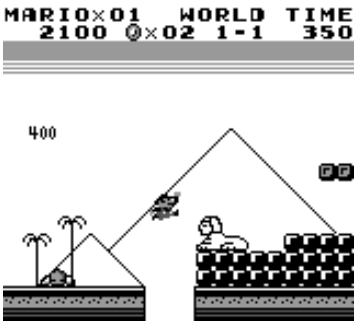
Train RL Mario AGENT

q_play

Playing Super Mario Land with Reinforcement Learning 🤖

Advances in AI - WS 2023/24 HSD

Vincent Bohm
Matrikelnummer: 897512



GitHub: [l4nz8/q_play]

Inhaltsverzeichnis:

1. Einleitung

1.1 Projektbeschreibung:

Bei meinem Projekt "Train RL Mario AGENT" (q_play) handelt es sich um ein Reinforcement Learning (RL)-Modell zur Automatisierung und Optimierung der Spielstrategie im Videospiel "Super Mario Land" auf der GameBoy-Emulator-Plattform PyBoy. Ziel dieses Projekts war die Entwicklung und Implementierung eines intelligenten Agenten, der durch das Spiel navigiert, Hindernisse und Gegner überwinden kann und das Level schnellstmöglich abschließt. Der Agent sollte kontinuierlich aus seinen Aktionen und deren Ergebnissen lernen. Um das zu schaffen, verwendet der Agent eine Double Deep Q-Network (DDQN)-Architektur, eine Weiterentwicklung des Deep Q-Network (DQN), um eine stabile und effiziente Lernleistung zu erzielen.

1.2 Motivation:

Meine Motivation hinter der Projektwahl liegt in dem großen Interesse an der Funktionsweise, dem Aufbau und Ablauf von selbstlernenden und optimierenden Netzwerken. Als mir schiksalhafterweise auf Youtube zusätzlich noch ein Video von einem auf dem PyBoy erstellten RL-Proximal-Policy-Optimization (PPO)-Modell vorgeschlagen wurde, welches Pokemon Red auf der GameBoy-Plattform gespielt hat, habe ich meine Projektwahl getroffen, selbst ein RL-Modell auf dem Gameboy zu erstellen. Nach der Wahl des KI-Modells und der Plattform brauchte ich nur noch eine Lernumgebung für den Agenten. Da "Super Mario Land" eine komplexe sowie variantenreiche Levelgestaltung und dabei noch einen linearen Spielaufbau besitzt, habe ich mich schließlich für dieses Spiel entschieden. Es bietet eine ideale Plattform zur Erforschung und Demonstration der Fähigkeiten von RL-Agenten, wodurch es schon oft als Beispielumgebung für RL Modelle genutzt wurde.

2. Grundlagen und State-of-the-Art

2.1 Reinforcement Learning (RL) und Deep Q-Networks (DQN):

“

Reinforcement Learning (RL) ist ein Bereich des maschinellen Lernens, bei dem ein Agent darauf abzielt, selbstständig eine optimale Policy (Verhaltensstrategie) zu entwickeln. Agentensysteme versuchen durch trial and error in einer Umgebung, ihre Belohnung für das richtige Handeln zu maximieren. Die Grundidee ist es, dass ein Agent durch Interaktion mit der Umgebung und der Beobachtung der Ergebnisse, festgelegt durch einen Belohnungswert, seine Aktionen lernt. Dadurch ist es ihm möglich, situationsbedingte Strategien zu entwickeln, die auf einen möglichst hohen Belohnungswert abzielen. [1]

“

“

Deep Q-Networks (DQN) vereinen das klassische Q-Learning mit der Leistungsfähigkeit tiefer neuronaler Netzwerke, wodurch es möglich wird, komplexe Probleme in umfangreichen Zustandsräumen zu lösen. Durch das Erlernen einer Näherung der Q-Funktion, die den erwarteten Wert einer Aktion in einem bestimmten Zustand darstellt, können DQNs in anspruchsvollen Umgebungen agieren. Diese Methode wurde erfolgreich in der Domäne der Atari-Videospiele angewendet, wo DQNs beeindruckende Fähigkeiten zeigten, indem sie in mehreren Spielen menschliche Spieler nicht nur gleichzogen, sondern in einigen Fällen sogar überflügelten. Diese Durchbrüche demonstrieren das enorme Potenzial von DQNs in der Praxis. [2]

“ *ChatGPT

2.2 State-of-the-Art in RL und Anwendung auf Videospiele

“

Seit der Einführung von DQNs hat sich das Feld des RL schnell weiterentwickelt. Neue Architekturen wie Double DQN, Dueling DQN, und A3C (Asynchronous Advantage Actor-Critic) haben die Leistung und Stabilität von RL-Agenten weiter verbessert. Diese Techniken adressieren verschiedene Herausforderungen von DQNs, wie die Überbewertung von Q-Werten oder die Ineffizienz in der Nutzung von Umgebungsinformationen.
“ *ChatGPT

Die Anwendung von RL auf Videospiele dient nicht nur als Benchmark für die Leistungsfähigkeit von solchen Algorithmen, sondern auch als Forschungsumgebung für die Weiterentwicklung des RL. Plattformer-Spiele wie "Super Mario" sind besonders interessant, da sie eine Kombination aus räumlicher Navigation, Timing und Strategie erfordern. Die Forschung in diesem Bereich hat gezeigt, dass RL-Agenten in der Lage sind, komplexe Strategien zu lernen und hohe Leistungen in diesen Spielen zu erzielen.

