Erstellung eines Custom Environments in OpenAI Gym für das Spiel Ökolopoly

Praxisprojektdokumentation
im Studiengang Allgemeine Informatik
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von:	Ralitsa Raycheva
Matrikel-Nr.:	111 33 802
Adresse:	ralitsa_ivanova.raycheva@smail.th-koeln.de

eingereicht bei:	Prof. Dr. Wolfgang Konen

Gummersbach, 03.07.2021

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

Der Biochemiker und Kybernetiker Frederic Vester hat sich tiefgehend mit dem Thema „Vernetztes Denken“ beschäftigt. In seinen Werken „Der blaue Planet in der Krise“ [4] und „Die Kunst, vernetzt zu denken“ [1] hat er deutlich darauf hingewiesen, dass sich kausales Denken und Handeln bei Umgang mit Fragen aus beliebigen Bereichen oft als kontraproduktiv oder sogar schädlich erweist. Das Anwenden von denselben bekannten linearen Mustern beim Lösen von Problemen, die in einer komplexen und sich stets ändernden Welt entstanden sind, führt auf die Dauer immer zu unerwarteten Ergebnissen. Ohne korrekte Erfassung der Funktionsweise und Verständnis für das Verhalten eines Systems, kann man mit diesem nicht richtig umgehen.

„Es ist notwendig, die komplexen, vernetzten Wechselwirkungen in der Natur wie in der Wirtschaft genauer zu erkennen und besser zu verstehen. Mit einem daraus resultierenden systemischen Management könnte es auch gelingen, eine Symbiose mit der heute mehr denn je gefährdeten Biosphäre zu schaffen.“ – so äußert er sich in Bezug auf die entstandene Umweltkrise [4].

Hauptziel des bekannten Kybernetikers war es, die Menschen zu animieren, über den eigenen Tellerrand zu schauen und ihre Vorstellung über die Welt zu vertiefen. Genau diesen Zweck erfüllt das Spiel Ökolopoly. Sein erster Entwurf wurde 1980 als Anhang in Zeitschrift Nature eingeführt und später 1984 für die Massen als Brettspiel bei Ravensburger veröffentlicht. In diesem Spiel bildet er ein vereinfachtes Weltmodell, das in acht Bereiche aufgeteilt ist. Mit seinen eingabauten Wechselwirkungen bietet das Spiel die Möglichkeit, das vernetzte Denken zu trainieren. Dadurch schult es das Erkennen von Mustern und den Beziehungen zwischen Komponenten innerhalb komplexer Systeme.

So ein Spiel bietet die Chance, die Möglichkeiten eines KI-Agenten zu erkunden, diese Art des vernetzten Denkens zu erlernen. Aus diesem Grund soll eine dafür geeignete Umgebung geschafft werden, wo „Entscheidungen und ihre Folgen in den verschiedensten Konstellationen“ [4] simuliert werden können.

In dieser Arbeit wird zunächst in Abschnitt 2 das Spiel Ökolopoly mit seinen gesamten Spielregeln und das Toolkit OpenAI Gym behandelt. In Abschnitten 3 und 4 wird die Umsetzung des Spiels als Custom Environment und das technische Realisieren der grafischen Oberfläche dokumentiert. Abschließend wird in Abschnitt 5 ein Fazit gezogen und Ausblick auf die Weiterentwicklung des Projekts gegeben.

1.1. Zielsetzung

Ziel der vorliegenden Ausarbeitung ist die Implementierung eines Python-Environments für das kybernetische Spiel Ökolopoly. Dabei sollen die Richtlinien von OpenAI Gym eingehalten werden und geeignete Action- und Observation-Spaces angelegt werden. Primär ist das richtige Abbilden der Abläufe im originalen Spiel Ökolopoly und das Erzeugen passender Testfälle, die

die Software auf ihre Korrektheit überprüfen. Zum Schluss soll eine simple grafische Oberfläche erstellt werden, die es menschlichen Spielern erlaubt, mit dem Spiel direkt zu interagieren und die Ergebnisse der angewandten Strategie leicht zu verfolgen.

2. Rahmenbedingungen

2.1. Ökolopoly

Ökolopoly ist ein Umweltssimulationsspiel, das auf kybernetischen Prinzipien beruht. Diese sind von Frederic Vester in den acht Grundregeln der Biokybernetik festgeschrieben [14] und beziehen sich auf die Abläufe eines Systems. Im Folgenden ist ein Teil davon sehr grob zusammengefasst, um ein besseres Verständnis des Spiels zu vermitteln.

Das erste Prinzip besagt, dass die negative Rückkopplung zwischen den Elementen im System für dessen erfolgreiches Funktioneren überwiegen muss. Außerdem ist Bedingung, dass seine Funktionen unabhängig von Größe und Rate des Wachstums vom System aufrechtgehalten werden können, laut des zweiten Prinzips. Weiterhin zieht ein anderes Prinzip die energiesparende und taktische Herangehensweise an Probleme im System vor. Dabei ist die Wiederverwendung von Organisationsstrukturen und Verfahren gewollt (hier passende Strategie), so das fünfte Prinzip. Das trägt eventuell dazu bei, dass eine Symbiose im System besteht und die Verbindung zwischen den einzelnen Elementen enger wird. In Ökolopoly sind diese Prinzipien ausgeprägt, stützend darauf ergibt sich der Regelkreis im Spiel. Die einzigen Komponenten wirken aufeinander und können auch indirekten Einfluss auf andere haben.

Schließlich erläutert Frederic Vester Kybernetik im nächsten Absatz wie folgt:

“Unter Kybernetik (vom griechischen kybernetes, der Steuermann) verstehen wir hier die Erkennung, Steuerung und selbsttätige Regelung ineinandergreifender, vernetzter Abläufe bei minimalem Energieaufwand.“ [14]

Ökolopoly stellt komplexe Zusammenhänge und Rückwirkungen zwischen den im Spiel beteiligten Komponenten dar. Diese sind so konzipiert, dass sie die Essenz der Abläufe in der Realität möglichst widerspiegeln. Zudem erfolgt der Übergang in verschiedene Zustände deterministisch.

Die Anleitung gestattet, dass Ökolopoly von bis zu sechs Spielern gespielt werden kann, die zusammen eine Lösung entwickeln und anwenden. Es ist nicht möglich, gegeneinander zu spielen. Ziel des Spiels ist, die vernetzten Zusammenhänge zu entdecken und darauf basierend, eine Strategie einzusetzen, um eine intakte Umwelt zu erhalten bzw. die Lage eines fiktiven Landes zu verbessern.

2.1.1. Lebensbereiche

Im Spiel sind die acht Lebensbereiche - Sanierung, Produktion, Umweltbelastung, Aufklärung, Lebensqualität, Vermehrungsrate, Bevölkerung und Politik - repräsentiert. Auf Grundlage davon ist das Gesamtwirkungsgefüge aufgebaut. Jeder Lebensbereich hat unterschiedliche Grenzwerte und Einflussgrößen. Diese werden in der folgenden Tabelle kompakt dargestellt:

Lebensbereich	Untergrenze	Obergrenze	Einflussbereich
Sanierung	1	29	F1, F2 (Produktion)
Produktion	1	29	F3, F4 (Umweltbelastung), FC, V
Umweltbelastung	1	29	F5, F6 (Lebensqualität)
Aufklärung	1	29	F7, F8 (Lebensqualität), F9 (Vermehrungsrate)
Lebensqualität	1	29	F10, F11 (Vermehrungsrate), F12 (Politik), FD
Vermehrungsrate	1	29	F13 (Bevölkerung)
Bevölkerung	1	48	F14 (Lebensqualität), FA, W
Politik	-10	37	FB
Aktionspunkte	1	36	Sanierung, Produktion, Aufklärung, Lebensqualität, Vermehrungsrate

Tabelle 1: Grenzwerte und Einfluss der einzelnen Bereiche

Beispielsweise kann Sanierung Werte von 1 bis 29 annehmen und wirkt auf die Felder 1 und 2 (F1 und F2). Der hinter Feld 2 in Klammern notierte Lebensbereich Produktion weist darauf hin, dass Produktion von Feld 2 beeinflusst wird. Dies bedeutet, dass Sanierung indirekt Auswirkung auf Produktion hat. Die Buchstaben V und W sind Multiplikationsfaktoren, die bei Errechnung der Bevölkerung und Aktionspunkte (Fenster A, auch FA) beteiligt sind. Sie hängen von Wert Bevölkerung bzw. Produktion ab.

Die Zahlen zwischen der Unter- und Obergrenze eines Bereiches geben an, in welchem Zustand er sich befindet. Nähert sich die Produktion, Umweltbelastung, Vermehrungsrate oder Bevölkerung ihrem Maximum, bedeutet dies, dass dieser Bereich beeinträchtigt ist. Je mehr die Sanierung, Aufklärung, Lebensqualität oder Politik sich der Obergrenze nähert, desto mehr blüht diese auf. Das gewährleistet aber keine weitgehende positive Auswirkung innerhalb des Systems. Beispielsweise erhöht sich die Vermehrungsrate rasch mit dem Wachstum der Lebensqualität und dadurch auch die Bevölkerung. Das richtige Erkennen des wirklichen Zusammenspiels und der Selbstregulation der einzelnen Komponenten im Spiel ist damit von großer Wichtigkeit bei der Entwicklung einer langfristig günstigen Strategie.

