

# BACHELORARBEIT

## Neuroevolution in RoboCup2D

Alexander Isenko

Entwurf vom 29. November 2016





# BACHELORARBEIT

## Neuroevolution in RoboCup2D

Alexander Isenko

Aufgabensteller: Prof. Dr. Claudia Linnhoff-Popien

Betreuer: Thomas Gabor, M.Sc  
Dr. Lenz Belzner

Abgabetermin: 9. Dezember 2016





Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 9. Dezember 2016

.....  
*(Unterschrift des Kandidaten)*

## Abstract

Wir untersuchen in dieser Bachelorarbeit verschiedene Ansätze zur Entwicklung von **neuronalen Netzen** am Beispiel der **Cross Entropy Method**, **genetische Algorithmen** und **CoSyNE** unter Einschränkung von spärlichen Fitnesssignalen, hochdimensionalen kontinuierlichen Zustandsräumen und simulationsbasierter Optimierung.

Der Suchraum wird durch **diskrete Kosinustransformationen** unter der Annahme reduziert, dass benachbarte Gewichte in neuronalen Netzen zueinander korreliert sind. Die Domäne ist ein Fußballsimulator, **Half Field Offense**, die Teams aus dem weltweiten Wettbewerb RoboCup2D mitliefert an denen wir uns messen können.

Dafür entwickeln wir mehrere Angreifertaktiken im **1 gegen 1 Szenario** gegen den Torwart aus der Standardimplementierung. Die Umsetzung erfolgt in Haskell und Python.

We analyze different approaches for **neuroevolution**, by means of the **Cross Entropy Method**, **Genetic Algorithms** and **CoSyNE** with the restriction of sparse fitness signals, continuous state spaces and simulation-based optimization.

The state space will be reduced with the help of **discrete cosine transformations** under the assumption of correlated weights in neural nets. The domain is a soccer simulator, **Half Field Offense**, which includes teams from the worldwide competition RoboCup2D.

We develop different attacker policies for the **1 versus 1 scenario** with the goal keeper from the standard implementation. The languages used were Haskell and Python.

# Inhaltsverzeichnis

<b>1 Einführung</b>	<b>1</b>
1.1 Aufgabenstellung . . . . .	1
1.2 Motivation . . . . .	2
1.3 Aufbau der Arbeit . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 Genetische Algorithmen . . . . .	3
2.1.1 Individuen . . . . .	4
2.1.2 Evaluation . . . . .	5
2.1.3 Selektion . . . . .	5
2.1.4 Kreuzung . . . . .	5
2.1.5 Mutation . . . . .	7
2.1.6 Repopulation . . . . .	8
2.2 Neuroevolution . . . . .	9
2.2.1 Künstliche neuronale Netze . . . . .	9
2.2.2 Backpropagation . . . . .	11
2.2.3 Verbindung mit genetischen Algorithmen . . . . .	12
2.3 Diskrete Kosinus Transformation . . . . .	13
2.3.1 Kodierung des Suchraums . . . . .	13
2.4 Cooperative Synapsen Neuroevolution . . . . .	14
2.4.1 Permutation . . . . .	14
2.5 Cross Entropy Method . . . . .	17
2.5.1 Normalverteilung . . . . .	17
2.5.2 Kodierung durch Normalverteilungen . . . . .	19
<b>3 Umsetzung in RoboCup2D</b>	<b>21</b>
3.1 Half Field Offense . . . . .	22
3.1.1 Zustandsraum . . . . .	23
3.1.2 Aktionsraum . . . . .	25
3.1.3 Einschränkungen . . . . .	26
3.2 Implementierung der Algorithmen . . . . .	27
3.2.1 Wahrscheinlichkeitsverteilung von Aktionen . . . . .	28
3.2.2 Agentenstrategien als KNN mit DCT . . . . .	31
<b>4 Resultate</b>	<b>33</b>
4.1 1vs1 . . . . .	33
4.1.1 Wahrscheinlichkeitsverteilung der Aktionen . . . . .	34
4.2 Cross Entropy . . . . .	36
4.3 Neuroevolution . . . . .	38

*Inhaltsverzeichnis*

4.4 CoSyNE . . . . .	40
4.4.1 Vergleich . . . . .	42

<b>5 Diskussion</b>	<b>43</b>
5.1 Anwendungsmöglichkeiten . . . . .	43
5.2 Ausblick . . . . .	44
5.3 Schlusswort . . . . .	45
<b>Literaturverzeichnis</b>	<b>47</b>



# 1 Einführung

Die Relevanz von Machine Learning Algorithmen und **Deep Learning** [1] hat in den letzten Jahren seit der Weiterentwicklung von **GPUs** (Graphics Processing Unit) stark zugenommen. Das Training wird dabei durch die Optimierungsmethode **SGD** (Stochastic Gradient Descend) durchgeführt, die uns erlaubt durch das Ableiten einer multidimensionalen Funktion zu einer Lösung zu konvergieren. Damit wurden bemerkenswerte Maßstäbe in der Beschreibung von Bildern in **ImageNet** [2], dem Lernen einer Strategie für das Brettspiel **Go** [3] oder der Nachahmung der menschlichen Sprache durch **WaveNet** [4] gesetzt.

Leider sind dadurch andere Methoden zur Entwicklung von neuronalen Netzen aus dem Fokus gefallen, die zu der Familie von **unsupervised Learning** gehören. Sie können umfangreicher eingesetzt werden, weil sie weniger Einschränkungen für die Anwendungsdomäne haben. Sie benötigen keine vorher beschriftete Daten und unterstützen simulationbasiertes Training.

Insbesondere untersuchen wir den **CoSyNE** Algorithmus der verschiedene Techniken verknüpft um die Suche im Raum von neuronalen Netzen zu beschleunigen. Dabei zeigen wir einen bisher nicht gesehenen Vergleich mit dem **Neuroevolutionsalgorithmus** ohne den zusätzlichen Permutationsschritt. Hinzu kommt der **Kompressionsfaktor von 1:55** im Gewichtsraum für ein größeres rekurrenten Netz als im Ursprungspaper [5].

## 1.1 Aufgabenstellung

In dieser Arbeit beschäftigen wir uns mit der Entwicklung von neuronalen Netzen mithilfe von genetischen Algorithmen für die Fußballdomäne **Half Field Offense** [6]. Sie hat ein **spärliches Fitnesssignal**, ein hochdimensionalen kontinuierlichen Zustandsraum und keine Möglichkeit für jede Situation eine perfekte Aktion festzulegen. Damit bietet sie Parallelen zu echte-welt Problemen für die man entweder nicht genug Wissen sammeln konnte, oder wollte.

Wir untersuchen verschiedene Kodierungen und Implementierungen für den neuroevolutionäre Algorithmen und versuchen den Nutzen für andere Domänen mit ähnlichen Einschränkungen zu erahnen.

## 1.2 Motivation

Die Industrie interessiert sich für allgemeine Problemlösungen, die in kurzer Zeit, mit wenig Daten und am besten automatisch zu einem akzeptablen Ergebniss kommt. Leider steht das den üblichen **Deep Learning** Techniken gegenüber, die lange Trainingszeiten haben, viele nicht homogene Daten in normalisierter Form brauchen und von Hand angepasste Fitness Funktionen benötigen die für das Ziel optimiert wurden.

Deshalb betrachten verschiedene Möglichkeiten mithilfe eines GAs neuronalen Netzen zu entwickeln die als Fitnesssignal lediglich das Ziel bekommen und sich in einem hochdimensionalen, stetig verändernden, kontinuierlichen Zustandsraum mit mehreren Akteuren bewegen.

## 1.3 Aufbau der Arbeit

Im Rahmen dieser Arbeit werden im Kapitel 2 die Grundlagen von Genetischen Algorithmen und deren Verknüpfung zu neuronalen Netzen und der Cross Entropy Method erklärt und anschaulich dargestellt. Kapitel 3 beschäftigt sich mit der Domäne, Parametrisierung der Algorithmen und der jeweiligen Resultate. Das Kapitel 4 gibt einen Ausblick in weitere Verbesserungsmöglichkeiten, stellt Vergleiche zu bisherigen Resultaten von CoSyNE dar und behandelt verwandte Felder.

## 2 Grundlagen

Dieses Kapitel bietet Einblick in die Grundlagen von **Genetischen Algorithmen** (Kap. 2.1) im Zusammenhang mit **neuronalen Netzen** (Kap. 2.2.1) und der **Cross Entropy Method** (Kap. 2.5). Außerdem werden einige Verbesserungen zu den naiven Methoden besprochen, wie die Reduzierung des Suchraums durch **Fouriertransformationen** und die Einführung von einer **kooperativen Evolution** durch Hinzufügen von einer neuen Aktion zu dem Ablauf des genetischen Algorithmus.

### 2.1 Genetische Algorithmen

Ein genetischer Algorithmus, im folgenden als **GA** abgekürzt, ist ein Optimierungsverfahren, das von der natürlichen Selektion und Evolution inspiriert ist. Ein formaler Leitfaden findet sich im Fundamentalwerk zu Genetischen Algorithmen [7].

Stellen wir uns anschaulicher Weise eine Gruppe Gazellen und einen Geparden vor.

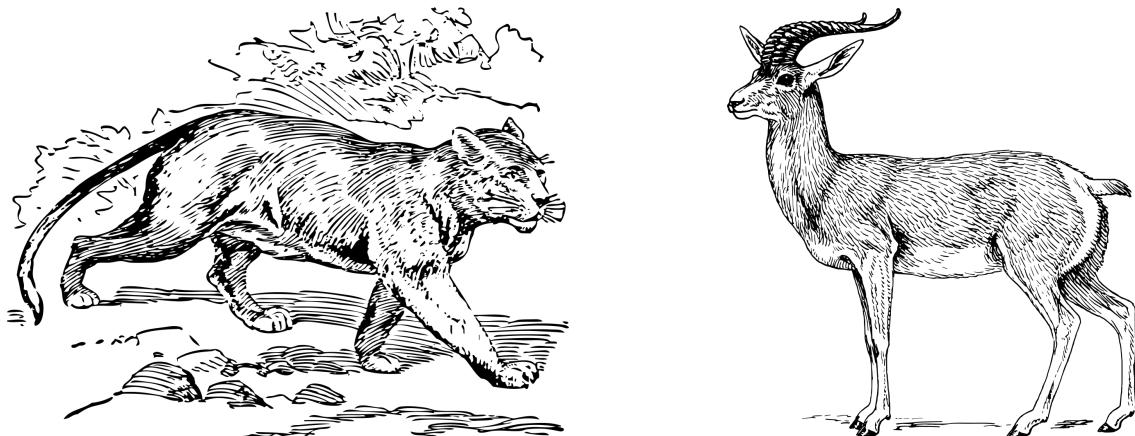


Abbildung 2.1: Illustration eines Geparden und einer Gazelle

Sei unser Gepard durch seine Geschwindigkeit den Gazellen überlegen, dann wird die Gazellenherde über Zeit in ihrer Anzahl sinken. Dabei werden die langsamen Gazellen dem Geparden erliegen und die Schnelleren überleben. Dieser Schritt wird als **Selektion** bezeichnet. Die Überlebenden werden sich fortpflanzen und mit hoher Wahrscheinlichkeit Gazellen-Babies bekommen die ähnlich schnell sind. Diesen Vorgang bezeichnen wir als **Kreuzung**. Mit welcher Wahrscheinlichkeit jedes einzelne Tier vor dem Geparden entwischen kann nennen wir **Fitness**.

Jede Gazelle, oder auch **Individuum** genannt, hat eine eigene Fitness, die es aber bei Geburt noch nicht weiß, da sie noch nie vor einem Geparden weglauen musste. Erst

## 2 Grundlagen

nachdem sie einmal erfolgreich entwischt ist, können wir uns vorstellen, was ihre Fitness ist.

Ganz selten wird ein Gazellen-Baby geboren, das ein etwas längere Beine hat als alle anderen, dabei hatte keiner dieses Merkmal vor ihr. Das erlaubt ihr schneller zu laufen, was für sie erstmal positiv ist. Diese Ausprägung hat jedoch den Nachteil, dass die Standhaftigkeit darunter leidet. Diese unerwartete Veränderung bei den Kindern heißt **Mutation**.

Fassen wir zusammen: Nachdem jede überlebte Gazelle sich fortgepflanzt hat, bekommen wir hoffentlich wieder eine vollzähliges Herde, die wir **Population** nennen. Nach all diesen Schritten fängt der Kampf um das Überleben wieder an und geht solange, bis sich entweder Gazellen entwickeln, die dem Gepard ständig entkommen können, oder bis die gesamte Population ausstirbt.

Damit haben wir die wichtigsten Begrifflichkeiten von einem genetischen Algorithmus erklärt und kommen zur Umsetzung der einzelnen Schritte.

### 2.1.1 Individuen

Ein Individuum besteht aus einer Kodierung, auch **Zustandsraum** genannt, die die aussagekräftigen Eigenschaften von ihm ausmachen. Für eine Gazelle wäre beispielweise die folgende Kodierung möglich.

Höchstgeschwindigkeit	95 $\frac{km}{h}$
Beinlänge	86 cm
Gewicht	43 kg
Hornlänge	12 cm

Tabelle 2.1: Kodierung einer Gazelle

Die Aufgabe von unserem GA ist ein oder mehrere Individuen zu finden, die es schaffen vor dem Geparden wegzu laufen. Da wir aber nicht wissen, ob die vorgeschlagene Kodierung gut oder schlecht ist, müssen wir Gazellen mit zufälligen Eigenschaften erstellen und dann den Algorithmus arbeiten lassen.

