Hochschule Düsseldorf
University of Applied Sciences
HSD

Hochschule Düsseldorf

Data Science & AI

Modul: Advances in Intelligent Systems (ISS)

Wintersemester 2024/25

Projektdokumentation

Prüfer: Prof. Dr. Dennis Müller

Abgabetermin: 16.02.2025

Verfasser: Justin Bergmann

Matrikelnummer: 893710

Inhaltsverzeichnis

1 EINLEITUNG

Künstliche Intelligenz (KI) und maschinelles Lernen (ML) haben in den letzten Jahren große Fortschritte gemacht und finden Anwendung in verschiedensten Bereichen, darunter auch in der Spieleentwicklung. Reinforcement Learning (RL), eine Form des maschinellen Lernens, ermöglicht es Agenten, durch Interaktion mit einer Umgebung selbstständig optimale Strategien zu erlernen. Ein bekanntes Beispiel ist das Training von KI-Agenten für Spiele wie Schach, Go oder Videospiele.

In diesem Projekt wird das klassische Snake-Spiel mittels Deep Q-Learning (DQL) automatisiert. Ziel ist es, eine künstliche Intelligenz zu entwickeln, die das Spiel selbstständig lernt und durch Optimierung ihrer Strategie immer bessere Ergebnisse

erzielt. Dabei kommen Techniken wie neuronale Netze, Q-Learning und Replay Memory zum Einsatz.

1.1 STATE OF THE ART

Die Steuerung von Spielfiguren durch Reinforcement Learning ist ein aktives Forschungsfeld in der KI. DeepMind demonstrierte 2015 mit AlphaGo die Leistungsfähigkeit von Deep Reinforcement Learning, und seither werden ähnliche Techniken in vielen anderen Spielen angewendet. Insbesondere das Q-Learning, eine Form des RL, wird häufig für Spiele eingesetzt, bei denen es klare Zustände und Aktionen gibt.

Beim Snake-Spiel gibt es verschiedene Ansätze, um eine KI zu entwickeln:

Regelbasierte Algorithmen arbeiten nach festen Regeln, wie zum Beispiel "Folge immer dem nächsten Futter". Sie bieten keine Möglichkeit, sich an neue Situationen anzupassen oder sich zu verbessern. Die Entscheidungen sind immer die gleichen, egal wie sich das Spiel verändert.

Evolutionäre Algorithmen sind da schon flexibler. Sie ahmen den Prozess der natürlichen Selektion nach, bei dem die besten Lösungen über Generationen hinweg

weitergegeben und verbessert werden. So entwickeln sich immer bessere Strategien, ohne dass man die Regeln direkt anpassen muss.

Deep Q-Learning (DQL) bringt das Ganze auf ein neues Level. Es kombiniert neuronale Netze mit Q-Learning, um der KI beizubringen, wie sie Entscheidungen basierend auf Erfahrungen trifft. Das Modell lernt aus vergangenen Ereignissen (Experience Replay) und verwendet mehrere Schichten im neuronalen Netzwerk, um die besten Aktionen zu schätzen. So wird der Agent immer besser und kann flexibler auf unterschiedliche Spielsituationen reagieren.

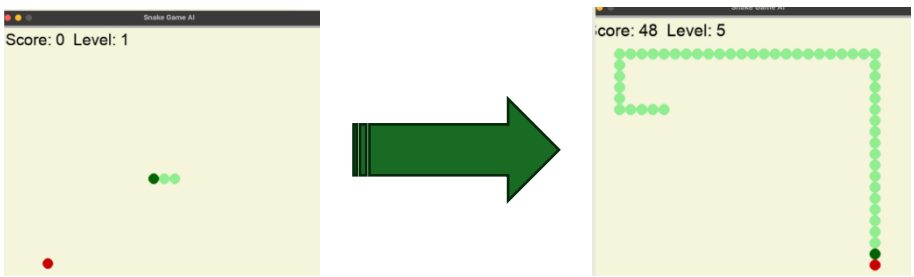
1.2 ZIEL DES PROJEKTES

Das Ziel dieses Projekts ist die Entwicklung eines KI-Agenten, der das Snake-Spiel selbstständig meistert und mit der Zeit durch Lernen immer bessere Leistungen erzielt. Die KI soll durch Deep Q-Learning trainiert werden, um basierend auf vergangenen Erfahrungen optimale Entscheidungen zu treffen.