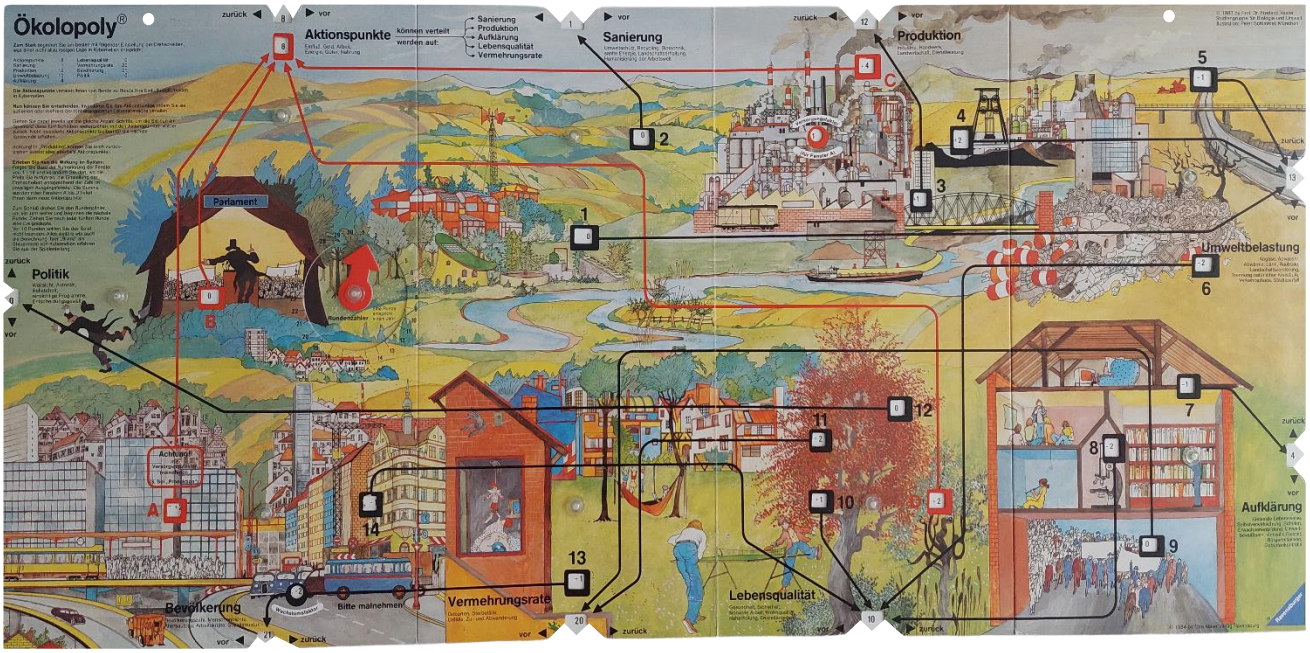

Abbildung 1: Spielbrett

Das reale Spielbrett ist in Abbildung 1 gezeigt.

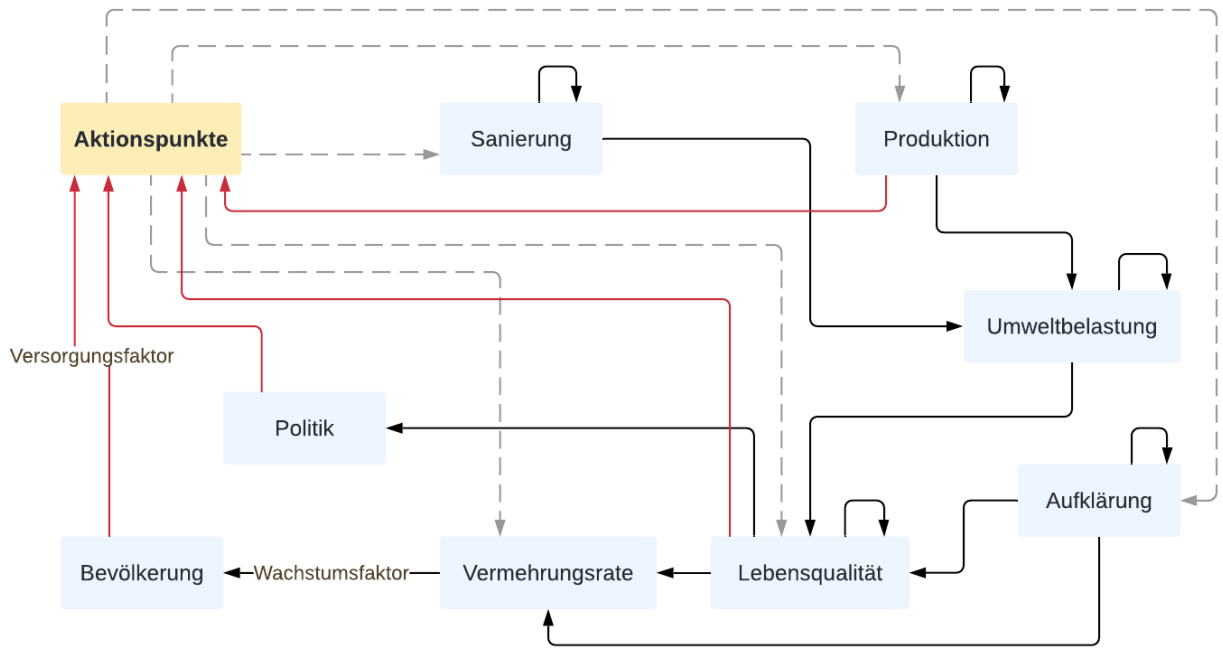

Abbildung 2: Vereinfachte Darstellung des Spielbretts

Abbildung 1 veranschaulicht die Verknüpfungen zwischen den einzelnen Bereichen. Die Aktionspunkte können optional unter den fünf Bereichen - Sanierung, Produktion, Aufklärung, Lebensqualität und Vermehrungsrate verteilt werden. Während die meisten Bereiche andere

beeinflussen, sind Sanierung, Produktion, Umweltbelastung, Aufklärung und Lebensqualität auch selbstregulierend. Bevölkerung wird zusätzlich mit dem Faktor Wachstum errechnet. Schließlich werden die Aktionspunkte für die nächste Runde in Abhängigkeit von Produktion, Lebensqualität, Politik, Bevölkerung und Versorgungsfaktor gebildet.

2.1.2. Vorbereitung

Die Lebensbereiche, Aktionspunkte und Rundenanzahl werden auf die jeweilige Startposition eingestellt. Jede Runde steht für ein Regierungsjahr. Die Ausgangslage entspricht der des fiktiven Landes „Kybernetien“, andere Ausgangslagen sind dabei auch möglich. Vor Beginn jeder Runde teilt der Spieler die ihm zur Verfügung stehenden Aktionspunkte den erlaubten Bereichen zu. Dabei ist ihm überlassen wie er vorgeht – ob er diese spart oder komplett investiert.
Zusätzlich ist dem Spieler erlaubt bei Produktion abzuziehen, was allerdings Aktionspunkte kostet.

Lebensbereich	Startposition
Sanierung	1
Produktion	12
Umweltbelastung	13
Aufklärung	4
Lebensqualität	10
Vermehrung	20
Bevölkerung	21
Politik	0
Aktionspunkte	8

Tabelle 2: Ausgangssituation von „Kybernetien“

2.1.3. Spielablauf

Der Spieler verfolgt das Wirkungsgefüge, das sich durch die Felder ergibt. Dadurch werden die neuen Werte der Bereiche erzeugt. Dabei sind die folgenden Sonderfälle zu beachten:

a) Der Wert von Bevölkerung ergibt sich aus der Multiplikation der Werte von Vermehrungsrate (Feld 13) und Wachstumsfaktor.
b) Erscheint eine Zahl mit +/- im Feld 9 darf der Spieler die Vermehrungsrate im Rahmen des Wertes im Feld beliebig steuern. Ein +/-3 gibt beispielsweise die Möglichkeit, Werte von -3 bis +3 zu wählen.

Anschließend ergibt sich der Zuwachs an Aktionspunkten für die nächste Runde aus der Summe der Felder A bis D. Zusätzlich zieht der Spieler jede fünfte Runde eine Ereigniskarte, die unerwartete äußere Einflüsse simulieren soll, was hier nicht weiter betrachtet wird.

2.1.4. Spielende

Gerät ein Bereich außerhalb seiner erlaubten Grenzwerte, wird ein sogennanter Umkippeffekt erreicht. Hiermit ist das Spiel beendet. Dies kann zwei Interpretationen haben – ein Bereich wurde so weit entwickelt, dass eine neue Verfassungsform erreicht wird oder er wurde so vernachlässigt, dass es zu einem Zusammenbruch gekommen ist. Das Spiel bietet eine Variante für eine Extremstrategie, die hier nicht weiter betrachtet wird.

