [self.getPunktzahl(aktion, gewuerfelteZahlen)]
        return punktzahlen

    def greedy(self, aktionNet):
        allePunktzahlen = self.getPunkteFuerJedeAktion()
        if aktionNet == allePunktzahlen.index(max(allePunktzahlen)):
            reward = max(allePunktzahlen)
        else:
            reward = (allePunktzahlen[aktionNet] - max(allePunktzahlen))*2
        nextState, pReward, done, sum = self.step(aktionNet)
        if pReward == -10:
            reward = -100
        return nextState, reward, done, sum
```

```
    def step(self, aktion):
        if self.listeGemacht[aktion] == 0:
            self.listeGemacht[aktion] = 1
            if aktion >= 0 and aktion <= 5:
                reward = np.count_nonzero(self.wuerfel == aktion + 1) * (aktion + 1)
                self.summen[0] += reward
            else:
                gewuerfelteZahlen = []
                for i in range(1, 7):
                    gewuerfelteZahlen += [np.count_nonzero(self.wuerfel == i)]
                if aktion == 6:
                    if 3 in gewuerfelteZahlen or 4 in gewuerfelteZahlen or 5 in gewuerfelteZahlen:
                        reward = self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4]
                    else:
                        reward = 0
                elif aktion == 7:
                    if 4 in gewuerfelteZahlen or 5 in gewuerfelteZahlen:
                        reward = self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4]
                    else:
                        reward = 0
                elif aktion == 8:
                    if 2 in gewuerfelteZahlen and 3 in gewuerfelteZahlen:
                        reward = 25
                    else:
                        reward = 0
                elif aktion == 9:
                    if (1 in self.wuerfel and 2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel) or (2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel) or (3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel and 6 in self.wuerfel):
                        reward = 30
                    else:
                        reward = 0
                elif aktion == 10:
                    if (1 in self.wuerfel and 2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel) or (2 in self.wuerfel and 3 in self.wuerfel and 4 in self.wuerfel and 5 in self.wuerfel and 6 in self.wuerfel):
                        reward = 40
                    else:
                        reward = 0
                elif aktion == 11:
                    if 5 in gewuerfelteZahlen:
                        reward = 50
                    else:
                        reward = 0
                elif aktion == 12:
                    reward = (self.wuerfel[0] + self.wuerfel[1] + self.wuerfel[2] + self.wuerfel[3] + self.wuerfel[4])/2
                self.summen[1] += reward
            self.listePunkte[aktion] = reward
```

```
            self.summen[2] = self.summen[0] + self.summen[1]
            if self.summen[0] >= 63:
                self.summen[2] += 35
        else:
            reward = -10
        if reward == 0:
            reward = -50
        self.wuerfel = np.random.randint(1, 6, size = 5)
        self.neugewuerfelt = 0
        self.state = np.concatenate(([self.neugewuerfelt], self.normalisiereWuerfel(self.wuerfel), self.listeGemacht))
        self.current_time_step = dict(state = np.concatenate((self.normalisiereWuerfel(self.wuerfel), self.listeGemacht, self.summen)), reward = reward, dtype=np.int64)
        self.done = False
        if np.count_nonzero(self.listeGemacht) == 13:
            self.done = True
        return self.state, reward, self.done, self.summen[-1]

    def neuWuerfeln(self, wuerfelBehalten):
        self.wuerfel = np.concatenate((np.random.randint(1, 6, size=5-len(wuerfelBehalten)), np.array(wuerfelBehalten)))
        self.neugewuerfelt += 1
        self.state = np.concatenate(([self.neugewuerfelt], self.normalisiereWuerfel(self.wuerfel), self.listeGemacht))
        return self.state


#4.3
from KniffelBesteAktion import KniffelEnv
from PolicyGradient import PolicyNet
import numpy as np
from berechnung import *
import torch
import matplotlib.pyplot as plt

def trainAgent(numEpisodes, nameNet):
    global avgSum, avgSteps, avgRewards, x1
    env = KniffelEnv()
    net = PolicyNet(19, 13, 15)
    steps = []
    collectRewards = []
    collectSums = []
    avgSum = []
    avgSteps = []
    avgRewards = []
    x = []
    x1 = []
    for episode in range(1, numEpisodes + 1):
        gammaBeg = 0.5
        gammaEnd = 0.8
        state = env.reset()
        logProbs = []
        rewards = []
        step = 0
        done = False
```

