Joseph-König-Gymnasium
Holtwicker Str. 3-5
45721 Haltern am See

Haltern, den 28. Februar 2021

Facharbeit zum Thema

NeuroEvolution of Fixed Topologies

von Paul Schulte

Fach:	Informatik
Betreuender Lehrer:	Herr Sala
Jahrgangsstufe:	Q1
Bearbeitungszeitraum:	12. Januar – 01. März 2021

Inhaltsverzeichnis

Vorwort

Für diese Facharbeit ist meine Wahl auf das Thema Neuroevolution gefallen, da ich mich schon seit längerem mit Neuronalen Netzen beschäftige und das Arbeiten mit diesen sehr interessant finde. Besonders interessant finde ich hier die verschiedenen Lernverfahren, was mich auch für die Neuroevolution begeistert hat.

Die größte Schwierigkeit ist es hier - meiner Meinung nach - dieses so umfangreiche Thema auf einer minimalen Seitenzahl festzuhalten und zu erklären. An einigen Stellen muss daher auf detaillierte Erläuterungen verzichtet werden.

Ein weiteres Hinderniss ist es, dass das Testen sehr schwierig ist. Es fällt oft schwer, zwischen Zufall und der empirischen, eigentlichen Funktionsweise zu unterscheiden.

1. Einleitung

In den letzen Jahren und Jahrzenten wurde dem Thema Künstliche Intelligenz immer mehr Aufmerksamkeit geschenkt. Das Verlangen ein Abbild der menschlichen Intelligenz zu erstellen, um bisher von Menschenhand erledigte Aufgaben oder Entscheidungen automatisch zu erledigen, existiert schon lange. Dabei werden sogenannte Künstliche Neuronale Netze genutzt. Diese können mithilfe des in dieser Arbeit behandelten Verfahrens Neuroevolution evolviert werden, um bestimmte Probleme zu lösen. Anwendungsgebiete sind unter anderem Robotik, Simulation von künstlichem Leben und Verhaltensweisen und das Erlernen von Strategien, um in Spielen zu gewinnnen oder generell besser abzuschneiden.

Der erste Abschnitt (siehe Abschnitt 2) *Neuronale Netze* beschäftigt sich daher mit der Funktionsweise von Neuronalen Netzen und der künstlichen Kopie, sowie der Implementation dieser. Hier wird schon ein kleiner Ausblick auf Lernverfahren gegeben, um den nächsten Abschnitt einzuleiten.

Der zweite Abschnitt (siehe Abschnitt 3) *Neuroevolution* geht konkreter auf das Thema dieser Arbeit ein und handelt vom Lernverfahren Neuroevolution mit feststehenden Topologien. Die Implementation wird genauer erläutert und beschrieben.

Der dritte und letze Abschnitt (siehe Abschnitt 4.1) *Testen der Implementation* soll den im vorherigen Abschnitt implementierten Algorithmus ausprobieren und analysieren. Des weiteren wird ein Fazit gezogen und es folgt ein kleiner Ausblick auf neue Ideen und Konzepte auf dem Gebiet der Neuroevolution.

2. Neuronale Netze

2.1. Die Motivation und der Weg hinter Neuronalen Netzen

Wie so oft in der Informatik oder generell in den Naturwissenschaften orientiert sich bei künstlichen Neuronalen Netzen die technische Kopie am biologischen Vorbild, an den Neuronen im menschlichen Gehirn sowie an deren Verbindung und Zusammenspiel untereinander. Es ist somit kein Wunder, dass die erste Form einer künstlichen Intelligenz unter anderem von einem Neurowissenschaftler entwickelt wurde. Die erstmalige Definition eines künstlichen Neurons ist dabei auf den Neurophysiologen W.S. McCulloch und den Mathematiker W.Pitts im Jahre 1943 zurückzuführen (vgl. Bourg Anne 2006, S. 3).

2.2. Das biologische Neuron

Allgemein bekannt ist schon lange, dass das Menschliche Gehirn aus vielen kleinen Nervenzellen besteht, die als Neuronen bezeichnet werden (siehe Abb. 1). Diese kommunizieren untereinander und reagieren auf verschiedene Signale. So kommt es dazu, dass ungefähr 86 Milliarden solcher Neuronen gleichzeitig miteinander verbunden sind und Informationen austauschen (vgl. Steinwendner und Schwaiger 2019, S. 29).

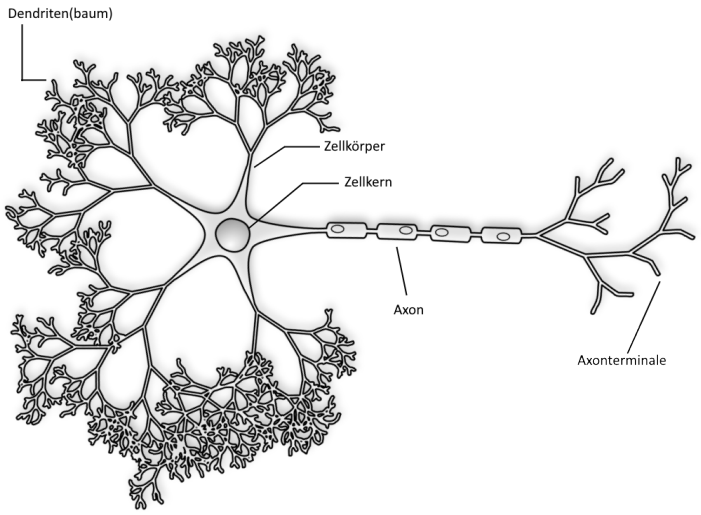

Abbildung 1: Das menschliche Neuron

Jedes Neuron hat dabei sogenannte *Dendriten*, die quasi als Eingang für die Signale und Informationen angesehen werden können. In den *Dendriten* kommen chemische Signale von anderen Neuronen an, die in ganz kleine elektrische Ströme umgewandelt werden. Von dort aus gelangen diese Ströme in den inneren *Zellkörper*. Dieser ist mit den *Axonen*

- bis zu einem Meter langen Fortsätzen der *Zellkörper* - verbunden, welche an die *Dendriten* anderer Neuronen münden und hier durch Neurotransmitter wie Noradrenalin, Acetylcholin, Dopamin oder Serotonin verschieden starke chemische Signale abgeben, die wieder in unterschiedliche elektrische Signale umgewandelt werden (vgl . Steinwendner und Schwaiger 2019, S. 29–30 & Rundfunk 2019). Es entsteht ein riesengroßes Netzwerk aus diesen Neuronen. Grundlegend ist hier auch, dass sowohl die *Dendriten*, als auch die *Axone* über *Synapsen* mit dem *Zellkörper* verbunden sind. Diese *Synapsen* sind für die Umwandlung von chemischen Signalen in elektrische verantwortlich und können das Signal hemmen oder erregen. Dadurch haben verschiedene Neurone unterschiedliche "Wichtigkeiten" oder auch Einflüsse.