2.1.5. Bewertung einer Strategie

Der Spieler muss zehn bis maximal dreißig Runden überleben und soll eine hohe Bilanzzahl erreichen. Je nach Wert der Bilanzzahl kann der Spieler seine Strategie bewerten. Ist das Spiel vor der zehnten Runde abgelaufen, wird keine Bilanz gebildet. Wenn mindestens die zehnte Runde erreicht wurde, wird die Bilanz wird nach folgender Formel errechnet:

$$B = \frac{(3 \times Feld\ D + Politik) \times 10}{Anzahl\ Runden + 3}$$

2.2. OpenAI Gym

OpenAI Gym [6] ist ein in Python entwickeltes OpenSource Toolkit für Erstellung und Vergleich von verschiedenen Reinforcement Learning (RL) Algorithmen.

Es stellt verschiedene Environments in standardisierter Form als Bibliothek bereit, was das Training und Testen von verschiedenen RL-Agenten ermöglicht. Das Hauptziel dabei ist Reproduzierbarkeit von Forschungsergebnissen zu erreichen und einfache Tools bereitzustellen, die den Einstieg im KI-Bereich erleichtern.

2.2.1. Environments

Alle Environments von OpenAI Gym implementieren das Shared Interface **`Env`** [13]. Grundlegende Funktionen sind **`reset()`** und **`step()`**, die minimal erfordert werden, damit die bestimmte Umgebung als Environment von Gym angenommen wird.

`reset()` initialisiert das Environment und gibt eine Anfangsobservation zurück. Anschließend führt **`step()`** die Dynamiken der entsprechenden Umgebung aus. Inputparamater ist dabei ein Actionsobjekt von Typ Space. Rückgabewerte sind:

- **observation** – vom Typ Space, repräsentiert den jeweiligen Zustand des Environments nach Ende einer Episode bzw. Ausführung der **`step()`**-Funktion
- **reward** - vom Typ float, vermittelt die intermediäre Belohnung nach Ausführung der vorherigen Action
- **done** – vom Typ bool, gibt an, ob die aktuelle Episode zu Ende ist. Ist dies nicht der Fall, wird **`step()`** aufgerufen.

- **info** – vom Typ dict, enthält beliebige Information, die aber für den Lernprozess des Agenten nicht eingesetzt wird

2.2.2. Registry

Jedes Environment kann im Register von Gym angemeldet werden und später mittels `gym.make('Name')` erstellt werden. Dies ist besonders hilfreich, wenn verschiedene Versionen eines Environments verglichen werden sollen. Eine Abmeldung ist auch möglich. Diese Aktionen sind in der Datei oekolopoly/oekolopoly/__init__.py definiert.

3. Umsetzung

Im folgenden Kapitel wird die grundlegende Logik der Klasse **`OekoEnv`** mit Hilfe von kurzen Ausschnitten des Quellcodes erläutert. Der vollständige Programmcode befindet sich in einem Github-Repository**.**

Das Ökolopoly-Environment gibt die oben beschriebenen Abläufe des Originalspiels wieder. Die initiale Observation entpsricht den Startpositionen in Tabelle 2. Bei der Umsetzung des Spiels wurde das Ziehen von Ereigniskarten und die Extremstrategie nicht berücksichtigt, da sie sich nicht als grundlegend für den essenziellen Spielablauf erweisen.

Sowohl Action Space als auch Observation Space sind als **`MultiDiscrete`** angelegt, was die spätere Anwendung von verschiedenen RL-Algorithmen erlaubt, da diese meistens dieses Format unterstützen. Die Zahlen, die als Argumente eingegeben werden, umfassen die Werte von Null bis einschließlich des Zahlenwertes selbst.

3.1. Action Space Ökolopoly

Die fünf Lebensbereiche, in denen Aktionspunkte investiert werden können, und der Spezialfall „Aufklärung > Vermehrungsrate“ bilden den Action Space des Environments.

Ein besonderer Augenmerk liegt auf die Produktion und auf dem Spezialfall, da diese über einen zusätzlichen Raum für Punktabzüge verfügen sollen. Wenn sie den höchsten Wert erreicht hat, kann die Produktion im Grunde genommen um bis zu 28 Punkte sinken, ohne dass das Spiel in derselben Runde zum Ende kommt. Die eingegebene Zahl muss also Werte zwischen -28 und 28 darstellen können, die Null und Grenzen inklusive. Somit ergibt sich ein Range von 57. Dieselbe Logik wird auch bei dem Spezialfall angewandt.

Eine Tranformation der Action Space wird durchgeführt, mittels der Arrays **`Amin`** und **`Amax`**, die die Grenzwerte der Actions speichern.

3.2. Observation Space Ökolopoly

Das Observation Objekt enthält zehn Integer - alle acht Lebensbereiche, einen Flag, ob ein gültiger Zug gewählt wurde und die Anzahl sowohl der Runden als auch der Aktionspunkte für die nächste Runde. Bei der Definition des Observation States wurde **`MultiDiscrete`** ein Array übergeben, das die oberen Grenzwerte der Bereiche enthält, so wie sie in Tabelle 2 aufgelistet sind.

Der Observation Space wird in ein für das Environment geeignetes Format transformiert, um mit **`MultiDiscrete`** kompatibel zu sein. Dies ist auf Grund der Produktion und Spezialfalls nötig, da diese auch mit negativen Werten arbeiten.

Die interne Repräsentation wird berechnet mittels der Arrays **`Vmin`** und **`Vmax`**, die jeweils das Minimum und Maximum eines Bereiches speichern. Nach Durchführung der erforderlichen Berechnungen für die übliche Observation wird diese umgerechnet und erst dann als return übergeben. Dazu wird noch überprüft, ob es von dem definierten Observation Space angenommen werden kann. Dies ist am Beispiel der **`return()`**-Funktion zu betrachten:

```
297     def reset(self):
298         self.V = np.array([
299              1,  #0 Sanierung
300             12,  #1 Produktion
301              4,  #2 Aufklärung
302             10,  #3 Lebensqualität
303             20,  #4 Vermehrungsrate
304             13,  #5 Umweltbelastung
305             21,  #6 Bevölkerung
306              0,  #7 Politik
307              1,  #8 Valid turn
308              0,  #9 Round
309              8, #10 Points
310         ])
311 
312           self.obs = self.V - self.Vmin
313           assert (self.observation_space.contains(self.obs)),
                      "AssertionError: obs not in observation_space"
314 
315           return self.obs
```

Codeschnipsel aus der Klasse OekoEnv

3.3. Funktionen

3.3.1. Update_values()

Die Funktion bildet das Gesamtwirkungsgefüge des Spiels ab. Als Eingabeparameter werden die vorliegenden Observation und Action akzeptiert. Zunächst wird der Wert des jeweiligen Felds in einer Variable (bspw. Feld 1 in **`box1`**) gespeichert und dann dem bestimmten Lebensbereich zugerechnet. Weiterhin wird geprüft, ob dieser Bereich außerhalb des zulässigen Ranges gelangt ist. Ist dies der Fall, werden die Variablen **`done`** auf **`True`** und **`done_info`** mit dem passenden String gesetzt.

Der Spezialfall „Aufklärung wirkt auf Vermehrungsrate“ wird so behandelt, indem zunächst die in der Action eingegebene Zahl in **`extra_points`** gespeichert wird und später an die Funktion zur Berechnung von Feld 9 übergeben wird. Da Aufklärung diesen Fall ab Einstellung 21 unterstützt und erstmals die Vergabe von -3 bis zu 3 Punkte an Vermehrungsrate erlaubt, wird dies davor geprüft. Sind die eingegebenen Punkte mehr als zugelassen, wird ihr Wert geclippt, um illegale Züge früh zu vermeiden.

3.3.2. Step()

Zu Beginn wird die Action geprüft, ob diese dem erforderlichen Format entspricht. Als nächstes werden die Produktion und der Spezialfall umgerechnet.

In der Variable **`used_points`** werden die vergebenen Aktionspunkte summiert. Fallen sie unter null oder über die berechnete Gesamtpunktzahl für die konkrete Episode, wird der Zug als invalid zurückgewiesen.

Weiterhin werden die eingegebenen Actions der Observation zugerechnet. Überschreitet deren Summe die zulässige Range, wird wieder von einem invaliden Zug gesprochen und das Spiel wird nicht weitergeführt:

```
214    if self.V[self.VALID_TURN]:
215        for i in range(5):
216            if (self.V[i] + action[i]) not in range (self.Vmin[i],
                                                        self.Vmax[i] + 1):
217                self.V[self.VALID_TURN] = 0
218                break
```

Codeschnipsel aus der Klasse OekoEnv

Ist **`V[VALID_TURN] = True`**, werden die Actions zu den Observation zugerechnet. Die verfügbaren Aktionspunkte und Rundenzahl werden aktualisiert. Falls sich ein Bereich außerhalb des Ranges befindet, wird dessen Wert an der oberen oder unteren Grenze geclippt und somit endet das Spiel. Die Punkte und Rundenanzahl werden auch geprüft.

