HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Abschlussarbeit

in

Medieninformatik B. Sc.

Neuroevolution benutzerdefinierter mehrbeiniger Kreaturen

Referent:	Prof. Christoph Müller
Korreferent:	Prof. Dr. Ruxandra Lasowski
Vorgelegt am:	28.02.2023
Vorgelegt von:	Calvin Jirka Friedel Dell'Oro
	263179
	Baumann-Str. 17
	78120 Furtwangen
	calvin.delloro@hs-furtwangen.de

Abstract

Evolutionäre Algorithmen (EA) lösen Probleme ähnlich biologischer Organismen, da genetische Operatoren wie Selektion, Rekombination und Mutation die Evolution durch Reproduktion der stärksten Gene steuern. In dieser Arbeit wird die topologische Flexibilität der EA "Neuroevolution of Augumenting Topologies" (NEAT) genutzt, um mehrbeinige benutzerdefinierte dreidimensionale Kreaturen aus Knochen-Gelenk-Systemen durch evolvierende Genome zu bewegen. Untersucht wird, welche Berechnungsmethode und Parameter-Konfiguration des NEAT-Algorithmus' zu der weitesten Laufdistanz verschiedener Kreaturen führt, ohne Laufmuster vorzugeben. Auch soll die Frage geklärt werden, ob es möglich ist, eine Anwendung zu entwickeln, die ohne weitere Eingriffe des Nutzers zu Laufmustern seiner erstellten Kreaturen führt. Die in Testabläufen gesammelten Daten zeigen die Möglichkeit eines solchen Editors, der Vorteil der Wahrnehmung der Pose, sowie eine starke Inkonsistenz der Daten auf.

Inhaltsverzeichnis

Vorwort ..... **Fehler! Textmarke nicht definiert.**

Abstract III

Inhaltsverzeichnis V

Abbildungsverzeichnis VII

Tabellenverzeichnis ..... **Fehler! Textmarke nicht definiert.**

Abkürzungsverzeichnis IX

1 Einleitung 1

1.1 Ausgangssituation 1

1.2 Zielsetzung 1

2 Grundlagen 3

2.1 Künstliche Neuronale Netzwerke 3

2.2 Neuroevolution 5

2.3 NEAT 6

2.3.0 Mutation 7

2.3.1 Crossing 8

3 Implementierung 9

3.1 Editor 9

3.1.0 State Machine 11

3.1.1 Glied-Docking 12

..... 13

3.1.2 Rotationslimit-Interface 14

3.1.3 Serialisierung 15

3.1.4 Editor-Kreatur zu Trainings-Kreatur 16

3.2 Training 17

3.2.1 NEAT 17

3.2.2 CWCreatureController - Inputs 18
3.2.3 CWCreatureController – Outputs 21
3.2.4 CWCreatureController – Fitness 22
4 Methode 25
4.1 Konfigurationen 25
4.2 Test-Kreaturen 30
4.3 Test-Ablauf 30
..... 31
5 Resultate 32
6 Auswertung und Diskussion 33
7 Ausblick 35
7.1 Didaktisch, Forschend und Unterhaltend 35
7.2 Ansätze für Verbesserungen 35
Fazit ..... **Fehler! Textmarke nicht definiert.**
Literaturverzeichnis 37
Stichwortverzeichnis ..... **Fehler! Textmarke nicht definiert.**
Eidesstattliche Erklärung 39
A. Anhang 41

Abbildungsverzeichnis

Abbildung 1: Auszug aus [3] 3
Abbildung 2: Auszug aus [4] 4
Abbildung 3 Auszug aus [7] 6
Abbildung 4 Auszug aus [7] 7
Abbildung 5 Auszug aus [7] 8
Abbildung 6 Editor Bau-Vorschau 9
Abbildung 7 Editor, Längeneinstellung eines Glieds 9
Abbildung 8 Handles für Einstellung der Rotationslimits 10
Abbildung 9 11
Abbildung 10 12
Abbildung 11 13
Abbildung 12 14
Abbildung 13 15
Abbildung 14 16
Abbildung 15 17
Abbildung 16 18
Abbildung 17 18
Abbildung 18 19
Abbildung 19 20
Abbildung 20 21
Abbildung 21 22
Abbildung 22 25
Abbildung 23 26
Abbildung 24 27
Abbildung 25 28
Abbildung 26 29
Abbildung 27 31
Abbildung 28 32

Abkürzungsverzeichnis

KGS	Knochen-Gelenk-System
NEAT	Neuroevolution of Augumented Topologies
ZMAX	höchste Z-Position, die von der Kreatur am Ende der Simulation erreicht wurde
ANN	Künstliches Neuronales Netzwerk (Artificial Neural Network)
DNN	Tiefes künstliches Neuronales Netzwerk (Deep Artificial Neural Network)

1 Einleitung

1.1 Ausgangssituation

In dieser Thesis sollen mehrbeinige Kreaturen in einer Simulation selbstständig lernen Laufen, ohne dass ihnen Laufmuster gezeigt werden. Die Kreaturen laufen in Intervallen von 20 Sekunden, bevor sie auf ihre Startposition und Startpose zurückgesetzt werden. Entscheidend für die Qualität des Lernens einer Kreatur ist die höchste Z-Position, die von der Kreatur am Ende der Simulation erreicht wurde (im weiteren bezeichnet als).

Im Rahmen dieser Arbeit lernen die Kreaturen durch den genetischen Algorithmus NEAT Laufen. Es werden die Auswirkung auf das Lernverhalten verschiedener Konfigurationen und Parameter des NEAT-Algorithmus‘, die Inputs/Sensoren, (Outputs), Entlinearisierung, Fitness-Funktion und Hyperparameter betreffen, untersucht.

Des Weiteren wird ein Editor implementiert, in dem Kreaturen benutzerdefiniert gebaut und anschließend mit ihnen die Simulation gestartet werden können. Die Kreaturen werden aus Knochen, Gelenken und Rotationslimits gebaut und dadurch definiert. Es können von jedem Glied aus in 25 Richtungen neue Glieder gebaut werden, sowie deren Länge festgelegt werden ebenso können die Rotationslimits auf den jeweiligen X- und Y-Achsen eingestellt oder gänzlich blockiert werden. Es ist möglich sie zu serialisieren und deserialisieren, da sie in einem eigenen Format, dem KGS, gespeichert werden.

1.2 Zielsetzung

Ziel dieser Thesis ist es, herauszufinden, welche Konfigurationen bei mehreren sich signifikant voneinander unterscheidenden Kreaturen im KGS zu dem besten Lernerfolg führt. Dieser wird an dem Durchschnitt der ZMAX über die verschiedenen Kreaturen bestimmt.

Auch soll die Frage beantwortet werden, ob es möglich ist, einen Editor zu realisieren, in dem dreidimensionale Kreaturen benutzerdefiniert erstellt werden können, die ohne weitere Eingriffe des Nutzers lernen zu Laufen.

2 Grundlagen

2.1 Künstliche Neuronale Netzwerke

Künstliche Neuronale Netzwerke (im Folgenden als ANN bezeichnet) sind Systeme, die von der Art der Problemlösung des menschlichen Gehirns inspiriert sind. Wie in seinem künstlichen Replikat, befinden sich im menschlichen Gehirn eine Vielzahl miteinander verbundener Neuronen [1]. Künstliche, digitale Neuronen stellen biologische Neuronen, auch Nervenzellen genannt [2], in der Funktionsweise ihrer Dendriten und Axone nach, indem sie mehrere Eingänge (von nun an als „Inputs“ bezeichnet) mit jeweiligen Gewichten verrechnen und zu einem Ausgang (von nun an als „Output“ bezeichnet) zusammenfassen.

Die Gewichte eines Neurons sind Skalare, die jeweils mit den skalaren Inputs multipliziert und deren Ergebnis summiert wird, um als Ergebnis einen Skalar für weitere Rechnungen zu erhalten. Werden die skalaren Inputs und Gewichte als Vektor betrachtet, stellt das Ergebnis das Skalarprodukt da. Zu diesem wird oft, wie in Abb. 1 zu erkennen, eine gewichtete Konstante als Bias addiert.

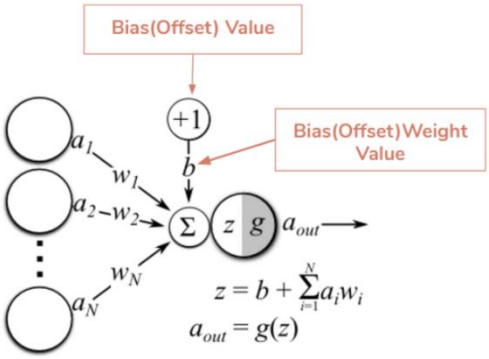


Abbildung 1: Auszug aus [3]

Die Summe des Skalarproduktes und eventuellem Bias wird meistens durch eine nichtlineare Aktivierungsfunktion, wie der Sigmoidfunktion, Logistischen Funktion, oder dem Tangens hyperbolicus, zwischen 0 und 1 oder -1 und 1 limitiert [1]. Das Ergebnis ist der Output des Neurons und in Abb. 2 als a_{out} bezeichnet.

Der Output eines Neurons kann wiederrum einer der Inputs von einem oder mehreren anderen Neuronen sein, wodurch sich ein Neuronales Netzwerk bildet. Dieses hat, wie in Abb. 2 erkennbar, Inputs, welche nicht Outputs anderer Neuronen sind, sondern die Eingaben der bekannten Werte des Problems darstellen. Zu diesem Problem berechnet das Neuronale Netzwerk durch die Vernetzung der Neuronen, die oft auf „Hidden Layers“ stattfindet, und den Gewichten dieser wiederrum Outputs, welche nicht Inputs anderer Neuronen sind, sondern die Lösung des Problems quantifiziert darstellen.

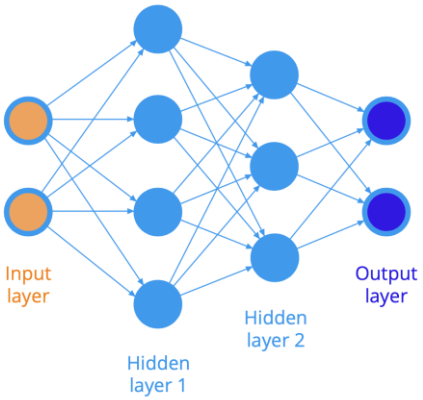


Abbildung 2: Auszug aus [4]

Für das Lernen Neuronaler Netzwerke gibt es verschiedene Ansätze, worunter „Supervised Learning“ das häufigste ist. Es stellt das Vergleichen von erwartetem Output eines Inputs mit dem berechnetem Output eines Inputs und anschließendem manipulieren der Gewichte, um den erwarteten Output näher zu kommen, dar. Wenn es sich allerdings nicht um ein Klassifizierungs- oder Regressionsproblem handelt, da es beispielsweise keine Testdaten gibt, aus denen zu bestimmten Inputs erwartete Outputs hervorgehen, spricht man von sogenannten „Reinforcement Learning“-Aufgaben. Hier eignen sich andere Ansätze besser.