Das schaffen wir, indem wir Grenzen für die Kodierung festlegen und später zufällige Werte in diesen Rahmen ausprobieren.

Eigenschaft	Minimaler Wert	Maximaler Wert
Höchstgeschwindigkeit	20 $\frac{km}{h}$	100 $\frac{km}{h}$
Beinlänge	40 cm	90 cm
Gewicht	12 kg	75 kg
Hornlänge	0 cm	35 cm

Tabelle 2.2: Grenzen für die Kodierung [8] [9]

## 2.1.2 Evaluation

Nachdem wir unsere Gazellenpopulation erstellt haben, müssen wir sie der Natur überlassen. Dann ist es unsere Aufgabe nach einer festen Zeitspanne und sie alle wieder aufzusammeln. Dadurch finden wir heraus wie viele Gazellen überlebt haben und können diese Information den nächsten genetischen Methoden übergeben.

## 2.1.3 Selektion

Nachdem die Evaluation vorbei ist, bekommen wir die Rückmeldung welche Gazellen überlebt haben. Aus dieser Menge können wir nun einen prozentualen Betrag wählen, die Eltern sein werden. In unserer Implementierung nennen wir diesen Parameter  $\alpha$ . Damit versichern wir, dass nur die erfolgreichen Eigenschaften weiter in der Population erhalten bleiben und der Rest wegfällt.

## 2.1.4 Kreuzung

Die erfolgreichen Individuen wurden ausgewählt und können sich nun fortpflanzen. Dafür nehmen wir jeweils zwei Individuen und vertauschen zufällig ihre Ausprägungen.

Eigenschaften
56 $\frac{km}{h}$
42 cm
51 kg
10 cm

Tabelle 2.3: Kodierung des Vaters

Eigenschaften
62 $\frac{km}{h}$
55 cm
49 kg
8 cm

Tabelle 2.4: Kodierung der Mutter

Eigenschaften
56 $\frac{km}{h}$
55 cm
49 kg
10 cm

Tabelle 2.5: Kodierung vom Kind Nr.1

Eigenschaften
62 $\frac{km}{h}$
42 cm
51 kg
8 cm

Tabelle 2.6: Kodierung vom Kind Nr.2

Das können wir nun sooft machen wie wir Eltern finden, oder bis wir genug Kinder produziert haben.

## 2 Grundlagen

In unserem Beispiel haben wir die Kinder mit dem folgenden Python-Code konstruiert:

```
1 vater = [56,42,51,10]
2 mutter = [62,55,49,8]
3 kind1 = []
4 kind2 = []
5 for i in range(kodierung.length):
6     r = random.uniform(0,1)
7     if (r > 0.5):
8         kind1[i] = vater[i]
9         kind2[i] = mutter[i]
10    else:
11        kind1[i] = mutter[i]
12        kind2[i] = vater[i]
```

In Z.1-2 definieren wir die Eigenschaften der Mutter und des Vaters. Dann iterieren wir durch die Länge der Kodierung (Z.5) und wählen mit einer 50% Wahrscheinlichkeit (Z.6-7) aus für jedes Kind aus, ob die gewählte Eigenschaft vom Vater oder von der Mutter kommt.

Diese Art und Weise zwei Individuen zu kreuzen nennt sich **n-point crossover**, weil wir die Kodierung an zufällig vielen Stellen unterbrechen. Es gibt noch andere Kreuzungsmethoden die eine eine feste Anzahl von Aufteilungen benutzen, wie **one-** oder **two-point crossover**.

Um einen Unterschied zwischen diesen Methoden zu erkennen, stellen wir uns vor dass die Beinlänge im Zusammenhang mit der Höchstgeschwindigkeit steht, weil längere Beine eine größere Sprungweite ermöglichen. Wenn nun ein Kind gezeugt wird, dass lange Beine vererbt, wird die Höchstgeschwindigkeit dadurch nicht automatisch angepasst. Deshalb wäre es besser, wenn diese Ausprägungen zusammen übernommen werden, weil dadurch eine höhere Fitness garantiert werden kann. Kreuzungsmethoden wie n-point-crossover verletzen diese Eigenschaft eher als wie one-point-crossover.

Je nach Implementierung verwendet man nur eins der beiden Kinder, weil das die Varianz der Gesamtpopulation weniger beeinflusst und trotzdem keinerlei Information verloren geht, weil die Eltern die Kodierung weiter tragen.

### 2.1.5 Mutation

Nachdem die Kinder erstellt wurden, müssen wir die Kodierung der Individuen etwas verändern, damit die Varianz in der Gesamtpopulation erhöht wird. Das machen wir indem wir durch die Kodierung der Kinder durchgehen und jede Ausprägung mit einer geringen Wahrscheinlichkeit verändern. Diese nennen wir  $\beta$ .

Eigenschaften
62 $\frac{km}{h}$
42 cm
51 kg
8 cm

Tabelle 2.7: Kodierung von einem Kind

Eigenschaften
62 $\frac{km}{h}$
42 cm
45 kg
8 cm

Tabelle 2.8: Mutierte Kodierung vom Kind

Die Mutation kann folgendermaßen in Python umgesetzt werden:

```

1 kinder = [k1, k2...]
2 beta   = 0.1
3 for i in range(kinder.length):
4     for j in range(kodierung.length):
5         r = random.uniform(0,1)
6         if (r > beta):
7             kinder[i][j] = sampleNewFrom(kodierung[j].range)

```

In Z.1-2 definieren wir Mutationswahrscheinlichkeit und die Kinder. Dann gehen wir jede Kodierung von jedem Kind durch (Z.3-4) und verändern die Eigenschaft mit einer 10% Wahrscheinlichkeit (Z.5-7).

Dieser Schritt ist wichtig, sodass trotz konvergierter Population neue Eigenschaften ausprobiert werden, da sie vielleicht eine bessere Lösung bieten. Der GA tendiert oft dazu sich erstmal für eine suboptimalen Lösung zu entscheiden und die Mutation erlaubt uns einen Ausweg daraus.

In manchen Fällen kann man die Mutation noch weiter parametrisieren, indem man ein Veränderungsfaktor als Argument hinzufügt. Diese Technik benutzt man, wenn die Kodierung nicht trivialerweise verändert werden kann, da sonst bestimmte Eigenschaften verloren gehen. In Kapitel 3 wird genau so ein Fall besprochen, weil wir unsere Individuen durch eine Wahrscheinlichkeitsverteilung darstellen.

### 2.1.6 Repopulation

Die Eltern wurden ausgewählt, die Kindern gezeugt und mutiert, nun müssen wir die Population in eine Form bringen, sodass die Evaluation neu gestartet werden kann. Wir stellen das Problem wieder an einem Beispiel dar.

```

1 population = [i1,i2,...]                      # population.length = 10
2 alpha      = 0.4
3 eltern     = selection(population, alpha)    # eltern.length = 4
4 kinder     = crossover(eltern)                # kinder.length = 4
5 beta       = 0.1
6 mutkinder  = mutation(kinder, beta)          # mutkinder.length = 4
7
8 newpopulation = eltern + mutkinder           # newpopulation.length = 8

```

Wir sehen dass uns zwei Individuen weniger als zu Anfang haben können deshalb die Evaluation nicht neu starten. Dieses Problem kann man auf viele Weisen angehen, die ihre eigenen Vorteile und Nachteile haben.

#### Mehr Kinder erstellen

Es ist möglich während der Kreuzung solange Kinder zu erzeugen, bis die Population wieder ihre Ausgangsgröße angenommen hat. Ein Vorteil wäre, dass diese Individuen mit wahrscheinlich besseren Ausgangskodierungen starten als Neue. Der Nachteil ist jedoch die gesenkten Varianz in der Population und die erhöhte Wahrscheinlichkeit zum Feststrecken in einem suboptimalen Lösungen.

#### Nicht selektierte Individuen nachfüllen

Man kann die nicht benutzen Individuen aus der vorherigen Population zum Auffüllen benutzen, was sich aber nur dann gewählt werden sollte, wenn die Chance bestünde, dass sie in der erneuten Simulation besser abschneiden als bisher. Ansonsten nehmen sie den Platz für ein potenziell besseres Individuum weg.

#### Neue Individuen erstellen

In unserer Implementierung haben wir uns für das Nachfüllen von völlig neuen Individuen entschieden, da dadurch die Varianz der Population angehoben wird und dadurch mehr Lösungen möglich sind. Ein Nachteil ist dabei sind die potenziellen Kinder die keinen Platz bekommen, aber da dadurch keine Information verloren geht, können wir es vernachlässigen.

## 2.2 Neuroevolution

Der Begriff der Neuroevolution wurde im Jahre 1988 von D. Whiteley [10] als alternative Möglichkeit zum Trainieren von künstlichen neuronalen Netzen (**KNNs**) vorgeschlagen. Dabei wird versucht aus der Synergie von dem **selbstlernenden Charakter** von KNNs und der **explorativen Suche** eines GAs eine Taktik oder ein Klassifikator zu entwickeln der völlig neue Lösungen finden kann. Den Beweis dafür hat man bereits im Jahr 1995 am Spiel *Othello* festgestellt. [11]

Wir versuchen in diesem Kapitel einen groben Überblick über die Funktionsweise von KNNs zu verschaffen und stellen den Bezug zu genetischen Algorithmen dar. Eine weitaus formalere Erklärung findet sich im Paper von ....

### 2.2.1 Künstliche neuronale Netze

Die Idee hinter künstlichen neuronalen Netzen ist der Versuch die Struktur vom menschlichen Gehirn nachzuahmen. Ein übliches KNN besteht jedoch aus vielfach weniger Neuronen, meist hundert bis mehrere tausend, wobei unser Gehirn 86 Milliarden[12] besitzt. Ein künstliches Neuron kann man sich anschaulich als eine Formel vorstellen, die **eine oder mehrere Eingaben** über **gewichtete Pfade** bekommt, sie **aufsummiert** und eine Aktivierungsfunktion auf das Ergebnis anwendet, die auf den Bereich  $[0, \infty]$ ,  $[0, 1]$ , oder  $[-1, 1]$  abbildet. Dieses Resultat nennen wir  $\hat{y}$ :

Sei  $n$  die Anzahl der Eingaben,  
 $X = \{x_0, x_1, \dots, x_n\}$  die Eingabe,  
 $W = \{w_0, w_1, \dots, w_n\}$  die jeweiligen Gewichte,  
 $\sigma(x) = \max(x, 0)$  als Aktivierungsfunktion:

$$\hat{y} = \sigma(\sum_{i=0}^n x_i \cdot w_i)$$

Abbildung 2.2: Formel zur Berechnung des Ergebnisses eines Neurons

Wenn man nun mehrere von diesen Neuronen in Reihe zusammenschaltet (Abbildung 2.3), kriegt man ein vollständig vermaschtes Netz, welches grundsätzlich in drei Schichten unterteilt werden kann.

- **Eingabeschicht**

Hier kommt der Ausgangszustand rein, sei es ein kodierter Zustand eines Spiels, RGB Werte von einem Bild, oder der DAX.

- **Versteckte Schicht(en)**

Dieser Teil des Netzes besteht oft aus mehreren Schichten, da er für die Abstraktion und die Lernfähigkeit verantwortlich ist [13]. Er bekommt die Signale aus der Eingabeschicht, die er verarbeitet und weiterleitet.

- **Ausgabeschicht**

Die Ausgabeschicht ist zum Sammeln der Signale von der vorherigen Schicht zuständig und auf ihren Ergebnissen wird dann eine **Aktivierungsfunktion** angewendet, die die kumulierten Resultate in eine passende Form bringt. Sie varrieren

## 2 Grundlagen

zwischen einfachen Ja/Nein Aussagen, oder wie wir später kennen lernen werden, auch Wahrscheinlichkeitsverteilungen.

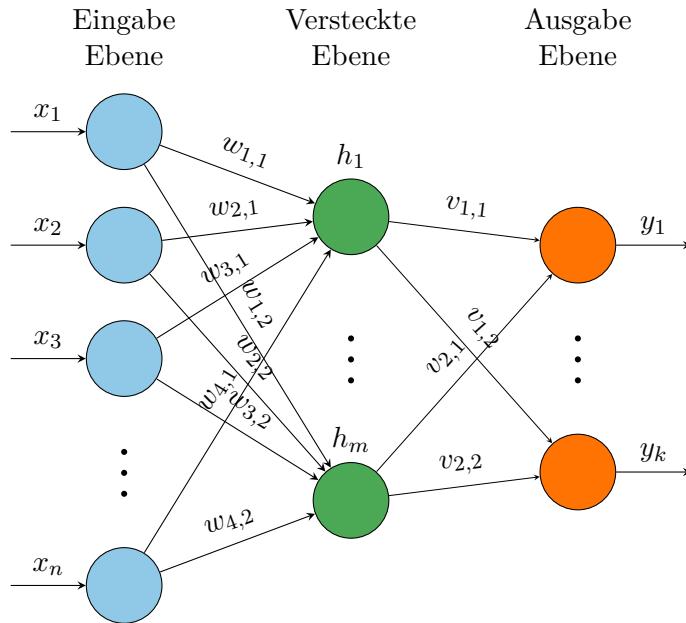


Abbildung 2.3: Skizze von einem vollständig vermaschten künstlichen neuronalem Netz

Neuronen wie in Abbildung 2.3 zusammen zu verknüpfen nennt sich ein **Feedforward** Netzwerk, da es keine Zyklen beinhaltet. Sie besitzen die Einschränkung das sie ohne Rücksicht auf die resultierenden Effekte in der Domäne ein Ergebnis liefern, da sie das Signal nur nach vorne weiterleiten.

