LUX AI Challenge

Gather the most resources and survive the night!

Dokumentation zur Projektarbeit

Von

Pol Hansen

Betreut durch

Prof. Dr. rer. nat. Patrick Baier

Worum geht es im Projekt?

Im Projekt geht es darum, eine Künstliche Intelligenz zu entwickelt, welche in einer Umgebung mit Städten und Ressourcen gegen eine andere Künstliche Intelligenz antreten muss. Am Ende des Spiels gewinnt der Spieler, der am meisten Städte hat. Dies wurde zuerst eigenständig mit einem Rule-Based-Agent also einem Regel Basierten Agenten, welcher nur im entferntesten Sinne mit einer Künstlichen Intelligenz zu tun hat, implementiert. Anschließend wurde ein Deep-Q-Learning Netzwerk analysiert.

Beschreibung der Lux AI Challenge

In der LUX AI Challenge kontrolliert man Einheiten, sogenannte Units, welche Ressourcen sammeln müssen, um Häuser sogenannte CityTiles bauen zu können, welche wiederum mehr Units „erzeugen“ können. Um Häuser bauen zu können, brauchen die Units ein volles Inventar, was 100 Ressourcen egal welcher Art entspricht. Um ein CityTile bauen zu können, werden alle 100 Ressourcen verwendet. Im Spiel gibt es 3 verschiedene Ressource Typen mit verschiedenen Werten allerdings braucht man länger, um höherwertige Ressourcen einzusammeln. Die Werte sind wie folgt:

Ressourcentyp	Wert in Fuel	Anzahl gesammelt pro Runde
Holz	1	20
Kohle	10	5
Uranium	40	2

Somit sieht man, dass Kohle einen höheren Wert hat als Holz und Uranium einen noch höheren Wert hat als Kohle. Allerdings muss man das Sammeln von Kohle bzw. Uranium „erlernen“. Dies tut man, indem die CityTiles Research betrieben. Die Units sowie auch die CityTiles erhalten nach jeder Aktion einen gewissen Cooldown den sie abwarten müssen, bevor sie neue Aktionen ausführen können. Auch ist pro Runde nur eine Aktion je CityTile und je Unit möglich. Das Ziel der Challenge ist es, am Ende des Spiels, welches über 360 Runden geht, die meisten CityTiles zu besitzen.

Setup

Dieses Projekt wurde mit Anaconda, PyCharm und Jupyter Notebook entwickelt. Um die virtuelle Umgebung zu erstellen, kann einfach folgender Befehl im Projektordner ausgeführt werden: `conda create --name LUXAICHALLENGE --file requirements.txt` , wobie die Python Version 3.7.11 verwendet wurde. Der Regel Basierte Agent wurde in PyCharm entwickelt und im Jupyter Notebook ausgeführt.

Inkrementelle Entwicklung des Regel Basierten Agenten

Als Ausgangmodell wurde das Python Starterkit der Lux AI Challenge verwendet, den sogenannten Simple Agent. (https://github.com/Lux-AI-Challenge/Lux-Design-2021/tree/master/kits/python). Hierbei wird über alle Spielfeldzellen iteriert, wobei alle Felder die Ressourcen enthalten, in eine Liste gespeichert werden. Anschließend wird über alle Units iteriert. Hierbei wird getestet, ob die Unit diese Runde eine Aktion ausführen kann. Wenn die Unit dies kann, so wird überprüft, ob die Unit noch Platz im Inventar hat. Ist dies der Fall, so geht die Unit in Richtung der nächsten (erforschten) Ressource. Wenn die Unit keinen Platz im Inventar mehr hat, geht sie in Richtung des nächsten CityTiles und versorgt dieses somit mit Fuel.

Erste Verbesserung: Neue CityTiles bauen und Units erzeugen.

Um den Simple Agent zu besiegen, braucht man nur ein weiteres CityTile zu bauen, und dieses bis zum Ende des Spiels zu erhalten. Die Units sollen dann die Citys auffüllen, bis diese genug Fuel für die nächste Nacht hat. Wenn jede Stadt genug Fuel für eine Nacht hat, kann ein neues CityTile gebaut werden. Alle neuen CityTiles sollen, dann wenn möglich neue Einheiten produzieren, wenn dies nicht möglich ist, sollen sie Research betreiben.