Da bei „Deep Q-Learning“ die Topologie des Netzes, insbesondere der Hidden Layer, sowohl eine große Rolle für den Lern- oder Trainingsprozess spielt, als auch vor Trainingsbeginn festgelegt werden muss und starr bleibt, wird nun in 2.2 der NEAT-Algorithmus beschrieben. Sein flexibles evolvieren der gesamten Topologie während des Trainings kommt hypothetisch der vollkommenen Unbekanntheit der Anzahl an Inputs und Outputs, sowie optimalen Anzahl von Hidden Layers oder Hidden Neuronen, zugute.

2.2 Neuroevolution

Laut Kenneth O. Stanley und seinen Kollegen werden DNNs typischerweise mittels Gradienten basierten Lernalgorithmen namens Backpropagation trainiert. Evolutionäre Strategien könnten mit Backpropagation-basierten Algorithmen, wie Q-Learning und Policy Gradients, um das Bestreiten von RL-Problemen rivalisieren[5].

Evolutionäre Systeme, auch Evolutionäre Algorithmen genannt, orientieren sich bei der Lösung eines Problems an der Evolution realer biologischer Organismen in Populationen, indem mehrere Individuen als mögliche Lösungen für das Problem über mehrere Generationen hinweg evolvieren. Dabei beschreibt die Fitness jedes Individuums quantitativ, wie gut es das Problem löst. Eine höhere Fitness eines Individuums führt zu einer höheren Wahrscheinlichkeit seiner Reproduktion in den nächsten Generationen. Die Evolution wird durch genetische Operatoren wie Selektion**********, bei der eine kleine Anzahl der Individuen mit der höchsten Fitness zur Reproduktion selektiert werden; Crossover, bei dem selektierte Individuen miteinander gekreuzt werden und Mutation, die zufällige Änderungen an Individuen vornimmt. Die Evolution der Individuen wird meistens wiederholt, bis eine maximale Anzahl an Generationen erreicht wurde oder eine andere Bedingung, wie das Vorliegen einer ausreichend guten Lösung, eintritt [6].

2.3 NEAT

Neuroevolution of Augumenting Topologies ist ein von Kenneth O. Stanley und Risto Miikkulainen entwickelter evolutionärer Algorithmus. In NEAT werden Genome als lineare Repräsentation des Aufbaus eines ANNs, definiert durch die Verbindungen von Neuronen, hier Knoten (Nodes) genannt, codiert. Jedes Genom enthält eine Liste von Verbindungsgenen (Connection Genes) und eine Liste von Knotengenen (Node Genes). [7]

Jedes Verbindungsgen bestimmt den Eingangsknoten (In-Node); Ausgangsknoten (Out-Node); das Gewicht, mit dem der Eingang multipliziert wird, einen Status über die Aktivierung (Enabled / Disabled) und eine Innovations-Kennzahl (Innovation Number). [7]

Jedes Knotengen ist eindeutig durch eine Kennzahl bestimmt und gibt Auskunft über die die Rolle im Netzwerk als Sensor, Hidden oder Output Node. [7]

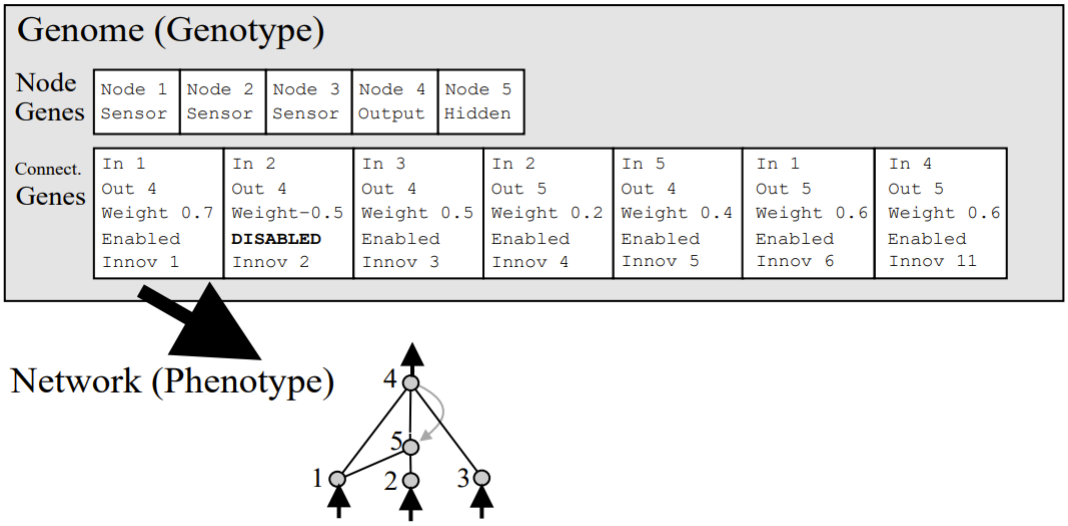


Abbildung 3 Auszug aus [7]

2.3.1 Mutation

In NEAT werden durch Mutation sowohl die Gewichte von Verbindungen, als auch die Netzwerkstruktur mit bestimmten Wahrscheinlichkeiten verändert, oder bleiben unberührt. Diese zwei Arten der Mutation, „Mutate Add Connection“ und „Mutate Add Node“ sind jeweils mit ihren Veränderungen in der Liste von Verbindungsgenen in Abb. 4 zu sehen [7].

Das Hinzufügen einer Verbindung erzeugt ein weiteres Verbindungsgen mit zufälligem Gewicht, hier die Verbindung von Knoten 3 zu Knoten 5 [7].

Das Hinzufügen eines neuen Knotens auf einer Verbindung verursacht das Deaktivieren des alten Verbindungsgens, sowie das Hinzufügen zwei neuer Verbindungsgene, hier das von Knoten 3 zu dem neuen Knoten 6, sowie das von diesem zu dem Knoten 4. In diesem Fall erhält das Verbindungsgen, welches als Output den neuen Knoten hat, ein Gewicht von 1, das Verbindungsgen zwischen dem neuen Knoten und dem Ursprünglichen Output das Gewicht des ursprünglichen Verbindungsgens, um den initialen Effekt der Mutation auf das Verhalten und die Fitness des Individuums zu verringern [7].

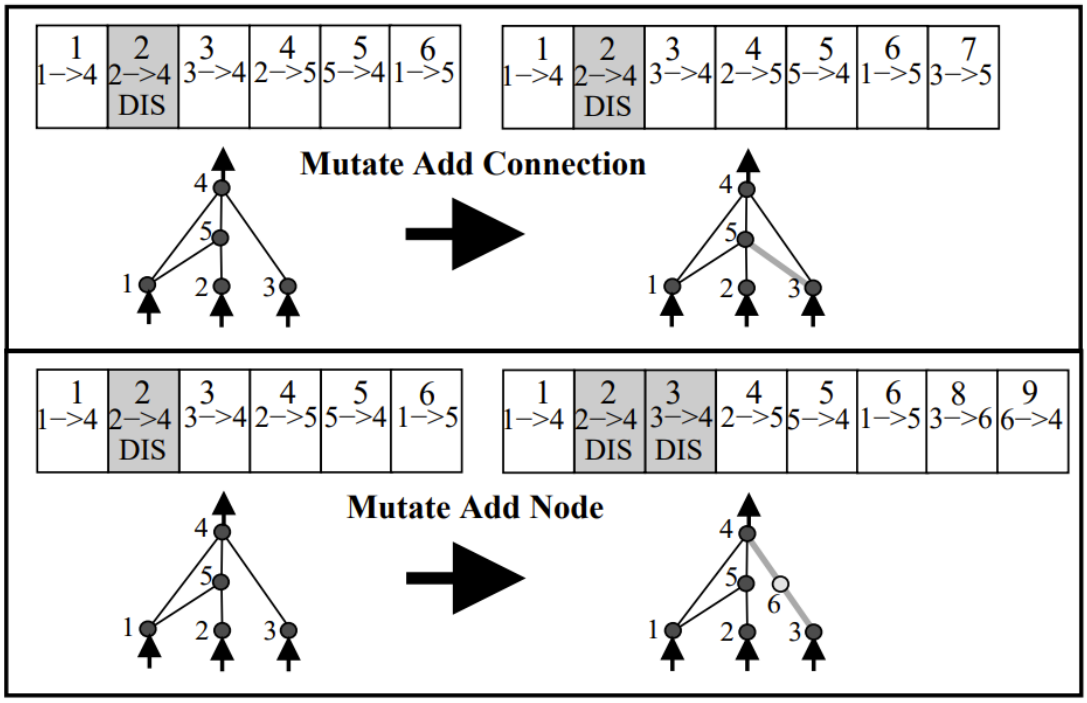


Abbildung 4 Auszug aus [7]

2.3.2 Crossing

Durch die Mutationen werden die Genome allmählich größer und unterscheiden sich immer mehr in ihrem Aufbau, was die Kreuzung erschwert. Kenneth O. Stanley und Risto Miikkulainen lösen dieses Problem, indem sie historische Verläufe von Genen implementieren. Bei jeder Mutation wird eine globale Innovations-Kennzahl erhöht und dem entstandenen Gen zugeordnet [7].

Wie in Abb. 5 erkennbar können mit diesem chronologischen Bezug der Gene im System zwei Genome, die nacheinander mutiert wurden und eine unterschiedliche Anzahl an Hidden Nodes aufweisen, gekreuzt werden. Beispielsweise folgt das Verbindungsgen 8 in der Liste von „Parent1“ auf das Verbindungsgen 5, weshalb bei der Kreuzung die Verbindungs-Gene 6 und 7 von „Parent2“ in „Parent1“ eingefügt werden können, wodurch 5 deaktiviert, ein neuer Knoten eingefügt und verbunden wird [7].

