HAW HAMBURG

BACHELORTHESIS
Daniel Kessener

Optimierung rekurrenter neuronaler Netze durch genetische Algorithmen der NEAT-Familie

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Daniel Kessener

Optimierung rekurrenter neuronaler Netze durch genetische Algorithmen der NEAT-Familie

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Informatik Technischer Systeme
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Stephan Pareigis

Eingereicht am: 06. Dezember 2019

Daniel Kessener

Thema der Arbeit

Optimierung rekurrenter neuronaler Netze durch genetische Algorithmen der NEAT-Familie

Stichworte

RNN, LSTM, GRU-MB, NEAT, HyperNEAT, ES-HyperNEAT

Kurzzusammenfassung

In dieser Arbeit wird die Eignung verschiedener *Genetischer Algorithmen* der NEAT-Familie für die Optimierung rekurrenter neuronaler Netze untersucht. Dabei werden konkret klassisches NEAT und ES-HyperNEAT in Augenschein genommen. Beide GAs werden mit verschiedenen RNN-Architekturen kombiniert. Konkret werden *Long Short-Term Memory* und *Gated Recurrent Units with Memory Block* verwendet. Es werden drei Untersuchungen mit verschiedenen Komplexitätsgraden und Ansprüche an das Erinnerungsvermögen der Agenten durchgeführt, die zeigen, dass GAs sich grundsätzlich gut zum Optimieren von RNNs eignen.

Daniel Kessener

Title of Thesis

Optimization of recurrent neural networks by genetic algorithms from the NEAT family

Keywords

RNN, LSTM, GRU-MB, NEAT, HyperNEAT, ES-HyperNEAT

Abstract

This paper tests how suited different *Genetic Algorithms* from the NEAT-family of GAs are to optimize recurrent artificial neural networks. Specifically this paper looks at classic NEAT and ES-HyperNEAT. Both GAs are combined with different RNN architectures - specifically *Long Short-Term Memory* and *Gated Recurrent Units with Memory Block* - and subjected to three tests of varying complexities and demands on the memory of the agents. It can be shown that GAs are able to optimize RNN reasonably well.

Inhaltsverzeichnis

1 Einleitung **1**

2 Verwendete Technologien **2**
2.1 Genetische Algorithmen 2
2.1.1 NeuroEvolution through Augmenting Topologies (NEAT) 3
2.1.2 Hypercube-based NEAT (HyperNEAT) 5
2.1.3 Evolvable Substrate HyperNEAT (ES-HyperNEAT) 8
2.2 Modelle rekurrenter neuronaler Netze 12
2.2.1 Einfaches Recurrent Neural Network (RNN) 12
2.2.2 Long Short Term Memory (LSTM) 13
2.2.3 Gated Recurrent Unit with Memory Block (GRU-MB) 14

3 Untersuchungen **16**
3.1 Simple Maze Navigation 16
3.2 Counting 21
3.3 Sequence Recall 25

4 Fazit **30**

Literaturverzeichnis **32**

A Hyper-Parameter **34**
A.1 NEAT 34
A.2 ES-HyperNEAT 34

Selbstständigkeitserklärung **35**

1 Einleitung

Von Apple's Siri über Google Assistant zu Amazon Echo sind *Virtual Assistants* aus dem Alltag vieler Menschen nicht mehr wegzudenken. Essentieller Bestandteil der Zugänglichkeit all dieser VAs steht deren Fähigkeit gesprochene Sprache verstehen und replizieren zu können. Rekurrente Neuronale Netze (RNN) sind neuronale Netze (NN) mit rekurrenten Verbindungen. Dies erlaubt es RNNs neben dem aktuellen Input auch auf Informationen aus vorigen Zeitschritten zuzugreifen - sie haben ein Gedächtnis. Dies eignet RNNs besonders für Aufgaben, bei denen ein Agent mit sequenziellen Informationen umgehen können muss, wie z.B. dem Erkennen von Sprache oder Handschrift [13] [8] [6].

Eins der beliebtesten Trainigsverfahren für NN ist *Gradient Descent*, bei dem die Gewichte des Netzes durch Ermitteln der partiellen Ableitungen der einzelnen Netzwerkschichten ausgehend vom Output "rückwärts" angepasst werden. Ein Problem dieses - und ähnlicher - Verfahren ist die multiplikative Natur von *Backpropagation*: Der Gradient, der die Gewichtsänderung vorgibt, wird aus dem Produkt der partiellen Ableitung einer Schicht mit dem Gradienten der nachfolgenden Schicht gebildet. Sind die Ableitungen recht klein, dann strebt dieses Produkt bei n Netzwerkschichten gegen 0 - die Gewichte ändern sich nicht mehr. Dies ist bei Aktivierungsfunktionen wie der *Sigmoid*-Funktion σ oder tanh der Fall. Sind die Ableitungen dagegen groß, wächst der Gradient exponentiell - die Gewichte verändern sich sprunghaft und erratisch - sie konvergieren nicht mehr. Dies kommt z.B. bei der *Rectified Linear Unit* (ReLU) vor. Diese Problematik wird daher *Vanishing Gradient Problem* bzw. *Exploding Gradient Problem* [3] [7] genannt. Durch ihre rekurrenten Verbindungen sind RNNs besonders stark von diesem Problem betroffen, was in der Praxis das Training solcher NN sehr aufwändig macht.

Als Lösung dieser Problematik bieten sich alternative Trainingsverfahren wie *Genetische Algorithmen* an. Nach dem Vorbild der biologischen Evolution nutzen GAs eine Kombination aus zufälliger Mutation und Selektion um NN zu trainieren. Da hierbei keine Gradienten berechnet werden müssen, sind GAs immun gegenüber dem *Exploding/Vanishing Gradient Problem*.

In dieser Arbeit soll die grundsätzliche Fähigkeit von GAs zur Optimierung RNNs erkundet werden. Dabei wird der GA *NeuroEvolution through Augmenting Topologies* (NEAT) und der darauf aufbauende GA ES-HyperNEAT in Kombination mit verschiedenen RNN-Architekturen - konkret *Long Short-Term Memory* (LSTM) und *Gated Recurrent Unit with Memory Block* (GRU-MB) - untersucht.

2 Verwendete Technologien

2.1 Genetische Algorithmen

Genetische Algorithmen [9] sind iterative, heuristische Optimierungsverfahren, die sich auf die Mechaniken der biologischen Evolution stützen. Eine Menge von Agenten (*Population*) wird hierbei zuerst mittels einer Fitness-Funktion evaluiert. Danach pflanzen sich die Agenten fort und bilden so die Nachfolgepopulation. Dabei wird durch die berechnete Fitness der Agenten Selektionsdruck ausgeübt: "fittere" Agenten pflanzen sich öfter fort als Agenten mit weniger Fitness. Fortpflanzung erfolgt typischerweise auf zwei Arten: Asexuelle Reproduktion, wobei ein Agent durch kleine Veränderungen (*Mutation*) aus seinem singulärem Elternteil hervorgeht, und sexuelle Reproduktion, wobei ein Agent stückweise aus Teilen seiner Eltern zusammengesetzt wird (*Cross-Over*). Durch das Wiederholen dieser zwei Schritte - Evaluierung und Reproduktion - wird eine Reihe von Populationen (*Generationen*) erzeugt, deren durchschnittliche Fitness ein lokales Maximum der Fitness-Funktion anstrebt.

Viele GAs nutzen *Encodings*. Encodings orientieren sich ebenfalls an einem biologischen Vorbild: biologische Agenten (Lebewesen) erzeugen Nachkommen nicht direkt. Stattdessen ist die essentielle Struktur eines Agenten in einem Encoding gespeichert, z.B. DNA. Mutation und Cross-Over wird auf Basis der Encodings der Eltern durchgeführt. Der eigentliche Agent wird in einem seperaten Schritt aus seinem Encoding erzeugt. Ein Encoding bildet eine Abstraktionsschicht zwischen genetischen Operationen (wie Mutation oder Cross-Over) und der Funktionalität des Agenten. Dies bietet die Möglichkeit ein Encoding zu wählen, das sich gut für genetische Operationen eignet, ohne Gefahr zu laufen die Funktionalität des Agenten zu beeinträchtigen. Im Weiteren lässt sich Symmetrie und Wiederhohlungen in der Topologie des Agenten in einem Encoding effizienter beschreiben. Dies erlaubt es kleineren genetischen Veränderungen im Encoding signifikant die Struktur des Agenten zu beeinflussen. Auch dies sieht man in der Natur: Das menschliche Genom beeinhaltet ca. 50.000 Gene, während das menschliche Gehirn ca. 100.000.000 Neuronen enthält. Ein Encoding, das 1-zu-1 die Topologie des Agenten beschreibt wird *Direct Encoding* genannt. Es kann Symmetrie und Regelmässigkeit im Agenten nicht gesondert beschreiben. Im Gegensatz dazu wird ein Encoding mit dieser Fähigkeit *Indirect Encoding* genannt.

2.1.1 NeuroEvolution through Augmenting Topologies (NEAT)

NEAT [14] zeichnet sich dadurch aus, die Topologie eines Agenten von einem Komplexitätsminimum ausgehend zu entwickeln. Es nutzt ein *Direct Encoding*, dass Genom genannt wird. Gebildet wird das Genom eines Agenten aus einer Kette von Genen. Ein Agent ist ein gerichteter Graph. Ausgehend von Input-Neuronen und einem Bias-Neuron führen gewichtete Kanten zu Hidden-Neuronen und Output-Neuronen. Jedes Neuron hat einen Output-Wert, der über ausgehende Kanten den verbundenen Neuronen als Input dient. Sei E die Menge aller Neuronen. Zu einem Neuron $N \in E$ gehören eine Aktivierungsfunktion $a_N : \mathbb{R} \to \mathbb{R}$ und ein Menge $\Lambda_N \subset \mathbb{R} \times E$ gewichteter Neuronen $(w, M) \in \Lambda_N$ - also die Neuronen, die mit einer Kante in Richtung N mit N verbunden sind. Der Output $\Omega_N \in \mathbb{R}$ eines Neurons wird errechnet durch das Anwenden der Aktivierungsfunktion auf die Summe der gewichteten Outputs aller verbundener Neuronen:

$$\Omega_N \;=\; a_N \left(\sum_{(w,I) \in \Lambda_N} w \cdot \Omega_I \right) \tag{2.1}$$

Der Output des Bias-Neurons b ist konstant 1 und es hat keine Inputs. Input-Neuronen haben ebenfalls keine Inputs. Ihr Output wird von außerhalb des NN vorgegeben. Der Agent muss kein azyklischer Graph sein - rekurrente Verbindungen sind erlaubt. Bei einer Implementation muss auf rekurrente Verbindungen Rücksicht genommen werden.

Jedes Neuron und jede Kante eines Agenten wird durch ein Gen beschrieben. Dabei hält ein Neuron-Gen fest, welche Aktivierungsfunktion das Neuron besitzt, und ein Kanten-Gen hält fest, welches Neuron mit welchem Neuron verbunden ist, wie groß das Gewicht der Vebindung ist und ob die Verbindung *aktiv* ist (also ob sie beim Erzeugen des Agenten tatsächlich gebildet werden soll). Jedes Gen wird durch einen numerischen *Marker* eindeutig identifiziert.

0	Bias
1	Input 1
2	Input 2
3	Hidden 1 (SIGMOID)
4	Output 1 (SIGMOID)
5	Link (from 0 to 3)
6	Link (from 1 to 3)
7	Link (from 2 to 3)
8	Link (from 3 to 3)
9	Link (from 3 to 4)

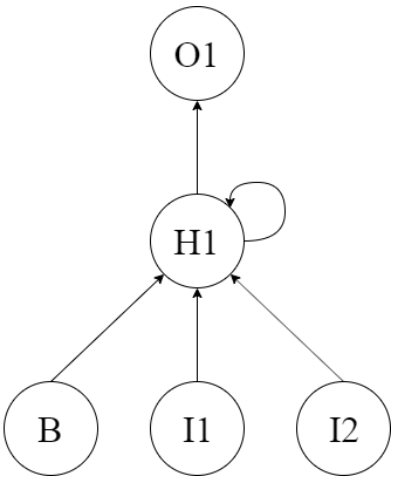


Abb. 2.1: Beispiel eines NEAT Genoms (links) mit instantiiertem neuronalen Netz (rechts). Im Genom wird links von jedem Gen der dazugehörige numerische Marker aufgeführt.

NEAT nutzt Mutation und Cross-Over zum Erzeugen neuer Genome. Dabei wird ein Teil der neuen Population asexuell durch Mutation alter Genome erzeugt und der Rest durch Cross-Over. Es werden drei elementare Mutationsoperationen unterschieden:

1. **Kantenveränderung**: Das Gewicht einer aktiven Kanten wird verändert; oder eine inaktive Kante wird aktiviert.

2. **Kantenbildung**: Ein Kantengen, ausgehend von einem Neuron zu einem Hidden-/Output-Neuron, wird zum Genom hinzugefügt; vorrausgesetzt, dass ein äquivalentes Kantengen (mit dem selben Start-/Ziel-Neuronen) noch nicht im Genom präsent ist.

3. **Kantensplit**: Eine aktive Kante wird geteilt. Dabei wird das korrespondierende Kantengen inaktiv. Ein neues Neurongen wird zum Genom hinzugefügt und mittels zwei neuer Kantengene mit den Start-/Ziel-Neuronen des deaktivierten Kantengens verbunden.