Weiterhin werden die Aktionspunkte berechnet und nochmal geprüft, ob sie die erlaubten Werte überschreiten:

```
245     # Points for next round
246     if done:
247         self.V[self.POINTS] = 0
248     else:
249         boxA  = gb.get_boxA  (self.V[self.BEVOELKERUNG])
250         boxB  = gb.get_boxB  (self.V[self.POLITIK])
251         boxC  = gb.get_boxC  (self.V[self.PRODUKTION])
252         boxV  = gb.get_boxV  (self.V[self.PRODUKTION])
253         boxD  = gb.get_boxD  (self.V[self.LEBENSQUALITAET])
254 
255         self.V[self.POINTS] += boxA * boxV
256         self.V[self.POINTS] += boxB
257         self.V[self.POINTS] += boxC
258         self.V[self.POINTS] += boxD
259 
260     if  self.V[self.POINTS] > 36:
261         self.V[self.POINTS] = 36
262         done = True
263         done_info = 'Maximale Anzahl von Aktionspunkten erreicht'
```

Codeschnipsel aus der Klasse OekoEnv

Ist das Spiel beendet, wird die Bilanz aus den erforderlichen Properties berechnet.

Dabei ist zwischen Bilanz und Reward zu unterscheiden. Erstere gibt Hinweise bei der Bewertung der angewandten Strategie und ist immer am Ende des Spiels zu berechnen. Im Gegensatz dazu ist der Reward für den konkreten Schritt ausschlaggebend, da sie dem Agenten später im Lernprozess über die getroffenen Aktionen eine Richtung geben wird.

Bei der Berechnung der Bilanz, gibt es schließlich zwei Szenarien:

1) Die Politik oder Lebensqualität (Feld D) ist out-of-range geraten. Ihr Wert wird dann geclippt und damit die Bilanz berechnet.
2) Ein anderer Bereich ist out-of-range geraten und zur Berechnung der Bilanz werden die Werte von Politik und Lebensqualität (Feld D) aus der vorherigen Runde benutzt.

Letztendlich wird der Spieler härter bestraft, wenn die Politik oder Lebensqualität den Ranges verlassen.

3.3.3. Reset()

Die reset()-Funktion setzt die initiale Observation gemäß Tabelle 2.

3.4. Testen des Environments

Der korrekte Ablauf des Environments wird durch einige Testfälle geprüft, die in der Datei **`test.py`** eingeführt werden. Zunächst wird das Spiel für zehn volle Runden gespielt, danach werden die eingebauten Exceptions **`Value`**- und **`AssertionError`** getestet und abschließend der Spezialfall „Aufklärung > Vermehrungsrate“ in verschiedenen Szenarien durchgeführt. Zusätzlich werden die Fälle behandelt, in denen keine Aktion vorgenommen wird, also keine Aktionspunkte werden verteilt.

Dafür wird zunächst eine Liste namens **`instructions`** angelegt, in der die auszuführenden Befehle als Elemente eingegeben werden. Diese sind entsprechend **`step`**, **`reset`** und **`check_v`**. In einer Schleife wird geprüft, welcher Befehl eingegeben wurde und die passende Funktion wird aufgerufen.

Im Fall von **`reset`** wird der Zustand des Environments neu gesetzt und der Titel des Tests wird ausgegeben. Falls eine andere Startobservation gewollt ist, kann diese auch zu der Liste der Befehle hinzugefügt werden. Diese wird dann an das Environment übergeben und gesetzt. Wichtig dabei ist, dass es vom Typ **`np.array`** ist.

Als nächstes wird die **`step()`**-Funktion in einem Try-Block aufgerufen. Dabei werden die Felder von Produktion und dem Spezialfall zusätzlich berechnet, damit sie im zulässigen Range für die Umgebung sind. Als Informationen werden die berechnete Observation, der Reward, der Wert der Variable **`done`** und die allgemeine Informationssequenz in Form eines Dictionary angezeigt. Ist die jeweilige Episode zum Ende gekommen, wird nur ausgegeben, dass die **`step()`**-Funktion nicht mehr ausgeführt werden kann. Dazu wird noch auf das Vorkommen von eventuellen Exceptions geachtet. Ist eine solche aufgetreten, wird die eingegebene Action ignoriert und mit der nächsten weitergeführt. **`ValueError`** fängt die Fälle ab, wo einem Lebensreich so viele Punkte zugeteilt werden, dass dieser über sein Range hinausgeht. **`AssertionError`** stellt sicher, dass jeder eingegebene Wert in Action oder Observation Space valid ist und die Grenzwerte einhält.

`Check_v` erlaubt es die berechnete Observation aus dem Environment mit dem richtigen Ergebnis zu vergleichen, das in die Liste mit Befehlen für jede Runde nach **`step`** und **`reset`** eingegeben wird. Mit dem Argumenten **`fatal`** kann ähnlich wie in einem Debug-Modus festgelegt werden, dass der Rest der Testfälle zu überspringen ist. Bei Eingabe von **`nonfatal`** wird der Fehler angezeigt, aber das Skript wird bis zum Ende ausgeführt.

4. Grafische Oberfläche mit PyQt5

PyQt [12] ist ein plattformübergreifendes Framework, das die Entwicklung von flexiblen und performanten grafischen Oberflächen ermöglicht. Es stellt einen Satz von Python Bindings für das ursprüngliche Framework Qt dar und verfügt somit über vollen Zugang zu den Qt Libraries.

Qt ist in C++ geschrieben und umfasst eine umfangreiche Reihe an Klassenbibliotheken, Development- und Design-Tools. Qt zeichnet sich dadurch aus, dass es die Klassen in modularer Struktur bereitstellt, was das Produzieren von lesbarem und wiederverwendbarem Code mit geringem Platzbedarf erlaubt. Realisierbar sind sowohl Desktop- als auch mobile und Embedded-Anwendungen. Außerdem werden neben dem Visualisieren verschiedener Arten von multimedialen Daten auch SQL Datenbanken, OpenGL, XML und ähnliche Features unterstützt. [2]

Ein anderes GUI-Framework ist Gradio [15], womit simple Demos von ML-Modellen in Python generiert werden. Dafür stehen einige vorgefertigte UI-Komponenten zur Verfügung, die an verschiedene Szenarien angepasst werden können. Die Benutzeroberfläche wird lokal gehostet, wobei es auch möglich ist, sie mit anderen zu teilen. Die Dokumentation von Gradio ist klar und sparsam. Gradio wird mit dem Befehl **`pip install gradio`** installiert und kann direkt eingesetzt werden, indem der Code für die grafische Oberfläche als Python-Skript ausgeführt wird.