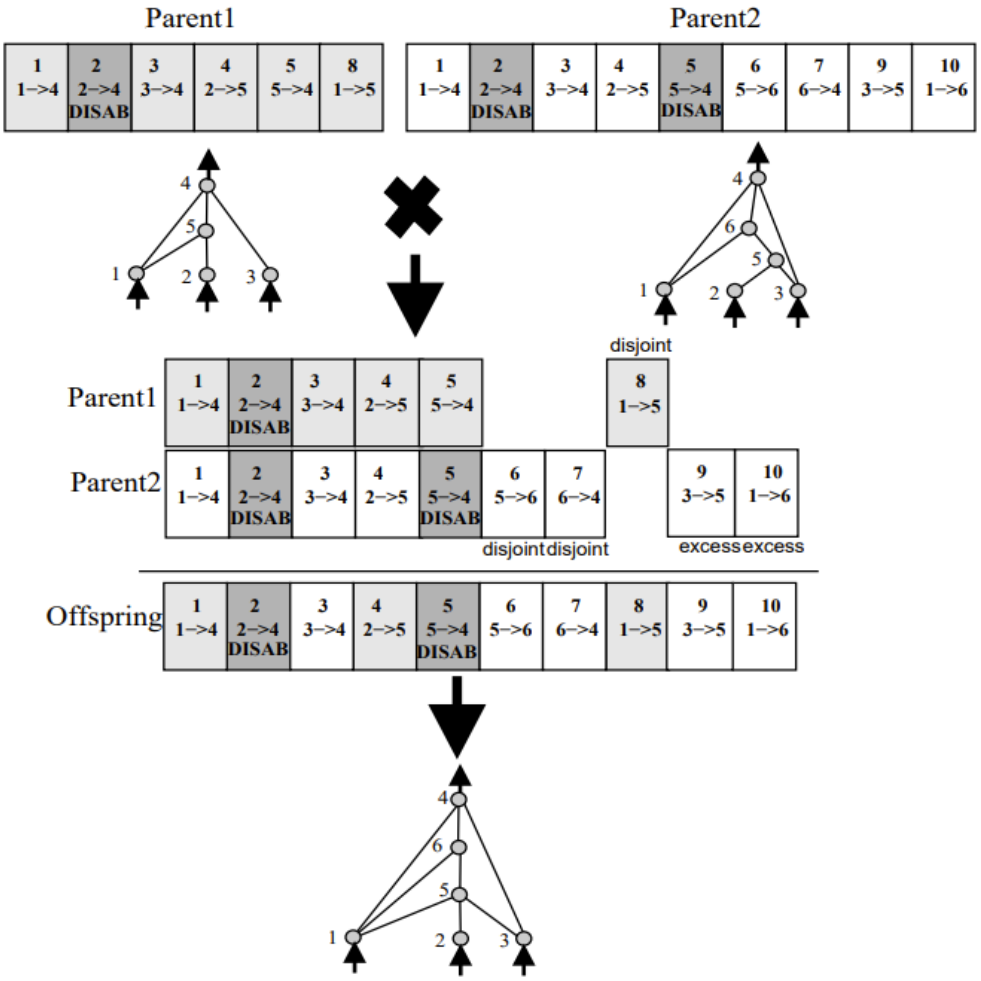


Abbildung 5 Auszug aus [7]

3 Implementierung

Für die Implementierung wurde der Editor und die Game-Engine Unity gewählt, da sie bereits eine stabile Physics Engine enthält, die für diese Arbeit notwendig ist. Programmiert wurde ausschließlich in C#.

3.1 Editor

Im Editor kann der User mit dem Drücken des Mausrades die Kamera um einen Fixpunkt rotieren. Drückt er gleichzeitig die Shift-Taste, verschiebt er die Kamera, sowie den Fixpunkt um die sie sich dreht. Diese Navigationssteuerung ist an die der Modelling-Software Blender angelehnt, in der die Geometrie für die Glieder erstellt wurde.

Der Editor soll es Benutzern ermöglichen, schnell Kreaturen im KGS zu bauen. Das Andocken neuer Glieder an andere soll mit möglichst wenigen Klicks und Komplikationen möglich sein. Schwebt die Maus über einem Glied, wird die Möglichkeit, ein Glied anzudocken, sowie die voraussichtliche Richtung, durch eine transparente Vorschau signalisiert.

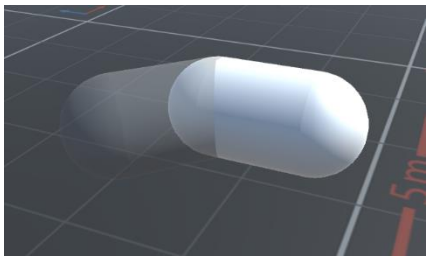


Abbildung 6 Editor Bau-Vorschau

Wird nun die rechte Maustaste betätigt, wird das Glied angedockt und bleibt in einem Modus der Längeneinstellung. Mit der Position des Mauscursors kann nun die Länge des Glieds in festen Schritten eingestellt werden. Das Glied wird final gebaut, wenn die linke Maustaste erneut betätigt wird. Mit der rechten Maustaste kann das Bauen des Glieds

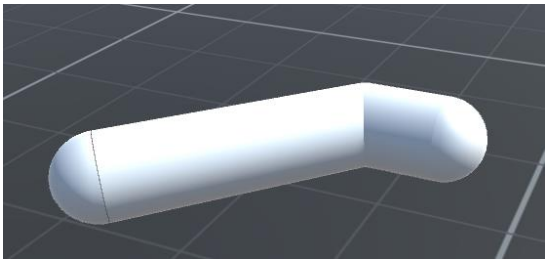


Abbildung 7 Editor, Längeneinstellung eines Glieds

abgebrochen werden. Es wurde sich dafür entschieden, in maximal 25 Richtungen pro Glied zu Bauen, anstatt in beliebig viele, um das Bauen symetrischer Kreaturen zu erleichtern. Dies erhöht die initiale Balance der Kreaturen, und beschleunigt somit das Lernen.

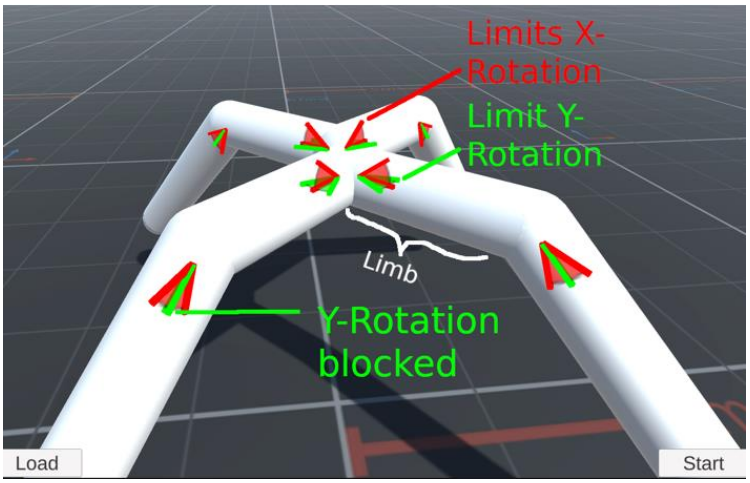


Abbildung 8 Handles für Einstellung der Rotationslimits

Das Einstellen der Rotationslimits soll intuitiv erfolgen, und getrennt von dem Bauen erfolgen, weshalb sich für getrennte Modi entschieden wurde.

Sobald mindestens ein Glied an das Startglied angedockt wurde, kann mit der Taste „R“ in den Rotationsmodus, und wieder zurück gewechselt werden.

Die einzelnen Limits (Abb. 8) können im Rotationsmodus durch „Greifen“ der roten und grünen Handles und anschließendes Bewegen der Maus eingestellt werden.

Durch Drücken des „Save“ Buttons wird die vorliegende Kreatur auf einem neuen Slot gespeichert. Durch Drücken des „Load“ Buttons wird die Kreatur des nächsten Slots geladen. Ein überschreiben der Kreaturen ist momentan nicht implementiert.

Durch Drücken des „Start“ Buttons wird die physikalische Simulation und das Training gestartet.

3.1.1 State Machine

In der Klasse CWEditorController wird in jedem Frame in der Update-Methode eine StateMachine durchlaufen, die die verschiedenen Editor-States (Navigating, BuildingLimb, ViewingRotations, SettingRotations), deren Wechsel und deren Logik steuert (Abb. 9).

```
81      void Update() {
82
83          this.CheckMousePress();
84
85          switch (this.editorState) {
86              case CWEditorState.Navigating:
87
88                  this.freezeCam = false;
89
90                  this.CheckSwitchBuildingLimb();
91                  this.CheckSwitchViewingRotation();
92
93                  this.MoveCam();
94
95                  break;
96              case CWEditorState.BuildingLimb:
97
98                  this.CheckABortBuildingLimb();
99
100                 try {
101                     this.ManageBuildingLimb();
102                 }
103                 catch (NullReferenceException nre) {
104                     Debug.Log("limbBuilding aborted");
105                 }
106
107                 this.MoveCam();
108
109                 break;
110             case CWEditorState.ViewingRotations:
111
112                 this.freezeCam = false;
113
114                 this.CheckSwitchSettingRotation();
115                 this.CheckEndRotationInterface();
116
117                 this.MoveCam();
118
119                 break;
120             case CWEditorState.SettingRotations:
121
122                 this.freezeCam = true;
123
124                 this.UpdateRotationInterface(this.startingLimb);
125                 this.CheckEndRotationInterface();
126
127
128                 break;
129             default:
130                 break;
```

Abbildung 9

3.1.2 Glied-Docking

Die Implementierung des Andockens von Gliedern in die maximal 25 Richtungen wird durch CWDockingBalls realisiert, die in jedem Glied aneinander gereiht und für den Benutzer unsichtbar sind. Sie bestehen aus einem kugelförmigen Collider, der Raycasts der Maus abfängt (Abb. 10), sowie die in Abb. 11 farbig markierten 26 DockingPoints, die lediglich aus Transform Komponenten bestehen, auf jedem CWDockingBall verteilt sind und vom Zentrum weg zeigen. Raycast-Hits auf einem CWDockingBall führen zur Suche des naheliegendste DockingPoints als Starttransformation des neuen Kind-Glieds.