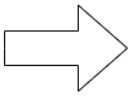
Parent 1		
0	Bias	M
1	Input	M
2	Output	M
3	Link (from 1 to 2)	M
4	Hidden	D
5	Link (from 0 to 4)	D
6	Link (from 1 to 4)	D
7	Link (from 4 to 2)	D
8	Link (from 0 to 2)	M

10	Link (from 4 to 4)	E
Fitness 0.6		

Parent 2		
0	Bias	M
1	Input	M
2	Output	M
3	Link (from 1 to 2)	M

8	Link (from 0 to 2)	M
9	Link (from 2 to 2)	D

Fitness 0.7		



Offspring	
0	Bias
1	Input
2	Output
3	Link (from 1 to 2)
8	Link (from 0 to 2)
9	Link (from 2 to 2)

Abb. 2.2: Beispiel einer Cross-Over Operation. ”M” bezeichnet ein ”Matched” Gen, das in beiden Eltern enthalten ist. ”D” sind *Disjoint*-Gene und ”E” sind *Excess*-Gene.

Cross-Over zweier Genome erfolgt in zwei Schritten. Zuerst werden aus den Genen beider Genome Paare gebildet. Dabei werden die Marker der Gene genutzt um festzustellen, welche Gene des einen Genoms mit welchen Genes des anderen Genoms korrespondieren. Wenn ein Gen kein korrespondierendes Gen hat bleibt es allein. Die Genpaare werden anhand ihrer Marker in aufsteigender Reihung sortiert. Alleinstehende Gene werden in die Kategorien *Disjoint* und *Excess* eingeteilt. Dabei sind *Excess*-Gene jene, dessen Marker größer ist, als der kleinere der größten Marker beider Eltern. *Disjoint*-Gene sind alle alleinstehenden Gene, die nicht *Excess* sind.

In einem zweiten Schritt wird jetzt das neue Genom gebildet. Dabei wird aus allen Gen-Paaren je ein Gen per Zufall genommen. Zudem erbt das neue Genom alle *Disjoint*- und *Excess*-Gene des Elternteils mit größerer Fitness.

Der numerische Marker der Gene wird so vergeben, dass zwei identische Gene, die zu unterschiedlichen Zeitpunkten in unterschiedlichen Genomen entstanden sind, identische Marker haben.

"Innovation" im Kontext von GAs beschreibt Additionen zur Struktur des NN, die langfristig zu einer verbesserten Leistung des Agenten führen - dabei aber selbst noch nicht unbedingt eine direkte Verbesserung der Fitness verursachen. Oft passiert sogar das Gegenteil: Innovation kann zu einer kurzfristigen Verschlechterung der Fitness führen, bevor weitere Mutationen die neue Struktur optimieren und damit langfristig die Fitness steigt. Dies ist ein Problem für GAs, da Innovationen durch ihre initial schlechtere Fitness oft sich nicht lange in der Agentenpopulation halten können. NEAT schützt Innovation durch einen Prozess namens *Speciation*. Dabei wird die Population in Spezies eingeteilt. Diese Einteilung erfolgt auf Basis von Ähnlichkeit. Sei A die Menge aller Agenten. Die Fitness F eines Agenten $G \in A$ wird nach Evaluation der Population angepasst, indem sie durch die Größe seiner Spezies $S \subseteq A$ geteilt werden. Diese Technik wird auch *Fitness-Sharing* [14] genannt. Diese angepasste Fitness F' dient im Weiteren als Basis zur Bestimmung der Fortpflanzung. Daraus folgt, dass Agenten, die Teil einer kleinen Spezies sind, einen evolutionären Vorteil gegenüber Agenten haben, die Teil einer großen Spezies sind - Agenten stehen mehr in Konkurrenz zu anderen Agenten der eigenen Spezies als allen Agenten der Population. Dabei ist die Idee, dass im Falle von Innovation die Strukturänderung zum Entstehen einer neuen Spezies führt, was sofortiges Aussterben unwarscheinlich macht. Dies gibt den Agenten Zeit die Innovation zu optimieren. Die Ähnlichkeit δ wird aus der Anzahl der *Disjoint*- und *Excess*-Genen (λ_D und λ_E), sowie der durchschnittlichen Gewichtsabweichungen aller *matching* Kantengene $\bar{\lambda}_W$ zweier Genome gebildet. Dabei sind die Gewichtungen w_D, w_E und w_W, sowie der *Specation-Threshold* $\hat{\delta}$ Hyper-Parameter von NEAT:

$$
\begin{aligned}
|G| &= \text{Anzahl der Gene im Genom des Agenten } G \\
\delta(G_1, G_2) &= \frac{w_D \lambda_D}{N} + \frac{w_E \lambda_E}{N} + w_W \bar{\lambda}_W \quad \text{mit } \; N = \max\{|G1|, |G2|\} \\
S(G) &= \{G' \in A : \; \delta(G, G') \leq \hat{\delta}\} \\
F'(G) &= \frac{F(G)}{|S(G)|}
\end{aligned}
\tag{2.2}
$$

2.1.2 Hypercube-based NEAT (HyperNEAT)

HyperNEAT [16] ist ein GA, der es ermöglicht ein trainierten Agenten zu generalisieren. D.h. die Neuronendichte eines Agenten kann erhöht werden ohne den Agenten noch einmal trainieren zu müssen. Dazu nutzt HyperNEAT *Compositional Pattern Producing Networks* (CPPN) [15] als

Indirect Encoding. Ein CPPN ist im Kern ein NN wie vorgestellt in Abschnitt 2.1.1. Es zeichnet sich durch eine breite Vielfalt von verschiedenen Aktivierungsfunktionen aus. Es hat zudem genau ein Output-Neuron. Als Input nimmt ein CPPN ein Koordinaten-Paar. Der Gedanke dahinter ist, dass wenn eine Menge von Punkten in einem n-dimensionalen Raum gegeben ist ein CPPN genutzt werden kann um Kantengewichte zwischen zwei Punkten zu errechnen. HyperNEAT weist also allen Neuronen des zu trainierenden Netzes Positionen in einem n-dimensionalen Raum zu und errechnet dann die Kantengewichte zwischen zwei Neuronen mittels eines CPPNs.

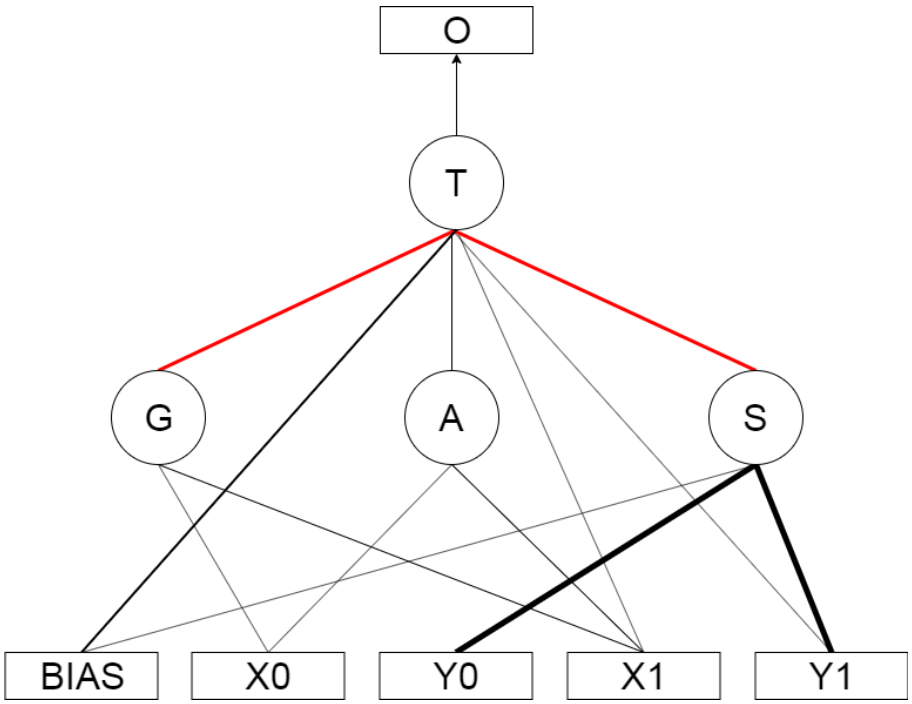


Abb. 2.3: Beispiel eines CPPN. Breite der Verbindung repräsentiert die Stärke des Gewichts der Kante. Rote Verbindungen signalisieren Kanten mit negativen Gewichten. Kreise repräsentieren Hidden-Neuronen. Der Buchstabe in einem Neuron ist die Aktivierungsfunktion: "A" ist der Betrag $|x|$, "S" ist die Sinus-Funktion $\sin(x)$, "G" ist die Gauss-Funktion $\exp(-\frac{1}{2}x^2)$ und "T" ist tanh.

Ein Agent, der von HyperNEAT trainiert wird, wird also aus zwei Komponenten gebildet: Dem *Substrat* - der räumlichen Anordnung der Neuronen - und einem CPPN - zur Bestimmung der Kantengewichte zwischen den Neuronen. Dabei wird das Substrat vor Beginn des Trainings festgelegt und verändert sich wärend des Trainings nicht. Stattdessen wird nur das CPPN mittels NEAT trainiert. Durch diese Trennung kann man nach Abschluss des Trainings durch Veränderungen des Substrats die Neuronendichte erhöhen ohne die Funktion des Agenten signifikant negativ zu beeinträchtigen, da die Kantengewichte von/zu den neu hinzugefügten Neuronen mittels des CPPNs auch im Nachhinein noch ermittelt werden können. Im Kontrast dazu kann z.B. ein mit klassischem NEAT trainierter Agent dies nicht: Das Gewicht neu eingefügter Kanten kann nicht im Nachhinein bestimmt werden. Dies ist eine direkte Konsequenz des *Direct Encoding*s von NEAT. Das Gewicht einer Kante wird durch genau ein Kantengen bestimmt. Eine im

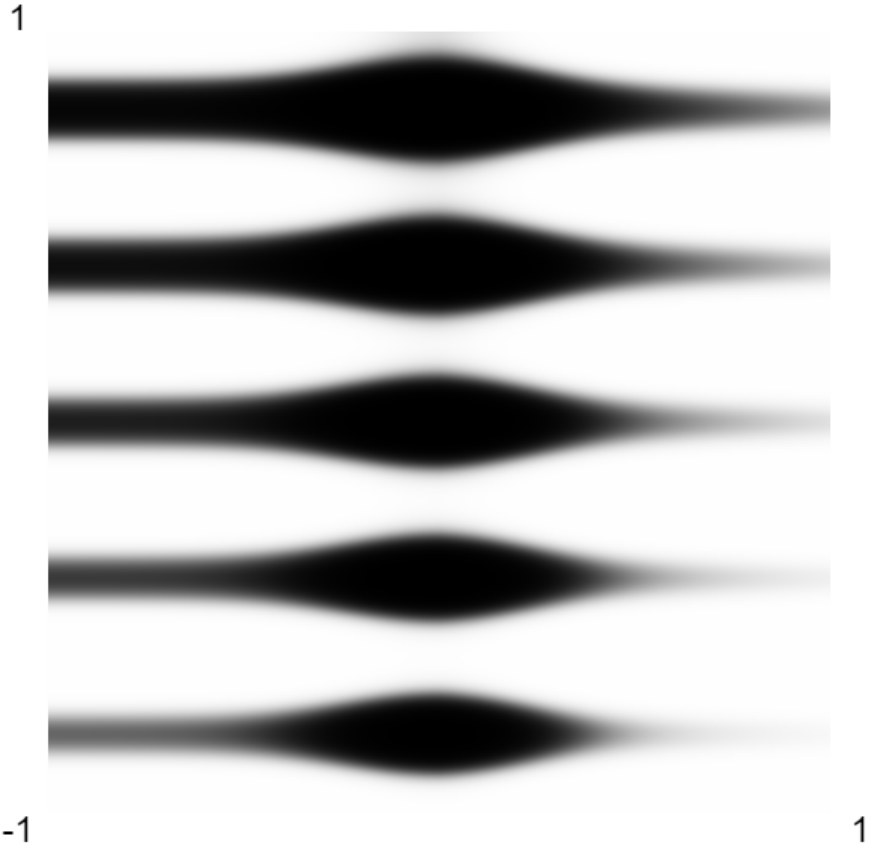


Abb. 2.4: Visualisierung eines 2D Slice des CPPNs in Abb. 2.3 für alle $X_1, Y_1 \in [-1, 1]$ mit $X_0 = Y_0 = 0$ erstellt. Funktionswerte $f(X_0, Y_0, X_1, Y_1)$ nahe bei -1 werden schwarz dargestellt. Funktionswerte nahe bei 1 werden weiss dargestellt.

Nachhinein hinzugefügte Kante hat kein Kantengen und demnach kann kein (sinnvolles) Gewicht für diese Kante bestimmt werden. In HyperNEAT werden Kantengewichte indirekt durch das CPPN kodiert. Dadurch kann das CPPN genutzt werden um Gewichte beliebig vieler Kanten zu ermitteln - selbst solcher, die beim Training nicht vorhanden waren. In [16] konnte ein Agent, der in einem *Visual Discrimination Task* mit einem Sichtfenster von 11 x 11 Punkten (121 Inputs) trainiert wurde, nur durch Erhöhen der Neuronendichte auch mit einem größerem Sichtfeld (33 x 33 und 55 x 55 Punkte) umgehen. Wenn das Substrat eine Neuronenverteilung in einem 2-dimensionalen Raum beschreibt modelliert ein CPPN eine reellwertige Funktion $f : \mathbb{R}^4 \to \mathbb{R}$ der Variablen x_0, y_0, x_1, y_1 wobei eine Kante von dem Neuron an Punkt (x_0, y_0) ausgehend zum Neuron an Punkt (x_1, y_1) mit dem Funktionswert $f(x_0, y_0, x_1, y_1)$ gewichtet wird. Somit ist die Funktionsmenge so eines CPPNs ein 4-dimensionaler Raum. Da valide Koordinaten für Neuronen oft beschränkt werden auf $[-1, 1] \times [-1, 1]$, ist die Funktionsmenge eines CPPNs ein *Hypercube*.

