Westfälische Hochschule
Gelsenkirchen Bocholt Recklinghausen
University of Applied Sciences

Fachbereich: Informatik und Kommunikation

Wissenschaftliche Vertiefung

Version 0.1: Gym-µRTS und Proximal Policy Optimization

vorgelegt von:

Phoi-Hoa Nguyen, Philipp Würz

Studiengang: Praktische Informatik

Matrikelnummer: 201222300, 201222258

Prüfer: Prof. Dr. Wolfram Conen

Abgabetermin: _____

- Aufteilung der geschriebenen Seiten:
  Philipp S.1-12 & S.21-26, Hoa S.13-20 & S.27-28

Schlüsselwörter

Deep-Learning, µRTS, Intelligenz, Künstliche Intelligenz, Reinforcement-Learning, Proximal Policy Optimization

Glossar

- **Unit Action Simulation:** UAS ruft die RL-Policy iterativ auf, das heißt bei jedem Schritt wählt die Richtlinie eine Einheit aus und gibt ihr eine Aktion.

- **Gridnet:** Gridnet prognostiziert eine Aktion für jede Zelle in der Karte, sprich es prognostiziert alle Aktionskomponentenebenen in einem Schritt, die dann festlegen, welche Aktionen jede Einheit durchführen wird.

- **Shaped Reward:** Mit Shaping Rewards wird ein Agent dafür belohnt, dass er geeignete Schritte in Richtung des Ziels unternimmt. Diese zusätzliche Belohnung ergänzt die Environment Reward. Diese soll durch viele kleinere Belohnungen, den Agenten motivieren kreativer beim gewinnen zu sein.

- **PySC2:** PySC2 ist DeepMinds Python-Komponente der StarCraft II Lernumgebung (SC2LE). Sie stellt die StarCraft II Machine Learning API von Blizzard Entertainment als Python RL Environment zur Verfügung.

- **OpenAI Five:** OpenAI Five ist der Name eines Projekts für maschinelles Lernen, das als Team von Videospiel-Bots gegen menschliche Spieler in dem kompetitiven Fünf-gegen-Fünf-Videospiel Dota 2 antritt. Das System wurde von OpenAI entwickelt, einem amerikanischen Forschungs- und Entwicklungsunternehmen für künstliche Intelligenz (KI).

- **Action Space:** RL ist eine Methode, die sich an dem natürlichen Lernverhalten des Menschen orientiert. Menschliches Lernen ist dabei abbhänig von der Umwelt. Dabei sind unsere Handlungen im Kontext des Lernproblems durch einen bestimmten Aktionsraum (Action Space) definiert.

- **Proximal Policy Optimization:** Eine Algorithmus-Methode, welches zu den Verfahren der Policy-Gradienten-Methoden für das Reinforcement Learning gehört und zudem eine modifizierte Version von Trust Region Policy Optimization ist.

- **Trust Region Policy Optimization:** Eine Policy-Gradient-Methode für das Reinforcement Learning, welches eine KL-Divergenz-beschränkung besitzt, um sicherzustellen, dass die neue aktualisierte Policy nicht weit von der alten Policy entfernt ist.

- **Kullback-Leibler-Divergenz:** Beschreibt ein Maß für die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen. KL-Divergenz wird auch als relative Entropie bezeichneit.

- **Clipped Surrogate Objective:** Ein Clipping Mechanismus aus PPO, der die Wahrscheinlichkeitsverhältnis unter der neuen bzw. alten Policy zwischen einem bestimmten Bereich abschneidet und nicht zulässt, dass sie sich weiter von diesem Bereich entfernt.

- **Policy Gradient:** Gehört der Reinforcement Learning Techniken an, die auf der Optimierung parametrisierter Policies im Hinblick auf den erwarteten Ertrag bzw. auch langfristige kumulative Belohnung durch den Gradientenabstieg beruhen.

- **Mini-Batch:** Jeder Stapel wird als Mini-Batch bezeichnet, sobald die Trainingsdaten in kleine Stapel aufgeteilt werden. Der Begriff steht im Zusammenhang mit dem Training neuronaler Netze mittels Gradientenabstiegs.

- **Invalid Action Masking:**

- **TimeStamp:**

- **Environment:**

- **Game State:**

- **Pathfinding:**

- **Behavior Funktionen:**

- **Actor:**

- **Critic:**

Anmerkung: Glossar wird in laufe der Zeit weitergepflegt und erweitert.

Inhaltsverzeichnis

1 Einleitung / Problem

In den letzten Jahren haben sich Anwendungen von Deep Reinforcement Learning (DRL) Algorithmen auf Echtzeit-Strategie-Spiele (RTS), z.B. mit starken autonomen Agenten in diversen Schritten weiterentwickelt. Eines der bekanntesten Beispiele ist OpenAI's Dota 2 Bot , der neben professionellen Spieler, 99,4 % der Spieler in Öffentlichen Spielen besiegte. *(Vinyals, Oriol et al. (2020), [22])*

Ein weiteres prominentes Beispiel ist AlphaStar , für das beliebte RTS StarCraft 2. Aufgrund der hohen Komplexität dieser Spiele , ist das Trainieren des Agenten mit einem sehr hohen Rechenaufwand verbunden, die in der Regel den Einsatz von hunderten GPUs und tausenden CPUs über Wochen fordern (Deswegen wurde vorher meist nur Teilprobleme innerhalb der Spiele angegangen). *(Han, Lei et al. (2021), [07])*

Innerhalb dieses Papers wird das Problem des Ressourcenmanagement in zwei Beiträgen näher vorgestellt. Zunächst wird Gym-µRTS als eine schnell zu verwendende sowie deutlich vereinfachte RL-Umgebung für die RTS-Forschung vorgestellt. Dafür werden zunächst die Grundlagen für Deep Reinforcement Learning und Echtzeitstrategiespiele vorgestellt. Der zweite Beitrag beschäftigt sich mit unterschiedlichsten RL-Techniken. Zum Einstieg beschäftigen wir uns zunächst mit Policy-Gradient-Methoden , mit Hauptfokus auf Proximal Policy Optimization (PPO). Die Policy-Gradient-Methoden sind grundlegend für die jüngsten Durchbrüche bei der Verwendung von tiefen neuronalen Netzen zur Steuerung von Videospielen bis hin zu 3D-Locomotion. Jedoch sind gute Ergebnisse mit Policy-Gradient-Methoden schwer zu erzielen und mit einer großen Herausforderung verbunden, weil sie beispielsweise empfindlich auf neue Aktionen reagieren. Diese Schwäche von Policy-Gradient-Methoden werden von Forschern durch Ansätze wie z. B. Trust Region Policy Optimization (TRPO) versucht zu beseitigen, indem sie die Größe eines Policy-Updates versuchen einzuschränken oder anderweitig zu optimieren. PPO wirkt als modifizierte Version der TRPO und schafft ein gelungenes Gleichgewicht zwischen der einfachen Implementierung und Einschränkung der Größe eines Policy-Updates und dessen Optimierung. *(OpenAI (2020), [13])*

2 Was heißt eigentlich Deep Reinforcement Learning?

Der Begriff ist aus Deep Learning (DL) und Reinforcement Learning (RL) zusammengesetzt. DL ahmt die Prozesse im menschlichen Gehirn nach und verwendet neuronale Netze, um Muster in Daten zu erkennen und auf deren Grundlage Entscheidungen zu treffen. Bei RL hingegen geht es um Software-Agenten, die lernen, wie sie sich in einer Umgebung verhalten sollen, um eine bestimmte Belohnung zu maximieren. Zusammengeführt bezeichnet man die Technik als Deep Reinforcement Learning (DRL). Im weiteren Verlauf der Arbeit sollen kurz auf Begrifflichkeiten, sowie die groben Grundlagen eingegangen werden. Das RL ist ein Teilbereich des maschinellen Lernens, zusammen mit dem Supervised-Lernen (wird mit Daten trainiert, für die sowohl die aktuellen Eingaben als auch die richtigen Ausgaben bereits existieren, um Zukünftige Fälle vorherzusagen) und Unsupervised-Lernen (um verborgene Muster und Strukturen in Daten zu erkennen, genannt Mustererkennung). Dabei kann man die Strategien, RL-Probleme lösen, wie nachfolgend vereinfacht benannt. *(Achiam, Josh et al. (2020), [01])*

1. Policy based - Eine Strategie ist eine genaue Art und Weise, wie der Agent eine bestimmte Aufgabe ausführt. Bei diesem Ansatz liegt der Hauptaugenmerk darauf, die optimale Strategie (Policy) unter allen verfügbaren zu finden.
2. Value based - Der Schwerpunkt liegt darauf, den optimalen Wert zu finden und die kumulative Belohnung zu maximieren.
3. Action based - Der Fokus liegt auf den optimalen Aktionen, die bei jedem Schritt ausgeführt werden sollen.