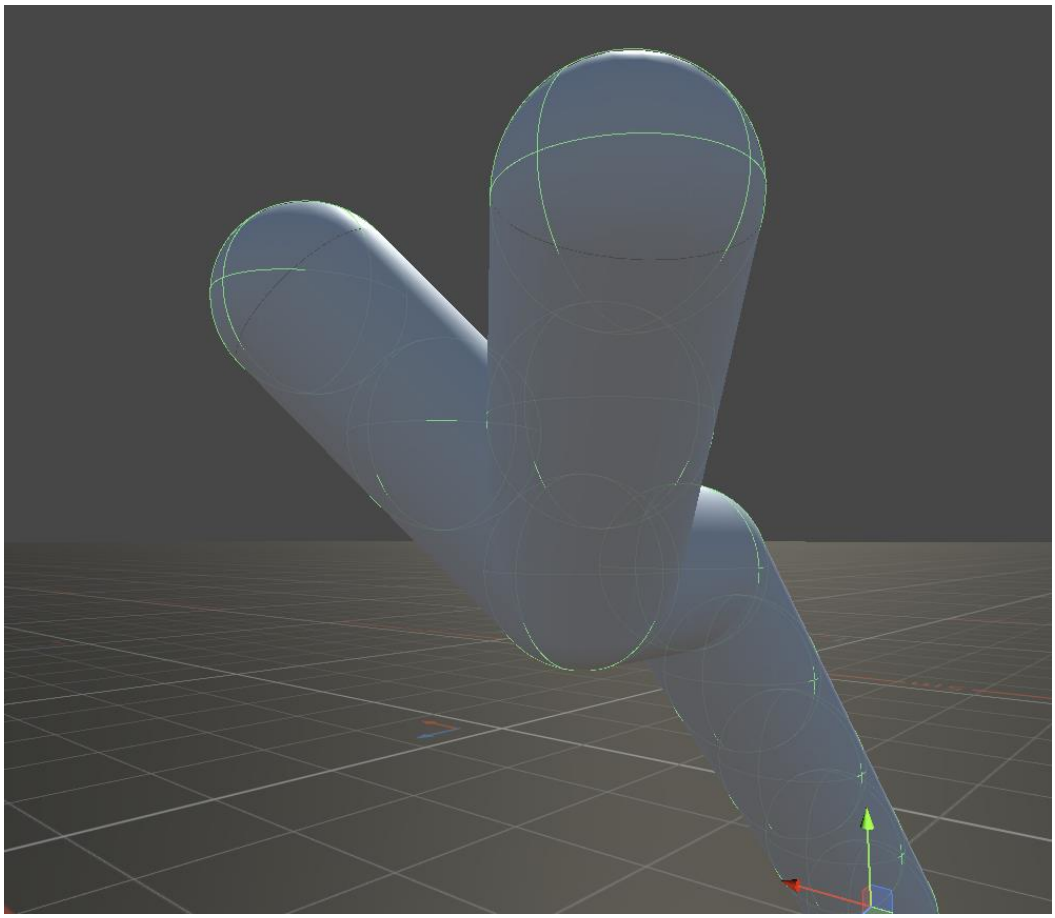


Abbildung 10

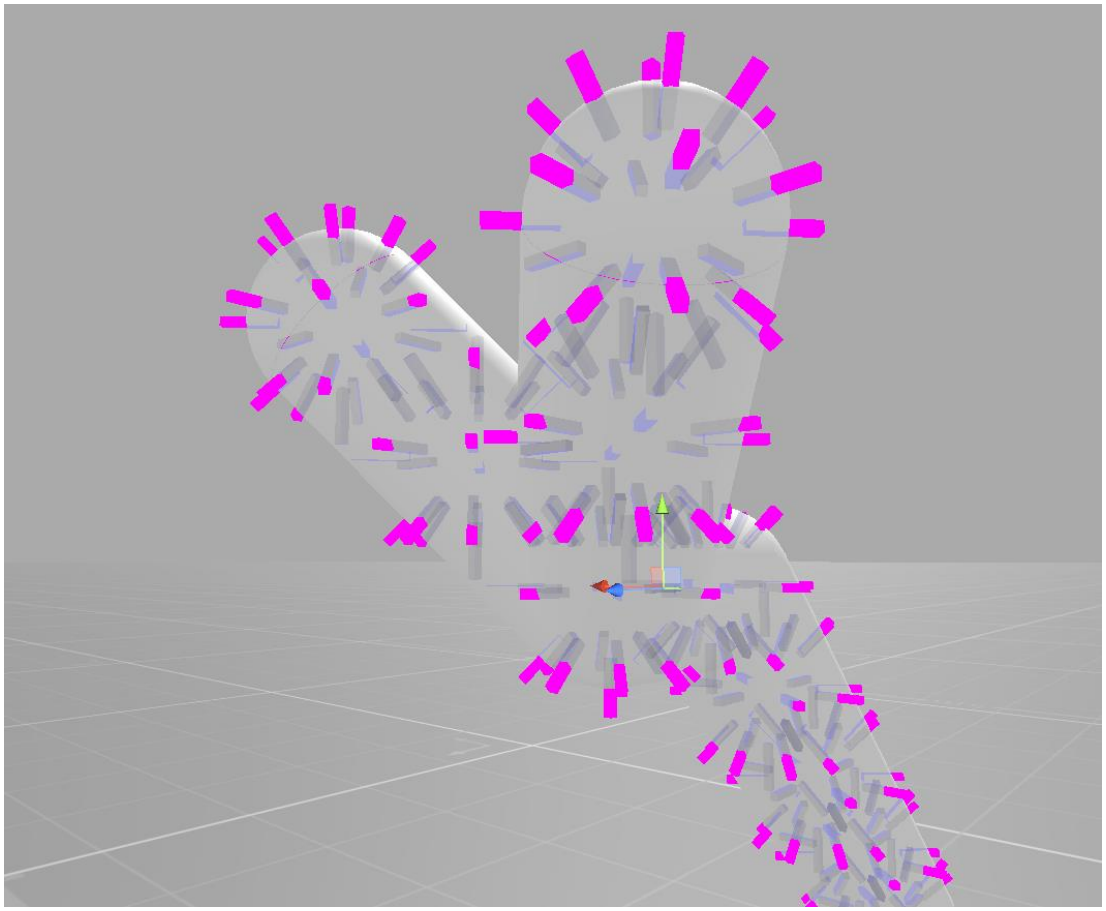


Abbildung 11

3.1.3 Rotationslimit-Interface

Das Einstellen der Rotationslimits (Abb. 12) wurde durch Verwendung eines kleinen Teils des umfangreichen Packages Unity3DRuntimeTransformGizmo [12] realisiert, welches die Transformations-Gizmos des Unity-Editors für Anwendung zur Laufzeit nachbildet. Es umfasst Translation, Rotation und Skalierung sowie viele Einstellungen. Für diese Arbeit wurden einige Stellen im Code geändert, um nur das Rotationstool zu nutzen und die Handles der Achsen auszublenden. Die farbigen Quader als provisorische Handles der X- und Y-Achsenrotationslimits werden durch die Logik dieses Tools gedreht, während das Script CWEditorRotationInterfaceSingleAxis jeweils die Winkel aktualisiert.

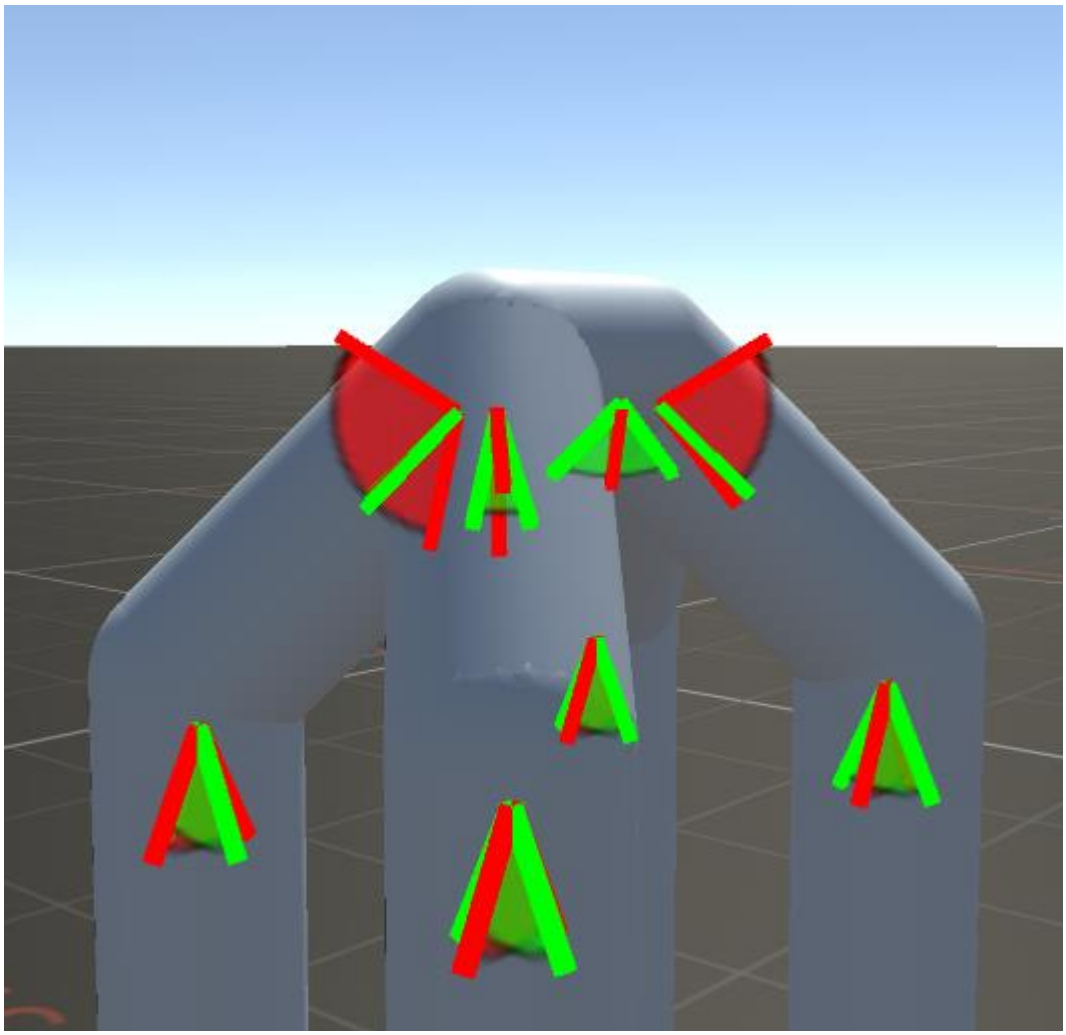


Abbildung 12

3.1.4 Serialisierung

Speichert der Benutzer die Kreatur, wird sie mittels der GetData-Methode, aufgerufen auf dem Start-Glied der Editorszene, serialisiert. GetData gibt zu jedem Glied die zu serialisierenden Daten, zusammengefasst zu einem CWEditorLimbData-Objekt, zurück. Dieses enthät neben der Position, Rotation, DockingBall-Informationen und Rotatoinslimits eine Liste der angedockten Kind-Glieder, die wiederrum CWEditorLimbData-Objekte darstellen, indem auf ihnen Rekursiv GeData ausgeführt wird.

Serialisiert wird im JSON-Format, wobei serialisierte Kreaturen sowie deren im Editor geladene Version KGS-Systeme darstellen.

Die serialisierte JSON eines Gliedes mit einem Kind-Glied, welches wiederrum ein Kind-Glied listet ist in Abb. 13 zu sehen.


```
{
    "position": {
        "x": 0.0,
        "y": 1.7999999523162842,
        "z": 0.0
    },
    "rotation": {
        "x": 0.0,
        "y": 0.0,
        "z": 0.0,
        "w": 1.0
    },
    "dockingBalls": [...
    ],
    "ballDocketAtIndex": 0,
    "isDockedLimb": false,
    "childLimbs": [
        {
            "position": {
                "x": 0.0,
                "y": 1.6939339637756348,
                "z": 0.25646623969078066
            },
            "rotation": {
                "x": 0.3826826512813568,
                "y": 5.7024149846540698e-9,
                "z": -1.3766880968546502e-8,
                "w": 0.923879861831665
            },
            "dockingBalls": [...
            ],
            "ballDocketAtIndex": 1,
            "isDockedLimb": true,
            "childLimbs": [
                {
                    "position": {
                        "x": -1.1102230246251566e-15,
                        "y": 1.3748878240585328,
                        "z": 0.7255134582519531
                    },
                    "rotation": {
                        "x": 0.0,
                        "y": 0.0,
                        "z": 0.0,
                        "w": 1.0
                    },
                    "dockingBalls": [...
                    ],
                    "ballDocketAtIndex": 1,
                    "isDockedLimb": true,
                    "childLimbs": [],
                    "rotationData": {
                        "rotationalFreedom": 3,
                        "minMaxRotationX": {
                            "x": -25.0,
                            "y": 24.999996185302736
                        },
                        "minMaxRotationY": {
                            "x": -25.0,
                            "y": 24.999996185302736
                        }
                    }
                }
            ],
            "rotationData": {
                "rotationalFreedom": 3,
                "minMaxRotationX": {
                    "x": -25.0,
                    "y": 24.999996185302736
                },
                "minMaxRotationY": {
                    "x": -25.0,
                    "y": 24.999996185302736
                }
            }
        }
    ],
    "rotationData": {
        "rotationalFreedom": 3,
        "minMaxRotationX": {
            "x": -25.0,
            "y": 24.999996185302736
        },
        "minMaxRotationY": {
            "x": -25.0,
            "y": 24.999996185302736
        }
    }
}
```


Abbildung 13

3.1.5 Editor-Kreatur zu Trainings-Kreatur

In der Editor-Szene existieren die erstellten Kreaturen nur als visuelle Repräsentation des KGS. Um in der Trainings-Szene physikalisch berechnet zu werden, sind einige Schritte nötig, die initial von der GetPhysicalCreature-Methode der CWEditorPhysicalCreatureFactory aus gesteuert werden, sowie rekursiv pro Glied von der CreateAttachChildLimb-Methode (Abb. 14).

