Carl von Ossietzky Universität Oldenburg

Bachelor Informatik
Projektarbeit - Anwendungen der KI

Game AI für das Spiel Retux

Autor:
Gruppe 3

Erstgutachter:
apl. Prof. Dr.-Ing. Jürgen Sauer

Zweitgutachter:
M. Sc. Julius Möller

Abteilung Systemanalyse und -optimierung
Department für Informatik

Oldenburg, 26. Januar 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AF	**A**ktivierungs**f**unktion
FCNN	**F**ully **C**onnected **N**eural **N**etwork
FLOSS	**F**ree**L**ibre **O**pen **S**ource-**S**oftware
KI	**k**ünstliche **I**ntelligenz
NEAT	**N**euro**e**volution of **A**ugmented **T**opologies
NN	**N**eural **N**etwork
PEAS	**P**erformance **E**nvironment **A**ctuators **S**ensors
ReLU	**Re**ctified **L**inear **U**nit
Tanh	**H**yperbolic **Tan**gent Function
UCNN	**Un**connected **N**eural **N**etwork

Kapitel 1

Einleitung

Anwendungen welche *künstliche Intelligenz* nutzen kommen in unserer Welt immer häufiger vor und Gebiete ihrer Anwendung findet man mittlerweile in so gut wie jedem Teil unseres Lebens (vgl. Avneet Pannu, 2015). Eines dieser Gebiete ist die Verwendung künstlicher Intelligenzen als Ersatz für einen menschlichen Spieler bei digitalen, wie auch analogen *Spielen*. Durch diese Verwendung kann die momentane Fähigkeit künstlicher Intelligenzen anhand eines einfach nachvollziehbaren Beispiels, wie dem Brettspiel *Go*, aufgezeigt werden (vgl. Silver, David u. a., 2016).
Eine mögliche Realisierung einer KI kann durch den *Neuroevolution of Augmented Topologies Algorithmus* (im folgenden *NEAT* genannt) geschaffen werden. NEAT wurde von Stanley (2004) im Rahmen seiner Doktorarbeit entwickelt und bildet die Grundlage für dieses Projekt.
Durch die Arbeit von Papavasileiou und Jansen (2016) wurde gezeigt, dass die anfängliche Wahl der initalen Topologie des *NN* von *NEAT* einen Einfluss auf die Leistung des Algorithmus haben kann. Standardmäßig verwendet der NEAT-Algorithmus eine initial minimale Netzwerk Struktur (im folgenden als UCNN bezeichnet) (vgl. Stanley, 2004, S. 27ff). Mit einer initialen unconnected Topologie sei es wahrscheinlicher, im Gegensatz zu einer initial voll verbundenen Topologie (im folgenden als FCNN bezeichnet), dass das Netzwerk eine effizientere und robustere Lösung liefern würde (vgl. ebd. S. VII). Die Arbeit von Shi (2008) zeigt außerdem auf, dass *FCNN* aufgrund der hohen Anzahl an Verbindungen Probleme mit *Rauschen* haben können.
Die vorliegende Arbeit dokumentiert die Implementation eines *AI-Agenten* mit NEAT, sowie der benötigten *Game-AI-Schnittstelle*. Des Weiteren werden die zwei Topologien *UCNN* und *FCNN* anhand dieser Implementation verglichen. Die Netzwerke werden dabei so konfiguriert, dass das *UCNN* sich *feature-deselective* und das *FCNN* sich *feature-selective* verhält. Dies bedeutet, dass das *UCNN* eher *Verbindungen* aufbaut und das *FCNN* eher *Verbindungen* abbaut.
Die Forschungsfrage hierbei lautet, inwiefern sich ein *feature-selective UCNN* und ein *feature-deselective FCNN* in ihrem Lernverhalten unterscheiden. Aufgrund der

bereits zitierten Arbeiten wird die Hypothese aufgestellt, dass das *UCNN* im Vergleich zum *FCNN* langsamer lernt, beide aber in ihren Fitnesswerten nach einem Zeitpunkt t konvergieren, wobei das *UCNN* häufiger zu einer gewünschten Lösung kommt. Es wird also vermutet, dass das *FCNN* schneller Lösungen für das Problem liefert, diese aber mehr *Rauschen* enthalten.
Diese Arbeit ist in fünf Kapitel gegliedert, wobei das erste Kapitel die Einleitung darstellt in der ein grober Überblick über die Arbeit und die in ihr behandelte Fragestellung gegeben wird. In dem darauffolgenden Kapitel „Grundlagen“ werden für das weitere Verständnis notwendige Begriffe und Konzepte geklärt. Das dritte Kapitel „Aufbau Agent“ beinhaltet die Beschreibung des Agenten und des Neuronalen Netzes und stellt eine Erweiterung des vorhergehenden Kapitels dar. Im vierten Kapitel „Durchführung und Auswertung“ werden die vier unterschiedlichen Durchläufe und ihre Besonderheiten, sowie die aus den Ergebnissen gezogen Schlüsse aufbereitet. Abschließend werden im letzten Kapitel „Fazit“ unsere Ergebnisse im Hinblick auf unsere zu Beginn verfasste Hypothese zusammengefasst und ein Ausblick auf mögliche Folgearbeiten geliefert.

Kapitel 2

Grundlagen

In diesem Kapitel sollen die Grundlagen der verwendeten Software NEAT sowie dem Spiel ReTux erläutert werden.
Das Spiel *ReTux* ist ein *FLOSS Platformer-Spiel*, welches von den *Mario*-Spielen inspiriert ist. Der Name geht zurück auf ein Wortspiel bezogen auf die Worte „redux“ und „Tux“. Das Spiel basiert auf der *Seclusion Game Engine* für *Python* und ist von dem EntwickerInnen-Duo *„The Diligent Circle“* vollständig in *Python* verfasst worden. (vgl. Circle, 2020)
NEAT ist eine Methode zur Entwicklung neuronaler Netze durch *Neuroevolution*. Es sollen durch *NEAT* gleichzeitig Lösungen optimiert und komplexer werden, indem zusätzlich zum Anpassen der *Weights* beim Training die *Architektur* des *Netzwerkes* kontinuierlich angepasst wird. (vgl. Stanley und Miikkulainen, 2002)

2.1. FLOSS

FLOSS, *free/libre open-source software*, bezeichnet *Software*, welche frei verfügbar und deren *Sourcecode* öffentlich ist. Das „frei“ bezieht sich hierbei jedoch nicht auf

Abbildung 2.1.: Screenshot des Spiels ReTux
Quelle: Circle (2020, Abbildung 1)

die Kosten, sondern auf die *Verfügbarkeit*, weswegen zur Klärung von *libre open-source software* gesprochen wird.
(vgl. Free and Open-Source Software, 2020)

2.2. Tux

Der Pinguin *„Tux“* ist nicht nur die *Spielfigur* des Spiels „ReTux“, sondern auch das offizielle Maskottchen des *Linux Kernels* und gilt als das generelle Symbol für *Linux*. Die Idee einen Pinguin als Maskottchen für den *Linux Kernel* zu nutzen, stammt von *Linus Torvalds*, dem Initiator und Entwickler des *Linux Kernels*. (vgl. Tux, 2020)

2.3. Platformer

Platformer- oder auch *„jump 'n' run“*-Spiele sind eine Unterkategorie von Aktion-Spielen und ihre Besonderheit ist, dass die SpielerInnen ihre Umgebung, wie der alternative Name des Genres es schon aussagt, durch Springen, Klettern und Rennen erkunden lässt und sie so zu dem, durch das Spiel definierten, Ziel gelangen können.
Die Umgebung besteht hierbei oftmals aber nicht nur aus unbeweglichen Objekten die überwunden werden müssen, sondern ist meistens noch mit sich bewegenden Objekten und Gegnern gefüllt.
All diese Hindernisse müssen von menschlichen oder Computer-SpielerInnen überwunden werden und das oftmals nur durch Springen, Laufen oder direktes interagieren mit der Umgebung, wie die Nutzung eines Gegners als improvisierte Leiter. (vgl. Platform Game, 2020)

2.4. Künstliche Intelligenzen und Platformer

Der *Agent* der künstlichen Intelligenz muss die Objekte seiner *Environment* klassifizieren und anhand der *Fitness-Function* eine Reaktion für die jeweiligen Objekte bestimmen.
Je nachdem welche Gegnertypen oder andere für den *Agenten* gefährliche, Lebenspunkte abziehende, Objekte ihm in seiner Umgebung begegnen, muss er eine Strategie formulieren mit ihnen umzugehen, um das Ende des *Levels*, sein Hauptsächliches *Goal*, zu erreichen.
Die Einordnung wie gut ein Durchlauf des Agenten für ein bestimmtes *Level* war, berechnet sich durch die *Fitness-Function*.

2.5. Besonderheiten von ReTux

Da *ReTux* vollständig in *Python* geschrieben worden ist und zudem *FLOSS* ist, ist nicht nur der gesamte *Source-Code* öffentlich, sondern es bot sich so die Möglichkeit ohne größere Probleme eine *Schnittstelle* zwischen der, in *Python* implementierten, *NEAT-Library* und dem Spiel *ReTux* zu schaffen.
Hierdurch war es uns möglich die Objekte des Spiels direkt abzugreifen und zu nutzen, ohne auf eine weitere Ebene, wie *Image-Processing*, zurückgreifen zu müssen.

2.6. Neuroevolution of Augmented Topologies

Für das *Neural Network* wird der *Neuroevolution of Augmented Topologies Algorithmus* verwendet, welcher die Philosophie verfolgt, dass jungen *Genomen* einer *Generation* eine Chance gegeben werden soll ihr volles Potential entfalten zu können. Bei der Lösung einer Idee sei es wichtig, so Stanley, dass neue Ideen getestet und verbessert werden, bevor sie verworfen werden. Um den *Lösungsraum* des *Agenten* nicht zu weit einzuschränken scheint dieser Ansatz sinnvoll, da der Agent im besten Falle nicht nur eine Lösung des Problems finden soll, sondern auch eine besonders Gute. Der *Agent* soll also die Möglichkeit bekommen auch unkonventionelle Lösungsansätze, also solche die erst einmal nicht direkt zielführend scheinen, verfolgen zu können, um diese in die Evaluierung einfließen zu lassen. (vgl. Stanley, 2004, S. 143f.)

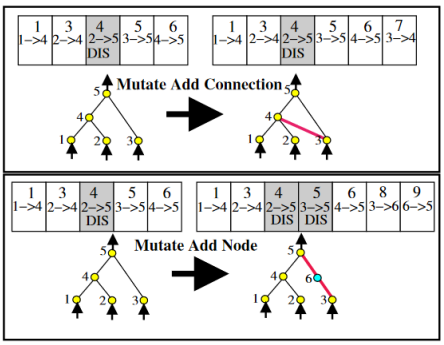


Abbildung 2.2.: NEAT Mutationen
Quelle: Stanley und Miikkulainen (2002, Abbildung 2)

Besonders geeignet sei *NEAT* daher, so Stanley, für offene Probleme. Also solche Probleme, für die es keine direkt beste Lösung gibt. In kompetitiven Spielen, sei es nicht möglich die Komplexität des besten Spielers abschätzen zu können, um daraus die Größe des benötigten Genoms zu bestimmen. Bei NEAT wird die Population durch hinzufügen neuer Spezies und Strukturen über den zeitlichen Verlauf komplexer, wodurch neue Lösungen gefunden werden können. *NEAT* nutzt hierbei einen in der *Neuroevolution* unkonventionellen Ansatz, indem durch das Training

des *Agenten* nicht nur die *Weights* angepasst werden, sondern das ganze *Netzwerk*. Der Vorteil dieses Ansatzes ist es, dass die *Netzwerk Topologie* nur zu einem gewissen Grad geplant werden muss. So müssen die *Inputs* und *Outputs* festgelegt werden, die *Connections* und *Weights* sowie die *Nodes* der *Hidden Layer* hingegen werden während des *Trainings* wie in Abbildung 2.2 zu sehen angepasst. (vgl. Stanley, 2004, S. 2ff.)
Dieser Ansatz ermöglicht also zum einen eine vereinfachte Planung des Netzwerkes, da Fehlplanungen der *Netzwerk Topologie* durch das *Training* kompensiert werden können. Zum Anderen scheint die Herangehensweise für das Spiel *ReTux* geeignet, da der *Lösungsraum* ebenso aufgrund der sich wandelnden *Netzwerk Topologie* nicht unnötig eingeschränkt wird.

2.6.1. Neat-Configuration

Zur richtigen Konfiguration der *KI* können Einstellungen in einer *Config-Datei* vorgenommen werden. Innerhalb dieser gibt es einige Unterpunkte welche im folgenden kurz erklärt werden sollen. Hierbei werden nicht alle Einstellungsmöglichkeiten genannt, lediglich solche welche im späteren Verlauf der Arbeit aufgegriffen werden und somit für das Verständnis basal sind.