RL verfolgt dabei einen weiteren Ansatz. Dieser beinhaltet - falls sie erfolgreich das Problem oder ein Teilproblem lösen - den Agenten eine passende Belohnung, in Form von steigenden Werten zu geben. Der Agent kann also durch bestimmte Züge Belohnungen erhalten bzw. weggenommen bekommen. Wie diese Belohnung ausfällt und wie der Agent mit Hilfe dieser Belohnung die „richtige“ Entscheidung trifft, ist das Zentrum der RL-Forschung. Der mathematische Rahmen der RL am einfachsten beschreibt, ist der Markov-Entscheidungsprozess. In diesem Rahmen befindet sich der **Agent** in einer **Umgebung,** in der er bestimmte **Aktionen** ausführen kann. Durch Beobachtung dieser Umgebung kann der Agent Aktionen durchführen, die den aktuellen Zustand verändern und Belohnungen gewähren. *(Achiam, Josh et al. (2020), [01])*

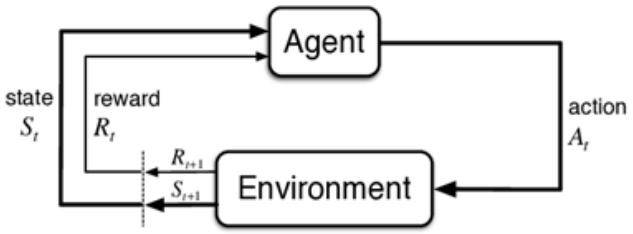

Abbildung 1 Markov-Entscheidungsprozess für RL (Sutton, Richard S & Barto, Andrew G. (2015), [19])

Im oberen Szenario wäre der Spieler der Agent. Aktionen sind Beispielsweise das Bewegen von Einheiten bei einem Echtzeitstrategiespiel. Dabei ist das gesamte Echtzeitstrategiespiel mit seinen Einheiten, Regeln und Terrain ein Teil der Umgebung. Belohnungen würde der Agent beispielsweise bekommen, wenn er mit den Einheiten wichtige Ziele, wie das Abbauen von Ressourcen erfüllt. Was dieses sehr vereinfachte Beispiel nicht darstellt, ist die Beziehung von mehreren Agenten untereinander - die zur selben Zeit - mit oft denselben Zielen, auch auf die Umgebung Einfluss nehmen. In einem Echtzeitstrategiespiel ist es wichtig, auf Aktionen des Gegners in Echtzeit zu reagieren, bzw. die eigenen Aktionen anzupassen. RL-Techniken sind sehr leistungsfähig. Durch die Hinzunahme von DL können RL-Techniken auf unterschiedlichste Art und Weiße genutzt werden. Aufgrund seiner Beschaffenheit beispielsweise, findet DRL offensichtlich Anwendungen, sowohl bei klassischen Spielen als auch bei Videospielen. Zu den größten Erfolgen zählen die bereits erwähnten Systeme AlphaGo und OpenAI, die beide in der Lage waren, große Leistungen zu erbringen. Bei den meisten Probleme mit simulierten Umgebungen ist der Zustandsraum (State space), äquivalent zum Beobachtungsraum (Observation spaces). Sowohl Zustands-, Beobachtungs- und auch Aktionsräume gibt es in beiden Varianten (d. h. diskret und kontinuierlich). In den einfachsten Beispielen stellt der Zustands- und Beobachtungsraum nur die Position des Agenten dar. Aber bei komplexeren und höherdimensionalen Beispielen kann der Zustands- und Beobachtungsraum zusätzliche Informationen oder komplexe visuelle Informationen, wie die RGB-Matrix der beobachteten Pixelwerte enthielt. *(Achiam, Josh et al. (2020), [01])*

Informationen über den Aktions- und Beobachtungsraum sind für die Bewertung von RL-Algorithmen sehr wichtig, da viele RL-Algorithmen nur für bestimmte Arten von Zustands-, Beobachtungs- und Aktionsräumen entwickelt wurden. Der PPO-Algorithmus kann beispielsweise für Umgebungen mit diskreten oder kontinuierlichen Aktionsräumen verwendet werden. *(AWS (2019), [06])*

Wie man erkennen kann, bieten DRL-Algorithmen viele Möglichkeiten Agenten deutlich effektiver zu gestalten. Was man bei DRL, bei Echtzeitstrategiespielen zu beachten hat, soll im nächsten Abschnitt beschrieben werden. Zunächst welche Eigenschaften haben diese?

3 Was sind Echtzeitstrategiespiele?

Echtzeitstrategiespiele sind komplexe Spiele, die von jedem Agenten ein hohes Maß an Dynamik, Flexibilität, taktischen Verständnis, sowie Einschätzen von Situationen verlangen. Man kann die meisten Echtzeitstrategiespiele auf 5 grundsätzliche Hauptprinzipien zurückführen:

Abbildung 2 – der Overlord spioniert eine feindliche Basis aus, die sich gerade im Aufbau befindet (Activison Blizzard (2010), [02])

1. Wie bereits im Namen zu erkennen ist, laufen die Spiele in **Echtzeit** ab. Dies soll den Spieler unter Zeitdruck setzen, und verlangt neben strategischer Weitsicht auch schnelle Reaktionen und Übersicht in chaotischen Situationen, Karten oder Kämpfen.
2. Fast alle Echtzeitstrategiespiele finden auf einer **begrenzten**, **rechteckigen Karte statt**. Der Spieler soll sich dabei einen Überblick in Vogelperspektive über das Geschehen machen. Ein weiterer Aspekt wäre dabei die **Line of Sight (LOS).** D. h., dass jedes Gebäude und jede Einheit des Spielers die Karte teilweise aufdecken. Entscheidend für das Spiel ist, ob der Rest der Karte - solang keine Einheit in der Nähe ist - durch den „**Nebel des Krieges**“ verdeckt wird. Das soll dem Spieler unrealistisches Wissen der Umgebung, zum Beispiel bei einer Truppenbewegung des Gegners, verschleiern.

3. Weiterhin nutzen Echtzeitstrategiespiele häufig eine **geringe Auswahl von Rohstoffen**, die als Grundlage für den Bau aller Gebäude, von Techniken und Einheiten dient. Meist werden diese Rohstoffe wie Nahrung, durch Arbeiter angebaut und Gold, Holz, Steine, etc. abgebaut. Weiterhin sind diese Rohstoffe begrenzt, was dazu führt, dass Kämpfe und die spätere Sicherung von Ressourcen ein Hauptteil des Spieles ausmachen.
4. Echtzeitstrategiespiele sind in **mehreren Spielphasen** unterteilt. Vereinfacht kann man diese in drei Phasen aufteilen. Die **Anfangsphase**, die meist 10-20 Minuten dauert, steht im zentralen Mittelpunkt die Anfangsressourcen zu erschließen, die Basis auszubauen und erste Einheiten zur Verteidigung zu erstellen (bei einem Rush viele) last but not least, die restlichen Karten mit Hilfe von Scout-Einheiten zu erkunden. In der **Haupt- und Kampfphase** sind die Anfangsressourcen meist verbraucht, was zum Bau weiterer Außenposten führt. Weiterhin müssen kleine bis große Mengen von Einheiten simultan kontrolliert und gegen den Gegner ins Feld geführt werden. Die Menge an Taktiken, in dieser Phase, sind bei den gegebenen Möglichkeiten nahezu grenzenlos. Zuletzt kommt immer die **Endphase**. In Dieser finalen Phase, werden die Siegesbedingungen gezielt erfüllt, um das Spiel zu gewinnen. Eine klassische Siegesbedingung ist die „Dominanz“, d. h. den Gegner militärisch zu bezwingen, indem man alle Gebäude zerstört.
5. Echtzeitstrategiespiele, wie die meisten Spiele mit mehreren Spielern, sind immer sehr **kompetitiv**. Wirkliche Gewinn-Spiel-Taktiken, sind meist erst bei **Multiplayer-Spielen** mit anderen oder gegen andere menschliche Spieler nötig.

4 µRTS & Gym-µRTS

µRTS heißt das Echtzeitstrategiespiel, was in der Arbeit verwendet wird. Es wird dabei als RL-Testumgebung für deutlich besser zu berechnende Vollspiel-RTS-Forschung genutzt, die sich auf die Grundlegenden Aspekte von RTS-Spielen konzentriert:

- Ernten von Ressourcen
- Rudimentären Base-Aufbau
- Steuerung der Anzahl von Arbeitern- und / Kampfeinheiten
- Steuern unterschiedlicher Einheiten auf der Karte
- Verteidigen von Einheiten und Angreifen von Gegnern

Die Entwicklung von KI-Techniken für RTS-Spiele ist dabei aus mehreren Gründen eine Herausforderung:

- Die Spieler müssen ihre Aktionen in Echtzeit ausführen, was das Rechenzeitbudget deutlich verringert
- Die Aktionsräume wachsen proportional mit der Anzahl der Einheiten im Spiel
- Die Möglichkeiten Belohnungen zu verteilen sind sehr spärlich
- Die Stochastizität der Spielmechanik, sowie der teilweisen Beobachtbarkeit. *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 2)*

Wenn alle diese Punkte berücksichtigt werden, können Systeme wie AlphaStar und OpenAIFive große Leistungen erzielen, diese sind aber mit hohen Rechenkosten verbunden. Beispielsweise nutzte AlphaStar 3072 TPU-Kerne und 50.400 preemptible CPU-Kerne für eine Dauer von 44 Tagen. Diese große Mengen an Rechenressourcen machen es schwierig, RTS-Forschung mit DRL durchzuführen. *(Vinyals, Oriol et al. (2020), [22])*

Im Paper werden dabei drei Möglichkeiten aufgezählt diesen Rechenaufwand zu verringern:

1. Der Erste Weg, ist der klassischste in der RTS-DRL – Forschung. Die Komplexität wird verringert, indem sich nur auf Teilprobleme z.B. Kampfszenarien oder den Basenbau konzentriert wird. *(Samvelyan, Mikayel et al. (2019), [17])*
2. Der Zweite Weg ist die Komplexität des gesamten Spiels zu reduzieren, indem man hierarchische Aktionsräume oder gescriptete Aktionen verwendet. *(Sun, Peng et al. (2018), [18])*
3. **Der dritte Weg ist die Verwendung alternativer Spielsimulationen**, die schneller und optimierter laufen. In diesem Paper wäre das Gym-µRTS, die für Forschung mit dem vollen Aktionsraum arbeitet und dabei erschwingliche Rechenressourcen verwendet. *(Andersen, Per-Arne et al. (2018), [04])*

Gym-µRTS ist dabei eine Reinforcement-Learning-Schnittstelle für das RTS-Spiel **µRTS**, was eine große Menge an KI-Techniken anbieten kann. Die Umgebung von AlphaStar PySC2 hat dabei einige Ähnlichkeiten zu Gym-µRTS *(Vinyals, Oriol et al. (2017), [21])*. Aufgrund das

Gym-µRTS eher für RTS-Forschung mit DRL ohne umfangreichen technischen Ressourcen gedacht ist, sind natürlich auch große Unterschiede vorhanden, auf die im weiteren Verlauf kurz drauf eingegangen wird. Beispielsweise wird der Aktionsraum bei PySC2 sehr menschenähnlich aufgebaut *(Vinyals, Oriol et al. (2017), [21])* (großer Vorteil dabei ist, dass es einfacher ist, diesen Agenten mit Menschen zu vergleichen, „höhere" Aktionen verbrauchen aber mehr Ressourcen) Gym-µRTS nutzt dagegen den Aktionsraum auf Ebene von Einheiten (was zur Folge hat, dass nur „primitive" Aktionen z.B. „Lauf nach Norden" an die Einheiten weitergegeben werden können). *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 3C)*