2.3 Double Deep Q-Network-Architektur:

“

Double Deep Q-Networks (DDQN) bauen auf den Grundlagen von Deep Q-Networks (DQN) auf und zielen darauf ab, ein zentrales Problem der DQNs – die systematische Überschätzung von Q-Werten – zu lösen. Durch die Einführung einer methodischen Trennung zwischen der Auswahl und der Bewertung von Aktionen minimieren DDQNs das Risiko einer Überbewertung. Dies wird erreicht, indem zwei separate Netzwerke eingesetzt werden: ein Online-Netzwerk, das für die Auswahl von Aktionen verantwortlich ist, und ein Ziel-Netzwerk, das diese Aktionen bewertet. Diese architektonische Innovation trägt dazu bei, den Lernprozess zu stabilisieren, und beschleunigt die Konvergenz gegenüber traditionellen DQN-Ansätzen. [3]
“ *ChatGPT

Obwohl es mittlerweile modernere RL-Methoden und situationsbedingt bessere Modelle wie z. B. Proximal Policy Optimization Algorithms (PPO) [4] gibt, sind DDQNs immer noch eine sehr effektive und häufig verwendete Architektur für Agentennetzwerke. Sie eignen sich besonders gut für Aufgaben mit diskreten Aktionen wie die Steuerung durch das Super-Mario-Land-Level, da nur eine klare Bewegung durch das Spiel gefordert wird.

3. Technologie und Methodik

3.1 PyBoy-Emulator:

Um ein RL-Modell auf dem Spiel Super Mario Land trainieren zu können, benötigt man eine Schnittstelle zu dem Spiel, wie z. B. in Form eines Emulators, der diese Umgebung erstellen kann. Der PyBoy-Emulator ist eine Open-Source-Plattform, die es ermöglicht, GameBoy-Spiele auszuführen. Zusätzlich bietet die Plattform eine Python-API, wodurch das Auslesen der Daten erheblich erleichtert wird, was für die Entwicklung eines interaktiven Lernumfelds entscheidend ist. Zu der Möglichkeit, Spielzustände zu manipulieren und visuelle Ausgaben zu extrahieren, was für das Training des RL-Agenten unerlässlich ist, kann der Agent direkte Rückmeldungen zu seinen Aktionen erhalten und seine Strategie zur Maximierung der Belohnungen im Spiel anpassen. Zusätzliche bietet der Emulator noch weitere Funktionen wie z. B. Screenshot, Videorecording, Pausieren und Game-State-Speicherung.

3.2. Argparse:

Der Argumentparser dient dazu, Befehle in der Konsole zu verarbeiten und im Skript auszuführen. Ich habe verschiedene Funktionen implementiert, um eine effiziente und interaktive Plattform zu erstellen. Alle

Argumente sind im Skript mithilfe von `*--help*` auszulesen. Sie bieten die Möglichkeit, Worldlevel zu wählen, Spiel- oder Trainingsmodus zu aktivieren, das Laden von Spielzuständen, Aktivieren der Debug-Funktion, Setzen von Hyperparametern etc. Diese Argumente ermöglichen eine flexible Konfiguration des AI-Agenten und des Trainingsprozesses, um unterschiedliche Szenarien und Anforderungen zu berücksichtigen.

3.3 Preprocessing-Techniken:

Für den Trainingsprozess habe ich zunächst das Ausgabebild der Spielumgebung durch verschiedene sogenannte „Wrapping-Prozesse“ (Preprocessing-Techniken) für das Netzwerk vorbereitet:

3.3.1 Frame Skipping:

Durch das Überspringen von Frames kann die Trainingszeit erheblich reduziert werden. Dies verringert die Komplexität des Zustandsraums, mit dem der Agent interagieren muss, was zu einer schnelleren Entscheidungsfindung führt.

3.3.2 Graustufenkonvertierung:

Da drei Farbkanäle für den Lernprozess nicht wichtige Zusatzinformationen besitzen, werden diese auf einen Graustufenkanal reduziert. Obwohl das Spiel bereits in Graustufen gerendert wird, gibt der Emulator dennoch 3 Farbkanäle mit denselben Schwarzweißwerten zurück. Um die Dimensionalität zu reduzieren, musste das Bild somit auf einen Farbkanal reduziert werden.