Innerhalb des [NEAT] Abschnittes werden allgemeine Einstellungen vorgenommen, also solche die für die Durchführung des *Trainings* relevant sind.

- *fitness_criterion*: Wie soll die Abbruchbedingung für das Training berechnet werden. Hier kann aus *Minimum, Maximum* und *Durchschnitt* gewählt werden.
- *fitness_threshold*: Erreicht der von der Funktion *fitness_criterion* erreichte Wert die hier gewählte Schwelle, wird das Training abgebrochen.
- *pop_size*: Die Anzahl der Genome innerhalb einer Generation.

Der umfangreichste Abschnitt [DefaultGenome] legt Parameter fest, welche von der *DefaultGenome Klasse* der *NEAT Library* genutzt werden. Hier werden die wichtigsten Entscheidungen bezüglich der *Netzwerk Topologie* getroffen.

- *activation_default* : Die Aktivierungsfunktion.
- *activation_mutate_rate* und *activation_options*: Diese beiden Einstellungen werden genutzt um mit einer gewählten Wahrscheinlichkeit die Aktivierungsfunktion mutieren zu lassen. Beispielsweise kann hier mit einer Wahrscheinlichkeit von *activation_mutate_rate* = 0.01 auf eine Aktivierungsfunktion aus der gewählten *activation_options* Menge z.B. *activation_options = sigmoid, relu* gewechselt werden.

- *conn_add_prob* und *conn_delete_prob*: Die Wahrscheinlichkeit mit der durch *Mutation* eine *Connection* erzeugt bzw. entfernt wird.

- *feed_forward*: Die Option ob das Netzwerk ausschließlich *Feedforward Connections* haben soll oder auch *Recurrent Connections* erlaubt sind.

- *initial_connection*: Legt fest in welcher Form initiale *Connections* vorhanden sind. Hierbei gibt es eine Vielzahl an Optionen, wovon für diese Arbeit aber nur die *full*- und *unconnected*-Option betrachtet werden. Hierdruch lässt sich eintellen, ob das *Neural Network* zu Beginn initiale *Connections* haben darf oder es diese ausschlielich durch *Mutation* selbst findet. Diese Einstellung ist ein wesentlicher Bestandteil der Forschungsfrage dieser Arbeit, weshalb diese Option im nächsten Abschnitt ausführlicher behandelt wird.

- *node_add_prob* und *node_delete_prob*: Die Wahrscheinlichkeit mit der durch Mutation ein neuer *Node* erzeugt bzw. einer vorhandener *Node* entfernt wird.

- *num_hidden*, *num_inputs* und *num_outputs*: Anzahl der initialen *Hidden*- sowie *Input*- und *Output-Nodes*

- *weight_max_value* und *weight_min_value*: Minimaler und maximaler Wert den die *Weights* halten können.

- *weight_mutate_rate* und *weight_replace_rate*: Wahrscheinlichkeit mit der sich der Wert eines *Weights* durch *Mutation* verändert. Dies geschieht bei der *Mutation* durch das hinzuaddieren eines zufälligen Wertes. Beim Ersetzen wird ein Zufallswert als Grundlage für ein neues *Weight* genommen, welches das Alte ersetzt.

Der [DefaultReproduction] Abschnitt dient als Grundlage für die *DefaultReproduktion Klasse*, welche für das *reproduzieren* von *Generationen* benötigt wird.

- *elitism*: Legt die Anzahl der *Genome* fest, welche bei der *Reproduktion* unverändert übernommen werden sollen. Diese *Genome* sind immer jene mit den höchsten *Fitnesswerten*.

- *survival_threshold*: Legt fest, welcher prozentuale Anteil jeder *Species* bei der *Reproduction* übernommen wird.

- *min_species_size*: Legt die minimale Anzahl der *Genome* pro *Species* fest.

(vgl. McIntyre u. a., 2019)

2.6.2. Initial Connection

Die *initial_connection* legt fest in welcher Form initiale *Connections* vorhanden sind und ist daher ein wesentlicher Bestandteil der Forschungsfrage dieser Arbeit.
Durch die *unconnected*-Option hat das *NN* initial keine *Connections*. Diese werden während des *Trainings* durch *Mutationen* hinzugefügt und auch wieder entfernt. Die Wahrscheinlichkeit mit der solche *Mutationen* auftreten lässt sich durch die Parameter für *conn_add_prob* und *conn_delete_prob* konfigurieren.
Die *initial_connection*-Option bietet zwei Varianten. Zum einen *full_direct* und zum Anderen *full_nodirect*. Diese unterscheiden sich dahingehend, ob lediglich die *Input Nodes* mit den *Hidden Nodes* und die *Hidden Nodes* wiederum mit den *Output Nodes* verbunden sind (*full_direct*) oder die *Input Nodes* nur mit den *Hidden Nodes* und diese wiederum mit den *Output Nodes* verbunden sind. Erwähnswert ist an dieser Stelle außerdem, dass auch *recurrent Connections* möglich sind, insofern die Option *feed_forward* auf False gesetzt ist. (vgl. McIntyre u. a., 2019)
Wie bereits in der Einleitung erwähnt wurde, wird für das *FCNN* die *Feature-deselective* und für das *UCNN* die *Feature-selective* Herangehensweise gewählt. In der Arbeit von Papavasileiou und Jansen (2017) wurde gezeigt, dass die initial Verbindungen *FCNN* und *UCNN* keine signifikanten Unterschiede in ihrer Leistung aufzeigen, wenn diese jeweils für *feature-selective NEAT* und *feature-deselective NEAT* genutzt werden. *Feature-deselective NEAT* deselektiert dabei Verbindungen, *feature-selective-Neat* hingegen funktioniert konträr und selektiert Verbindungen. Aus den Ergebnissen ihrer Arbeit folgerten sie, dass beide Herangehensweisen den *Connections* an weniger signifikanten *Inputs* kleinere *Weights* und den *Connections* an relevanteren *Inputs* höhere *Weights* zuweisen und somit zu gleichen Ergebnissen konvergieren. (vgl. Papavasileiou und Jansen, 2017)

Kapitel 3

Aufbau Agent

In diesem Kapitel wird der Aufbau des Agenten beschrieben, welcher keinen *klassischen Reinforcement-Learning* Ansatz verfolgt. Der klassische Ansatz zeichnet sich durch eine direkte Belohnung des Agenten durch ein Feedback der Umgebung aus und dieses Feedback fließt direkt in die Entscheidungsfindung der nachfolgenden Aktionen des Agenten ein (vgl. Kaelbling u. a., 1996). In dem implementierten Ansatz erfolgt das Feedback über die Leistung des Agenten, welche über die Fitness-Funktion berechnet wird und die erst nach dem Abschluss eines Durchlaufs eines Individuums einen Einfluss auf die zukünftigen Entscheidungen des Agenten nimmt. Beginnend wird der Agent durch die *PEAS-Beschreibung* und die Darlegung seiner Umgebung definiert. Im weiteren Verlauf werden wichtige Aspekte zur Konstruktion des Agenten, durch die Beschreibung der verwendeten Fitness- und Aktivierungsfunktion, erläutert. Um eine Basis für das Training des Agenten vorzugeben wird die gewählte Konfiguration von *NEAT* eingeführt und im weiteren Verlauf dieser Arbeit ggf. angepasst. Zum Ende dieses Kapitels werden zwei Implementierungen und die Funktionsweisen ihres Neuronalen Netzes vorgestellt. Dabei werden die Nachteile der ersten Implementierung herausgestellt, welche durch die darauf folgende Implementierung gelöst werden sollen.

3.1. PEAS

Im Folgenden wird die Aufgabenumgebung des Agenten spezifiziert. Dies ist notwendig, da eine Aufgabenumgebung sich direkt auf das Design des Agenten auswirkt. Hierbei wird die sogenannte *PEAS-Beschreibung* verwendet. *PEAS* setzt sich dabei aus fünf Aspekten zusammen.

1. *Agententyp:* Gibt den Namen des zu erstellenden Agenten an.
2. *Performance:* Beschreibt, welche Qualitäten der Agent haben soll d.h. an welchen Qualitätsmerkmalen soll der Agent bewertet werden.

3. *Environment:* Das *Environment* beschreibt die Umgebung des Agenten in dem diese agiert. Hier soll beschrieben werden auf Ereignisses der Agent zu reagieren hat bzw. welche Ereignisse eintreten können in der Umgebung.

4. *Actuators:* Beschreibt die Schnittstelle zwischen Agent und Umgebung. Der Agent kann über Aktuatoren seine Umgebung manipulieren.

5. *Sensors:* Der Agent nimmt über die ihm zur Verfügung stehenden Sensoren seine Umgebung war.

Nachdem diese Aspekte für den Agenten definiert wurden kann die Entwicklung des Agenten durchgeführt werden. (vgl. Russell und Norvig, 2012, S. 66ff)

PEAS	
Agententyp	Pinguin
Performance	Zurückgelegte Distanz in Richtung des Ziels. Einsammeln von Bonusgegenständen. Minimierung von erlittenem Schaden.
Environment	Bonusgegenstände interagierbare Gegenstände Böden Gegner Pinguin bewegliche Plattformen Hindernisse, welche Schaden verursachen können
Actuators	Simulierte Tasteneingaben
Sensors	Schnittstelle zum Spiel in Form einer Matrix

Tabelle 3.1.: PEAS

Der *Agententyp* hat den Bezeichner *Pinguin* bekommen. Die *Performance* des Agenten wird anhand drei verschiedener Aspekte bewertet. Zum einen wird der Agent anhand der *zurückgelegte Distanz in Richtung des Ziels* bewertet. Dies ist wichtig, da bei dem Spiel das Entfernen vom Ziel nicht zum gewinnen führt. Aus diesem Grund wird der Agent dafür belohnt, wenn dieser sich dem Ziel nähert. Des Weiteren wird der Agent für das *einsammeln von Bonusgegenständen* belohnt. In dem Spiel existieren verschiedene Arten von Bonusgegenständen, welche zu einem höheren Punktestand führen. Dies soll den Agenten fördern auch alternative Routen zu nehmen. Ein weiterer Aspekt ist das *minimieren von erlittenem Schaden*. In der Umgebung des Agenten existieren Gegner, welche bei der Kollision mit dem Agenten schaden verursachen. Sobald der Agent zu viel Schaden in einem Level bekommt stirb dieser und das Level ist für diesen Agenten verloren. Der Agent soll durch dieser Kriterium dazu animiert werden darauf zu achten, dass dieser Gegnern ausweicht oder diese eliminiert. Diese Punkte werden in dem Kapitel 3.3 zu einer *Fitness-Function* ausgearbeitet.

Die *Environment* des Agenten besteht aus verschiedenen Objekten, welche unterschiedliches verhalten Ausweisen auf die sich der Agent einstellen muss. In dem vorherigen Abschnitt wurden bereits *Bonusgegenstände* und *Gegner* erklärt. In diesem Spiel können *interagierbare Gegenstände* von dem *Pinguin* aufgehoben, aktiviert oder teilweise geworfen werden. Des Weiteren existieren einige Gegnertypen, welche durch das eliminieren durch den *Pinguin* selbst zu einem *interagierbaren Gegenstände* werden. Neben *Böden* auf denen der *Pinguin* laufen kann existieren weitere *bewegliche Plattformen*. Diese Plattformen bewegen sich in einem regelmäßigem Intervall von einem Punkt zu einem anderen. Neben *Gegnern*, welcher Schaden an dem *Pinguin* verursachen existieren *Hindernisse, welche Schaden verursachen können*. Der Aspekt des *Environment's* wird später näher erläutert.
Damit der Agent mit seiner Umgebung wahrnehmen kann muss dieser diese sowohl durch *Sensors* wahrnehmen und *Actuators* manipulieren. Die Wahrnehmung der Umgebung wird durch eine zweidimensionale Matrix abgebildet und dem Agenten zur Verfügung gestellt. Durch diese Eingabe kann der Agent dann durch *simulierte Tasteneingaben* mit der Umgebung interagieren und diese manipulieren.

