TECHNISCHE HOCHSCHULE MITTELHESSEN | CAMPUS GIESSEN | MNI Mathematik, Naturwissenschaften und Informatik

Projekt 4: Pygame mit RFL

Grundlagen der Data Science

THM - SS25 - 11.07.2025

Dozent: Prof. Dr. Frank Kammer

Gruppe: Belmin Mulahusic (5371620)
Tom Graf (5406537)
Link zum GitHub Repository: https://github.com/belminmulahusic/AI-snake-game

Inhaltsverzeichnis

1 Einleitung

Reinforcement Learning (RFL) ist eine Methode aus dem Bereich der Künstlichen Intelligenz, bei der ein Agent durch Belohnungen und Bestrafungen lernt, in einer Umgebung möglichst gute Entscheidungen zu treffen. In diesem Projekt wurde dafür das Spiel Snake ausgewählt. Die Regeln sind sehr einfach: Der Agent (die Schlange) soll so viele Äpfel wie möglich einsammeln, ohne dabei mit Hindernissen oder sich selbst zu kollidieren. Genau diese wenigen Spielregeln und das einfache Belohnungssystem machen Snake zu einem guten Beispiel für eine eigene RFL-Umgebung. Für die Entscheidungsfindung des Agenten wurde Deep Q-Learning (DQN) verwendet. Dabei handelt es sich um eine Weiterentwicklung des klassischen Q-Learnings, bei dem ein neuronales Netzwerk genutzt wird, um den sogenannten Q-Wert (der erwartete Nutzen einer Aktion) vorherzusagen. Das normale Q-Learning kommt bei komplexeren Umgebungen schnell an seine Grenzen, da dort riesige Tabellen entstehen, die alle Zustände und Aktionen speichern müssen. DQN umgeht dieses Problem indem es ein neuronales Netzwerk verwendet.

Das Spiel wurde zunächst mit der Pygame Bibliothek in einer ersten einfachen Version nachgebaut. Um die Bedingungen der Aufgabe 1 zu erfüllen, mussten wir nur ein Snake Spiel, mit einer festen Anzahl an Äpfeln, an festen Positionen, ohne Hindernisse, bauen. Diese Version sollte nur für uns Menschen spielbar sein und brauchte noch keinen RFL-Agenten. Diese einfache Anfangsversion unseres Projekts diente dann als Grundlage für die weiteren Aufgaben. In der zweiten Aufgabe kam die Gymnasium Bibliothek zum Einsatz. Hier mussten wir die konkrete RFL-Umgebung definieren. Auch die Strafen und Belohnungen mussten hier definiert werden. Das konkrete Konzept der Strafen und Belohnungen wird im Laufe dieser Arbeit genauer erläutert. In einem nächsten Schritt wurden Aufgabe 3 und 4 gleichzeitig gelöst. Um den Agenten trainieren zu können, kamen die Stable-baselines 3 Bibliothek und der DQN-Algorithmus zum Einsatz. In diesem Teil des Projekts wurde der Agent auf die fixe und einfache Umgebung der ersten beiden Aufgaben trainiert. In der letzten Aufgabe wurde die Komplexität der Umgebung nochmal deutlich erhöht. Die Äpfel spawnen nun in zufälligen Positionen und neben den äußeren Wänden des Spielfelds kommen auch zufällige Hindernisse mitten im Spielfeld vor.

Im Laufe der Entwicklung sind wir iterativ vorgegangen, um unseren Agenten kontinuierlich zu verbessern. Dabei wurde das Modell nach und nach erweitert und komplexer, bis es für uns ein gutes Leistungsniveau erreicht hat. Im weiteren Verlauf der Dokumentation gehen wir zunächst auf unsere RFL-Umgebung und unser Belohnungssystem genauer ein. Anschließend betrachten wir das Training und den Trainingsverlauf des Agenten genauer. Danach folgt ein Abschnitt zur Transparenz und Interpretierbarkeit unseres DQN-Modells, in dem wir versuchen, dessen Entscheidungen nachvollziehbar zu machen. Im Anschluss folgt die Bewertung unseres Projekts mit 3 beispielhaften Ausführungen, den Herausforderungen und der eigentlichen statistischen Evaluation. Abschließend fassen wir die wichtigsten Erkenntnisse in einem Fazit zusammen und geben einen kurzen Ausblick auf mögliche Verbesserungen und Weiterentwicklungen.

2 Installation und Ausführung

Zur Nutzung des Projekts ist die Installation einiger Programme erforderlich. Diese können mittels `pip install -r requirements.txt` installiert werden. Dabei ist auf die korrekten Versionen der Pakete zu achten: `Gymnasium` in Version `1.1.1`, `Pygame` `2.5.2`, `stable-baselines3` `2.6.0`, `NumPy` `2.1.3` sowie `Python` `3.12.3`.

```
1  gymnasium==1.1.1
2  numpy==2.1.3
3  pygame==2.5.2
4  stable_baselines3==2.6.0
5  tensorboard==2.19.0
```

Listing 2.1: Inhalt der requirements.txt

Das Projekt ist in mehrere Dateien mit unterschiedlichen Funktionen unterteilt. Durch Ausführen der Datei `train_env.py` wird ein neues DQN-Modell trainiert. Dieses Modell wird unter dem Namen `dqn_snake_model.zip` im Projektordner gespeichert, ebenso wie die zugehörige TensorBoard-Ausgabe unter `dqn_snake_tensorboard`.

Über `game.py` kann das normale Spiel mit zufällig positionierten Äpfeln und zufälligen Hindernissen gestartet werden. Im Hauptmenü lässt sich über `Play` ein manuell steuerbares Snake-Spiel starten, wobei per Leertaste jederzeit der KI-Modus aktiviert werden kann, sodass das DQN-Modell die Steuerung übernimmt. Über den Menüpunkt `Run AI Model` wird direkt ein Durchlauf vom Modell gespielt.

Die Datei `evaluate.py` ermöglicht eine umfassende Evaluation des Modells. Beim Programmaufruf muss der Pfad zum gewünschten DQN-Modell übergeben werden. Zusätzlich kann über den Parameter `--episodes` festgelegt werden, wie viele Durchläufe durchgeführt werden sollen. Mit dem Argument `--render` lässt sich optional die grafische Darstellung aktivieren. Die Ausgabe des Programms umfasst unter anderem die durchschnittlich erreichte Punktzahl, den Median, die Standardabweichung sowie die Anzahl an Spielen, die aufgrund von Endlosschleifen abgebrochen wurden. Des Weiteren kann man sich mit dem `-q` Flag die Q-Werte der Entscheidungen des Agenten anzeigen lassen. Außerdem kann man sich das Modell auch mit der `-dqn` Option anschauen.

Die Datei `main.py` startet das Spiel im manuellen Modus, ohne Hindernisse und mit immer gleich platzierten Äpfeln. In der Datei `snake_env.py` ist die Reinforcement-Learning-Umgebung definiert. Beim alleinigen Ausführen dieser Datei wird lediglich die Umgebung initialisiert, ohne dass eine Interaktion oder Ausgabe erfolgt.

3 RFL-Umgebung

Bei der *Gymnasium*-Bibliothek handelt es sich um eine Open-Source-Bibliothek, die zur standardisierten Umsetzung von Reinforcement-Learning-Umgebungen genutzt wird. Sie stellt eine einheitliche Schnittstelle bereit, über die gängige Lernalgorithmen wie DQN, PPO oder A2C effizient mit der Umgebung interagieren können. Ziel ist es, den Austausch zwischen Agent und Umgebung zu vereinfachen und die Entwicklung sowie das Testen von Lernalgorithmen zu verbessern. Um eine eigene Umgebung mit *Gymnasium* zu realisieren, muss zunächst eine Klasse erstellt werden, die von der Klasse `gymnasium.Env` der Bibliothek erbt. Damit *Gymnasium* mit der Umgebung korrekt arbeiten kann, müssen einige zentrale Methoden implementiert werden, die festlegen, wie das Spiel funktioniert, wie Zustände aussehen, wie Aktionen behandelt werden und wie die Belohnungen vergeben werden [1].