3.3.3 Resize Observation:

Zur schnelleren Verarbeitung der Informationen wird das Ausgabebild in ein kleineres Format runterskaliert. Eine höhere Auflösung ist nicht unbedingt nötig, um dieselben Bildinformationen weiterzugeben.

3.3.4 Frame Stacking:

Um dem Agenten ein Verständnis für zeitliche Abfolgen zu vermitteln, werden mehrere aufeinanderfolgende Frames zu einem einzigen Eingabestapel zusammengefasst. Dadurch ist es dem Agenten möglich, Bewegungen im Spiel zu verstehen und somit bessere Entscheidungen treffen zu können.

Diese Preprocessing-Techniken sind wichtig, da sie dem Netzwerk die Möglichkeit geben, die Lernumgebung besser zu verstehen und dabei nicht zu überfordern. Damit tragen sie dazu bei, die Effektivität des Lernprozesses zu steigern.

3.4 Action space:

Der Action space beschreibt die Menge aller möglichen Aktionen, die ein Agent in seiner Umgebung ausführen kann, um eine bestimmte Aufgabe zu erfüllen. Je größer der Action space, desto komplexer und rechenintensiver wird das Modell, da mehr mögliche Handlungen bewertet werden müssen. Um den Rahmen des Projektes nicht zu sprengen, habe ich somit den Action space auf das Nötigste reduziert, um ein Level erfolgreich abzuschließen. In diesem Projekt umfasst der Aktionsraum nur Vorwärtsbewegung, Rückwärtsbewegung, Springen und Kombinationen aus diesen Aktionen.

Possible actions: [['PRESS_ARROW_RIGHT'], ['PRESS_BUTTON_A'], ['PRESS_ARROW_LEFT'], ['PRESS_ARROW_RIGHT', 'PRESS_BUTTON_A'], ['PRESS_BUTTON_A', 'PRESS_ARROW_LEFT']]

Diese Reduzierung hilft dabei, die Komplexität und benötigte Rechenleistung zu minimieren und es dem Agenten immer noch zu ermöglichen, das Level erfolgreich abzuschließen.

4. Implementierung des RL-Agenten

4.1 Netzwerkarchitektur:

Der DDQN-Agent verwendet ein Convolutional Neural Network (CNN), welches für die Verarbeitung der Pixel-Eingaben des Spiels "Super Mario Land" nötig ist. Die Eingabe für das Netzwerk ist ein Stapel von vier vorverarbeiteten Frames, die jeweils auf eine Auflösung von 84x84 Pixel in Graustufen skaliert wurden. Die Architektur besteht aus drei Convolutional-Layer, gefolgt von zwei Fully Connected-Layer.

```
67        # Define the CNN architecture
68        return nn.Sequential(
69            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4, dtype=torch.float32),
70            nn.BatchNorm2d(32), # Mitigate the problem of internal convariate shift
71            nn.LeakyReLU(), # Avoid the issue of "dead neurons"
72            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, dtype=torch.float32),
73            nn.BatchNorm2d(64),
74            nn.LeakyReLU(),
75            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, dtype=torch.float32),
76            nn.BatchNorm2d(64),
77            nn.LeakyReLU(),
78            nn.Flatten(), # Flatten the output for the fully connected layers
79            nn.Linear(3136, 512, dtype=torch.float32), # 3136 = number of features from the last conv layer
80            nn.LeakyReLU(),
81            nn.Linear(512, output_dim, dtype=torch.float32) # Output layer
82        )
```

- **Convolution-Layer (Conv2d):** Verarbeiten Eingabebilder, indem es räumliche Features extrahiert. Die Größe des Filters mit Stride bestimmen, wie die Eingabe reduziert wird und welche Merkmale hervorgehoben werden.
- **Batch-Normalisierung (BatchNorm2d):** Normalisiert Features, wodurch das Training stabilisiert und beschleunigt wird. Es minimiert das Problem des internen covarianten Shifts.
- **LeakyReLU-Aktivierung:** Fügt Nichtlinearität hinzu, um das Problem der "toten Neuronen" zu vermeiden.
- **Flatten:** Wandelt die multidimensionalen Feature-Maps in einen eindimensionalen Vektor um.
- **FullyConnected-Layer (Linear):** Bildet hochdimensionale Features auf einen niedrigerdimensionalen Ausgaberaum ab, welcher die möglichen Aktionen darstellt.

Nach vollständiger Verarbeitung gibt das Netzwerk eine Anzahl von 512 möglichen Situation bedingten Aktionen als Ausgabe zurück.