```
        iteration = 1000
        if episode % iteration == 0:
            x1 += [episode]
            avgSum += [np.mean(collectSums[-iteration:])]
            avgSteps += [np.mean(steps[-iteration:])]
            avgRewards += [np.mean(collectRewards[-iteration:])]
            print(x1[-1], avgSum[-1], avgSteps[-1], avgRewards[-1])
        while not done:
            gleich = False
            while state[0] <= 2 and not gleich:
                wuerfelNorminalisiert = state[1:6]
                wuerfel = wuerfelWiederherstellen(wuerfelNorminalisiert)
                wuerfelBehalten = berechneWuerfelBehalten(wuerfel.tolist(), 2-state[0], state[6:].tolist())
                state = env.neuWuerfeln(wuerfelBehalten[1])
            action, logProb = net.get_action(state)
            nextState, reward, done, sum = env.greedy(action)
            logProbs += [logProb]
            rewards += [reward]
            step += 1
            state = nextState
        gamma = round((gammaEnd - gammaBeg)* episode/numEpisodes + gammaBeg, 2)
        net.update_policy(rewards, logProbs, gamma)
        steps += [step]
        collectRewards += [np.sum(rewards)]
        collectSums += [sum]
        x += [episode]
    print(steps)
    print(collectRewards)



    plt.figure(1)
    plt.plot(x1, avgSum, label='erreichte Summen')
    plt.plot(x1, avgSteps, label='Anzahl der Runden')
    plt.title('Auswertung erreichte Summen/Runden')
    plt.xlabel('Anzahl Iterationen')
    plt.ylabel('Runden/Punkte')

    # Plot the second graph
    plt.figure(2)
    plt.plot(x1, avgRewards)
    plt.title('Auswertung Rewards')
    plt.xlabel('Anzahl Iterationen')
    plt.ylabel('Punkte')
    plt.legend()

    plt.figure(1)
    plt.savefig('4.1.png')
    plt.figure(2)
    plt.savefig('4.2.png')

    plt.show()

    return net
```

```
net = trainAgent(100000, 'policyNet1.pth')


#4.4
from KniffelBesteAktion import KniffelEnv
from PolicyGradient import PolicyNet
import numpy as np
from berechnung import *
import torch
import matplotlib.pyplot as plt

def trainAgent(numEpisodes, nameNet):
    env1 = KniffelEnv()
    env2 = KniffelEnv()
    net1 = PolicyNet(19, 13, 15)
    net2 = PolicyNet(19, 13, 15)
    steps1 = []
    steps2 = []
    collectRewards1 = []
    collectSums1 = []
    avgSum1 = []
    avgSteps1 = []
    avgRewards1 = []
    x1 = []
    collectRewards2 = []
    collectSums2 = []
    avgSum2 = []
    avgSteps2 = []
    avgRewards2 = []
    x2 = []
    x = []
    sum1 = 0
    sum2 = 0
    win = 100000
    for episode in range(1, numEpisodes + 1):
        gammaBeg = 0.5
        gammaEnd = 0.8
        state1 = env1.reset()
        state2 = env2.reset()
        logProbs1 = []
        logProbs2 = []
        rewards1 = []
        rewards2 = []
        step1 = 0
        step2 = 0
        done1 = False
        done2 = False
        iteration = 1000
        if episode % iteration == 0:
            x1 += [episode]
            avgSum1 += [np.mean(collectSums1[-iteration:])]
            avgSteps1 += [np.mean(steps1[-iteration:])]
            avgRewards1 += [np.mean(collectRewards1[-iteration:])]
            x2 += [episode]
            avgSum2 += [np.mean(collectSums2[-iteration:])]
```