Trotz der vereinfachten Implementierung fängt **µRTS** wichtige Herausforderungen von RTS-Spielen ein:

- Kombinatorische Aktionsräume
- Entscheidungsfindung in Echtzeit
- Optionale partielle Beobachtbarkeit
- Stochastizität
  (Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 2)

Zunächst schauen wir uns die Umgebung von Gym-µRTS an. Der **Observation Space** ist zunächst eine Rechteckige Karte mit der Größe h x w (high x weight). Technisch gesehen wird dieser Observation Space im Form eines Tensors mit der Form (**h , w, n_f**), wobei **n_f** eine Anzahl von diversen Merkmalsebenen, mit binären Werten, sind. *Der vollständige Beobachtbarer* Observation Space (Beobachtungsraum) verwendet dabei 27 Merkmalsebenen, der Observation Space für *die partielle Beobachtbarkeit* verwendet 29 Merkmalsebenen. Eine Merkmalsebene kann als eine Verkettung mehrerer One-Hot-codierter Merkmale betrachtet werden. Mit einem kurzen Blick auf die erste Tabelle des Papers zeigt, wie der Observation Space codiert wird. *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 3A)*

Observation Features	Planes	Description
Hit Points	5	0, 1, 2, 3, ≥ 4
Resources	5	0, 1, 2, 3, ≥ 4
Owner	3	player 1, -, player 2
Unit Types	8	-, resource, base, barrack,worker, light, heavy, ranged
Current Action	6	-, move, harvest, return, produce, attack

Abbildung 3 Tabelle Observation Features (Huang, Shengyi & Ontanon, Santiago (2021), [09])

Es ist erkennbar, dass 5 verschiedene Beobachtungsmerkmale für den Observation Space relevant sind. Wie sieht diese Codierung als Tensor aus? Beispiel: Wir haben eine heavy Militäreinheit mit vier Trefferpunkten und keine Ressource zum Tragen. Spieler 1 gehört diese Einheit und es wird keine Aktion ausgeführt, dann sieht die One-Hot-codierten Merkmale folgenderweise aus. Die Reihenfolge der Merkmale kann man an der obigen Tabelle auslesen (Trefferpunkte, Resources, Owner, Unit Types, Current Action):

[0; 0; 0; 0; 1], **[1; 0; 0; 0; 0]**, **[1; 0; 0]**, **[0; 0; 0; 0; 0; 0; 1; 0]**, **[1; 0; 0; 0; 0; 0]**

Jedes dieser Merkmale codiert am Ende Eigenschaften für eine jeweilige Einheit. Jede Merkmalsebene enthält einen Wert für jede Koordinate auf der Karte. Die Werte für die 27 Merkmalsebenen für die Position einer solchen Einheit auf der Karte sind also (eine Einheit pro Feld!):

[0; 0; 0; 0; 1; 1; 0; 0; 0; 0; 1; 0; 0; 0; 0; 0; 0; 0; 0; 1; 0; 1; 0; 0; 0; 0; 0]

Für den Fall, dass ein Spiel mit teilweiser Sichtbarkeit gespielt wird, haben die Einheitentypen auch eine "sightRadius"-Variable, die steuert, wie weit jede Einheit sehen kann. Die Spieler können nur sehen, was ihre Einheiten sehen können. Ein weitere Wichtiger Aspekt von Gym-µRTS ist der **Action Space** (Aktionsraum). Im Vergleich zu traditionellen Reinforcement-Learning-Umgebungen ist der Entwurf des Aktionsraums von RTS-Spielen schwieriger, da es je nach Spielzustand eine unterschiedliche Anzahl von Einheiten zu steuern gibt und jede Einheit eine andere Anzahl von Aktionen zur Verfügung hat (was viele Kombinationen ermöglicht). Dies stellt eine Herausforderung für die direkte Anwendung handelsüblicher DRL-Algorithmen wie PPO dar, die im Allgemeinen von einem festen Ausgang für die Aktionen ausgehen. *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 3A & 3B)*

Um diese Probleme anzugehen, wird der Aktionsraum in zwei Teile zerlegt:

I. Den Einheitsaktionsraum, wo jeweils nur eine Einheit Aktionen zugeteilt bekommt

Im Einheitsaktionsraum ist die Einheitsaktion bei einer Karte der Größe h x w ein 8-dimensionaler Vektor mit diskreten Werten. Im Code werden die Koordinaten als X und Y dargestellt. Die erste Komponente des Einheitsaktionsvektors stellt die Einheit auf der Karte dar, an die Befehle ausgegeben werden sollen. Die Einheitsaktionstypen können an folgender Tabelle gut erkannt werden:

Action Components	Range	Description
Source Unit	$[0, h \times w - 1]$	the location of unit selected to perform an action
Action Type	$[0, 5]$	NOOP, move, harvest, return, produce, attack
Move Parameter	$[0, 3]$	north, east, south, west
Harvest Parameter	$[0, 3]$	north, east, south, west
Return Parameter	$[0, 3]$	north, east, south, west
Produce Direction Parameter	$[0, 3]$	north, east, south, west
Produce Type Parameter	$[0, 6]$	resource, base, barrack, worker, light, heavy, ranged
Relative Attack Position	$[0, a_r^2 - 1]$	the relative location of unit that will be attacked

Abbildung 4 Tabelle Action Components (Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 2)

Wenn der RL-Agent z. B. Arbeitseinheiten mit den Koordinaten **x = 2; y = 3** , **x = 0; y = 2**, **x = 3; y = 1** auf einer Karte der **Größe 5 x 4 die Aktion "move south"** erteilt, wird die Einheitenaktion wie folgt kodiert:

[3 + 2 ∗ 4; 1; 2; 0; 0; 0; 0; 0] [2 + 0 ∗ 4; 1; 2; 0; 0; 0; 0; 0] [1 + 3 ∗ 4; 1; 2; 0; 0; 0; 0; 0]

Die Position der Einheiten wird durch eine kleinen Term codiert. Bei der Multiplikation wird der X-Wert mit dem Spalten-Wert w der Karte multipliziert. Das Ergebnis ist die Spalte, wo sich die Einheit befindet. Der erste Summand der Addition steht für den Y-Wert. Dieser Definiert genau auf welcher Zelle, der ausgewählten Spalte sich die Einheit befindet. Bsp. **x = 3 + y = 2 * 4 => 11**. Einheit 1 befindet sich also auf **Feld 11**.

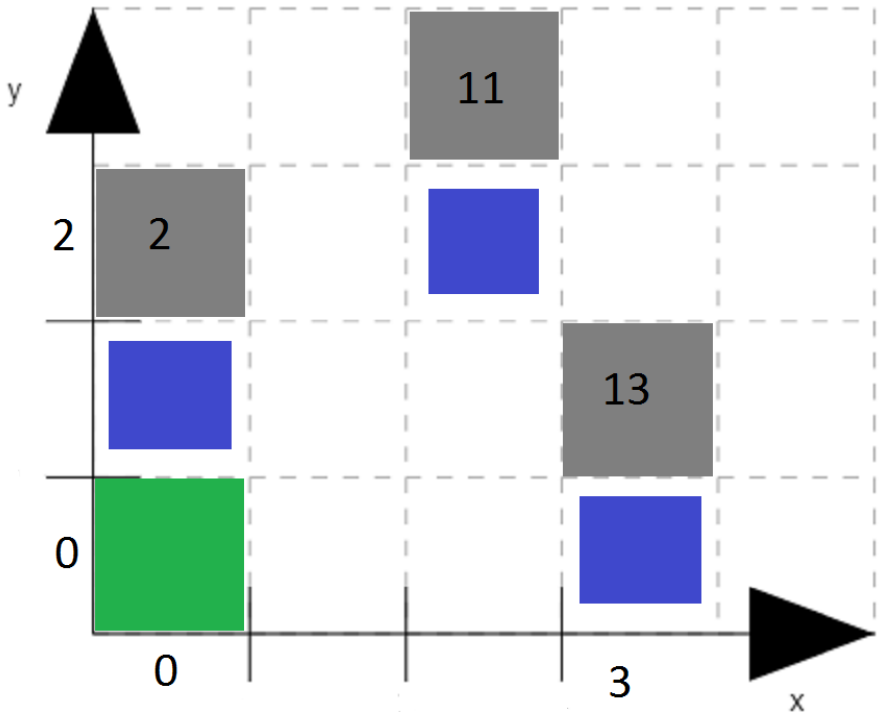

Abbildung 5 5x4 Spielfeld. Auf Grauen Feldern stehen Einheiten , Blau wo Sie hinwollen, Grün Ressourcen, Felder starten von Unten Links mit 0 x 0

Die übrigen Actionkomponenten stellen die verschiedenen Parameter dar, die ~~die~~ verschiedene~~n~~ Einheitsaktionstypen annehmen können. Der zweite Wert ist die Action Type, sie definiert, welcher Aktionentyp die Einheit durchführen will (1 für Move). Der dritte Wert ist der Move Parameter. Er beschreibt, in welche Richtung sich die Einheit bewegen möchte (Richtung Süden). Der Return Parameter verdeutlicht in welcher Richtung eine Einheit, die bei sich tragende Ressource ablegen will. Falls eine neue Einheit produziert werden soll (aus Gebäude erzeugt, oder von einem Arbeiter gebaut) sind die Parameter Produce Direction und Produce Type relevant. Ersteres zeigt in welche Richtung - von der Einheit aus - die neue Einheit, bzw. das Gebäude erscheint. Letzteres bezeichnet den Typ der neu erstellten Einheit (Beispiel: Eine Baracke oder eine neue Heavy-Unit). Falls vorhanden, wird zuletzt, noch die Position der zu attackierten feindlichen Einheit hinzugefügt. *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 3)*

II. **Sowie den Spieleraktionsraum der alle Einheits-Aktion für alle Einheiten des jeweiligen Spielers hält.**

Im Bereich der Spieleraktionen vergleichen wir zwei Möglichkeiten, Spieleraktionen an eine variable Anzahl von Einheiten pro Frame zu liefern. Einmal Unit Action Simulation (UAS), und zum zweiten Gridnet. An zwei Abbildungen lassen sich diese beiden Möglichkeiten gut erklären:

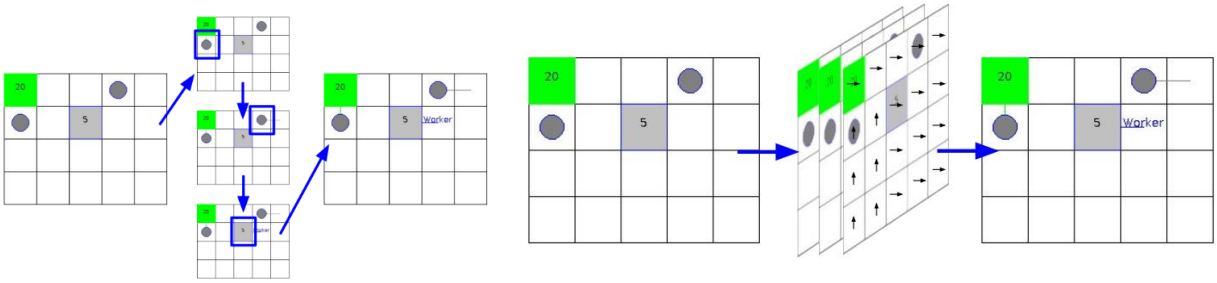

Abbildung 6 UAS vs Gridnet (Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 2)

Der RL-Agent sieht drei Einheiten, die auf Aufträge warten. **Auf der linken Seite** erkennt man die **Unit Action Simulation (UAS)**. UAS ruft die RL-Policy iterativ auf, das heißt bei jedem Schritt wählt die Richtlinie eine Einheit aus und gibt ihr eine Aktion. Sie berechnet dann einen "simulierten Spielzustand" in dem diese Aktion ausgegeben und alle potenziellen Belohnungen gesammelt wurden. Sobald alle drei Einheiten die Aktionen ausgegeben haben, werden die simulierten Spielzustände verworfen, die drei Aktionen gesammelt und an die tatsächliche Spielumgebung gesendet. *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 3)*

Gridnet, was auf der rechten Seite zu erkennen ist, prognostiziert eine Aktion für jede Zelle in der Karte, es sagt alle Aktionskomponentenebenen in einem Schritt voraus. Sie legen dann fest, welche Aktionen jede Einheit durchführen wird. Der RL-Agent gibt in einem einzigen Schritt Aktionen an jede Zelle in dieser Karte aus, d. h. er gibt insgesamt $4 * 5 = 20$ Einheiten-Aktionen aus. Die Umgebung führt dann die drei gültigen Aktionen aus (Aktionen in Zellen ohne spielereigene Einheiten werden ignoriert). *(Huang, Shengyi & Ontanon, Santiago (2021), [09] Abschnitt 3)*

Weiterhin werden **Belohnungen** je nach UAS oder Gridnet anders verteilt. UAS schreibt die Belohnungen den einzelnen Aktionen der Einheiten zu, während Gridnet die Belohnungen den Spieleraktionen zuteilt. Für dieses Paper wurde dafür einheitlich shaped reward functions genutzt, um die Agenten zu trainieren. Die Funktion gibt dem Agenten +10, für einen Sieg 0,

für Unentschieden -10, für Verlieren +1, für das Ernten einer Ressource +1, für die Produktion eines Arbeiters +0,2, für den Bau eines Gebäudes +1, für jede gültige Angriffsaktion, die er ausführt, +4 für jede Kampfeinheit, die der Agent produziert. Es gibt Belohnungen für den Frame, bei dem die Ereignisse initialisiert werden (Beispiel: ein Angriff dauert 5 Spielframes, aber die Angriffsbelohnung wird beim ersten Frame gegeben). Die shaped reward functions ist sehr ähnlich zur Reward-Function in Open AI Five for Dota 2. Wie bei Open AI Five ist das Ziel durch viele kleine Belohnungen (keine große Gewinn-Belohnung), dass der Agent nicht nur schnell gewinnen, sondern auch ein „schönes“ Spiel spielen möchte. Schnell gewinnen heißt in diesem Kontext nicht das der RL-Algorithmus gut ist! Den Agenten ist es dadurch einfacher möglich, deutlich länger Spiele mit größeren Belohnungen zu spielen. *(Huang, Shengyi & Ontanon, Santiago (2020), [08])*

Für das Training des Agenten wird im Folgenden ein populären Policy-Gradienten-Algorithmus, den Proximal Policy Optimization (PPO) verwendet. Hierbei dient die PPO-Implementierung - zusammen mit der Architektur - wie z. B. eines Nature-CNN, als Baseline. PPO wird als Kernalgorithmus in der Arbeit von Gym-µRTS genutzt. Im nächsten Kapitel wird PPO erklärt, was die Optimierung ist und welche Aufgaben dieser Algorithmus hat.

5 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) ist ein Algorithmus, welches 2017 vom OpenAI-Team präsentiert und eingeführt wurde. PPO gehört zu den Verfahren der Policy-Gradienten-Methoden für das Reinforcement Learning und wurde schnell zu einer der beliebtesten Reinforcement Learning-Methoden. PPO ermöglicht eine Richtlinienoptimierung und besitzt einige Vorteile gegenüber den führenden Anwärter wie z. B. Deep-Q-Learning, klassischen Policy-Gradient Methoden und Trust Region Gradient Methode. Der Grundgedanke von PPO besteht darin, die Policy direkt zu aktualisieren, um die Wahrscheinlichkeit von Aktionen zu erhöhen, die eine größere zukünftige Belohnung anbieten. Des Weiteren soll somit auch eine zu große Aktualisierung der Policy vermieden werden, indem die neue Policy nahe an der alten Policy gehalten werden soll. (Mónika Farsang (2021), [12])

Der Algorithmus zur PPO wird in seiner Gesamtheit in der folgenden Abbildung 1 als „Actor-Critic Style“ Implementierung dargestellt. Der Algorithmus lässt sich folgenderweise beschreiben, dass PPO unter Verwendung von *N* parallelen *actors* ausführt wird, die jeweils Daten sammeln und daraus ausgewählte *mini-batches* für das Training nutzen. Die *Actors* entscheiden welche Aktion getroffen werden und die *mini-batches* sind kleine Stapel der Trainingsdaten. Hierbei wird das Training für *K* Epochen unter Verwendung der Funktion „Clipped Surrogate Objective“ durchgeführt, welches in späteren Abschnitten nochmal genauer erklärt wird. Bei jeder Iteration wird die Policy π genau gleich $\pi_{\theta_{Old}}$ sein, nachdem in der Codezeile 3 die Umgebung mit $\pi_{\theta_{Old}}$ für jedem Zeitschritt T gesampelt wurde und die Optimierung, wie in Zeile 6 dargestellt, beginnt. Das heißt in der Zeile 3 und 4 interagiert die aktuelle Policy mit der Umwelt und erzeugt Episodenfolgen, für die sofort die „advantage function“ $\hat{A}_t$ unter Verwendung der angepassten „Baseline-Schätzung“ für die Zustandswerte berechnet werden. Und dann sammelt in der Zeile 6 ein zweiter Thread alle paar Episoden all diese Erfahrungen und führt einen Gradientenabstieg auf dem Policy-Netzwerk unter Verwendung des PPO-Objekts der Clips durch. (John Schulman (2017), [10])

Algorithm 1 PPO, Actor-Critic Style

1 **for** iteration=$1, 2, \ldots$ **do**
2 **for** actor=$1, 2, \ldots, N$ **do**
3 Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
4 Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
5 **end for**
6 Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
7 $\theta_{\text{old}} \leftarrow \theta$
8 **end for**

Abbildung 1: PPO, Actor-Critic Style Algorithmus (John Schulman (2017), [10])

Im Gegensatz zu herkömmlichen Methoden für den Policy-Gradienten und aufgrund der Verwendung der Funktion „Clipped Surrogate Objective“ kann mit PPO mehrere Epochen des Gradientenanstiegs für die „Samples“ durchgeführt werden, ohne destruktiv große Policy-Updates zu verursachen. Es kann aus den Daten mehr herausgeholt werden und zudem die Ineffizienz der „Samples“ reduzieren. Um PPO zu verstehen, ist der erste Schritt mit den Algorithmen für Policy-Gradienten vertraut zu sein. (Mónika Farsang (2021), [12])

5.1 Was sind Policy Gradient Methoden?

Policy Gradient Methoden gehören der Reinforcement-Learning Techniken an, die auf der Optimierung parametrisierter Policies in Bezug auf den erwarteten Ertrag bzw. Rendite durch Gradientenabstieg beruhen. Diese Methoden gehören zur Klasse der Techniken für die Suche nach Strategien, die den erwarteten Ertrag einer Strategie in einer festen Strategieklasse maximieren, im Gegensatz zu den traditionellen Ansätzen für die Annäherung an eine Wertfunktion, die Strategien aus einer Wertfunktion ableiten. Die Policy-Gradienten Algorithmen entscheiden über die zu ergreifende Aktion, indem sie den Zustand oder die Beobachtung einer Umgebung akzeptieren und dann für einen diskreten Aktionsraum eine Wahrscheinlichkeit für jede mögliche Aktion ausgeben. Eines der häufigste verwendete Gradientenschätzer hat die Formel: (Ruitong Huang (2020), [16])

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

Formel 1: Vanilla Policy-Gradient Formel (John Schulman (2017), [10])

Hierbei beschreibt π die Policy und θ die Gewichte des Netzes. So gesehen wird die Ausgabe des neuronalen Netzes durch seine Gewichte bestimmt und mit π_θ geschrieben. Die Funktion Advantage $\hat{A}_t$ beschreibt zum Zeitpunkt *t*, wie gut eine Aktion im Vergleich zur durchschnittlichen Aktion für einen bestimmten Zustand ist. Hier geht es darum, die hohe Varianz in der Gradientenschätzung zwischen der alten und der neuen Richtlinie zu verringern. Die Verringerung der Varianz trägt dazu bei, die Stabilität des Reinforcement Learning-Algorithmus zu erhöhen. Hier zeigt die Erwartung $\hat{E}_t$ den empirischen Durchschnitt über eine endliche Menge von Stichproben in einem Algorithmus an, der zwischen Stichproben und Optimierung wechselt. Policy-Gradienten haben auch eine sehr schlechte Stichproben-Effizienz, da sie eine riesige Anzahl von Schritten zur Optimierung benötigen.