3.2. Environment

Environment
Fully Observable
Stochastic
Sequential
Dynamic
Continuous
Singleagent

Tabelle 3.2.: Environment

Die Umgebung des *Agenten* wurde in der obigen Tabelle beschrieben und nachfolgend wird aufgezeigt, warum die obige Auswahl getroffen worden ist.
Der erste Eintrag der Tabelle sagt aus, dass die Umgebung für den *Agenten vollständig beobachtbar* ist. Dies ist der Fall, da die *Sensoren* des *Agenten*, die *Matrix* aus der die *Inputs* errechnet werden, alle für eine Aktion relevanten Aspekte seiner *Umgebung* erkennen können.
Bevor wir ein *Raster*, welches überprüft, was für ein Gegenstand an einer bestimmten Stelle ist und diese Informationen dann als *Input* an den *Agenten* weitergibt, als *Sensor* implementiert hatten, hatten wir einen über *Raycasts* implementierten *Sensor*, durch den die Umwelt nicht vollständig wahrgenommen worden ist. Wären wir bei diesem *Modell* geblieben, dann wäre die Umwelt für unseren *Agenten* nur

teilweise beobachtbar gewesen, was aus unserer Sicht zu vermeiden war.
Da die *Folgezustände* des *Agenten* nicht vollständig durch die, durch den *Agenten* ausgeführten Aktionen und den aktuellen Zustand in dem sich der *Agent* befindet, bestimmt werden können, ist die Umgebung als *stochastisch* eingestuft worden. Das Verhalten der Objekte ist nicht für alle Objekte in der Umgebung vorhersehbar, bzw. aus dem aktuellen Zustand und den, durch die von dem *Agenten* getätigten, Aktionen ablesbar.
Des Weiteren ist die Umgebung als *sequenziell* eingestuft, da die vom *Agenten* getroffenen Entscheidungen alle zukünftigen Entscheidungen beeinflussen.
Da die Umgebung sich während der Entscheidungsfindung des *Agenten* verändert handelt es sich um eine *dynamische* Umgebung. Der *Agent* muss also selbst während der Entscheidungsfindung die Umgebung betrachten und auf Änderungen eingehen.
Der *Agent* befindet sich, da der Zustand der Umgebung und der *Pinguin* sich durchgehend ändern und es *keine diskrete Menge* an möglichen Zuständen gibt, in einer *stetigen* Umgebung.
Abschließend ist noch zu sagen das es sich um eine *Einzelagentenumgebung* handelt, da der *Agent* andere Objekte der Umgebung auch nur als solche behandeln muss und es sich nicht um weitere Agenten handelt, die zu dem *Hauptagenten* in *Konkurrenz* existieren oder mit ihm *kooperieren.*(vgl. Russell und Norvig, 2012, S.69ff.)

3.3. Fitness-Function

Fitnessfunktionen bilden die Grundlage genetischer Algorithmen. Sie stellen eine Abbildung in einem zu durchsuchendem Raum dar, in dem sie jedem Element eine reelle Zahl zuweisen. Aus den Summen der zu den einzelnen Elementen zugeordneten Zahlen, ergeben sich so die jeweiligen Fitnesswerte, mithilfe derer sich die Genome vergleichen lassen. So wird bei der Bewertung von Genomen, oft dasjenige mit der größten Fitness bevorzugt. (vgl. Gerdes u. a., 2013, s.37ff)

Für die Bewertung eines *Genoms* in dem System der ersten Implementierung 3.6.1, gibt es einige Elemente die zu beachten sind. Am relevantesten ist in diesem Fall, ob das *Genom* das vorhandene *Level* erfolgreich abgeschlossen hat oder nicht. Der Abschluss des *Levels* lässt sich in einen binären Wert konvertieren. Da dieser Wert jedoch von erhöhter Relevanz ist, wird er mit 2.000 multipliziert, um sicher zu stellen, dass hierdurch die erreichte Fitness, positiv beeinflusst wird. Im Gegensatz zu einem erfolgreichen Abschluss, steht eine Niederlage durch frühzeitigen Abbruch, ausgelöst durch den Tod des zu steuernden Pinguins. Dieser wird zwar auch binär erfasst, jedoch mit 300 multipliziert und von der restlichen Fitness subtrahiert. Sollte das *Genom* durch Kontakt mit Objekten, wie zum Beispiel *Gegner* Schaden erlitten

haben, so werden der Fitness weitere Punkte in Höhe von 300 pro verlorenem Leben abgezogen. Die grundlegende Anforderung, das Spiel zu beenden, ist so abgedeckt. Eine weitere Methode um die Fitness zu erhöhen, bieten die *Coins*, welche in der Umgebung versteckt sind. *Coins* werden bei Kontakt eingesammelt und erhöhen die Fitness um jeweils 200.
Um *Genome*, die den Pinguin entweder gar nicht oder nicht zielführend steuern zu bestrafen, wurde ein *Timeout* implementiert. Dieser war zu Beginn auf einen Wert von 50 gesetzt und wurde im weiteren Verlauf angepasst. So lässt sich auch die für den Durchlauf einer Generation notwendige Zeit stark beschränken. Der *Timeout* wird dabei bei jedem *Game-Tick* inkrementiert, bis der vorgegebene Wert erreicht wurde. Beim Erreichen dieses Wertes wird das *Training* des *Genoms* beendet.
Um Beschränkungen des *Trainings* durch den *Timeout* zu verhindern, wird bei dem Erreichen von *Checkpoints* der *Timeout* zurückgesetzt. *Checkpoints* erreicht der *Agent* durch das Zurücklegen einer festgelegten Entfernung in Höhe von jeweils 50 Pixeln. Nach abgelaufener Zeit wird der aktuelle Durchlauf des Genoms beendet und das nächste kann gestartet werden. Zu Beginn wird ein Konstanter Wert von 5.000 festgelegt, welcher dann durch die Anzahl der erreichten *Checkpoints* (bei dieser Berechnung ist der Nenner >= 1) geteilt und am Ende des Durchlaufs von der restlichen Fitness abgezogen wird. Darüber hinaus wird die Fitness für jeden erreichten *Checkpoint* mit einem Wert von 150 addiert. So lassen sich auch Genome, die das Level nicht erfolgreich abschließen, anhand ihres Fortschrittes bewerten.

$$\begin{aligned} genome_{fitness} = {} & (2000 * level_{won}) - (300 * genome_{died}) - (\frac{5000}{checkpoints_{reached}}) \\ & + (200 * CoinsCollected) + (150 * Checkpoint_{Multiplier}) \\ & - (300 * HP_{lost}) \end{aligned} \tag{3.1}$$

Bei der Zweiten Implementierung, wurde die Fitnessfunktion dem fortschreitenden Projekt angepasst und die Bewertungen für den Abschluss des Levels von 2.000 auf 5.000 erhöht. Auch der Abzug für den Tod, wurde auf einen Wert von 750 festgelegt.

$$\begin{aligned} genome_{fitness} = {} & (5000 * level_{won}) - (750 * genome_{died}) - (\frac{5000}{checkpoints_{reached}}) \\ & + (200 * CoinsCollected) + (150 * Checkpoint_{Multiplier}) \\ & - (300 * HP_{lost}) \end{aligned} \tag{3.2}$$

Die Funktionen 3.1 und 3.2 sind nicht als solche in den Implementierungen zu finden. Die genauen Werte werden während der Laufzeit berechnet und die Formel wird hier

nur zur Veranschaulichung zusammenhängend dargestellt.

3.4. Aktivierungsfunktion

Aktivierungsfunktion bilden die gewichtete Summe der *Neuronen* auf einen festen Bereich ab (vgl. Stanley, 2004, S. 9). Die eingehenden Daten werden manipuliert und eine Ausgabe für das Netz, wird erzeugt. Je nach verwendeter Funktion, können sie entweder linear oder nichtlinear sein, wobei sich die nicht linearen AF besser für die meist nichtlinearen Berechnungen eignen. (vgl. Nwankpa u. a., 2018, s. 3)
Die verschiedenen AF, die in den unterschiedlichen Durchläufen verwendet wurden, werden im Folgenden kurz vorgestellt.

3.4.1. Rectified Linear Unit - ReLU

Die 2010 von Nair und Hinton vorgestellte Funktion $ReLU$, ist die zurzeit am weitesten verbreitete AF und findet besonders im Bereich *Deep Learning* Anwendung. Da sie eine nahezu lineare AF darstellt, ist sie schnell zu berechnen und einfach zu optimieren. $ReLU$ arbeitet mit einem Grenzwert von 0, wobei Eingaben aus dem negativen Bereich zu einer 0 konvertiert werden. Im positiven Bereich, gibt es jedoch keine Einschränkungen. Daraus können hohe Zahlen resultieren, die sich je nach Anwendungsfall, gut interpretieren lassen. Jedoch kann der gegebene Grenzwert, bei vermehrt negativen Eingaben, zu unerwünschtem Verhalten führen. (vgl. Nwankpa u. a., 2018, s. 8)

$$f(x) = max(o, x) \tag{3.3}$$

3.4.2. Sigmoid

Sigmoid oder auch *logistische Aktivierungsfunktion*, ist eine nicht lineare AF, die oft in *Feedforward neuronalen Netzwerken* verwendet wird und *sigmoidal* also S-Förmig verläuft. Sie liefert Ausgaben in dem Bereich von 0 bis 1 und eignet sich daher für Probleme, die eine binäre Ausgabe erfordern oder für die Bestimmung von Wahrscheinlichkeiten, da diese sich ausschließlich in dem gegebenen Bereich bewegen. (vgl. Nwankpa u. a., 2018, s. 5)

$$f(x) = \frac{x}{\sqrt{1 + x^2}} \tag{3.4}$$

3.4.3. Hyperbolic Tangent Function - Tanh

Tanh ist wie *Sigmoid*, eine nicht lieare *AF* die *sigmoidal* verläuft, deren Ausgaben hingegen von -1 bis 1 reichen. Außerdem ist es hier möglich ein durch Fehlerrückführung geleitetes Lernverfahren zu implementieren, was zu besseren Trainingsergebnissen führt. Sie wird oft in den Bereichen der Sprachverarbeitung und Spracherkennung verwendet. (vgl. Nwankpa u. a., 2018, s. 7)

$$f(x) = (\frac{e^x - e^{-x}}{e^x + e^{-x}}) \tag{3.5}$$

Da zu Beginn das Ausmaß, die notwendigen Variablen und die benötigten Ausgaben des Agenten noch unbekannt waren, wurde für die erste Implementierung 3.6.1 die *AF ReLU* verwendet. So war es möglich, besonders in dem Bereich der Schnittstelle, schnelle Anpassungen durchzuführen und verschiedene Ansätze auszuprobieren.
Nach den ersten Testläufen kam es zu ungewolltem Verhalten des Pinguins. Der unbegrenzte Ausgabebereich war nur schwer auf die begrenzten Aktionen abzubilden und es kam zu plötzlichen Laufrichtungswechseln. Hinzukommend wurden die möglichen Aktionen des Pinguins auf binäre Eingaben beschränkt, also Taste gedrückt bzw. Taste nicht gedrückt.
In der zweiten Implementierung 3.6.2, für die Durchläufe 4.2, 4.3, 4.4 wurde *Sigmoid* als *AF* verwendet, da diese den erwünschten Ausgabebereich von 0 bis 1 Liefert und die von *ReLU* verursachten Probleme beseitigt.
Für den letzten Durchlauf 4.5 wurden *Tanh*, *Sigmoid* vorgezogen. Die von *Tanh* generierten Ausgaben, die sich in dem Bereich von – 1 bis 1 aufhalten, lassen sich einfach in binäre umwandeln. Besonders das bessere Lernverhalten, war hier ausschlaggebend für den Wechsel der *AF*.

3.5. Neat-Configuration-Settings

Da die in den Grundlagen 2, unter Neat-Configuration 2.6.1 beschriebene Konfiguration ein wichtiger Aspekt ist, der sich stark auf die in dem Abschnitt Durchführung 4 erhobenen Daten auswirken kann, wird hier die Basis der *Config-Dateien* vorgestellt.
Die hier nicht explizit benannten Werte der *Config-Datei*, entsprechen den empfohlenen Standardeinstellungen von McIntyre u. a. (2019) und orientieren sich teilweise an ähnlichen Projekten wie dem von Verma (2020). Alle Werte der *Config-Dateien* sind in der Tabelle B zu finden.
Basiswerte, die nicht den Standardeinstellungen entsprechen, sind wie folgt:

- *fitness_threshold*: Wurde auf einen nicht erreichbaren Wert von 1000000 ge-

setzt, um einen vorzeitigen Abbruch des Trainings zu verhindern.

- *pop_size*: Die Anzahl der *Individuuen* pro Generation wurde auf 100 gesetzt, dadurch wurde sichergestellt, dass initial eine hohe Variation an *Individuuen* in einer Population existieren. Dieser Wert wurde in den weiteren Durchläufen auf 50 reduziert, um die Trainingsdauer zu reduzieren.