Umsetzung:

Man iteriert über alle CityTiles, wenn die CityTiles eine Action durchführen können `can_act()`, dann wird überprüft, ob Einheiten produziert werden können. Wenn ja, dann soll eine Einheit erstellt werden, wenn nicht, soll Research betrieben werden. Zusätzlich wurde eine Methode entwickelt, die über alle Citys iteriert und überprüft, ob der Fuel der City gleich dem `get_light_upkeep()` der jeweiligen City entspricht. Wenn dies nicht der Fall, wird ein Parameter `fill_up_city = True` gesetzt, und wenn beim Iterieren den Units dann eine Unit volle Kapazität hat, soll Sie zu einer der entsprechenden Citys zurückkehren, um diese mit Fuel zu versorgen. Hierfür wurde auch noch eine neue Methode geschrieben, die `get_closest_citytile_from_city(unit, targetCity)` Methode, um das nächst gelegene CityTile der gegebenen City zu bekommen.

Analyse des Resultats:

Wenn ein oder mehrere CityTiles auf dem Weg, zur Ziel City liegen, dann geht die Unit einfach durch die City durch und verliert somit alle Ressourcen an das durchlaufene CityTile. Dadurch kann eine ganze City aussterben, nur weil eine andere City ungünstig platziert wurde. Da den Units das Ziel nicht fest zugewiesen wurde, kehren diese zurück zum Ressourcen sammeln, nachdem sie unplanmäßig auf einem CityTile gelandet sind. Auf diese Weise konnte es passieren, dass ganze Citys nie wieder aufgefüllt wurden und deswegen starben. Hierfür muss den Units ein festes Ziel bekannt sein, da es mehrere Citys geben kann und wir vermeiden wollen, dass eine City ausstirbt. Diesen 3 erkannten Problemen wollen wir uns nun widmen.

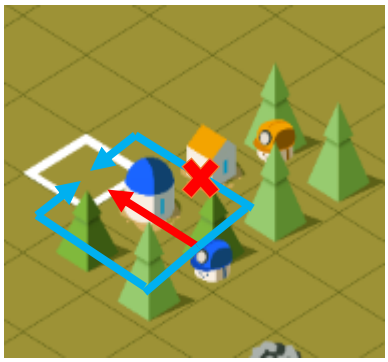

Abbildung 1: Wegfindungsproblem

Verbesserungsversuch: Nur eine Große City bauen

Wenn man nur eine große zusammenhängenden City bauen würde, so wäre es egal, auf welches CityTile man sich zubewegt, da es immer nur eine City gibt und immer diese gefüllt wird. Zusätzlich braucht man für jedes angebaute CityTile weniger Fuel als alleinstehende CityTiles. Hierbei soll somit die Bauposition neuer CityTiles immer angrenzend zur einzigen City bestimmt werden. Dies wurde mit der `get_closest_empty_tile_adjacent_to_city()` Methode umgesetzt.

Umsetzung:

Es wurde für jedes CityTile der City überprüft, ob ein angrenzendes Feld zum Bauen frei ist. Wenn ja, wurde das freie Feld zur Liste der möglichen Bauplätze hinzugefügt. Anschließend wurde der „Bauplatz“ ausgewählt, welcher sich am nächsten zur Unit befindet.

Beobachtung:

Auf kleinen Spielfeldern mit wenig Ressourcen und auf großen Spielfeldern, wo die Ressourcen weit verbreitet sind, funktioniert diese Lösung nicht. Der Grund hierfür ist, dass irgendwann nicht mehr genug Ressourcen zur Verfügung stehen, um die City am Leben zu erhalten oder diese zu weit entfernt sind und die ganze Stadt somit irgendwann stirbt. Auf kleinen Spielfeldern mit viel Ressourcen ist diese Lösung sehr gut geeignet. Wir wollen aber eine Lösung, die auf möglichst vielen Ausgangssituationen gut funktioniert. Deswegen wollen wir uns dem Problem der Bewegungsmuster der Units widmen.