2.3. Das künstliche Neuron

Um sich dem biologischen Neuron mit der mathematischen Kopie anzunähern, hilft es eine vorerst simplifizierte Darstellung zu wählen. Das folgende Modell bezieht sich nur auf den Teil des biologischen Neurons von den *Dendriten* bis zu den *Zellkörpern*, die dann wieder als Eingabesignal für andere *Dendriten* verwendet werden (s. Abb. 2).

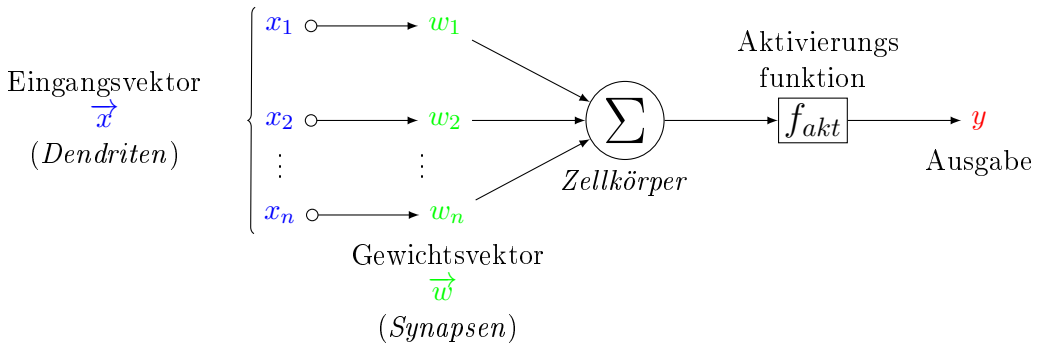

Abbildung 2: Ein künstliches Neuron

Generell werden alle Elemente des künstlichen Neurons mithilfe der Mathematik realisiert, angefangen bei den *Dendriten*, die als Eingangsvektor $\overrightarrow{x}$ dargestellt werden. Dieser besteht aus n Elementen, sein letzes Element ist x_n . Die Synapsen finden Entsprechung in den sogenannten *Gewichten*, die hier als Gewichtsvektor $\overrightarrow{w}$ angegeben sind. Das Signal, das dann im *Zellkörper* ankommt, ist die *gewichtete Summe* aller Produkte zwischen x_i und w_i :

$$\sum_{i=1}^{n} x_i \cdot w_i$$

Dieser im *Zellkörper* ankommende "Reiz" wird dann in eine *Aktivierungsfunktion* weitergeleitet. Es gibt sehr viele unterschiedliche Aktivierungsfunktionen - lineare und nicht

lineare - , die für eine feinere Abstufung des Ergebnisses oder eine klare Trennung zwischen 1 und 0, wahr und falsch, Hund oder Katze zu gebrauchen sind. Geläufig ist die *Sigmoid-Funktion* (siehe Abb. 3), weil ihre Ausgabe zwischen 0 und 1 liegt. Das ist besonders nützlich, wenn es um eine Wahrscheinlichkeitsvorhersage geht.

$$sig(t) = \frac{1}{1 + e^{-t}}$$

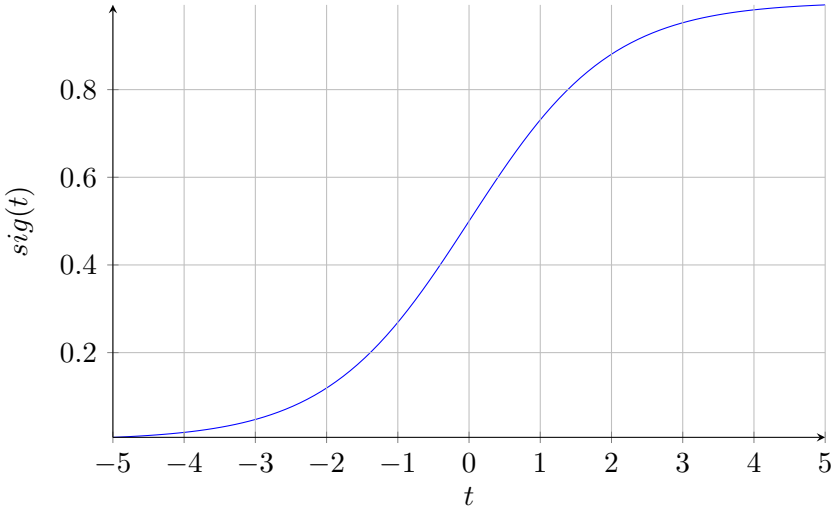

Abbildung 3: Sigmoid Funktion

Diese Sigmoid-Funktion lässt sich auch implementieren. Hierbei wird die mathematische Bibliothek *numpy* verwendet, da diese Operatoren oder Terme auf Listen oder Matrizen anwenden kann. Liegt zum Beispiel eine Matrix mit gewichteten Summen vor, lässt sich diese Funktion direkt auf jeden Wert in der Matrix gleichzeitig anwenden (siehe Abb. 4).

```
1 import numpy as np
2
3  def sigmoid(x):
4     return 1/(1+np.exp(-x))
```

Abbildung 4: Sigmoid-Funktion Implementation

(vgl. Steinwendner und Schwaiger 2019, S. 30–32)

2.4. FeedForward-Netz

Um komplexere Aufgaben zu lösen, wird mehr als ein einziges Neuron benötigt. Es müssen mehrere Neuronen miteinander verbunden werden, dass sie - genauso wie auch beim Menschlichen Gehirn - untereinander kommunizieren können. Es gibt hierbei viele verschiedene Arten, die Neuronen untereinander zu verbinden und sie in unterschiedlichen

Schichten oder Strukturen - auch *Topologien* - auszurichten. Das Verfahren *Neuroevolution* braucht hauptsächlich das sogenannte *Mehrschichtige FeedForward-Netz*. Es liegen mehrere nacheinander verkettete Schichten von Neuronen vor, die von links nach rechts befeuert werden (siehe Abb. 5).