- *max_stagnation*: Dieser Wert wurde von dem Standardwert 15 auf 2 herunter gesetzt, um eine häufigere Neubildung der *Species* über die geplanten 100 Generationen zu erreichen

- *species_elitism*: Dieser Wert wurde von dem Standardwert 0 auf den Wert 2 heraufgesetzt mit der Intention Genome, die eine hohe maximale Fitness aufweisen, in der Speziation der Generationen stärker zu berücksichtigen

- *elitism*: Dieser Wert wurde ebenfalls auf 2 angepasst, um sicherzustellen, dass *Individuuen*, welche eine gute Leistung zeigen, aber in einer im Durchschnitt schlechterer Spezies sind, trotzdem eine Chance haben sich in den nachfolgenden Generationen auszubreiten.

- *species_fitness_func*: Der Standardwert für diese Einstellung ist „mean“, wurde aber auf „max“ geändert, um die gewünschten Daten bei der Berechnung der Fitness zu erhalten.

3.6. Neural Network

Für das *NN* wurde ein *Feed Forward Network* gewählt, statt einem *Recurrent Neural Network*. Ein *Recurrent Neural Network* besitzt anders als ein *Feed Forward Network* rekurente Verbindungen zu den Knoten. Dies kann genutzt werden, damit ein Netzwerk temporäre zeitliche Abhängigkeiten lernen kann (vgl. Stanley, 2004, S. 9f). Im Rahmen dieses Projektes wird der Einfachheit halber ein *Feed Forward Network* verwendet. Aufgrund der der hohen Diversität an unterschiedlichsten NEAT-Parametern und möglichen Realisierungen der *KI-Game-Engine-Schnittstelle* wurde zuerst eine Implementierung entworfen, welche später zum Teil verworfen bzw. modifiziert wurde. Auf die jeweiligen Erkenntnisse und den daraus resultierenden Designentscheidungen, welche bei diesem Prozess erlangt wurden, wird in den folgenden beiden Unterkapiteln eingegangen.

3.6.1. Erste Implementierung des NN

Bei der ersten Umsetzung des *NN* wurde für die *Inputs* ein Sichtradius um den Pinguin vorgegeben, innerhalb dessen alle Objekte gescannt und an das *NN* übergeben

wurden. Die Übergabe dieser Objekte geschah hierbei als X- und Y-Koordinate, weshalb es pro Objekt zwei Inputs gab.

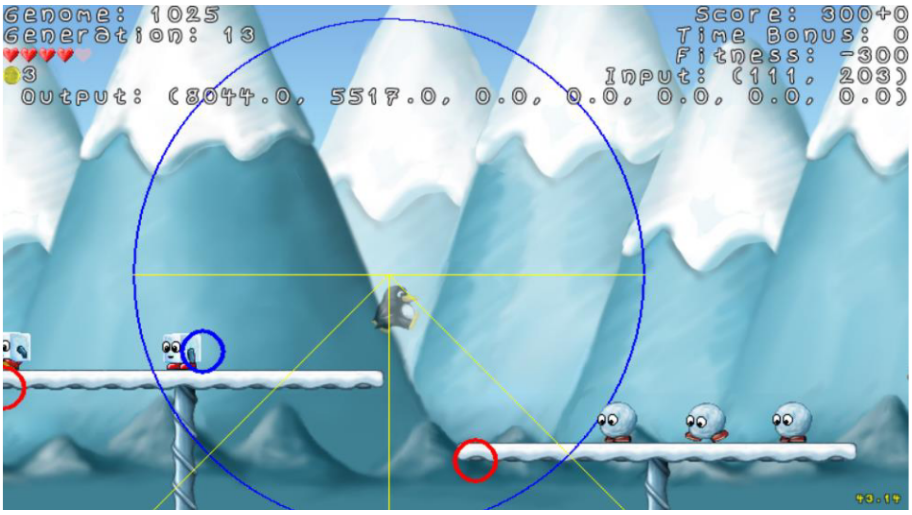


Abbildung 3.1.: Screenshot der ersten Visualisierung

Zusätzlich wurden fünf *Raycasts* jeweils ausgehend vom Pinguin bis zum Sichtradius genutzt, welche die Distanz vom Pinguin bis zum ersten kollidierenden Objekt übergab. Kollidiert der *Raycast* mit keinem Objekt wird eine maximale Distanz von 500 übergeben. Auf dem Screenshot in Abbildung 3.1 ist diese Darstellung visualisiert zu sehen. Die gelben Linien sind hierbei die Raycasts, erkannte Objekte sind durch einen Kreis gekennzeichnet, wie z.B. der blau umkreiste Eisblock-Gegner. Zu erkennen ist hier außerdem ein Problem mit den Plattformen, diese werden von der Engine als ein zusammenhängendes Objekt behandelt und geben immer die Anfangskoordinaten der Plattform bei Kollision mit den *Raycasts* wieder. Daher musste für die exakte Distanzberechnung Gleichung 3.6 bis 3.9 verwendet werden.

$$dist(ray_{left}) = Player_x - Object_x \tag{3.6}$$

$$dist(ray_{right}) = Object_x - Player_x \tag{3.7}$$

$$dist(ray_{down}) = Player_y - Object_y \tag{3.8}$$

$$dist(ray_{diag\_left}) = dist(ray_{diag\_right}) = \sqrt{(Player_y - Object_y)^2 + (x_{off})^2} \tag{3.9}$$

Grundlage für die Formeln 3.6, 3.7 und 3.8 ist eine einfache Distanzberechung ausgehend von den Koordinaten des Spielers und der Objekte mit denen der *Ray* kollidiert. Gleichung 3.9 liegt einer Distanzberechnung durch Pythagoras zugrunde. Hierbei werden ebenso die Koordinaten der Objekte und des Spielers sowie (x_{off}) als Konstante für die Diagonalen *Rays* zur Berechnung genutzt. Für ray_{diag_left} gilt

$(x_{off}) = -10$ und für ray_{diag_right} $(x_{off}) = 10$. Berechnet wird die Länge der Hypotenuse und dadurch die Distanz zwischen dem Spieler und dem Objekt. Es gilt $dist(ray_{diag\_left}) = dist(ray_{diag\_right})$, da das Quadrat von x_{off} sowohl für einen negativen als auch einen positiven Wert die Gleiche Lösung liefert.

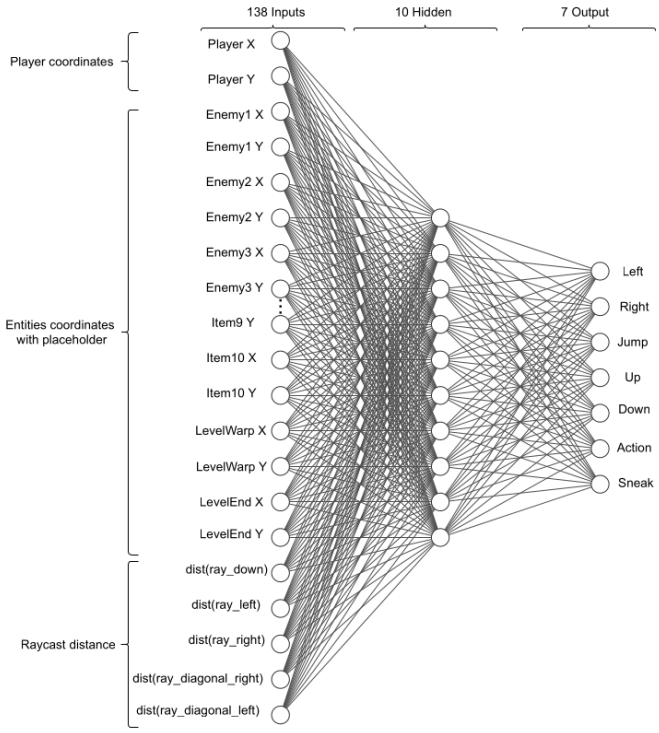
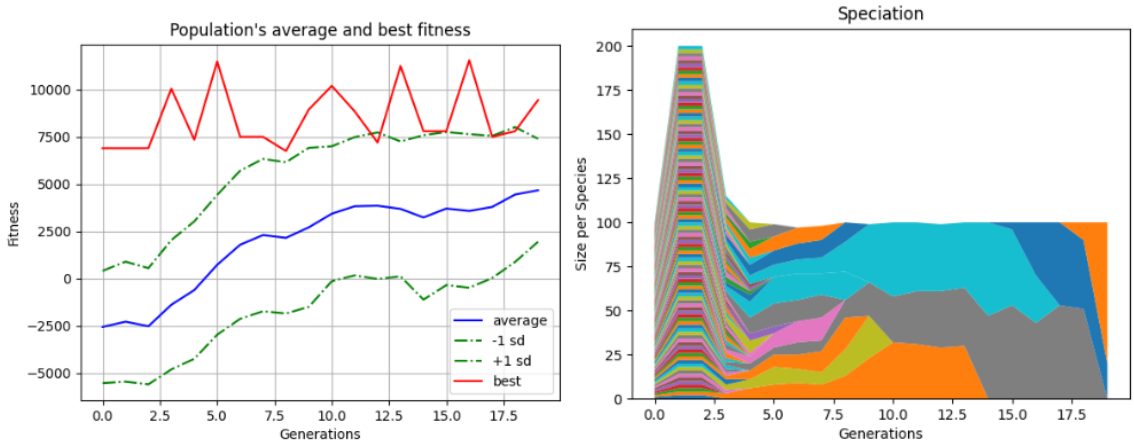


Abbildung 3.2.: Erste Version des *NN*

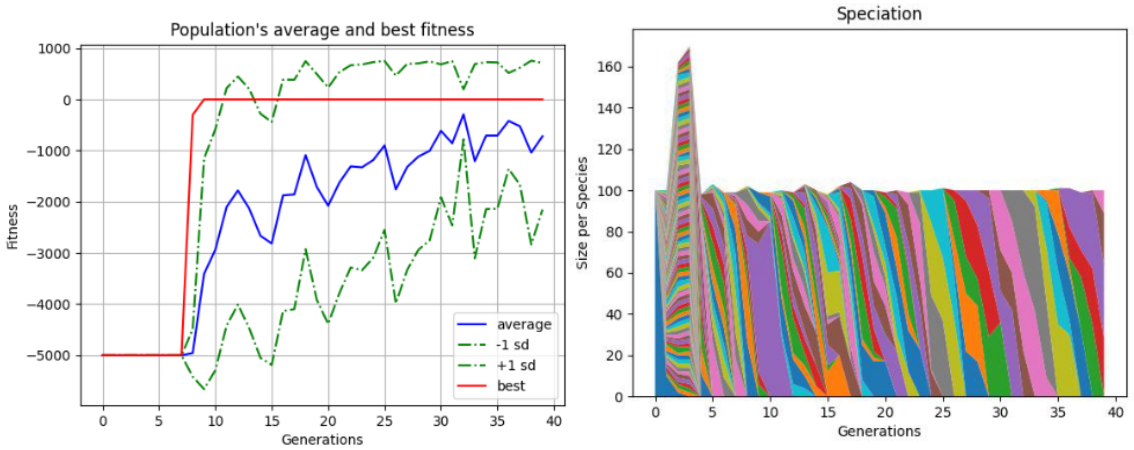
Da das *NN* eine feste Anzahl an Eingaben erwartet, die beschriebene erste Implementierung aber je nach Situation eine unterschiedliche Anzahl an Objekten erfasst, mussten Platzhalter an das *NN* übergeben werden. Wie in der schematischen Darstellung in Abbildung 3.2 zu sehen, werden die Koordinaten des Spielers, eine feste Anzahl an Entitäten sowie die jeweilig berechneten Distanzen der Raycasts als *Input Nodes* in das *NN* gegeben. Entitäten sind hierbei - so wie bei der späteren Version des *NN* auch - unter anderem Gegner, Blöcke, interagierbare Objekte und Coins. Ist beispielsweise gerade kein Gegner-Objekt innerhalb des Sichtradius des Pinguins, so werden die 20 *Input Nodes* - welche für potentiell 10 Gegner-Objekte vorgesehen sind - mit 0 belegt. Wären zwei Gegner innerhalb des Radius des Pinguins, so beinhalteten die *Input Nodes Enemy1 X* und *Enemy1 Y* die Koordinaten des ersten Gegner-Objekts, die *Input Nodes Enemy2 X* und *Enemy2 Y* die Koordinaten des zweiten Gegner-Objekts und die restlichen *Enemy-Input-Nodes* wären mit 0 belegt. Hierbei entstehen die ersten Probleme mit dieser Implementierung. Aufgrund der Tatsache, dass für die Positionen der Objekte zwei Parameter übergeben werden kommt es zu doppelten Eingaben. Dadurch könnte die Trainingsdauer des *NN* negativ beeinflusst werden. Des Weiteren ist die Eingabe des Netzwerkes uneinheitlich

gewählt worden. Einerseits werden Koordinaten von Objekten in das Netzwerk gegeben und auf der anderen Seite werden Distanzen zu Objekten eingeben. Darüber hinaus werden durch die notwendigen Platzhalter nicht zwingend alle *Input-Nodes* genutzt. Befinden sich beispielsweise nie mehr als drei Gegner innerhalb des Sichtradius des Pinguins, so bleiben die restlichen 14 *Input-Nodes*[1] ungenutzt. Diese *Input-Nodes* tragen somit keinen wesentlichen Beitrag zu der Entwicklung des *NN* bei. Ein Weiteres Problem im Zusammenhang mit der Aktivierungsfunktion des *NN* wurde bereits in Kapitel 3.4 erläutert.