Um eigene Resultate und zeitliche Abstraktionen einfacher zu berücksichtigen gibt es **rekurrente Netze** die direkte Zyklen beinhalten.

### LSTM Ebene

Ein spezielles Neuron aus dem rekurrenten Netzen bestehen können, ist das **Long Short Term Memory** (LSTM) Neuron[14]. Es zeichnet sich durch die Eigenschaft aus, dass es über lange Zeitfenster Information behalten kann. Der Aufbau basiert auf dem Modell einer Speicherzelle, sodass wir durch verschiedene Eingänge (**Gates**), die Schreib-, Lese- und Reset-Aktionen nachbauen können [15]. Einer der wichtigsten Aspekte von diesen Neuronen ist jedoch, dass sie ableitbar sind, weil dadurch die Trainingsmethode **Backpropagation** aus Kapitel 2.2.2 ermöglicht wird [16].

### Softmax Ebene

Es gibt eine Aktivierungsfunktion die wir besondere Aufmerksamkeit widmen, da sie die Neuronen der Ausgabeschicht zu einem nützlichen Ergebnis zusammenfassen kann. Die generalisierte logistische Funktion, oder auch **normalisierte Exponentialfunktion** nimmt als Argument einen  $k$ -dimensionalen Vektor  $\mathbf{z}$  von reellen Zahlen und gibt uns

widerrum den gleichen Vektor zurück, wo alle Werte auf den Bereich [0,1] normalisiert wurden.

Sei  $j = 1, 2, \dots, K$ :

$$\sigma(\mathbf{z})_j = \frac{e^{\mathbf{z}_j}}{\sum_{k=1}^K e^{\mathbf{z}_k}}$$

Abbildung 2.4: Definition der Softmax Funktion

Sei  $t \in \mathbb{R}$ :

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

Abbildung 2.5: Definition der Sigmoid Funktion

Wir könnten argumentieren, dass sich die einfache Normalisierung ohne die Exponentialfunktion dafür genauso dafür eignet. Wenn wir aber als Aktivierungsfunktion die **Sigmoid Funktion** (2.5) und als Kostenfunktion den **logistic-loss** oder **cross entropy loss** benutzt, kürzt sich das  $e$  beim partiellen Ableiten weg.

## 2.2.2 Backpropagation

Um diese Technik zum Trainieren von KNNs zu erklären müssen wir zunächst zeigen wie man den Fehler von einem neuronalen Netz misst. Dafür brauchen wir eine **Kostenfunktion** die uns die Abweichung zum Soll-Ergebnis gibt. Die Ergebnisse des Netzes für  $\hat{y}$  ist in Abbildung 2.2 definiert.

Sei  $m$  die Größe des Trainingssets,

$Y = \{y_0, y_1, \dots, y_m\}$  ein Vektor von Soll-Ergebnissen,

$\hat{Y} = \{\hat{y}_0, \hat{y}_1, \dots, \hat{y}_m\}$  ein Vektor von Resultaten des KNNs, dann ist:

$$\text{cost}(Y, \hat{Y}) = \frac{1}{m} \cdot \sum_{j=0}^m (y_j - \hat{y}_j)^2$$

die mittlere quadratische Abweichung.

Abbildung 2.6: Formel zur Berechnung des *MSE* Fehlers von einem KNN

Eine der möglichen Kostenfunktionen sieht man in Abbildung 2.6, die **mittlere quadratische Abweichung** (MSE). Je kleiner diese Kostenfunktion ist, umso besser kann unser KNN die Ergebnisse nachahmen. Um die Kosten zu minimieren, müssen wir die gesamte Funktion samt der Berechnung vom  $\hat{y}$  partiell nach den Gewichten ableiten.

Mit dem resultierenden Gradienten verändern wir die Gewichte, sodass der Fehler möglichst verringert wird. Dafür leiten wir die Anpassungsformel für jedes Gewicht her:

## 2 Grundlagen

Pro Beobachtung  $j$  in  $Y$ ,  
pro Gewicht  $i$  in  $W$  leiten wir partiell nach  $w_i$  ab:

$$\begin{aligned}\frac{\partial \text{cost}(y_j, \hat{y}_j)}{\partial w_i} &= \frac{\partial}{\partial w_i} (y_j - \hat{y}_j)^2 \\ &= \frac{\partial}{\partial w_i} (y_j - \max(\sum_{i=0}^n x_i \cdot w_i, 0))^2 \\ &= -2 \cdot \frac{\partial}{\partial w_i} \max(\sum_{i=0}^n x_i \cdot w_i, 0) \\ &= \begin{cases} 0 & \sum_{i=0}^n x_i \cdot w_i < 0 \\ -2 \cdot x_i & \sum_{i=0}^n x_i \cdot w_i > 0 \end{cases}\end{aligned}$$

Damit bekommen wir für Anpassungsregel für jedes Gewicht:

$$w_i \leftarrow w_i + \begin{cases} 0 & \sum_{i=0}^n x_i \cdot w_i < 0 \\ -2 \cdot x_i & \sum_{i=0}^n x_i \cdot w_i > 0 \end{cases}$$

Abbildung 2.7: Berechnung der Anpassungsregel für jedes Gewicht nach MSE

Diese Technik kann man benutzen, solange das gesamte Netz als Formel dargestellt ableitbar ist, selbst wenn es mehrere versteckte Schichten hat. Leider braucht sie dafür ein Trainingsset von Daten, was in oft aufwendig zu generieren ist. Deshalb benutzen wir eine simulationsbasierte Lernmethode.

### 2.2.3 Verbindung mit genetischen Algorithmen

Wenn wir nun zum Trainieren von ANNs genetische Algorithmen benutzen wollen, müssen wir das Netz als Liste von Gewichten kodieren, aus denen es besteht. Ein Beispiel dafür bietet der **GENITOR**[17] Algorithmus. Dabei werden die einzelnen Gewichte der Kreuzung und einer speziellen Mutation ausgesetzt die von der Varianz der Gesamtpopulation abhängt.

Ein weiterer Ansatz ist **SANE**[17], der einzelne Neuronen für die *Hidden Ebene* entwickelt und daraus ein Netz generiert.

Leider benutzen neuronale Netze heutzutage je nach Anwendungsgebiet immer aufwendigere Strukturen die extrem viele Gewichte besitzen. Eine naive genetische Suche in so einem hochdimensionalen Zustandsraum dauert zu lange und deshalb untersuchen wir Techniken die uns erlauben zielsicherer und effizienter den Raum aller Möglichkeiten zu durchsuchen.

## 2.3 Diskrete Kosinus Transformation

Eine Technik zum reduzieren des Suchraums bietet die diskreten Kosinustransformation (**DCT**), die zur Familie der Fouriertransformationen gehört. Eine **Fouriertransformation** teilt ein Signal in beliebig viele trigonometrische Funktionen, wie Sinus oder Kosinus auf, und über die Summe dieser Funktionen kann jedes Signal beschrieben werden.

Diese Fouriertransformation liefert uns ein diskretes Frequenzspektrum, das in Form von Koeffizienten dargestellt wird. Dabei wird pro Datenpunkt ein Koeffizient erzeugt. Um die Daten wiederherzustellen gibt es die inverse Kosinustransformation die die Koeffizienten wieder umwandelt.



Abbildung 2.8: Unkomprimiert



Abbildung 2.9: Komprimiert (1:5)

Wenn wir viele der Koeffizienten nicht benutzen und trotzdem versuchen die Datenpunkte wiederherzustellen, kriegen wir lediglich eine Annäherung, wie man in Abbildung 2.9 sieht. Sie ist meistens aber so gut genug, sodass wir keinen Unterschied merken. Eine sehr ähnliche Kompressionsmethode wird bei dem Bildformat JPEG oder dem Videoformat MPEG benutzt.

### 2.3.1 Kodierung des Suchraums

Diese Technik wenden wir nun auf die Gewichte von unserem neuronalen Netz an. Dafür beschränken wir den Suchraum auf eine kleine Anzahl der Koeffizienten und benutzen die inverse Kosinustransformation, um aus ihnen die nötige Anzahl von Gewichten zu erstellen. In Abbildung 2.10 und 2.11 sieht man eine Anwendung auf 100 Gewichte, die ein Kompressionsverhältnis von 1:2 haben. Das bedeutet, dass wir den Suchraum mit dem in 2.11 sichtbaren Genauigkeitsverlust halbiert haben.

Bei größeren Verhältnissen bemerken wir eine starke örtliche Korrelation (3.5) zwischen den benachbarten Zahlen und diese Eigenschaft passt zu der Annahme dass sich Gewichte in neuronalen Netzen ähnlich verhalten. Eine ausführlichere Erklärung findet sich im Ursprungspaper für die Anwendung in der Neuroevolution.[5]

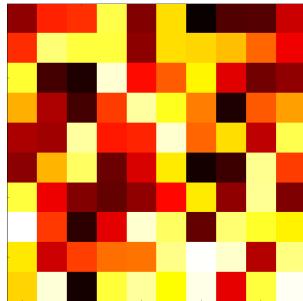


Abbildung 2.10: Unkomprimiert

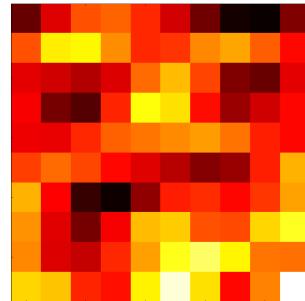


Abbildung 2.11: Komprimiert (1:2)

## 2.4 Cooperative Synapsen Neuroevolution

Nachdem wir den Zustandsraum komprimiert haben, sodass ein genetischer Algorithmus ihn in absehbarer Zeit entwickeln und ein neuronales Netz befüllt werden kann, erschließt sich die Verküpfung zu einem mächtigen Werkzeug das viele interessante Eigenschaften besitzt. Dieser Algorithmus wird **Cooperative Synapsen Neuroevolution**[18], oder auch **CoSyNE** genannt.

Er zeichnet sich speziell dadurch aus, dass er auf kontinuierlichen Zuständen und Aktionen funktioniert und spärliche Fitnesssignale interpretieren kann. Das schafft er indem er rekurrente Netze aufbaut und die genetische Suche mit aggressiver Mutation im Zustandsraum beschleunigt. Ein gutes Beispiel dafür ist das Rennspiel **TORCS**[19], wo der Algorithmus 993 Gewichte in 33 Koeffizienten kodiert (*Faktor 1:30*) und lediglich durch die Bilddaten ähnlich gute Ergebnisse liefert wie die per Hand programmierten Agenten, die die Physik des Spieles kennen.

Ein großer Nachteil von genetischen Algorithmen ist, das sie oft bei lokalen Maxima feststecken bleiben und lange brauchen um aus diesem Tal rauskommen. Um dieses Problem anzugehen, versucht man die Stellschrauben wie Mutationswahrscheinlichkeit oder Kinderanzahl per Hand zu verändern [20]. CoSyNE benutzt dafür eine eigene **genetische Methode**, um die Suche einfacher zu gestalten. Sie nennt sich Permutation und erzeugt innerhalb der gesamten Population Unterteilungen in kleinere Populationen die in einer **kooperativen und koevolutionären** Beziehung stehen.

### 2.4.1 Permutation

Der Permutationsschritt wird ganz am Ende von dem genetischen Algorithmus statt der Repopulation aufgerufen und vermischt jeden **Eigenschaftsraum** der Gesamtpopulation.

Individuum	Höchstgeschwindigkeit	Beinlänge	Gewicht	Hornlänge
1	60 $\frac{km}{h}$	40 cm	50 kg	10 cm
2	61 $\frac{km}{h}$	41 cm	51 kg	11 cm
3	62 $\frac{km}{h}$	42 cm	52 kg	12 cm
4	63 $\frac{km}{h}$	43 cm	53 kg	13 cm
5	64 $\frac{km}{h}$	44 cm	54 kg	14 cm
6	65 $\frac{km}{h}$	45 cm	55 kg	15 cm

Tabelle 2.9: Vor der Permutation

Individuum	Höchstgeschwindigkeit	Beinlänge	Gewicht	Hornlänge
1	61 $\frac{km}{h}$	43 cm	54 kg	12 cm
2	60 $\frac{km}{h}$	44 cm	51 kg	14 cm
3	64 $\frac{km}{h}$	45 cm	53 kg	10 cm
4	63 $\frac{km}{h}$	40 cm	52 kg	13 cm
5	62 $\frac{km}{h}$	41 cm	54 kg	11 cm
6	65 $\frac{km}{h}$	42 cm	50 kg	15 cm

Tabelle 2.10: Nach der Permutation

Wenn wir uns die Population als zweidimensionale Liste vorstellen, wo jedes Individuum eine eigene Liste mit seinen spezifischen Eigenschaften ist, können wir die Population transponieren, wobei nun jede Eigenschaft eine eigene Liste ist, diese zufällig vermischen und wieder zurück transponieren um die neuen Individuen zu bekommen. Der folgende Pythoncode veranschaulicht das Prinzip unter Verwendung der *numpy* Bibliothek.

```

1 import numpy as np
2
3 i_1 = [1,10,100,1000] # Individuum 1-5
4 i_2 = [2,20,200,2000]
5 i_3 = [3,30,300,3000]
6 i_4 = [4,40,400,4000]
7 i_5 = [5,50,500,5000]
8
9 population = np.array([i_1, i_2, i_3, i_4, i_5])
10 eigenschaftsraum = np.transpose(population)
11
12 for eig in eigenschaftsraum:
13     np.random.shuffle(eig)
14
15 population = np.transpose(eigenschaftsraum)
16
17 print population
18 > [[ 2   40   500 5000]
```

## 2 Grundlagen

```
19      [ 1 50 200 1000]
20      [ 4 10 300 3000]
21      [ 3 30 100 2000]
22      [ 5 20 400 4000]]
```

Man erkennt leicht, dass keine der ursprünglichen Individuen erhalten bleibt und wir vollkommen neue bekommen. Der Sinn hinter dem Vermischen in der Eigenschaftsebene basiert auf der **Verknüpfung mit der Kreuzungsmethode**. Wenn wir zwei Individuen kreuzen, werden ihre Kinder die gesamte Information von ihren Eltern in der Population übernehmen. Da CoSyNE den Repopulationsschritt nicht ausführt, dafür aber alle restlichen Individuen wegwirkt, werden nur die Eigenschaften der Kinder übrig bleiben.