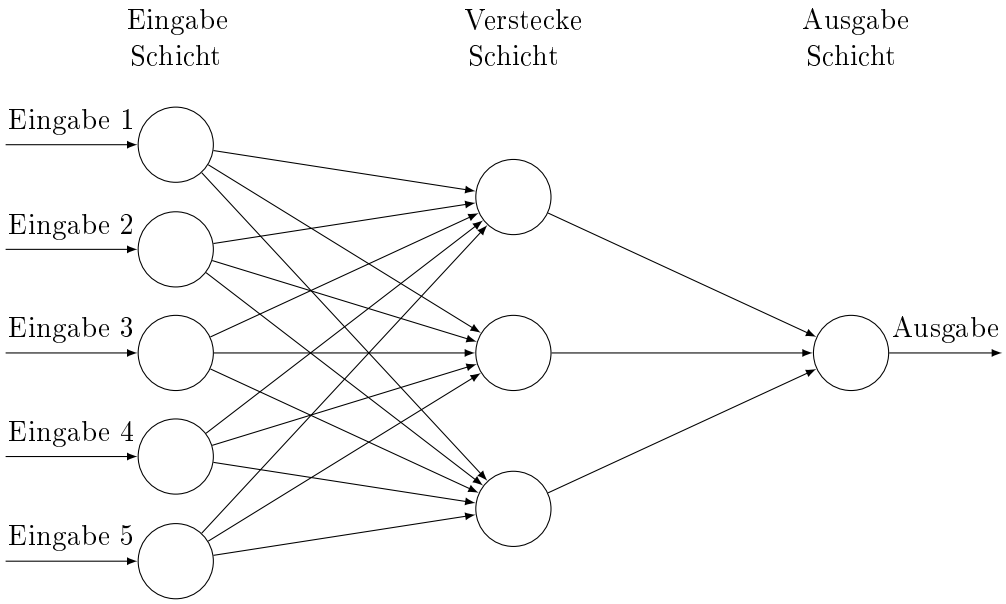

Abbildung 5: Mehrschichtiges Feedforward Netz

Diese Mehrschichtigen FeedForward Netze bestehen - wie der Name verrät - aus mehreren Schichten, dessen erste als *Eingabeschicht* verstanden wird. Die letzte ist logischerweise als *Ausgabeschicht* zu bezeichnen. Alle Schichten dazwischen - hier können auch mehrere auftreten - werden als *"versteckte Schichten"* (engl.: hidden layers) bezeichnet. Dieser Name kommt daher, dass der Benutzer, der das Netz aufsetzt, quasi keine Interakion mit diesen Schichten hat und diese so als "Blackbox" verstanden werden können. Um ein solches Neuronales Netz zu implementieren, reicht es, die Gewichte und die Aktivierungsfunktionen zu speichern, denn die Eingabewerte sind immer unterschiedlich (siehe Abb. 6).

(vgl. Svozil, Kvasnicka und Jiri 1998)

```
# Neuronales Netz Klasse
class neural_network:
        # Netzwerk initialisieren
        def __init__(self,network):
            self.weights = []
            self.activations = []
            for layer in network:
                if layer[0] == None:
                    input_size = network[network.index(layer)-1][1]
                else:
                    input_size = layer[0]
                output_size = layer[1]
                activation = layer[2]
                self.weights.append(np.random.randn(input_size,output_size))
                self.activations.append(activation)
```

Abbildung 6: Neuronales Netzwerk Implementation

2.5. Propagation

Links, am Anfang des Netzes in der Eingabeschicht, werden Eingaben getätigt. Hier werden zum Beispiel die Pixel eines Bildes eingegeben. Dann finden die oben benannten Berechnungen der *gewichteten Summen* in jedem Neuron statt, die durch eine *Aktivierungsfunktion* laufen und dann an das jeweilige verbundene Neuron in der ersten *verstecken Schicht* weitergeleitet werden. Dieser Prozess findet solange statt, bis das *Ausgabeneuron* - auch hier kann es mehrere geben - erreicht ist. Die Ausgabe findet in Form von Zahlen statt (siehe Abb. 7). Ob auf dem eingegebenen Bild laut dem Neuronalen Netz ein Hund oder eine Katze zu erkennen sein soll, wird zum Beispiel durch eine Zahl von 0 bis 1 ausgegeben. Die *Ausgabeneuronen* sind für die Überlieferung oder Ausgabe der Aktion oder Entscheidung des Netzes verantwortlich.

```
        # Einschaetzung des Neuronalen Netzes
        def propagate(self,data):
            inputs = data
            for i in range(len(self.weights)):
                z = np.dot(inputs,self.weights[i])
                a = self.activations[i](z)
                inputs = a
            yhat = a
            return yhat
```

Abbildung 7: Befeuerung - Propagation Implementation

(vgl. Zell 2003)

Lernstrategien

Bisher wurde beschrieben, wie ein Feedforward Netz funktioniert. Analogisch zum Menschen muss aber auch jedes Neuronale Netz erst einmal etwas lernen. Unter "Lernen" versteht man bei künstlichen Neuronalen Netzen das Anpassen der Gewichte. Diese bleiben bei der Propagation immer gleich, unabhängig vom Eingangsvektor. Wie effizient ein Netz ist, hängt demnach nicht nur von der *Topologie* ab, sondern auch von den *Gewichtsvektoren.*

Bei der Arbeit mit Neuronalen Netzen und zum Beispiel der Klassifizierung zwischen Hund und Katze anhand eines Bildes, muss zunächst mit ernüchternden Ergebnissen gerechnet werden. Ohne Training trifft das Neuronale Netz nur willkürliche Entscheidungen - wie ein kleines Kind, das noch nicht weiß, was Hunde von Katzen unterscheidet.