(a) (b)

Abbildung 3.3.: 20 Generationen FCNN - Erste Version des *NN*



(a) (b)

Abbildung 3.4.: 40 Generationen UCNN - Erste Version des *NN*

Nachdem die Implementierung des *NN* abgeschlossen war, wurden anschließend die ersten Netzwerke trainiert. In Abbildung 3.3 und Abbildung 3.4 sind die ersten Ergebnisse zu erkennen. Auf den ersten Blick scheint das *NN* mit der initial unconnec-

[1] Insgesamt gibt es 10 Platzhalter für Gegner-Objekte, welche jeweils 2 Positionsparameter haben. Werden nur drei Gegner-Objekte erkannt, so bleiben 7*2=14 Input-Nodes ungenutzt

ted Verbindung in Abbildung 3.4 schnellere Ergebnisse zu liefern, jedoch muss die Skalierung beachtet werden. Das *NN* mit der initial unconnected Verbindung erreicht keine Positive Fitness im Durchschnitt über 40 Generationen, während das *NN* mit einer initial fully-connected Verbindung bereits nach drei Generationen die ersten Positiven Ergebnisse liefert. Des Weiteren ist in Abbildung 3.4 zu erkennen, dass das unnconnected *NN* in kurzer Zeit viele verschiedene Spezies erzeugt, während das *FCNN* nach wenigen Generationen eine geringe Anzahl an Spezies aufweist. Aufgrund dessen wird vermutet, dass das *UCNN* eine längere Zeit benötigt als das *FCNN*, um Verbindungen aufzubauen, welche zu hohen Fitness Werten führen. Dieser Verdacht ist dadurch begründet, dass das *UCNN* zu Beginn keine bzw. wenige Verbindungen zwischen Knoten besitzt und diese erst durch Mutationen über die Generationen aufbauen muss. Das *FCNN* hingegen besitzt bereits Verbindungen zwischen allen Knoten benachbarter Layer und erzeugt somit direkt Ausgaben, welche zu Aktionen des Pinguins führen. Hierbei sind allerdings viele redundante Verbindungen vorhanden, welche, wie in der Arbeit von Shi (2008) gezeigt, ein Rauschen verursachen könnten und es daher zu unvorhersehbaren Ausgaben kommen kann. Deshalb wurde eine zweite Version des *NN* entworfen, welche diese Probleme umgehen soll. Diese wird im nächsten Abschnitt beschrieben.

3.6.2. Zweite Implementierung des NN

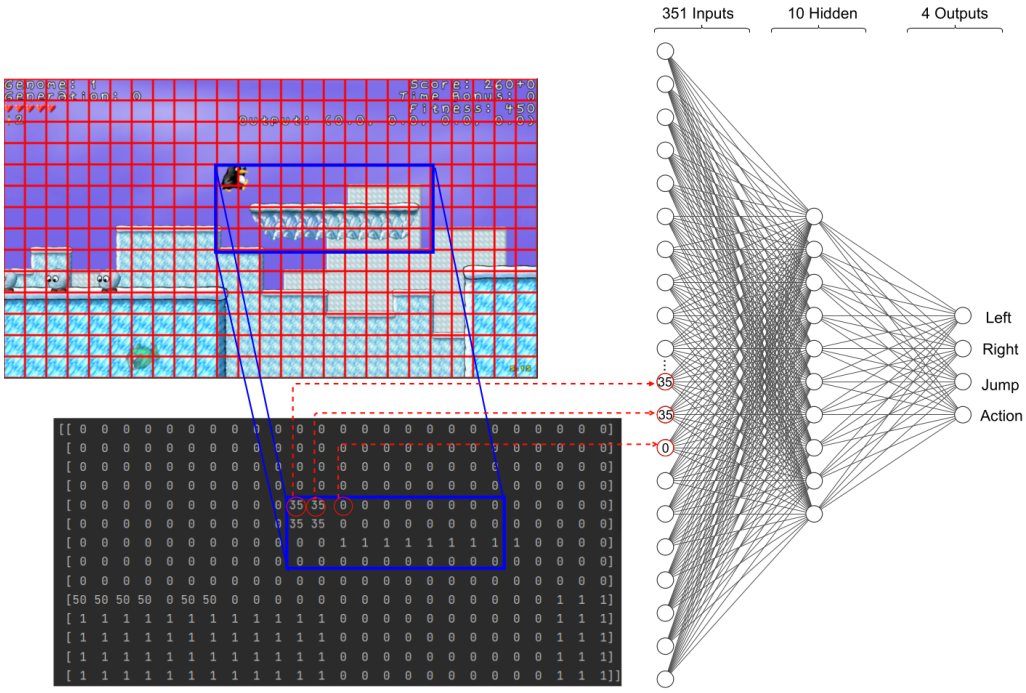


Abbildung 3.5.: Zweite Version des *NN*

Bei der zweiten Implementierung bildet nicht mehr der Pinguin mit seinem Sichtradius den Mittelpunkt, sondern die Sicht des Spielers. Dafür wurde das Spielfens-

ter wie in Abbildung 3.5 dargestellt in 350 Blöcke aufgeteilt, da die *Game-Engine* alle Objekte als 32x32 Pixel Elemente visualisiert und verwaltet. Das Spielfenster hat eine Größe von $800 Pixel * 448 Pixel = 35.8400 Pixel$ welche durch die $(32 * 32) Pixel * 350 Inputs = 35.8400 Pixel$ erfasst werden können, da jedes Objekt aus einem oder mehreren 32x32 Pixel großen Blöcken besteht. Diese Blöcke werden je nachdem ob und welcher Objekt-Typ sich darin befindet auf einen Wert gemappt, wodurch wie in Abbildung 3.5 zu sehen eine Wertematrix ensteht, welche von den konkreten Objekte des Spielfeldes abstrahiert. Jeder Eintrag dieser Matrix wird dann auf einen festen *Input Node* des *NN* gemappt. Befindet sich beispielsweise der Pinguin innerhalb eines Blocks, wird eine 13 übergeben, bei einem Gegner-Objekt eine 50 und bei Plattformen- und Boden-Objekten eine 1. Befindet sich kein Objekt innerhalb des Blocks, wird eine 0 übergeben. Des Weiteren wurde ein weiterer Input in Form eines Booleans ergänzt, ob der Pinguin ein Objekt hält. Diese Änderung wurde vorgenommen, damit der *Agent* die Möglichkeit erhält zu Lernen, dass er Objekte aufheben und nutzen kann. Dies wurde bei der ersten Implementierung nicht beachtet. Durch die genannten Änderungen der Schnittstelle zwischen *KI* und *Spiel-Engine* - und somit der Art und Weise wie und vor allem welche *Inputs* übergeben werden - soll das Lernen optimiert werden. Vorherige Probleme der ersten Implementierung wie z.B. die uneinheitliche Vermittlung der *Inputs* durch *Raycasts* und *Platzhalter*, sollen durch das einheitliche *Grid-System* behoben werden.

Kapitel 4

Durchführung und Auswertung

Zur Auswertung und Vergleichbarkeit der *Agenten* wurden drei Durchläufe absolviert, bei denen die *UCNN* und die *FCNN* jeweils ähnlichen Einstellungen unterlagen. Diesen Durchläufen folgt ein weiterer, vierter und abschließender Durchlauf, bei dem stärkere Änderungen der Einstellungen vorgenommen worden sind.
Im folgenden Abschnitt werden die jeweiligen Einstellungen erklärt, sowie die Erkenntnisse und die daraus resultierenden Änderungen diskutiert. Wichtig ist hierbei, dass die erreichten *Fitnesswerte* zwar innerhalb der Durchläufe direkt vergleichbar sind, zwischen diesen aber nicht, da teilweise unterschiedliche *Level* verwendet wurden. Erklärungen hierzu befinden sich in den jeweiligen Abschnitten.

4.1. Erwartungen

Durch die kommenden Durchläufe wird versucht eine Aussage über die in der *Einleitung* definierte Hypothese zu treffen. Die in dieser Arbeit formulierte Hypothese befasst sich mit der Fragestellung, ob ein *UCNN* im Vergleich zu einem *FCNN* bei sonst ähnlichen Einstellungen langsamer lernt. Darüber hinaus wird jedoch angenommen, dass beide zu einem Zeitpunkt t zu einem gleichen Fitnesswert konvergieren. Des Weiteren wird behauptet, dass das *UCNN* häufiger zu einer gewünschten Lösung kommt und das das *FCNN* schneller Lösungen für das Problem liefert, aber mehr *Rauschen* enthält.
Konkret wird erwartet, dass die Durchläufe keinen signifikanten Unterschiede in der Leistung der initial Topologien aufweisen. Falls sich jedoch Unterschiede aufzeigen sollten, so sollten diese nach einer gewissen Zeit auf einen ähnlichen Fitnesswert konvergieren.

4.2. Erster Durchlauf

Das *UCNN* in Abbildung A.2 (a) hat mit 35.800 eine höhere maximale Fitness als das *FCNN* aus Abbildung A.1 (a) mit 27 400 erreicht. Dieses Maximum ist

beim *UCNN* innerhalb der ersten 30 Generationen erreicht worden und bei dem *FCNN* innerhalb der ersten zehn Generationen. Eine annähernd große Fitness wurde von dem *FCNN* in diesem Durchlauf nicht erneut erreicht, weshalb sich dafür entschieden wurde die Anzahl der *Generationen*, sowie die Anzahl der *Individuen* pro Population, von 100 auf 50 zu reduzieren. Im Durchschnitt erreichen beide *Topologien* eine Durchschnitts-Fitness von etwa 10.000. In Abbildung A.1 (a) ist ein Einbruch der Durchschnitts-Fitness zwischen Generation 40 und 60 zu erkennen. Dies ist möglicherweise darauf zurückzuführen, dass es durch Änderungen, durch die Operatoren von *NEAT*, welche zur Erstellung von neuen *Individuen* dienen oder dazu bestehende *Individuen* anzupassen, in diesem Fall zu schlechteren Ergebnissen gekommen ist. Dieser starke Einbruch der Fitness hat sich jedoch in den darauf folgenden Generationen nicht mehr eingestellt und die Durchschnitts-Fitness hat sich auf den oben genannten Wert eingependelt.
Bei der Betrachtung der Speziation ist festzustellen, dass beim *FCNN* zu Beginn eine Populationsgröße von bis zu 200 Individuen vorhanden ist, während es beim *UCNN* in der populationsreichsten Generation nur 120 Individuen sind. Im weiteren Verlauf wird diese hohe Individuenzahl weder von *FCNN*, noch von *UCNN* wieder erreicht und sie pendeln sich bei den vorgegebenen 100 Individuen pro Generation ein. Die Zusammensetzung der Generationen aus den unterschiedlichen Spezies beinhaltet beim *UCNN* im Mittel eine höhere Anzahl an Spezies, als beim *FCNN*.
Aufgrund der oben beschriebenen Auswertung und weiteren Aspekten, welche beim Beobachten des Verhaltens des *Pinguins* aufgefallen sind, konnte festgestellt werden, dass das gewählte *Level 27* eine zu hohe Komplexität aufweist. Aus diesem Grund wurde sich dafür entschieden, dass weniger komplexe *Level 02* zur weiteren Auswertung zu nutzen. Darüber hinaus musste ein weiterer Fehler behoben werden, bei dem der *Pinguin interagierbare Gegenstände* nicht nutzen konnte.