Die wichtigste Methode zu Beginn ist `__init__()`, in der das Spiel initialisiert und alle notwendigen Variablen definiert werden, die beim Start des Spiels gesetzt werden müssen. Besonders relevant sind hier die Variablen `action_space` und `observation_space`. Dabei handelt es sich um die Bewegungsmöglichkeiten und die Umgebungsinformationen, die dem Agenten zur Verfügung stehen. Die Bewegungsmöglichkeiten bestimmen, wie viele verschiedene Aktionen dem Agenten zu jedem Zeitpunkt zur Auswahl stehen. In unserem Fall besteht der `action_space` aus drei Optionen. Der Agent kann aus seiner Perspektive nach links, nach rechts oder geradeaus weiterlaufen. Die Umgebungsinformationen umfassen alle Daten, die der Agent bei jedem Schritt von der Umgebung erhält, um darauf basierend Entscheidungen zu treffen. Dazu zählen beispielsweise Informationen über Hindernisse, die Position des Apfels oder die eigene Bewegungsrichtung. Die genaue Struktur dieser Beobachtungen ist ein zentrales Element des Lernprozesses und wird daher ausführlich im nächsten Unterkapitel behandelt.

Aktion	**Bedeutung**
1	Geradeaus gehen
2	Nach links abbiegen
3	Nach rechts abbiegen

Tabelle 3.1: Definition des Action Space

Die Methode `reset()` wird zu Beginn jeder Episode aufgerufen und setzt das Spiel wieder auf den Anfangszustand zurück. Sie liefert anschließend die erste Beobachtung an den Agenten. Ein weiteres zentrales Element ist die Methode `step()`, da sie die eigentliche Spiellogik abbildet und es dem Agenten ermöglicht, gezielt Aktionen in der Umgebung auszuführen. Nach der Durchführung einer Aktion bekommt der Agent von dieser Methode Feedback, wodurch er lernen kann. Zur besseren Nachvollziehbarkeit kann außerdem eine `render()`-Methode implementiert

werden, die den aktuellen Zustand der Umgebung grafisch darstellt, um ihn besser nachvollziehbar zu machen. Die letzte zu implementierende Methode heißt `close()`. Sie wird ganz am Ende aufgerufen und gibt die Ressourcen wieder frei.

Durch diese klar vorgegebene Struktur der *Gymnasium*-Bibliothek ergibt sich eine hohe Kompatibilität mit unterschiedlichen Algorithmen und es wird so die Entwicklung, das Testen und der Vergleich verschiedener Reinforcement-Learning-Ansätze erheblich erleichtert.

3.1 Observation Space

Der Observation Space, also der Beobachtungsvektor, den der Agent bei jedem Schritt vom Spiel erhält, setzt sich aus insgesamt sechs verschiedenen Komponenten zusammen, die alle eine wichtige Rolle für das Lernverhalten spielen. Diese werden dem Agenten als Vektor, bestehend aus 182 Komponenten, übergeben.

Zunächst wird die aktuelle Bewegungsrichtung der Schlange übergeben, also ob sie sich nach oben, unten, links oder rechts bewegt. Die Information ist essenziell, damit der Agent weiß, wohin er sich gerade bewegt, und darauf basierend sinnvolle Entscheidungen treffen kann. Die zweite Komponente beschreibt die relative Position des Apfels zum Kopf der Schlange. Dadurch kann der Agent erkennen, in welche Richtung er sich bewegen muss, um den Apfel zu erreichen und somit eine Belohnung zu erhalten. Außerdem enthält der Observation Space die normierte Länge der Schlange. Diese Information hilft dem Agenten, die Spielsituation besser einzuschätzen und sein Verhalten beispielsweise bei zunehmender Länge der Schlange anzupassen. Ein besonders zentraler Bestandteil ist das Sichtfeld der Schlange. Hierbei handelt es sich um ein 13 mal 13 großes Raster rund um den Kopf der Schlange. In diesem Raster sind Hindernisse, Wände oder der eigene Körper als Gefahren markiert. Diese räumliche Übersicht soll es dem Agenten ermöglichen, gefährliche Kollisionen frühzeitig zu erkennen und zu vermeiden. Insbesondere in späteren Spielphasen, wo der Körper der Schlange sehr lang ist und der weitere Spielverlauf sehr komplex ist, ist dieses Raster sehr nützlich. Zusätzlich bekommt der Agent die normierte Distanz zum Apfel übermittelt. Diese Information ermöglicht eine präzisere und effizientere Navigation als eine rein binäre Richtungsangabe. Zuletzt werden auch die letzten drei ausgeführten Aktionen übergeben. Diese Information hilft dem Agenten dabei, wiederkehrende Bewegungsmuster zu erkennen, zum Beispiel das ungewollte Schleifen der Schlange, und zu verhindern.

Name	**Bedeutung**	**Anzahl an Elementen**
dir_vec	Bewegungsrichtung der Schlange	4
food_vec	Richtung des nächsten Apfels	4
snake_length_vec	Länge der Schlange	1
apple_dist_vec	Entfernung zum nächsten Apfel	2
action_history_vec	Letzte Aktionen der Schlange	3
fov_vec	Sichtfeld der Schlange	168

Tabelle 3.2: Struktur des Observation Space

Insgesamt ermöglichen diese sechs Komponenten dem Agenten, das Spielgeschehen in seiner Gesamtheit zu erfassen und fundierte Entscheidungen zu treffen. Im nächsten Abschnitt wird das Belohnungs- und Bestrafungssystem genauer erläutert.

3.2 Belohnungssystem

Ein wichtiger Bestandteil beim Trainieren eines Agenten im Reinforcement-Learning ist das Belohnungssystem. Es legt fest, wie das Verhalten des Agenten in unterschiedlichen Situationen bewertet wird und hat damit einen entscheidenden Einfluss auf den Lernprozess. Eine ausgewogene Gestaltung von Belohnungen und Bestrafungen ist daher unerlässlich, um ein effektives Modell zu entwickeln. Im folgenden Abschnitt wird das in diesem Projekt eingesetzte Reward-System im Detail vorgestellt. Das Ziel dieses Systems besteht darin, das Verhalten des Agenten im Sinne des Spiels zu steuern und zu optimieren. Dabei werden alle Aktionen belohnt, die den Agenten direkt oder indirekt dazu bringen, einen Apfel zu erreichen und somit einen Punkt zu erzielen. Gleichzeitig werden Aktionen, die eine gegenteilige Auswirkung haben, bestraft.

Die Definition des Wertebereichs für Belohnungen und Bestrafungen ist ein kritischer Punkt bei der Erstellung eines guten Belohnungssystems. Eine geeignete Skalierung ist entscheidend, damit das Modell die Belohnungen korrekt einordnen kann und keine übermäßigen Gewichtungen entstehen, die zu unerwünschtem Verhalten führen. Wir haben deshalb zunächst eine Skalierung gewählt, die sich an dem Beispiel aus Kapitel 6 der Vorlesungsfolien orientiert, mit einem Wertebereich von +100 (für das Sammeln eines Apfels) bis -10000 (für die Kollision mit sich selbst oder der Wand). Dabei zeigte sich jedoch schnell, dass das Modell kaum zielgerichtetes Verhalten lernte und die Schlange entsprechend sehr früh mit sich selbst oder einer Wand kollidiert ist. Im weiteren Verlauf haben wir verschiedene Skalierungen ausprobiert und uns schließlich für einen kleineren Wertebereich von -1.0 bis +1.0 entschieden. In unseren Tests erwies sich dieser Wertebereich als der stabilste und lieferte die verlässlichsten und konstantesten Ergebnisse. Das Einsammeln eines Apfels sollte grundsätzlich stark belohnt werden, da es das primäre Ziel des Agenten darstellt. Der damit verbundene Anreiz muss in jeder Situation wirksam sein, selbst dann, wenn der Apfel noch weit entfernt ist oder zunächst Umwege nötig sind, um den Apfel zu erreichen. Deshalb haben wir uns dazu entschieden, die Belohnung für das Sammeln eines Apfels auf +1.0 festzulegen. Durch diesen bewusst hoch gewählten Wert wird das Sammeln von Äpfeln als Hauptziel des Spiels definiert.