Diese Schritte sind das Instanziieren neuer GameObjekte, die als physikalisch berechenbare Versionen der KGS existieren; das Setzen von Layern; das Transformieren und Herstellen der Objekthierarchie; das Übertragen der Meshes; das Aufsetzen von Collidern, die sich über die ganzen Glieder spannen; sowie das Zuweisen und Einstellen von Rigidbody-, GroundContact- und ConfigurableJoint-Komponenten. Der letzte Schritt ist der rekursive Aufruf der CreateAttachChildLimb-Methode auf allen Kindern eines Glieds.

```
51      private static void CreateAttachChildLimb(CWLimb physicalParentLimb,
52          CWEditorLimb editorChildLimb, bool parentUnderTransform) {
53
54          if (editorChildLimb.isDockedLimb) {
55
56              CWLimb physicalChildLimb = new GameObject("ChildLimb").AddComponent<CWLimb>();
57
58              physicalChildLimb.parent = physicalParentLimb;
59              physicalParentLimb.childLimbs.Add(physicalChildLimb);
60
61              physicalChildLimb.childLimbs = new List<CWLimb>();
62
63              SetupLayer(physicalChildLimb);
64              SetupTransforms(physicalChildLimb, editorChildLimb, physicalParentLimb, parentUnderTransform);
65              SetupMeshes(physicalChildLimb, editorChildLimb);
66              SetupOptimizedColliders(physicalChildLimb, editorChildLimb);
67              SetupRigidybody(physicalChildLimb, 1.5f);
68              SetupGroundContact(physicalChildLimb, false, false);
69
70              SetupConfigurableJoints(physicalChildLimb, editorChildLimb);
71
72              if (editorChildLimb.childLimbs.Count > 0) {
73                  CreateAttachChildLimbs(physicalChildLimb, editorChildLimb.childLimbs, true);
74              }
75          }
76      }
```

Abbildung 14

3.2 Training

3.2.1 NEAT

Für die Implementierung des NEAT-Algorithmus‘ und Berechnungen von Neuronalen Netzwerken wurde auf das Package ANN&NEAT von VirtualStar zurückgegriffen [10].

ANN&NEAT registriert Instanzen der Klasse „CWCreatureBrain“ als „StudentChilds“, welche in 2.2 beschriebenen Individuen oder in 2.3 beschriebenen Genome verkörpern, und steuert anhand eingestellter Hyperparameter die in 2.2 und 2.3 beschriebenen genetischen Operationen. ANN&NEAT verfügt über ein Interface, auf dessen Variable NL (NeatLearning) Hyperparameter eingestellt werden können. So auch, wie in der Klasse CrawlerLearner zu sehen, welche Variable im CWCreatureController die „StudentCrash“, als die Bedingung des Sterbens einer Kreatur, und welche die „StudentLife“, als Fitness (in diesem Package auch „longevity“ genannt) darstellt.

CWCreatureBrain stellt, wie in Abb. 15 erkennbar, die Schnittstelle zwischen dem ANN&NEAT Package, und CWCreatureController dar. Hier wird in jedem Frame der Output des Netzwerks berechnet und dem CWCreatureController übergeben.

```
39          this.Network.Input = this.creatureController.Inputs;
40
41          this.Network.Solution();
42
43          this.creatureController.Outputs = this.Network.Output;
```

Abbildung 15

3.2.2 CWCreatureController - Inputs

Im CWCreatureController befindet sich der Teil der eigenen Implementierung, der die Sensoren der Kreaturen als Inputs berechnet, und die Outputs in Steuerbefehle der Gliedmaßen umsetzt, um die Kreatur zu bewegen. Die einzelnen Gliedmaßen werden mit BodyParts und JointDriveControllern realisiert, die Steuerbefehle von ConfigurableJoints, wie dem Einstellen eines JointDrives, zusammenfassen. BodyParts und JointDriveController, sowie wenige Methoden zum Aufsetzen dieser, wurden aus dem Unity-eigenen Package ML-Agents entnommen [11].

Bei Instanziierung eines CWCreatureController wird die Anzahl an Inputs und Outputs definiert. Diese hängt von der Anzahl beweglicher Rotationsachsen der Gliedmaßen ab, unterscheidet sich also bei verschiedenen im Editor erstellten Kreaturen stark.

Es wird versucht, die Anzahl an Inputs und Outputs minimal zu halten, da größere Netzwerke meistens wesentlich langsamer lernen als kleine. Um dies zu erreichen, werden zunächst nur bewegliche Gliedmaßen mit eigenen Sensoren, und somit eigenen Inputs ausgestattet (Abb.16), sowie Outputs nur je rotierbare Achse angelegt (Abb.17). Somit resultieren effizient aufgebaute Kreaturen in effizienteren Netzwerken.

```
71          int amountMovableBodyParts = 0;
72
73          for (int bodyIndex = 0; bodyIndex < this.bodyParts.Length; bodyIndex++) {
74
75              amountMovableBodyParts +=
76                  this.m_JdController.bodyPartsDict[this.bodyParts[bodyIndex]].rotationalFreedom
77                  == CWRotationInterfaceRotationalFreedom.Nothing
78                  ? 0 : 1;
79          }
80
81          CWCreatureController.inputs = amountMovableBodyParts * 3 + 7;
```

Abbildung 16

```
foreach (ConfigurableJoint joint in joints) {

    if (joint.transform != this.body) {

        tempBodyParts.Add(joint.transform);

        //SetupBodyPart returns the amount of rotatable axis'
        CWCreatureController.outputs += this.m_JdController.SetupBodyPart(joint.transform);
    }
```

Abbildung 17

Unabhängig von dem Aufbau der Kreatur werden sieben Inputs pro Kreatur angelegt (Abb. 16), die als Sensoren des Startglieds, welches in ausnahmslos jeder Kreatur vorhanden ist, fungieren.

In der Funktion InputInputs() (Abb. 18) werden die Inputs auf Variablen der Kreatur, der Umgebung oder dem Bezug der Kreatur zur Umgebung, gesetzt. Die Wahl einiger dieser Werte resultiert biologisch inspirierten oder intuitiven Hypothesen, sie könnten im Training des NEAT Algorithmus‘ zu einer guten Wahrnehmung der momentanen Pose sowie daraus resultierendem Handlungsbedarf führen. Zuerst werden die sieben den Body betreffenden Sensoren verwertet, darunter die Geschwindigkeit, der Winkel der Geschwindigkeit zu der globalen Z-Achse, der Winkel des Bodys selbst zu der globalen Z-Achse sowie Skalarprodukte der lokalen Z-Achse des Bodys mit der globalen Z- und Y-Achse.

```
198     private void InputInputs() {
199
200         Vector3 avgVel = this.GetAvgVelocity();
201
202         this.Inputs[this.sensorIndex++] = (float)Math.Tanh(avgVel.x);
203         this.Inputs[this.sensorIndex++] = (float)Math.Tanh(avgVel.y);
204         this.Inputs[this.sensorIndex++] = (float)Math.Tanh(avgVel.z);
205
206         float angleTweenAvgVAndForward =
207             Vector3.SignedAngle(avgVel, Vector3.forward, Vector3.forward);
208
209         float normalizedAngle = angleTweenAvgVAndForward / 180f;
210         this.Inputs[this.sensorIndex++] = normalizedAngle;
211
212         float angleTweenBodyAndForward =
213             Vector3.SignedAngle(this.body.transform.forward, Vector3.forward, Vector3.forward) / 180f;
214
215         this.angleBodyForward = angleTweenBodyAndForward;
216         this.Inputs[this.sensorIndex++] = angleTweenBodyAndForward;
217
218         this.Inputs[this.sensorIndex++] = Vector3.Dot(Vector3.forward, body.forward);
219         this.Inputs[this.sensorIndex++] = Vector3.Dot(Vector3.up, body.forward);
220
221
222         this.totalCoM = this.GetTotalCoM();
223
224
225         this.CollectObservationBodyPartOptimized(this.m_JdController.bodyPartsDict[this.body]);
226
227         for (int partIndex = 0; partIndex < this.bodyParts.Length; partIndex++) {
228             this.CollectObservationBodyPartOptimized(this.m_JdController.bodyPartsDict[this.bodyParts[partIndex]]);
229         }
230
231         this.sensorIndex = 0;
232     }
```

Abbildung 18

```
270     private void CollectObservationBodyPartOptimized(BodyPart bodyPart) {
271
272
273         if (bodyPart.rb.transform == this.m_JdController.bodyPartsDict[this.body].rb.transform)...
280
281         if (bodyPart.rotationalFreedom == CWRotationInterfaceRotationalFreedom.Nothing) {
282             return;
283         }
284
285         switch (CWTrainingManagerDataCollector.instance.GetCurrentTrainingConfiguration().inputType) {
286
287             case CWTrainingConfiguration.CWTrainingInputType.comDistances:
288
289                 Vector3 distanceToCoM = bodyPart.rb.transform.position - this.totalCoM;
290                 Vector3 sizeRelativeDistanceToCOM = distanceToCoM / this.maxDistanceToCOM;
291
292                 this.Inputs[this.sensorIndex++] = tanH(sizeRelativeDistanceToCOM.x);
293                 this.Inputs[this.sensorIndex++] = tanH(sizeRelativeDistanceToCOM.z);
294
295                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.rb.transform.position.y / this.maxDistanceToCOM);
296
297                 break;
298             case CWTrainingConfiguration.CWTrainingInputType.rotationalFactor:
299
300                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.currentXNormalizedRot);
301                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.currentYNormalizedRot);
302
303                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.rb.transform.position.y / this.maxDistanceToCOM);
304
305                 break;
306         }
307     }
```

Abbildung 19

Es folgt die Eingabe je dreier Werte pro bewegliches Glied durch die Methode CollectObservationBodyPartOptimized (Abb. 19). Zwei der Inputs werden auf Distanzen zu dem Masseschwerpunkt der gesamten Kreatur gesetzt, sowie mit der maximal geschätzten Distanz, die in der Kreatur vorkommen kann, skaliert. Die Ursache hierfür ist die Hypothese, eine Wahrnehmung des Bezugs eines jeden Glieds zu dem Masseschwerpunkt könne den Kreaturen helfen, Genome zu evolvieren, die Umkippen und Stolpern durch bessere Balance vermeiden. Ein weiterer Input erzeugt eine Abhängigkeit zu der Distanz zum Boden, der sich auf der Höhe Y=0 befindet.

Die Codezeilen 298-303 werden genauer in 4 erklärt.

3.2.3 CWCreatureController – Outputs

In der Methode OutputOutputs werden die beweglichen Glieder gemäß Abbildung 20 gesteuert. Die Methode SetJointTargetRotation setzt die Zielrotation der Glieder auf den beweglichen Achsen je nach Wert des zugehörigen Outputs, welcher zwischen -1 und 1 liegt, auf einen Wert zwischen dem benutzerdefinierten Rotationsminimum und -Maximum.