4.3. Zweiter Durchlauf

In diesem Durchlauf ist zu erkennen, dass beide *NN* eine ähnliche maximale Fitness von 27.080 (*FCNN*, Abbildung A.3) und 27.150 (*UCNN*, Abbildung A.4) erreicht haben. Das *UCNN* besitzt mit 1.285 in Generation 0 eine geringere minimale Fitness und mit 7.500 eine geringere durchschnittliche Fitness als das *FCNN*. Der geringste beim *FCNN* erreichte Fitnesswert liegt bei 12.480 und der durchschnittliche Fitnesswert bei 10.000.
Die im ersten Durchlauf festgestellte erhöhte Anzahl der Individuen der frühen Generationen tritt auch in diesem Durchlauf wieder auf und beim *FCNN* ist zum zweiten Mal eine Dopplung der Individuen festzustellen. Auch in diesem Durchlauf pendelt sich die Individuenzahl auf die, für diesen Durchlauf, festgelegten 50 Individuen pro

Generation ein.
Des Weiteren sieht man beim *FCNN*, dass die Anzahl der Spezies für die Zusammensetzung der Generation sich ab der neunten Generation auf ein oder zwei Spezies pro Generation beschränkt. Dies ist beim *UCNN* nicht der Fall. Hier sind bei Generation neun sechs verschiedene Spezies Teil der Generation und es kommt erst bei späteren Generationen zu einer Reduzierung der vorhandenen Spezies innerhalb einer Generation.
Bei beiden *NN* wurde nach ungefähr 5 Generationen die maximale Fitness erreicht, welche im weiteren Verlauf an diesem Wert stagniert. Außerdem konnte bei der Beobachtung des Verhaltens des *Agenten* festgestellt werden, dass solche Genome, welche gelernt haben sich nach rechts zu bewegen und dabei zu springen, hohe Fitnesswerte erlangen. Genome die ein ausgeprägteres Verhalten zeigten, wurden aufgrund eines schnellen *Timeouts* aussortiert. Um ein größeres Spektrum an Verhalten zu erhalten und den *Agenten* so nicht zu stark in seiner Handlungsfreiheit einzuschränken, wurde im folgenden Durchlauf die Zeit bis zu einem *Timeout* verachtfacht. Außerdem wurden die Eingabeparameter für eine bessere Differenzierung der Objekte durch den Agenten angepasst.

4.4. Dritter Durchlauf

Analog zu dem Durchlauf in Abschnitt 4.3 ist hier eine ähnliche maximale Fitness von 27.050 (*FCNN*, Abbildung A.5) und 29.950 (*UCNN*, Abbildung A.6) erreicht worden. Das *UCNN* besitzt mit -1 100 in Generation 0 eine geringere minimale Fitness als das *FCNN* mit 13.430 in den Generationen 0, 1, 6 und 38. In diesem Durchlauf entspricht die maximale Durchschnitts-Fitness für *FCNN* 12.180, welche in Generation 39 erreicht wird. Die maximale Durchschnitts-Fitness von *UCNN* beträgt 12.450 und wird in Generation 30 erreicht. Die minimale Durchschnitts-Fitness ist sowohl für *UCNN* als auch für *FCNN* in den ersten Generationen festzustellen. Beide Topologien haben im Durchschnitt eine Fitness von 10.000.
Wie in den vorherigen Durchläufen erzeugt *FCNN* in Abbildung A.5 (b) eine Dopplung der Population. Die Individuenanzahl pendelt sich im Verlauf des Training wieder auf die vorgegeben 50 Individuen pro Population ein. Für das *UCNN* ist dieses Verhalten in Abbildung A.6 nur minimal zu erkennen. Es wird lediglich eine maximale Individuenanzahl pro Population von $\sim$ 60 in den ersten Generationen erreicht. Zu erkennen ist, dass die Anzahl der erstellten Spezies von *FCNN* sich auf wenige Spezies einpendelt, während *UCNN* über die gesamte Entwicklung neue Spezies erstellt.
Durch genauere Analyse der *Fitness-Funktion* ist aufgefallen, dass nicht jedes vom *Agenten* eingesammelte *Bonusobjekt* bei der Berechnung der Fitness berücksichtigt

worden ist. Die *Fitness-Funktion* wurde dahingehend angepasst, dass der *Agent* für jedes eingesammelte *Bonusobjekt* eine Fitnesssteigerung erhält und es zu einem Zurücksetzen des *Timeouts* kommt. Dies wurde vorgenommen um dem *Agenten* einen Anreiz für das Sammeln von Bonusobjekten zu liefern. Des Weiteren wurden für den nächsten Durchlauf weitere Aspekte der *Fitness-Funktion* angepasst.
Die Bestrafung, welche der *Agent* für das erhalten eines *Timeout* bekommt, ist von 5.000 Minuspunkten auf 750 Minuspunkte verringert worden und es wird keine Betrachtung der zurückgelegten Strecke zur Bewertung der Bestrafung bezüglich der *Checkpoints* mehr vorgenommen. Gleichzeitig wurde die für den Agenten benötigte Distanz zum Erreichen eines *Checkpoints* von 50 Einheiten auf 500 Einheiten um eine Größenordnung erhöht und die für einen Timeout benötigte Zeit von 400 Einheiten auf 1.500 Einheiten heraufgesetzt worden.
Außerdem wurde der *Spezieselitismus* von zwei auf eins halbiert und die *max_stagnation* von zwei auf sechs erhöht. Diese Änderung wurde aufgrund der Überlegung getroffen, dass der *Agent* aufgrund der vorgenommenen Änderungen eine größere Varianz an Ergebnissen produzieren würde und die Spezies mit der höchsten maximalen *Fitness* zu schnell wieder verworfen werden würde, da die außergewöhnlich hohen *Fitnesswerte* besonders bei *FCNN* eine Besonderheit darstellen. Die Aktivierungsfunktion wurde, wie im Unterabschnitt 3.4.3 erläutert, von *Sigmoid* auf *Hyperbolic Tangent* geändert.

4.5. Vierter Durchlauf

Für diesen Durchlauf wurden Änderungen an der *Fitness-Funktion* vorgenommen, welche nachfolgend durch die Angabe der aktualisierten Version der *Fitness-Funktion* dargelegt werden. Diese hier aufgestellte *Funktion* weicht in einigen Punkten von der in Abschnitt 3.3 definierten *Fitness-Funktion*, ab.
Die Bestrafung des Genoms durch einen *Timeout* wird nun nicht mehr durch das Erreichen von *Checkpoints* abgemildert und ein *Timeout* wird nun wie der *Tod* des jeweiligen Genoms beim Berechnen der *Fitness* behandelt. Dies wird innerhalb der Funktion durch die Variablen $genome_{timeout}$ und $genome_{died}$ behandelt.
Des Weiteren kommt es durch die Vergrößerung der notwendigen *Distanz*, zum Erreichen eines *Checkpoints*, zu einer weiteren Minimierung der erreichbaren *maximalen Fitness* im gewählten *Level*. Durch diese Änderungen kommt es zu einer Anpassung der Skala bei den Diagrammen dieses Durchlaufs, aufgrund der geringeren, maximal

erreichbaren *Fitness*.

$$\begin{aligned} genome_{fitness} = &(5000 * level_{won}) - (750 * genome_{died}) - (750 * genome_{timeout}) \\ &+ (200 * CoinsCollected) + (150 * Checkpoint_{Multiplier}) \\ &- (300 * HP_{lost}) \end{aligned} \tag{4.1}$$

Das *FCNN* hat in Generation 32 eine *maximale Fitness* von 9.500 erreicht, während das *UCNN* seine *maximale Fitness* von 8.125 in Generation 45 erreicht hat. Die geringste Fitness, die von dem besten Genom der Netzwerke erreicht wurde, ist beim *FCNN* 715 in Generation 2 und beim *UCNN* -765 in Generation 0. Innerhalb dieses Durchlaufs erreichte *FCNN* eine maximale Durchschnitts-Fitness von 455 in Generation 32 und 47 und eine minimale Durchschnitts-Fitness von -170 in Generation 1. Das *UCNN* hat eine maximale Durchschnitts-Fitness von 1.100 in Generation 33 und eine minimale Durchschnitts-Fitness von -1.300 in Generation 5 erreicht. Die durchschnittlich erreichte *Fitness* liegt bei dem *FCNN* bei 300 und beim *UCNN* bei 800. Bei der *Speziation* sieht man bei *FCNN*, wie auch bei *UCNN*, dass die Anzahl der Generation in denen eine erhöhte Größe der Spezies zu Beginn des Durchlaufs für bis zu 12 Generationen(*FCNN*), bzw. 9 Generationen(*UCNN*). Des Weiteren bestehen die Generationen beim *FCNN* ab Generation 22 nur noch aus zwei verschiedenen Spezies und ab Generation 38 nur noch aus einer Spezies. Dies ist bei *UCNN* nicht der Fall, da die Zusammensetzung der Generationen über den Verlauf, bis auf die Generationen 42-46, aus mehr als einer Spezies pro Generation besteht.

4.6. Ergebnisse

Bei der Betrachtung der Durchläufe eins bis drei lassen sich keine aussagekräftigen Unterschiede der Fitness feststellen. Lediglich der vierte Durchlauf zeigt eine Tendenz, dass das *UCNN* einen leicht höheren Fitness-Durchschnitt erlangt. Daraus lässt sich ableiten, dass es wie erwartet keine signifikanten Unterschiede der Fitness zwischen *UCNN* und *FCNN* gibt, da diese langfristig auf einen gleichen Wert konvergiert.
Bei der visuellen Observation des *Pinguins* beim *Training* lässt sich über die unterschiedlichen Durchläufe hinweg feststellen, dass beim *FCNN* eine hohe Diversität von Verhaltensmustern der *Genome* vorliegt. Es wurden unterschiedliche Bereiche des *Levels* abgesucht, *Coins* gesammelt und mit *Objekten* interagiert. Es wird vermutet, dass dies an der hohen Anzahl an *Verbindungen* liegt, wodurch erst einmal alle für den Agenten möglichen Aktionen ausgeführt und deren Auswirkungen getestet werden.

Bei dem *UCNN* konnte festgestellt werden, dass es keine hohe Variationsbreite der ausgeführten Aktionen gibt. Sobald der *Agent* gelernt hat, dass Bewegungen nach rechts mit der Kombination von Springen hohe *Fitnesswerte* erzielen, stagniert er in diesem Verhaltensmuster. Bei dem *FCNN* zeigt sich dieses Verhalten ab einem späteren Zeitpunkt ebenso. Es wird daher vermutet, dass bei dem *UCNN* aufgrund der geringen Anzahl an *Verbindungen*, welche erst nach und nach aufgebaut werden, nur diese entstehen, welche direkt zu einer höheren Belohnung führen.
Für den vierten Durchlauf wurden viele Änderungen der Parameter vorgenommen. Auf die Betrachtung bezüglich der Stichhaltigkeit der Hypothese hat dieser Durchlauf jedoch keine Wirkung. Es wird zwar eine leicht bessere Fitness im Durchschnitt bei *UCNN* festgestellt, jedoch müsste dies durch weitere Durchläufe mit denselben Einstellungen überprüft werden.
Bezüglich der Robustheit lässt sich feststellen, dass die Durchläufe eins bis drei keine Unterschiede aufweisen. Bei dem vierten Durchlauf lässt sich hingegen eine höhere Robustheit des *FCNN* erkennen. Die Standardabweichung der Durchschnitts-Fitness ist hier deutlich geringer als die des *UCNN*.
Die Hypothese, dass das *UCNN* schneller als das *FCNN* lernt konnte nicht bestätigt werden. In der Geschwindigkeit des Lernverhaltens konnte kein signifikanter Unterschied festgestellt werden. Es wurde wie bereits beschrieben, lediglich ein Unterschied des Verhaltens des *Agenten* festgestellt. Bezüglich der Robustheit und Effizienz kann aufgrund der geringen Datenmenge keine stichhaltige Aussage getroffen werden.
Abschließend lässt sich feststellen, dass es bezüglich der Vergleichbarkeit der unterschiedlichen Parameter-Einstellungen sinnvoller gewesen wäre, diese nur einzeln zwischen den Durchläufen zu verändern. Des Weiteren wurden zusätzlich zu den Parametern Änderungen am Code vorgenommen. Es lässt sich daher beispielsweise beim vierten Durchlauf nicht ganz klar abgrenzen, worauf die Änderungen im Verhalten des Agenten zurückzuführen sind, da hier zum einen auf $activation\_default = tanh$ und zusätzlich $max\_stagnation = 6$ umgestellt wurde.