4.2 Hyperparameter:

Die wichtigsten Hyperparameter für den DDQN-Agenten umfassen die Lernrate, Gamma-Wert, Exploration Rate, Zerfall-Batch-Größe, Replay-Memory-Größe und das Synchronisieren der beiden Netzwerke.

exploration_rate = 1
exploration_rate_decay = 0.99999975
exploration_rate_min = 0.1
batch_size = 32
storage=LazyMemmapStorage(100000)
learning_rate=0.00025
gamma = 0.9
self.sync_every = 10000

- Der Agent verwendet anfänglich eine Exploration Rate von 1, um die Umgebung durch heufig zufällige Aktionen zu erkunden, bis er einen Minimalwert von 0.1 erreicht.

- Die Exploration wird über die Zeit um den Faktor von 0.99999975 dekrementiert, um häufiger Aktionen mit dem höchsten Q-Value auszuführen.
- Die Replay-Memory gibt die Größe der Speicherung der vergangenen Erfahrungen in einer Episode an, mit einer Batch-Größe von 32 für das Training.
- Die Lernrate für den Optimizer beträgt 0.00025 mit einem Gamma-Wert von 0.9 für zukünftige Belohnungen, welche vom "StepLR"-Scheduler genutzt wird.

Es ist möglich, den *"Cyclic"*-Scheduler anstelle des *"StepLR"* zu wählen, welcher eine Schrittweite von 2000 zwischen den Minimal- (0.0001) und Maximal-Wert (0.001) besitzt.

4.3 Interaktion mit der Umgebung:

Die Interaktion des Agenten mit der Spielumgebung erfolgt durch eine speziell für Super Mario Land abgewandelte Klasse, die auf der von PyBoy implementierten OpenAI Gym-Schnittstelle basiert. Diese ermöglicht es dem Agenten, Aktionen auszuführen und Beobachtungen sowie Belohnungen von der Umgebung zurückzubekommen.

Die `CustomPyBoyGym`-Klasse verarbeitet Details der Spielumgebung, wie das Laden des Spiels, das Setzen des Levels/der Welt, und das Extrahieren relevanter Spielinformationen für den Agenten.

4.4 Komponenten des RL-Agenten:

4.4.1 Action Selection:

Die Aktionsauswahl erfolgt durch eine Agenten-Epsilon-Greedy-Politik. Das bedeutet, dass der Agent dazu tendiert, Aktionen zu wählen, die nach dem aktuellen Modell den höchsten Q-Wert besitzen. Es wird aber mit einer Wahrscheinlichkeit von Epsilon (Exploration-Rate) eine zufällige Aktion gewählt, um die Erkundung der Umgebung zu fördern.

4.4.2 Reward System:

Das Belohnungssystem, basierend auf vier Komponenten: Schnelleres Fortbewegen durch das Level, inwieweit der Agent im Level vorangekommen ist, eine negative Belohnung für die Tode und eine positive Belohnung für das Absolvieren des Levels.

Die gesamte Belohnung ist die Summe dieser vier Komponenten: Zeitdifferenz, Fortbewegungsdifferenz, Lebenverlustbestrafung und Level-Fortschrittsbelohnung. Dieses System zielt darauf ab, schnelles Vorankommen, Überleben und das Erreichen neuer Level oder Welten zu fördern.

```
66    # Reward components
67    clock = current_mario.time_left - prevGameState.time_left
68    movement = current_mario.real_x_pos - prevGameState.real_x_pos
69    death = -15*(current_mario.lives_left - prevGameState.lives_left)
70    levelReward = 15*max((current_mario.world[0] - prevGameState.world[0]), (current_mario.world[1] - prevGameState.world[1]))
71
72    reward = clock + death + movement + levelReward
73    return reward
```

4.4.3 Memory Buffer:

Der Replay-Memory-Buffer speichert die Erfahrungen des Agenten in Form von Zustand, Aktion, Belohnung, nächstem Zustand und dem Indikator des Episodenendes. Diese Erfahrungen werden dann verwendet, um die Stabilität des Lernprozesses zu verbessern und die Korrelationen zwischen aufeinanderfolgenden Lernschritten zu verringern.

5. Experimente und Bewertung

5.1 Experimentelles Setup:

Um die Leistungsfähigkeit des Mario-RL-Agentennetzwerks zu evaluieren, habe ich zwei verschiedene Netzwerke verwendet und deren Lern-Performance auf einem Level verglichen. Es wurden zwei Modellgrößen mit zwei verschiedenen Lern-Optimizer-Schedulern verwendet:

1. *"StepLR"* Optimizer-Scheduler mit einer Netzwerkgröße von 3 Convolution-Layern und ohne Pedding.
2. *"Cyclic"* Optimizer-Scheduler mit einer Netzwerkgröße von 4 Convolution-Layern und auf den 2 bis 4 Schichten jeweils ein Pedding von 1.

Dadurch wollte ich schauen, welches Netzwerk besser performt und ob die Netzwerkgröße und das Pedding zu signifikanten Unterschieden führen.