Auch ist aufgefallen, dass im Spezialfall, wenn das Start CityTile von Ressourcen umgeben war oder die City ausgestorben war, die Units blockiert waren. Somit wurde geprüft, ob sich ein freies Feld um die City herum befindet. Ist dies nicht der Fall, so wurde eine neue City auf dem nächsten freien Platz gebaut.

Verbesserung: Zuweisung von Positionszielen und Verbesserung der Bewegungsmuster

Bisher sind die Units immer strikt den kürzesten/direkten Weg zum Ziel gegangen und konnten dabei stecken bleiben bzw. ihr Ziel verfehlen. Hierbei wurde die bereits bestehende Methode `direction_to()` verwendet, welche für jede Richtung in festgelegter Reihenfolge getestet hat, ob die Position näher am Ziel ist. Wenn ja wird diese Richtung gespeichert. Da es keine Abbruchbedingung gibt, wird immer die letzte Passende Richtung ausgegeben. Die Reihenfolge, in welcher die Richtungen getestet wird, lautet wie Folgt: NORTH, EAST, SOUTH, WEST. Somit ist WEST oder SOUTH wahrscheinlicher als Output als NORTH oder EAST.

Umsetzung:

Um das Problem zu lösen, gibt es die Möglichkeit, zu überwachen, ob eine Unit sich schon mehrere Züge nicht mehr bewegt hat. Wenn dies der Fall ist, kann man die Unit in eine zufällige Richtung gehen lassen. Dies ist aber sehr ineffizient, da sich die Unit bis zum Erkennen mehrere Runden nicht bewegen darf/kann, und die zufällige Richtung ggf. dafür sorgt, dass sie wieder einen Schritt zurück

geht und sich danach direkt wieder auf die vorherige Position begibt und somit wieder steckenbleibt. Deswegen soll überprüft werden, ob ein Schritt in die direkte Richtung möglich ist. Wenn ja, so soll die Unit diesen Schritt gehen. Wenn nicht, so soll die Unit in die Richtung gehen, in der die absolute Differenz der jeweiligen Koordinate (x oder y) am größten ist. Dann soll der Schritt in die gewählte Richtung überprüft werden. Ist dieser Schritt möglich, so wird dieser ausgeführt. Ist er nicht möglich, so wird die andere Koordinatenrichtung überprüft.

Hierbei gibt es noch den zusätzlichen Parameter „wants_to_build“, welche angibt, ob die Unit eine neue City bauen will und somit freundliche CityTiles umgangen werden sollen oder nicht. Auch werden feindliche CityTile betrachtet, da die Units diese auch nicht betreten können sowie Tiles, auf denen schon eine Unit steht. Der Sonderfall, dass mehrere Units auf einem freundlichen CityTile stehen können, wurde auch beachtet.

Zusätzlich wurde ein neues Dictionary angelegt, worin die Bewegungen der Units in der Aktuellen Runde gespeichert werden. Wenn sich also eine Unit in der aktuellen Runde von ihrem Feld wegbewegt, befindet sich diese Bewegung in dem Dict und das neue Feld wird somit „Blockiert“ für andere Units. Es wird anschließend genutzt, um herauszufinden, ob sich eine Unit in der Runde wegbewegt. Somit weiß man, ob das Feld die nächste Runde frei wird und somit sich eine Unit auf das Feld bewegen kann oder nicht.