Der Agent erhält außerdem eine Belohnung in Höhe von +0.1, wenn er sich in Richtung des Apfels bewegt, und eine Bestrafung in Höhe von - 0.05, wenn er sich von diesem entfernt. Diese Belohnungen schaffen einen kontinuierlichen Anreiz, sich zu einem Apfel hin zu bewegen, auch wenn dieser noch sehr weit entfernt ist und deshalb die Belohnung für das Sammeln des Apfels noch schwach gewichtet wird. Die Bestrafung für das Entfernen wurde dabei bewusst geringer gewählt als die Belohnung für das Annähern, um den Agenten nicht unangemessen zu benachteiligen, wenn er sich in Situationen befindet, in denen er aufgrund seiner Länge komplexere Bewegungen ausführen muss. Insbesondere bei höherem Punktestand kann es erforderlich sein, sich vorübergehend vom Apfel zu entfernen, um eine geeignete Position für den nächsten Annäherungsschritt zu erreichen.

Grundsätzlich bekommt der Agent zusätzlich eine kleine Belohnung in Höhe von +0.005 für jeden Schritt, den er ausführt. Die Idee hinter dieser Belohnung ist ein Überlebensbonus, welcher den Agenten dazu motivieren soll, nicht ausschließlich auf dem schnellsten Weg zum nächsten Apfel zu gehen, sondern auch generell möglichst lange zu überleben. Dadurch soll eine etwas vorsichtigere Spielweise gefördert werden, bei der der Agent lernt nicht gegen Wände zu laufen und, insbesondere in späteren Spielsituationen, mit höherer Länge der Schlange, nicht mit sich selbst zu kollidieren. Falls kein neuer Apfel mehr platziert werden kann, weil die Schlange das gesamte Spielfeld einnimmt, gilt das Spiel als gewonnen. In diesem Fall erhält der Agent eine Belohnung von +1.0, um deutlich zu machen, dass das Beenden des Spiels in diesem Fall ein sehr erstrebenswertes Ziel ist.

Für alle anderen Aktionen, die zum Beenden des Spiels führen, muss der Agent stark bestraft werden, da diese zu einer Niederlage führen. Solche Aktionen sind das Kollidieren mit sich selbst, dem Rand des Spielfelds oder einem Hindernis. Deshalb haben wir uns dazu entschieden, die Bestrafung auf -1.0 festzulegen. Durch diese sehr harte Bestrafung soll der Agent auch davon abgehalten werden, das Spiel absichtlich zu beenden. Im Fall, dass der Agent in einer Endlosschleife gefangen ist, wird dieses Verhalten ebenfalls mit -1.0 bestraft.

Aktion	**Belohnung**
Apfel gefressen	+1.0
Letzten Apfel gefressen (Spielfeld voll)	+1.0
Abstand zum Apfel verringert	+0.1
Überlebensbonus pro Schritt	+0.005
Abstand zum Apfel vergrößert	-0.05
Kollision mit Wand oder sich selbst	-1.0
Kein Apfel nach über 500 Schritten	-1.0

Tabelle 3.3: Belohnungsübersicht des Reward-Systems

4 Training

Nachdem die Umgebung und die Grundlagen des Reinforcement Learnings erklärt wurden, geht es nun darum, wie das Training genau abläuft. Dafür wird die verwendete Bibliothek beschrieben und es werden die wichtigsten Hyperparameter des DQN-Algorithmus vorgestellt.

In diesem Projekt wurde für das Training des Snake Agenten die Bibliothek Stable-Baselines 3 verwendet. Dabei handelt es sich um eine weit verbreitete Python Bibliothek für Reinforcement Learning, die auf PyTorch basiert und mehrere moderne Algorithmen wie DQN, PPO oder A2C implementiert. Sie eignet sich gut für dieses Projekt, weil sie direkt mit Gymnasium Umgebungen kompatibel ist. Bevor das Training beginnen kann, mussten wir, wie in Aufgabe 3 beschrieben, die Umgebung mithilfe von `check_env()` überprüfen, um sicherzustellen, dass alle Gymnasium Schnittstellen korrekt implementiert sind. Bei der Auswahl und dem Verständnis der Hyperparameter haben wir auf KI-Tools wie ChatGPT und Gemini zurückgegriffen, sowie auf die Dokumentation der Stable-Baselines 3 Bibliothek [2]. Diese halfen uns vor allem dabei, die Bedeutung der einzelnen Parameter besser nachzuvollziehen und deren Auswirkungen auf das Training besser zu verstehen. Aufgrund dieser Erfahrungen haben wir unsere finalen Parameter durch Ausprobieren bestimmen können. Die im Folgenden aufgeführten Einstellungen haben sich letztlich als am effektivsten erwiesen und lieferten die besten Ergebnisse.

```
    model = DQN(
    "MlpPolicy",
    env,
    learning_rate=5e-4,
    buffer_size=1_000_000,
    learning_starts=200_000,
    batch_size=64,
    target_update_interval=20_000,
    exploration_fraction=0.2,
    exploration_final_eps=0.02,
    verbose=1,
    tensorboard_log="./dqn_snake_tensorboard_182_10m/"
    )
```

Listing 4.1: Hyperparameter des DQN-Modells

Da die Aufgabenstellung Deep Q-Learning vorsieht, haben wir als Policy die „MlpPolicy“ aus Stable-Baselines 3 verwendet. Für die Lernrate haben wir einen Wert von 0,0005 gewählt. Da die Lernrate im Grund bestimmt, wie stark das Modell seine Gewichte nach jedem Lernschritt anpasst, hat sich dieser Wert nach mehreren Versuchen als wirksam herausgestellt.

Der Replay Buffer wird verwendet, um Erfahrungen aus dem Spielverlauf zu speichern und später wiederzuverwenden. Eine solche Erfahrung besteht typischerweise aus dem aktuellen Zu-

stand, der ausgeführten Aktion, der erhaltenen Belohnung sowie dem resultierenden Folgezustand. Der Vorteil dieses Verfahrens liegt im zufälligen Ziehen aus dem Buffer. Anstatt, dass direkt aufeinanderfolgende, oft sehr ähnliche Situationen das Training dominieren, sorgt das zufällige Ziehen von Erfahrungen für eine gleichmäßigere und vielfältige Verteilung der Trainingsdaten. Das macht das Lernen robuster und stabiler. Wir haben uns für eine Buffer-Größe von einer Million Einträgen entschieden. Zudem beginnt das eigentliche Training erst, nachdem der Replay Buffer mit 200.000 zufälligen Aktionen gefüllt wurde. So stellen wir sicher, dass der Agent auf einer ausreichenden Basis an Erfahrungen lernt und nicht zu früh mit zu wenig Daten trainiert wird.

Die Batch Size gibt an, wie viele Erfahrungen pro Trainingsschritt zufällig aus dem Replay Buffer entnommen werden. Ein ausgewogen gewählter Wert ist dabei entscheidend. Ist die Batch Size zu klein, kann das Training instabil werden und starke Schwankungen im Lernverhalten verursachen. Eine zu große Batch Size hingegen erhöht den Rechenaufwand erheblich. Wir haben uns daher für einen Wert von 64 entschieden, da er in unserem Fall ein gutes Gleichgewicht zwischen Stabilität und Effizienz bietet.

Eine `exploration_fraction` von 0.2 bringt den Agenten dazu, in den ersten 20% der Trainingszeit noch viel auszuprobieren, also zufällige Aktionen zu wählen, um in neue Situationen zu kommen. Diese Rate geht linear runter bis zur `exploration_final_eps`, welche auf 0.02 gesetzt ist. Dies bedeutet, dass der Agent auch später im Training mit einer Wahrscheinlichkeit von 2% eine zufällige Aktion wählt, wodurch er besser auf unerwartete Situationen reagieren kann. Zuletzt wurde Tensorboard benutzt, um die Trainingsdaten zu visualisieren. Es ist jedoch wichtig zu beachten, dass bestimmte Hyperparameter proportional zur Gesamtzahl der Trainingsschritte gewählt werden sollten. Dazu zählen insbesondere die Größe des Replay Buffers, die `exploration_fraction`, `learning_starts` sowie das `target_update_interval`.

Einer der Vorteile von DQN ist die Einführung eines Zielnetzwerkes, welches seltener aktualisiert wird als das Hauptnetzwerk. Durch diese seltenen Aktualisierungen bleibt das Zielnetzwerk konstanter und es kommt nicht zu starken Schwankungen beim Lernen [3]. Das `target_update_interval` gibt an, nach wie vielen Trainingsschritten das Zielnetzwerk die aktuellen Gewichtungen des Hauptnetzwerks übernimmt. Ein hoher Wert führt zu einem stabileren, aber langsamer reagierenden Lernprozess. Ein niedriger Wert ermöglicht schnellere Anpassungen, erhöht jedoch das Risiko von instabilem Lernen. Daher ist eine ausgewogene Wahl dieses Parameters entscheidend für ein effektives Training. Wir haben uns für ein `target_update_interval` von 10.000 entschieden.