Beide Modelle wurden über einen Zeitraum von ungefähr 48 Stunden mit insgesamt 6 Millionen getätigten Aktionen laufen lassen. Die Modelle selbst wurden alle auf dem ersten Level der ersten Welt des Spiels (W1 / L1) getestet und deren Lernfortschritte am Ende miteinander abgeglichen. Beide Modelle wurden mit der Implementierung von Cuda auf einer NVIDIA GeForce RTX 2080 Ti mit 19 GB GPU-Speicher trainiert.

5.2 Trainingsresultate und Diskussion:

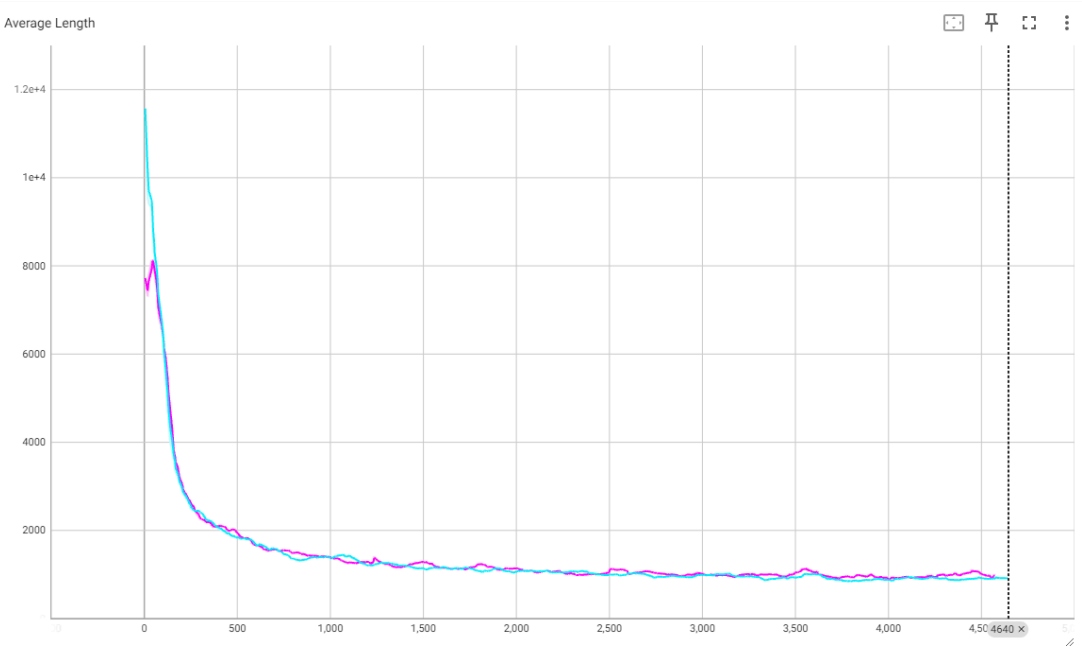
Um die Unterschiede des Lernverhaltens der zwei Modelle zu beobachten, wurden 4 Metriken (Average Length, Average Loss, Average Q-Value, Average Reward) während des Lernprozesses geloggt:

“

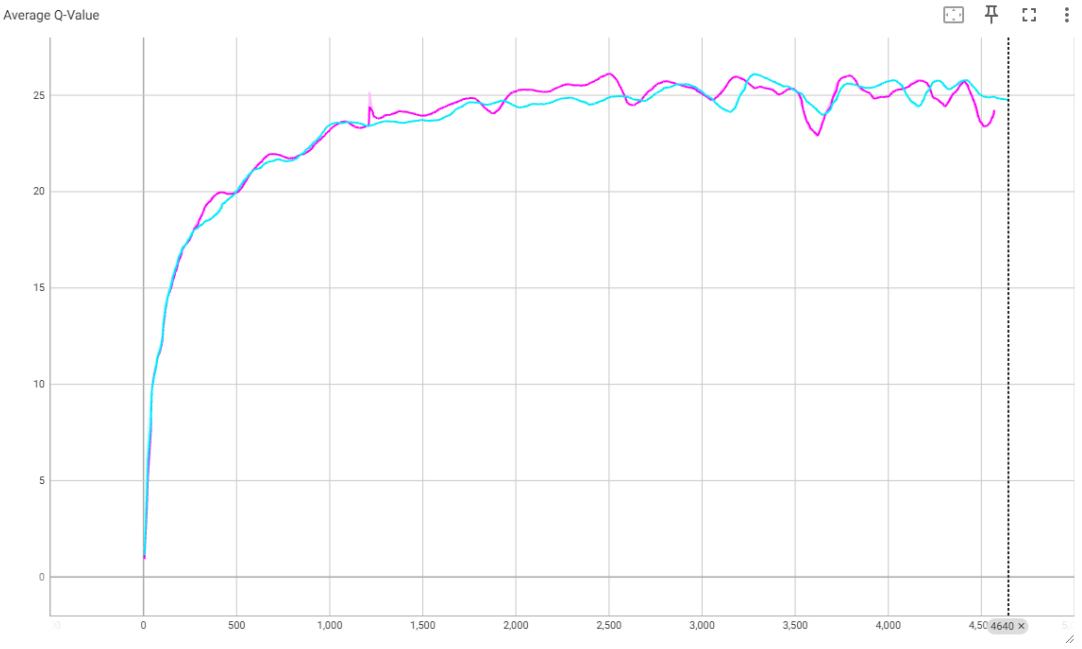
- **Average Length:** Dies bezieht sich auf die durchschnittliche Dauer der Episoden, gemessen in Zeitschritten oder Aktionen, bevor eine Episode endet (z.B. durch das Erreichen des Ziels oder das Scheitern der Aufgabe). Längere durchschnittliche Episodenlängen deuten darauf hin, dass der Agent sich länger in einer Episode befindet, was jedoch nur situationsbedingt gut ist.
- **Average Loss:** Der durchschnittliche Verlust (Loss) gibt an, wie gut das Modell die erwarteten Belohnungen vorhersagt. Ein niedrigerer Verlustwert deutet darauf hin, dass die Vorhersagen des Agenten bezüglich der zukünftigen Belohnungen genauer werden, was auf ein effektiveres Lernen hinweist.
- **Average Q-Value:** Der durchschnittliche Q-Wert spiegelt die erwartete Belohnung wider, die der Agent für seine Aktionen in einem bestimmten Zustand schätzt. Höhere durchschnittliche Q-Werte bedeuten, dass der Agent höhere Belohnungen für seine Aktionen erwartet, was ein Indikator für das Lernen von effektiveren Strategien sein kann.
- **Average Reward:** Der durchschnittliche Belohnungswert zeigt, wie viel Belohnung der Agent im Durchschnitt pro Episode erhält. Eine Zunahme der durchschnittlichen Belohnung über die Zeit ist ein direkter Indikator dafür, dass der Agent lernt, seine Aktionen zu optimieren, um höhere Belohnungen zu erzielen.

“ *ChatGPT

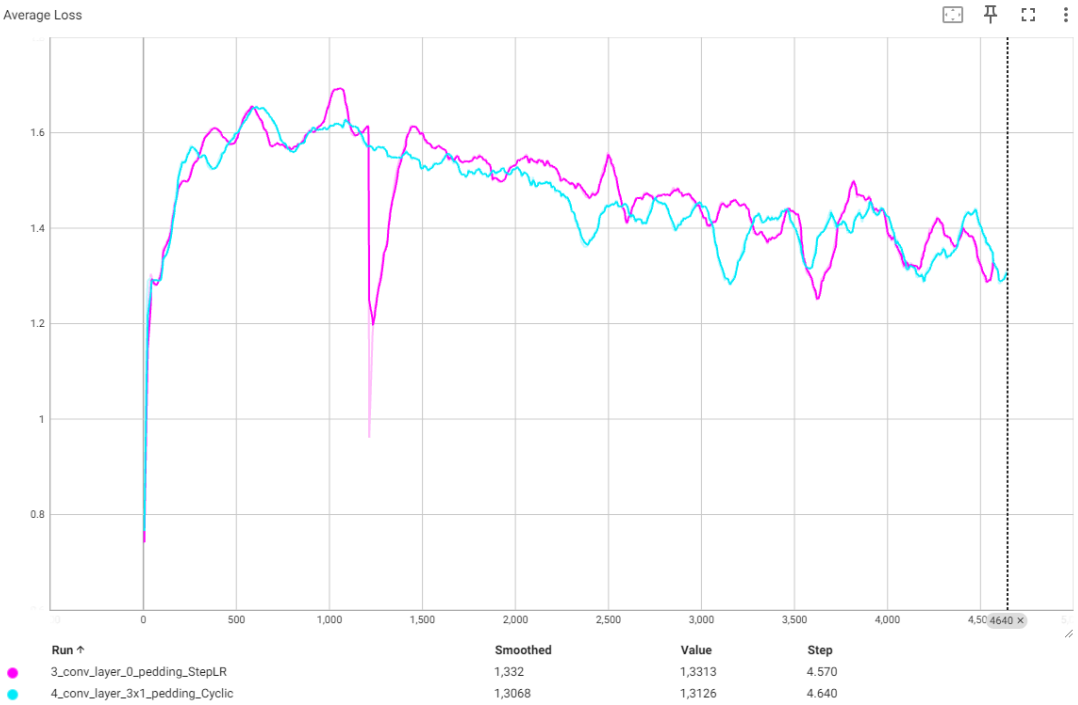
5.3 Bewertung der Leistung:

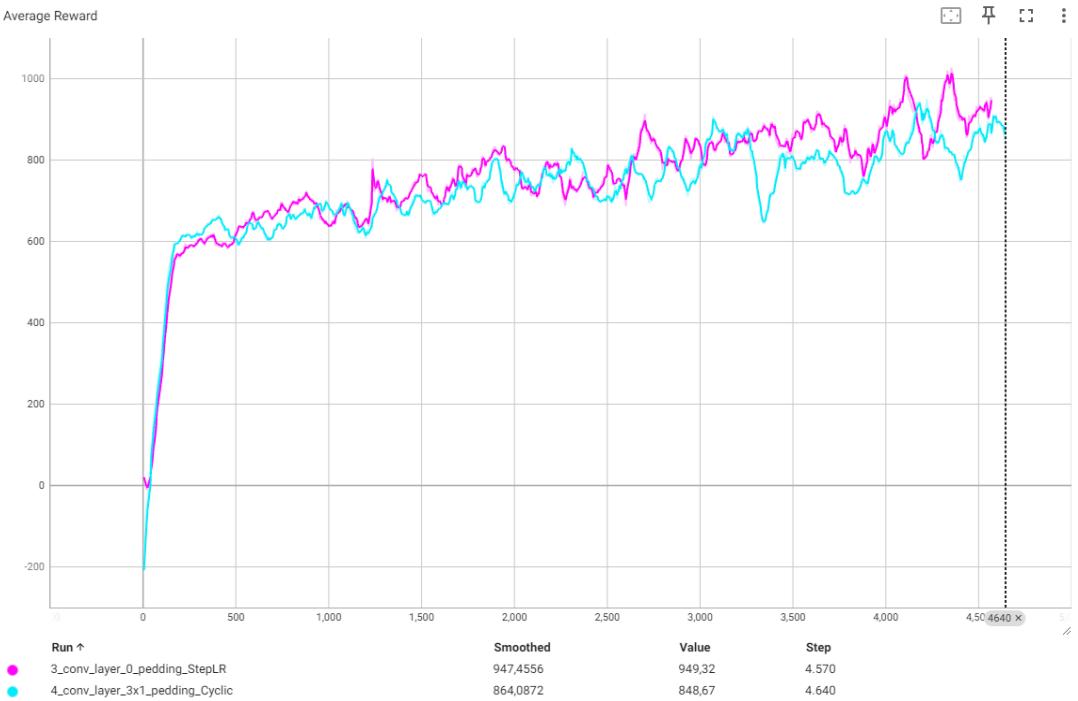


Run	Smoothed	Value	Step
3_conv_layer_0_pedding_StepLR	983,2201	983,12	4.570
4_conv_layer_3x1_pedding_Cyclic	912,3663	906,88	4.640



Run	Smoothed	Value	Step
3_conv_layer_0_pedding_StepLR	24,2308	24,2569	4.570
4_conv_layer_3x1_pedding_Cyclic	24,7665	24,7596	4.640





Die beiden Modelle zeigen anfänglich ein identisches Lernverhalten mit minimalen Abweichungen im Loss und im Reward auf, die aber womöglich durch den Zufall Faktor Epsilon entstanden sind. Dennoch zeigt das kleinere Modell (1) tendenziell einen leicht höheren Reward am Ende, aber dafür einen niedrigeren Q-Value als das größere Modell (2). Im Großen und Ganzen ist aber über den Zeitraum von 48 Stunden keine große Varianz der zwei Netzwerke zu erkennen, bis auf einen starken Ausreißer im Loss an Stelle 1250, der aber durch eine Unterbrechung des Trainings zurückzuführen ist.

5.4 Zusammenfassung der Ergebnisse:

Insgesamt kann man bezogen auf das eine Level keine gravierenden Unterschiede von Modell 1 zu Modell 2 sehen. Das Lernverhalten beider ist in etwa gleich und hebt sich nicht merklich voneinander ab. Der einzige

Unterschied, der hervorsticht, ist die Geschwindigkeit bzw. der Ressourcenbedarf der zwei Modelle. Im Vergleich nutzt das kleinere Modell (1) 27 % weniger Leistung als der Bedarf des größeren Modells (2).
Es gibt aber noch weitere Hyperparameter, die allerdings nicht verändert wurden, welche auch starke Auswirkungen auf die Performance des Modells haben können, wie z. B. die Größe der Batchsize.

6. Zusammenfassung und Ausblick:

6.1 Zusammenfassung des Projektes:

Das Projekt "Train RL Mario Agent" (q_play) demonstriert die Anwendung eines Double Deep Q-Networks (DDQN) auf dem Videospiel "Super Mario Land" mithilfe des PyBoy-Emulators. Insgesamt muss ich sagen, dass ich einiges durch den Entwicklungsprozess über den Aufbau und das Anwenden solcher RL-Methoden lernen konnte. Außerdem konnte ich durch die Nutzung eines Python-basierten Emulators zusätzlich das Arbeiten mit unterschiedlichen Plattformen näher verstehen. Besonders hat mir das Experimentieren mit den unterschiedlichen Netzwerken gefallen und das Arbeiten mit der Spielumgebung.