Policy-Gradienten-Ansätze haben verschiedene Vorteile und ermöglichen z. B. die unkomplizierte Einbeziehung von Domänenwissen in die Policy-Parametrisierung wobei eine optimale Policy oft kompakter dargestellt, als die entsprechende Wertfunktion ist. Solche Methoden garantieren eine Konvergenz zu einer zumindest lokal optimalen Policy. Die Methoden können mit kontinuierlichen Zuständen und Aktionen und oft sogar mit unvollkommenen Zustandsinformationen umgehen. Hierbei kann die „Policy Loss“ Funktion als folgende Formel definiert werden: (Ruitong Huang (2020), [16])

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_\theta(a_t \mid s_t)\hat{A}_t\right].$$

Formel 2: Loss-function als Integration des Policy-Gradient (John Schulman, [10])

Es kann dazu motiviert werden mehrere Optimierungsschritte auf die Funktion L^{PG} mit der gleichen Bahnkurve durchzuführen bzw. mehrere Durchgänge über die Daten Laufen zu lassen, so dass mehr aus jeden „sample“ gelernt werden kann. Jedoch führt es oft bei einer Aktualisierung zu Problemen in der Schrittweite, welches dann die Policy zerstört. Ist sie zu klein, war der Trainingsprozess zu langsam und ist sie zu groß, gabt es zu viel Variabilität im Training. Deswegen bietet PPO die Möglichkeit, über einen eingebauten „Clipping Mechanismus“ eine zu große Aktualisierung zu verhindern. (John Schulman, [10])

5.2 Verwendung von PPO

Es gibt zwei Hauptvarianten von PPO, die sich **PPO-Penalty** und **PPO-Clip** nennen und als Reinforcement-Learning Algorithmus genutzt werden. PPO-Penalty bzw. „PPO mit Adaptive KL Penalty“ ist eine Alternative zu PPO-Clip. In dieser wissenschaftliche Arbeit wird PPO-Penalty nur als wichtige Baseline erwähnt, da PPO-Clip im Durchschnitt besser abschneidet und deswegen in den nächsten Abschnitt erklärt wird. (OpenAI Spinning Up (2018), [14])

PPO-Clip bzw. „PPO mit Clipped Surrogate Objective“ ist eine Art Clipping Mechanismus, der die Wahrscheinlichkeitsverhältnis unter der neuen bzw. alten Policy r_t zwischen einem bestimmten Bereich abschneidet und nicht zulässt, dass sie sich weiter von diesem Bereich entfernt durch die Verwendung von sogenannten „surrogate objectives“. In der folgenden Abbildung 2 ist der Algorithmus von „PPO mit Clipped Surrogate Objective“ als Pseudocode dargestellt.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Abbildung 2: Pseudocode PPO-Clip (OpenAI Spinning Up (2018), [14])

In der Zeile 1 der Abbildung 2 werden die Actor- und Critic-Netzwerke und der Parameter ϵ initialisiert. In Zeile 3 werden eine Reihe von Trajektorien aus der neuesten Actor-Policy in der Umgebung gesammelt. Im Anschluss wird in der nächsten Zeile 4 die genaue Belohnung für jede Kurve in jedem Schritt berechnet. Im nächsten Schritt 5 wird der Advantage $\hat{A}_t$ für jede Kurve aus dem neuesten Critic-Netzwerk berechnet. In Zeile 6 werden die Actor-Parameter durch stochastischen Gradientenanstieg aktualisiert, indem die PPO-Clip Zielfunktion über *K* Epochen maximiert wird. Hierbei ist der stochastische Gradientenanstieg (SGA) eine iterative Methode zur Optimierung einer Zielfunktion mit geeigneten Glättungseigenschaften und zielt auf die Maximierung einer Zielfunktion ab. (Tom Hope (2018), [20]) Sobald die Optimierung beginnt, wird die alte Policy die gleiche wie die aktualisierte. Das heißt, das Verhältnis ist 1. Wenn die Policy jedoch weiter aktualisiert wird, dann würde das Verhältnis an die „clipping“ Grenzen stoßen. In Zeile 7 werden dann die Parameter des Critic durch Gradientenabstieg auf den „mean squared error“ bzw. die mittlere quadratische Abweichung aktualisiert. Mean squared error (MSE) gibt an, wie sehr ein Punktschätzer um den zu schätzenden Wert streut. (JonskiC (2019), [11])

Als nächsten Schritt wird die „Clipped Surrogate Objective function“ L^{CLIP}, welches in Formel 3 dargestellt ist, betrachtet und das „Clipping Mechanismus“ genauer erklärt.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\ ]$$

Formel 3: Surrogat-Funktion, Clipping Mechanismus von PPO (John Schulman, [10])

- θ beschreibt die Gewichte des Netzes
- $\hat{E}_t$ bezeichnet den empirischen Erwartungswert über Zeitschritte
- r_t ist das Verhältnis der Wahrscheinlichkeit unter der neuen bzw. alten Politik (Policy)
- $\hat{A}_t$ ist der geschätzte Vorteil (Advantage) zum Zeitpunkt *t*
- **ε** ist ein Hyperparameter, normalerweise 0,1 oder 0,2

Der „Clipping Mechanismus“ bei PPO legt ein Clipping-Intervall für den Wahrscheinlichkeitsverhältnis-Term $r_t(\theta)$ in Formel 3 fest, der in einem Bereich [1 - ϵ , 1 + ϵ] abgeschnitten wird, wobei ϵ ein Hyperparameter ist. Dann entnimmt die Funktion das Minimum zwischen dem ursprünglichen Verhältnis und dem abgeschnittenen Verhältnis. Der Wahrscheinlichkeitsverhältnis-Term $r_t(\theta)$ lässt sich folgenderweise in Formel 4 darstellen:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$$

Formel 4: r(ϑ)-function als Wahrscheinlichkeitsverhältnis zwischen Aktion unter aktuelle und vorherige Policy (Mónika Farsang (2021), [12])

Das Wahrscheinlichkeitsverhältnis $r_t(\theta)$ wird zwischen der alten Policy $\pi_{\theta_{Old}}(a_t|s_t)$ und der neuen Policy $\pi_\theta(a_t|s_t)$ definiert. Man beachte, dass $r_t(\theta)$ größer als 1 ist, wenn die bestimmte Aktion für die aktuelle Policy wahrscheinlicher ist, als für die alte Policy. Und bei $r_t(\theta)$ kleiner als 1, wenn die Aktion für die aktuelle Policy weniger wahrscheinlich ist. (Alina Vereshchaka (2019), [03])

Insgesamt wird der Erwartungswert $\hat{E}_t$ über ein Minimum von zwei Termen berechnet: Der erste Term ist ein „unclipped objective“ und der zweite Term ein „clipped objective“. Die Schlüsselkomponente kommt aus dem zweiten Term, clip($r_t(\theta)$, 1-ϵ, 1+ϵ) $\hat{A}_t$, wo ein unclipped objective“ mit einer Beschneidungsoperation zwischen 1-ϵ und 1+ϵ abgeschnitten wird. Das heißt, PPO zwingt $r_t(\theta)$ innerhalb eines kleinen Intervalls um 1 zu bleiben bzw. zwischen 1-ϵ

und 1+ϵ. Das Clipping kann anhand eines Beispiels erklärt werden. (Andrew Zhang (2018), [05])

Hierbei wird ein Vektor v = [0,2, 0,6, 0,4, 0,1] genommen und auf einen Bereich [0,3, 0,5] zugeschnitten. Nach dem Clipping ist der Vektor v = [0,3, 0,5, 0,4, 0,3]. Das bedeutet, die Werte, die kleiner als der Minimalwert des Bereichs sind, bekommen den Minimalwert des Bereichs zugewiesen und die Werte, die größer als der Maximalwert des Bereichs sind, bekommen dementsprechend den Maximalwert des Bereichs zugewiesen. (OpenAI Spinning Up (2018), [14])

Im Anschluss wird das Minimum des „clipped-“ und des „unclipped objective“ genommen, so dass das endgültige „objective“ eine untere Schranke des „unclipped objective“ ist. Folglich können zwei Fälle betrachtet werden, die in der Abbildung 3 dargestellt werden:

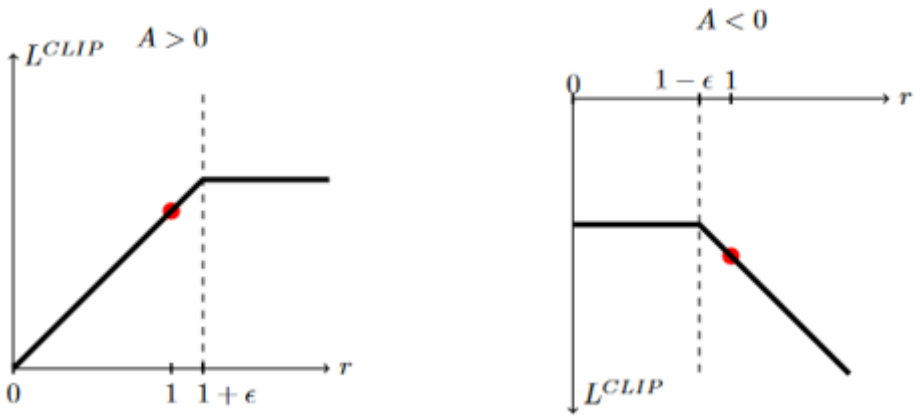

Abbildung 3: Diagramme der Surrogate-Funktion L^{CLIP} in Abhängigkeit von Wahrscheinlichkeitsverhältnis für positive und negative Advantage (John Schulman, [10])

- Advantage $\hat{A}_t > 0$ ist:

  Wenn $\hat{A}_t > 0$ ist, bedeutet dies, dass die Aktion besser ist als der Durchschnitt aller Aktionen in diesem Zustand. Dies ist dann der Fall, wenn die Aktion unter der aktuellen Policy sehr viel wahrscheinlicher geworden ist als unter der alten Policy. Wegen des Clips wächst $r_t(\theta)$ jedoch nur bis zu 1+ϵ und wird ab da „beschnitten“. Der Gradient wird mit einer flachen Linie blockiert, um die Richtlinienfunktion davon abzuhalten, zu viel zu aktualisieren, wenn die Aktion zu günstig ist. Es soll schließlich dazu führen, dass im Falle eines positiven Advantage, die Wahrscheinlichkeit der Aktion in diesem Schritt erhöht werden soll, jedoch nicht zu einer zu großen Belohnung führen soll. (Rokas Balsys (2020), [15])

- Advantage $\hat{A}_t < 0$ ist:

  Wenn $\hat{A}_t < 0$ ist, sollte die Aktion abgeraten werden, da sie sich negativ auf das Ergebnis auswirkt. Folglich wird r_t verringert und die Aktion bei der aktuellen Policy ist unwahrscheinlicher als bei der alten Policy. Aber wegen des Clips wird r_t nur bis auf 1-ϵ sinken.