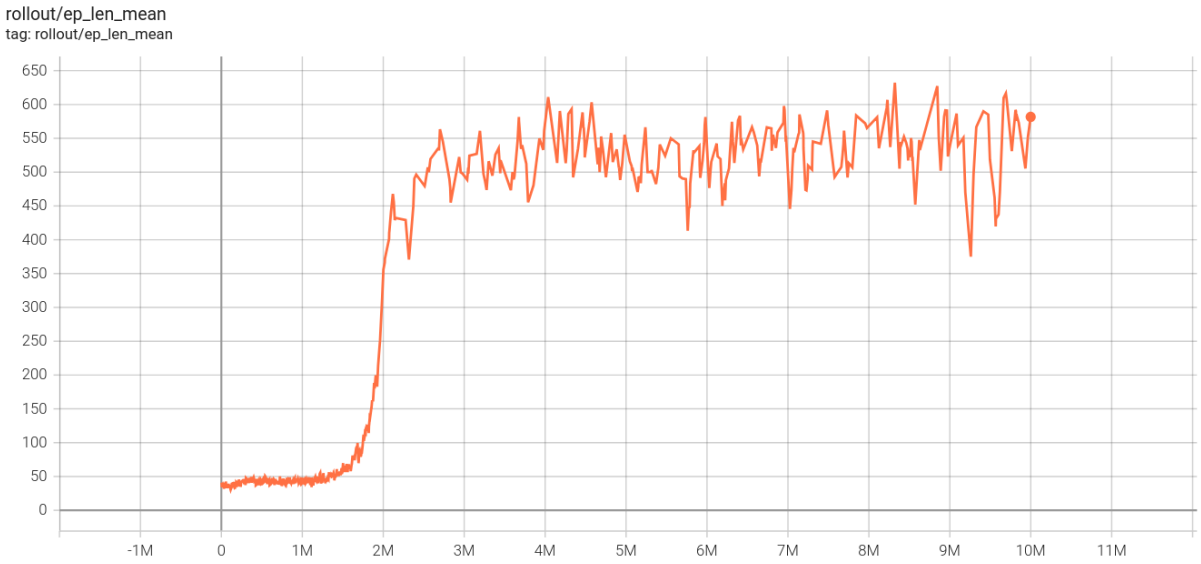

Abbildung 4.1: Verlauf der Lernkurve (Tensorboard)

Für das Training des Agenten haben wir uns letztlich für 10 Millionen Trainingsschritte entschieden. Zwar zeigte sich schon früh, dass der Agent relativ schnell lernt, da er bei Trainiervorgängen von nur 10.000 Schritten bereits den Hindernissen ausweichen und auch mehrere Äpfel einsammeln konnte. Wir vermuten, dass dies so ist, da das grundlegende Spielprinzip von Snake sehr einfach ist, weshalb erste Fortschritte schnell sichtbar waren. In Abbildung 4.1 ist außerdem zu erkennen, dass das Modell bereits ab etwa 4 Millionen Schritten sehr leistungsfähig ist. Zu diesem Zeitpunkt hätte das Training theoretisch auch beendet werden können. Dennoch haben wir uns bewusst für eine längere Trainingsdauer entschieden, da sich bei vielen Testläufen gezeigt hat, dass der Agent mit zunehmender Trainingszeit immer häufiger neue Highscores erreicht hat. Je länger er trainiert wurde, desto besser konnte er mit komplexeren Spielsituationen umgehen.

Wir haben auch mit anderen Schrittzahlen experimentiert, wie zum Beispiel 100.000, 1 Million, 5 Millionen, 20 Millionen oder auch 30 Millionen. Die Lernergebnisse haben wir mit Tensorboard visualisiert, und dabei hat sich gezeigt, dass mit den Parametern, die wir benutzen und eingestellt haben, die Lernkurve ab ca. 10 Millionen Schritten deutlich abflacht. Ab diesem Punkt wäre der zusätzliche Rechenaufwand kaum noch gerechtfertigt, da der Agent nur noch minimale Verbesserungen erzielt. Wir sind uns hier bewusst, dass es bestimmt noch weitere Kombinationen von Hyperparametern gibt, die genauso gut oder vielleicht sogar besser funktioniert hätten.

5 Erklärung des Modells

Künstliche Intelligenz in Form von Deep Reinforcement Learning wird oft als eine Blackbox betrachtet. Die Entscheidungen des Agenten erscheinen aus der Sicht des Nutzers nicht sofort nachvollziehbar, da das neuronale Netzwerk dahinter sehr komplex ist und für Menschen schwer interpretierbar ist. Das Ziel dieses Abschnitts ist es, das Verhalten des Agenten transparenter und nachvollziehbarer zu machen. Im Bereich von XAI (Explainable AI) versucht man somit Methoden zu entwickeln, die die Entscheidungsprozesse von KI-Modellen verständlich und nachvollziehbar für Menschen machen [4]. Im Rahmen dieses Projektes haben wir einige Maßnahmen ergriffen, um unseren trainierten DQN-Agenten so transparent wie möglich zu machen. Zunächst wurde die Architektur des verwendeten neuronalen Netzes offengelegt. Dies kann man in Stable-Baselines 3, nachdem man das Modell geladen hat, machen indem man `print(model.q_net)` ausführt.

```
QNetwork(
  (features_extractor): FlattenExtractor(
    (flatten): Flatten(start_dim=1, end_dim=-1)
  )
  (q_net): Sequential(
    (0): Linear(in_features=182, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=3, bias=True)
  )
)
```

Abbildung 5.1: Konsolenausgabe bei `print(model.q_net)`

Die Ausgabe zeigt, dass unser neuronales Netz aus mehreren Schichten besteht. Die erste Schicht erhält 182 Eingabewerte, die den aktuellen Zustand der Umgebung repräsentieren. Das ist also das gesamte Beobachtungsfeld, das der Agent vom Spiel erhält. Diese Eingangsschicht transformiert die 182 Eingaben in 64 Merkmale. Daraufhin folgt eine sogenannte ReLU Funktion. Diese sorgt dafür, dass negative Werte auf null gesetzt werden [5]. Dies ermöglicht dem Agenten nichtlineare Zusammenhänge zu lernen und so komplexere Muster zu erkennen. Die zweite Schicht verarbeitet die vorherigen 64 Ausgaben und liefert ebenfalls 64 Ausgangswerte, die wiederum durch eine ReLU Funktion aktiviert werden. Abschließend gibt es noch eine dritte Schicht, welche die 64 Merkmale der vorherigen Schicht auf 3 Ausgaben reduziert. Diese drei Ausgaben entsprechen den Q-Werten für die möglichen Aktionen im Spiel: Geradeaus gehen, nach links gehen, nach rechts gehen.

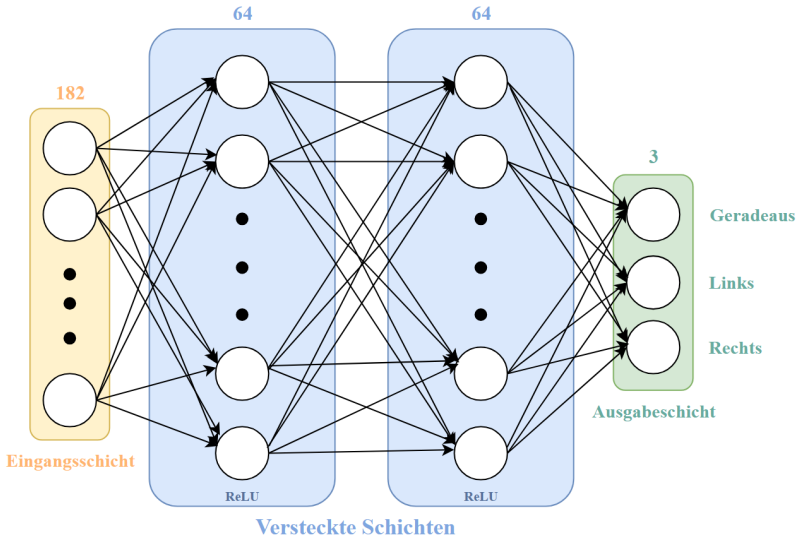

Abbildung 5.2: Übersicht über das Modell

Diese Q-Werte tragen zur Verständlichkeit des Modells bei. Um diese Werte einsehen zu können, kann man bei der `evaluate.py` das `-q` Flag benutzen. Dies gibt bei der Auswertung des Modells in der Konsole eine Ausgabe welche folgendes enthält: Die aktuellen Q-Werte für die jeweiligen Aktionen in dem aktuellen Zustand. Wohlgemerkt, dass der Zustand der Observation Space ist, also alle Infos die der Agent über die Umgebung bekommt. Darunter steht dann die aufgrund der Q-Werte ausgewählte Aktion des Agenten und der Reward, den er dafür bekommen hat. Um das richtig zu verstehen, wird im Folgenden ein Beispiel vorgezeigt.

```
------------------------------------------------------------
Q-Werte: Geradeaus = 13.21, Links = -0.52, Rechts = 13.08

Step 1283: Action = Gerade, Reward = 0.105, Score = 55
------------------------------------------------------------
```