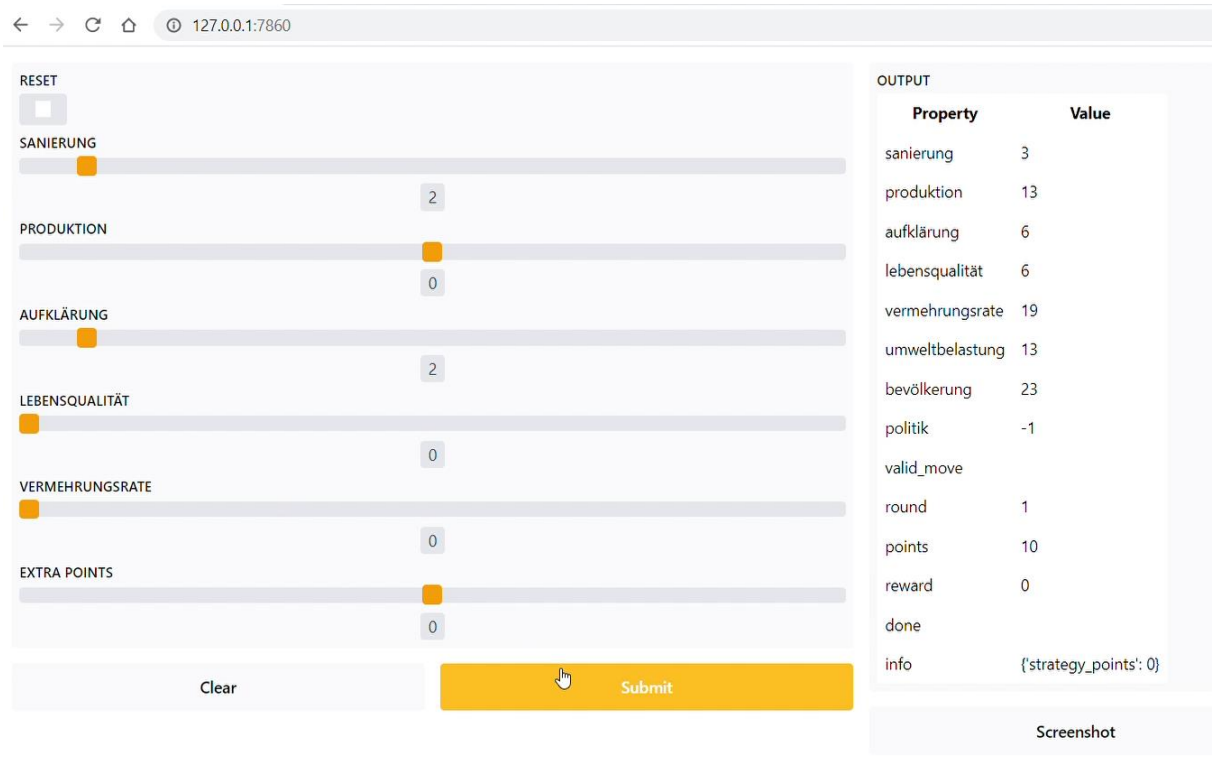

Abbildung 3: Screenshot aus 01-GUI-Gradio.mp4

Abbildung 3 veranschaulicht die erste Nutzeroberfläche für Ökolopoly. Das Design ist schlicht und die Sliders ermöglichen die direkte Auseinandersetzung mit dem Spiel. Im Code wird eine Instanz des Environments angelegt. Nach Drücken von Submit werden die Werte der Bereiche berechnet und in der Tabelle rechts aktualisiert. Ist die Reset-Checkbox angewählt, werden beim nächsten Drücken von Submit alle Bereiche auf ihre Anfangswerte zurückgesetzt. Anschließend werden die Sliders mit dem Button Clear zurückgesetzt.

Der im Anhang gezeigte Code der Gradio-UI ist für eine frühere Version des Environments erstellt worden und funktioniert für die aktuellste Umgebung nicht. Bemerkenswert ist, dass mit wenigen Codezeilen eine gut aussehende und funktionierende Nutzeroberfläche mit minimalem Zeitaufwand erstellt werden kann. Als nachteilig haben sich die im Vergleich zu PyQt5 eher langen Wartezeiten zwischen Komplieren des Codes und dem Erscheinen auf dem Bildschirm erwiesen, was das Testen erschweren kann.

In Gegensatz zu Gradio verfügt PyQt5 über eine signifikante Vielfalt an Klassen und Attributen für unterschiedliche UI-Komponenten, was die Perspektive für Weiterentwicklung unterstützt. Zudem kann das Aussehen der GUI leicht durch Einsatz von Templates eingestellt werden, wie es auch für dieses Projekt getan wurde. Weiterhin kann die Benutzeroberfläche durch kleine Änderungen benutzerfreundlicher werden wie zum Beispiel mittels eingebauten Constraints für die Sliders, die den Nutzer begrenzen, mehr Punkte als vorhanden für die bestimmte Runde zu verteilen oder einer Tabelle mit dynamisch erscheinenden Einträgen, was eine Übersicht mit den Ergebnissen schafft. Schließlich muss eine gute GUI informative Rückmeldungen über Status im Spiel vermitteln, was mit PyQt5 möglich ist.

Aus den obengenannten Vorteilen und den Erweiterungsmöglichkeiten hat sich PyQt5 als ein geeigneter Ansatz für das Realisieren der grafischen Oberfläche des Ökolopoly Environments angeboten.

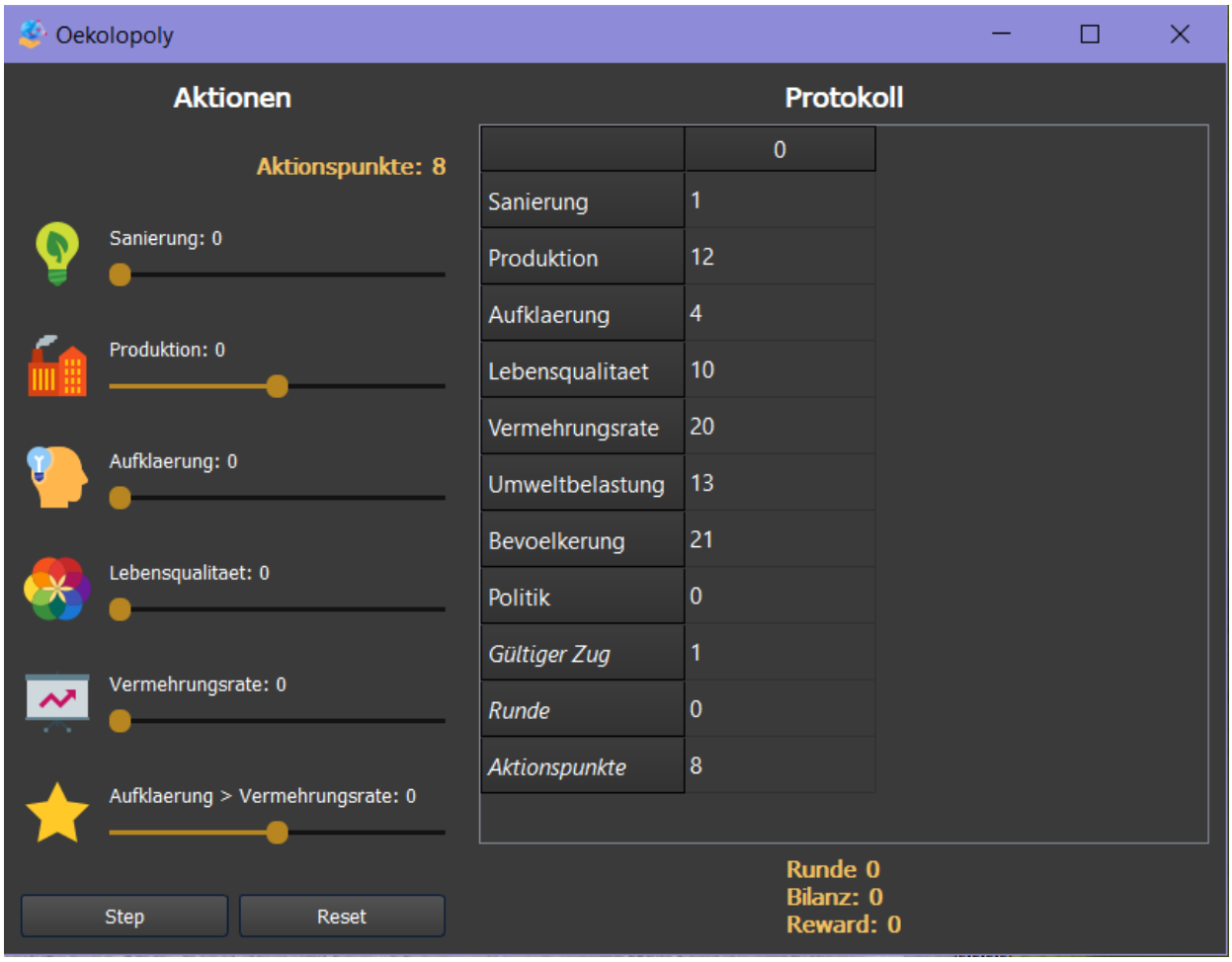

Abbildung 4: *Screenshot aus 02-GUI-PyQt5.mp4*

In Abbildung 3 wird die fertige GUI gezeigt. Informationen über ihre Bestandteile und genaue Umsetzung sind im nächsten Kapitel ausführlich behandelt.

PyQt wird durch Eingabe des Befehls **`pip install PyQt5`** in den Command Prompt installiert. Der Code kann ähnlich wie bei Gradio als Python-Skript ausgeführt werden.

Im folgenden werden alle importierten Module in Abschnitten aufgeführt und in Zusammenhang mit dem Quellcode erläutert.

4.1. QtCore

`QtCore` enthält wichtige Klassen wie Event Loop und den Signalmechanismus von Qt, der bei der Funktionaliät der Buttons zum Einsatz kommt. Das Modul ist für die Kommunikation zwischen den unterschiedlichen Widgets und den Umgang mit Threads zuständig.

4.2. QtGui

`QtGui` unterstützt grafische Elemente wie **`QPixmap`** und **`QIcon`**, womit Bilder in der Anwendung eingebunden werden können.

4.2.1. QFont

Diese Klasse ist auch Teil des Moduls **`QtGui`**. Durch sie kann die Schriftart weiter spezifiziert werden, indem Attribute wie **`Weight`** und **`Point`** gesetzt werden, die jeweils für die Schriftstärke und -größe stehen.

4.3. QtWidget

Widgets sind UI-Komponente, mit denen der Nutzer meistens interagiert. In diesem Projekt werden alle eingesetzten Widgets von dem **`QtWidget`** Modul importiert.

Das Aussehen der einzelnen Widgets kann durch die sogenannten StyleSheets angepasst werden. Das StyleSheet ist üblicherweise eine separate Datei, in der die Attribute der UI-Elemente mit den gewünschten Werten gesetzt werden. Für dieses Projekt erfüllt die Datei **`Combinear.qss`** diese Rolle, die ein vorgefertigtes Template darstellt. [10]

In Folgenden werden alle Widgets mit ihren Hauptfunktionalitäten in Bezug auf die für das Projekt erstellte User Interface vorgestellt.