  Auch hier soll keine große Änderung der Policy vorgenommen werden, da dies zu einem negativen Advantage führt, wenn die Wahrscheinlichkeit einer durchzuführende Aktion vollständig reduziert und zu viel aktualisiert wird. (Rokas Balsys (2020), [15])

Zusammenfassend lässt sich sagen, dass die „Clipping Mechanismen" dazu beitragen, dass man nicht zu gierig wird und versucht, zu viel auf einmal zu aktualisieren und den Bereich verlässt, in dem das „Sample" eine gute Schätzung bietet. Der Anreiz für das Wahrscheinlichkeitsverhältnis $r_t(\theta)$ sich außerhalb des Intervalls zu bewegen wird somit beseitigt. Denn der Clip hat die Wirkung einer Steigung. Wenn das Verhältnis größer 1+ϵ oder kleiner 1+ϵ ist, ist die Steigung gleich 0 bzw. keine Steigung.

Nach der Richtlinienaktualisierung erfolgt im nächsten Schritt die endgültige "Clipped Surrogate Objective Loss" Funktion, welches für das Trainieren eines Agenten genutzt wird. Das beschnittene L^{CLIP} wird hierbei mit zwei zusätzlichen Terme addiert, welches unter der Formel 5 zu sehen ist:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right],$$

Formel 5: Surrogat-Funktion mit Fehlerterm für Wertschätzung und Entropieterm (John Schulman, [10])

Die *c1* und *c2* sind Hyperparameter. Der erste zusätzliche Term der "Clipped Surrogate Objective Loss" Funktion beschreibt hierbei den „squared-error value loss" bzw. ein mittlerer quadratischer Fehler der Wertfunktion, die für die Aktualisierung der „Baseline-Netzwerk" bzw. „Critic-Netzwerk" zuständig ist: $L^{VF}(\theta) = (V_\theta - V^{Target})^2$. Das heißt es handelt sich um einen Fehlerterm für die Wertschätzung. Es wird hierbei versucht für das „Critic-Netzwerk", die Differenz zwischen dem geschätzten Wert und dem tatsächlichen Wert zu minimieren. (John Schulman, [10])

Der zweite Term beschreibt einen Entropieterm, der verwendet wird, um eine ausreichende Exploration für den Agenten zu gewährleisten bzw. zu fördern. Dieser Term drängt die Policy dazu, sich spontaner zu verhalten, bis der andere Teil des Ziels zu dominieren beginnt. (John Schulman, [10])

PPO, welches mehrere Epochen des stochastischen Gradientenanstiegs verwendet, wird hierbei in der Arbeit „Gym-µRTS“ als beliebten Policy-Gradienten-Algorithmus verwendet, um die Reinforcement-Learning-Agenten zu trainieren. Dabei bringt PPO die Stabilität und Zuverlässigkeit von TRPO mit sich und sind des Weiteren viel einfacher zu implementieren. Außerdem erfordern sie nur wenige Zeilen Codeänderung gegenüber gewöhnliche Policy-Gradient-Implementierungen und haben zuletzt eine bessere Performanceleistung. (John Schulman, [10])

6 Quellcodeanalyse

Alle hier zu lesenden Texte sind zu den jeweiligen Codebeispielen in unserem GitHub-Projekt [https://github.com/PWuerz/wiss.Vertiefung] hochgeladen.

6.1 Teil A: MicroRTS

6.1.1 Intro – ein Einfacher Agent und dessen Units

Zunächst schauen wir uns einfachere Agenten an. Komplexere Agenten besitzen neben allen Sachen die auch einfachere Agenten besitzen, noch deutlich komplexere Mechaniken, auf die im weiteren Verlauf noch eingegangen wird. Die einfachsten Agenten von MicroRTS sind ziemlich übersichtlich. Als Beispiel wird der Agenten RandomAI genutzt, der nur Random-Aktionen auswählen kann. Sofort zu erkennen sind einige überladene Funktionen, sowie eine Erweiterung des Agenten durch die abstrakte Klasse AI. AI ist die Basisklasse für jeden MicroRTS-Agenten, entweder direkt wie bei RandomAI, oder indirekt, über andere AI-Basisklassen, wie bei komplexeren Agenten. Die interessanteste Funktion bei allen Agenten ist die Funktion public PlayerAction getAction (int player, GameState gs). Diese Funktion ist bei allen Agenten die Funktion die die verschiedenen Eigenschaften, Verhaltensweisen und Taktiken des Agenten, sowie den GameState miteinander verknüpft.

In der AI-Basisklasse sind neben einigen Grundfunktionen, einige Funktionalitäten verpackt. Beispielsweise deklariert diese Abstrakte Klasse wichtige Funktionen wie getAction. Zwei Klassen sind bei erster Sicht sofort zu erkennen. Innerhalb des Konstruktors und der reset-Funktion wird die Klasse UnitTypeTable genutzt, und die Funktion getAction nutzt den GameState, sowie als Return-Typ die Klasse PlayerAction. Zusätzlich existiert eine Klasse Namens ParameterSpecification. Zunächst das UnitTypeTable.

Das UnitTypeTable speichert die Einheitentypen, die es im Spiel geben kann. Sie bestimmt auch die Eigenschaften der einzelnen Einheitentypen. Das UnitTypeTable bestimmt also das Gleichgewicht des Spiels (also wie stark Einheiten sind). Neben Funktionen, wie das Angebot ein UnitTypeTable per XML zu importieren oder zu exportieren und eine Menge weiterer Eigenschaften, sind die wichtigsten Funktionen von void setUnitTypeTable (int version, int crs). Wie bereits an der Tabelle zu erkennen, existieren vorgebaute Unit-Typen wie Worker, Base und Ressourcen, die abgebaut werden können. Als Entwickler kann man auch eigene Units einführen. Dabei ist jeder ein Unit-Typ innerhalb der Klassenstruktur des Typen

UnitType. Man kann also einfacher sagen das UnitTypeTable ist nichts Weiteres als eine Array-Liste des Typen UnitType mit zusätzlichen Eigenschaften und Funktionalitäten.

Die Klasse UnitType definiert mit seinen vielen Eigenschaften den abstrakten Typ einer Einheit. Neben den Eigenschaften wie Namen, hat diese Klasse boolesche Eigenschaften. Ob sich diese Einheit beispielsweise bewegen kann, sowie auch Array Listen oder was diese Einheit für andere Einheiten produzieren kann. Diese Klasse beinhaltet auch einige Funktionalitäten XML und JSON Dateienimportieren und exportieren zu können und diese zu Updaten. Jeder Unit wird genau einer UnitType zugesprochen.

Die Klasse Unit stellt die Instanz einer beliebigen Einheit im Spiel dar. Sie definiert wie oben genannt, welchen Typ diese Einheit hat, zu welchem Spieler sie gehört, welche Position diese auf dem Spielfeld einnimmt, sowie weitere Eigenschaften. Neben vielen Hilfsfunktionen, sowie der Möglichkeit per XML zu importieren und zu exportieren sind die folgenden zwei Funktionen die wichtigsten dieser Klasse:

- public List<UnitAction> getUnitActions (GameState s, int noneDuration)
- public boolean canExecuteAction (UnitAction ua, GameState gs)

Um diese Funktionen nachvollziehen zu können muss zunächst definiert werden, was eine Aktion im Kontext von MicroRTS überhaupt bedeutet.

6.1.2 Aktionen für MicroRTS – Ein Abstraktes Aktion Layer

Wie beim vorherigen Teil zu erkennen ist, hat zunächst jede Unit eine Menge von UnitActions, die am Ende eine Teilmenge der PlayerActions des Agenten sind. Wie wird eine Abstrakte Aktion in MicroRTS definiert? Als einzige Eigenschaft hat die Klasse AbstractAction eine Instanz einer Unit. Im Allgemeinen kann man zwei wichtige Funktionen erkennen:

- public abstract UnitAction execute (GameState pgs, ResourceUsage ru)
- public abstract boolean completed (GameState pgs)

Die Funktion execute(…) definiert, wie eine bestimmte Aktion auf der Gameoberfläche ausgeführt wird. Dagegen definiert die Funktion completed(…) ob eine Aktion beendet worden ist. Im Folgenden Bereich kann dies anhand einer expliziten Aktion Move nachvollziehen.

Zunächst stellt sich die Frage welche Eigenschaften, für eine Move-Aktion gebraucht werden. Da sich bei Move die Koordinaten der Einheit ändert sind die X und Y Koordinaten als Eigenschaften, die die Koordinaten nach einer Bewegungsaktion halten, zwingend nötig. Beispielsweise die Attack-Aktion benötigt ein Angriffsziel was immer eine Unit sein wird. Am Beispiel von Move können auch execute und completed besser verstanden werden. In completed von Move wird nur getestet, ob die Unit, die in der super-Klasse gehalten wird, dieselben X und Y Koordinaten besitzt, wie die Klasse Move. Bei execute hingegen wird zunächst der GameState abgefragt, genauer der PhysicalGameState. Dann wird mithilfe des Pathfinding Algorithmus (Standard A*) die beste Position auf der Karte ausgesucht. Falls eine Position auf der Karte gefunden wurde, und diese auch gültig ist, gibt die Funktion eine UnitAction zurück , sonst null. Wichtigste Funktionen für die Aktionsfindung werden in dieser Klasse definiert.