Das führt zur Homogenität in den einzelnen Eigenschaften, die zum Beispiel dafür verantwortlich ist, dass alle Individuen gleich große Hörner haben. Wenn nun innerhalb der Hornlänge zufällig gemischt wird, bleibt alles gleich, da die gesamte Liste aus dem gleichen Element besteht.

Die Annahme von CoSyNE ist dass die Lösung für das Problem in der Kombination von den Eigenschaften von allen Individuen liegt, die wir am Anfang erstellen. Durch das aggressive Aussortieren durchsuchen wir den Raum aller Möglichkeiten schneller und darin liegt der größte Vorteil von diesem Algorithmus. Dieser Mechanismus wird als **kooperative Koevolution** im Eigenschaftsraum interpretiert, da wir jede Eigenschaft als Population betrachten und darin eine genetische Suche abhängig voneinander vollziehen [18].

Wenn diese Annahme nicht stimmt und die Lösung nicht in einem Subset von allen Eigenschaften liegt, hat CoSyNE leider nur die Möglichkeit durch Mutation zum Ziel zu kommen. Deshalb wird dieser Parameter oft hoch gewählt [5][18][19].

## 2.5 Cross Entropy Method

Eine andere Möglichkeit die Individuen in dem GA zu kodieren stellt die **Cross Entropy Method** dar. Das ist ein Algorithmus der oft als Vergleichskriterium verwendet wird, da er oft zu suboptimalen Strategien konvergiert [21]. Die Anwendung in unserem neuroevolutionären Schema basiert darauf, dass wir Individuen nicht als Ansammlung von Zahlen zu speichern, sondern jede Eigenschaft als Normalverteilung dargestellt wird. Diese Abläufe illustrieren wir anhand der Erklärung für eine Normalverteilung.

### 2.5.1 Normalverteilung

Eine Normalverteilung ist eine sehr bekannte stetige Wahrscheinlichkeitsverteilung. Ihre Wahrscheinlichkeitsdichtefunktion wird oft Gauß-Kurve oder Glockenkurve genannt und findet Anwendung in verschiedensten Anwendungsgebieten, weil sie natürliche Vorgänge exakt oder ähnlich genug modelliert. Bekannte Beispiele dafür sind die Streuung von Messfehlern oder die irreguläre Bewegung von Partikeln in einer Flüssigkeit (**Brownsche Bewegung**).

Die Wahrscheinlichkeitsdichtefunktion der Normalverteilung ist von zwei Parametern abhängig, dem Durchschnitt und der Standardabweichung:

Sei  $\mu$  der Durchschnitt,  
 $\sigma$  die Standardabweichung, dann ist

$$\mathcal{N}(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

die Wahrscheinlichkeitsdichtefunktion der Normalverteilung.

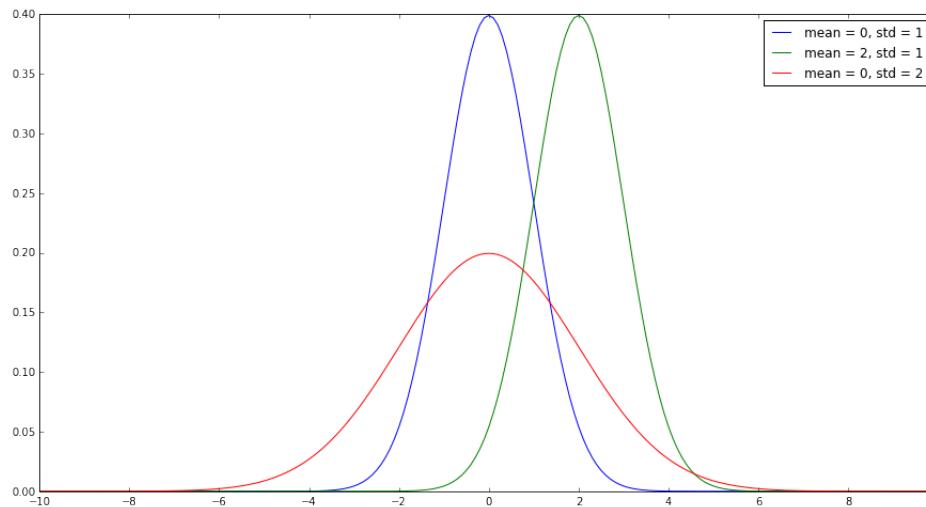


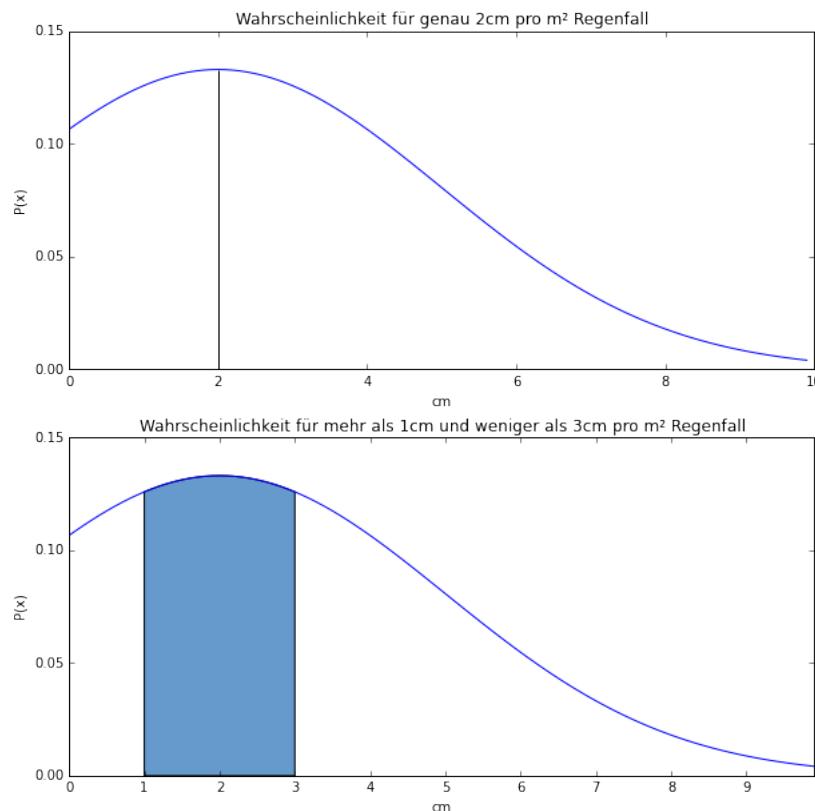
Abbildung 2.12: Dichtefunktionen von Normalverteilungen

## 2 Grundlagen

Die Abbildung (2.12) zeigt beispielhafte Ausprägungen der Dichtefunktion, an denen man erkennen kann, dass die Standardabweichung für die Amplitude und der Durchschnitt für die Phase verantwortlich ist. Um die Wahrscheinlichkeit zu berechnen, dass eine zufällige Variable  $\mathcal{X}$  mit den Grenzen  $a \leq \mathcal{X} \leq b$  eintritt ist das Integral zwischen den Grenzen der Dichtefunktion:

$$P(a \leq \mathcal{X} \leq b) = \int_a^b \mathcal{N}(\mu, \sigma^2) \quad (2.1)$$

Diese Formel lässt sich an dem Beispiel von Regenfall pro Quadratmeter veranschaulichen:



Die Wahrscheinlichkeit, dass pro Quadratmeter *genau* 2 cm Regenwasser fällt ist extrem gering, weil nicht ein Yoctometer ( $10^{-24}$ ) mehr oder weniger fallen darf. Wenn wir diese zufällige Ausprägung jedoch als Grenzen definieren, dann steigt die Wahrscheinlichkeit. Die Wahrscheinlichkeit dass zwischen 1 cm und 3 cm Regenwasser fällt ist eine viel wertvollere Aussage und ist als Integral zwischen den Grenzen der Dichtefunktion definiert.

### 2.5.2 Kodierung durch Normalverteilungen

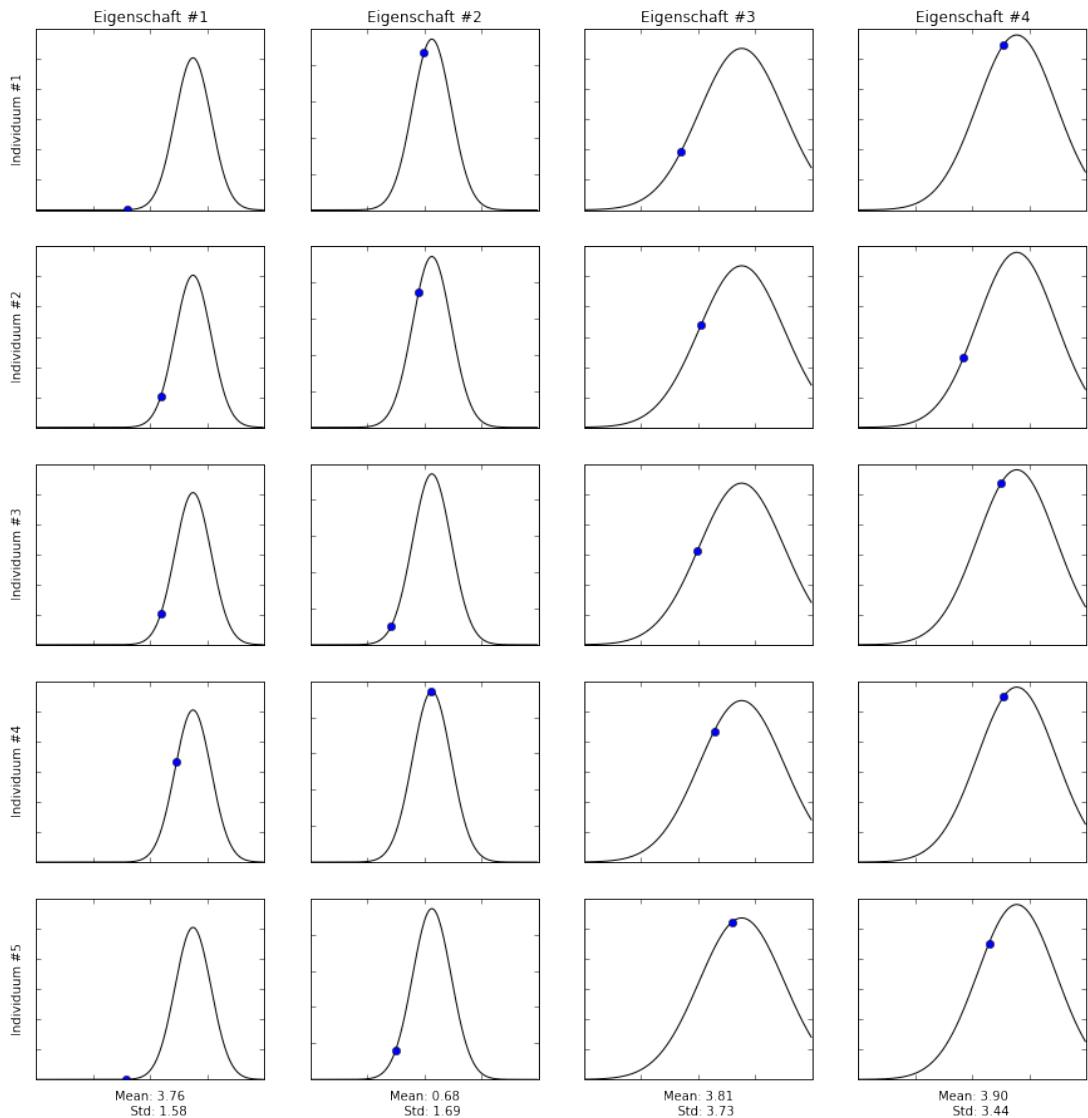


Abbildung 2.13: Kodierung der Eigenschaften durch Normalverteilungen

Stellen wir nun die Kodierung der Eigenschaften von jedem Individuum als Normalverteilung dar. Das bedeutet wir haben für jede Eigenschaft, zum Beispiel *Hornlänge*, einen Durchschnitt und eine Standartabweichung. Die Abbilung (2.13) zeigt dies beispielhaft für fünf Individuen mit jeweils 4 Eigenschaften dar.