Abbildung 5.3: Konsolenausgabe beim Ausgeben von Q-Werten

In der Ausgabe erkennt man einen besonders hohen Q-Wert für die Aktion „geradeaus“. Dies passiert vermutlich, da der Agent sich höchstwahrscheinlich auf dem Weg zum Apfel befindet und sich ihm somit weiter nähert. Wir sehen auch beim Reward, dass er den 0.005 Überlebensbonus und den 0.1 Reward bekommt, da er sich in Richtung Apfel bewegt. Gleichzeitig sieht man niedrigere Q-Werte für andere Aktionen und sogar einen negativen Q-Wert für die Aktion “Links”. Dies deutet vermutlich darauf hin, dass sich links von der Schlange ein Hindernis oder ihr eigener Körper befindet. Der Agent hat also offensichtlich gelernt, dass eine Kollision sehr schlimm ist.

Trotz dieser Maßnahmen bleibt die vollständige Interpretierbarkeit unserer KI begrenzt. Es ist weiterhin nicht möglich zu verstehen, wie genau die 182 Eingabewerte im Detail gewichtet werden order warum bestimmte interne Neuronen bestimmte Aktivierungen zeigen. Die Komplexität und Verteilung der Gewichte in den verborgenen Schichten macht das Modell im Kern weiterhin zu eine Blackbox. Dennoch konnte durch die Offenlegung der Netzwerkstruktur und die Ausgabe der Q-Werte ein erster Schritt in Richtung Erklärbarkeit und Transparenz erreicht werden.

6 Bewertung

6.1 Herausforderungen

Während des Trainings unseres Snake-Agenten sind wir auf mehrere Herausforderungen gestoßen, die teilweise sehr zeitintensiv waren. Eine der größten Schwierigkeiten war es, den Agenten davon abzuhalten, sich im Kreis zu drehen. Gerade in den ersten Trainingsphasen kam es häufig vor, dass er in eine Art Endlosschleife geriet und sich dauerhaft im Kreis oder in einer festen Bewegungsabfolge bewegte, ohne dabei den Apfel einzusammeln oder sich selbst zu töten. Nach einiger Recherche stellten wir fest, dass dieses Verhalten beim Reinforcement Learning von Snake nicht ungewöhnlich ist. Es tritt auf, wenn der Agent in einem lokalen Optimum der Q-Werte feststeckt und keine bessere Strategie mehr findet. Trotz zahlreicher Anpassungen am Reward-System, dem Observation Space und der Trainingsdauer konnten wir das Problem nicht vollständig beheben. Es gelang uns jedoch, die Häufigkeit deutlich zu verringern. Durch gezielte Veränderungen an den Belohnungen reduzierte sich dieses Verhalten auf unter 1 % der Spiele. Da wir keine Lösung fanden, mit der sich das Problem ganz beseitigen ließ, haben wir diesen seltenen Fehler letztlich akzeptiert.

Ein weiteres Problem, das immer wieder auftrat, war, dass der Agent in späteren Spielphasen regelmäßig mit sich selbst kollidierte. Meistens geschah das, weil er sich selbst „einwickelte“ und keinen Ausweg mehr fand. Zwar wurde dieses Verhalten mit zunehmendem Training besser, allerdings flachte die Lernkurve nach einer gewissen Zeit ab und es kam kaum noch zu Verbesserungen. Da unser Spielfeld 30x30 Felder groß ist und der Agent im Durchschnitt eine Länge von über 50 erreicht, während zusätzlich zufällige Hindernisse auf dem Spielfeld erscheinen, waren wir mit dem Ergebnis insgesamt zufrieden. In vielen Fällen schafft es der Agent eindrucksvoll, sich selbst aus dem Weg zu gehen und einen Umweg zum Apfel zu machen. Es kommt aber auch vor, dass er sich frühzeitig in eine ungünstige Lage bringt und dann etwa bei 10 Punkten stirbt. Dieses Verhalten ist nachvollziehbar, da es für den Agenten sehr schwierig ist, solche langfristigen Konsequenzen frühzeitig zu erkennen. Zum einen hat er nur begrenzte Informationen über die Umgebung, zum anderen erschweren die zufällige Platzierung von Hindernissen und Äpfeln das Lernen zusätzlich. Da wir mehrere Hindernisarten verwenden, ist es für den Agenten sehr schwierig mit all diesen Situationen sicher umgehen zu können.

Auch die Wahl der richtigen Hyperparameter stellte eine große Herausforderung dar. Stable-Baselines 3 bietet zwar viele Einstellungsmöglichkeiten, allerdings war zunächst unklar, welche Werte sich wie auf das Training auswirken. Ein großes Problem war dabei der hohe Rechenaufwand. Selbst kleine Änderungen an einem einzelnen Parameter mussten über mehrere Millionen Trainingsschritte getestet werden, um ein repräsentatives Ergebnis zu erhalten. Da manche dieser Änderungen starke Auswirkungen auf das Verhalten des Agenten hatten, war dieses Experimentieren sehr zeitintensiv.

Schließlich war auch das Balancing des Reward-Systems nicht einfach. Hier mussten wir viel ausprobieren, um eine sinnvolle Gewichtung von Belohnungen und Bestrafungen zu finden. Insgesamt hat uns dieses Projekt gezeigt, wie komplex es sein kann, auch für ein vermeintlich einfaches Spiel wie Snake einen gut funktionierenden Agenten zu entwickeln. Wir mussten vieles ausprobieren und anpassen, und trotzdem blieben manche Probleme bestehen.

6.2 Beispielausführungen

In folgendem Abschnitt werden 4 beispielhafte Spielsituationen vorgestellt die jeweils die Stärken und Schwächen unseres Agenten vorzeigen.

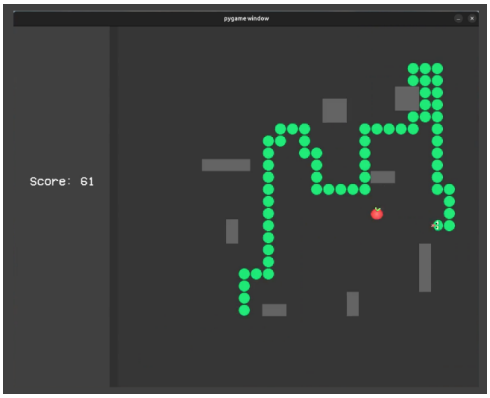

Abbildung 6.1: Screenshot aus dem Spiel

Dieses Beispiel zeigt, dass der Agent wirklich gelernt hat. Er startet als kleine Snake und bewegt sich zielgerichtet zu den Äpfeln und weicht dabei Hindernissen aus. Man sieht hier, dass das bereits sehr gut klappt, da der Agent schon eine Größe über 50 hat, was in einem 30 mal 30 Feld sehr eindrucksvoll ist. Ab einer gewissen Länge sieht man, dass der Agent große Teile des Spielfelds mit seinem Körper blockiert. Bei 61 Punkten beginnt er bereits, sich selbst den Weg zum Apfel abzuschneiden. Trotzdem findet er Wege, um mit dieser Situation umzugehen.