Wie bereits an der Tabelle im zweiten Kapitel zu erkennen, werden die verschiedenen UnitAction codiert (string und int). Beispielsweise ist die UnitAction Move 1, und der Versuch auf eine bestimmte Location anzugreifen 6. Desweitern wird auch die Richtung bei Bewegung der Einheiten also up, down, right und left codiert (String und int). Dabei entsprechen die Indizes den Konstanten, die in dieser Klasse verwendet werden. Der durch die Bewegungsrichtung verursachter Offset wird zuletzt ebenfalls definiert. Die 5 Konstruktoren stehen dabei für die 5 verschiedenen Aktionen. Beispielsweise UnitTyp und Koordinaten für eine Train-Aktion. Neben Funktionalität zum Importieren und Exportieren von XML und JSON Dateien definiert auch diese Klasse wieviel Ressourcen für diese UnitAction gebraucht werden. Die beiden Wichtigen Funktion in dieser Klasse sind public int ETA (Unit u) und public void execute (Unit u, GameState s). Die Funktion ETA gibt dabei die geschätzte Zeit des Abschlusses dieser Aktion zurück. Die Funktion execute nimmt dabei eine wichtige Rolle ein. Es wird zunächst geschaut welche UnitAction vorliegt. Bei einer Bewegung beispielsweise befindet man sich im Case TYPE_MOVE. Je nachdem in welche Richtung die Einheit die Move-Aktion ausführen will, werden vier weitere Cases für die Bewegungsrichtung ausgewählt. Auch für den Spieler gibt es einen Aktionsraum, dieser ist in der Klasse PlayerAction gekapselt.

Die Klasse PlayerAction verwaltet eine Liste von Paaren aus Units und UnitActions. Des Weiteren hat diese Klasse noch einiges an Funktionalität. Der erste Teil verwaltet die Menge der Aktionen, ob Non-Aktionen in der Liste sind. Sie weiß auch über eine andere Funktion, wie viele andere Aktionen in der Liste sind. Falls eine Unit keine Aktion zugewiesen bekommen

hat, wird fillWithNones() diese mit Non-Aktionen befüllen. Eine weitere wichtige Funktion für die späteren Aktionsmenge ist die Funktion merge, die zwei PlayerAction zu einer kombiniert (auch von den Ressourcen). Jetzt muss dieses Konstrukt aus Einheiten und Aktion mit den Agenten zu einem Layer zusammengeführt werden. Dafür wird als Grundlage die Klasse AbstractionLayerAI genutzt.

Zunächst sind einige neue Klassen in den Eigenschaften zu erkennen. Wie man sieht, bietet dieser Layer die Möglichkeiten PathFinding-Algorithmen einzubinden. Wie diese genau implementiert werden, wird im späteren Verlauf kurz gezeigt. Zu dieser Klasse reicht die Information, dass jede „Highlevel-Aktion“ wenigstens A* nutzt. In MicroRTS sind folgende Highlevel-Aktionen definiert:

- move (x, y)
- train (type)
- build (type, x, y)
- harvest (target)
- attack (target)

Diese Aktionen werden als einfache Funktionen definiert, die eine Hashmap füllen. Innerhalb dieser HashMap werden Units zusammen mit AbstractAction zusammengehalten. Die wichtigste Funktion in dieser Klasse ist die Funktion public PlayerAction translateActions (int player, GameState gs), die man sich näher anschauen sollte. Einfach gesagt wird in dieser Funktion alle bereits ausgeführten Einheitenaktionen zu einer PlayerAction zusammengefasst. Innerhalb dieser Funktion wird auch execute genutzt. In dieser Klasse wird auch definiert, wie eine Building-Position gesucht wird. Dies ist der grobe Unit- & Aktionsaufbau von MicroRTS. Schauen wir uns anhand eines expliziten Agenten an, wie alle diese Klassen zusammenarbeiten.

6.1.3 Taktiken – Komplexere Agenten und dem Beginn von Strategie und Verhalten

An diesen komplexeren Agenten HeavyRush kann man neben einigen Eigenschaften und den bereits oben genannten wichtigen Funktion getAction, weitere Funktionen mit dem Namen ...Behavoir erkennen. Zunächst implementiert jeder komplexere Agent eine Taktik. Zum Beispiel HeavyRush möchte so schnell wie möglich eine Baracke bauen und schwere Einheiten ausbilden, die direkt den Gegner angreifen sollen. Worker müssen dabei schnell die richtigen

Gebäude bauen und den Spieler dauerhaft mit Ressourcen versorgen. Wie man bereits an den Eigenschaften erkennen kann, muss man für die jeweilige Taktik die passenden UnitTypen vordefinieren (werden meist auf UnitTypeTable gelesen). Wie der Agent innerhalb der Taktik mit den verschiedenen UnitTypen umgeht, wird in der Behavior-Funktion definiert. HeavyRush hat folgende Behavoir-Funktionen:

- void baseBehavior (Unit u, Player p, PhysicalGameState pgs),
- void barracksBehavior (Unit u, Player p, PhysicalGameState pgs),
- void meleeUnitBehavior (Unit u, Player p, GameState gs),
- void workersBehavior (List<Unit> workers, Player p, GameState gs)

Genutzt werden diese Funktionen innerhalb der Funktion PlayerAction getAction(int player, GameState gs). An dieser Stelle wird pro Behavior-Funktion jeweils eine for each Schleife programmiert. Die Funktion wird bei jedem Spielzyklus mit dem aktuellen Spielstand aufgerufen und gibt zurück, welche Aktion der Agent in diesem Zyklus ausführen will. Mithilfe dem IF werden dabei nur die Units herausgezogen, die für das jeweilige Verhalten gebraucht werden. Im ersten If werden die Units durch den UnitType baseType aussortiert. Am Ende der jeweiligen Schleifen wird in die passende Behavior-Funktion gesprungen. Sobald alle Schleifen durch sind , wird die Funktion translateActions (player, gs) gecallt, die die Aktionen ausführt und zu einer PlayerAction zusammenfasst. Schauen wir uns im nächsten Abschnitt die Klasse GameState genauer an.

6.1.4 Die Umgebung – Status und Eigenschaften von MicroRTS

Der GameState besteht dabei aus mehreren Teilen. Der erste kleine Teil ist die Eigenschaft int time, die für ein Echtzeitstrategiespiel notwendig ist. Bei MircoRTS zeigt dieser Wert an, wie viele Zyklen seit Beginn des Spiels vergangen sind. Ein weiterer Wert ist das UnitTypeTable, das die Eigenschaften von Einheiten dieses Spieles speichert. Zwei Methoden, die in den Aktionen öfters genutzt werden, sind free und getAllfree. Mit den Hilfe dieser Funktionen getAllFree wird geschaut, ob sich Einheiten an der angegebenen Position befinden und keine Einheit eine Aktion durchführt, die diese Position nutzt. Zusätzlich werden mit boolean isUnitActionAllowed (Unit u, UnitAction ua) und boolean canExecuteAnyAction (int pID) die Aktionen kontrolliert. Die erste Funktion prüft, ob die vorgesehene UnitAction Konflikte mit einer anderen Action hat. Sie geht davon aus, dass die UnitAction ua gültig ist (d.h. eine der Aktionen, die die Unit potenziell ausführen kann). Die zweite Funktion hingegen prüft, ob ein

Spieler in diesem Zustand eine Aktion ausführen kann. Um die Aktionen mit den Einheiten für den GameState aufzubereiten, speichert UnitActionAssignment die Einheiten mit der Zeit, die eine zugewiesene Aktion braucht. Zuletzt ist die Funktion boolean cycle () zuständig einen Spielzyklus und alle zugewiesenen Aktionen auszuführen. Viele weitere Funktionen werden für komplexere Techniken, wie zum Beispiel getPlayerActionsSingleUnit (Unit unit), bei UCT genutzt. Als letzten Teil werden Eigenschaften der physikalischen Welt in einer weiteren Klasse gekapselt.

PhysicalGameState hält die physikalischen Grenzen der Map und die Units, die auf der Map sind. Wie groß diese Map ist, wird in dieser Klasse im Konstruktor definiert. Zudem weiß die Klasse, welche Spieler spielen. Damit implementiert sie auch die Funktionalität von winner () und gameover (), da diese Zustände Abhängig sind, ob die Spieler jeweils mindestens noch ein Spieler eine Unit auf dem Feld besitzen. Auch die Funktionalität, ob ein Feld frei oder bereits von einer anderen Unit besetzt wird, wird in dieser Klasse gekapselt. Weiterhin bietet die Klasse:

- Unit getUnitAt (int x, int y)
- Collection<Unit> getUnitsAround (int x, int y, int squareRange)
- Collection<Unit> getUnitsInRectangle (int x, int y, int width, int height).

Mithilfe dieser Funktionen können auf der Karte alle Einheiten in einem bestimmten Rechteck , oder Dreieck, wo im Zentrum die x und y Koordinate liegt, erfasst werden. Diese Funktionen helfen den Agenten später, die direkte Umgebung seiner Einheiten besser zu kennen, und damit analysieren zu können. Wie eine neue Unit hinzugefügt oder eine besiegte entfernt werden soll wird auch in dieser Klasse definiert. Weiterhin gibt es auch einige clone () und equals () Funktionen, die unter anderem definieren wie ein Agent kopiert wird und wie GameStates miteinander verglichen werden. Zusätzlich sind große Mengen von XML und JSON bearbeitende Funktionen in dieser Klasse vertreten.

6.2 Teil B: PPO

6.2.1 PPO im Projekt: Gym-microRTS

Der folgende Codeabschnitt aus ppo_diverse.py beschreibt die Verwendung von PPO, wodurch der Agent eine Optimierung der Policy erhält. In folgenden wird das Training für K Epochen in der Codezeile 20 "for i_epoch_pi in range(args.update_epochs)" durchgeführt. Dabei werden die Daten und die daraus ausgewählte mini-batches(minibatch_ind) für das Training gesammelt und genutzt. Das Wahrscheinlichkeitsverhältnis "ratio" wird in der Codezeile 31 zwischen der alten Policy und der neuen Policy definiert.

Ab der Zeile 36-40 wird die "Policy Loss"-Funktion definiert. Hierbei beschreibt "pg_loss1" den "unclipped objective"-Term mit der Berechnung des Wahrscheinlichkeitsverhältnis und des Advantages. "pg_loss2" beschreibt den "clipped Objective"-Term, in der das Wahrscheinlichkeitsverhältnis in einem Bereich zwischen $1-\epsilon$ und $1+\epsilon$ abgeschnitten wird und auch mit der Advantages berechnet wird. Die „Clipped Surrogate Objective function“ wird in der Codezeile 39 dargestellt, welches das Wahrscheinlichkeitsverhältnis unter der neuen bzw. alten Policy zwischen einem bestimmten Bereich abschneidet und nicht zulässt, dass sie sich weiter von diesem Bereich entfernt.