2.1.3 Evolvable Substrate HyperNEAT (ES-HyperNEAT)

ES-HyperNEAT [10] [11] [12] baut direkt auf HyperNEAT auf. Es adressiert Schwächen von HyperNEAT, speziell im Punkt der Substratbildung. HyperNEAT kann selbst kein Substrat erzeugen, es muss "von Hand" konstruiert werden. Dies ist ein eher fehleranfälliger Ansatz, da Menschen in der Regel keine gute Intuition bei der Konstruktion neuronaler Netze haben. ES-HyperNEAT führt daher ein Verfahren zur Generation von Hidden-Neuronen und verbindender Kanten eines Substrats auf Basis der Input-/Output-Neuronen und des CPPNs ein. Dabei wird ausgehend von einem Neuron an dem Punkt (x_0, y_0) eine Abbildung $h : \mathbb{R}^2 \to \mathbb{R}$ mit dem Funktionswert $h(x, y) = f(x_0, y_0, x, y)$ auf dem CPPN gebildet. h beschreibt so einen 2-dimensionalem Raum in dem jeder Punkt (x, y) das Kantengewicht zwischen dem Neuron an Punkt (x_0, y_0) und einem hypothetischen Neuron an Punkt (x, y) beschreibt. ES-HyperNEAT plaziert jetzt Neuronen in dem von h beschriebenen Raum. Dabei ist die Dichte der plazierten Neuronen in einem Teilraum proportional zu der Varianz des Funktionswertes von h in diesem Teilraum. Dieser Prozess wird auch "rückwärts" - ausgehend von einem Output-Neuron an Punkt (x_1, y_1) mit der Abbildung $h'(x, y) = f(x, y, x_1, y_1)$ - durchgeführt. Diese Schritte können mit bereits gefundenen Hidden-Neuronen beliebig oft wiederholt werden um weitere Neuronen zu plazieren. Aus dem so entstandenen Graphen werden jetzt alle Neuronen und Kanten gelöscht, die keine Verbindung zu Output-Neuronen oder von Input-Neuronen haben. Was bleibt ist das fertige Substrat.

Auf den Abbildungen 2.5-2.8 ist beispielhaft die Bildung eines Substrats illustriert. Das zu bildende Substrat hat einen Input an der Position $(0, -1)$ und einen Output an der Position $(0, 1)$. In Abb. 2.5 ist die Visualisierung des CPPNs - ausgehend von dem Input (grüner Punkt) - zu sehen. Es wurde ein mögliches Hidden-Neuron an der Stelle $(-0.5, 0.5)$ (roter Punkt) gefunden. Ausgehend von allen gefundenen, möglichen Hidden-Neuronen werden weitere mögliche Hidden-Neuronen gesucht (Abb. 2.6). Dieser Schritt kann beliebig oft wiederholt werden. Typischerweise wird dieser Schritt jedoch nur ein bis zwei Mal durchgeführt. Dann werden in Abb. 2.7 zum Output (grüner Punkt) führende Verbindungen gefunden. Letztlich werden alle gefundenen Neuronen und Verbindungen zu einem Graphen verknüpft. Dieser Graph ist in Abb. 2.8 zu sehen. Alle Verbindungen, die nicht auf einem Pfad von Input- zu Output-Neuronen liegen werden entfernt (graue Kanten). Nun werden überflüssige Neuronen ebenfalls entfernt (weisse Kreise). Was bleibt ist das fertige Substrat.

Um entscheiden zu können, wo Neuronen plaziert werden können, wird der durch h beschriebene Raum in eine *QuadMap* [2] eingeordnet. Eine QuadMap ist eine baumartige Datenstruktur bei der jeder Knoten eine 2-dimensionale Fläche beschreibt. Alle Knoten, die Kinder haben, haben exakt vier Kinder. Die Fläche jeder dieser Kinder bedeckt exakt ein Viertel der Fläche es Elternknoten. Die Kinder überlappen sich nicht. Unter der Annahme, dass alle Neuronen in dem Raum $[-1, 1] \times [-1, 1]$ plaziert werden, kann also eine QuadMap den Raum aller möglicher Neuronenplazierungen abdecken und partitioniert diesen in beliebig kleine Abschnitte. Hierbei

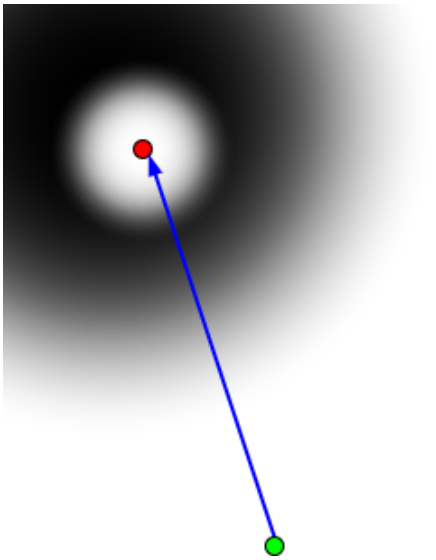


Abb. 2.5: Input-Neuron bei (0, -1)

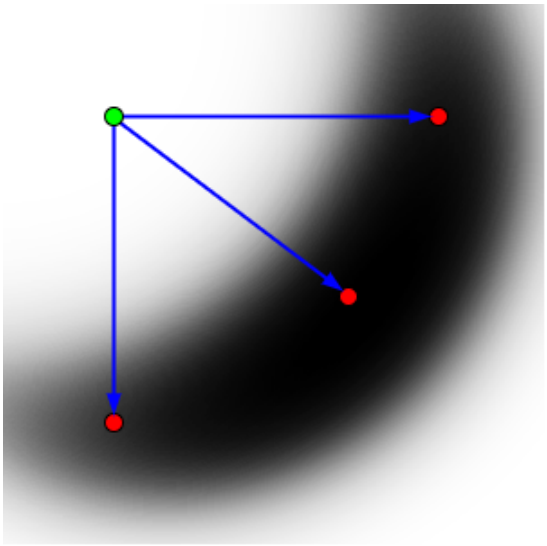


Abb. 2.6: Hidden-Neuron bei (-0.5, 0.5)

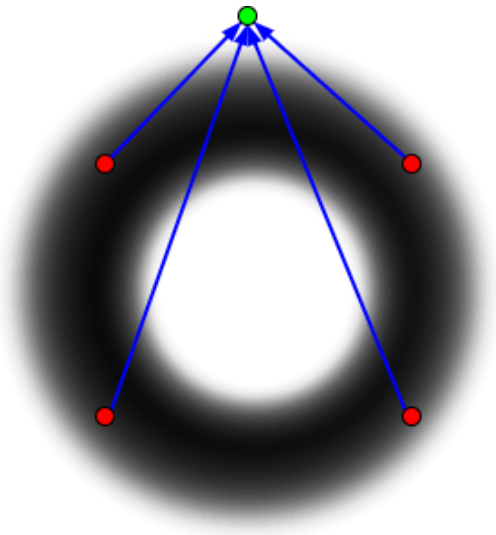


Abb. 2.7: Output-Neuron bei (0, 1)

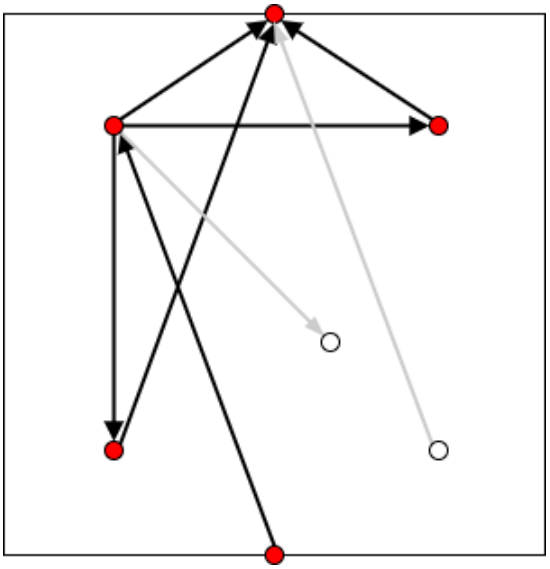


Abb. 2.8: Fertiges Substrat.

wird die Entscheidung, ob ein Knoten in der Quadmap weiter geteilt werden soll, abhängig von der Varianz σ^2 innerhalb der Fläche des Knoten getroffen. Sei $K = (C, w)$ ein kinderloser Knoten in der QuadMap mit einem Mittelpunkt $C = (x_c, y_c),\ x_c, y_c \in \mathbb{R}$ und Seitenlänge $w \in \mathbb{R}$. Dann kann die Varianz $\sigma^2(K)$ approximiert werden durch die Varianz über eine Menge von Punkten aus der Fläche von K: $\sigma^2(K) \approx \sigma^2(h(x_c - \frac{w}{4}, y_c - \frac{w}{4}),\ h(x_c - \frac{w}{4}, y_c + \frac{w}{4}),\ h(x_c + \frac{w}{4}, y_c - \frac{w}{4}),\ h(x_c + \frac{w}{4}, y_c + \frac{w}{4}))$. Liegt die Varianz von K über einem Schwellenwert σ_t^2 dann wird K geteilt und rekursiv für alle Kinder von K der Algorithmus noch einmal angewandt. Die Mittelpunkten aller Blattknoten der QuadMap sind jetzt potentielle Neuronenpositionen. ES-HyperNEAT filtert jetzt noch alle Knoten aus, die nicht in einem "Band" liegen. Dabei ist ein "Band" eine Fläche, deren horizontale **oder** vertikale, gleichgroße Nachbarflächen einen änlichen Durchschnittswert μ besitzen, der sich von dem Durchschnittswert des Bandes unterscheidet. Dabei ist der Durchschnittswert einer Fläche approximiert durch den Funktionswert $h(C)$ des Mittelpunktes C der Fläche. Um zu Testen, ob ein Knoten K in einem Band liegt, wird also die Differenz von $\mu(K)$ zu den vier benachbarten Flächen $N = ((x_c \pm w, y_c \pm w), w)$ bestimmt: $d_i = |\mu(K) - \mu(N_i)|$ Dass der Durchschnittswert beider jeweils gegenüberliegenden Flächen gleichermaßen von $\mu(K)$ abweicht wird durch $\delta_h = \min\{d_{left}, d_{right}\}$ bzw. $\delta_v = \min\{d_{up}, d_{down}\}$ bestimmt - ist $\mu(N_i)$ nahe bei $\mu(K)$ ist d_i entsprechend klein und demnach ist auch $\min\{d_i, d_j\}$ klein. Um zu überprüfen, dass entweder die vertikalen **oder** die horizontalen Nachbarn von $\mu(K)$ abweichen wird $\delta = \max\{\delta_h, \delta_v\}$ bestimmt. Ist δ_h oder δ_v groß, so ist auch δ groß. Liegt δ jetzt über einem Schwellwert so gilt $K = (C, w)$ als *in-band* und ein Neuron wird bei C plaziert.

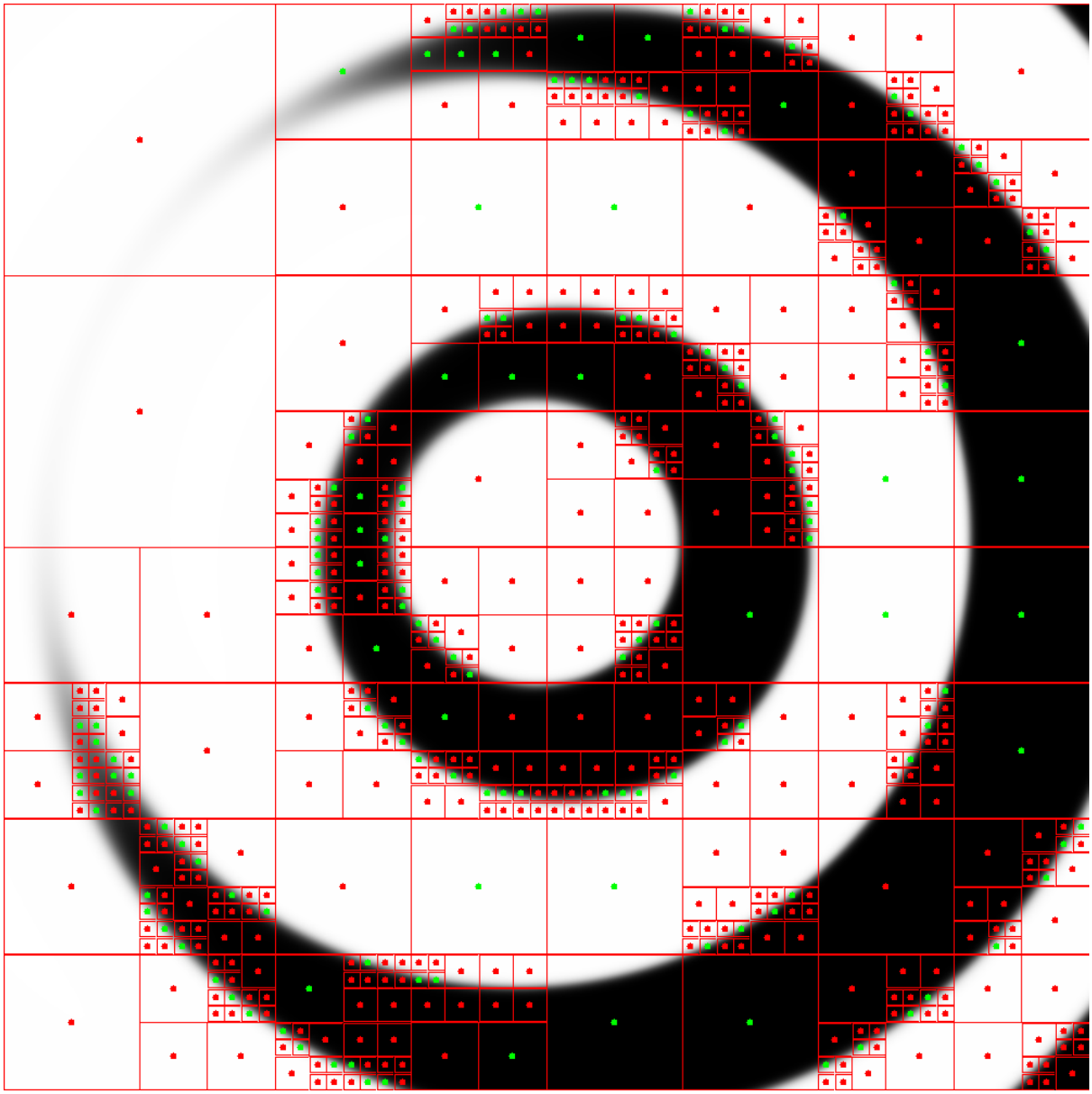


Abb. 2.9: Beispiel einer CPPN mit visualisierter QuadMap. Grüne Punkte symbolisieren plazierte Neuronen.