Um die Gewichte entsprechend immer besser anzupassen, gibt es zahlreiche Lernstrategien, die unterschiedliche Anwendungsgebiete finden. Bekannt ist die sogenannte *Backpropagation.* Hierbei wird händisch eine Klassifizierung von zum Beispiel 10.000 Hundebildern sowie 10.000 Katzenbildern erstellt. Das Netz kriegt immer ein Eingabebild, das es klassifizieren soll. Diese Vorraussagung des Netzes wird anschließend mit dem wahren Inhalt des Bildes (Hund oder Katze) verglichen und es findet eine *Fehlerauswertung* mithilfe des Verfahrens *Gradient descent* statt, die die Gewichte immer besser anpasst, sodass nach tausenden von Wiederholungen eine richtige Klassifizierung sehr viel wahrscheinlicher wird. Je mehr ein Neuronales Netz trainiert wird und je mehr Eingabedaten dabei verwendet werden, desto präziser ist das Ergebnis, also seine Fähigkeit, die richtige Aussage über die Eingabedaten zu treffen (vgl. Steinwendner und Schwaiger 2019, 149ff).

Das Verfahren *Neuroevolution* ist ebenfalls eine Lernstrategie, basiert aber im Gegensatz zu der Backpropagation auf naturähnlicher Evolution der Neuronalen Netzwerke.

3. Neuroevolution

Im Gegensatz zum *Backpropagation*-Verfahren benötigt die Neuroevolution nicht unbedingt unzählige korrekte "Lerndaten". Soll der Algorithmus zum Beispiel das Spielen eines Spiels lernen, reicht es nach jedem Durchgang eine Rückmeldung (engl.: "fitness") über den Erfolg mit dem aktuellen Neuronalen Netz zu erhalten. Daraufhin findet eine eventuelle Anpassung der Gewichte oder der Topologie statt (siehe Abb. 8) (vgl. Stanley 2017, Eiben und Smith 2016, S. 25–28).

Die Veränderung der Topologie durch das Neuroevolutionsverfahren ist erst später entstanden, hier fehlte noch das richtige System - die Topologienveränderung führt schnell zu vielen Problemen. In den nächsten Abschnitten wird die Neuroevolution deswegen zunächst mit festen Topologien (engl.: "fixed topologies") implementiert.

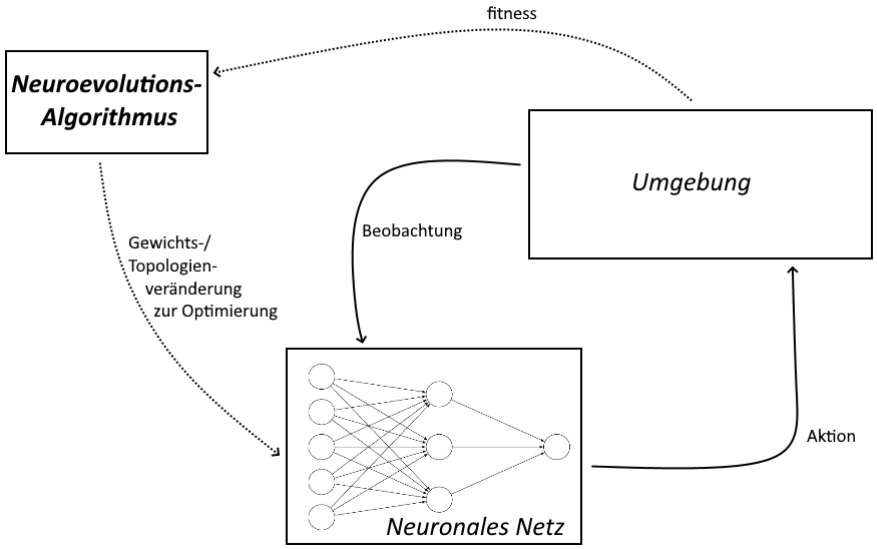

Abbildung 8: Neuroevolution

3.1. Generationsbegriff

Neuroevolutionsverfahren werden auch als *evolutionäre Algorithmen* bezeichnet. Diese Art von Algorithmen orientiert sich an der Natur und ist generationsbasiert. Jeder Lernschritt findet in Form einer *Generation* statt. Jede einzelne Generation besteht aus einer Bevölkerung von vielen einzelnen "Agenten", die mit verschiedenen Neuronalen Netzen unterschiedliche Lösungsansätze für das zu lösende Problem anbieten (siehe Abb. 9). Die Neuronalen Netze und somit die Agenten der ersten Generation werden meistens per Zufall generiert (siehe Abb. 10). Es wird nicht mehr nur ein Neuronales Netz wie bei der *Backpropagation* verwendet, sondern mehrere - manchmal bis zu tausend Stück - danach wird ausgewählt, welches Netz am besten abgeschnitten hat (vgl. Eiben und Smith 2016, S. 25–28).

```
1 # Agent Klasse
2 class Agent:
3     def __init__(self,network):
4         self.neural_network = neural_network(network)
5         self.fitness = 0
```

Abbildung 9: Agent Implementation

```
def generate_agents(pop_size, network):
    return [Agent(network) for _ in range(pop_size)]
```

Abbildung 10: Zufällige Generationserzeugung Implementation

3.2. Fitness-function

Die einzelnen Agenten interagieren während einer Generation auf ihre Art und Weise mit der Umgebung, solange bis die Umgebung eine Rückmeldung - zum Beispiel über den Erfolg in einer bestimmten Situation - liefert. Dafür wird die sogenannte *"Fitness function"* genutzt, welche von der Umgebung realisiert wird und von Problem zu Problem unterschiedlich ist. Mithilfe einer Zahl wird hier jedem Agenten eine Information gegeben, wie gut er mit seiner Strategie - seinem Neuronalen Netz - abgeschnitten hat. Je höher die Zahl ist, desto besser war die Strategie (vgl. Eiben und Smith 2016, S. 30).

3.3. Selection

Um die einzelnen Agenten zu verbessern und besser auf den nächsten Durchgang der Umgebung vorzubereiten, wird direkt die nächste Generation erstellt. Zur bestmöglichen Anpassung der Agenten werden daher zunächst diejenigen herausgesucht, die in der vorherigen Generation am besten abgeschnitten haben - sprich diejenigen, die die höchsten *fitness*-Werte hatten. Dieser Prozess nennt sich *Selection*, wie nach Charles Darwin überleben die Anpassungsfähigsten oder zumindest die, die besser abschneiden, als die anderen.
Zur Auswahl der besten Lösungen gibt es viele Ansätze. Bei meiner Implementation werden alle Agenten der gesamten Generation zuerst sortiert. Danach werden die besten 50% zum "Überleben" ausgewählt, der Rest wird verworfen (siehe Abb. 11) (vgl. Eiben und Smith 2016, S. 31 – 33).

```
def selection(agents):
    agents = sorted(agents, key=lambda agent: agent.fitness, reverse=True)
    agents = agents[:int(0.5 * len(agents))]
    return agents
```

Abbildung 11: Selection Implementation

3.4. Crossover

Der nächste Schritt der Reproduktion der alten und somit Erzeugung der neuen Generation nennt sich "Crossover" - zu Deutsch "Kreuzung". Für den nächsten Durchgang

wird wieder eine gesamte Population benötigt, die es erst zu erstellen gilt. Für das Crossover gibt es ebenfalls viele unterschiedliche Verfahren und Vorgehensweisen. In meiner Implementation wird zunächst der Gewinner mit dem höchsten Fitness-Wert der letzen Generation ohne Änderungen übernommen, danach wird Agent um Agent aufgefüllt, bis die Populationsgröße wieder erreicht ist.