Einführung einer Zufallsbasierten Aufgabenauswahl

Als letzter Schritt wurde versucht, den Agenten etwas dynamischer zu machen, indem jedem Agenten eine feste Aufgabe zugeteilt wurde. Also beispielweise konnte ein Agent die Aufgabe „EXPLORE“ erhalten, bei welcher die Unit zu einem anderen Ressourcenfeld gehen sollte, um dort dann eine neue Stadt zu bauen. Die Aktionen wurden hierbei basierend auf dem Tag und Nacht Zyklus und dem Platz im Inventar ausgewählt. Hierbei wurde beispielsweise bei vollem Inventar der Unit, mit einer Wahrscheinlichkeit von 90% eine neue Stadt Bauen und zu 10% Exploration betreiben. Zusätzlich sollten in den ersten 20 Runden nur neue Städte gebaut werden, um möglichst schnell viele Units zu haben. Auch wurde nun beim Sammeln von Kohle und Uranium, nicht mehr das gesamte Inventar gefüllt, bevor die Einheiten eine neue Aufgabe bekommen, da das Sammeln dieser viel Länger dauert und man die Ressourcen früher braucht, um eine City mit Fuel zu befüllen. Kohle und Uranium wollten dabei fast ausschließlich zum Auffüllen von Citys genutzt werden, da man beim Bau von CityTile nur viele Ressourcen verschwenden würde.

Diese zufallsbasierte Aufgabenauswahl schnitt etwas besser ab, also die vorherigen Agenten enthielt aber noch Fehler. Nachdem diese Behoben wurden Schnitt der Agent erstaunlicher weise schlechter ab. Auch ein Versuch die höherwertigen Ressourcen nur zum Auffüllen des Fuels der Citys zu nutzen, schnitt noch immer schlechter ab als der fehlerhafte zufallsbasierte Agent, was an den zufälligen Maps und Gegnern liegen könnte.

Analyse einer Deep Q Learning Lösung

Link zum untersuchten Kaggle Notebook: https://www.kaggle.com/isaacyong/lux-ai-tensorflow-dqn

Bei einem Deep-Q-Learning (DQN) werden von einem Neuronalen Netz sogenannte Q-Values berechnet. Diese geben an, welcher Reward in einem bestimmten State zu erwarten ist, wenn man eine Bestimmte Aktion durchführt. Da in unserem Fall auch Convolutional Layer benutzt werden, wird ein einheitlicher Input und somit eine Datenvorverarbeitung benötigt.

Daten Vorverarbeitung

Wenn das Environment erstellt wurde, wir eine Ressource Map mit 3 Layern mit jeweils einer Dimension pro Ressourcenart erstellt. Hierbei entsprechen die Werte der Anzahl der Ressourcen, welche bis zu 800 betragen kann. Diese werden jedoch nicht normalisiert, sodass diese sehr hohe Werte von bis zu 800 einnehmen können. Zusätzlich gibt es jeweils eine Map für die eigenen Units, CityTiles und fremden Units, CityTiles, sowie eine Road Map für die Wege.

Die Maps sind alle so groß, wie die Maximale Spielfeldgröße von 32x32. Wenn ein Spielfeld kleiner ist, so wird es in der Mitte der Map zentriert, um einen einheitlichen Input zu gewährleisten.