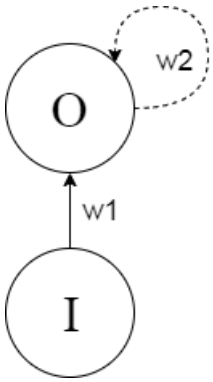
2.2 Modelle rekurrenter neuronaler Netze

Klassische *Feed-Forward* NN können keinen temporalen Zusammenhang verschiedener Aktivierungen des Netzwerks modellieren. Sie haben kein "Gedächtnis" - unter Ausschluss von Zufallselementen berechnet das Netzwerk für einen Input I immer den Output O, völlig unabhängig von möglichen zuvor berechneten Werten. Bei *rekurrenten neuronalen Netzen* (RNN) ist dies anders. RNNs sind in der Lage bei der Berechnung des Outputs auf zuvor durchgeführte Arbeit zurüzugreifen. Dazu berechnet ein RNN neben dem Output O auch einen *Hidden State* H, der bei der nächsten Aktivierung des Netzes neben dem Input I ebenfalls berücksichtigt wird.

$$f : \begin{cases} \mathbb{R}^n \times \mathbb{R}^k & \rightarrow & \mathbb{R}^k \times \mathbb{R}^m \\ (I_t,\, H_{t-1}) & \mapsto & (H_t,\, O_t) \end{cases} \tag{2.3}$$

2.2.1 Einfaches Recurrent Neural Network (RNN)

Das einfachste Modell eines RNN ist einfach ein *Feed-Forward* NN, das rekurrente Kanten in der Topologie erlaubt. Der *Hidden State* des Netzes wird dabei typischerweise durch Merken der Aktivierungen implementiert: Als Input einer rekurrenten Kante wird der Output des verbundenen Neurons aus der vorigen Berechnung genommen und nicht der aktuelle Output, der u.U. ja noch gar nicht berechnet ist.



$$\Omega_t(O) \;=\; a\,(w_1\,\Omega_t(I) \;+\; w_2\,\Omega_{t-1}(O))$$

Abb. 2.10: Beispiel eines einfachen RNN

Theoretisch kann ein einfaches RNN jede Form von temporalen Zusammenhang zwischen verschiedenen Inputs erfassen. In der Praxis fällt es einfachen RNN jedoch schwer Zusammenhänge zu bilden zwischen Inputs die zeitlich weit auseinander liegen - sein Gedächtnis ist sehr kurz. Dies folgt aus der Art und Weise wie rekurrente Zusammenhänge gespeichert werden: die rekurrenten Kanten nehmen jeweils den Ouput des verbundenen Neurons aus dem vorigen Zeitschritt. Die Warscheinlichkeit, dass ein Wert jetzt über längere Zeit "gehalten" werden kann, fällt proportional zum Abstand der Ursprünglichen Berechnung und dem Punkt in dem dieser Wert tatsächlich gebraucht wird.

2.2.2 Long Short Term Memory (LSTM)

Ein LSTM [4] ist eine Form von RNN das konzipiert wurde um auch mit langfristigen Abhängigkeiten in Input-Sequenzen umgehen zu können. Dazu wird der *Hidden State* in einer *Cell* gespeichert. Informationsfluß in und aus der *Cell* wird durch *Gates* reguliert. Ein *Gate* ist ein elementares NN, dessen Output ein Vektor reeller Zahlen im Wertebereich $[0, 1]$ ist. Der Output eines Gates wird punktweise mit einem anderen Vektor multipliziert. Die hat den Effekt einer "Maskierung": Wo der Output des *Gate*s bei 0 liegt ist das Produkt ebenfalls bei 0 - der Wert des anderen Vektors geht verloren. Liegt der Output des *Gate*s bei 1 verändert sich der Wert des anderen Vektors nicht. Neben drei *Gate*s hat ein LSTM noch zwei weitere NN: Das Input-Netz, das die *Cell*-Veränderungen aus dem Input und *Hidden State* berechnet, und das Output-Netz, das aus dem neuen *Hidden State* den Output des LSTMs berechnet.

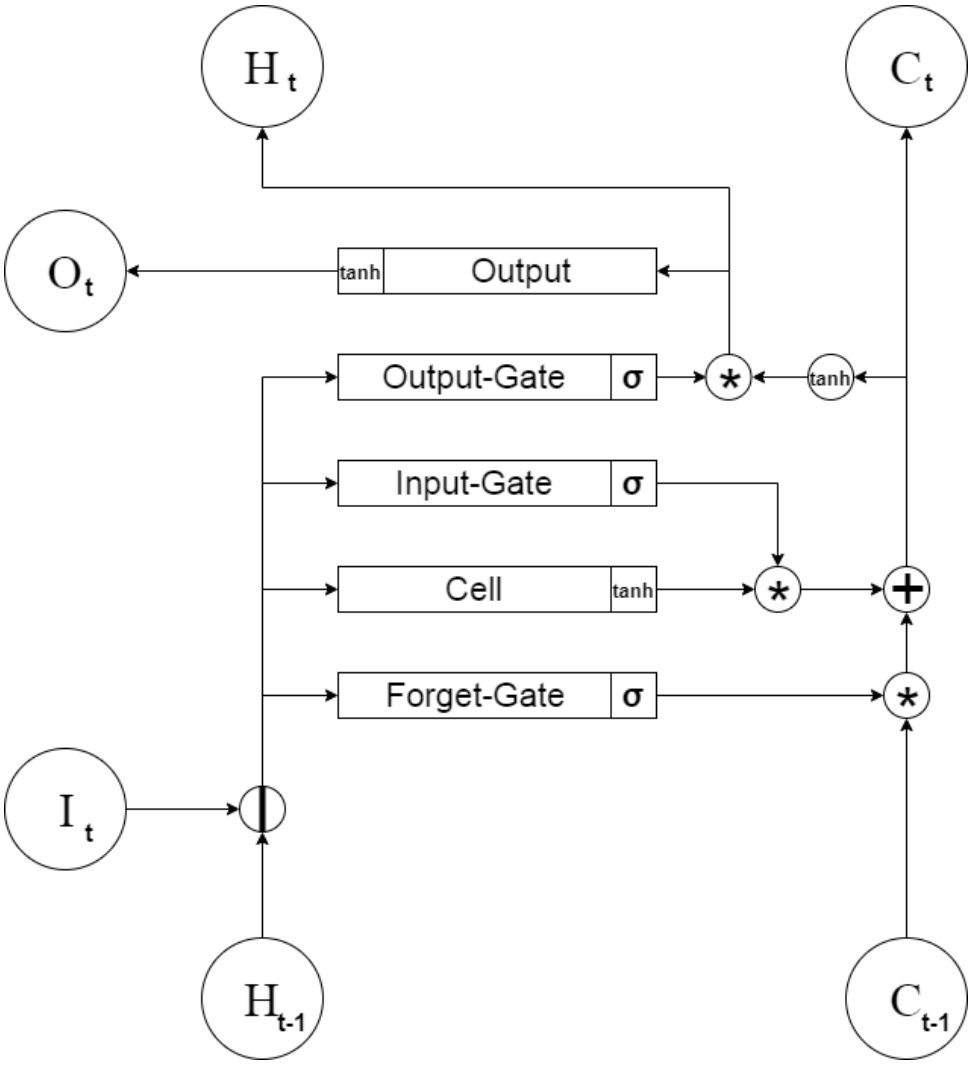


Abb. 2.11: übersicht eines LSTM. Kästen beschreiben NN. Kleine Kreise sind Operationen: "|" ist Vektor-Concatenation, "+" ist punktweise Addition, "*" ist punktweise Multiplication und "tanh" ist punktweise Anwendung der *tanh* Funktion.

Der *Hidden State* eines LSTM ist der Vektor C_t. Dieser wird in jedem Zeitschritt durch Anwenden des *Forget-Gates* und Addieren des bearbeiteten Inputs modifiziert. Ein Hilfsvektor H_t wird durch normalisieren (mittels *tanh*) und dem *Output-Gate* aus C_t gebildet. Dieser Hilfsvektor dient als Basis zur Berechnung des Outputs O_t und der änderung von C_t im Folgeschritt $t + 1$.

2.2.3 Gated Recurrent Unit with Memory Block (GRU-MB)

Die *Gated Recurrent Unit* (GRU) [1] ist eine vereinfachte Form eines LSTM. GRU-MB [5] baut auf GRU auf indem es den *Hidden State* weiter entkoppelt. Änderungen am *Hidden State* werden durch explizite *Read-*/ *Write*-Operationen modelliert. Realisiert werden diese durch *Read-* und *Write-Gates*. Im Kontrast dazu ist beim LSTM der Hidden State Teil der zentralen *Feedforward*-Operation: Aus dem Input I_t und Hilfsvektor H_{t-1} wird ein Update-Vektor berechnet, der zum alten *Hidden State* C_{t-1} addiert wird. Dann wird aus dem neuen *Hidden State* C_t der neue Hilfsvektor H_t berechnet. Aus diesem wird der Output O_t berechnet. Der *Hidden State* C_t ist also zentral in der *Feedforward*-Operation: Der Output O_t wird (durch H_t) aus C_t errechnet. In GRU-MB ist dies nicht der Fall: Aus Input I_t und *Hidden State* H_{t-1} wird ein Hilfsvektor T errechnet. Aus diesem Vektor T wird sowohl der Output O_t wie auch die Änderung für H_t errechnet. Somit ist die Änderung des *Hidden State* und die Berechnung von O_t nicht sequenziell, wie beim LSTM, sondern parallel:

$$\text{LSTM}: \quad I_t + H_{t-1} \rightarrow C_t \rightarrow H_t \rightarrow O_t$$

$$\text{GRU-MB}: \quad I_t + H_{t-1} \rightarrow T \rightarrow \left| \begin{array}{l} \rightarrow O_t \\ \rightarrow H_t \end{array} \right.$$

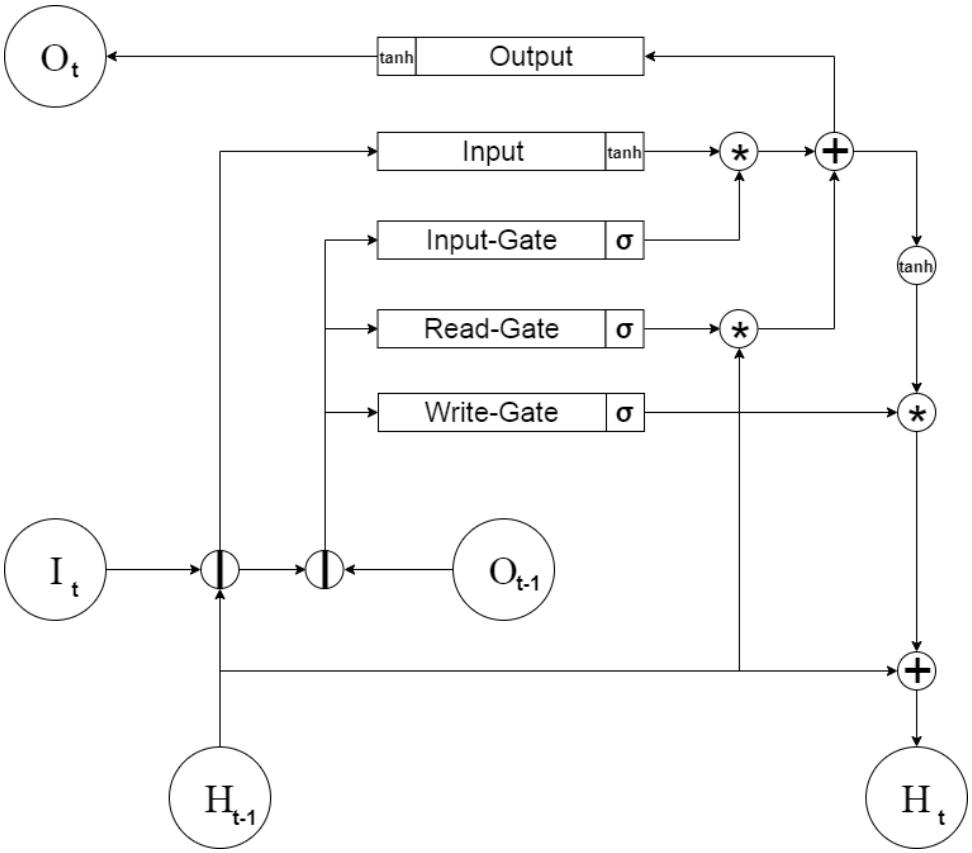


Abb. 2.12: übersicht einer GRU-MB. Kästen beschreiben NN. Kleine Kreise sind Operationen: "|" ist Vektor-Concatenation, "+" ist punktweise Addition, "*" ist punktweise Multiplication und "tanh" ist punktweise Anwendung der *tanh* Funktion.