## 2 Grundlagen

Wenn wir Individuen erstellen wollen, müssen wir aus jeder Normalverteilung eine Stichprobe (die Punkte entlang der Graphen in 2.13) nehmen. Die Kreuzung und Mutation fallen aus, dafür berechnen wir nach jeder Simulation pro Eigenschaft neue Durchschnitte und Standartabweichungen aus den besten Individuen. Damit versuchen wir die Suche auf den vielversprechendsten Raum einzuschränken. Der Ablauf der Cross Entropy Method sieht daher folgendermaßen aus:

Sei  $d$  die Anzahl der Eigenschaften,  
 $n$  die Anzahl der Individuen,  
 $\mu, \sigma$  jeweils ein Vektor der Form  $\mathbb{R}^d$ ,  
dann führen wir pro Iteration die folgenden Schritte durch:

1. Wir nehmen  $n$  Stichproben  $\theta_i$  aus  $\mathcal{N}(\mu, \sigma^2)$
2. Evaluiere  $\theta_i, i \in [1..n]$  in der Simulation
3. Wähle die besten  $p\%$  Prozent der Stichproben, wir nennen sie Eliteset
4. Berechne aus den dem Eliteset neue  $\mu$  und  $\sigma$

Eine viel detaillierte Erklärung zu diesem Thema findet sich am Beispiel von Tetris im Paper von István Szita [21].

## 3 Umsetzung in RoboCup2D

RoboCup ist ein Fußball Simulator, der seine Anfänge in 1993 in Japan, Tokyo gefunden hat. Eine Gruppe von Forschern, inklusive Minoru Asada, Yasuo Kuniyoshi und Hiroaki Kitano, haben als einen Wettbewerb unter dem Namen **Robot J-League** gestartet. Der Name stammt von einer professionellen japanischen Fußball Liga [22].

Nach einem Monat haben sie jedoch weltweit überwältigendes Feedback bekommen und haben die Initiative als internationales Projekt weitergeführt, daher kam die Umbenennung zur **Robot World Cup Initiative**, kurz RoboCup.

Die RoboCup Initiative betreibt derzeit sechs große Wettbewerbe, die sich jeweils wieder in Ligen und Subligen aufteilen lassen. Darunter fällt **RoboCup Soccer**, **RoboCup Rescue**, **RoboCup Junior**, **RoboCup Logistics**, **RoboCup @ Work** und **RoboCup @ Home**. Unsere Implementierung fällt in die Subliga **2D Soccer Simulation**, in der es darum geht in einer zweidimensionalen Welt zwei Fußballmannschaften gegeneinander antreten zu lassen.

Die Aufgabe, die wir angehen, gehört zu einem Fragment von RoboCup2D, genannt **Half Field Offense**.

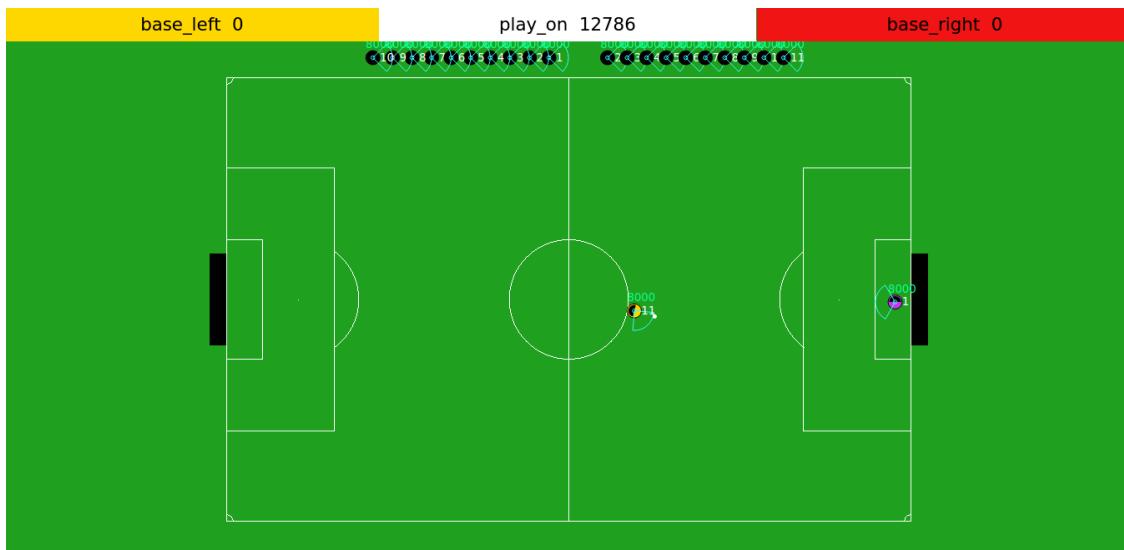


Abbildung 3.1: Screenshot von dem gesamten Spielfeld von RoboCup2D

### 3.1 Half Field Offense

Die Domäne Half Field Offense grenzt das Spielfeld auf eine Hälfte ein, sodass wir 4 Angreifer und 3 Verteidiger + Torwart haben. Diese Einschränkung vereinfacht den Such- und Zustandsraum immens und erlaubt trotzdem noch eine Wiederverwendbarkeit der Agenten, wenn eine vollständige Mannschaft aufgebaut wird.

In unserer Implementierung haben wir lediglich ein 1vs1 Szenario, also ein Angreifer gegen ein Torwart. Die Modellierung erlaubt dennoch eine nahtlose Skalierung auf ein 4vs4 Szenario, sodass weitere Parametrisierung ohne viel Aufwand ausprobiert werden können.

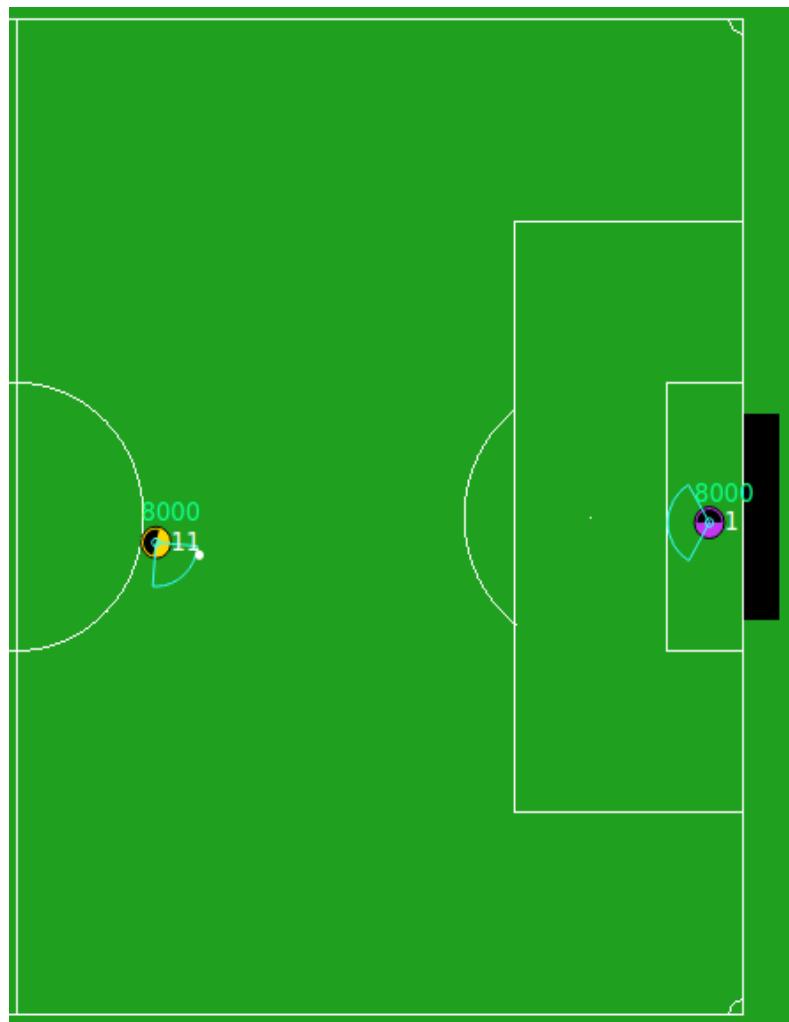


Abbildung 3.2: Screenshot von dem Spielfeld für den Subtask HFO

Im Folgenden wird die Domäne samt Zustandsraum und Aktionen erklärt, sowie ihren Einschränkungen für die Anwendung von Machine Learning Algorithmen.

### 3.1.1 Zustandsraum

Der Zustandsraum der HFO Domäne kann in den **High Level State** und den **Low Level State** aufgeteilt werden. Der Unterschied liegt in der Dimensionalität, da man aus dem Low Level State den High Level State ableiten kann. Die Zustandsräume werden durch folgende Formeln definiert:

Sei  $T$  die Anzahl der Teammitglieder,  $O$  die Anzahl der Gegner:

$$\begin{aligned} \dim(\text{High Level State}) &:= 10 + 6T + 3O \\ \dim(\text{Low Level State}) &:= 58 + 8T + 8O \end{aligned}$$

In unserem 1vs1 High Level Setting haben wir damit 13 Zustandsparameter. Vier von diesen Parametern gehören zu dem Torwart, aber da seine Position implizit durch andere Eigenschaften gegeben ist, beachten wir sie nicht. Redundante Information würde den Suchraum unnötig aufblähen und die Suche verlängern. Deshalb wurden nur die folgenden 9 Zustände bereitgestellt:

Zustandsbeschreibung	Winkel	Boole'sch	Lage	Anderes
x-Koordinaten				X
y-Koordinaten				X
Sichtrichtung	X			
Nähe zum Ball			X	
Winkel zum Ball	X			
Kann eine Ballaktion ausgeführt werden		X		
Winkel zum Mittelpunkt des Tors	X			
Größte offene Winkel zwischen Torwart und Torpfosten	X			

Der Zustand kann in 4 Kategorien unterteilt werden, die alle auf den Wertebereich von  $[-1, +1]$  reduziert wurden.

#### Winkelkodierung

Die Winkel sind im Bereich von  $[0, \pi]$  und durch folgende Formel auf den Bereich  $[-1, +1]$  transformiert <sup>1</sup>:

Sei  $f : [0, \pi] \rightarrow [-1, +1]$  die Kodierungsfunktion  
 $g : [-1, +1] \rightarrow [0, \pi]$ , die Inverse:

$$f(x) = \left( \frac{x}{\pi} - 1 \right) \cdot 2 \quad g(x) = \left( \frac{x}{2} + 1 \right) \cdot \pi$$

#### Boole'sche Kodierung

Die boole'schen Werte sind binär und  $-1$  entspricht *false* und  $1$  entsprechend *true*.

<sup>1</sup>Formel sind aus dem E-Mail Verkehr mit dem Entwickler entstanden

### Lagekodierung

Die Lagekodierung ist normalisiert auf der maximale diagonale Länge des Spielfelds die durch  $\max_l = \sqrt{l^2 + w^2}$  gegeben ist, wobei  $l$  die Länge und  $w$  die Breite ist. In dem Fall für HFO, entspricht sie  $\sqrt{2^2 + 2^2} = 2.82$ . Die folgenden Formeln sind

Sei  $f : [-\max_l, \max_l] \rightarrow [-1, +1]$  die Kodierungsfunktion  
 $g : [-1, +1] \rightarrow [-\max_l, \max_l]$ , die Inverse:

$$f(x) = \frac{x}{\max_l} \quad g(x) = x \cdot \max_l$$

### Anderes

Unter diese Kategorie fallen nur die x- und y-Koordinaten und sind trivialerweise im Bereich von  $[-1, +1]$ , weil das die Grenzen des Spielfelds sind.

### 3.1.2 Aktionsraum

Es gibt 8 parametrisierte und 6 nicht parametrisierte Aktionen. Wir haben die Algorithmen auf 5 der Aktionen trainiert die nicht parametrisiert sind. Die Aktion *CATCH* ist für Angreifer illegal und wurde deshalb weggelassen. Die folgende Aufzählung beschreibt alle Aktionen [6]:

Parametrisierte	Nicht parametrisierte
• Dash(power, degrees)	• Move
• Turn(degrees)	• Shoot
• Tackle(degrees)	• Dribble
• Kick(power, degrees)	• Intercept
• Kick_To(x-coords, y-coords, speed)	• Catch
• Move_To(x-coords, y-coords)	• No-Op
• Dribble_To(x-coords, y-coords)	
• Pass(playernumber)	

#### Aktion: Move

*Move* repositioniert den Agenten nach der vorprogrammierten Strategie von *Agent2D* und funktioniert nur dann, wenn der Spieler den Ball nicht hat.

#### Aktion: Shoot

Diese Aktion versucht den Ball mit der besten Parametern für die Aktion *Kick\_To* aufzurufen, sodass ein Tor geschossen wird.

#### Aktion: Dribble

Der Agent benutzt kurze *Kick\_To* und *Move* Sequenzen um in die Nähe des Tores zu laufen.

#### Aktion: Intercept

Der Agent versucht in die Nähe des Balls zu laufen indem er die Geschwindigkeit vom Ball in Betracht zieht. Diese Aktion ist effektiver als *Move*.

#### Aktion: No-Op

Wenn diese Aktion gewählt wird, macht der Agent nichts.

#### Abfrage der Aktionen

Jedes Spiel hatte eine maximale Laufzeit die in Frames aufgeteilt wird und jeder Agent pro Frame gefragt, welche Aktion er ausführen will. Wenn über längere Zeit keine Antwort von dem Agenten kommt, wird automatisch die No-Op Aktion ausgeführt.

### 3.1.3 Einschränkungen

Diese Domäne hat besondere Einschränkungen, die es zu einem schwierigen Problem machen. Zum einen haben wir keine Möglichkeit in die Zukunft zu schauen um herauszufinden wie gut eine Aktion für einen bestimmten Zustand war. Stattdessen müssen wir uns auf das Resultat der Simulation verlassen, dass uns lediglich am Ende sagt ob ein Tor geschossen wurde.