Bei jedem Agenten, der in diesem Prozess hinzugefügt wird, wird zwischen Klonen und Kreuzen unterschieden. Zu einer 25%-igen Wahrscheinlichkeit wird ein Agent aus der alten Generation ohne Kreuzvorgang übernommen. Trifft hingegen die andere, 75%-ige Wahrscheinlichkeit ein, werden - wie vorher beschrieben - zwei Eltern aus der vorherigen Generation gesucht und gekreuzt, indem die Gewichte der beiden gemischt werden. Für diese Kreuzung zweier Gene in Form von Gewichten wird ein Punkt per Zufall ausgesucht, an dem die Gewichte des ersten und des zweiten Elternteils geteilt werden. Das Kind bekommt dann einen Teil des Gewichtsvektors vom ersten Elternteil und den anderen Teil vom anderen Elternteil, sodass wieder ein vollständiger, gekreuzter Gewichtsvektor entsteht (siehe Abb. 12).

Abbildung 12: Crossover - Kreuzung zweier Neuronale Netze

Unabhängig davon, ob nur geklont oder auch gekreuzt wurde, wird dann bei jedem neu entstandenen Kind zu einer festgelegten Wahrscheinlichkeit eine Mutation durchgeführt (siehe Abschnitt 3.5).

Problematisch erscheint es, dass die Gewichte nicht als eine simple lineare Liste vorliegen, sondern teilweise auch als komplexe Matrix. Damit diese gekreuzt werden können, müssen sie erst in eine lineare Liste konvertiert werden (siehe Abb. 20). Bevor das geschieht, werden die Größen der Dimension in einer Variable festgehalten, um diese Dimension später wiederherstellen zu können. Danach wird unter anderem die Funktion `flatten()` der Bibliothek *numpy* verwendet, die einen mehrdimensionalen Array in eine einzige Dimension herunterbrechen kann. Nun kann diese lineare Liste an einem zufälligen Index gespalten werden und nur einen Teil für den Nachwuchs zur Verfügung stellen.

Das entstandene Kind hat nun Gene sowohl vom einen als auch vom anderen Elternteil. Um die oben beschriebene Zurückführung in die eigentliche Dimension zu erledigen, wird die Funktion `unflatten()` definiert. Sie nimmt den rohen linearen Array und den strukturellen Aufbau entgegen und gibt die zurückgeführte Matrix zurück (siehe Abb. 13).

```
def unflatten(flattened,shapes):
    newarray = []
    index = 0
    for shape in shapes:
        size = np.product(shape)
        newarray.append(flattened[index : index + size].reshape(shape))
        index += size
    return newarray
```

Abbildung 13: Unflatten-Funktion Implementation

Das neu erzeugte Kind wird als neuer Agent in die Population eingepflegt. Ist die Population voll, dann ist der Crossovervorgang abgeschlossen (vgl. Eiben und Smith 2016, S. 31 – 33).

Rückwirkend sollte noch einmal auf die Auswahl der Elternpaare eingegangen werden. Diese werden nicht nur zufällig aus der Menge der "überlebenden Hälfte" ausgewählt, sondern nach der Exponentialverteilung gewählt. Hier haben in der vorherigen Generation besser Abschneidende eine höhere Wahrscheinlichkeit ausgewählt zu werden, als schlechter Abschneidende. Die aus dem Crossover hervorgehenden Agenten sind daher potentiell eher von den besseren Agenten der vorherigen Generation abhängig. Die Exponentialverteilung kann durch viele Methoden implementiert werden. Für die Generierung von exponentiell abnehmenden Zufallszahlen wie in diesem Beispiel verwendet man dabei unter anderem die *Wahrscheinlichkeitsdichtefunktion*. Diese ist definiert als:

$$f(x;\lambda) = \begin{cases} \lambda e^{-(\lambda x)} & x \geq 0, \\ 0 & x < 0. \end{cases} \tag{1}$$

Dabei ist λ ein Parameter für die Verteilung. Damit für jede beliebige Populationsgröße immer eine gleichmäßig leicht nach links geneigte Wahrscheinlichkeitsdichte auftritt, - sei n definiert als Populationsgröße - wird lambda mit

$$\lambda = n^{-1} = \frac{1}{n} \tag{2}$$

von der Populationsgröße abhängig gemacht. Diese Wahrscheinlichkeitsdichtefunktion wird dann auf jeden Index eines möglichen Elternteils angewandt, sodass sich ein Array mit unterschiedlichen Wahrscheinlichkeiten zur Auswahl ergibt, in dem die weiter links orientierten Indexe, welche den besseren Agenten der letzten Generation entsprechen, eine höhere Wahrscheinlichkeit haben, für den Crossover gewählt zu werden. Implementiert wird diese Wahrscheinlichkeitsdichtefunktion mithilfe der Bibliothek *scipy.stats* und der Funktion `expon.pdf(x, scale)`, die mit x den Populationsarry mit Indexen und mit *scale* ein invertiertes lambda, $\frac{1}{\lambda}$ entgegennimmt. Anschließend wird eine gleichmäßig verteilte Zufallszahl erzeugt und mit der Gesamtsumme der Wahrscheinlichkeitsdichtefunktion multipliziert. Der zurückzugebene und ausgewählte Index ist der, der links vom Start

des Arrays aus im Bereich dieser Zufallszahl liegt (siehe Abb. 14). (vgl. *Exponential distribution* 2021, *scipy.stats.expon* o. D.)

```
def get_expon_dist_random(n):
    inv_l = 1.0/(n**float(-1))
    x = np.array([i for i in range(0,n)])
    p = expon.pdf(x, scale=inv_l)
    rand = np.random.random() * np.sum(p)
    for i, p_i in enumerate(p):
        rand -= p_i
        if rand < 0:
            return i
    return 0
```