4.3.1. QApplication

Eine Anwendung wird immer durch **`QApplication`** initialisiert und mit **`qapp.exec()`** ausgeführt. Zwischen **`QApplication()`** und **`qapp.exec_()`** soll ein Fenster erzeugt werden, das vom Typ **`QWindow`** ist. Attribute wie Titel, Icon und Größe können gesetzt werden und alles wird mittels **`window.show()`** angezeigt.

`QApplication` wird immer ein Mal für die konkrete GUI-Anwendung instanziert, unabhängig von der Anzahl der zurzeit gestarteten Fenster. Diese Klasse verwaltet die Hauptereignisschleife, das Initialisieren von Widgets und die Events die davon abgerufen werden.

4.3.2. QPushButton

Die Step- und Resetbuttons werden mit **`QPushButton`** erzeugt. Eine Funktion wird an einen Button gebunden, indem sie in Form eines Lambda-Ausdrucks als Eingabeparameter der Methode **`connect()`** übergeben wird. Die anonymen Funktionen sind in diesem Fall eine geeignete Lösung, da sie nicht zusätzlich definiert werden sollen.

Wird ein Button durch beliebige Interaktion (Mausklick oder Tastendruck) aktiviert, wird das Signal **`clicked()`** ausgesendet. Dieses ruft die **`connect()`**-Funktion und führt die gewünschte Aktion aus.

Durch **`isEnabled(False)`** wird der Button deaktiviert und dadurch ausgegraut. Das passiert, wenn das Spiel zum Ende gekommen ist bzw. ein Bereich sich außerhalb des zulässigen Ranges befindet oder ein ungültiger Zug zustande gekommen ist.

4.3.3. QSlider

Slider in PyQt sind per Default vertikal ausgerichtet, was bei dem Instanzieren der Klasse durch Eingabe von **`Qt.Horizontal`** in den Konstruktor geändert werden kann. Die Grenzwerte können durch **`setMinimum()`** und **`setMaximum()`** gesetzt werden. Weiterhin es ist möglich Funktionen mittels **`valueChanged.connect()`** an Slider zu binden.

4.3.4. QLabel

Dieses Widget stellt ein einfaches Textstück dar und dient zum Anzeigen von relevanten Anwendungsinformationen. Daher ist keine Interaktion mit dem Widget möglich. Text kann entweder beim Initialisieren als Eingabeparameter im String Format übergeben werden oder durch die Funktion **`setText()`** gesetzt werden.

Die Position des Textes kann durch **`setAlignment()`** vertikal und/oder horizontal bestimmt werden. Als Input bekommt die Funktion einen Flag wie **`Qt.AlignRight`** oder **`Qt.AlignBottom`**, der den Text jeweils rechts oder im unteren Teil des Fensters annordnet.

Mit **`setStyleSheet()`** kann zusätzlich die Farbe für die konkrete Instanz unabhängig vom benutzten StyleSheet geändert werden.

`SetFont()` funktioniert ähnlich und ermöglicht weitere Optionen zur Änderung der Schriftart. Es nimmt als Parameter Variablen vom Typ QFont an.

Mit Objekten vom Typ **`QLabel`** können Bilder angezeigt werden, indem der Bildpfad als QPixmap Objekt in **`setPixmap()`** übergeben wird wie es im folgenden Codeschnipsel zu sehen ist:

```
26  class ActionSlider:
27       def __init__(self, options, env, sliders, points_label):
...           ...
42          self.icon = QLabel ()
43          self.icon.setPixmap (QPixmap (options['icon']))
...           ...
156 def main ():
        ...


171     action_options = [
172         {...,       "icon": "res/imgs/s.png",  ...},
...           ...
178     ]
```

In der Liste **`action_options`** werden die Attribute, die für das Initialisieren der Objekte der Klasse **`ActionSlider`** benötigt werden, mit Werten versehen und später durch eine Schleife an den Konstruktor übergeben.

4.3.5. QTableWidget und QTableWidgetItem

Mit QTableWidget können Tabellen erzeugt werden, deren Einträge vom Typ **`QTableWidgetItem`** sind. Für die jeweilige Tabellenzelle können Flags wie **`ItemsIsSelectable`** und **`ItemIsEnabled`** eingegeben werden, die es ermöglichen, sie zu markieren. Die Größe der Tabelle kann zu einem späteren Punkt nach dem Instanziieren spezifiziert werden. Mit **`setHorizontalHeaderItem()`** und **`setVerticalHeaderItem()`**werden die Spalten- und Zeilenüberschriften eingegeben.

Die Spalten stellen die aktuelle Observation dar und die einzelnen Zeilennamen werden von der Liste **`table_headers`** abgelesen. Im Gegensatz dazu stehen die Spaltenüberschriften für die Rundeanzahl und werden dynamisch gesetzt.

Die Tabelle wird in der **`reset()`**-Funktion geleert und zeigt die initiale Observation an. Beim Aufruf von **`step()`**wird eine neue Spalte mit dem neuberechneten Environmentszustand generiert und Zeile für Zeile als QTableWidgetItem hinzugefügt.

4.4. Layout Management

Mit Layouts kann die Position der einzelnen Widgets auf dem Bildschirm verwaltet werden. Für das vorhandene Projekt ist von folgenden drei Gebrauch gemacht:

- QHBoxLayout – Horizontale Anordnung
- QVBoxLayout – Vertikale Anordnung
- QGridLayout – Anordnung in einem Grid mit Indizes

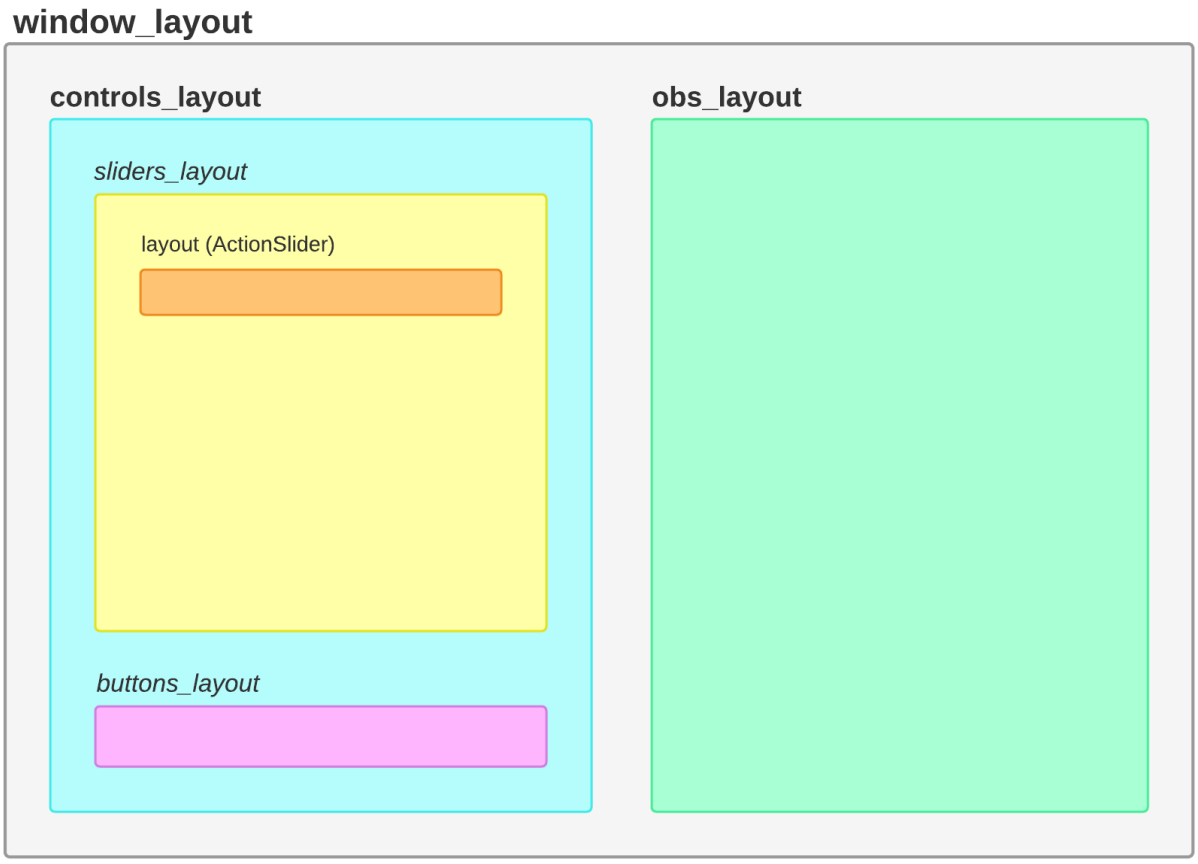

Abbildung 5: Schematische Darstellung der eingesetzten Layouts

Das Hauptfenster wird durch **`QMainWindow()`** instanziert. Diese Klasse verfügt über vorgefertigte Features wie Statusleiste, Menüs und Toolbars. In der vorhandenen Oberfläche werden nur die Attribute WindowTitle und WindowIcon gesetzt, die den Namen und das Bild im oberen linken Teil des Fensters bestimmen. Mit **`setCentralWidget()`** wird vorgegeben, dass das Layout, das als Eingabeparameter – in diesem Fall **`window_layout`** vom Typ **`QHBoxLayout`** – übergeben wird, den Platz des ganzen Fensters einnimmt. Dieses Layout schließt in sich alle anderen ein und wird als Top-Level Widget bezeichnet.