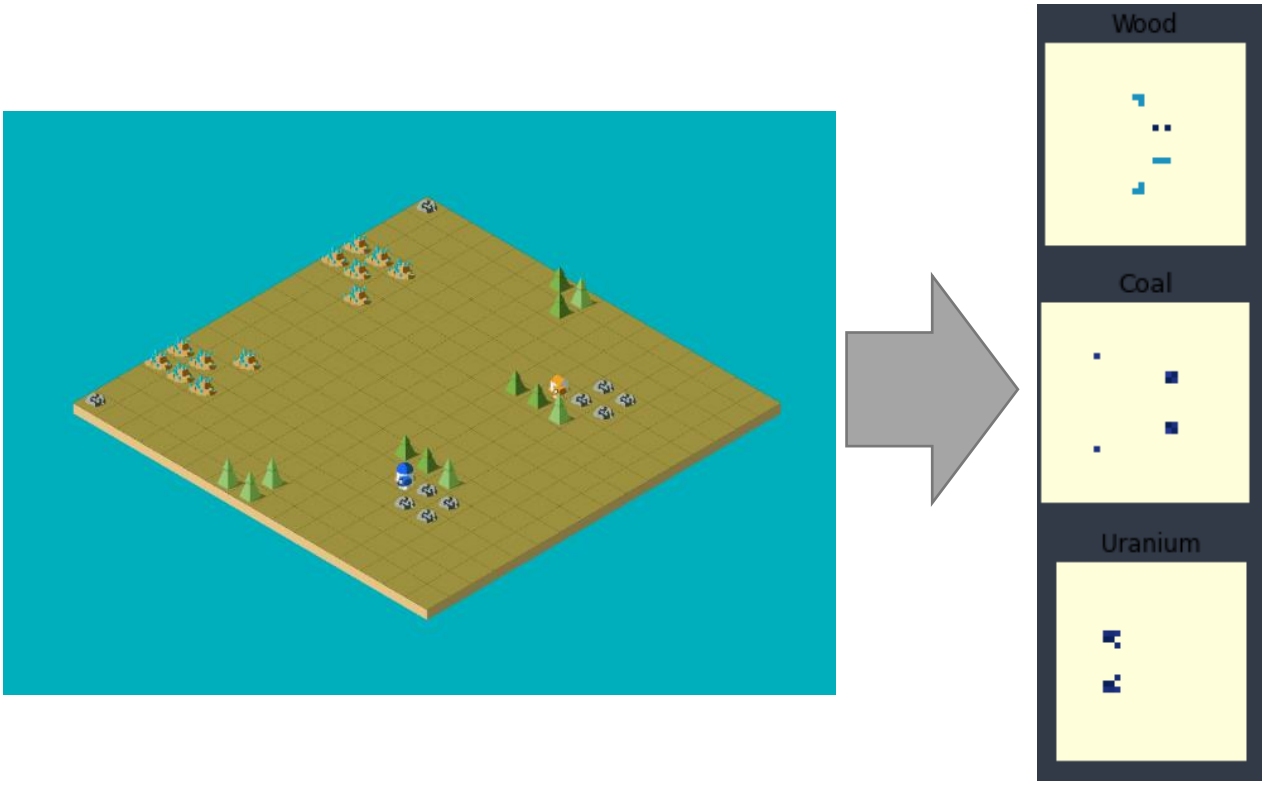

Abbildung 2: Beispiel der Datenvorverarbeitung

Die Netzarchitektur

Das Neuronale Netz besteht aus 3 Teilen. Einem Hauptteil, welcher DeLux genannt wird, wo die Maps als Input genutzt werden und dem sogenannten WorkerNet und CityNet, welche den Output des Hauptteils als Input nutzen.

Das WorkerNet und CityNet sind gleich aufgebaut. Sie bestehen aus 4 Dense Layern, welche dazwischen jeweils ein BatchNormalization Layer haben. Das Ausgabelayer besitzt dabei eine Softmax Aktivierungsfunktion. Sie Unterscheiden sich nur in der Input Size, wo das WokerNet eine Input Size von 1x29 besitzt und das CityNet eine Input Size von 1x26.