Abbildung 14: Zufall per Exponentialverteilung Implementation

Zusammengefasst sieht die Crossover Funktion wie folgt aus (siehe Abb. 15).

```
def crossover(agents,network,pop_size):
    offspring = []
    offspring.append(agents[0])
    while(len(offspring) < pop_size):
        parent1 = agents[get_expon_dist_random(len(agents))]
        if np.random.uniform() > 0.75:
            offspring.append(parent1)
        else:
            parent2 = agents[get_expon_dist_random(len(agents))]
            child1 = Agent(network)
            shapes = [a.shape for a in parent1.neural_network.weights]
            genes1 = np.concatenate([a.flatten() for a in parent1.
                neural_network.weights])
            genes2 = np.concatenate([a.flatten() for a in parent2.
                neural_network.weights])
            split = random.randint(0,len(genes1)-1)
            child1_genes = np.array(genes1[0:split].tolist() + genes2[split
                :].tolist())
            child1.neural_network.weights = unflatten(child1_genes,shapes)
            offspring.append(child1)
        offspring[-1] = mutation(offspring[-1])
    return offspring
```

Abbildung 15: Crossover-Funktion Implementation

3.5. Mutation

Wie bereits in Abschnitt 3.4 angerissen wurde, wird für jedes neue Kind und jeden neu erzeugten Agenten eine *Mutation* durchgeführt. Diese hat den Sinn neue Variation und

Innovation einzubringen. Zu einer bestimmten Wahrscheinlichkeit, die sich in dieser Implementation auf 10% beläuft, wird ein zufälliges Gewicht durch einen ebenfalls per Zufall erzeugten Wert ersetzt. Dabei wird dieselbe Technik der Konvertierung und Zurückführung in eine lineare Liste benutzt, wie bereits oben beschrieben (siehe Abb. 16). Die Größe der Wahrscheinlichkeit muss dabei vorsichtig gewählt werden. Wenn sie zu groß ist, wird ein zu unbeständiger, fast zufälliger Verlauf erscheinen. Wählt man die Wahrscheinlichkeit zu niedrig, entwickelt sich das System zu langsam, da keine Innovation vorhanden ist, auf der eine neue Vorgehensweise oder Strategie basieren könnte, die zur Lösung des Problems beiträgt. Außerdem assimilieren sich die unterschiedlichen Agenten bei einer zu geringen Mutationsrate viel schneller, sodass der Lernprozess noch länger dauert (vgl. Eiben und Smith 2016, S. 31 – 32).

```
def mutation(agent):
    if random.uniform(0.0, 1.0) <= 0.1: # 10% Chance auf Mutation
        weights = agent.neural_network.weights
        shapes = [a.shape for a in weights]
        flattened = np.concatenate([a.flatten() for a in weights])
        randint = random.randint(0,len(flattened)-1)
        flattened[randint] = np.random.randn()
        newarray = []
        indeweights = 0
        for shape in shapes:
            size = np.product(shape)
            newarray.append(flattened[indeweights : indeweights + size].
                reshape(shape))
            indeweights += size
        agent.neural_network.weights = newarray
    return agent
```

Abbildung 16: Mutation-Funktion Implementation

4. Testen der Implementation und Fazit

4.1. Der Lernprozess am Beispiel der "Auto-Stab" Umgebung

Um den Implementierten Algorithmus zu testen, wird eine Umgebung benötigt. Das Unternehmen OpenAI aus den USA hat zum Testen von Lernverfahren und besonders zum Testen von reinforcement-learning Verfahren wie Neuroevolution das open source Projekt *gym* gestartet. Dieses umfasst mehrere Beispielumgebungen, die für das Testen der Lernverfahren optimiert sind. Um den wie oben beschrieben implementierten Neuroevolutionsalgorithmus zu testen, wird die Umgebung "CartPole-v1" verwendet. Diese enthält einen Wagen, der sich nach links oder nach rechts bewegen kann. Das Neuronale Netz als Agent muss vor jedem Frame und zu renderndem Bild eine Entscheidung treffen: nach

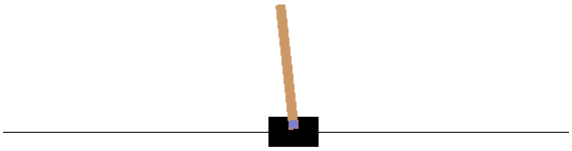

Abbildung 17: CartPole

links fahren (0) oder nach rechts fahren (1). Es reicht daher ein einziges Ausgabeneuron. Zentral auf dem Wagen befindet sich ein Stab, den es mithilfe des Fahrverhaltens auszubalancieren gilt (siehe Abb. 17).

Der Ablauf ist genau wie zu Anfang des Abschnittes Abschnitt 3 beschrieben. Alle Agenten der aktuellen Generation entscheiden mithilfe von verschiedenen Beobachtungen ob links oder rechts zu lenken ist. Als Beobachtung wird bei dieser Umgebung von *gym* nach jedem Frame ein Array mit vier Elementen zurückgegeben. In diesem Array befinden sich die Aktuelle x-Position des Wagens (von -2.4 bis 2.4), die Geschwindigkeit des Wagens (von -∞ bis +∞), der Winkel des Stabes (von -41.8° bis 41.8°) und die Geschwindigkeit der Stabspitze (von -∞ bis +∞). Daher benötigen die Neuronalen Netze der Agenten eine Eingabeschicht mit vier Neuronen.

Damit das Netz das Problem gut lernen kann, werden hier 5 Neuronen in der versteckten Schicht eingesetzt. Werden an dieser Stelle zu wenige Neuronen gewählt, kann das Neuronale Netz das Problem nicht richtig erlernen, da es nicht genug Fälle unterscheiden kann. Werden hingegen zu viele Neuronen verwendet, benötigt der Prozess zu viel Rechenleistung und funktioniert nur sehr langsam.