Zum anderen ist ein Spiel eine Abfolge von mehreren hundert Aktionen die alle gemeinsam bewertet werden und deshalb ist es unmöglich eine Fragmentierung in Teilziele zu schaffen.

Zusätzlich zum spärlichen Signal ob wir ein Tor geschossen haben, kommt der kontinuierliche 9-dimensionale Zustandsraum den wir mit normalverteiltem Rauschen empfangen haben.

## 3.2 Implementierung der Algorithmen

In diesem Abschnitt schauen wir uns die Parametrisierung der Algorithmen und den Aufbau der Simulation genauer an. Dafür waren die folgenden drei Programme zuständig:

### **Simulationsserver**

Der Simulationsserver ist in C++ geschrieben und wurde aus [6] übernommen. Er wird durch Flags beim Starten parametrisiert.

### **Agenten**

Die Agenten sind in Python geschrieben und stellen eine Erweiterung von einem der Beispielskripte dar [6]. Diese Prozesse werden mit eigenen Kommandozeilenparametern, die ihnen sagen welcher Spieler sie sind, wie sie die Kodierung des KNNs benutzen, um damit das Netz zu befüllen.

### **Koordinator**

Der Koordinator ist für die Umsetzung des GAs und den jeweiligen Kodierungen zuständig, startet den Server, die Agenten Skripte und überwacht die Simulation. Er ist, wie alle folgenden Codebeispiele, in Haskell geschrieben.

### **Rahmenbedingungen für die Simulation**

Jede Simulation bestand aus 300 Generationen und jedes Team hat pro Generation 25 Spiele gespielt. Ein Spiel (Episode) war maximal 500 Sekunden lang und wenn der Ball 50 Sekunden lang nicht berührt wurde, zählt das Spiel als verloren. Die Algorithmen die Selektion und Mutation unterstützen, wurden mit folgenden Parametern gestartet:

Generationen	300
Populationsgröße	50
Teamepisoden	25
Episodenzeit	500s
Ball nicht berührt	50s
Selektion $\alpha$	25%
Mutation $\beta$	10%

### 3.2.1 Wahrscheinlichkeitsverteilung von Aktionen

Der erste Algorithmus hat als Kodierung der Individuen eine diskrete Wahrscheinlichkeitsverteilung über 5 Aktionen benutzt. Wenn der Agent gestartet wurde samplet er jeden Zeitschritt ohne Wissen über jeglichen Zustand aus dieser Verteilung raus.

#### Kodierung

Sei das Set von allen Aktionen  $X := \{\text{Move}, \text{Shoot}, \text{Dribble}, \text{Intercept}, \text{No-Op}\}$ ,  $P(x)$  die Wahrscheinlichkeit dass  $x$  eintrifft, dann gilt:

$$\forall x \in X : P(x) \geq 0 \quad \wedge \quad \sum_{x \in X} P(x) = 1$$

Abbildung 3.3: Kodierung der Aktionen als Wahrscheinlichkeitsverteilung

#### Generierung der Individuen

Für die Generierung von einer Wahrscheinlichkeitsverteilung über  $n$  Aktionen werden  $n - 1$  zufällige Zahlen erstellt, sortiert und es wird jeweils eine 0 von vorne und eine 100 am Ende angehängt.

```

1 > let n = 5
2 > take (n-1) <\$> getRandomRs (0,100)
3 [87, 15, 55, 38]
4 > sort it
5 [15, 38, 55, 87]
6 > 0 : it ++ [100]
7 [0, 15, 38, 55, 87, 100]
```

Um die Verteilung zu erstellen werden die Zahlen dupliziert, um ein Element nach rechts verschoben und nach dem Index voneinander abgezogen.

```

1 > let l1 = [0, 15, 38, 55, 87, 100]
2 > drop 1 l1
3 [15, 38, 55, 87, 100]
4 > let l2 = it
5 > f-
6   [15, 38, 55, 87, 100]
7 - [0, 15, 38, 55, 87, 100]
8 = [15, 23, 17, 32, 13]
9 -}
10 > zipWith (-) l2 l1
11 [15, 23, 17, 32, 13]
12 > sum it
13 100
```

Damit haben wir eine Wahrscheinlichkeitsverteilung über 5 Aktionen und können uns sicher sein dass sie aufsummiert immer 100 ergibt.

### Kreuzung Version 1

Wir haben zwei verschiedene Kreuzungsmethoden ausprobiert die sich in unserer Simulation als gleichwertig herausgestellt haben. Die erste Methode wurde mit den Listen umgesetzt, aus denen sie generiert wurden (die jeweils die 0 und 100 angehängt bekommen haben). Dafür wurde elementweise der Durchschnitt berechnet und daraus entsteht dann eine neue Generatorliste aus der sich die Verteilung berechnen lässt.

```

1 > let individualA = [0, 15, 38, 55, 87, 100]
2 > let individualB = [0, 7, 22, 35, 51, 100]
3 > zipWith (\x y -> (x + y) `div` 2) individualA individualB
4 [0, 11, 30, 45, 69, 100]
```

### Kreuzung Version 2

Die zweite Methode hat beide Verteilungen genommen, die Wahrscheinlichkeiten für jeweiligen Aktionen addiert und folgendermaßen normalisiert:

Seien  $\mathcal{A}, \mathcal{B}$  diskrete Wahrscheinlichkeitsverteilungen,  $l = |\mathcal{A}|$ ,  $i \in \{1 \dots l\}$ :

$$\mathcal{C} := \left\{ \frac{(a_i + b_i)}{l} \mid a_i \in \mathcal{A}, b_i \in \mathcal{B} \right\}$$

dann ist  $\mathcal{C}$  ist ihre Verknüpfung.

Abbildung 3.4: Kreuzungsmethode 2 für Wahrscheinlichkeitsverteilungen

### Mutation

Die Mutation wurde auch auf zwei verschiedene Weisen umgesetzt. Im Kern ist jedoch die Funktion die das  $\delta$  benutzt und es mit zufälligen Vorzeichen in die Anzahl der Aktionen  $n$  aufgeteilt. Man kann sich das  $\delta$  als Veränderungsfaktor vorstellen, je höher er ist, umso unterschiedlicher wird die Wahrscheinlichkeitsverteilung.

```

1 > let delta = 20
2 > splitDelta delta 5
3 [-4, +4, +4, -4, -4] > let delta = 100
                           > splitDelta delta 4
                           [-25, +25, +25, -25]
```

### Mutation Version 1

Wir teilen das  $\delta$  in  $n - 1$  Teile auf, fügen eine 0 von vorne und 100 von hinten hinzu und addieren sie zu der Generatorliste der Verteilung. Diesmal müssen wir jedoch die Zahlen per Hand auf den Bereich von 0 – 100 begrenzen.

```

1 > let delta = 100
2 > splitDelta delta 4
3 [-25, +25, +25, -25]
4 > let mutGen = 0 : it ++ [100]
```

### 3 Umsetzung in RoboCup2D

```
5 > let child = [0, 14, 31, 49, 75, 100]
6 > {-  
7   [0, -25, +25, +25, -25, 100]  
8   + [0, 14, 31, 49, 75, 100]  
9   = [0, -11, 56, 74, 50, 200]  
10  min 0  
11  [0, 0, 56, 74, 50, 200]  
12  max 100  
13  [0, 0, 56, 74, 50, 100]  
14  sort  
15  [0, 0, 50, 56, 74, 100]  
16  -}  
17 > sort $ zipWith (((max 0 . min 100) .) . (+)) child mutGen  
18 [0,0,50,56,74,100]
```

Aus diesem Generator kann wieder eine Wahrscheinlichkeitsverteilung erstellt werden.

#### Mutation Version 2

Für die zweite Variante der Mutation spalten wir das  $\delta$  in  $n$  Teile, summieren sie elementweise mit der Verteilung, überprüfen auf die Grenzen von [0, 100] und normalisieren sie wie in Kreuzung Version 2.

```
1 > let delta = 50
2 > splitDelta delta 5
3 [-10, +10, +10, -10, -10]
4 > let mutGen = it
5 > let child = [15, 8, 34, 21, 22]
6 > zipWith (((max 0 . min 100) .) . (+)) child mutGen
7 [5,18,44,11,12]
8 > normalizeDist it
9 [5,20,48,12,15]
```

Damit bekommen wir die mutierte Wahrscheinlichkeitsverteilung zurück.

### 3.2.2 Agentenstrategien als KNN mit DCT

Für alle folgenden Algorithmen haben wir ein neuronales Netz mit 9 Eingaben, 12 LSTM (2.2.1) Neuronen und 5 Dense Ausgaben benutzt, wo ein Softmax (2.2.1) drübergelegt wurde. Damit wird der Zustand aus (3.1.1) dem Netz gegeben und er gibt uns eine Wahrscheinlichkeitsverteilung über 5 Aktionen aus (3.1.2) zurück.

Wenn der Agent spielt, wird jedes Frame der aktuelle (verrauschte) Zustand dem Netz gegeben und wir samplen aus der Wahrscheinlichkeitsverteilung die Aktion raus, die dann ausgeführt wird.

Das KNN besitzt 1108 Gewichte und wir reduzieren es mithilfe von DCT (2.3) auf 20 Koeffizienten, was eine Komprimierungsverhältnis von 1:55 ist. Die Abbildung 3.5 zeigt diese Kompressionsrate.

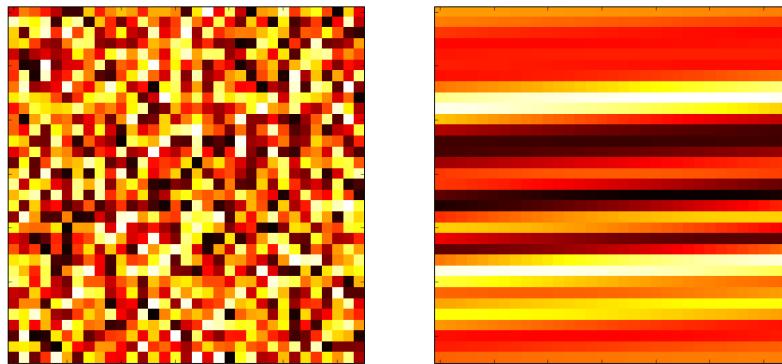


Abbildung 3.5: Kompressionsrate 1:55 mit DCT

Man erkennt eine extreme Korrelation zwischen benachbarten Gewichten die der Reihe nach das KNN befüllt haben. Die folgenden drei Algorithmen benutzen die Kompression:

- **Cross Entropy** (2.5)

Hier haben wir die 20 Koeffizienten als Normalverteilung kodiert, 25% der Population als Eltern gewählt und die restlichen 75% aus den neu berechneten Normalverteilungen der Eltern gesamplet. Die Grenzen zum Erstellen der Normalverteilung waren  $\mu = 0$  und  $\sigma \in [-1.5, 1.5]$ .

- **Neuroevolution** (2.2)

Bei der naiven Neuroevolution hat jedes Individuum die 20 Koeffizienten als Zahlen gespeichert. 25% der Population wurden als Eltern gewählt die 50% Kinder erzeugen und die restlichen 25% wurden völlig neu erstellt. Die Grenzen zur Erstellung der Koeffizienten war  $c \in [-3, 3]$ .

- **CoSyNE** (2.4)

Der CoSyNE Algorithmus benutzt die gleichen Ablauf wie Neuroevolution, mit der Ausnahme, dass er statt neue Individuen zu erstellen, genug Kinder erstellt und dann die gesamte Population im Eigenschaftsraum zufällig vermischt.



## 4 Resultate

Im folgenden Teil beschreiben wir die Resultate und versuchen diese zu begründen. Durchschnittlich hat eine Trainingsphase mit 300 Generationen, Population der Größe 50 und 25 Episoden pro Team 30 Stunden gedauert. Der Suchraum für die Neuroevolution wurde von 1108 Gewichten auf 20 Koeffizienten reduziert, welches einer Kompressionsrate 1:55 entspricht. Die Simulationen wurden auf einem Laptop mit einem Intel i5 mit Dual Core 2.9GHz und 4GB Arbeitsspeicher ausgeführt.

Nachdem wir pro Algorithmus die besten 5 Individuen ermittelt haben, ließen wir sie jeweils 10000 Spiele spielen, um die erfasste Fitness auf ihre Stabilität zu testen. Wir haben sie wie folgt quantifiziert:

Sei  $F_{Training}$  die trainierte Fitness,  
 $F_{Test}$  die neu getestete Fitness, dann ist der Fehler definiert als:

$$\text{Fehler} = \frac{F_{Entwicklung} - F_{Test}}{F_{Entwicklung}}$$

Abbildung 4.1: Berechnung des Fehlers für ein Individuum

Je kleiner die Abweichung zwischen dem Training und dem Test, umso kleiner ist der Fehler und daher können wir schließen, dass das Individuum sicherer seine versprochene Leistung bringt.

### 4.1 1vs1

Unser Lernziel für die HFO Domäne war einen offensiven Spieler zu trainieren der gegen einen vom Server gesteuerten Torwart so gut es geht Tore schießt. Das haben wir mit vier in Kapitel 2 angesprochenen Algorithmen getestet und stellen die Resultate vor.