Im nächsten Schritt wird die "Value Loss"-Funktion bzw. “Clipped Surrogate Objective Loss“-Funktion ab der Codezeile 43-53 berechnet, die hierbei den „squared-error value loss“ bzw. ein mittlerer quadratischer Fehler der Wertfunktion beschreibt. Hier wird versucht die Differenz zwischen dem geschätzten Wert und dem tatsächlichen Wert zu minimieren.

Die endgültige “Clipped Surrogate Objective Loss“-Funktion wird in der Codezeile 55 "loss = pg_loss - args.ent_coef entropy_loss + v_loss args.vf_coef" definiert. Die Variablen "args.ent_coef" und "args.vf_coef" beschreiben die Hyperparameter. Die Variable "v_loss" und "entropy_loss" sind die zusätzlichen Terme, die bei der endgültige “Clipped Surrogate Objective Loss“-Funktion ergänzt werden. "v_loss" definiert den Fehlerterm für die Wertschätzung und "entropy_loss" beschreibt einen Entropieterm, der für die ausreichende Exploration des Agenten fördert.

6.2.2 PPO Code-Nutzung mit der Baseline

Der folgende Codeabschnitt aus der Baseline von pposgd_simple.py:

- **ratio** → Wahrscheinlichkeitsverhältnis zwischen der neuen und alten Policy
- **surr1** → Unclipped Objective
- **surr2** → Clipped Objective, Beschneidung zwischen 1-ϵ und 1+ϵ
- **pol_surr** → „Clipped Surrogate Objective"-Funktion (Clipping Mechanismus von PPO)
- **vf_loss** → Beschreibt den „squared-error value loss“ bzw. ein mittlerer quadratischer Fehler der Wertfunktion
- **total_loss** → Endgültige “Clipped Surrogate Objective Loss“-Funktion, welches für das Trainieren eines Agenten genutzt wird.

6.2.3 PPO im Projekt: RL-Adventures-2

Der folgende Codeabschnitt mit der Funktionen "ppo_iter" und "ppo_update" beschreiben die mehrere Epochen für die Aktualisierung der Policy und das Clipping Mechanismus, welches die Trainingsstabilität verbessern soll, indem es die Änderungen an der Policy bei jedem Schritt begrenzt.

In der Funktion "ppo_update" werden mehrere Epochen des Gradientenanstiegs für die "Samples" durchgeführt. Hierbei beschreibt "ratio" in der Codezeile 16 das Wahrscheinlichkeitsverhältnis (alten Policy und der neuen Policy).

Die Funktion "actor_loss" beschreibt die „Clipped Surrogate Objective"-Funktion, welches aus den Funktionen "surr1" (unclipped objective) und "surr2" (clipped objective) besteht.

Die Funktion "critic_loss" versucht die Differenz zwischen dem geschätzten Wert und dem tatsächlichen Wert zu minimieren ("squared-error value loss"-Funktion)

Die endgültige “Clipped Surrogate Objective Loss“ Funktion wird in der Codezeile 23 "loss" definiert, welches aus den Werten der “Clipped Surrogate Objective Loss“ Funktion(actor_loss) mit den zwei zusätzlichen Termen „squared-error value loss“(critic-loss) und Entropieterm (entropy) und deren Hyperparameter berechnet wird.

7 Ausblick

Die folgende Abgabe beinhaltet als Schwerpunkt die Codeanalyse vom Projekt Gym-microRTS und der Korrektur der letzten Abgabe. Zusätzlich wird noch ein Zeitplan im Anhang mitgeliefert.

Zu Kapitel 6: Wir wollten Ihnen die Texte, die wir für die Codeanalyse verfasst haben, innerhalb des wissenschaftliche Arbeitsdokuments zeigen, damit Sie ungefähr vergleichen können wieviel Text zu vorherigen Abgabe dazu gekommen ist. **Ist es Sinnvoll das sechste Kapitel in der Form in dieser Arbeit so einzubinden oder macht es mehr Sinn das sechste Kapitel zusammen mit den Codebeispielen(die umfangreich sind) als Anhang unten an die Arbeit anheften?**

Zu Kapitel 6.2 wird Gridnet und UAS noch weiter ausführlich ausarbeitet, um den ausführlichen Grundablauf genauer zu erklären und wie die Policy am Ende zu Entscheidung der Aktion beiträgt. Haben Sie vielleicht weitere Gedankenanstöße zu diesem Thema? Bisher haben wir nur folgende wissenschaftliche Paper darüber finden können:

- https://proceedings.mlr.press/v97/han19a/han19a.pdf
- https://arxiv.org/pdf/1707.07958.pdf

Die Verbindung von Gridnet und UAS zum restlichen Projekt ist uns anhand des Projektes und des Papers nicht eindeutig. Die restlichen Codebereiche, die wir vor allem im Abschnitt 6 analysiert haben, haben wir nach Absprache miteinander vermutlich gut verstanden.

Bis zum nächsten Termin werden wir hauptsächlich Brainstroming betreiben, welche Methodik und welche wissenschaftliche Frage unser wissenschaftliche Vertiefung haben soll. Wir tendieren momentan zu einem praktischen Projekt, da wir uns intensiv mit der Codeanalyse beschäftigt haben.

Erste Überlegungen wären hierbei, einen Agenten zu entwickeln, der eine Taktik, die wir uns ausgedacht haben, implementiert. Zusätzlich werden wir uns noch weitere praktische Gedanken machen, wie wir z. B. die Line of Sight einprogrammieren (grafisch nicht observeable). Zudem wäre es interessant einen Agenten zu implementieren der partially observeable ist. Und diesen mit den observeable Agenten von microRTS zu vergleichen.

Natürlich wären auch weitere „Pathfinding“ Methoden oder GameTreeSearch-Techniken zu implementieren, ein interessanter Ansatz microRTS zu erweitern. Nach momentanen Kenntnisstand wäre dies aber kompliziert.

8 Quellenverzeichnis

[01] Achiam, Josh et al. (2020): *Part 1: Key Concepts in RL.* *https://github.com/openai/spinningup/blob/master/docs/spinningup/rl_intro.rst 15.11.21*

[02] Activison Blizzard (2010): *Screenshot aus Starcraft 2*.

[03] Alina Vereshchaka (2019): Advanced Actor-Critic Methods (PPO, TRPO), *https://cse.buffalo.edu/~avereshc/rl_fall19/lecture_22_Actor_Critic_PPO_TRPO.pdf* 15.11.21

[04] Andersen, Per-Arne et al. (2018): *Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games.* https://arxiv.org/pdf/1808.05032.pdf 15.11.21

[05] Andrew Zhang (2018): Deep Reinforcement Learning for Classis Control Tasks, *https://cs230.stanford.edu/projects_fall_2018/reports/12449630.pdf* 15.11.21

[06] AWS (2019): *AWS DeepRacer Trainingsalgorithmus*. https://docs.aws.amazon.com/de_de/deepracer/latest/developerguide/deepracer-how-it-works-reinforcement-learning-algorithm.html 15.11.21

[07] Han, Lei et al. (2021): *TStarBot-X: An Open-Sourced and Comprehensive Study for Efficient League Training in StarCraft II Full Game*. https://arxiv.org/pdf/2011.13729v2.pdf 15.11.21

[08] Huang, Shengyi & Ontanon, Santiago (2020): *Action Guidance: Getting the best of Sparse Rewards and Shaped Rewards for real-time Strategy Games.* https://arxiv.org/pdf/2105.13807.pdf 15.11.21

[09] Huang, Shengyi & Ontanon, Santiago et al. (2021): *Gym-μRTS: Toward Affordable Full Game Real-time Strategy Games Research with Deep Reinforcement Learning.* https://arxiv.org/pdf/2105.13807.pdf 15.11.21

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov (2017): Proximal Policy Optimization Algorithms, *https://arxiv.org/pdf/1707.06347.pdf* 15.11.21

[11] JonskiC (2019): Mittlere quadratische Abweichung, *https://de.wikipedia.org/wiki/Mittlere_quadratische_Abweichung* 02.10.21

[12] Mónika Farsang, Dr. Luca Szegletes (2021): Decaying Clipping Range in Proximal Policy Optimization, *https://arxiv.org/pdf/2102.10456.pdf* 15.11.21

[13] OpenAI (2020): *Baselines*. https://github.com/openai/baselines 15.11.21

[14] OpenAI Spinning Up (2018). *Proximal Policy Optimization* https://spinningup.openai.com/en/latest/algorithms/ppo.html#pseudocode 08.10.21

[15] Rokas Balsys (2020): Reinforcement Learning tutorial (PPO), *https://pylessons.com/PPO-reinforcement-learning/* 03.10.21

[16] Ruitong Huang, Tianyang Yu, Zihan Ding, and Shanghang Zhang (2020): Deep Reinforcement Learning: Policy Gradient, Springer Verlag Nature Singapore

[17] Samvelyan, Mikayel et al. (2019): *The StarCraft Multi-Agent Challenge. https://arxiv.org/abs/1902.04043* 15.11.21

[18] Sun, Peng et al. (2018): *TStarBots: Defeating the Cheating Level Builtin AI in StarCraft II in the Full Game.* https://arxiv.org/abs/1809.07193 15.11.21

[19] Sutton, Richard S & Barto, Andrew G. (2015): *Reinforcement Learning: An Introduction*. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf 15.11.21

[20] Tom Hope (2018): *Einführung in TensorFlow*, O´Reilly Verlag

[21] Vinyals, Oriol et al. (2017): *StarCraft II: A New Challenge for Reinforcement Learning. https://www.researchgate.net/figure/Comparison-between-how-humans-act-on-StarCraft-II-and-the-actions-exposed-by-PySC2-We_fig3_319151530* 15.11.21

[22] Vinyals, Oriol et al. (2019): *Grandmaster level in StarCraft II using multi-agent reinforcement learning.* https://www.nature.com/articles/s41586-019-1724-z 15.11.21