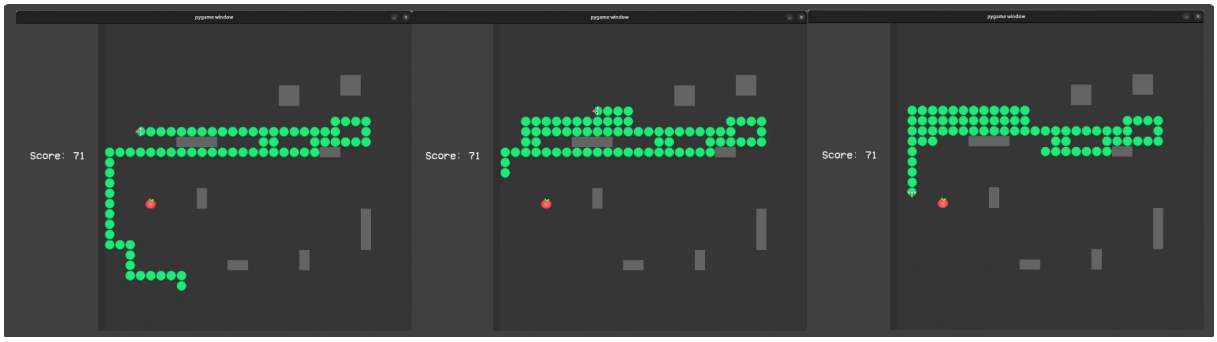

Abbildung 6.2: Ausweichmanöver des Agenten

Als er bei 71 Punkten ankam, erkennt er, dass der direkte Weg zum Apfel versperrt ist, also zieht er sich zunächst zurück und schafft sich durch Ausweichbewegungen Platz und geht dann gezielt auf den Apfel zu. Einige Schritte später ist er bereits bei 85 Punkten.

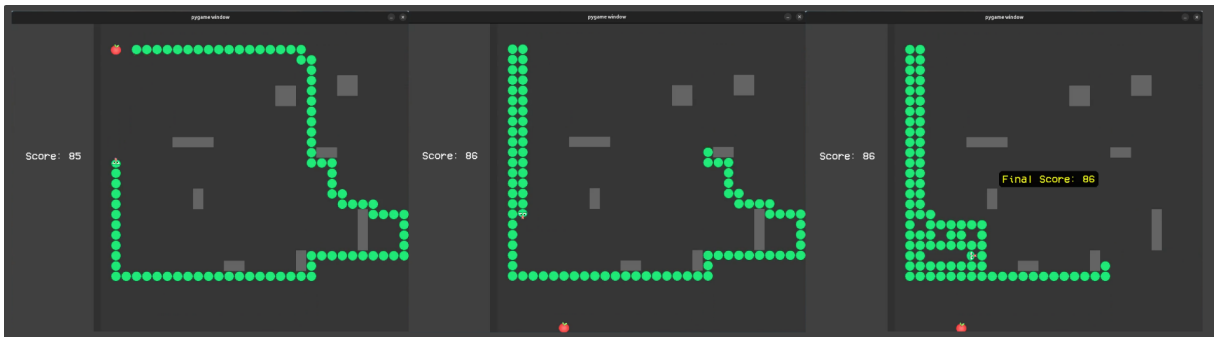

Abbildung 6.3: Komplizierte Spielsituation

Hier sieht man, dass der Agent sehr lange ist, und es wird immer schwerer, sich selbst aus dem Weg zu gehen. Hier muss angemerkt werden, dass der DQN Algorithmus nicht vorausschauend Planen kann: Als Spieler, würde man jetzt entweder links statt rechts runter gehen, und sogar beim rechts gehen würde man erstmal ein paar leere Bewegungen machen, damit man sich nicht einschlingelt. Der Agent geht jedoch gerade runter und macht eine schlechte Bewegung nach links. Somit hat er sich selbst eingekesselt und stirbt bei 86 Punkten.

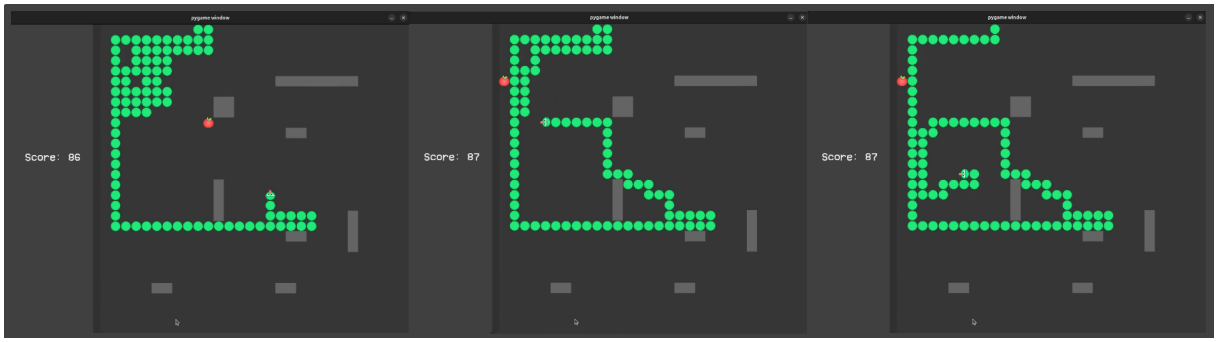

Abbildung 6.4: Komplizierte Spielsituation

Bei einem weiteren Beispiel sehen wir wieder, dass der Agent es schafft, sehr lange zu überleben und dass er bewusst mit seinem Körper umgeht und ihm ausweicht. Aber hier, genau wie im Beispiel davor, trifft er irgendwann eine falsche Entscheidung und wickelt sich selbst ein.

Wie bereits bei den Herausforderungen erwähnt, ist es sehr schwer, den Agenten in dieser zufälligen Umgebung gut zu trainieren. Wir haben unterschiedliche Arten von Hindernissen und diese spawnen, genau wie die Äpfel, an zufälligen Positionen. Somit kann es manchmal dazu kommen, dass die Strategie, die der Agent erlernt hat, um ausweichen zu können, ihn sehr früh das Leben kostet.

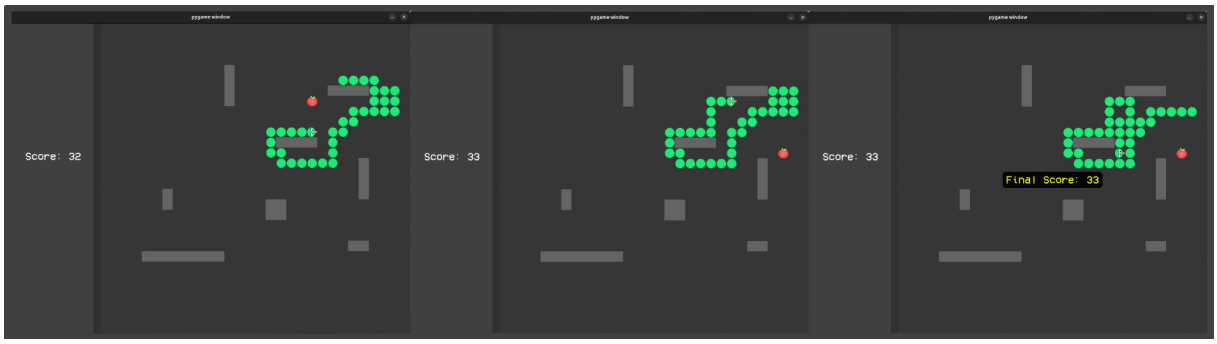

Abbildung 6.5: Kurzsichtigkeit des Agenten

Wir sehen bei diesem Beispiel, dass der Agent bereits eine Länge von 32 hat und der Apfel ist quasi direkt über ihm. Anstatt jedoch den Apfel einzusammeln und weiter nach oben oder nach links zu gehen, begibt sich die Schlange in eine Sackgasse. Wir vermuten, dass dies aufgrund der im Training erlernten Strategien passiert. Der Agent versucht immer wieder, seinen Körper besser in Position zu setzen, damit er sich am Ende nicht im Weg steht. Hier sehen wir jedoch, dass das schief geht. Er erwischt den Apfel, jedoch kann er den nächsten nicht mehr erreichen und er bringt sich selbst um.