3 Untersuchungen

In dieser Arbeit werden Kombinationen verschiedener GAs und RNN-Architekturen untersucht. Da ES-HyperNEAT eine direkte Erweiterung von HyperNEAT ist wurde auf einen Vergleich mit HyperNEAT verzichtet. Zur Referenz wurde neben klassischem NEAT und ES-HyperNEAT auch ein sehr einfacher GA angewandt. Dieser GA - *flat-nn* genannt - erzeugt einfache Feedforward-NN mit exakt einem Feedforward-Layer:

$$\Omega \;=\; \tanh\left(W \cdot \Lambda\right) \tag{3.1}$$

Die Gewichtsmatrix W wird wärend der Evolution mittels *Gaussian Noise* Punktweise mutiert [5].

Neben LSTM und GRU-MB wird als Referenz auch eine einfache RNN Architektur untersucht - schlicht RNN genannt. RNN-Agenten bestehen intern einfach aus einem NN, dass durch den jeweiligen GA gebildet wurde. Bei LSTM und GRU-MB werden für die einzelnen *Gates* und anderen NN ebenfalls Netze verwendet, die durch den jeweiligen GA erzeugt wurden. Daraus folgt auch, dass bei der Kombination von "einfachem RNN" und "flat-nn" gar kein rekurrentes Netz entsteht, da das ganze Netz lediglich eine einzelne, einfache Feedforward-Schicht ist. Demnach sollte "flat-nn RNN" in allen Aufgaben deutlich schlechter abschneiden, als die anderen GA-RNN Kombinationen.

Im Folgenden werden die einzelnen Versuche vorgestellt und die Beobachtungen festgehalten. Die Auswertungen der Ergebnisse werden im Folgekapitel dargelegt.

3.1 Simple Maze Navigation

In dieser Untersuchung muss der Agent ein Labyrinth durchqueren. Das Labyrinth ist eine quadratische Anordnung von $n \times n$ Feldern. Jedes Feld ist entweder begehbar (Boden) oder nicht (Wand). Ein Agent sitzt immer im Mittelpunkt des Feldes, auf dem er sich befindet. Seine Blickrichtung ist entweder Norden, Süden, Westen oder Osten. Ein Simulationsdurchlauf beginnt mit der zufälligen Generation eines Labyrinths fester Größe. Der zu evaluierende Agent wird in einer zufälligen Ecke des Labyrinths plaziert. Sein Ziel ist es, die entgegengesetzte Ecke des Labyrinths zu erreichen. Als Input $\Lambda \subset \{-1, 1\}^3$ wird dem Agenten das Feld direkt vor ihm

(entsprechend seiner Blickrichtung), links neben ihm und rechts neben ihm gegeben (Boden = -1, Wand = 1). Als Output $\Omega \subset [0,1]^3$ liefert der Agent seine Absicht einen Schritt vorwärts zu tun, sich 90° nach links oder 90° nach rechts zu drehen. Wenn $\Omega_1 \geq 0.5$ versucht der Agent sich einen Schritt vorwärts zu bewegen. Wenn dagegen $\Omega_1 < 0.5$, dann wird der Rotationsfaktor $\delta = \Omega_3 - \Omega_2$ errechnet. Ist $|\delta| \geq 0.5$ dreht sicht der Agent um 90°. Wenn $\delta < 0$ dreht sich der Agent nach links, ansonsten nach rechts. Mit der aktualisierten Position und Orientierung wird der Schritt wiederholt, bis der Agent die Zielecke erreicht hat oder die Simulationszeit abgelaufen ist. Die Fitness eines Agenten wird aus seiner Distanz zum Ziel errechnet.

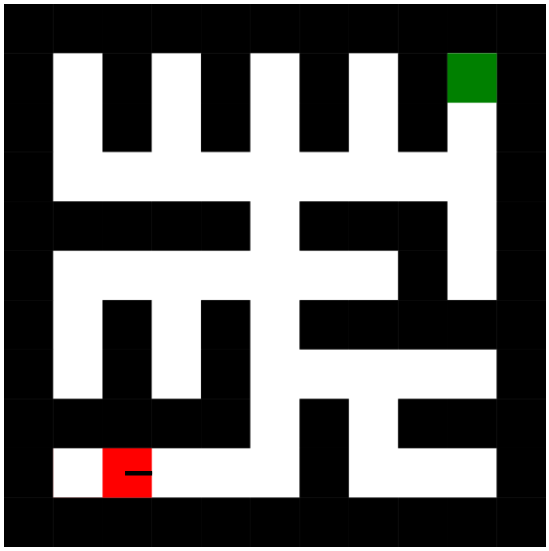


Abb. 3.1: Beispiel eines generierten 11x11 Labyrinthes. Betretbarer Boden ist weiss, Wände sind schwarz. Der Agent ist rot, seine Blickrichtung visualisiert durch einen schwarzen Balken. Das Ziel ist grün.

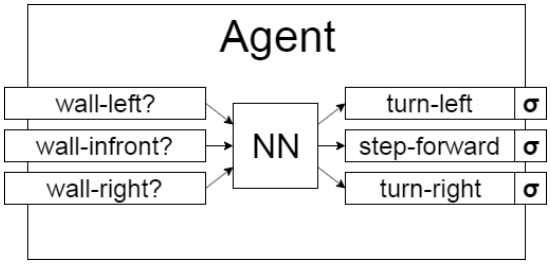


Abb. 3.2: Aufbau eines Agenten zur Labyrinthnavigation mit drei Inputs und drei Outputs.

Da bei den generierten Labyrinthen alle Wände miteinander verbunden sind, ist die optimale Strategie zum Durchqueren sehr einfach: Der Agent muss nur einer Wand folgen. Früher oder später erreicht der Agent so das Ziel. Für den Agenten heißt dies also, dass der Agent beim Erreichen einer Kreuzung immer zuerst versuchen sollte rechts abzubiegen. Ist dies nicht möglich, dann sollte er versuchen Geradeaus zu gehen. Ist auch das nicht möglich versucht er links abzubiegen. Letztlich, wenn auch das nicht möglich ist, sollte er eine Kehrtwende machen. Da ein Agent sich nicht gleichzeitig Drehen und einen Schritt vorwärt machen kann, muss er in der Lage sein sich zu merken, ob er gerade eine Drehung durchgeführt hat oder nicht. Ansonsten würde nach einer rechts-Drehung das Feld, das vor der Drehung hinter dem Agenten war (also von dem er ursprünglich gekommen ist), jetzt rechts von ihm liegen. Ohne ein Gedächtnis würde der Agent sich jetzt wieder nach rechts drehen, da wiederum ein freies Feld rechts von ihm liegt.

Bei einer 4-weg Kreuzung würde der Agent also nur noch um die eigene Achse rotieren. Zur erfolgreichen Navigation ist also eine Form von Gedächtnis unablässig. Die Anforderungen an dieses Gedächtnis sind gering - es muss nur in der Lage sein sich an die zuletzt durchgeführte Aktion zu erinnern. Ereignisse, die weiter zurück liegen, können vergessen werden. Durch diese einfache Lösungsstrategie und geringe Anforderung an ein Gedächtnis wird keine außerordentliche Leistungssteigerung bei Anwenden von ES-HyperNEAT erwartet. LSTM, GRU-MB und auch einfache RNN sollten allesamt in der Lage sein, die Aufgabe gut zu lösen.

Die Fitness F wird aus der Distanz der Position $\vec{p_t}$ des Agenten am Ende der Simulation zum Ziel $\vec{g}$ berechnet:

$$F \;=\; \max\left\{0,\;\; 1 \;-\; \frac{|\vec{g} \;-\; \vec{p_t}|}{|\vec{g} \;-\; \vec{p_0}|}\right\} \tag{3.2}$$

Ein Agent, der in jedem Zeitschritt seine Outputs per Zufall wählt, erreicht im Durchschnitt eine Fitness von ≈ 0.25.

Für jede der 9 GA-RNN Kombinationen wurden 20 Versuche durchgeführt. Dabei wurde jeweils für 500 Generationen trainiert.

Da bei der Kombination aus "flat-nn" und "rnn" keine rekurrenten Verbindungen bestehen, sollte so ein NN nicht in der Lage sein das Labyrinth erfolgreich zu navigieren. Dies bestätigt sich in den Ergebnissen in Abb. 3.3: "flat-nn" ist leistungstechnisch auf dem selben Niveau wie ein zufallsbasierter Agent. RNN Agenten, deren NN durch NEAT bzw. ES-HyperNEAT erzeugt wurden, schneiden deutlich besser ab. Beide nähern sich nach 500 Generationen einem Fitness-Wert von 0.9. Dies zeigt, dass in beiden Fällen die Agenten in der Lage sind einen substanziellen Teil des Labyrinths zu erkunden - was eine rudimentäre Form von Gedächtnis voraussetzt ("Bin ich gerade abgebogen oder muss ich das noch?"). Dabei schneidet ES-HyperNEAT wie erwartet noch ein bisschen besser ab als klassisches NEAT.

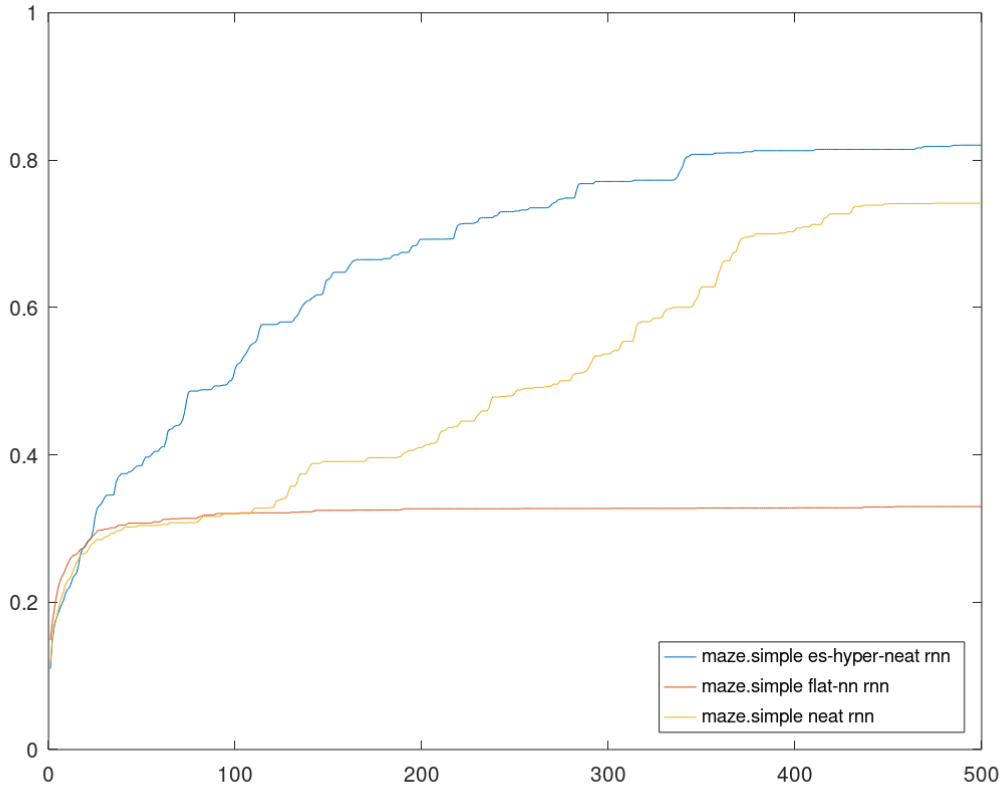


Abb. 3.3: Ergebnisse des "Maze" Versuchs mit einfachen RNN. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

Wie in Abb. 3.4 zu sehen ist schneiden LSTM-basierende Agent wie erwartet noch etwas besser ab als Agenten, die nur ein einfaches RNN besitzen. Dabei erreichen die LSTM-Agenten mit NEAT und ES-HyperNEAT beide eine maximale Fitness von ≈ 1. Selbst mit einfachen Feedforward-NN schneidet ein LSTM-Agent noch deutlich besser ab, als ein Zufallsagent - allerdings nicht so gut, wie mit NEAT/ES-HyperNEAT.

Eine kleine Leistungssteigerung von ES-HyperNEAT gegenüber klassischem NEAT ist auch hier zu verzeichnen.

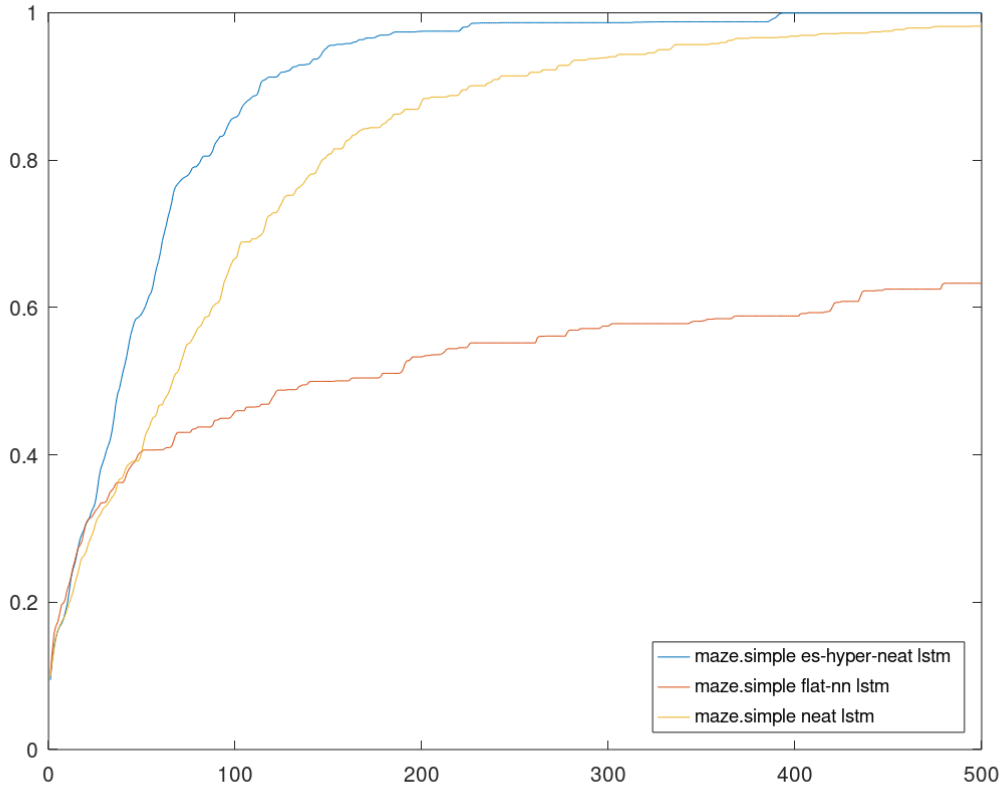


Abb. 3.4: Ergebnisse des ”Maze” Versuchs mit LSTM. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

Abb. 3.5 zeigt die Ergebnisse von Agenten, die GRU-MB benutzen. Grundsätzlich sind diese den Ergebnissen der LSTM-Agenten sehr ähnlich - NEAT und ES-HyperNEAT können die Aufgabe recht zuverlässig lösen; Mit einfachen Feedforward-NN ist nur eine gute Leistung zu erzielen.