Die Topologie des Netzes ist jedoch nicht die einzige Konstante, die es anzupassen gilt. Die Anzahl der Generationen und die Größe der Population müssen ebenfalls geregelt werden. Damit die Vorteile des Neuroevolutionsverfahrens zur Geltung kommen, sollte die Anzahl der Generationen hoch genug sein. Meistens kommt es aber vor, dass die Agenten schon eine gute Strategie herausgefunden haben, bevor die beschlossene Anzahl an Generationen durchlebt wurde. Es wird oft ein sogenannter Fitness-Threshold eingeführt. Dieser ist quasi eine Obergrenze, die zum Abbruch des Programms führt. Die Agenten haben, jetzt ausreichend gelernt, um das Problem angemessen zu meistern. Das Limit an Generationen beträgt bei meiner Implementation 200, die Populationsgröße jeweils 30 und der Fitness-Threshold wird auf 400 gesetzt.

Da bei der Auto-Stab Simulation nur eine Simulation gleichzeitig stattfinden kann, werden alle Agenten einer Generation nacheinander ausgeführt. Für jeden Zeitschritt, den

der Agent überlebt, wird seine Fitness erhöht. Kippt der Stab zu weit nach links oder rechts oder bewegt sich der Wagen außerhalb des dargestellten Bildschirms, wird die aktuelle Simulation abgeschlossen und es wird mit der nächsten fortgefahren. Danach finden die oben beschriebenen Vorgange Fitness-Evaluierung, Selection, Crossover und Mutation statt. Die neu erzeugte Generation wird dann wieder Agent für Agent ausprobiert. Nach jeder Generation wird in der Konsole ein kleiner Zwischenstand ausgegeben, der Informationen über den Verlauf des Lernen abgeben soll (siehe Abb. 18, eine größere Abbildung ist im Anhang zu finden).

```
if __name__ == "__main__":
  env = gym.make("CartPole-v1")
  generations = 200
  pop_size = 30
  threshold = 400
  network = [[4,5,sigmoid],[None,1,sigmoid]]
  # zufaellig Agenten initialisieren
  agents = generate_agents(pop_size, network)
  # Generationen entwickeln
  try:
    for i in range(generations):
            print('****** Generation ',str(i),'******\n')
            start = time.time()
            for agent in agents:
                observation = env.reset()
                agent.fitness = 0
                for _ in range(1000):
                    env.render()
                    action = agent.neural_network.propagate(observation)
                    if action[0] >= 0.5:
                        action = 1
                    else:
                        action = 0
                    observation, reward, done, info = env.step(action)
                    agent.fitness += reward
                    if done:
                        observation = env.reset()
                        break
            # Fakten ueber Generation ausgeben
            summe = 0
            for agent in agents:
                summe += agent.fitness
            summe = summe / len(agents)
            agents_sorted = sorted(agents, key=lambda agent: agent.fitness, reverse=True)
            print("Durchschnittliche Fitness der Population:", summe)
            print("Beste Fitness", agents_sorted[0].fitness)
            print("Population von", pop_size)
            print("Zeit fuer die Generation:", time.time() - start, "Sekunden\n")
            # ab bestimmter Fitness aufhoeren
            if summe >= threshold:
                print("Grenze erreicht ab Generation:", str(i))
                break
            # Reproduktion fuer die naechste Generation
            agents = selection(agents)
            agents = crossover(agents,network,pop_size)
  except:
    pass
  env.close()
```

Abbildung 18: Ausführung des Algorithmus (kleine Darstellung)

Es kommt bei dieser Neuroevolutionsmethode auch dazu, dass die am Anfang zufällig gewählten Gewichte keine gute Vorraussetzung sind, das Problem zu lösen. Die Agenten machen selbst durch die Evolution keinen Fortschritt und werden nicht besser. Das Crossover-Prinzip greift deswegen auch nicht mehr vernünftig. In diesem Fall ist durch die Zwischenstände in der Konsole zu sehen, dass die durchschnittliche Fitness sich nicht verbessert und das Programm neu gestartet werden muss. Dieser Fall tritt bei guter

Werteoptimierung nur selten ein, sollte aber bei einer möglichen Fehleranalyse der Implementation berücksichtigt werden.

4.2. Testen und Evaluierung des Algorithmus

Ein großer Teil der Entwicklung eines Algorithmus ist immer das Testen seiner Effektivität. Daher soll auch der implementierte Neuroevolutionsalgorithmus getestet werden. Dazu wird der Programmcode aus Abschnitt Abschnitt 4.1 genutzt. Es lässt sich erkennen, dass die durchschnittliche Fitness meistens von Generation zu Generation steigt. Der Algorithmus zeigt Wirkung (siehe Abb. 19). Der Lernprozess wurde auch in einem Video festgehalten (siehe `https://youtu.be/Xp02PP2fgyQ`). Um den Algorithmus noch besser testen zu können, könnten noch komplexere Probleme verwendet werden und hier der Verlauf der Fitness-Kurve analysiert werden.

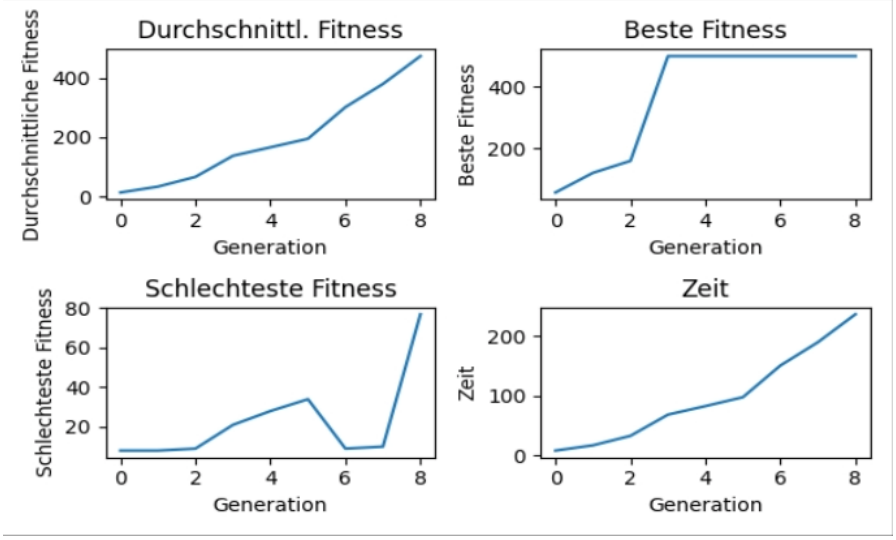

Abbildung 19: Evaluierung

4.3. Fazit und Ausblick

Aus dem Ergenis dieser Implementierung der Neuroevolution mit feststehenden Topologien kann gezogen werden, dass es viele verschiedene Lernmethoden gibt, die funktionieren und Wirkung zeigen. Früher wurde die Neuroevolution als unzuverlässig empfunden, über die letzen Jahre wurde sich jedoch mehr dieser Art von reinforcement learning gewidmet, da auch die Computer immer besser wurden und mehr Rechenleistung zur Verfügung stellen. Es kommt daher auch zu vielen Neuigkeiten und neuen Verfahren auf diesem Gebiet. 2002 wurde das Verfahren NEAT veröffentlicht. Bisher gab es viele Verfahren von Neuroevolution mit feststehenden Topologien, so wie auch in der obigen Implementation.