Die Methode SetJointStrength wird in dieser Arbeit immer mit dem Wert 0 auf jedem Glied aufgerufen, da dies einer mittleren Stärke entspricht. Weitere Erläuterungen hierzu folgen in 7.

```
390     public void OutputOutputs() {
391
392         var bpDict = this.m_JdController.bodyPartsDict;
393         int i = 0;
394
395         for (int partIndex = 0; partIndex < this.bodyParts.Length; partIndex++) {
396
397             BodyPart currentBodyPart = bpDict[this.bodyParts[partIndex]];
398
399             switch (currentBodyPart.rotationalFreedom) {
400
401                 case CWRotationInterfaceRotationalFreedom.X:
402
403                     bpDict[this.bodyParts[partIndex]].SetJointTargetRotation(this.Outputs[i++], 0, 0);
404                     break;
405
406                 case CWRotationInterfaceRotationalFreedom.Y:
407
408                     bpDict[this.bodyParts[partIndex]].SetJointTargetRotation(0, this.Outputs[i++], 0);
409                     break;
410
411                 case CWRotationInterfaceRotationalFreedom.XY:
412
413                     bpDict[this.bodyParts[partIndex]].SetJointTargetRotation(this.Outputs[i++], this.Outputs[i++], 0);
414                     break;
415
416                 default:
417                     break;
418             }
419
420             bpDict[this.bodyParts[partIndex]].SetJointStrength(0);
421         }
422     }
423 }
```

Abbildung 20

3.2.4 CWCreatureController – Fitness

Die Methode CalculateFitness berechnet, wie in Abb. 21 erkennbar, im einfachsten Fall die Fitness als Z-Position der Kreatur. Dies geht aus der in 1.2 beschriebenen Zielsetzung hervor. Da Kreaturen mit negativer Fitness, wie im CWCreatureController codiert, sofort sterben, also aus der Simulation entfernt werden, wird auf diesen Wert ein kleiner Bias addiert. Somit haben die Kreaturen in den ersten Frames einen Spielraum, für Sprünge oder Schritte auszuholen.

```
163     private void CalculateFitness() {
164
165         if (CWTrainingManagerDataCollector.instance.GetCurrentTrainingConfiguration().fitnessFunctionType
166             == CWTrainingConfiguration.CWTrainingFitnessFunctionType.alsoPunishX) {
167             this.Fitness += Time.deltaTime;
168         }
169
170         Vector3 avgPosXZ = Vector3.ProjectOnPlane(this.GetAvgPosition(), Vector3.up);
171         Vector3 avgVelXZ = Vector3.ProjectOnPlane(this.GetAvgVelocity(), Vector3.up);
172
173
174         switch (CWTrainingManagerDataCollector.instance.GetCurrentTrainingConfiguration().fitnessFunctionType) {
175
176             case CWTrainingConfiguration.CWTrainingFitnessFunctionType.zPosOnly:
177                 this.Fitness = 1 + avgPosXZ.z;
178                 break;
179
180             case CWTrainingConfiguration.CWTrainingFitnessFunctionType.alsoPunishX:
181
182                 float absAvgPosZ = Math.Abs(avgPosXZ.z);
183
184                 this.Fitness += avgVelXZ.z * Time.deltaTime;
185                 this.Fitness += absAvgPosZ * Time.deltaTime;
186                 this.Fitness -= Math.Abs(avgPosXZ.x) * Mathf.Clamp(absAvgPosZ, 0, 10) * 0.1f * Time.deltaTime;
187
188                 if (avgPosXZ.z < -0.2f) {
189                     this.Fitness -= 10 * Time.deltaTime;
190                 }
191
192                 this.Fitness -= (float)Math.Pow(Math.Abs(this.angleBodyForward), 2) * 10 * Time.deltaTime;
193
194                 break;
195         }
196     }
```

Abbildung 21

Für die Implementierung gibt es an mehreren Stellen verschiedene Konfigurationen. Die Standard-Konfiguration zeichnet den Ausgangszustand aus, die Variations-Konfiguration zeichnet den zu testenden Zustand aus.

- NEAT Genom pro Kreatur-Instanz
- Einige Kreaturen pro Welle
- NEAT Outputs
    - -1 bis 1
    - Standard-Config.:
        - Steuern Zielrotationen pro Glied
            - -1: kleinster möglicher Winkel
            - 1: größter möglicher Winkel
    - Variations-Config:
        - Steuern Zielrotationen pro Glied
            - -1: kleinster möglicher Winkel
            - 1: größter möglicher Winkel
        - Steuern maximale Kräfte pro Glied
- NEAT Inputs
    - Triviale und vorhersehbare Probleme
        - Negative Werte erreichen
        - Wertebereich -1 bis 1 ausschöpfen
        - Lernrelevante Einflüsse kodieren
    - Lösungen in dieser Thesis: Verschiedene Sensoren aus der Simulation
    - Standard-Konfig.:
        - Distanz zum CoM für quantifizierbaren Bezug zur Gewichtsverteilung des gesamten KGS pro Kreatur
    - Variations-Konfig:
        - Faktor der momentanen Rotation für Bezug zur Pose des gesamten KGS pro Kreatur

Bias?

    - Delinearisierung
        - Standard-Konfig.:

        - Tangens Hyperbolikus
    - Variations-Konfig:
        - Keine

- Wellen
    - Nach Dauer einer Welle -> nächste Generation, nächste Welle
    - UI für die MaxWaveTime
- Fitness
    - Standard-Konfig.:
        - Fitness = Z-Position
    - Variations-Konfig:
        - Kontinuierlich Z-Geschwindigkeit auf Fitness addieren
        - Kontinuierlich Z-Position auf Fitness addieren
        - Kontinuierlich X-Position von Fitness abziehen
        - Wenn Z-Pos zu negativ, abziehen
        - Kontinuierlich Winkel zwischen Body und forward von Fitness abziehen

- Hyperparameter
    - Standard-Konfig.:
        - Crossing aus
    - Variations-Konfig:
        - Crossing an

Aktivitätsdiagramm, Klassendiagramm

4 Methode

4.1 Konfigurationen

Konfigurationen werden im Rahmen dieser Arbeit definiert als Konstellation aus verschiedenen Berechnungstypen von Werten, die hypothetisch Einfluss auf den Lernfortschritt der Individuen nehmen. Die Berechnungstypen haben jeweils zwei Zustände und werden definiert als

1. Input Type (Abb. 22)

   Unterschied in der Berechnung der Inputs:

   a. Com Distances – Distanz zum Masseschwerpunkt
   b. Rotational Factor – Normalisierte momentane Rotationen zwischen -1 als minimale, und 1 als maximale Rotation

```
289             case CWTrainingConfiguration.CWTrainingInputType.comDistances:
290
291                 Vector3 distanceToCoM = bodyPart.rb.transform.position - this.totalCoM;
292                 Vector3 sizeRelativeDistanceToCOM = distanceToCoM / this.maxDistanceToCOM;
293
294                 this.Inputs[this.sensorIndex++] = tanH(sizeRelativeDistanceToCOM.x);
295                 this.Inputs[this.sensorIndex++] = tanH(sizeRelativeDistanceToCOM.z);
296
297                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.rb.transform.position.y / this.maxDistanceToCOM);
298
299                 break;
300             case CWTrainingConfiguration.CWTrainingInputType.rotationalFactor:
301
302                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.currentXNormalizedRot);
303                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.currentYNormalizedRot);
304
305                 this.Inputs[this.sensorIndex++] = tanH(bodyPart.rb.transform.position.y / this.maxDistanceToCOM);
306
307                 break;
```

Abbildung 22

Die Idee, die momentane Rotationen der Glieder als Input zu verwerten, wurde von der Methode anderer wissenschaftlicher Arbeiten inspiriert [13][14].

2. Delinearization Type (Abb. 23)

   Unterschied in dem Filter, der auf die Inputs angewendet wird:

   a. Tanh – Tangens Hyperbolikus als nichtlineare Funktion
   b. None – Kein zusätzlicher Filter

```
311     private float tanH(float value) {
312
313         switch (CWTrainingManagerDataCollector.instance.GetCurrentTrainingConfiguration().delinearizationType) {
314
315             case CWTrainingConfiguration.CWTrainingDelinearizationType.none:
316                 return value;
317
318             case CWTrainingConfiguration.CWTrainingDelinearizationType.tanh:
319                 return (float)Math.Tanh(value);
320
321             default:
322                 return value;
323         }
324     }
```

Abbildung 23

Es wird vermutet, dass ein nichtlinearer Filter zu einem besseren Lernfortschritt der Kreaturen führt, da der Wertebereich, falls die maximale Distanz zum Masseschwerpunkt der Kreatur nicht richtig vorhergesehen wurde, in jedem Fall zwischen -1 und +1 liegt. Des Weiteren benutzt die ANN&NEAT Implementierung [10] als Aktivierungsfunktion der Neuronen ebenfalls den Tangens Hyperbolikus, weswegen eine Synergie mit dem NEAT-Algorithmus‘ vermutet wird.

3. Hyperparameter Type (Abb. 24)

   Unterschied eines der Hyperparameter des NEAT-Algorithmus‘:

   a. Crossing Off – Crossing (in 2.3.1 beschrieben) ist deaktiviert
   b. Crossing On – Crossing ist aktiviert

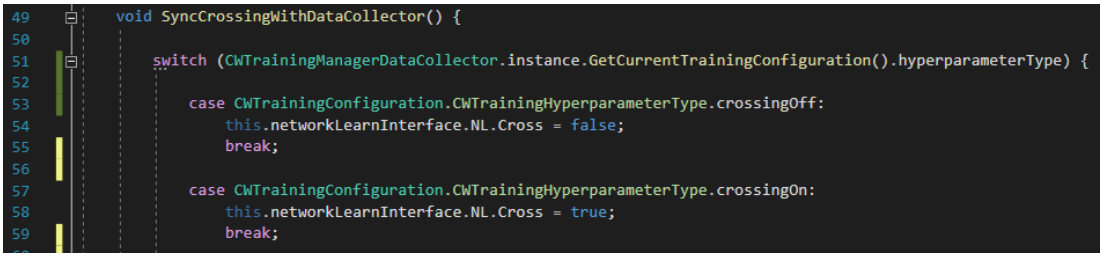


Abbildung 24

Es soll getestet werden, welchen Einfluss deaktiviertes oder aktiviertes Crossing auf das Lernverhalten der benutzerdefinierten Kreaturen hat, da diese Entscheidung in O. Stanleys Analysen von NEAT Einfluss hat.