Als stabilster und bester Algorithmus ist die naive Neuroevolution mit einer durchschnittlichen Gewinnrate von **42%** im Training und **20%** im Test. Auf Platz 2 kam CoSyNE mit 37% im Training und 12% im Test. Cross Entropy und die Wahrscheinlichkeitsverteilung der Aktionen haben zwar beide im Training 30% erreicht, aber im Test nur knapp 6% und sind damit extrem instabil.

## 4 Resultate

### 4.1.1 Wahrscheinlichkeitsverteilung der Aktionen

Die Wahrscheinlichkeitsverteilung war der erste Ansatz um zu überprüfen ob die Domäne bereits durch eine einfache Kodierung lösbar ist. Leider ging die Varianz in der Population nach der 10 Generation gegen 0 und die Verteilung sah folgendermaßen aus:

Aktionen	P(Aktion)
Move	22%
Dribble	22%
Intercept	22%
No-Op	22%
Shoot	2%

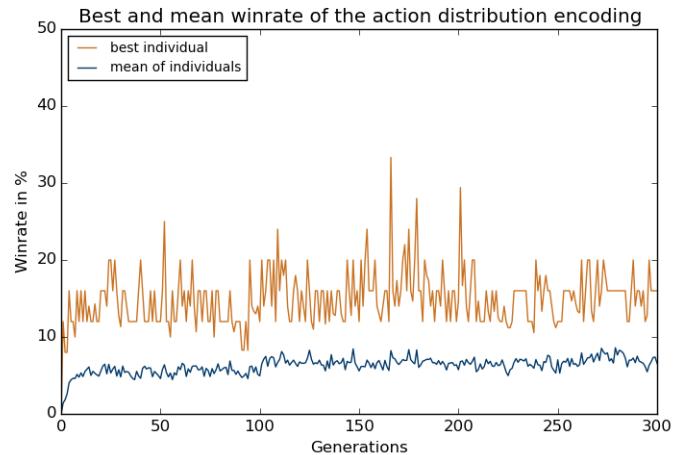


Abbildung 4.2: Fitness Graph für die Wahrscheinlichkeitsverteilung

Die gesamte Population ist zu dem Ergebnis konvergiert, dass jede Aktion gleich wahrscheinlich ist, bis auf *Shoot*. Die Schussaktion hat wahrscheinlich zu oft zu einem Schuss ins Aus geführt, was sofort das Spiel als verloren wertet.

Die maximale erreichte Fitness beträgt **33%** und schwankt eher im Bereich von [15, 25]. Leider stellt sich heraus, dass die besten Agenten Ausreißer waren und keinesfalls die durchschnittliche Gewinnwahrscheinlichkeit darstellen.

	Trained Fitness	Tested Fitness	Error
Nr.1	33.33%	5.26%	82.22%
Nr.2	29.41%	8.87%	70.52%
Nr.3	28.00%	6.06%	78.36%
Nr.4	25.00%	2.18%	91.28%
Nr.5	24.00%	6.42%	73.25%
Mean	<b>27.95%</b>	<b>5.72%</b>	<b>79.53%</b>

Tabelle 4.1: Stabilität der besten 5 Individuen

Die durchschnittliche Gewinnwahrscheinlichkeit liegt bei 5.72% und wenn man den Spieler beobachtet, kann man sich beim besten Willen nicht erklären, wie er es überhaupt schafft Tore zu schießen, da er meistens versucht ins Tor zu laufen während ihm der Ball abgenommen wird. Das ist aber auch nicht verwunderlich, da der Spieler weder weiß, wo der Torwart ist, noch ob er den Ball hat, oder wo er sich auf dem Spielfeld befindet.

**Illustration eines Torversuchs für einen der Top 5 Agenten**

Abbildung 4.3

In Abbildung 4.3 sehen wir einen vielversprechenden Torversuch von dem Individuum. Leider läuft er einfach nur in den Torwart rein und ihm wird sein Ball abgenommen. Es passiert auch sehr oft, dass

## 4.2 Cross Entropy

Die Cross Entropy Methode hat nach der statistischen Analyse keine besonders unterschiedliche Werte produziert, da sie durchschnittlich eine 2% bessere Fitness hat und die Stabilität um 1% besser ist, als wie die Wahrscheinlichkeitsverteilung.

Sie ist nach ungefähr 50 Generationen konvergiert, die maximale erreichte Fitness beträgt **32%** und schwankt im Bereich von [15, 30].

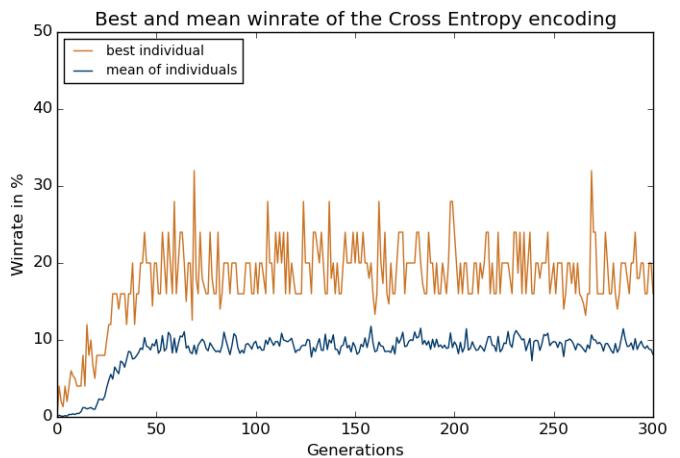


Abbildung 4.4: Fitness Graph für Cross Entropy

	Trained Fitness	Tested Fitness	Error
Nr.1	32.00%	7.26%	77.31%
Nr.2	32.00%	7.46%	76.69%
Nr.3	28.00%	7.37%	73.68%
Nr.4	28.00%	7.27%	74.04%
Nr.5	28.00%	3.46%	87.64%
Mean	<b>29.60%</b>	<b>6.56%</b>	<b>77.87%</b>

Tabelle 4.2: Stabilität der besten 5 Cross Entropy Individuen

Von den Werten sieht man kaum einen Unterschied zu der Wahrscheinlichkeitsverteilung, aber in der Simulation merkt man ein extrem aggressives Verhalten vom Spieler. Der Agent schießt den Ball sehr oft und versucht bereits nachdem er die Mitte des Spielfeldes überquert hat ein Tor zu schießen, unabhängig davon ob er in einer guten Position ist. Das führt natürlich dazu dass er öfter ins Aus schießt, ist aber wesentlich interessanter anzuschauen, da er jedes Spiel unberechenbar ist.

Es ist zu beachten, dass dieser Algorithmus die **schnellste Lernkurve** von den drei Algorithmen hatte, die im komprimierten DCT Raum arbeiten.

**Illustration eines Torversuchs für einen der Top 5 Agenten**

*Beispielhafte Abfolge von einem Spiel für den Cross Entropy Method*

### 4.3 Neuroevolution

Der Ansatz die Gewichte naiv als DCT Koeffizienten darzustellen, führte zu den besten Ergebnissen. Das stärkste Individuum hat knapp **jedes zweite Spiel gewonnen** und ist mehr als 3-mal stabiler als die Cross-Entropy Lösung. Die Fitness hat sich nach ungefähr 150 Generationen im Bereich von [30, 45] eingependelt.

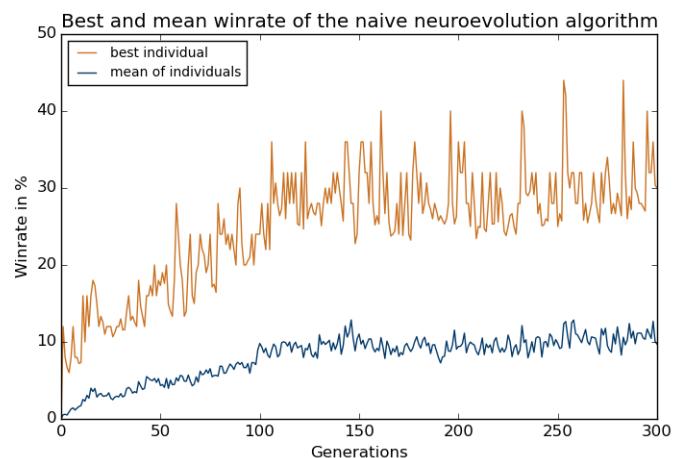


Abbildung 4.5: Fitness Graph für Cross Entropy

	Trained Fitness	Tested Fitness	Error
Nr.1	44.00%	19.04%	56.72%
Nr.2	44.00%	20.18%	54.14%
Nr.3	42.00%	20.07%	52.21%
Nr.4	40.00%	20.10%	49.75%
Nr.5	40.00%	21.28%	46.80%
Mean	<b>42.00%</b>	<b>20.13%</b>	<b>51.93%</b>

Tabelle 4.3: Stabilität der besten 5 Individuen

Die stabile Fitness ist bei knapp 20% und damit gewinnen diese Individuen durchschnittliche jedes fünfte Spiel. Hier haben wir bereits starke Anzeichen für eine Taktik gesehen. Der Agent rennt von Anfang an zum Tor, bleibt kurz vor dem Strafraum stehen und pendelt von Ecke zu Ecke bis der Torwart weiter rausgeht um ein Tor zu schießen.

Wenn er mal verliert, ist es weil er sofort zum Beginn des Spieles sich ins Aus schießt, oder zu nah am Tor ist, sodass ihm der Ball abgenommen wird. Die Tendenz am Ende des Graphen deutet darauf hin, dass bei längeren Laufzeiten noch bessere Ergebnisse möglich wären.

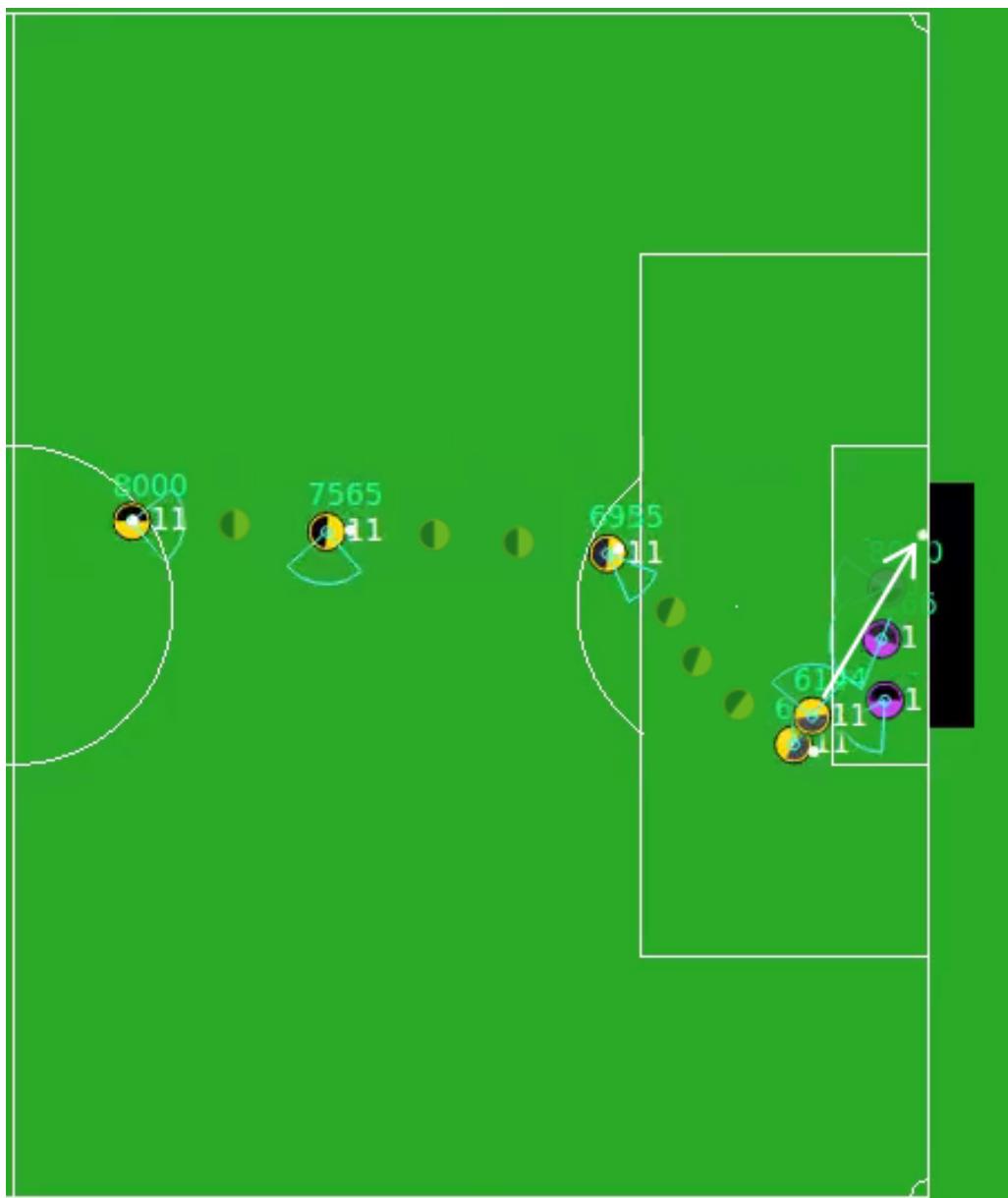
**Illustration eines Torversuchs für einen der Top 5 Agenten**

Abbildung 4.6

## 4.4 CoSyNE

Der CoSyNE Algorithmus ist in der durchschnittlichen Fitness knapp 5% hinter der Neuroevolution, hat dafür aber ganze 15% in der Stabilität verloren. Aus der Natur von dem CoSyNE gab es selbst bei der 300 Generation noch extrem unterschiedliche Individuen und eine Konvergenz war nicht zu erkennen.