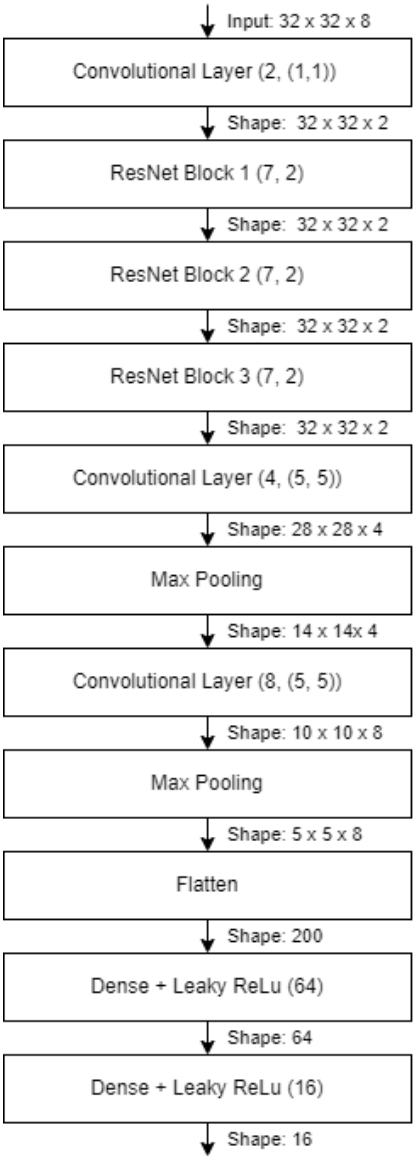

Abbildung 3: DeLux Netzarchitektur

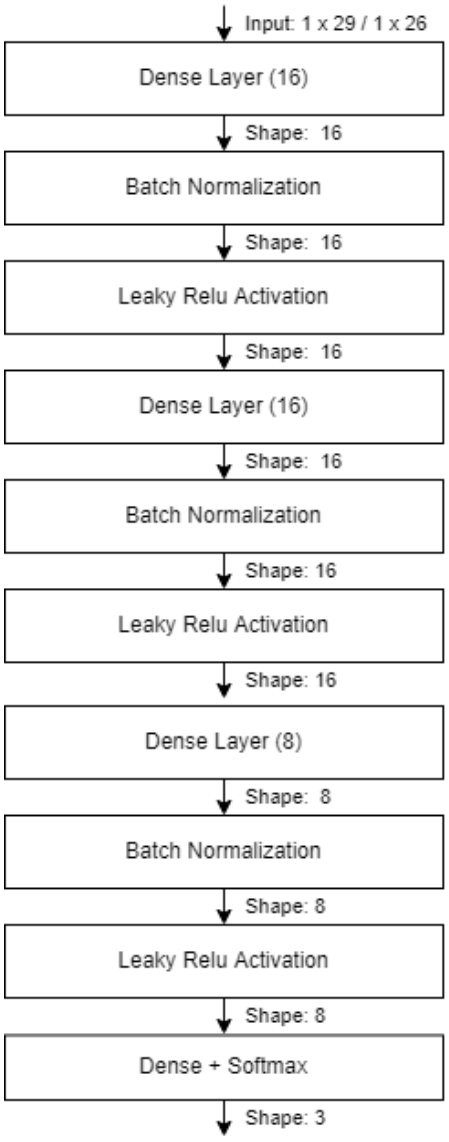

Abbildung 4: WorkerNet/CityNet Netzarchitektur

Ausgabe des Netzes

Die Ausgabe des Hauptteils ist eine State Map, sie enthält die State Value, jeder einzelnen Zelle. Bei einem DQN sollten diese im Normalfall Mehrdimensional sein, also für jede mögliche Aktion einen Wert enthalten. Hier gibt es aber nur einen Wert, weswegen es sich eher um State Values handelt als um Q-Values.

Dieser State-Value wird dann für jede Unit/jedes CityTile, welche eine Aktion ausführen können zusätzlich als Input für das jeweilige WorkerNet/CityNet genutzt.

Das WorkerNet und CityNet geben dann jeweils einen Softmax-Verteilten Wert über alle möglichen Actions aus. Hierbei wird über den größten Wert die Aktion der jeweiligen Unit/des jeweiligen CityTiles entschieden.

Auswertung

Auch nach längerem Training, ist die Leistung des Netzes sehr begrenzt, denn das beste Resultat, das erzielt werden konnte, war, dass die anfängliche Unit sich auf den Ressourcenfeldern hin und her bewegt hat. Meistens bewegt sich die Unit auf ein bestimmtes Feld wo meistens keine Ressource zu finden ist und bleibt dann dort, bis sie stirbt und somit das Spiel endet. Dies passiert üblicherweise bei Runde 30. Läuft die Unit dabei (zufällig) über ein Ressourcenfeld, so überlebt die Unit bis Runde 115 und stirbt dann. Dies ist erklärbar, da die Unit, wenn sie sich über ein Holz Ressourcenfeld bewegt, durch den Cooldown genau 6x20 Units Holz aufsammelt, welche dem Maximum von 100 Einheiten/Fuel entsprechen. Eine Unit verbraucht dabei 4 Fuel pro Zeiteinheit in der Nacht. Da eine Nacht genau 10 Zeiteinheiten entspricht, verbraucht somit eine komplette Nacht 40 Fuel. Also kann die Unit damit 2 ganze Nächte und 5 Zeiteinheiten einer Nach überleben. Dies entspricht 40+40+30+5 = 115 Zeiteinheiten, also den beobachten Zeiteinheiten.