Im Vergleich mit LSTM ist jedoch zu Bemerken, dass die Fitness von LSTM-Agenten deutlich schneller wächst als bei GRU-MB: Mit ES-HyperNEAT erreichen LSTM-Agenten eine Fitness von ≈ 1 im Durchschnitt nach 200 Generationen - GRU-MB-Agenten brauchen dafür 300 Generationen. Mit NEAT erreichen LSTM-Agenten eine Fitness von ≈ 0.8 nach 150 Generationen - GRU-MB-Agenten brauchen dafür 250 Generationen.

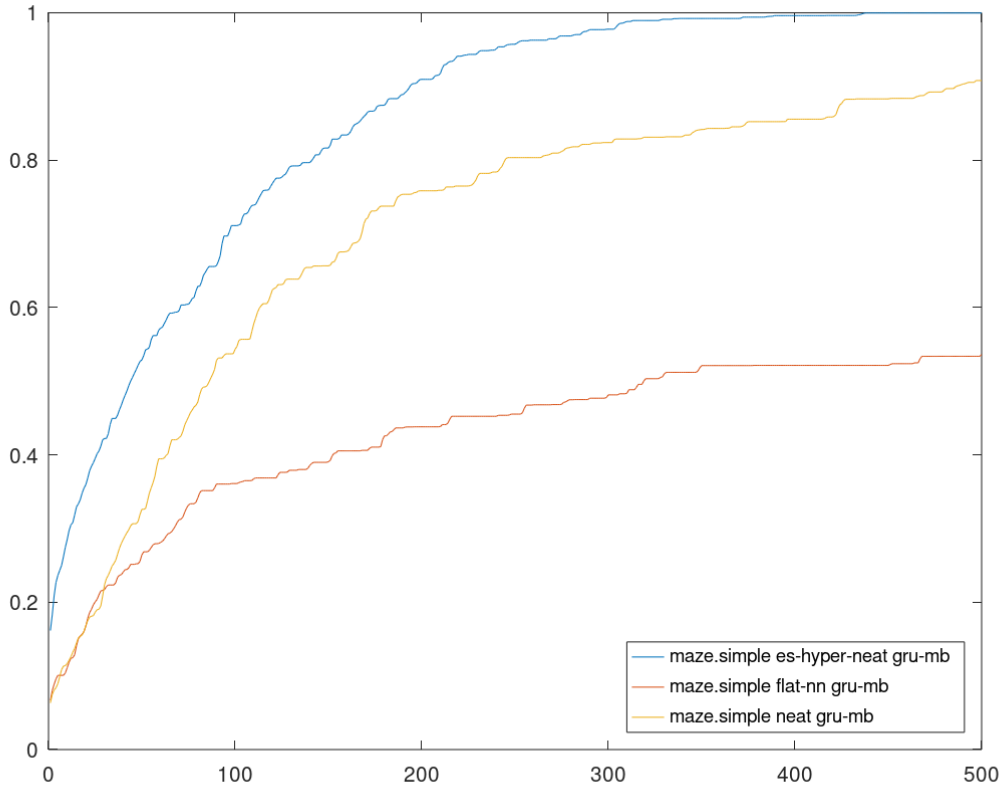


Abb. 3.5: Ergebnisse des "Maze" Versuchs mit GRU-MB. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

3.2 Counting

In dieser Untersuchung bekommt der Agent eine Reihe von n zufälligen Werten $\varphi \in \{-1, 0, 1\}^n$. Dabei wird ihm in jedem Zeitschritt t ein Wert φ_t aus φ als Input $\Lambda(t) = \varphi_t$ gegeben. Sein Ziel ist es die Anzahl von 1en und -1en zu zählen. Sein Output soll 1 sein wenn er mehr 1en als -1en bekommen hat, -1 wenn mehr -1en als 1en gegeben wurden und ansonsten 0. Alle 0en, die der Agent als Input bekommen hat, sollen ignoriert werden - sie beeinflussen den Output nicht. Eine Lösung für dieses Problem ist

$$\Omega(t) \;=\; \operatorname{sgn}\left(\sum_{i}^{t} \Lambda(i)\right) \tag{3.3}$$

Zum Lösen dieser Aufgabe muss ein Agent in der Lage sein sich Informationen aus allen verstrichenen Zeitschritten zu merken. Dabei ist es jedoch nicht notwendig, dass der Agent den exakten Input-Wert jedes Zeitschritts weiß. Der Gleichung 3.3 kann entnommen werden, dass

zur korrekten Errechnung des Outputs es reicht, dass sich der Agent einen Zähl-Wert H merkt, der aus der Summe aller vorigen Inputs gebildet wird. Dies bedeutet, dass in einem Zeitpunkt t dieser Zähler $H(t)$ rekursiv aus dem Zähler des vorigen Zeitschritts $H(t-1)$ und dem Input $\Lambda(t)$ gebildet werden kann. Die Gleichung 3.3 kann also modelliert werden mit

$$\begin{aligned} H(t) &= H(t-1) + \Lambda(t) \\ \Omega(t) &= \operatorname{sgn}(H(t)) \end{aligned} \tag{3.4}$$

Die Fitness $f(\varphi)$ des Agenten für einen Versuch mit $|\varphi| = n$ Input-Werten ergibt sich aus der Abweichung seines Outputs $\Omega(n)$ in Zeitschritt n von dem erwarteten Output $\hat{\Omega}(\varphi)$, berechnet nach Gleichung 3.3. Alle Outputs der vorigen Zeitschritte $\Omega(1), \ldots, \Omega(n-1)$ fließen nicht in die Bewertung ein. Der Versuch wird je m mal für ein $\varphi \in \{-1, 0, 1\}^n$ für $\nu \leq n \leq \mu$ durchgeführt. Die Gesamtfitness F des Agenten ist der Durchschnitt der Fitness-Werte $f(\varphi)$ aller Durchläufe:

$$\begin{aligned} f(\varphi) &= 1 - \tfrac{1}{2} \left| \hat{\Omega}(\varphi) - \Omega(n) \right| \\ \Phi_n &\subseteq \{-1, 0, 1\}^n, \quad |\Phi_n| = m \\ F &= \frac{1}{\mu - \nu} \sum_{n=\nu}^{\mu} \left[\frac{1}{m} \sum_{\varphi \in \Phi_n} f(\varphi) \right] \end{aligned} \tag{3.5}$$

In den folgenden Untersuchungen ist $m = 25$, $\nu = 1$ und $\mu = 9$.

Ein Agent, der per Zufall ein Wert aus $\{-1, 1\}$ als Lösung für das Counting-Problem liefert, erreicht im Durchschnitt eine Fitness von ≈ 0.5.

Durch die akkumulative Natur des Zählers H sollte selbst ein einfaches RNN noch in der Lage sein, diese Aufgabe zumindest in Ansätzen zu lösen. LSTMs und GRU-MBs sollten hierbei jedoch besser abschneiden.

In dieser Untersuchung wurden für jede GA-RNN Kombination je 20 Versuchsdurchläufe mit 200 Trainingsgenerationen durchgeführt.

In Abb. 3.6 sind die Resultate der Agenten mit einfachen RNNs zu sehen. Wie erwartet schneidet dabei ein einfaches Feedforward-NN ohne rekurrente Verbindungen nicht viel besser ab als ein zufallsbasierter Agent. Mit NEAT und ES-HyperNEAT kann selbst ein einfaches RNN noch die Aufgabe in adequatem Rahmen lösen. Dabei ist auch hier wieder ein geringfügiger Leistungsunterschied zwischen NEAT und ES-HyperNEAT zu Gunsten des Letzteren zu verzeichnen.

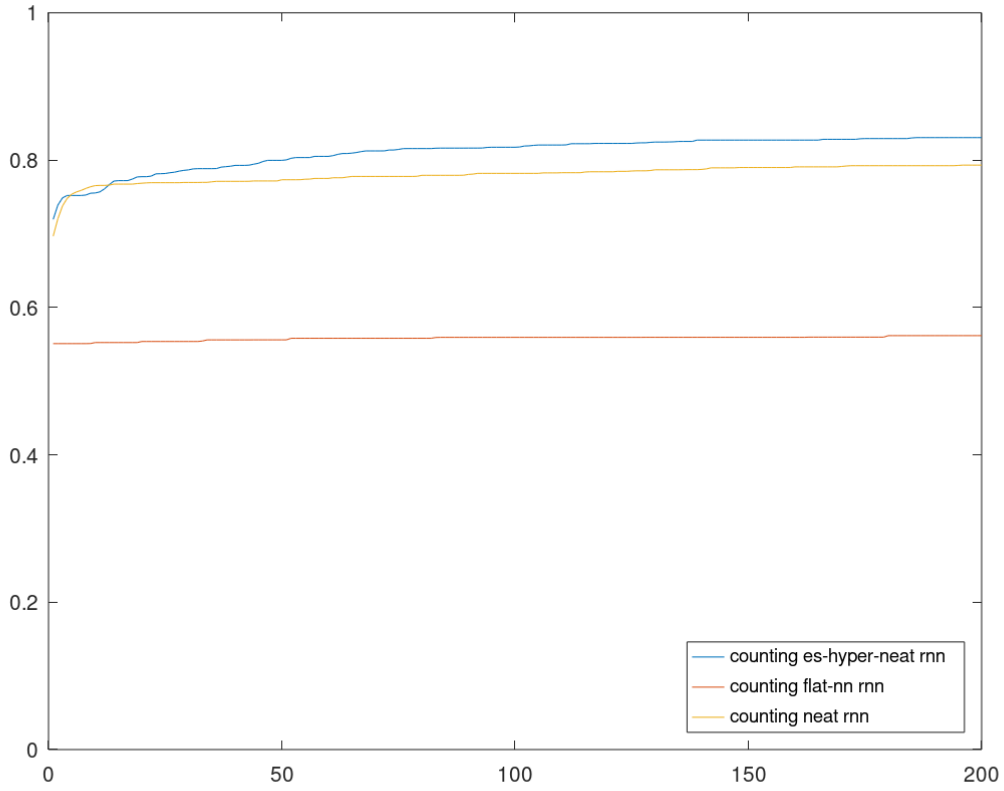


Abb. 3.6: Ergebnisse des ”Counting” Versuchs mit einfachen RNN. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

Wie in Abb. 3.7 zu sehen schneiden LSTM-Agenten auch bei ”Counting” deutlich besser ab als einfache RNN. Dabei erreichen NEAT und ES-HyperNEAT eine Fitness von über 0.95. Mit einfachen Feedforward-NNs dauert der Optimierungsprozess etwas länger, aber auch hier kann eine gute Leistung erreicht werden. Wieder schneidet ES-HyperNEAT etwas besser ab als klassisches NEAT.

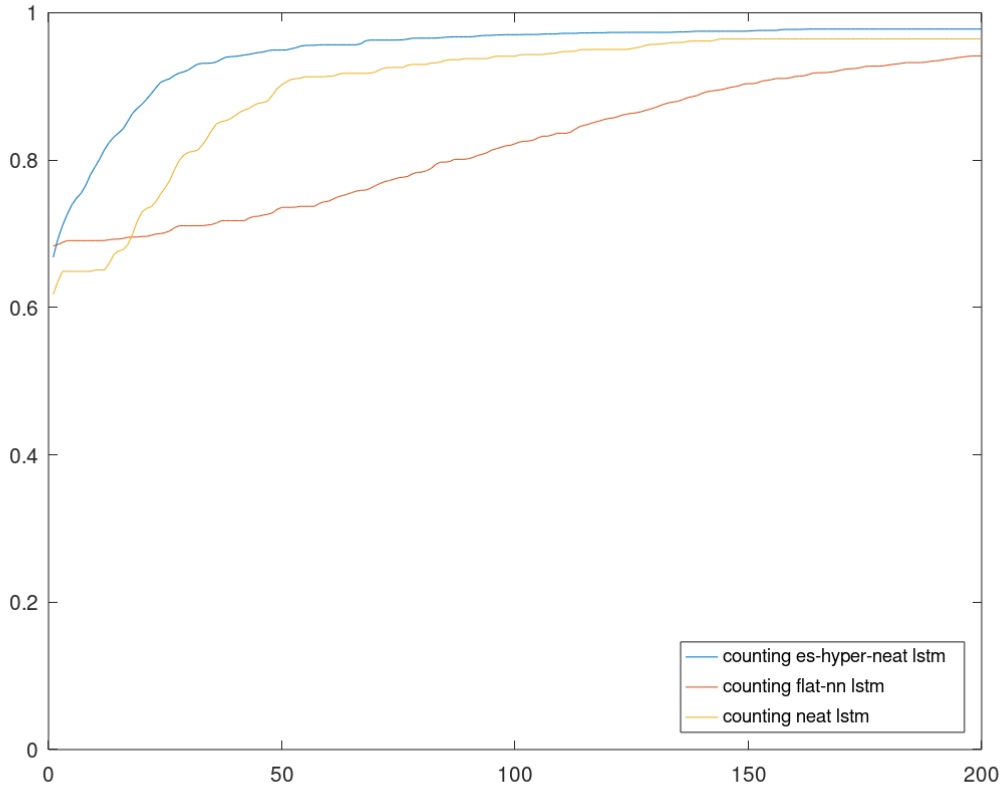


Abb. 3.7: Ergebnisse des "Counting" Versuchs mit LSTM. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

GRU-MB schneidet deutlich schlechter ab als LSTM. Dabei sind die Ergebnisse aller GAs gleichermaßen schlecht. Die Leistung von GRU-MB ist vergleichbar mit der eines einfachen RNNs.