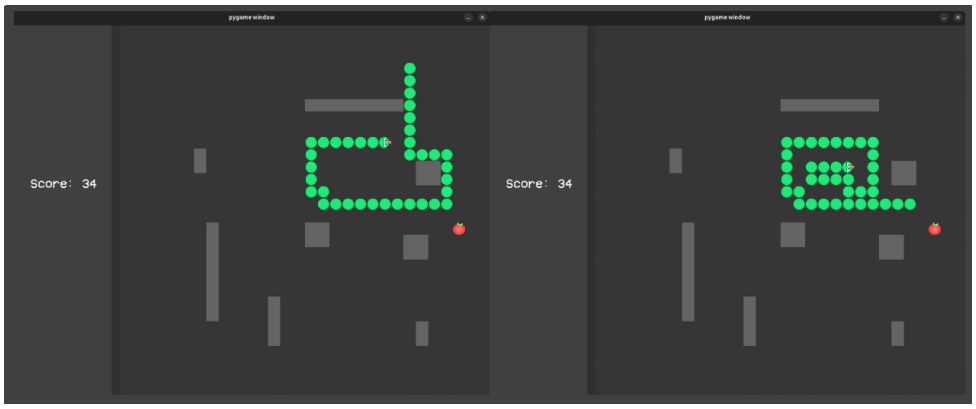

Abbildung 6.6: Kurzsichtigkeit des Agenten

Ein weiteres Beispiel zeigt ein ähnliches Problem: Der Agent holt einen Apfel, sieht den nächsten und läuft direkt auf ihn zu, obwohl es sicherer gewesen wäre, zunächst einen kleinen Umweg zu machen. Durch diese Entscheidung schneidet er sich selbst den Weg ab. Hätte er ein paar Züge vorher ein paar leere Bewegungen gemacht, wäre der Weg vielleicht noch offen gewesen. Dieses Verhalten zeigt, dass DQN zwar kurzfristig effektiv ist, aber langfristig nicht gut planen kann.

Die vier gezeigten Ausführungen verdeutlichen die Stärken und Schwächen unseres trainierten Agenten. Im Durchschnitt gelingt es ihm, zuverlässig die Äpfel einzusammeln und den Hindernissen aus dem Weg zu gehen. Dies ist ein Zeichen dafür, dass er die Grundmechaniken des Spiels gelernt hat.

6.3 Evaluation

Zum Auswerten eines trainierten Modells wird von Stable-Baselines 3 mit der `evaluate policy` eine Funktion bereitgestellt. Diese Funktion haben wir mehrfach getestet, jedoch ist sie in ihren Ausgabemöglichkeiten relativ begrenzt. Deshalb haben wir uns dazu entschlossen, ein eigenes Programm zu schreiben, das die entsprechenden Auswertungen durchführt und optimal an unsere Vorstellungen angepasst ist. Dieses Programm ist in der `evaluate.py` zu finden und umfasst die Ausgabe von Q-Werten, Aktion, Reward, Score und Schritten für jede Episode. Zudem werden am Ende des Testlaufs der Median Score, durchschnittliche Score und Schritte, die Standardabweichung, der höchste und niedrigste Score, die Anzahl valider Episoden und die Anzahl von Schleifen ausgegeben, sowie optional die Architektur des Modells. Abgesehen von den reinen Werten werden auch Graphen zur Score-Verteilung und den Q-Werten erzeugt. Im Folgenden werden die Ergebnisse der Evaluation analysiert und kritisch bewertet. Wie bereits im Abschnitt Herausforderungen besprochen, besteht weiterhin das Problem der Endlosschleifen. Diese haben wir bewusst aus der Bewertung des Modells herausgenommen, da sie nicht repräsentativ sind. Am Ende jedes Evaluierungsdurchlaufs wird die Anzahl der validen Episoden und die Anzahl der aufgrund von Endlosschleifen abgebrochenen Episoden angezeigt. Zur Berechnung von Mittelwert, Median usw. werden nur die validen Episoden betrachtet.

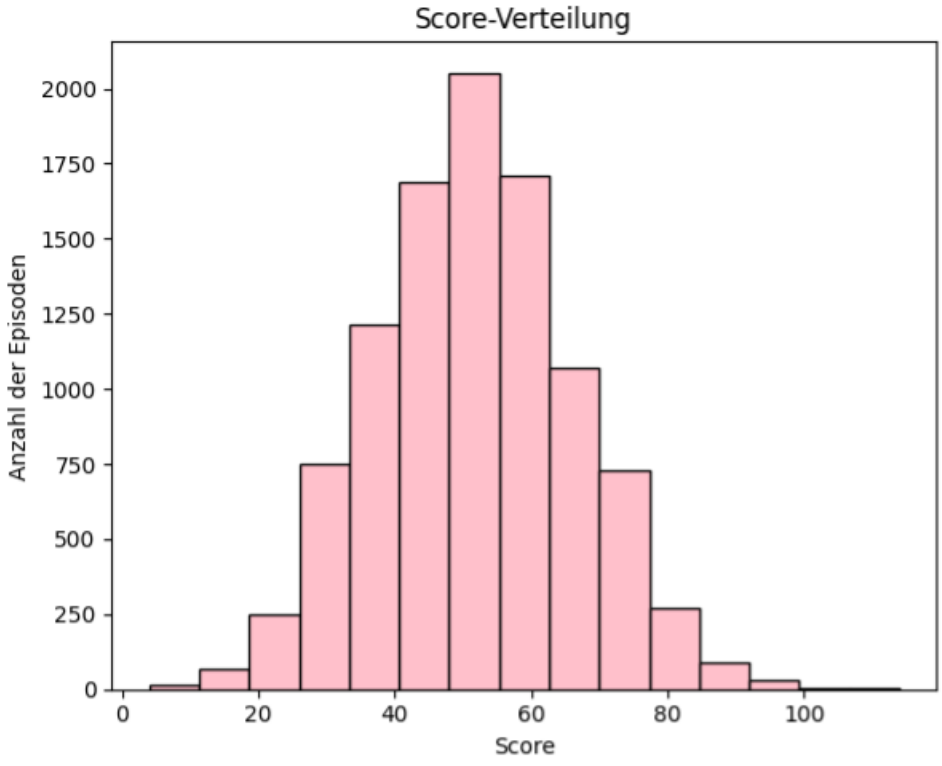

Abbildung 6.7: Score-Verteilung bei 10.000 Testepisoden

Dieser Graph zeigt, wie häufig bestimmte Score-Bereiche in den gespielten Episoden erreicht wurden. Die Verteilung ist deutlich glockenförmig, mit einem Schwerpunkt im Bereich von 45 bis 55 Punkten, was auf eine stabile und zuverlässige Leistung des Modells hinweist. Ausreißer in sehr niedrigen oder sehr hohen Score-Bereichen sind zwar vorhanden, treten jedoch selten auf

und unterstreichen insgesamt die Konstanz des Modells.