```
            avgSteps2 += [np.mean(steps2[-iteration:])]
            avgRewards2 += [np.mean(collectRewards2[-iteration:])]
            print(x1[-1], avgSum1[-1], avgSteps1[-1], avgRewards1[-1])
            print(x2[-1], avgSum2[-1], avgSteps2[-1], avgRewards2[-1])

        while not done1 and not done2:
            gleich = False
            while state1[0] <= 2 and not gleich:
                wuerfelNorminalisiert = state1[1:6]
                wuerfel = wuerfelWiederherstellen(wuerfelNorminalisiert)
                wuerfelBehalten = berechneWuerfelBehalten(wuerfel.tolist(), 2-state1[0], state1[6:].tolist())
                state1 = env1.neuWuerfeln(wuerfelBehalten[1])
            if  not done1:
                action, logProb = net1.get_action(state1)
                nextState, reward, done1, sum1 = env1.greedy(action)
                logProbs1 += [logProb]
                rewards1 += [reward]
                step1 += 1
                state1 = nextState


            gleich = False
            while state2[0] <= 2 and not gleich:
                wuerfelNorminalisiert = state2[1:6]
                wuerfel = wuerfelWiederherstellen(wuerfelNorminalisiert)
                wuerfelBehalten = berechneWuerfelBehalten(wuerfel.tolist(), 2-state2[0], state2[6:].tolist())
                state2 = env2.neuWuerfeln(wuerfelBehalten[1])
            if not done2:
                action, logProb = net2.get_action(state2)
                nextState, reward, done2, sum2 = env2.greedy(action)
                logProbs2 += [logProb]
                rewards2 += [reward]
                step2 += 1
                state2 = nextState
        if episode > win:
            if sum1 > sum2:
                k = 1
            else:
                k = -1
            for i in range(len(rewards1)):
                rewards1[i] = rewards1[i] + k*5
            for i in range(len(rewards2)):
                rewards2[i] = rewards2[i] + k*(-5)
        gamma = round((gammaEnd - gammaBeg)* episode/numEpisodes + gammaBeg, 2)
        net1.update_policy(rewards1, logProbs1, gamma)
        steps1 += [step1]
        collectRewards1 += [np.sum(rewards1)]
        collectSums1 += [sum1]

        net2.update_policy(rewards2, logProbs2, gamma)
        steps2 += [step2]
        collectRewards2 += [np.sum(rewards2)]
        collectSums2 += [sum2]
```

```python

    plt.figure(1)
    plt.plot(x1, avgSum1, label='erreichte Summen')
    plt.plot(x1, avgSteps1, label='Anzahl der Runden')
    plt.title('Auswertung erreichte Summen/Runden')
    plt.xlabel('Anzahl Iterationen')
    plt.ylabel('Runden/Punkte')

    plt.figure(2)
    plt.plot(x1, avgRewards1)
    plt.title('Auswertung Rewards')
    plt.xlabel('Anzahl Iterationen')
    plt.ylabel('Punkte')
    plt.legend()

    plt.figure(3)
    plt.plot(x2, avgSum2, label='erreichte Summen')
    plt.plot(x2, avgSteps2, label='Anzahl der Runden')
    plt.title('Auswertung erreichte Summen/Runden')
    plt.xlabel('Anzahl Iterationen')
    plt.ylabel('Runden/Punkte')

    plt.figure(4)
    plt.plot(x2, avgRewards2)
    plt.title('Auswertung Rewards')
    plt.xlabel('Anzahl Iterationen')
    plt.ylabel('Punkte')
    plt.legend()

    plt.figure(1)
    plt.savefig('3.1.png')
    plt.figure(2)
    plt.savefig('3.2.png')
    plt.figure(3)
    plt.savefig('3.3.png')
    plt.figure(4)
    plt.savefig('3.4.png')

    plt.show()

    return net1


net = trainAgent(400000, 'policyNet1.pth')
```

9 Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

Graphische Umsetzung des Würfelspiels Kniffel und Einbindung einer Künstlichen Intelligenz

selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum: ______________________________________

Unterschrift: ______________________________________