Kapitel 5

Fazit

Die Ergebnisse zeigen, dass zwischen *feature-deselective FCNN* und *feature-selective UCNN* keine Unterschiede in der Durchschnitts-Fitness über eine Trainingszeit von 50 Generationen festzustellen sind. Es wird vermutet, dass die Wahl einer längeren Trainingszeit und einer größeren Anzahl an Durchläufen diese Ergebnisse festigen würden. Es ist außerdem zu beobachten, dass innerhalb der ersten *drei Durchläufe* die *maximum average fitness* beim *FCNN* und *UCNN* beginnt auf einen Wert zu konvergieren. Bei dem *vierten Durchlauf* ist dies nicht festzustellen. Diese Beobachtungen könnten im Rahmen weiterer Auswertungen und damit verbundenen weiteren Durchläufen evaluiert werden. Aufgrund der *geringen Menge an Daten*, die, bedingt durch die begrenzte Zeit, innerhalb der Ausarbeitung dieser Arbeit gesammelt worden sind, ist es nicht möglich konkrete und stichhaltige Aussagen über die zu Beginn dieser Arbeit formulierten Hypothesen zu treffen.
Auch im Bezug auf die in der Einleitung formulierte *Hypothese*, dass eine der *initial Topologien* schneller zu einem Ergebnis gelangt, kann keine definitive Aussage getroffen werden, da keine klare *Tendenz* zu beobachten war. Das Fehlen dieser Tendenz ließ sich sowohl bei der *maximum best fitness*, als auch bei der *maximum average fitness*, erkennen. Darüber hinaus konnte festgestellt werden, dass die Wahl der Eingabeparameter eine wichtige Einstellung für das Netzwerk darstellt. Die erste Implementierung 3.6.1 des *Agenten* hatte viele Eingabeparameter, welche zur Laufzeit oftmals nicht genutzt wurden. Diese führten zu einem ungewollten Verhalten des *Agenten* in seiner Umgebung. Es wird vermutet, dass die irrelevanten Eingabeparameter zu einem *Rauschen* in dem Netzwerk geführt haben, wodurch wiederum ungewollte Ausgaben aus dem Netzwerk kommen könnten.
Des Weiteren wurde einleitend eine Aussage über die Robustheit der *initial Topologien* getroffen, dass ein *UCNN* eine robustere Lösung liefert. Auf Basis der vorliegenden Daten kann diese Aussage nicht gestützt werden.
Es kann allerdings nicht ganz ausgeschlossen werden, dass es diese Unterschiede mit der Wahl anderer Parameter geben könnte. Hierzu wäre weitere Forschungsarbeit nötig, die Vergleiche mit den Parametern *max_stagnation*, *species_elitism* sowie

$activation\_default$ aufstellt. Hierbei ist festzustellen, dass aufgrund der Überprüfbarkeit bestenfalls nur ein Parameter zwischen den Durchläufen geändert werden sollte. Änderungen der Fitness sowie des Verhaltens des *Agtenten* sind sonst nicht direkt kausal einer Parameter-Änderung zuweisbar, was die Aussagekraft bezüglich dieser Änderungen erschwert hat.
Weiterhin wäre hierbei der Vergleich zwischen einem *UCNN* und einem *FCNN* mit rekurrenten Verbindungen interessant, da diese Option in dieser Arbeit keine Beachtung gefunden hat. Die Wahl der Aktionen durch den *Agenten* könnten durch seine vorherigen Aktionen beeinflusst werden.
Für die alleinige Realisierung einer *Platformer-Game-AI* könnte der Ansatz eines *klassischen reinforcement learning* in Betracht gezogen werden. Dies könnte in Form eines *state-of-the-art* Algorithmus, wie Proximal Policy Optimization (PPO), realisiert werden. Dadurch könnte in einer weiteren Arbeit ein Vergleich zwischen der Leistung von *NEAT* und *PPO* erörtert werden.
Zusammenfassend lässt sich sagen, dass kein signifikanter Unterschied zwischen *feature-deselective FCNN* und *feature-selective UCNN* festgestellt werden konnte, was die aufgestellte Hypothese stützt. Zu den weiteren genannten Punkten der Hypothese bezüglich unterschiedlicher Robustheit und Lernverhalten lässt sich allerdings keine Aussage treffen.

Literaturverzeichnis

[Avneet Pannu 2015] Avneet Pannu: *Artificial Intelligence and its Application in DifferentAreas*. International Journal of Engineering and Innovative Technology (IJEIT)Volume 4, Issue 10, April 2015. 2015. – Abgerufen am: 13.01.2021

[Circle 2020] Circle, The D.: *ReTux*. `https://retux-game.github.io/`. 2020. – Abgerufen am: 26.12.2020

[Free and Open-Source Software 2020] Free and Open-Source Software: *Wikipedia*. `https://en.wikipedia.org/wiki/Free_and_open-source_software`. 2020. – Abgerufen am: 26.12.2020

[Gerdes u. a. 2013] Gerdes, Ingrid ; Klawonn, Frank ; Kruse, Rudolf: *Evolutionäre Algorithmen - Genetische Algorithmen — Strategien und Optimierungsverfahren — Beispielanwendungen*. Berlin Heidelberg New York : Springer-Verlag, 2013. – ISBN 978-3-322-86839-8

[Kaelbling u. a. 1996] Kaelbling, Leslie P. ; Littman, Michael L. ; Moore, Andrew W.: Reinforcement Learning: A Survey. In: *J. Artif. Int. Res.* 4 (1996), Mai, Nr. 1, S. 237–285. – ISSN 1076-9757

[McIntyre u. a. 2019] McIntyre, Alan ; Kallada, Matt ; Miguel, Cesar G. ; Silva, Carolina F. da: *Neat-Python-Documentation*. `https://neat-python.readthedocs.io/en/latest/config_file.html`. 2019. – Abgerufen am: 29.12.2020

[Nwankpa u. a. 2018] Nwankpa, Chigozie ; Ijomah, Winifred ; Gachagan, Anthony ; Marshall, Stephen: *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018

[Papavasileiou und Jansen 2016] Papavasileiou, E. ; Jansen, B.: A comparison between FS-NEAT and FD-NEAT and an investigation of different initial topologies for a classification task with irrelevant features. In: *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, S. 1–8

[Papavasileiou und Jansen 2017] Papavasileiou, Evgenia ; Jansen, Bart: An Investigation of Topological Choices in FS-NEAT and FD-NEAT on XOR-Based

Problems of Increased Complexity. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA : Association for Computing Machinery, 2017 (GECCO '17), S. 1431–1434. – URL https://doi.org/10.1145/3067695.3082497. – ISBN 9781450349390

[Platform Game 2020] Platform Game: *Wikipedia*. https://en.wikipedia.org/wiki/Platform_game. 2020. – Abgerufen am: 26.12.2020

[Russell und Norvig 2012] Russell, Stuart J. ; Norvig, Peter ; Langenau, Frank (Hrsg.): *Künstliche Intelligenz: Ein moderner Ansatz*. 3., aktualisierte Auflage. München : Pearson, Higher Education, 2012. – ISBN 978-3-86894-098-5

[Shi 2008] Shi, Min: An Empirical Comparison of Evolution and Coevolution for Designing Artificial Neural Network Game Players. In: *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA : Association for Computing Machinery, 2008 (GECCO '08), S. 379–386. – URL https://doi.org/10.1145/1389095.1389164. – ISBN 9781605581309

[Silver, David u. a. 2016] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent ; Driessche, George van den ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Panneershelvam, Veda ; Lanctot, Marc ; Dieleman, Sander ; Grewe, Dominik ; Nham, John ; Kalchbrenner, Nal ; Sutskever, Ilya ; Lillicrap, Timothy ; Leach, Madeleine ; Kavukcuoglu, Koray ; Graepel, Thore ; Hassabis, Demis: Mastering the game of Go with deep neural networks and tree search. In: *Nature 529*, Nature, 2016, S. 484–489. – URL https://doi.org/10.1038/nature16961

[Stanley und Miikkulainen 2002] Stanley, K. O. ; Miikkulainen, R.: Efficient evolution of neural network topologies. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)* Bd. 2, 2002, S. 1757–1762 vol.2

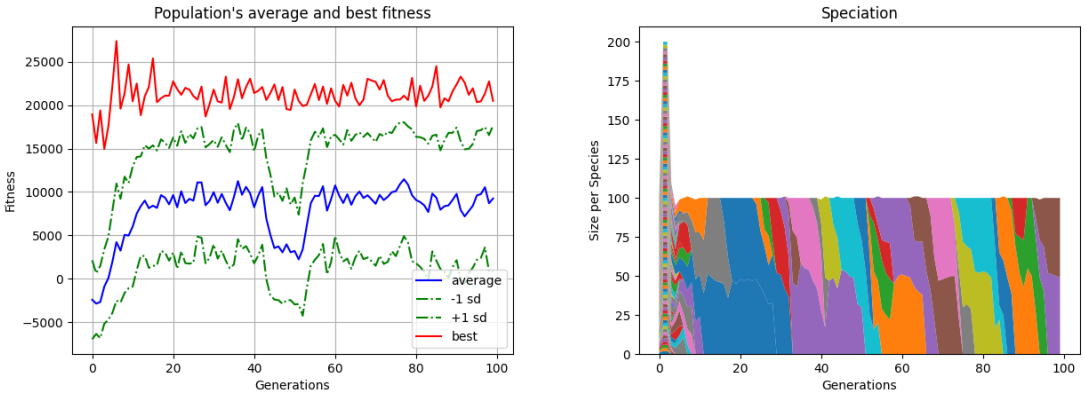
[Stanley 2004] Stanley, Kenneth O.: *Efficient Evolution of Neural Networks Through Complexification*, Department of Computer Sciences, The University of Texas at Austin, Dissertation, 2004. – URL http://nn.cs.utexas.edu/?stanley:phd2004

[Tux 2020] Tux: *Wikipedia*. https://en.wikipedia.org/wiki/Tux_(mascot). 2020. – Abgerufen am: 26.12.2020

[Verma 2020] Verma, Vivek: *Super Mario NEAT*. https://github.com/vivek3141/super-mario-neat. 2020. – Abgerufen am: 26.01.2021

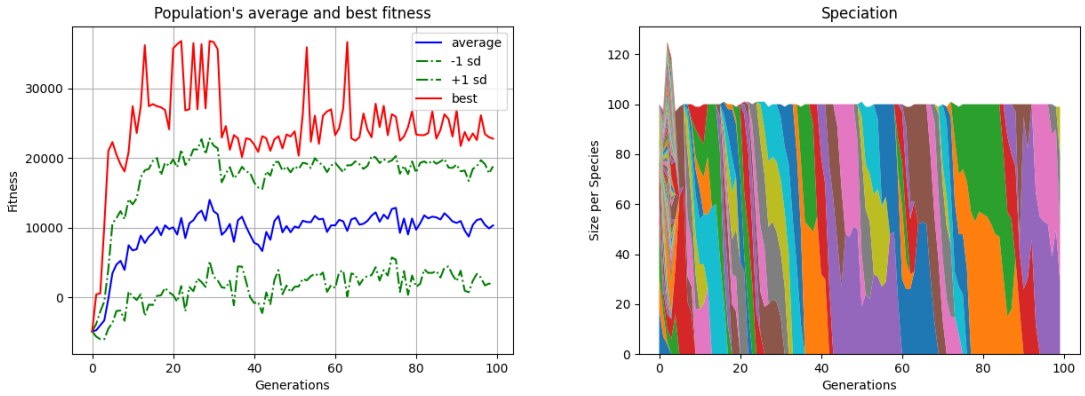
Anhang A

Grafiken



(a) (b)

Abbildung A.1.: 100 Generationen FCNN - Erster Durchlauf

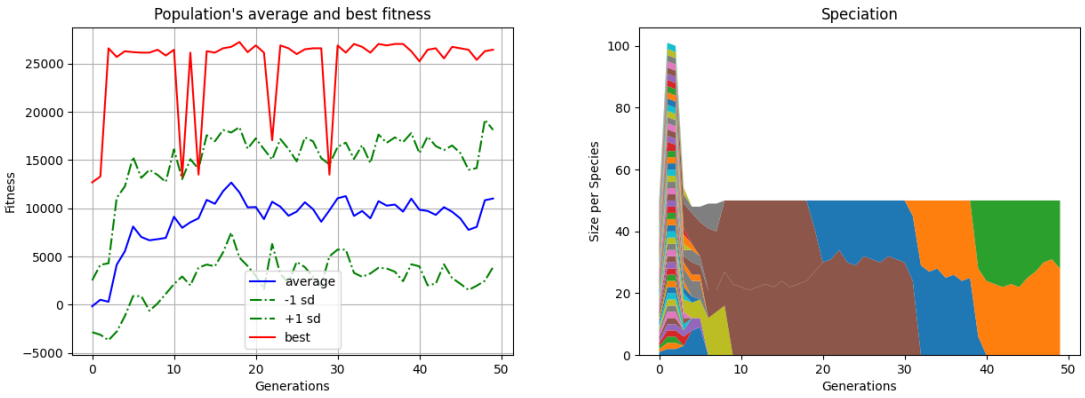


(a) (b)