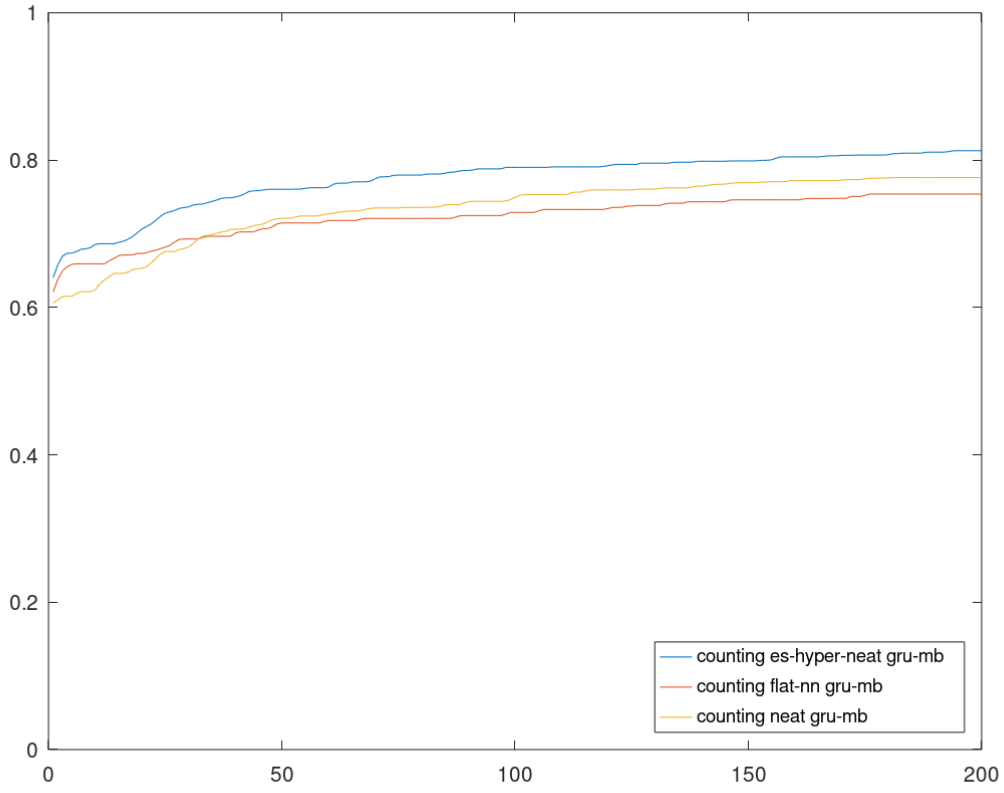


Abb. 3.8: Ergebnisse des "Counting" Versuchs mit GRU-MB. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

3.3 Sequence Recall

In dieser Untersuchung ist die Aufgabe des Agenten sich eine Sequenz $\varphi \in \{-1, 1\}^n$ von n Zahlen zu merken und anschließend wiederzugeben. Dazu wird der Testablauf in zwei Phasen eingeteilt. In Phase 1 wird dem Agenten in n Zeitschritten je eine Zahl der Sequenz φ gegeben. In Phase 2 wird von dem Agenten erwartet, dass er die in Phase 1 übertragenen Zeichen in n Zeitschritten rückwärts wiedergibt. Formal gesehen soll also die Sequenz S aller Inputs Λ der Phase 1 und Outputs Ω der Phase 2 (mit $S = \Lambda(1), \ldots, \Lambda(n), \Omega(n+1), \ldots, \Omega(n+n)$) ein Wort der folgenden kontextfreien Sprache sein:

$$\begin{aligned} S &\rightarrow 1\ S\ 1 \\ &\rightarrow -1\ S\ -1 \\ &\rightarrow \epsilon \end{aligned} \tag{3.6}$$

Ein Agent hat zwei Inputs $\Lambda \in \{-1, 0, 1\}^2$. Λ_1 ist der Werte-Eingang. Hier werden die Elemente aus φ in Phase 1 an den Agent übergeben. In Phase 2 ist Λ_1 konstant 0. Λ_2 ist das Lese-Signal.

Wenn der Agent Werte einlesen soll (Phase 1) ist $\Lambda_2 = 1$. Wenn der Agent gespeicherte Werte wiedergeben soll (Phase 2) ist $\Lambda_2 = -1$. Der Agent hat einen Ausgang $\Omega \in [-1, 1]$. Wenn $\Lambda_2 = 1$ (Phase 1), dann soll Ω 0 sein. Wenn $\Lambda_2 = -1$ (Phase 2) soll Ω die in Phase 1 über Λ_1 eingelesenen Werte wieder ausgeben.

Die Fitness $f(\varphi)$ eines Agenten basiert auf dem gewichteten Durchschnitt aller Abweichungen $\delta(t)$ des Outputs $\Omega(t)$ in allen Zeitschritten t von den erwarteten Output-Werten $\hat{\Omega}(\varphi, t)$. Dabei beschreibt $\alpha \in [0, 1]$ das Gewichtsverhältnis der Abweichungen in Phase 2 zu den Abweichungen in Phase 1. Im weiteren fließen in Phase 2 nur Abweichungen in die Wertung ein, bis die Abweichung in einem Zeitschritt t mehr als 1 ist - der Agent also deutlich vom Erwartungswert abweicht. Alle folgenden Outputs der Zeitschritte $\tau > t$ des Agenten werden nicht gewertet. Es werden wieder m Durchläufe mit je einem $\varphi \in \{-1, 1\}^n$ für $\nu \leq n \leq \mu$ durchgeführt:

$$
\begin{aligned}
\alpha' &= 1 - \alpha \\
\psi &= (\varphi_n, \ldots, \varphi_1) \\
\hat{\Omega}(\varphi, t) &= \begin{cases} 0 & ; \ t \leq n \\ \psi_{t-n} & ; \ t > n \end{cases} \\
\delta(t) &= |\Omega(t) - \hat{\Omega}(\varphi, t)| \\
\lambda(t) &= \begin{cases} 0 & ; \ \exists \tau \in \mathbb{N}\colon \ n < \tau < t \ \wedge \ \delta(\tau) > 1 \\ 1 - \frac{1}{2}\,\delta(t) & \end{cases} \\
f(\varphi) &= \frac{1}{n} \sum_{t=1}^{n} \alpha' \lambda(t) \ + \ \frac{1}{n} \sum_{t=n+1}^{2n} \alpha \lambda(t) \\
\Phi_n &\subseteq \{-1, 1\}^n, \quad |\Phi| = m \\
F &= \frac{1}{\mu - \nu} \sum_{n=\nu}^{\mu} \left[\frac{1}{m} \sum_{\varphi \in \Phi_n} f(\varphi) \right]
\end{aligned}
\tag{3.7}
$$

In den folgenden Untersuchungen ist $m = 25$, $\nu = 2$, $\mu = 6$ und $\alpha = 0.8$.

Ein Agent, der nur zufällige Werte in jedem Zeitschritt liefert, erreicht eine durchschnittliche Fitness von ≈ 0.35.

Diese Aufgabe fordert ein exaktes Gedächtnis. Der Agent muss in der Lage sein alle eingegebenen Werte zu reproduzieren. Dies ist eine sehr anspruchsvolle Aufgabe bei der ein einfaches RNN schlecht abschneiden sollte.

Es wurden je 20 Durchläufe pro GA-RNN Kombination für 200 Generationen durchgeführt.

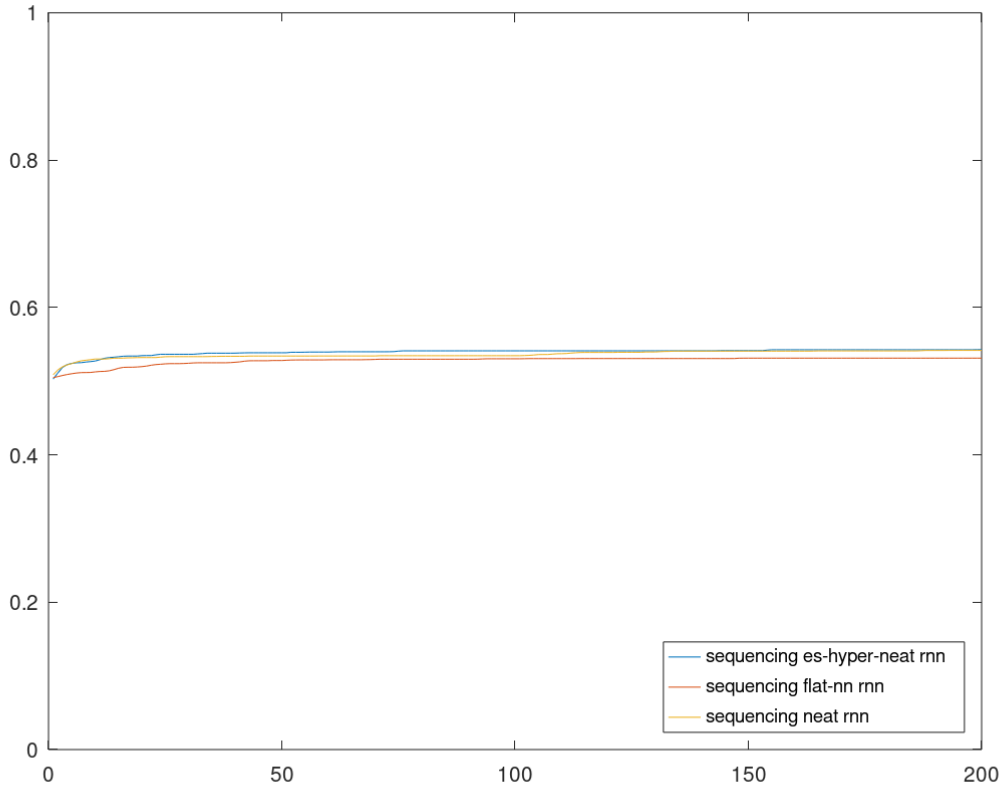


Abb. 3.9: Ergebnisse des "Sequence" Versuchs mit einfachem RNN. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

Wie erwartet kann ein einfaches RNN diese Aufgabe nicht lösen - unabhängig vom angewandten GA. Die Leistung eines einfachen RNN ist aber besser als die Leistung eines zufallsbasierten Agentens. Es sind keine nennenswerte Unterschiede zwischen den verschiedenen GAs festzustellen.

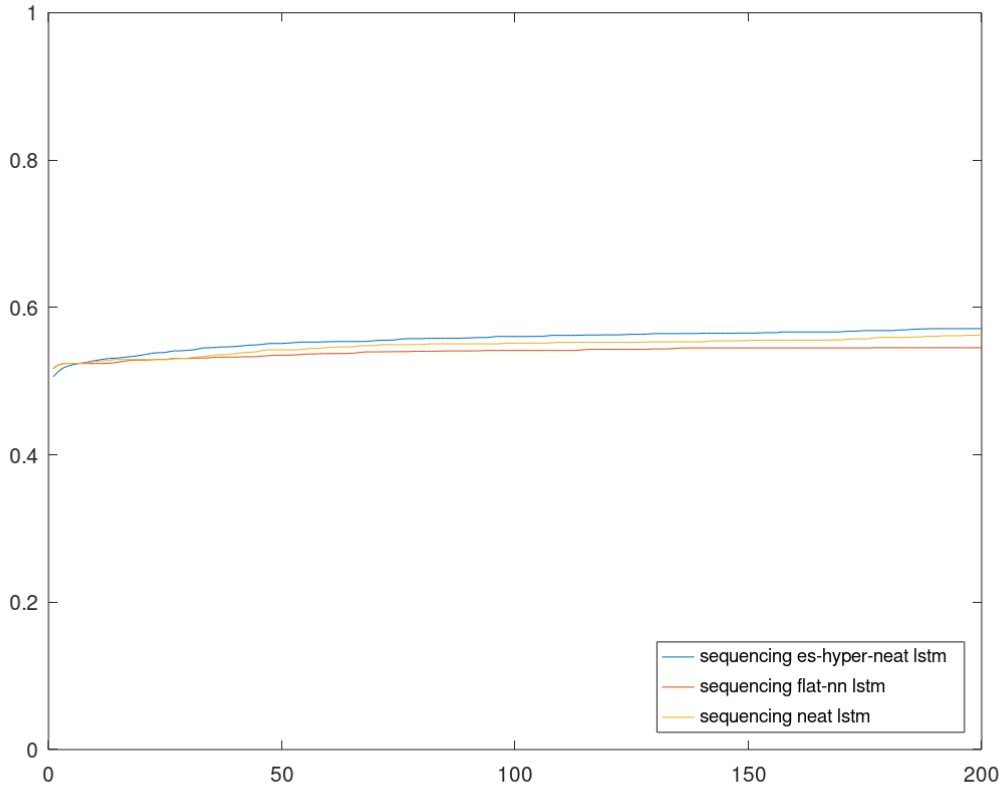


Abb. 3.10: Ergebnisse des ”Sequence” Versuchs mit LSTM. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

Auch LSTM-Agenten können die ”Sequence” Aufgabe nicht lösen. Ihre Leistung ist marginal besser als die Leistung einfacher RNN-Agenten. Es ist ein schwaches Wachstum der Fitness bei allen GAs zu verzeichnen.