Die spezifischen Ziele sind:

1. **Implementierung des Spiels** mit einer geeigneten Umgebung für das Training der KI.
2. **Entwicklung eines neuronalen Netzwerks**, das als Q-Funktion approximiert.
3. **Anwendung von Reinforcement Learning-Techniken**, um die KI zu trainieren.
4. **Optimierung des Agenten**, um eine möglichst hohe Punktzahl im Spiel zu erreichen.

Dieses Projekt liefert nicht nur Einblicke in die Anwendung von Reinforcement Learning, sondern bietet auch eine Grundlage für weiterführende Arbeiten im Bereich der KI-gesteuerten Spiele.



2 VORARBEIT

2.1 VERWENDETE METHODEN UND TECHNOLOGIEN

In diesem Projekt wurde Reinforcement Learning mit einem Deep Q-Network (DQN) eingesetzt, um das Spiel Snake zu trainieren. Dabei basiert die Methodik auf dem Q-Learning-Ansatz, bei dem der Agent durch wiederholte Interaktionen mit der Spielumgebung optimale Aktionen erlernt. Ein neuronales Netzwerk, implementiert mit PyTorch, dient zur Approximation der Q-Werte für verschiedene Zustände. Die Architektur des Netzwerks besteht aus einer Eingabeschicht, einer versteckten Schicht und einer Ausgabeschicht, wobei die ReLU-Aktivierungsfunktion für Nicht-Linearität sorgt. Zur Optimierung der Gewichte wird der Adam-Algorithmus verwendet, während die Mean Squared Error (MSE)-Verlustfunktion zur Minimierung der Fehler eingesetzt wird.

Um die Stabilität des Lernprozesses zu gewährleisten, wird ein Replay Memory verwendet, das vergangene Spielzüge speichert. Diese gespeicherten Erfahrungen werden erneut für das Training genutzt, um Korrelationen zwischen aufeinanderfolgenden Aktionen zu reduzieren. Die Exploration-Exploitation-Balance wird durch eine Epsilon-Greedy-Strategie gesteuert, bei der der Agent zu Beginn des Trainings vermehrt zufällige Aktionen ausprobiert, um verschiedene Strategien zu erkunden. Im weiteren Verlauf des Lernprozesses nimmt der Zufallsfaktor ab, sodass das Modell zunehmend auf bereits erlernte Entscheidungen zurückgreift.

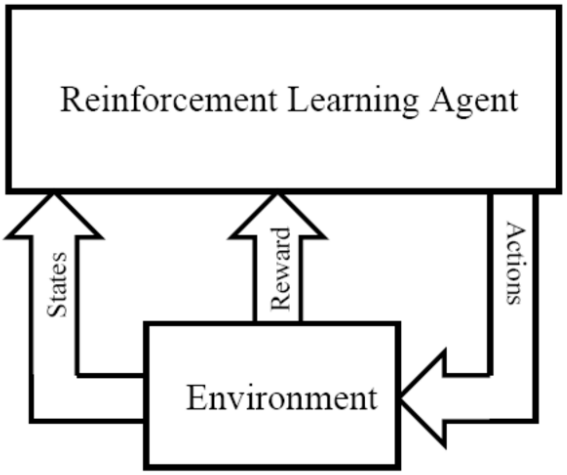
Die Spielmechanik selbst wurde mit der Bibliothek Pygame umgesetzt. Dabei umfasst die Implementierung wesentliche Elemente wie die Steuerung der Spielfigur, die Kollisionserkennung und das Belohnungssystem. Um den Lernfortschritt des Agenten zu analysieren, werden die erzielten Punktzahlen und Durchschnittswerte mit Matplotlib visualisiert. Dies ermöglicht eine detaillierte Bewertung der Leistungsentwicklung des Modells über mehrere Trainingsdurchläufe hinweg.