Abbildung A.2.: 100 Generationen UCNN - Erster Durchlauf

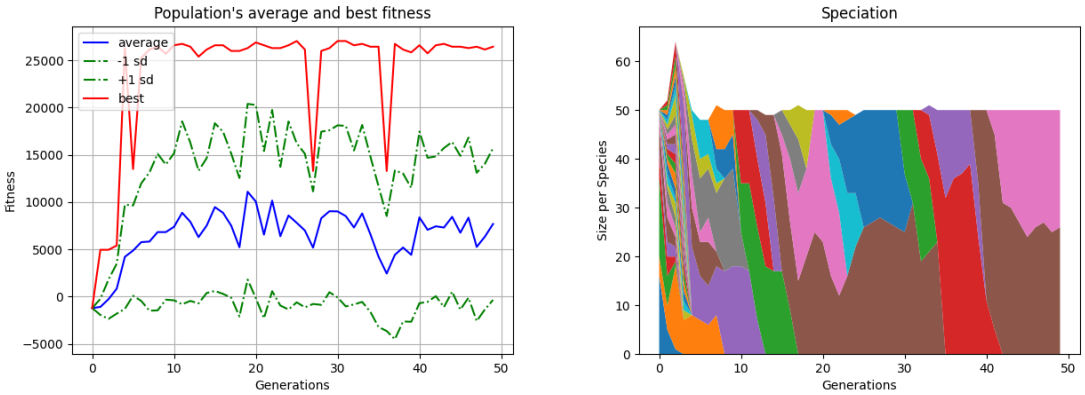
	FCNN	UCNN
Maximum best fitness	27.400 in Generation 6	35.800 in Generation 22, 29, 63
Minimum best fitness	15.000 in Generation 3	-4.840 in Generation 0
Maximum average fitness	11.600 in Generation 75	13.670 in Generation 28
Minimum average fitness	-2.900 in Generation 1	-4.840 in Generation 0
Average	$\sim$ 10.000	$\sim$ 10.000

Tabelle A.1.: Erster Durchlauf



(a) (b)

Abbildung A.3.: 50 Generationen FCNN - Zweiter Durchlauf

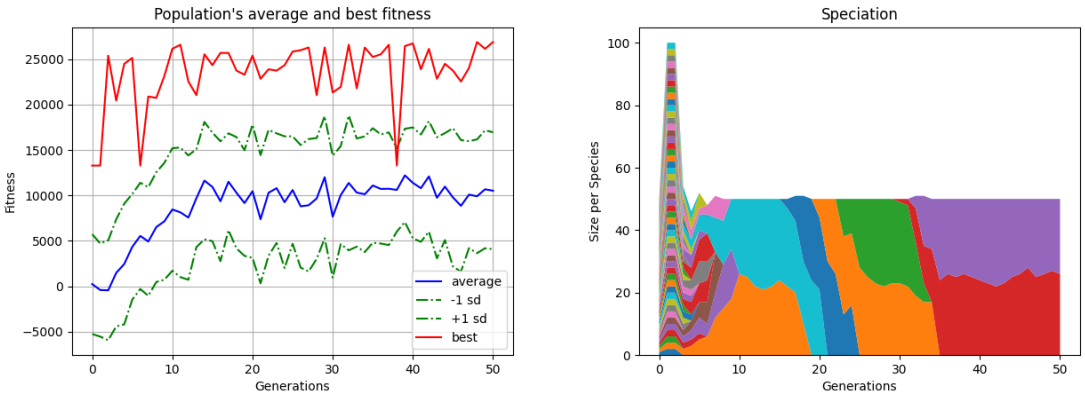


(a) (b)

Abbildung A.4.: 50 Generationen UCNN - Zweiter Durchlauf

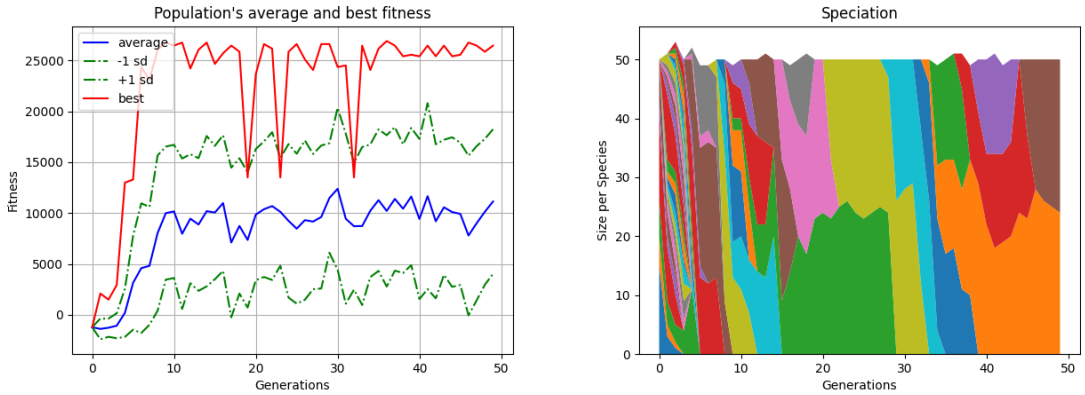
	FCNN	UCNN
Maximum best fitness	27.080 in Generation 18	27.150 in Generation 30
Minimum best fitness	12.480 in Generation 0	- 1.285 in Generation 0
Maximum average fitness	12.570 in Generation 17	11.285 in Generation 19
Minimum average fitness	0 in Generation 0	-1.285 in Generation 0
Average	$\sim$ 10.000	$\sim$ 7.500

Tabelle A.2.: Zweiter Durchlauf



(a) (b)

Abbildung A.5.: 50 Generationen FCNN - Dritter Durchlauf

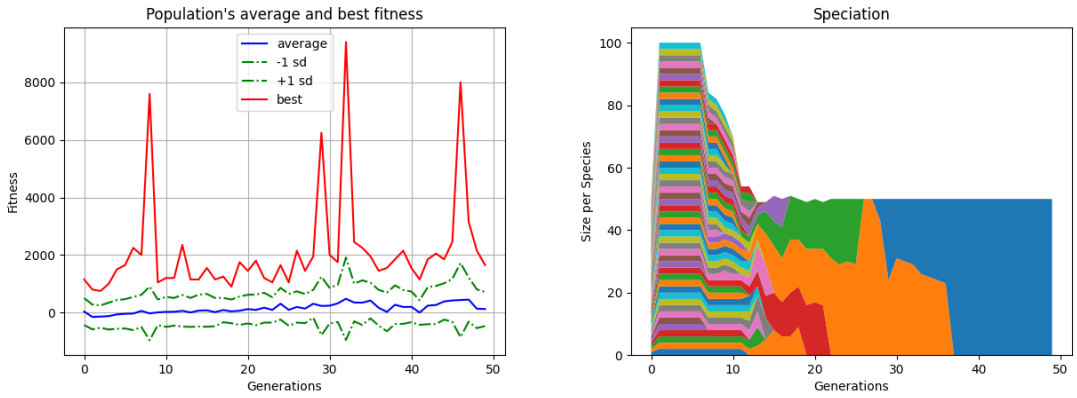


(a) (b)

Abbildung A.6.: 50 Generationen UCNN - Dritter Durchlauf

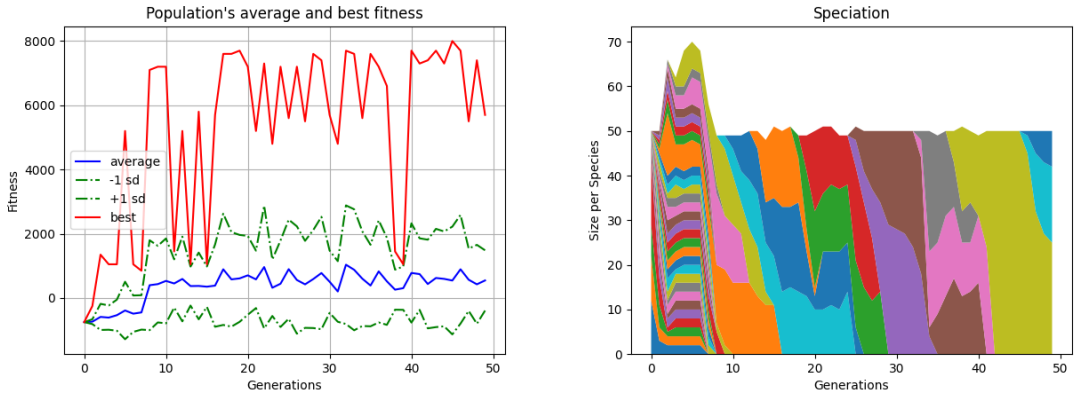
	FCNN	UCNN
Maximum best fitness	27.050 in Generation 48	26.950 in Generation 36
Minimum best fitness	13.430 in Generation 0, 1, 6 und 38	-1.100 in Generation 0
Maximum average fitness	12.180 in Generation 39	12.450 in Generation 30
Minimum average fitness	-470 in Generation 1 und 2	-1.100 in Generation 0
Average	$\sim$ 10.000	$\sim$ 10.000

Tabelle A.3.: Dritter Durchlauf



(a) (b)

Abbildung A.7.: 50 Generationen FCNN - Vierter Durchlauf



(a) (b)

Abbildung A.8.: 50 Generationen UCNN - Vierter Durchlauf

	FCNN	UCNN
Maximum best fitness	9.500 in Generation 32	8.125 in Generation 45
Minimum best fitness	715 in Generation 2	-765 in Generation 0
Maximum average fitness	455 in Generation 32 und 47	1.100 in Generation 33
Minimum average fitness	-170 in Generation 1	-1.300 in Generation 5
Average	~ 300	~ 800

Tabelle A.4.: Vierter Durchlauf

Anhang B

NEAT-Config

NEAT-Config	1. Durchlauf	2. Durchlauf	3. Durchlauf	4. Durchlauf
fitness_criterion	max	max	max	max
fitness_threshold	1000000	1000000	1000000	1000000
pop_size	100	50	50	50
reset_on_extinction	True	True	True	True
activation_default	sigmoid	sigmoid	sigmoid	tanh
activation_mutate_rate	0.00	0.00	0.00	0.00
activation_options	sigmoid	sigmoid	sigmoid	tanh
aggregation_default	sum	sum	sum	sum
aggregation_mutate_rate	0.0	0.0	0.0	0.0
aggregation_options	sum	sum	sum	sum
bias_init_mean	0.0	0.0	0.0	0.0
bias_init_stdev	1.0	1.0	1.0	1.0
bias_max_value	30.0	30.0	30.0	30.0
bias_min_value	-30.0	-30.0	-30.0	-30.0
bias_mutate_power	2.093	2.093	2.093	2.093
bias_mutate_rate	0.1	0.1	0.1	0.1
bias_replace_rate	0.1	0.1	0.1	0.1
compatibility_disjoint_coefficient	2.0	2.0	2.0	2.0
compatibility_weight_coefficient	0.5	0.5	0.5	0.5
conn_add_prob	0.03/0.97	0.03/0.97	0.03/0.97	0.03/0.97
conn_delete_prob	0.97/0.15	0.97/0.15	0.97/0.15	0.97/0.15
enabled_default	True	True	True	True
enabled_mutate_rate	0.01	0.01	0.01	0.01
feed_forward	True	True	True	True
initial_connection	full/uncon	full/uncon	full/uncon	full/uncon
node_add_prob	0.40	0.40	0.40	0.40
node_delete_prob	0.15	0.15	0.15	0.15

num_hidden	10	10	10	10
num_inputs	351	351	351	351
num_outputs	4	4	4	4
response_init_mean	1.0	1.0	1.0	1.0
response_init_stdev	0.0	0.0	0.0	0.0
response_max_value	30.0	30.0	30.0	30.0
response_min_value	-30.0	-30.0	-30.0	-30.0
response_mutate_power	0.1	0.1	0.1	0.1
response_mutate_rate	0.1	0.1	0.1	0.1
response_replace_rate	0.0	0.0	0.0	0.0
weight_init_mean	0.0	0.0	0.0	0.0
weight_init_stdev	1.0	1.0	1.0	1.0
weight_max_value	30	30	30	30
weight_min_value	-30	-30	-30	-30
weight_mutate_power	0.8	0.8	0.8	0.8
weight_mutate_rate	0.40	0.40	0.40	0.40
weight_replace_rate	0.02	0.02	0.02	0.02
compatibility_threshold	3.0	3.0	3.0	3.0
species_fitness_func	max	max	max	max
max_stagnation	2	2	2	6
species_elitism	2	2	2	1
elitism	2	2	2	2
survival_threshold	0.2	0.2	0.2	0.2
min_species_size	2	2	2	2

Tabelle B.1.: NEAT-Config

Erklärung

Hiermit versichern wir, Gruppe 3, dass wir diese Arbeit selbständig verfasst haben und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Außerdem versichern wir, dass wir die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt haben.

Gruppe 3