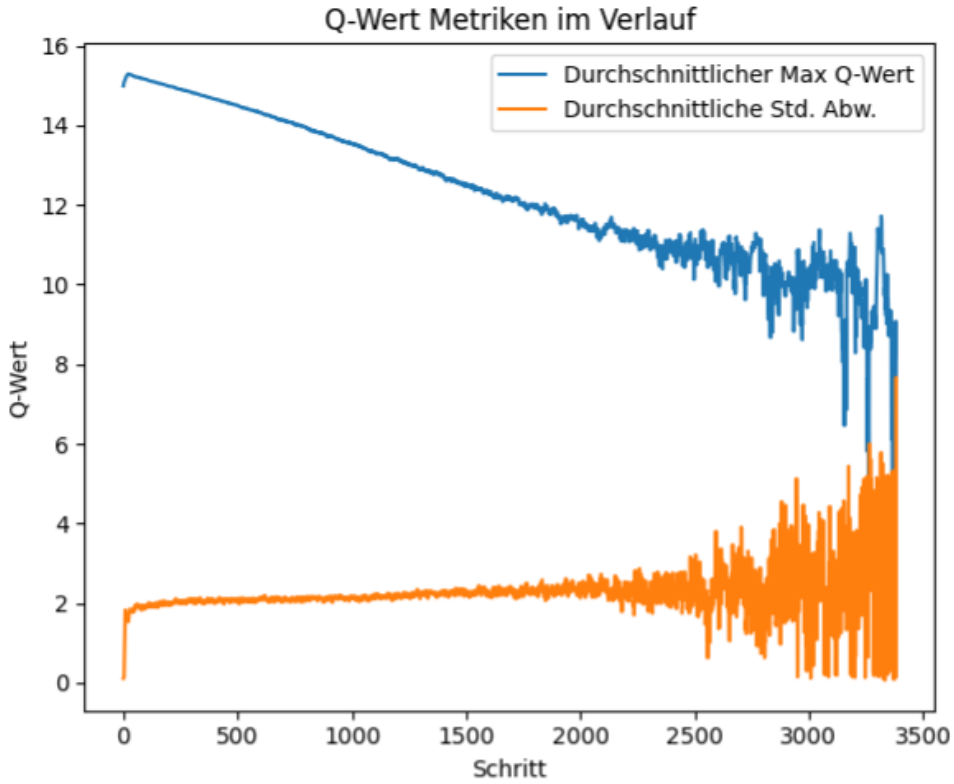

Abbildung 6.8: Q-Werte im Verlauf der 10.000 Testepisoden

Dieser Graph zeigt, wie selbstsicher der Agent in den 10.000 Test Episoden agiert, abhängig von der jeweiligen Spielsituation. Man erkennt dabei zwei klare Verhaltensmuster. Zu Beginn der Episoden, also im Bereich 0 bis ungefähr 2.500 Schritten, zeigt der Agent ein sehr zuversichtliches Verhalten. In diesen Anfangsphasen ist die Schlange noch kurz und hat viel Platz, wodurch die Situationen relativ einfach und ungefährlich sind. Der hohe durchschnittliche maximale Q-Wert zeigt, dass der Agent eine hohe Belohnung erwartet, während die niedrige Standardabweichung zeigt, dass viele Züge als ähnlich gut und sicher eingestuft werden. Das Bild ändert sich jedoch deutlich in den späteren Phasen des Spiels. Ab ungefähr 2.500 Schritten gibt es wachsende Ausschläge bei beiden Kurven. Diese Anstiege, vor allem bei der Standardabweichung, sind ein positives Zeichen. Sie treten in den Momenten auf, in denen die Schlange bereits sehr lange ist oder wenn sie sich in eine schwierige Position begeben muss, um einen Apfel zu erreichen. Hier erkennt der Agent, dass eine einzelne Aktion über Überleben oder Sterben entscheidet. Der große Unterschied zwischen dem Wert der richtigen und der falschen Aktionen führt zu diesen Ausschlägen und zeigt, dass der Agent die Gefahr erkennt.

```
--- Auswertung ---
Durchschnittlicher Score: 51.77
Median Score: 51.00
Standardabweichung: 14.63
Höchster Score: 110
Niedrigster Score: 7
Durchschnittliche Schritte: 1262.38
Anzahl valider Episoden: 9934
Anzahl von Schleifen: 66
```

Abbildung 6.9: Ausgabe der evaluate.py

Es ist zu beobachten, dass der durchschnittliche Score mit 51.77 und der Median Score mit 51 nah beieinander liegen. Das spricht zwar für eine symmetrische Verteilung der Scores, jedoch zeigt der Hohe Unterschied zwischen dem höchsten Score (110) und dem niedrigsten Score (7), sowie die hohe Standardabweichung in höhe von 14.63, dass es in beide Richtungen zu Extremwerten kommt. Das bedeutet, dass das Modell zwar meistens stabile Ergebnisse liefert, es jedoch gelegentlich zu stärkeren Abweichungen kommen kann. In diesem Fall (10.000 Testepisoden) zeigt sich, dass lediglich 0.66% der Episoden in Schleifen enden. Auch wenn dieser Anteil gering ist, stellen gerade diese Episoden ein erhebliches Problem dar.

7 Fazit und Ausblick

Im Rahmen dieses Projektes ist es uns gelungen, einen auf DQN basierenden Agenten für das Spiel Snake in Python zu entwickeln. Dabei haben wir die Gymnasium Bibliothek zur Umsetzung der Reinforcement-Learning Umgebung genutzt. Das Training des Agenten wurde dabei mit Stable-Baselines 3 realisiert. Zentrale Konzepte der Umsetzung wurden dabei erläutert, wie etwa der Aufbau des Observation Space, das Belohnungssystem und das Training. Der Trainingsverlauf hat dabei gezeigt, wie komplex und herausfordernd es ist, ein robustes Modell zu erstellen. Insbesondere die Vermeidung von Endlosschleifen und Kollisionen mit dem eigenen Körper stellen eine erhebliche Herausforderung dar. Erstere ließen sich dabei trotz zahlreicher Anpassungen von Belohnungen, Observation Space und Hyperparametern nicht komplett ausschließen. Dennoch gelang es uns, die Wahrscheinlichkeit des Auftretens dieses Problems auf ein Minimum zu begrenzen und die Fälle, in denen dieses Problem auftritt, automatisch zu erkennen und abzubrechen. Dadurch konnte die Performance des Agenten trotzdem auf ein gutes Niveau gebracht werden.

Die Evaluierung des finalen Modells ergab, dass der Agent nicht nur in der Lage ist, einfachen Hindernissen auszuweichen und Äpfel einzusammeln, sondern auch langfristige Strategien zu entwickeln, um sich selbst aus kritischen Situationen zu befreien. Gezeigt hat sich das insbesondere in späten Spielsituationen, in denen der Agent trotz einer beachtlichen Länge von über 50 Blöcken noch in der Lage ist, seinen Körper sinnvoll zu verwalten, um in Ausnahmefällen sogar eine Punktzahl jenseits der 100 Punkte zu erreichen. Dennoch wurde auch deutlich, dass der Agent noch Potential für Verbesserungen hat. Abgesehen von den weiterhin bestehenden Schleifen Problemen zeigte sich, dass der Agent teilweise schon sehr früh mit sich selbst kollidiert, was auch durch die hohe Standardabweichung widergespiegelt wird. Anpassungen der Hyperparameter, des Belohnungssystems und des Observation Space könnten dabei jedoch noch weitere Verbesserungen mit sich bringen und die Probleme eventuell sogar komplett lösen.

Abschließend lässt sich festhalten, dass auch vermeintlich einfache Spiele wie Snake eine nicht zu verachtende Herausforderung für Reinforcement-Learning-Algorithmen darstellen können. Die Umsetzung einer solchen Umgebung und das Trainieren eines Agenten erfordern ein fundiertes Verständnis der zugrunde liegenden Konzepte und auch eine gewisse Frustrationstoleranz, da ein großes Maß an Trial-and-Error notwendig ist, um ein gutes Modell zu erstellen. Die dabei gesammelten Erfahrungen bilden eine gute Grundlage für die Umsetzung weiterer Projekte im Bereich Reinforcement-Learning, aber auch in der Data Sciene im Allgemeinen.

Interessante Ansatzpunkte zur Weiterentwicklung sind die schon erwähnten Hyperparameter, das Belohnungssystem und der Observation Space. Unsere Tests haben gezeigt, dass Anpassungen dieser Komponenten einen erheblichen Einfluss auf die Häufigkeit von Dauerschleifen und auf den durchschnittlichen Score haben. Mit mehr Zeit für das Projekt, würden wir hier zuerst

ansetzen und so hoffentlich das Schleifenproblem final lösen können. Auch die hohe Standardabweichung könnte durch Anpassungen an diesen Werten vermutlich deutlich verbessert werden. Dabei könnte konkret der Observation Space überarbeitet werden, da das Sichtfeld, das dem Agenten übergeben wird, noch Optimierungsbedarf hat.

Da wir noch keine Erfahrungen mit Pygame hatten, haben wir zum Erstellen des Spiels an manchen Stellen ChatGPT bzw. Gemini zur Unterstützung genutzt. Außerdem haben wir, wie schon erwähnt, zum Verständnis der Hyperparameter beim Trainieren des Modells ebenfalls ChatGPT bzw. Gemini genutzt. Der Stil und die Rechtschreibung in der Dokumentation wurden an manchen Stellen ebenfalls von diesen Tools unterstützt.

Literaturverzeichnis

[1] Gymnasium, "Gymnasium: A standard interface for reinforcement learning environments." https://gymnasium.farama.org, 2024.

[2] stable baselines3, "Stable-baselines3: Reliable reinforcement learning implementations." https://stable-baselines3.readthedocs.io/en/master/, 2020.

[3] S. Amin, "Deep q-learning (dqn)." https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae, 2024.

[4] IBM, "What is explainable ai (xai)?." https://www.ibm.com/think/topics/explainable-ai, 2025.

[5] DataCamp, "A beginner's guide to the rectified linear unit (relu)." https://www.datacamp.com/blog/rectified-linear-unit-relu, 2025.