4. Fitness Function Type (Abb. 25)

   Unterschied in der Berechnung der Fitness:

   a. Z Pos Only – Die Fitness einer Kreatur wird auf die Z-Position + einem kleinen Bias gesetzt

   b. Also Punish X – Auf die Fitness einer Kreatur wird in jedem Frame während einer Welle die Z-Geschwindigkeit sowie die Z-Positoin addiert. Von der Fitness abgezogen wird in jeder Welle der Betrag der X-Position, nachdem er durch die Z-Position skaliert wurde. Auch von der Fitness abgezogen wird das Quadrat des Winkels des Bodys zu der globalen Z-Achse

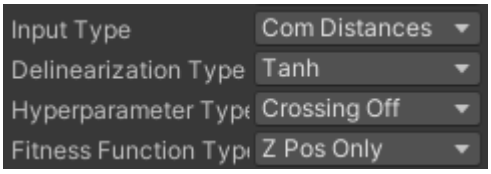
```
174     switch (CWTrainingManagerDataCollector.instance.GetCurrentTrainingConfiguration().fitnessFunctionType) {
175
176         case CWTrainingConfiguration.CWTrainingFitnessFunctionType.zPosOnly:
177             this.Fitness = 1 + avgPosXZ.z;
178             break;
179
180         case CWTrainingConfiguration.CWTrainingFitnessFunctionType.alsoPunishX:
181
182             float absAvgPosZ = Math.Abs(avgPosXZ.z);
183
184             this.Fitness += avgVelXZ.z * Time.deltaTime;
185             this.Fitness += absAvgPosZ * Time.deltaTime;
186             this.Fitness -= Math.Abs(avgPosXZ.x) * Mathf.Clamp(absAvgPosZ, 0, 10) * 0.1f * Time.deltaTime;
187
188             if (avgPosXZ.z < -0.2f) {
189                 this.Fitness -= 10 * Time.deltaTime;
190             }
191
192             this.Fitness -= (float)Math.Pow(Math.Abs(this.angleBodyForward), 2) * 10 * Time.deltaTime;
193
194             break;
195     }
```

Abbildung 25

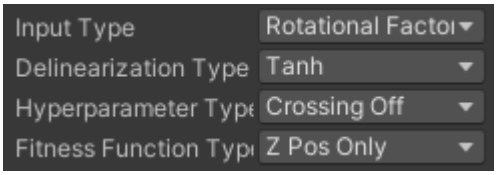
Es wird die Hypothese aufgestellt, dass die Kreaturen höhere ZMAX erreichen, wenn ihre momentane Geschwindigkeit und Abweichung der geraden Bewegung nach vorne Einfluss auf die Fitness hat, da sie hypothetisch besonders in den ersten Sekunden einer Welle präziseres „Feedback“ über den Erfolg ihrer Mutationen bekommen.

Aus diesen vier Berechnungstypen werden fünf Konfigurationen gebildet. In der Default-Konfiguration sind alle Berechnungstypen in ihrem Grundzustand. In den vier weiteren Konfigurationen ist jeweils ein Berechnungstyp in seinem Testzustand, während die anderen im Grundzustand bleiben. Die fünf Konfigurationen sind in Abb. 26 zu sehen.

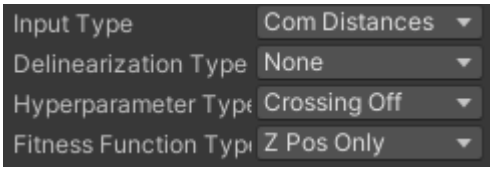
- Default



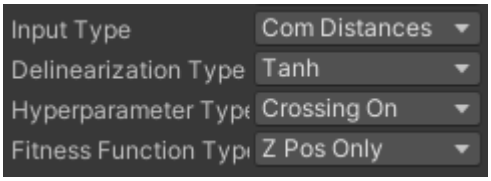
- RotationalFactor



- LinearInputs



- CrossingOn



- AlsoPunishX

Abbildung 26

4.2 Test-Kreaturen

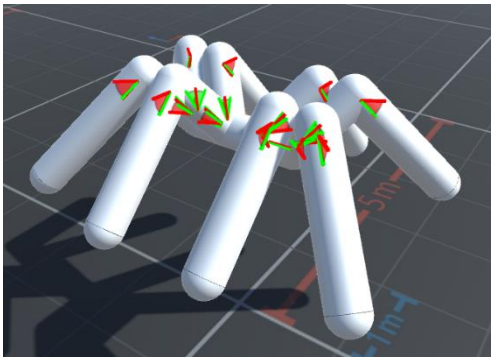
Der Lernerfolg wird an fünf verschiedenen Kreaturen gelistet. Diese sind, mit Ausnahme des Dreibeiners, in ihrer Form und den Rotationslimits an reale Lebewesen angelehnt. Da diese Lebewesen in der Realität bereits Bewegungsmuster evolviert haben, werden sie als sinnvoll für diese Untersuchung angenommen.

Um für eine Variation der Test-Kreaturen zu sorgen, unterscheidet sich die Beinanzahl jeder der Kreaturen. In Abb. 27 sind die Test-Kreaturen und ihre Anzahl an Inputs wie Outputs gelistet.

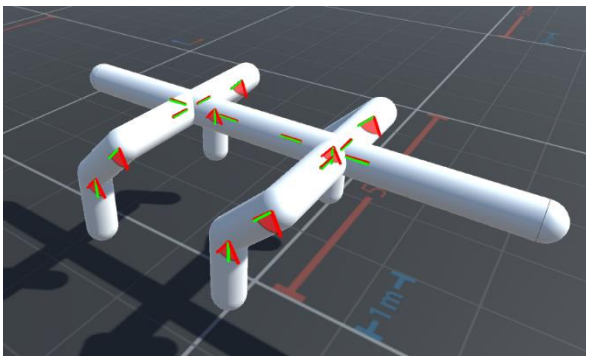
4.3 Test-Ablauf

Der CWTrainingManagerDataCollector testet innerhalb von C#-Coroutinen jede der fünf Konfiguration an jeder der fünf Kreaturen. Pro Kombination aus Konfiguration und Kreatur wird der Lernprozess fünf Mal neu gestartet, um den Zufall durch die zufällige Initialisierung der Genome in der Auswertung ausgleichen zu können. Jeder Lernprozess dauert fünf Minuten, während die einzelnen Wellen, und somit die Abstände zwischen den Mutationen und Generationen, maximal 20 Sekunden andauern. Sterben alle Kreaturen vor Ablauf der maximalen Wellendauer, startet ebenso die neue Generation. Somit können verschiedene Lernprozesse verschiedene Anzahlen an Generationen aufweisen. Am Ende jedes Lernprozesses wird eine Liste der Distanz der Kreatur, die die höchste Z-Max aufwies, sowie das ANN des Genoms gespeichert.

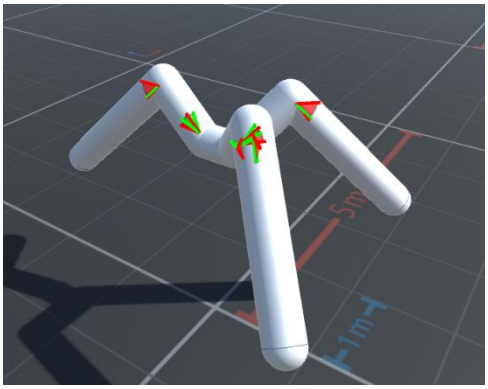
Unabhängig der Konfigurationen werden pro Welle 13 Individuen simuliert. Die Genome starten, wie in O. Stanleys Arbeit unter 3.4 beschrieben [7], ohne Hidden Nodes, während alle Inputs mit allen Outputs verbunden sind. Die Gewichte der Verbindungsgene wird bei dem Start jedes Lernprozesses zufällig bestimmt.



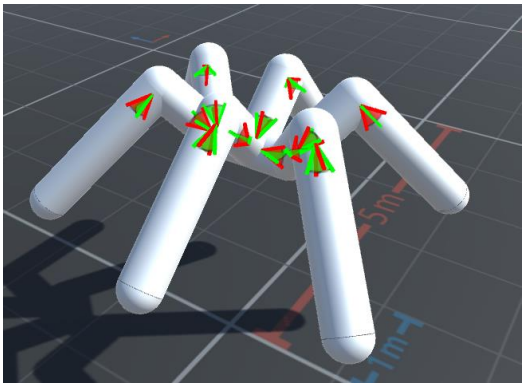
Spinne. In: 55, Out: 25



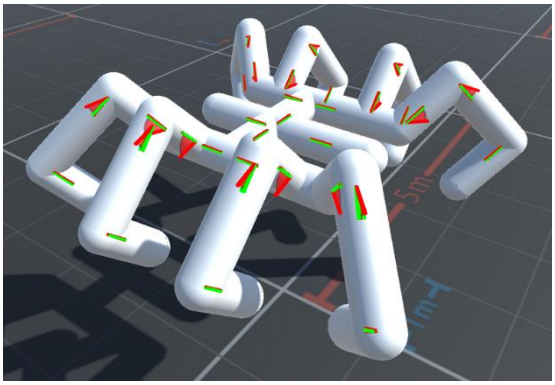
Hund. In: 34, Out: 9



Dreibeiner. In: 25, Out: 9



Käfer. In: 46, Out: 26



Krabbe. In: 55, Out: 18

Abbildung 27

5 Resultate

In Abb. 28 sind die Durchschnitte der ZMAX über die fünft Lernprozesse pro Kreatur bei verschiedenen Konfigurationen graphisch dargestellt.

Der Durchschnitt dieses Wertes über die fünf Kreaturen wird als „Average per Config“ bezeichet, im weiteren APC genannt. Diese sind in Tab. 1 als konkrete Werte aufgelistet

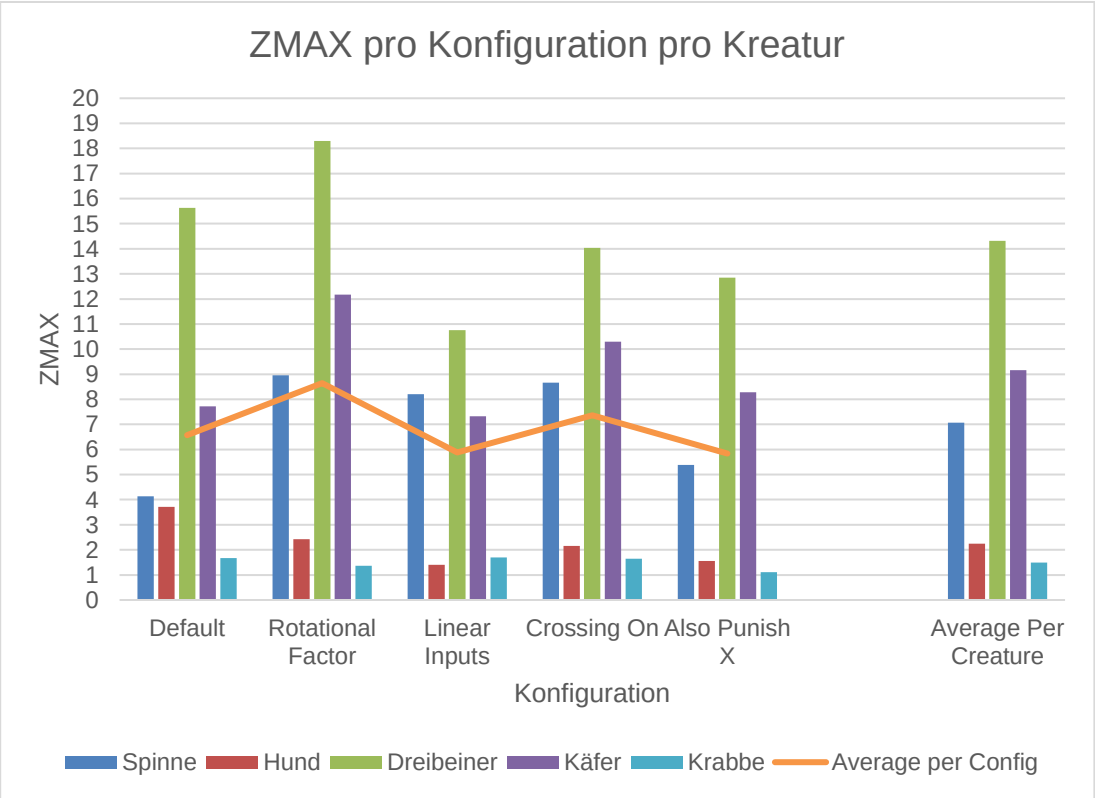


Abbildung 28

	Average per Config
Default	6,571
Rotational Factor	8,645
Linear Inputs	5,877
Crossing On	7,361
Also Punish X	5,838

Tabelle 1

Genauigkeit der Daten, Standardabweichung

6 Auswertung und Diskussion

Die beste APC hatte die Konfiguration “Rotational Factor” mit einem Wert von 8,65, während die Konfigurationen „Linear Inputs“ und „Also Punish X“ mit einem Wert von 5,88 und 5,84 die niedrigsten APC haben. Unter diesen beiden Konfigurationen fällt auf, dass die Kreatur „Spinne“ bei ersterer eine höhere ZMAX als bei letzterer erreicht, während bei der Kreatur „Dreibeiner“ das Umgekehrte der Fall ist.