2.2 VERWENDETE DATENSÄTZEN

Das Projekt basiert nicht auf externen Datensätzen, sondern nutzt ausschließlich Spieldaten, die während des Trainingsprozesses dynamisch generiert werden. Während der Interaktionen mit der Spielumgebung speichert der Agent relevante Informationen, darunter den aktuellen Zustand, die gewählte Aktion, die erhaltene Belohnung und den daraus resultierenden Folgezustand. Diese Daten werden in einem Replay Memory abgelegt und für das kontinuierliche Training des neuronalen Netzwerks verwendet. Da keine statischen Datensätze vorliegen, erfolgt das Lernen vollständig durch die eigene Erfahrung des Agenten. Das Modell passt seine Strategie kontinuierlich an, indem es aus den gesammelten Spieldaten lernt und sich iterativ verbessert. Diese Art der Live-Datengenerierung ermöglicht eine flexible und adaptive Optimierung des Spielverhaltens, wodurch der Agent langfristig eine effiziente Spielstrategie entwickeln kann.

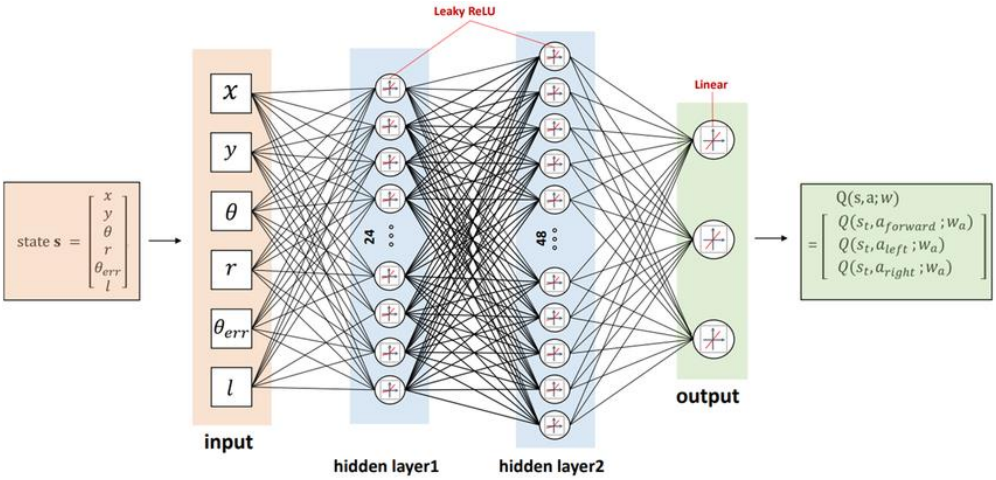
3 MEINE ARBEIT

Meine Arbeit besteht im Wesentlichen aus dem Training des Netzwerks und der Durchführung der Spielsimulationen.



3.1 NETZWERK TRAINIEREN

Das Training des neuronalen Netzwerks erfolgt in mehreren Schritten, wobei verschiedene Python-Module für spezifische Aufgaben genutzt werden. Die Hauptbestandteile umfassen die Datenverwaltung, die Netzwerkarchitektur, das Training sowie die Hauptausführung des Modells.



3.2 CODE- ERKLÄRUNG

3.2.1 Importieren von Bibliotheken

Im Code werden eine Reihe von Bibliotheken importiert, die für das Spiel und das Q-Learning-Modell benötigt werden. pygame sorgt dafür, dass das Spiel funktioniert, indem es die Grafiken zeichnet, die Spielschleife verwaltet und Benutzereingaben bearbeitet. Mit random werden zufällige Ereignisse erzeugt, wie das Platzieren des Futters an zufälligen Positionen. Die Bibliothek numpy hilft dabei, Arrays zu verarbeiten, was für die Berechnungen und den Zustand des Spiels wichtig ist. Sie sorgt dafür, dass Spielzustände und Aktionen kompakt und effizient repräsentiert werden. PyTorch, ist eine Deep-Learning-Bibliothek, die zum Trainieren des Q-Learning-Modells verwendet wird. Sie hilft, ein neuronales Netzwerk zu erstellen und die Q-Werte zu berechnen. Schließlich wird matplotlib.pyplot verwendet, um Diagramme zu erstellen, die den Fortschritt des Agenten im Spiel visualisieren. So lässt sich die Leistung des Agenten im Laufe der Zeit beobachten.

3.2.2 Die Direction-Klasse und Point-Tupel

Die Direction-Klasse ist ein Enum (Aufzählung), das vier mögliche Bewegungsrichtungen für die Schlange definiert: RIGHT, LEFT, UP und DOWN . Diese Richtungen helfen dem Spiel, die Bewegungen der Schlange zu steuern und sind wichtig für das Q-Learning-Modell, das lernt, wie die Schlange am besten navigiert. Das Point-Tupel ist ein benanntes Tupel, das die x- und y-Koordinaten eines Punktes im Spiel repräsentiert. Es wird sowohl für die Position der Schlange als auch für das Futter verwendet, sodass die beiden mit ihren jeweiligen Koordinaten auf dem Spielfeld eindeutig identifiziert werden können.