Die Funktionen **`setSpacing()`** und **`setContentsMargins()`** spezifizieren jeweils den Abstand zwischen den einzelnen Komponenten und dem Raum außerhalb des Layouts.

Spezifisch bei dem Erzeugen eines Layouts ist dessen Übergabe an ein Dummy Widget mittels der Funktion **`setLayout()`**. Auf diese Weise können weitere UI-Komponenten und Layouts zum konkreten Layout hinzugefügt werden

Alle Sliders der Klasse ActionsSlider und die Buttons werden in **`controls_layout`** gepackt. Der Grund für verschachtelte Layouts ergibt sich daraus, dass manche Elemente wie die Labels Titel und Aktionspunkte sich auf diese Weise leichter anordnen lassen. Die logische Trennung

zwischen den einzelnen Komponenten ist ein weiterer Grund dafür – alle Slider gehören zu **`sliders_layout`** und die Buttons zu **`button_layout`**. Hierdurch können noch Elemente hinzugefügt werden, die diese Logik einhalten und der Code bleibt einheitlich.

Der individuelle Slider besteht aus einem Icon, einem Titel und dem Slider selbst. Zusätzlich werden noch die Werte angezeigt, die bei Bewegung der Slider entstehen.

Der rechte Teil des Fensters stellt die Ergebnisse nach Ausführen der **`step()`**-Funktion in Form einer Tabelle dar. Neben dieser Tabelle enthält das obs_layout noch zwei Labels - den Titel „Protokoll“ und einen Status, der die Anzahl der Runden, Bilanz und Reward anzeigt. Ist das Spiel beendet oder ist ein Fehler aufgetreten, wird dies auch als Information im Status ausgegeben.

4.5. Die Klasse ActionSlider

Die Slider in der grafischen Oberfläche ermöglichen die Vergabe von Aktionspunkten. Beim Initialisieren wird eine Liste übergeben, die den Namen, Icon, Grenzwerten und Defaultwert eines konkreten Bereichs enthält. Weitere Inputparameter sind eine Instanz des Environments, eine Liste mit Sliders und ein Label, das für die Anzahl der Aktionspunkte steht.

In der Funktion **`change()`** werden die verteilten Aktionspunkte im Label des Sliders in Abhängigkeit der vorhandenen Gesamtpunkte behandelt. Hilfefunktionen dabei sind **`get_available_points()`** und **`update_points_label()`**.

Erstere holt die eingestellten Werte von jedem Slider, bildet deren Summe und zieht diese von der gesamten Aktionspunktenanzahl ab, um die verfügbaren Punkte zurückzugeben. Zweitere setzt das übergeordnete Label der Gesamtpunktanzahl auf den zurückgegebenen Wert. Sind die verfügbaren Punkte unter null gelangt, kann der Slider nicht mehr bewegt werden. Das ist ein Constraint, das den Nutzer einschränkt. Hierdurch kann der Nutzer keine illegalen Züge bezüglich des Zuteilens von Punkten durchführen.

4.6. Allgemeiner Spielablauf

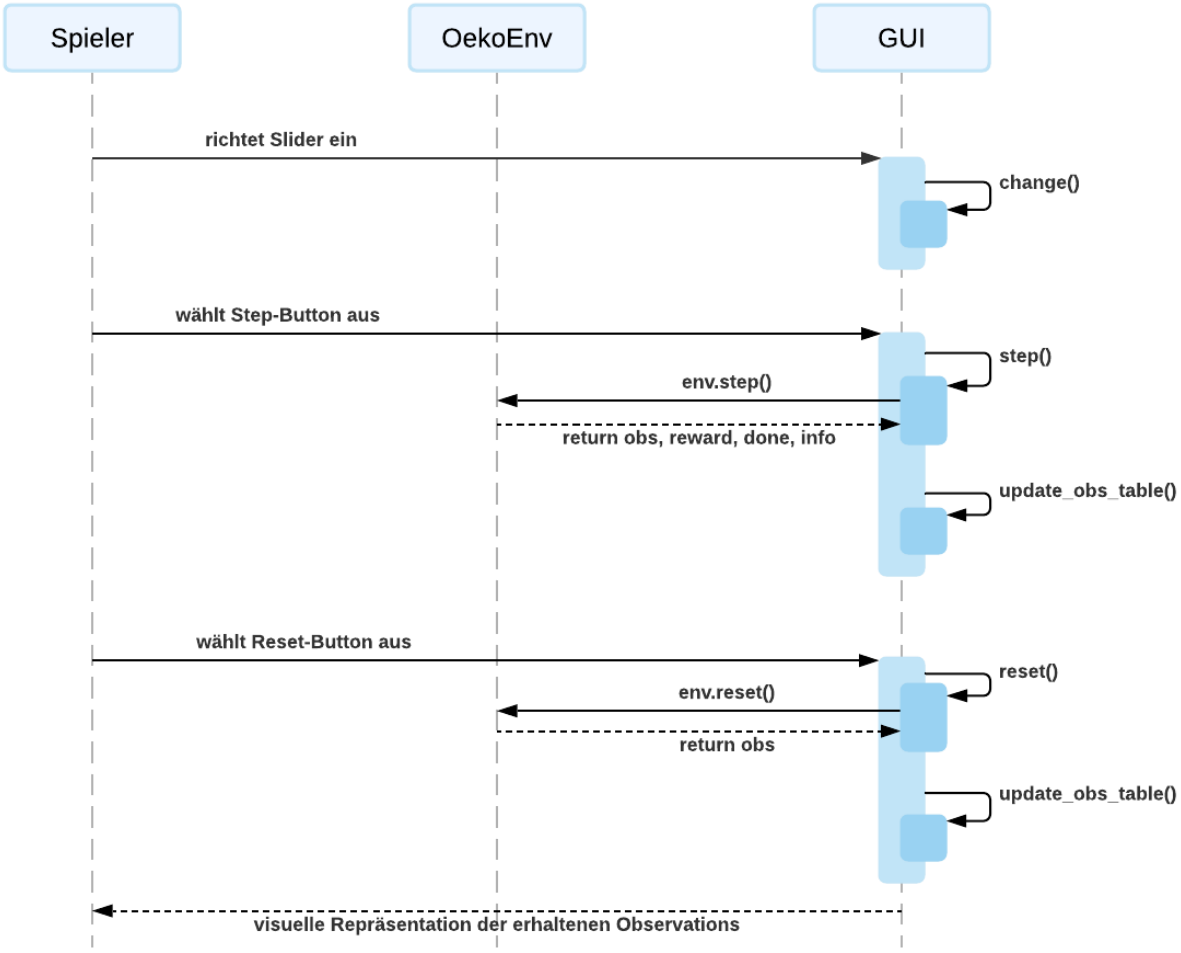

Abbildung 6: Allgemeiner Ablauf

In Abbildung 4 wird der allgemeine Spielablauf veranschaulicht. Der Spieler verteilt Aktionspunkte, indem er die Slider mit den gewünschten Werten einstellt. An dieser Stelle wird die change()-Funktion ausgeführt, die die beiden Labels der Aktionspunkte aktualisiert. Nun wählt der Spieler den Step-Button aus und die **`step()`**-Funktion der GUI-Klasse wird aufgerufen. Um die erforderlichen Rückgabewerte zu erhalten, ruft sie die **`step()`**-Funktion des Environments auf. Anschließend wird die Tabelle um eine Spalte erweitert. Ähnlich ist der Ablauf beim Aufruf von **`reset()`**. Der Unterschied dabei ist, dass die Einträge in der Tabelle gelöscht werden und nur die initiale Observation angezeigt wird.