Eine große Schwierigkeit, welche ich anfänglich hatte, war das Verarbeiten des Beobachtungsraum der Spielumgebung für das CNN, welches ich aber im Praktikum großteils mit Hilfe lösen konnte. Das zweite Hindernis war die Cuda-Implementierung meines Heimrechners, den ich unbedingt für das Trainieren nutzen wollte, was ich aber auch gelöst bekommen habe. Nach diesen zwei Hürden gab es kaum noch nennenswerte Probleme, die zu komplexen Fehlern geführt haben, die ich nicht selbst durch Recherche gelöst bekommen habe.

6.2 Verbesserungen:

Obwohl das Projekt sehr gut und umfangreich geworden ist, muss ich sagen, dass es noch einige Möglichkeiten zur Weiterentwicklung offen lässt. Dennoch ist es sehr effizient und bietet die Möglichkeit, weitaus bessere Modelle mit der richtigen Hardware zu trainieren. Zudem wurde der RL-Agent nur auf dem ersten Level trainiert. Somit bietet es noch viel Spielraum nach oben, den Agenten über verschiedene Spiellevel und -situationen hinweg weiterlernen zu lassen. Auch der Aktionsraum, der sich bei dem Agenten nur auf Vorwärts, Rückwärts und Springen beschränkt, kann potenziell mit zwei weiteren Aktionen erweitert werden. Auch ist es möglich, das Projekt so zu erweitern, dass mehrere Agenten in separaten Umgebungen gleichzeitig das selbe Modell trainieren.

6.3 Ausblick:

“
Die zukünftige Forschung könnte sich darauf konzentrieren, die Lerngeschwindigkeit und -effizienz solcher Agenten durch verbesserte Algorithmen und Netzwerkarchitekturen weiter zu steigern. Ein spannendes Feld ist die Untersuchung des kooperativen und kompetitiven Multi-Agenten-Lernens in Mehrspieler-Videospielen, das neue Dimensionen der Interaktion und des strategischen Denkens eröffnet.

Darüber hinaus besteht großes Interesse daran, die entwickelten Techniken und Methodologien auf andere Anwendungsbereiche außerhalb der Spieleindustrie zu übertragen, beispielsweise auf Probleme der Robotik, autonomer Navigation und komplexen Entscheidungsfindung.

Der Bereich des Reinforcement Learnings (RL) ist ein spannender Bereich, in dem weiterhin stark geforscht wird. Es öffnet die Tür zu spannenden neuen Forschungsrichtungen und praktischen Anwendungen, die das Potenzial haben, unser Verständnis und unsere Nutzung künstlicher Intelligenz grundlegend zu erweitern.
“ *ChatGPT

7. Anhang Quelle:

A. Quellcode-Verzeichnis:

- **main.py:** Hauptskript zur Initialisierung und Ausführung des Trainings- oder Spielmodus des Reinforcement Learning (RL) Agenten.
  Beinhaltet Setup für die Command-Line Arguments (Argparse), Initialisierung des PyBoy Emulators, Konfiguration der Lernumgebung und die Trainings- bzw. Spiel-Logik.
- **gym_env.py:** Definiert die `MarioGymAI` Klasse, die als Schnittstelle zwischen dem PyBoy Emulator und dem RL-Agenten dient.
  Verantwortlich für die Verwaltung des Spielstandes, die Ausführung von Aktionen.
- **deep_q.py:** Implementierung der Double Deep Q-Network (DDQN) -Architektur,
  beinhaltet die Definition des neuronalen Netzwerks und Logik.
- **wrapper.py:** Enthält verschiedene Wrapper-Klassen zur Anpassung der Beobachtungen aus der Umgebung.
- **console_ui.py:** Hilfsklasse für Benutzerinteraktionen, zum Laden von Modellen und Anzeigen verfügbarer Checkpoints.
- **pyboy_gym.py:** Erweiterung des `PyBoyGymEnv` für angepasste Interaktionen mit dem Spiel "Super Mario Land".
- **qnet_interface.py:** Anpassung des Aktionsraumes und Belohnungsberechnungen spezifisch für "Super Mario Land".

B. Referenzen

***** Gekennzeichnete Abschnitte wurden durch ChatGPT zusammengefasst, generiert oder zitiert.

1. C.J.C.H. Watkins and P. Dayan, "Q-Learning," in Machine Learning, vol. 8, no. 3-4, pp. 279-292, 1992.
2. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning" arXiv:1312.5602, Dec. 2013.
3. H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," arXiv:1509.06461, Sept. 2015.
4. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms" arXiv:1707.06347, Jul. 2017