3.2.3 Die SnakeGameAI-Klasse

Die SnakeGameAI-Klasse stellt das Spielfeld und die grundlegende Spielmechanik bereit. Die Methode __init__ ist der Konstruktor der Klasse, der dafür verantwortlich ist, das Spielfeld zu initialisieren, die Anzeige zu konfigurieren und den Zustand des Spiels festzulegen. Dazu gehört das Festlegen der Position der Schlange und des Futters sowie die Initialisierung von wichtigen Parametern wie dem Punktestand und den Levelinformationen. Die Methode reset setzt das Spiel zurück. Sie sorgt dafür, dass die Schlange und das Futter erneut platziert werden und der Punktestand sowie die Levelinformationen zurückgesetzt werden. Diese Methode wird auch nach jedem Spielende aufgerufen, um das Spiel für eine neue Runde vorzubereiten.

3.2.4 Das Q-Learning-Modell (Linear_QNet)

Das Linear_QNet ist ein einfaches neuronales Netzwerk, das speziell für das Q-Learning entwickelt wurde, um Q-Werte für die verschiedenen möglichen Bewegungen der Schlange zu berechnen. Es besteht aus drei Hauptkomponenten: einer Eingabeschicht, einer verborgenen Schicht und einer Ausgabeschicht. Die input_size legt fest, wie viele Eingabewerte dem Netzwerk zugeführt werden, was hier die Zustände der Schlange und des Futters umfasst. Die hidden_size gibt die Anzahl der Neuronen in der mittleren Schicht an, welche dafür zuständig sind, komplexe Beziehungen und Muster zwischen den Eingabewerten zu lernen. Mit der output_size wird die Anzahl der Aktionen definiert, die der Agent ausführen kann – in diesem Fall sind es drei: nach rechts, nach links und nach oben oder unten. Die forward-Methode beschreibt, wie die Eingabedaten durch das Netzwerk fließen. Nachdem sie die erste Schicht durchlaufen haben, wird eine ReLU-Aktivierungsfunktion verwendet, um nichtlineare Zusammenhänge zu lernen, bevor die Daten an die Ausgabeschicht weitergegeben werden.

3.2.5 Das QTrainer-Objekt

Der QTrainer ist für das Training des Q-Learning-Modells zuständig. Die Methode train_step ist der zentrale Prozess im Training, der die Fehlerberechnung und das Anpassen der Modellgewichte umfasst. Zunächst gibt das Modell eine Vorhersage der Q-Werte für jede mögliche Aktion zurück. Anschließend wird der Ziel-Q-Wert berechnet, der auf der Belohnung und den zukünftigen Q-Werten basiert. Wenn das Spiel noch nicht beendet ist, fließt die Belohnung mit der maximalen Vorhersage des nächsten Zustands in die Berechnung ein, was den Zielwert aktualisiert. Der Verlust (Fehler) wird dann mit einer mittleren quadratischen Fehlerfunktion (MSE) berechnet. Um das Modell zu optimieren, wird die Backpropagation verwendet, um die Gradienten zu berechnen und die Gewichte des Modells anzupassen. So lernt das Modell, wie es seine Aktionen optimieren kann, um die Belohnung zu maximieren und das Spiel besser zu spielen.

3.2.6 Der Agent

Der Agent ist das zentrale Element, das das Q-Learning durchführt, indem er das Spiel spielt und lernt, welche Aktionen er basierend auf dem aktuellen Zustand des Spiels ausführen sollte. Dabei lernt er, die besten Entscheidungen zu treffen, um seine Belohnung zu maximieren. Eine wichtige Methode, die der Agent verwendet, ist die get_state-Methode. Diese Methode erstellt einen Zustand, der als Eingabe für das Modell dient und umfasst wichtige Informationen wie die Position der Schlange, ihre Bewegungsrichtung sowie die Position des Futters auf dem Spielfeld.