In der GUI wird neben der Runde, die zum Abbruch führte, auch angezeigt, warum das Spiel abgebrochen wurde – ob wegen eines Bereiches/Anzahl von Runden oder Punkte. Ist ein **`ValueError`** Exception abgefangen, wird die problematische Runde nicht angezeigt, da sie nicht weiterberechnet wird. Wenn ein Wheel-out-of-range passiert, werden die restlichen Werte

im Array **V** nicht berechnet, sondern behalten ihre Werte aus der vorherigen Runde. Derjenige Bereich, der seinen Range verlässt, wird an seinem Minimum/Maximum geclippt:

```
230     # Clip values if not in range
231     for i in range(8):
232         if  self.V[i] not in range (self.Vmin[i],  self.Vmax[i] + 1):
233             self.V[i] = max( self.Vmin[i], min( self.Vmax[i],
                                                    self.V[i]))
234             done = True
```

Codeschnipsel aus der Klasse OekoEnv

4.7. Policy

Als eine gute Strategie, um mindestens zehn Runden zu überleben, hat sich das Investieren von Aktionspunkten in Aufklärung herausgestellt. Auf diese Weise erhält man schnell Kontrolle über die Vermehrungsrate und die Lebensqualität bleibt auf einem guten Niveau. Als nächstes kann man sich auf Produktion konzentrieren und dort den Wert in guten Grenzen halten (zwischen 10 und 15). Am Ende kann man seine Punkte in Sanierung investieren. Da sich die Anzahl der Aktionspunkte im Verlauf des Spiels rasant erhöht, kann man diese vebrauchen, indem man die Vermeherungsrate erst erhöht und dann schnellstmöglich wieder verringert. Somit sind alle Bereiche in ihrem bestem Zustand und eine Bilanzzahl von 20 wird erreicht.

5. Fazit und Ausblick

Im Verlauf des Praxisprojekts wurde viel Erfahrung mit dem Erstellen eines eigenen Environments in OpenAI Gym gesammelt. Als wesentlicher Entscheidungspunkt bei der Umsetzung hat sich das Festlegen der Action- und Observation-Spaces erwiesen, da diese Spaces die Basis der Umgebung bilden und weitgehende Auswirkungen auf ihre Funktionsweise haben. Da Gym keine Space Typen anbietet, die negative Zahlen behandeln, müssen die übergebenen Action- und Observation-Spaces intern stets umgerechnet werden. Weiterhin hat die Umsetzung der User Interface eine gute Gelegenheit geboten, Gradio als GUI-Framework auszuprobieren und sich mit PyQt zu befassen und mehr Wissen über die von ihm bereitgestellten Klassen zu erwerben.

Spiele wie Ökolopoly, die über Milliarden von Aktionen und Zuständen verfügen, bieten eine gute Perspektive für Erforschen des Verhaltens von Agenten. Dadurch dient dieses Projekt als eine geeignete Grundlage für das Testen diverser RL-Algorithmen auf verschiedenen Repräsentationen des Aktions- und Zustandsraumes des Spiels. Diese können dann qualitativ analysiert und verglichen werden. Um dies zu ermöglichen, ist es das Ziel der anschließend geplanten Bachelorarbeit, die gigantischen Aktions- und Zustandsräume durch Übersetzung in wenige Kategorien (z.B. niedrig/mittel/hoch) zu verkleinern. Das wird durch die Erstellung

passender Environment Wrappers erreicht, die auch das Auftreten von illegalen Zügen korrigieren. Zusätzlich ist geplant, unterschiedliche Rewardsysteme auszuprobieren, um festzustellen, welches zur größten Leistung des Agenten führt. Adäquate Belohnungen können den Agenten immens helfen, seine Policy für den konkreten Zustand zu verbessern und zu verfeinern. Anschließend kann die grafische Oberfläche weiterentwickelt werden, um Grafiken über das Verhalten der einzelnen Agenten darzustellen.
Zu diesem Zeitpunkt ist nicht geplant, die fehlenden Funktionalitäten – Ziehen von Ereigniskarten und Variante für eine Extremstrategie – zum Environment hinzuzufügen. Dies würde den bereits gigantischen Aktions- und Zustandsraum des Spiels vergrößern und deren Erforschen voraussichtlich erschweren. Abschließend liegt der Schwerpunkt bei der bevorstehenden Bachelorarbeit auf dem Untersuchen verschiedener Repräsentationen von Aktions- und Zustandsraum auf Basis des vorhandenen Environments.

Literaturverzeichnis

[1] Frederic Vester, „Die Kunst, vernetzt zu denken“, 1. Aufl., Pantheon Verlag, 2019.

[2] Joshua Willman, „Modern PyQt: Create GUI Applications for Project Management, Computer Vision, and Data Analysis”, APRESS Verlag, 2021

[3] Jörg Frochte, „Maschinelles Lernen – Grundlagen und Algorithmen in Python“, 2. Aufl., Hanser Verlag, 2019.

[4] Frederic Vester, „Der blaue Planet in der Krise“, Gewerkschaftliche Monatshefte, 12-88, 713

[5] Jan-Philipp Küppers, „Komplexe Konfliktlagen systemisch & partizipativ bearbeiten“, eNewsletter Netzwerk Bürgerbeteiligung 02/2018 vom 13.07.2018, https://www.netzwerk-buergerbeteiligung.de/fileadmin/Inhalte/PDF-Dokumente/newsletter_beitraege/2_2018/nbb_beitrag_kueppers_180713.pdf, 03.07.2021

[6] OpenAI Gym: „Getting Started with Gym“, https://gym.openai.com/docs/, 19.06.2021

[7] Martin Fitzpatrick: „PyQt Example applications“, https://www.mfitzp.com/pyqt-examples/, Zugriff am 03.07.2021

[8] Guru99: „PyQt5 Tutorial: Design GUI using PyQt in Python with Examples“, https://www.guru99.com/pyqt-tutorial.html#6, Zugriff am 03.07.2021

[9] „PyQt5 - QTableWidget“, https://www.geeksforgeeks.org/pyqt5-qtablewidget/, Zugriff am 03.07.2021

[10] DEV SEC Studio: „Templates“, https://qss-stock.devsecstudio.com/templates.php, Zugriff am 03.07.2021

[11] Qt Company: „Widgets Classes“, https://doc.qt.io/qt-5/widget-classes.html, Zugriff am 03.07.2021

[12] Riverbank Computing: „What is PyQt?”, https://riverbankcomputing.com/software/pyqt/intro, Zugriff am 03.07.2021

[13] OpenAI Gym: „open/gym/core.py“, https://github.com/openai/gym/blob/master/gym/core.py, Zugriff am 03.07.2021

[14] Frederic Vester: Neuland des Denkens. Vom technokratischen zum kybernetischen Zeitalter, Stuttgart 1980, https://www.academia.edu/30931729/Literaturauszug_aus_Frederic_Vester_Neuland_des_Denkens._Vom_technokratischen_zum_kybernetischen_Zeitalter, s. 15, Zugriff am 03.07.2021

[15] Gradio: https://gradio.app/, Zugriff am 03.07.2021

Anhang

- Git-Repo: https://github.com/cherrisimo/oekolopoly
- 01-GUI-Gradio: https://th-koeln.sciebo.de/s/parkPnEelK3GaPS
- 02-GUI-PyQt5: https://th-koeln.sciebo.de/s/3z4uK3mKmYJqS1k
- Code von Gradio-UI:

```
import gradio as gr
import gym

env = gym.make('oekolopoly:Oekolopoly-v0')
init_obs = env.reset()

def get_input(reset, sanierung, produktion, aufklärung, lebensqualität, vermehrungsrate, extra_points):
  if reset:
    obs = env.reset()

    r = {
      'sanierung':       int(env.V[0]),
      'produktion':      int(env.V[1]),
      'aufklärung':      int(env.V[2]),
      'lebensqualität':  int(env.V[3]),
      'vermehrungsrate': int(env.V[4]),
      'umweltbelastung': int(env.V[5]),
      'bevölkerung':     int(env.V[6]),
      'politik':         int(env.V[7]),
      'valid_move':     bool(env.V[8]),
      'round':           int(env.V[9]),
      'points':          int(env.V[10]),
    }
  else:
    produktion -= env.Amin[env.PRODUKTION]
    extra_points -= env.Amin[5]

    obs,reward,done,info = env.step ((sanierung, produktion, aufklärung, lebensqualität, vermehrungsrate, extra_points))

    r = {
      'sanierung':       int(env.V[0]),
      'produktion':      int(env.V[1]),
      'aufklärung':      int(env.V[2]),
      'lebensqualität':  int(env.V[3]),
      'vermehrungsrate': int(env.V[4]),
```

```
36        'umweltbelastung': int(env.V[5]),
37        'bevölkerung':     int(env.V[6]),
38        'politik':         int(env.V[7]),
39        'valid_move':     bool(env.V[8]),
40        'round':           int(env.V[9]),
41        'points':          int(env.V[10]),
42
43        'reward':      reward,
44        'done':        done,
45        'info':        str(info),
46      }
47
48    return r
49
50  iface = gr.Interface(
51    fn=get_input,
52    inputs=[
53      "checkbox",
54      gr.inputs.Slider(int(env.Amin[0]), int(env.Amax[0]), step=1),
55      gr.inputs.Slider(int(env.Amin[1]), int(env.Amax[1]), step=1,
    default=0),
56      gr.inputs.Slider(int(env.Amin[2]), int(env.Amax[2]), step=1),
57      gr.inputs.Slider(int(env.Amin[3]), int(env.Amax[3]), step=1),
58      gr.inputs.Slider(int(env.Amin[4]), int(env.Amax[4]), step=1),
59      gr.inputs.Slider(int(env.Amin[5]), int(env.Amax[5]), step=1,
    default=0),
60    ],
61    outputs=["key_values"]
62  )
63
64  iface.launch()
```