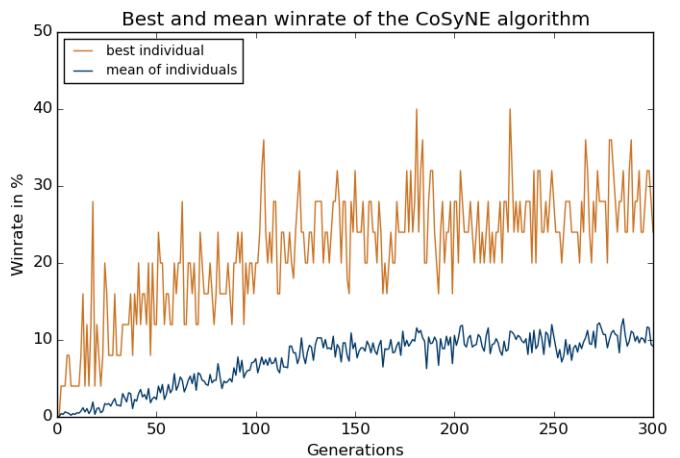


Abbildung 4.7: Fitness Graph für Cross Entropy

	Trained Fitness	Tested Fitness	Error
Nr.1	40.00%	14.21%	64.47%
Nr.2	40.00%	14.22%	64.45%
Nr.3	36.00%	12.68%	64.48%
Nr.4	36.00%	15.42%	57.16%
Nr.5	36.00%	5.75%	84.02%
Mean	<b>37.60%</b>	<b>12.45%</b>	<b>66.98%</b>

Tabelle 4.4: Stabilität der besten 5 Neurevolution Individuen

Die beste Individuen haben lediglich nur knapp 12% ihrer Spiele gewonnen und man kann eine ähnliche Taktik wie die Neuroevolution Agenten erahnen, nur wesentlich schlechter umgesetzt. Es passiert häufig, dass der Agent kurz vor dem Strafraum stehen bleibt und sich für eine sehr lange Zeit nicht bewegt. Da der Torwart nicht zu weit von dem Tor rausgeht, befinden sie sich im Deadlock bis der Agent versucht ein Tor zu schießen. Das Schießen ins Aus am Anfang des Spiels tritt hier gehäuft auf.

Eine interessante Eigenschaft von CoSyNE ist das stufenartige Entwickeln der Fitness die ab bestimmten Generationen nie wieder schlechter wird.

**Illustration eines Torversuchs für einen der Top 5 Agenten**

*Beispielhafte Abfolge von einem Spiel für CoSyNE*

#### 4.4.1 Vergleich

Im Vergleich zwischen allen Algorithmen sieht das Ranking folgendermaßen aus:

Algorithmus	E(Trained Fitness)	E(Tested Fitness)	E(Error)
Neuroevolution	42.00%	20.13%	51.93%
CoSyNE	37.60%	12.45%	66.98%
Cross-Entropy	29.60%	6.56%	77.87%
Wahrscheinlichkeitsverteilung	24.60%	6.15%	74.88%

Tabelle 4.5: Alle Algorithmen gegenübergestellt

Neuroevolution gewinnt eindeutig in allen getesteten Merkmalen und hat während den Aufnahmen den raffiniertesten Eindruck gemacht. Wir sehen, dass die Aggressivität bei der Cross-Entropy Method zwar interessante Züge macht, jedoch nicht tauglich für den Einsatz auf dem echten Spielfeld ist.

Sicherheit und die *Planung* machen auf lange Sicht viel mehr Sinn und sollten verstärkt werden. Der CoSyNE Algorithmus unterstützt diese Art von Entwicklung in dieser Dimensionalität schlechter als die naive Suche über alle Parameter. Es ist zu überprüfen ob diese Aussage für gleichzeitiges Lernen in einem 2v1 Setting übereinstimmt.

(*Link zur Best-Of-Compilation Video*)

# 5 Diskussion

In diesem Kapitel schauen wir uns die Ergebnisse im Bezug zur Problemstellung und dessen Einschränkungen an, erwägen den potenziellen Nutzen für ähnliche Probleme und stellen Verbesserungsvorschläge dar. Das Schlusswort umfasst verwandte Felder die im Bezug zu unseren Algorithmen stehen.

## 5.1 Anwendungsmöglichkeiten

Unsere Aufgabe hat Einschränkungen, die für viele Probleme aus der realen Welt zu treffen, wie seltene Fitnesssignale, kontinuierliche und verrauschte Zustandsräume, sowie aufeinander aufbauende Aktionsketten. Daher glauben wir dass die potenziellen Anwendungsbereiche wie Kontrolle von Robotern, sowie die Nutzung in Multi-Agenten Systemen möglich sind.

Vor allem war es interessant zu sehen, dass KNNs entwickelt werden können, die lediglich aus 20 Koeffizienten bestehen, extrem korellierte Gewichte produzieren und trotzdem lernen können. Ohne die Tests könnten man aus Abbildung 3.5 ableiten, dass es extrem unwahrscheinlich ist, dass das befüllte Netz in irgendeiner Art Nutzen bringt. Deshalb können wir die Annahme, dass Gewichte in neuronalen Netzen keine große Abweichung zueinander haben müssen, bestätigen. Es wäre interessant zu schauen, ob bei Netzen die viel größer sind ähnliche Ergebnisse möglich wären.

CoSyNE wirft mit der treppenförmigen Fitnesskurve die Idee auf, ob damit eine stetige Verbesserung, oder explizit keine Verschlechterung als Garantie gegeben werden kann. Diese Eigenschaft wurde bisher in keinem der Ursprungspaper untersucht.

## 5.2 Ausblick

Diese Arbeit hat leider einen festen Zeitrahmen gehabt und wir konnten vieles nicht ausprobieren. Wir sprechen mögliche Verbesserungen an, die man in zukünftigen Arbeiten beachtet werden können.

### Genetische Algorithmen

Die genetische Suche wurde mit ausgewogenen Parametern durchgeführt, die jedoch keine besondere Spezialisierung für die Anwendung bekommen haben. Darunter fällt der Mutationsparameter in CoSyNE, der im Urspurungspaper sehr hoch gewählt war [18] und die Begrenzung der Koeffizienten auf [-3,3] die wir gewählt haben, weil es keine Quellen dafür gab.

### Aufbau des neuronalen Netzes

Der Aufbau des KNNs kann mit von der Schicht aus LSTM Neuronen in Tiefe und Größe verändert werden, da wir potenziell bessere Lernfähigkeiten bekommen können. Vorallem in der Kombination mit der DCT Kompression besteht die Möglichkeit Netze mit über 1 Million Gewichten erfolgreich zu kodieren [23].

Ausser DCT gibt es noch eine andere Suchraumverringerung die auf Wavelets basiert und vielversprechendere Ergebnisse in vielen Arcade Learning Environments geschafft hat [?]. Die Wavelets kodieren den Zustandsraum auch in hochfrequenten Domänen, behalten aber im Gegensatz zu Fouriertransformationen die Stetigkeit in der Ordnung der Daten, was zu schnelleren Lösungen in der Octopusarm Aufgabe geführt hat.

### Cross Entropy Method

Die Cross Entropy Method Lösung wurde lediglich in der einfachsten Form umgesetzt und es fand weder Repopulation mit neuen Individuen statt, noch haben wir einen Mutations schritt gehabt. Das Hinzufügen von diesen Methoden würde wahrscheinlich zu besseren Lösungen führen. Viele Verberbesserungen finden sich auch in [21].

### Aktionsraum

Der High-Level Aktionsraum wurde während der Arbeit im Urspurungspaper erweitert, sodass wir Aktionen wie *Go\_To\_Ball* oder *Reduce\_Angle\_To\_Ball* nicht benutzt haben. Da die Dokumentation der übrigen Aktionen sehr spärlich ausgefallen ist, würden eigene Aktionen eine bessere Möglichkeit bieten über die Effektivität der Algorithmen zu argumentieren. Sie würden auch sehr wahrscheinlich mehr Taktiken zulassen.

### Multi-Agenten Systeme

Da die Simulation skalierend modelliert wurde, erlaubt sie uns einfaches Testen mit mehreren Agenten, wie 2vs1, oder 4vs4. Leider haben wir keine Möglichkeit gehabt diese Domänen ausführlich zu testen.

## 5.3 Schlusswort

### Implementierung für OpenAI Gym

Während der Entwicklung dieser Arbeit gab es eine Implementierung der HFO Domäne für das 1vs1 Szenario in der OpenAI Gym, das ein bekanntes machine learning Framework in Python ist. Die Umsetzung von unseren Algorithmen dafür wäre der nächste logische Schritt.

### ConvNet und CoSyNE

In [23] wird CoSyNE und DCT für die Komprimierung von über 1 Million Gewichten in einem Convolutional Neural Network benutzt und hat beeindruckende Ergebnisse für die Steuerung von einem Auto ausschließlich durch Bilddaten erzielt. Deshalb würde es sich anbieten die Verknüpfung von CoSyNE und DCT auf anderen Aufgaben, die auf hochdimensionalen korrelierten Daten basieren, anzuwenden.

### Backpropagation und Neuroevolution

Dass Neuroevolution in Aufgaben ohne vorher vorbereiteten Daten funktioniert, wurde gezeigt. Die interessantere Frage wäre wie gut diese Trainingsmethode gegenüber Backpropagation abschneidet, wenn Daten bereitgestellt werden. Der Vergleich in Geschwindigkeit der Konvergenz und das Finden von neuen Lösungen, lässt Rückschlüsse über Vorteile und Nachteile der jeweiligen Algorithmen zu.



# Literaturverzeichnis

- [1] J. Schmidhuber, “Deep learning in neural networks: An overview.” <http://www.sciencedirect.com/science/article/pii/S0893608014002135>, October 2014.
- [2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks.” <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>, 2012.
- [3] A. H. David Silver, “Mastering the game of go with deep neural networks and tree search.” <http://airesearch.com/wp-content/uploads/2016/01/deepmind-mastering-go.pdf>, 2016.
- [4] S. D. Aaron van den Oord, “Wavenet: A generative model for raw audio.” <https://arxiv.org/pdf/1609.03499.pdf>, 2016.
- [5] J. S. Jan Koutník, Faustino Gomez, “Evolving neural networks in compressed weight space,” *GECCO '10*, 2010.
- [6] P. M. Matthew Hausknecht, “Half field offense: An environment for multiagent learning and ad hoc teamwork.” <http://www.cs.utexas.edu/~pstone/Papers/bib2html-links/ALA16-hausknecht.pdf>, 2016.
- [7] D. R. B. David Beasley, “An overview of genetic algorithms: Part 1, fundamentals.” <http://www.geocities.ws/francorbusetti/gabeasley1.pdf>, 1993.
- [8] Wikipedia, “Gazelle — Wikipedia, the free encyclopedia.” <https://en.wikipedia.org/wiki/Gazelle>, 2016. [Online; accessed 12-Nov-2016].
- [9] A. Bradford, “Gazelle: Facts and pictures.” <http://www.livescience.com/27545-fun-facts-about-gazelles.html>, 2014. [Online; accessed 12-Nov-2016].
- [10] D. Whiteley, “Applying genetic algorithms to neural network problems: A preliminary report,” 1988.
- [11] R. M. David E. Moriarty, “Game playing othello neuro-evolution marker-based encoding.” <http://dx.doi.org/10.1080/09540099509696191>, 1995.
- [12] S. Herculano-Houzel, “The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost,” *Proc Natl Acad Sci USA*, 2012.
- [13] K.-l. Hsu, H. V. Gupta, and S. Sorooshian, “Artificial neural network modeling of the rainfall-runoff process,” *Water Resources Research*, vol. 31, no. 10, pp. 2517–2530, 1995.
- [14] J. S. Sepp Hochreiter, “Long short term memory.” [http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97\\_lstm.pdf](http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf), 1997.
- [15] A. G. Jürgen Schmidhuber, “Framewise phoneme classification with bidirectional lstm and other neural network architectures,” *IJCNN*, 2005.

## Literaturverzeichnis

- [16] Hecht-Nielsen, “Theory of the backpropagation neural network.” <http://ieeexplore.ieee.org/document/118638/>, 1989.
- [17] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette, “Evolutionary algorithms for reinforcement learning,” *J. Artif. Intell. Res.(JAIR)*, vol. 11, pp. 241–276, 1999.
- [18] F. Gomez, J. Schmidhuber, and R. Miikkulainen, “Efficient non-linear control through neuroevolution,” in *Proceedings of the European Conference on Machine Learning*, (Berlin), pp. 654–662, Springer, 2006.
- [19] J. S. Jan Koutník, Faustino Gomez, “Evolving deep unsupervised convolutional networks for vision-based reinforcement learning,” *GECCO ’14*, 2014.
- [20] J. J. Grefenstette, “Optimization of control parameters of genetic algorithms,” *IEEE Transactions on system, man and cybernetics*, 1986.
- [21] I. Szita and A. Lörincz, “Learning tetris using the noisy cross-entropy method,” *Neural Computation*, 2006.
- [22] Robocup, “A brief history of robocup.” [http://www.robocup.org/a\\_brief\\_history\\_of\\_robocup](http://www.robocup.org/a_brief_history_of_robocup), 2016. [Online; accessed 27-Nov-2016].
- [23] J. S. G. C. Jan Koutník, Faustino Gomez, “Evolving large-scale neural networks for vision-based reinforcement learning,” *GECCO ’13*, 2013.