Dieses Neue Verfahren NEAT führt alternierende Topologien ein. Dabei tauchen wieder verschiedene Probleme auf, die es zu lösen gilt.

Aus dem Überblick über die Neuroevolutions-Verfahren lässt sich erkennen, dass es in diesem Bereich so gut wie kein Falsch und kein Richtig gibt. Der einzige zu optimierende Faktor ist in dieser Gebiet - sowie in der Informatik oft - die Effizienz.

Literaturverzeichnis

Bourg Anne Paulus Lena, Reiser Manon (2006). "Neuronale Netze". In: *Leopold - Franzens - Universität Innsbruck Institut für Psychologie*. URL: https://www.uibk.ac.at/psychologie/mitarbeiter/leidlmair/neuronale_netze.pdf.

Eiben, A. E. und James E. Smith (2016). *Introduction to Evolutionary Computing*. Springer Berlin. ISBN: 978-3-540-40184-1.

Exponential distribution (Feb. 2021). URL: https://en.wikipedia.org/wiki/Exponential_distribution.

Rundfunk, Bayerischer (Apr. 2019). *Biologie - 2. Nervensystem: Nervenzelle - Aufbau und Funktion*. URL: https://www.br.de/telekolleg/faecher/biologie/biologie-2-nervenzelle100.html.

scipy.stats.expon (o. D.). URL: https://het.as.utexas.edu/HET/Software/Scipy/generated/scipy.stats.expon.html.

Stanley, Kenneth O. (Juli 2017). *Neuroevolution: A different kind of deep learning*. URL: https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/.

Steinwendner, Joachim und Roland Schwaiger (2019). *Neuronale Netze programmieren mit Python*. 1. Auflage. Bonn: Rheinwerk Verlag GmbH. ISBN: 978-3-836-26144-9.

Svozil, Daniel, Vladimr Kvasnicka und Pospichal Jiri (Apr. 1998). *Introduction to multi-layer feed-forward neural networks*. URL: https://www.sciencedirect.com/science/article/abs/pii/S0169743997000610.

Zell, Andreas (2003). *Simulation neuronaler Netze*. 1. Auflage. Mnchen: Oldenbourg. ISBN: 3-486-24350-0.

Abbildungsverzeichnis

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die dem Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.

Ort, Datum: _____

Unterschrift:_____

A. Abbildungen

A.1. Beispielmatrix mit zufälligen Gewichten

In dieser Abbildung ist der beispielhafte Gewichtsvektor eines Neuronalen Netzes mit 4 Eingangsneuronen, 5 versteckten Neuronen und 1 Ausgangsneuron dargestellt.

$$\begin{bmatrix} -0.7255 & 0.4228 & -0.7799 & -1.2115 & -0.9617 \\ -0.3162 & 1.4284 & -0.0100 & 0.6101 & 0.5824 \\ -1.8437 & -0.3541 & -1.8160 & -0.449 & 0.1436 \\ 0.0789 & 0.7453 & 0.1674 & 0.2487 & -0.903 \end{bmatrix} \begin{bmatrix} -1.5728 \\ -2.3026 \\ -1.2218 \\ 0.4414 \\ 0.2672 \end{bmatrix}$$

Seine Struktur wird in einer Liste gespeichert als $[(4, 5), (5, 1)]$. Dann wird der Array auf eine Dimension heruntergebrochen:

$$[-0.7255, 0.4228, -0.7799, -1.2115, -0.9617, -0.3162, 1.4284, -0.0100, 0.6101, 0.5824, -1.8437, -0.3541, -1.8160, -0.449, 0.1436, 0.0789, 0.7453, 0.1674, 0.2487, -0.9037, -1.5728, -2.3026, -1.2218, 0.4414, 0.2672]$$

Jetzt kann die Kreuzung starten und das enstandene Kind wieder in die ursprüngliche Form zurückgeführt werden.

Abbildung 20: Beispielmatrix mit zufälligen Gewichten

A.2. Ausführung des Algorithmus (große Darstellung)

```
if __name__ == "__main__":
  env = gym.make("CartPole-v1")
  generations = 200
  pop_size = 30
  threshold = 400
  network = [[4,5,sigmoid],[None,1,sigmoid]]
  # zufaellig Agenten initialisieren
  agents = generate_agents(pop_size, network)
  # Generationen entwickeln
  try:
    for i in range(generations):
            print('****** Generation',str(i),'******\n')
            start = time.time()
            for agent in agents:
                observation = env.reset()
                agent.fitness = 0
                for _ in range(1000):
                    env.render()
                    action = agent.neural_network.propagate(observation)
                    if action[0] >= 0.5:
                        action = 1
                    else:
                        action = 0
                    observation, reward, done, info = env.step(action)
                    agent.fitness += reward
                    if done:
                        observation = env.reset()
                        break
            # Fakten ueber Generation ausgeben
            summe = 0
            for agent in agents:
                summe += agent.fitness
            summe = summe / len(agents)
            agents_sorted = sorted(agents, key=lambda agent: agent.fitness,
                reverse=True)
            print("Durchschnittliche Fitness der Population:", summe)
            print("Beste Fitness", agents_sorted[0].fitness)
            print("Population von", pop_size)
            print("Zeit fuer die Generation:", time.time() - start, "Sekunden
                \n")
            # ab bestimmter Fitness aufhoeren
            if summe >= threshold:
                print("Grenze erreicht ab Generation:", str(i))
                break
            # Reproduktion fuer die naechste Generation
            agents = selection(agents)
            agents = crossover(agents,network,pop_size)
  except:
    pass
  env.close()
```

Abbildung 21: Ausführung des Algorithmus (große Darstellung)