3.2.7 Trainingslogik

In der Trainingslogik wird dies durch die train-Methode realisiert, die die Hauptschleife des Trainingsprozesses darstellt. Hier spielt der Agent das Spiel und lernt durch Q-Learning, indem er Erfahrungen sammelt und diese speichert. Während des Spiels trifft der Agent Entscheidungen, die auf dem aktuellen Zustand basieren, und führt dann die entsprechende Aktion aus. Nachdem die Aktion ausgeführt wurde, wird die Belohnung ermittelt, und der neue Zustand des Spiels wird berechnet. Diese gespeicherten Erfahrungen werden später dazu verwendet, das Modell langfristig zu verbessern und zu trainieren.

4 RÜCKBLICK

4.1 Was lief gut und wo gab es Schwierigkeiten?

Das Projekt verlief insgesamt positiv, besonders in Bezug auf die Implementierung der Spiel- und Trainingslogik. Die Integration des Snake-Spiels mit einem Reinforcement-Learning-

Agenten funktionierte gut, und der Agent konnte durch die Verwendung des Q-Learning-Algorithmus erfolgreich lernen, wie er sich im Spiel verhalten sollte, um seine Punktzahl zu maximieren.

Die Herausforderung bestand darin, das Modell optimal zu trainieren und die richtigen Hyperparameter für das Reinforcement Learning zu finden. Der Balanceakt zwischen Exploration (random actions) und Exploitation (Verwendung des Modells für die besten Vorhersagen) stellte sich als knifflig heraus, da es Zeit und Anpassung erforderte, um das Modell stabil und effizient arbeiten zu lassen. Ein weiteres Problem war die Anpassung der Spielgeschwindigkeit basierend auf dem Level. Dies führte zu einigen unerwarteten Herausforderungen beim Training des Agenten, da das Modell sich kontinuierlich an die sich verändernde Geschwindigkeit und das wachsende Spielfeld anpassen musste.

4.2 Wie hätte man das Projekt noch fortführen können?

Ein potenzieller nächster Schritt könnte darin bestehen, das Modell weiter zu optimieren, indem zusätzliche Features in den Zustand des Spiels aufgenommen werden, wie etwa die Position der Wände oder die Vermeidung von Kollisionen mit der eigenen Schlange. Auch die Einführung von fortgeschrittenen Reinforcement-Learning-Techniken wie Double Q-Learning könnte die Leistung des Agenten verbessern.

Zudem wäre es interessant, die Modell-Performance zu visualisieren und eine detailliertere Analyse des Lernprozesses vorzunehmen, indem man Metriken wie die Lernkurve, die Anzahl der gewonnenen Spiele und die durchschnittliche Punktzahl über die Zeit verfolgt. Langfristig könnte man auch versuchen, das System mit mehreren Agenten oder in einer Multi-Agenten-Umgebung zu erweitern.

5 FAZIT

Insgesamt war das Projekt eine sehr gute Gelegenheit, die Grundlagen des Reinforcement Learnings praktisch anzuwenden und zu verstehen, wie man einen Agenten trainiert, um in einer komplexen Umgebung zu agieren. Trotz einiger Herausforderungen war es ein spannender Lernprozess, und das Modell zeigte positive Fortschritte, indem es zunehmend bessere Strategien im Spiel entwickelte. Der nächste Schritt wird darin bestehen, die Leistung des Agenten weiter zu verbessern und neue Herausforderungen hinzuzufügen, um das System robuster und vielseitiger zu gestalten.

6 QUELLEN

Open-Source-Projekte und Frameworks:

PyTorch-Dokumentation:
https://pytorch.org/docs/

Pygame-Dokumentation:
https://www.pygame.org/docs/

Schlüsseltechnologien und Konzepte im Code

Q-Learning (Watkins, 1989):
Der Code basiert auf diesem klassischen Algorithmus, bei dem der Agent durch Belohnungen lernt, optimale Aktionen auszuführen.
https://link.springer.com/article/10.1007/BF00992698

Experience Replay (Lin, 1992):
Die Speicherung und Wiederverwendung von Erfahrungen zur Stabilisierung des Trainings ist eine zentrale Methode in dem Code.
https://link.springer.com/article/10.1007/BF00992698

Epsilon-Greedy-Strategie:
Diese Methode wird verwendet, um ein Gleichgewicht zwischen **Exploration** und **Exploitation** zu erreichen.
https://link.springer.com/article/10.1007/BF00992698