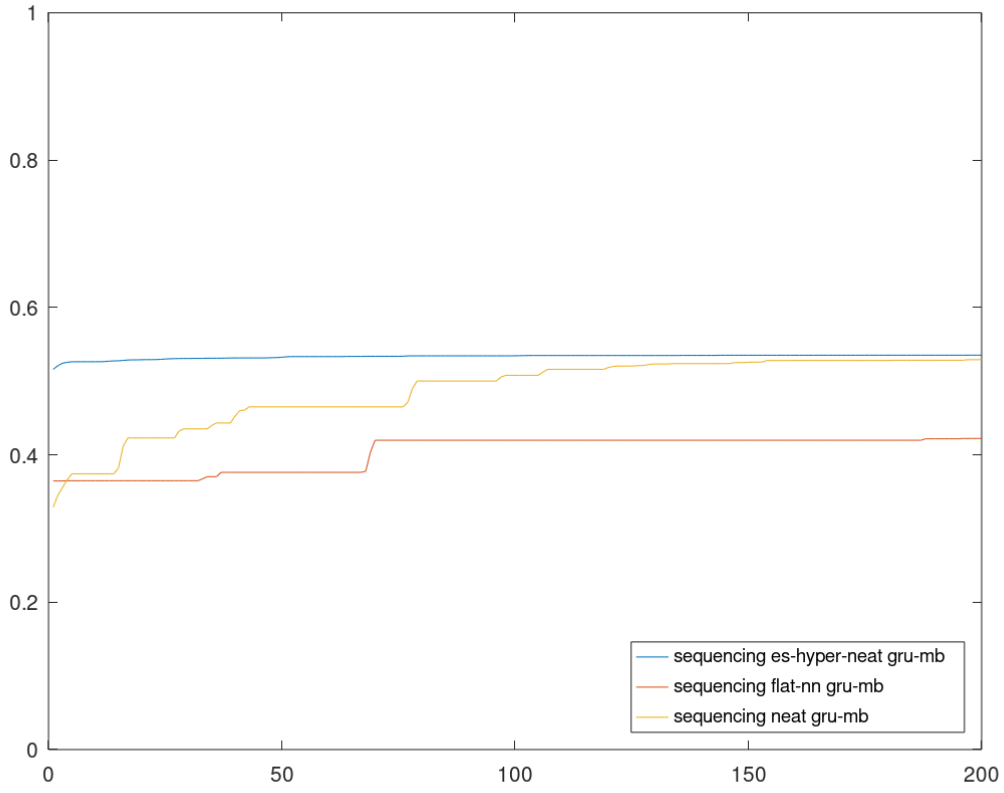


Abb. 3.11: Ergebnisse des ”Sequence” Versuchs mit GRU-MB. Aufgetragen ist die durchschnittliche Fitness in Abhängigkeit von der Trainingsgeneration.

Auch GRU-MB-Agenten können die ”Sequence” Aufgabe nicht lösen. Mit ES-HyperNEAT ist die Leistung von GRU-MB vergleichbar mit LSTM. Mit klassischem NEAT braucht GRU-MB deutlich länger um die selbe Leistung wie mit ES-HyperNEAT zu erreichen. Dies unterscheidet sich deutlich von LSTM-Agenten, bei denen auch mit klassischem NEAT innerhalb weniger Generationen das erreichte Fitness-Maximum approximiert werden konnte. Mit ”flat-nn” schneidet GRU-MB sogar deutlich schlechter ab als ein einfaches Feedforward-NN ohne rekurrente Verbindungen (einfaches RNN mit ”flat-nn”).

4 Fazit

Es konnte gezeigt werden, dass RNN mit GAs trainiert werden können. Sowohl Aufgaben mit geringen temporalen Anspruch an das Gedächtnis, wie auch Aufgaben, die eine Form von langfristigem, akkumulativen Gedächtnis fordern, konnten von allen getesteten GA-RNN Kombinationen gut bis souverän gelöst werden. Dabei hat ES-HyperNEAT das Fitness-Maximum etwas schneller erreicht als klassisches NEAT. Ansonsten konnte kein signifikater Leistungsunterschied zwischen den beiden GAs festgestellt werden.

Bei der "Sequence"-Aufgabe haben jedoch alle RNNs mangelhaft abgeschnitten. Eine mögliche Ursache ist die Trainingsdauer. Der "Sequence"-Versuch ist eine Abwandlung des *n-deep T-Maze Problem*, das in [5] zur Untersuchung der Leistung von GRU-MB verwendet wurde. In den Untersuchungen in [5] ist der Optimierungsprozess über 10.000 Generationen gelaufen. Im Vergleich dazu ist der "Sequence"-Versuch in dieser Arbeit aus Zeitmangel nur für 200 Generationen gelaufen. Bei einer Nachuntersuchung (mit nur eine Versuchsdurchführung - daher nicht-repräsentativ) von LSTM und GRU-MB mit "flat-nn" konnten beide RNN Architekturen nach 10.000 Trainingsgenerationen eine Fitness von 0.95 (LSTM) bzw. 0.75 (GRU-MB) erreichen - es war also eine deutliche Verbesserung der Leistung im Vergleich zu den Ergebnissen in 3.3 zu beobachten. Es liegt nahe, dass auch mit NEAT und ES-HyperNEAT eine bessere Leistung zu erzielen wäre, wenn mehr als 200 Trainingsgenerationen evaluiert werden würden. Diese Hypothese könnte in einer weiterführenden Arbeit untersucht werden.

Bei der Bestimmung der Hyper-Parameter ist aufgefallen, dass ES-HyperNEAT empfindlich gegenüber kleiner Änderungen der Hyper-Parameter ist. Dies manifestiert sich in Form von Stagnation (Fitness ändert sich kaum bis gar nicht) und Oszillation (die Fitness ändert sich erratisch - die Agenten konvergieren nicht). NEAT dagegen zeigt sich als relativ stabil gegenüber Änderungen der Hyper-Parameter. Die Nutzung von ES-HyperNEAT erfordert also eine besondere Sorgsamkeit bei der Wahl der Hyper-Parameter.

In [5] wurde GRU-MB vorgestellt und in Untersuchungen mit LSTM verglichen. Dabei schnitt GRU-MB besser ab als LSTM. In den Untersuchungen in dieser Arbeit lag die Leistung von GRU-MB-Agenten jedoch merklich unter der von LSTM-Agenten. Es ist warscheinlich, dass dies das Resultat einer mangelhaften Implementation ist. Die Implementation von GRU-MB war schwierig, da die Beschreibung der GRU-MB Architektur in [5] widersprüchlich ist. Konkret wird auf Seite 3 in den Gleichungen (5) der Block-Input $p^l = \text{sigm}\left\{K_p x^l + N_p m^{t-1} + b_p\right\}$ in

Abhängigkeit des Inhalts des *Hidden State* m^{t-1} aus dem vorigen Zeitschritt definiert. Auf der Info-Graphik (Figure 2) auf Seite 4 dagegen, wird der Block-Input aus dem Output des vorigen Zeitschritts gebildet - nicht dem *Hidden State*.

Ein in den Untersuchungen merkbares Problem war die zum Bilden des lauffähigen Agenten nötige Zeit. Ganz besonders bei ES-HyperNEAT wächst die Zeit zum Bilden eines fertigen NN aus dem Genom des CPPNs exponentiell mit der Komplexität des CPPNs. Profiling zeigt, dass in den letzten Trainingsgenerationen das System fast 90% der Zeit mit dem Bilden des Substrats verbringt. Dies relativiert den Geschwindigkeitsvorsprung von ES-HyperNEAT gegenüber NEAT etwas: Es braucht zwar weniger Trainingsgenerationen um das Fitness-Maximum zu erreichen, aber deutlich mehr Zeit. Die ES-HyperNEAT Implementation in dieser Arbeit ist rein *Single-Threaded*. ES-HyperNEAT hat aber Parallelisierungspotenzial. Inwieweit Parallelisierung die Substratfindung in ES-HyperNEAT beschleunigen kann, könnte man in einer weiterführenden Arbeit erkunden.

Literaturverzeichnis

[1] Cho, Kyunghyun ; Merrienboer, Bart van ; Gulcehre, Caglar ; Bahdanau, Dzmitry ; Bougares, Fethi ; Schwenk, Holger ; Bengio, Yoshua: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* 2014

[2] Finkel, Raphael ; Bentley, Jon: Quad Trees: A Data Structure for Retrieval on Composite Keys. In: *Acta Inf.* 4 (1974), 03, S. 1–9

[3] Hochreiter, Sepp: Untersuchungen zu dynamischen neuronalen Netzen. (1991), 04

[4] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-term Memory. In: *Neural computation* 9 (1997), 12, S. 1735–80

[5] Khadka, Shauharda ; Chung, Jen J. ; Tumer, Kagan: Evolving Memory-augmented Neural Architecture for Deep Memory Problems. In: *Proceedings of the Genetic and Evolutionary Computation Conference.* New York, NY, USA : ACM, 2017 (GECCO '17), S. 441–448. – URL http://doi.acm.org/10.1145/3071178.3071346. – ISBN 978-1-4503-4920-8

[6] Klimkov, V. ; Moinet, A. ; Nadolski, A. ; Drugman, T.: Parameter Generation Algorithms for Text-To-Speech Synthesis with Recurrent Neural Networks. In: *2018 IEEE Spoken Language Technology Workshop (SLT)*, Dec 2018, S. 626–631. – ISSN null

[7] Kolen, J. F. ; Kremer, S. C.: *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies.* S. 237–243. In: *A Field Guide to Dynamical Recurrent Networks*, IEEE, 2001. – URL https://ieeexplore.ieee.org/document/5264952. – ISBN null

[8] Li, Xiangang ; Wu, Xihong: *Constructing Long Short-Term Memory based Deep Recurrent Neural Networks for Large Vocabulary Speech Recognition.* 2014

[9] Mitchell, Melanie: *An Introduction to Genetic Algorithms.* Cambridge, MA, USA : MIT Press, 1996. – 2 S. – ISBN 0-262-13316-4

[10] Risi, Sebastian ; Lehman, Joel ; Stanley, Kenneth: Evolving the placement and density of neurons in the HyperNEAT substrate. In: *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10*, 01 2010, S. 563–570

[11] Risi, Sebastian ; Stanley, Kenneth: Enhancing ES-HyperNEAT to evolve more complex regular neural networks. In: *Genetic and Evolutionary Computation Conference, GECCO'11*, 01 2011, S. 1539–1546

[12] Risi, Sebastian ; Stanley, Kenneth O.: An Enhanced Hypercube-based Encoding for Evolving the Placement, Density, and Connectivity of Neurons. In: *Artif. Life* 18 (2012), Oktober, Nr. 4, S. 331–363. – URL http://dx.doi.org/10.1162/ARTL_a_00071. – ISSN 1064-5462

[13] Sak, H. ; Senior, Andrew ; Beaufays, F.: Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH* (2014), 01, S. 338–342

[14] Stanley, K. O. ; Miikkulainen, R.: Evolving Neural Networks through Augmenting Topologies. In: *Evolutionary Computation* 10 (2002), June, Nr. 2, S. 99–127. – ISSN 1063-6560

[15] Stanley, Kenneth: Compositional pattern producing networks: A novel abstraction of development. In: *Genetic Programming and Evolvable Machines* 8 (2007), 06, S. 131–162

[16] Stanley, Kenneth O. ; D'Ambrosio, David B. ; Gauci, Jason: A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. In: *Artificial Life* 15 (2009), Nr. 2, S. 185–212. – URL https://doi.org/10.1162/artl.2009.15.2.15202. – PMID: 19199382

A Hyper-Parameter

Die Populationsgröße war 150. 20% einer Population wurde durch asexuelle Reproduktion erzeugt. Eine Spezies gilt als "groß" wenn sie größer als 5% der Population ist. Die besten 7.5% der Population galten als "Elite" und wurde unverändert in die Nachfolgepopulation übernommen.

A.1 NEAT

Die Warscheinlichkeit einer Mutationsoperation eine neue Kante in das Genom einzufügen war 5%. Die Warscheinlichkeit eine Kante zu Teilen war 3%. Bei einer kantengewichtsändernden Mutation wurde ein Faktor aus $[-1.5, 1.5]$ (gleichverteilt) aufaddiert. Die Warscheinlichkeit bei einer Kantenänderung ein neues Gewicht zu kriegen war 10%. Die Warscheinlichkeit eine deaktivierte Kante zu reaktivieren war 20%. Alle Hidden- und Output-Neuronen hatten die Aktivierungsfunktion tanh.

Die *Speciation*-Gewichte waren $w_D = w_E = 1$ und $w_W = 0.1$. Der *Speciation-Threshold* war $\hat{\delta} = 3$.

A.2 ES-HyperNEAT

Die minimale Anzahl an QuadMap Divisionen war 3, die maximale Anzahl 4. Der *Variance-Threshold* zur weiteren Teilung war 0.05. Der *Variance-Threshold* beim Pruning war ebenfalls 0.05. Der *Banding-Threshold* war 0.3. Es wurde 2 mal neue Hidden-Neuronen gesucht.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: **KESSENER**

Vorname: **DANIEL**

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Optimierung rekurrenter neuronaler Netze durch genetische Algorithmen der NEAT-Familie

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Osnabrück	**02.12.19**	
Ort	Datum	Unterschrift im Original