Die Konfiguration „Rotational Factor“ ist ebenso die Konfiguration, mit der die höchste ZMAX von 18,2 erreicht wurde. Dies erreichte die Kreatur „Dreibeiner“, die auch mit allen anderen Konfigurationen die höchsten ZMAX unter den Kreaturen erreichte. Dies könnte auf die Tatsache zurückzuführen sein, dass sie das kleinste initiale Netzwerk besitzt, sowie auf die standsichere und stabile Bauart.

Ebenso standsicher und stabil ist die Kreatur „Krabbe“ gebaut, die allerdings mit allen Konfigurationen weniger als die Hälfte ihrer eigenen Körpergröße zurücklegte. Dies könnte auf ihr größtes initiale ANN zurückzuführen sein, sowie der Tatsache, dass Krabben in der Natur seitliche Bewegungsmuster evolviert haben, während sie in dieser Arbeit frontal zur Z-Achse ausgerichtet startet.

Wird die „Default“ Konfiguration gegen die „Linear Inputs“-Konfigurationen verglichen, fällt auf, dass die ZMAX der „Spinne“ steigt, die des „Dreibeiners“ jedoch abnimmt. Die verringerte Lernleistung des „Dreibeiners“ kann, insbesondere bei Vergleich der resultierenden Laufmuster, dargestellt in den Videos A.1 („Dreibeiner“ mit „Default“-Konfiguration, sowie A.2 („Dreibeiner“ mit „Linear Inputs“-Konfiguration“) erklärt werden: das fehlen einer nichtlinearen Funktion führt bei dieser Kreatur zu zuckenden Bewegungsmustern, da das Setzen der Zielrotationen der Glieder direkt zu einer Veränderung des Masseschwerpunkts führt, welcher wiederrum linear als Input eingegeben wird.

Auf eine positive Auswirkung der Änderung von „Default“- zu „Linear Inputs“-Konfiguration auf die „Spinne“ kann jedoch nicht geschlossen werden, da sich

die ZMAX der fünf einzelnen Lernprozesse, aus denen in Abb. 28 der Durchschnitt zu sehen ist, zu sehr voneinander unterscheiden.

Dies trifft bei allen Kreaturen zu, insbesondere bei denen, die in manchen Lernprozessen ein vielfaches ihrer Körpergröße zurücklegen. In Tab. 2 ist pro Konfiguration und Kreatur die Differenz der kleinsten und größten ZMAX unter den fünf Lernprozessen dargestellt.

	Spinne	Hund	Dreibeiner	Käfer	Krabbe
Default	7.186	11.255	12.662	5.758	1.503
Rotational Factor	7.617	1.940	12.157	19.339	1.184
Linear Inputs	8.585	0.766	15.860	7.447	2.273
Crossing On	14.563	1.444	11.228	8.397	0.847
Also Punish X	7.656	1.246	18.261	17.230	1.050

Tabelle 2

Auf Grund dieser starken Abweichung sind die ermittelten Daten nicht aussagekräftig genug, um weitere Zusammenhänge mit Laufmustern und Auswirkungen der verschiedenen Konfigurationen sinnvoll untersuchen zu können.

Als Ursache dieser Abweichungen wird unter anderem die geringe Anzahl an simulierten Individuen angenommen, da die Diversität der initialen Genome bei einer Anzahl von 13 nicht ausreicht, um bei jedem Start des Lernprozesses zufällig ein Genom zu generieren, welches durch wenige Mutationen zu dem Erreichen einer ZMAX führt, die signifikant größer als die Körpergröße der Kreatur ist.

7 Ausblick

7.1 Didaktisch, Forschend und Unterhaltend

Mit den Untersuchungen dieser Arbeit wird versucht, der Entwicklung eines dreidimensionalen Kreaturen-Editors und AI-Baukasten näher zu kommen, der sowohl didaktisch, forschend als auch unterhaltend genutzt werden kann. Hierfür wird intensive Forschung in mehreren Dimensionen benötigt.

Diese Arbeit hat gezeigt, dass es möglich ist, in fünf Minuten Laufzeit bestimmte KGS-Kreaturen beim Lernen von Bewegungsmustern zu beobachten, wobei die Zuverlässigkeit sehr gering ist.

7.2 Ansätze für Verbesserungen

7.2.1 Aussagekraft der Daten

Um weitere Untersuchungen am Lernen des KGS-Systems durchführen zu können, müssen die Daten wesentlich aussagekräftiger sein. Unter der in 6 beschriebenen Annahme, führt eine größere Startpopulation zu einer höheren Konsistenz der Daten. Die Startpopulation wurde klein gewählt, da bereits Netzgrößen von Kreaturen wie der „Spinne“ oder „Krabbe“ bei 13 Individuen zu Einbußen in der Performance bei der zur Verfügung stehenden Hardware führt. Es gibt die Möglichkeit, eine Generation in mehreren Wellen zu simulieren, während nur eine Teilmenge der Population auf einmal simuliert wird. Dies führt allerdings auch dazu, dass in gleicher Zeit weniger Generationen evolvieren. Einen Kompromiss zwischen der Größe der gleichzeitig simulierten und der gesamten Population zu finden könnte eine Lösung für dieses Problem darstellen

7.2.2 Lerngeschwindigkeit

In meiner Arbeit ebenso hinderlich für das schnelle Lernen von Bewegungsmustern ist die Einstellung der dynamischen und statischen Reibung auf den Wert 0. Diese Entscheidung resultiert aus gescheiterten Versuchen mit höheren Reibungskoeffizienten, führt aber überwiegend zu Sprung-Bewegungen, und schließt das Simulieren von Kreaturen, die Würmern oder Schlangen ähneln, überwiegend aus.

Vielversprechend ist die Forschung mit erweiterten NEAT-Algorihmen, wie Hyper-NEAT, da Hyper-NEAT im Stande ist, sich geometrische Symmetrien zu Nutze zu machen[15]. Im KGS System kommen diese Symmetrien im Fall ähnlicher Körperteile, wie Beine, vor.

7.2.3 Editor

Auch der Editor kann in vielen Weisen verbessert werden. Eine noch einfachere und praktischere Bedienung wäre möglich, behielten die Gizmos zur Rotationslimit-Einstellung auf dem Bildschirm ihre Größe beim Zoomen bei. Auch die Funktion, Glieder zu löschen, die Position zu ändern, oder zu drehen, sowie ein überarbeitetes User-Interface mit Auswahl-, Überschreib- und Löschfunktion gespeicherter Kreaturen wäre sinnvoll

Literaturverzeichnis

[1] R. E. Uhrig, "Introduction to artificial neural networks," Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics, Orlando, FL, USA, 1995, pp. 33-37 vol.1, doi: 10.1109/IECON.1995.483329.

[2] Larsen R. Nervensystem. Anästhesie und Intensivmedizin für die Fachpflege. 2016 Jun 14:13–25. German. doi: 10.1007/978-3-662-50444-4_2. PMCID: PMC7531560.

[3] https://machine-learning.paperspace.com/wiki/weights-and-biases

[4] https://machinelearningmastery.com/calculus-in-action-neural-networks/

[5] „Deep artificial neural networks (DNNs) are typically trained via gradient-based learning algorithms, namely backpropagation. Evolution strategies (ES) can rival backprop-based algorithms such as Q-learning and policy gradients on challenging deep reinforcement learning (RL) problems.“ (Übersetzung durch Calvin Dell'Oro) Such, Felipe & Madhavan, Vashisht & Conti, Edoardo & Lehman, Joel & Stanley, Kenneth & Clune, Jeff. (2017). Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning.

[6] E. Galván and P. Mooney, "Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges," in IEEE Transactions on Artificial Intelligence, vol. 2, no. 6, pp. 476-493, Dec. 2021, doi: 10.1109/TAI.2021.3067574.

[7] Kenneth O. Stanley, Risto Miikkulainen; Evolving Neural Networks through Augmenting Topologies. Evol Comput 2002; 10 (2): 99–127. doi: https://doi.org/10.1162/106365602320169811

[8] Blender https://www.blender.org/

[9] Unity Platform https://unity.com/products/unity-platform

[10] VirtualSTAR, ANN & NEAT Addon, Unity Asset Store https://assetstore.unity.com/packages/tools/ai/ann-neat-138940

[11] MLAgents, Unity https://docs.unity3d.com/Manual/com.unity.ml-agents.html

[12] Hidden Monk, Unity3DRuntimeTransformGizmo Package https://github.com/HiddenMonk/Unity3DRuntimeTransformGizmo

[13] Ben Jackson, Alastair Channon; July 29–August 2, 2019. "Neuroevolution of Humanoids that Walk Further and Faster with Robust Gaits." Proceedings of the ALIFE 2019: The 2019 Conference on Artificial Life. ALIFE 2019: The 2019 Conference on Artificial Life. Online. (pp. pp. 543-550). ASME. https://doi.org/10.1162/isal_a_00219

[14] Vinod K. Valsalam and Risto Miikkulainen. 2008. Modular neuroevolution for multilegged locomotion. In Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO '08). Association for Computing Machinery, New York, NY, USA, 265–272. https://doi.org/10.1145/1389095.1389136

[15] J. Clune, B. E. Beckmann, C. Ofria and R. T. Pennock, "Evolving coordinated quadruped gaits with the HyperNEAT generative encoding," 2009 IEEE Congress on Evolutionary Computation, Trondheim, Norway, 2009, pp. 2764-2771, doi: 10.1109/CEC.2009.4983289

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Thesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben. Der Einsatz von KI-Anwendungen ist dem betreffenden Thesisteil, der Art sowie dem Umfang nach detailliert benannt.

_____

Furtwangen, 28.02.2023, Calvin Dell‘Oro

A. Anhang

A.1

DefaultInputsDreibeiner.mp4

A.2

LinearInputsDreibeiner.mp4