Erkannte Probleme und Fehler

Das Netz aktualisiert seine Gewichte nicht während des eigentlichen Trainings, sondern nur während des Experience Replays. Dieser wird nur alle 10 Episoden ausgeführt. Da der Replay Buffer aber initial auf 5 Episoden begrenzt ist, verliert man 5 Episoden. Und von den 1800 Zeitschritten werden auch nur 1200 im Experience Replay genutzt, was dann nur effektiv 3,3 Episoden entspricht. Somit müssen die Gewichte des Netzes unbedingt während des Trainings aktualisiert werden und nicht nur während des Experience Replays.

Der Epsilon Decay wird bei jeder Exploration ausgeführt. So verringert sich Epsilon am Anfang sehr schnell und später kaum noch.

Auch ist mir beim Untersuchen der Distanz Maske aufgefallen, dass hier die Koordinaten vertauscht waren. Mit den Richtigen Koordinaten ist die Unit nicht mehr so weit über das Spielfeld gelaufen wie es vorher der Fall war.

Für die Distanz Maske wir die Euklidische Distanz verwendet, welche nicht dem Bewegungsmuster der Units entspricht, somit wurde diese Distanz in eine Manhattan Distanz umgewandelt, welche dem Bewegungsmuster der Units entspricht.

Es wurde zusätzlich noch festgestellt, dass das CityNet und WorkerNet je Episode immer die gleiche Ausgabe liefen, welches auf einen Fehler der Architektur des Netzes hindeutet und somit müsste diese überarbeitet werden.

Die Maps werden nicht normalisiert, was zu exploding Gradients führen kann.

Es wird immer das gleiche Spielfeld benutzt, welches zu Over Fitting führen kann. Somit ist es sinnvoller das Spielfeld immer zu wechseln. Allerdings ändert sich damit auch mit jeder Episode der State Space.

Mögliche Verbesserungen

- Das Netz enthält kein Target Network, welches essenziell für ein DQN ist.
- Die Distanz Map könnte stark verkleinert werden, sodass nur nahe Ziele angepeilt werden.
- Der Regel Basierte Agent kann mit den vom DQN gezeigten Maps trainiert werden.
- Eine zusätzliche Bau Map, wo Orte in der Nähe von Ressourcen und anderen CityTiles bevorzugt werden.
- Einzelne Netze für die CityTiles und Units sind möglicherweise besser geeignet.

Die Ressource Map kann man verbessern, indem man die angrenzenden Felder, von welchen auch die Ressource gesammelt werden können, berücksichtigt werden. Beispielsweise, indem man jeweils in der Ressource Map alle Felder der Ressource und die Angrenzenden Felder um 0.2 erhöht. Somit wäre der Wert einer Ressource, welche von 4 angrenzenden Ressourcen umgeben ist 1. Das könnte man auch mit der Menge der Ressourcen kombinieren.

Schlussfolgerung

Die Weiterentwicklung des Regel Basierten Agenten zeigt, dass hier viel Logisches Denken erforderlich ist. Je weiter die Entwicklung vorangeschritten ist, umso schwerer wurde es neue Ideen zu verwirklichen. Zur Lösung dieses Komplexen Problems können daher Neuronale Netze, wie zum Beispiel ein DQN eingesetzt werden. Dies ist mit diesem Netz leider nicht wirklich gelungen, allerdings konnte man durch die Analyse wichtige Erkenntnisse gewinnen, die einem helfen, den Regel Basierten Agenten weiterzuentwickeln oder die verschiedenen Maps weiterzuentwickeln und auf Basis dessen dann ein eigenes Neuronales Netz zu trainieren. Auch könnte man bspw. die Reward Funktion so anpassen, dass auch die gesammelten Ressourcen berücksichtigt werden, was dem Netz helfen soll, einfacher zu lernen.