

# BACHELORARBEIT

## Kooperative Neuroevolution in RoboCup2D

Alexander Isenko

Entwurf vom 11. November 2016





# BACHELORARBEIT

## Kooperative Neuroevolution in RoboCup2D

Alexander Isenko

Aufgabensteller: Prof. Dr. Claudia Linnhoff-Popien

Betreuer: Thomas Gabor, M.Sc  
Lenz Belzner, Dipl.-Medieninf.

Abgabetermin: 11. November 2016





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

München, den 11. November 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 Co-SyNE unter Einschränkung von spärlichen Fitnesssignalen, hochdimensionalen kontinuierlichen Zustandsräumen und simulationsbasierter Optimierung.

Der Suchraum wird durch diskrete Cosinustransformationen (DCT) unter der Annahme reduziert, dass benachbarte Gewichte zueinander korreliert stehen. Die Domäne ist ein Fußballsimulator, Half Field Offense (HFO), der Teams aus dem weltweiten Wettbewerb RoboCup zum Vergleichen bereitstellt. Die Implementierung erfolgt in Haskell und Python.

# Inhaltsverzeichnis

<b>1. Einführung</b>	<b>1</b>
1.1. Aufgabenstellung . . . . .	1
1.2. Motivation . . . . .	1
1.3. Aufbau der Arbeit . . . . .	1
<b>2. Definitionen</b>	<b>3</b>
2.1. Genetische Algorithmen . . . . .	3
2.1.1. Individuen . . . . .	4
2.1.2. Simulation . . . . .	5
2.1.3. Selektion . . . . .	5
2.1.4. Kreuzung . . . . .	5
2.1.5. Mutation . . . . .	6
2.1.6. Repopulation . . . . .	7
2.2. Neuroevolution . . . . .	8
2.2.1. Neuronale Netze . . . . .	8
2.2.2. Verbindung mit genetischen Algorithmen . . . . .	8
2.3. Diskrete Cosinus Transformation - DCT . . . . .	8
2.3.1. Kodierung des Suchraums . . . . .	9
2.4. Cooperative Synapsen Neuroevolution - CoSyNE . . . . .	9
2.4.1. Permutation . . . . .	9
2.5. Cross Entropy . . . . .	9
2.5.1. Normalverteilung . . . . .	9
<b>3. Umsetzung in RoboCup2D</b>	<b>11</b>
3.1. Half Field Offense . . . . .	12
3.1.1. Zustandsraum . . . . .	13
3.1.2. Aktionsraum . . . . .	13
3.1.3. Einschränkungen . . . . .	14
3.2. Implementierung der Algorithmen . . . . .	14
3.2.1. Wahrscheinlichkeitsverteilung von Aktionen . . . . .	15
3.2.2. Cross Entropy mit DCT . . . . .	17
3.2.3. Neuroevolution mit DCT . . . . .	17
3.2.4. CoSyNE mit DCT . . . . .	18
3.3. Resultate . . . . .	18
3.3.1. 1v1 . . . . .	18
3.3.2. Vergleich . . . . .	23

<b>4. Diskussion</b>	<b>25</b>
4.1. Anwendungsmöglichkeiten . . . . .	25
4.2. Coevolutionärer Aspekt . . . . .	25
4.3. Ausblick . . . . .	25
4.3.1. Genetische Algorithmen . . . . .	25
4.3.2. Aufbau des neuronalen Netzes . . . . .	25
4.3.3. Cross Entropy . . . . .	25
4.3.4. Aktionsraum . . . . .	25
4.3.5. Multi-Agenten Systeme . . . . .	25
4.4. Verwandte Felder . . . . .	25
4.4.1. Implementierung für OpenAI Gym . . . . .	26
4.4.2. Bestärkendes Lernen - Black Box RL . . . . .	26
4.4.3. Convolutional neuronale Netze und CoSyNE . . . . .	26
<b>A. Appendix</b>	<b>27</b>
A.1. Architektur . . . . .	27
A.1.1. Haskell Server . . . . .	27
A.1.2. Python Agent . . . . .	27
A.1.3. Kommunikation . . . . .	27
A.1.4. Parallelisierungsmöglichkeiten . . . . .	27
A.2. Statistik . . . . .	27
A.2.1. Lineares Laufzeit und Speicherkomplexität für Evaluation . . . . .	28
A.2.2. Stabile Varianzfunktion . . . . .	28
A.3. Problematiken . . . . .	28
A.3.1. HFO Server . . . . .	28
A.3.2. HFO Python Library . . . . .	28



# 1. Einführung

Für viele Probleme benutzt man heutzutage neuronale Netze (zitat), zum Beispiel zur Klassifikation von Bildern mit **Convolutional Neural Nets** (MNIST) oder zur Generierung von Text mithilfe von **Recurrent Neural Nets** ([karpathy.github.io/2015/05/21/rnn-effectiveness](https://karpathy.github.io/2015/05/21/rnn-effectiveness)). Diese allgemeinen Klassifikatoren (zitat) werden mithilfe von Stochastic Gradient Descent **SGD** trainiert, einer Technik die es erlaubt Minima in der Graphlandschaft anzustreben. Andere Möglichkeiten wie man neuronale Netze trainieren kann fallen dabei aus dem Fokus. Manche Domänen erlauben uns jedoch nicht SGD zu benutzen und ich spreche in dieser Arbeit anderweitige Möglichkeiten an und untersuche sie auf ihre Nützlichkeit in Domänen mit spärlicher Fitness.

## 1.1. Aufgabenstellung

In dieser Arbeit beschäftige ich mich mit der Entwicklung von neuronalen Netzen mithilfe von genetischen Algorithmen für die Domäne **Half Field Offense**. Diese hat spärliche Fitnesssignale, einen hochdimensionalen kontinuierlichen Zustandsraum und hat keine Möglichkeit die perfekte Lösung für eine Situation zu ermitteln. Aus dem Vergleich zwischen den verschiedenen Implementierungen kann man den Nutzen für andere Domänen mit ähnlichen Einschränkungen erahnen.

## 1.2. Motivation

Die Industrie interessiert sich für allgemeine Problemlösungen, die in kurzer Zeit, mit wenig Daten und am besten von alleine eine Aufgabe lösen. Leider steht das den üblichen Deep Learning Techniken gegenüber, die lange Trainingszeiten haben, extrem viele nicht homogene Daten in normalisierter Form brauchen und von Hand angepasste Fitness Funktionen benötigen die für das Ziel optimiert wurden. Wir betrachten etwas in Vergessenheit geratene Möglichkeiten zur Entwicklung von neuronalen Netzen 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 und legt verwandte Felder dar. Im Appendix wird die Implementierung vom gesamten System überschlagen, Problematiken angesprochen und Lehren die für eine zukünftige Arbeit gezogen werden können angesprochen.



## 2. Definitionen

Dieses Kapitel bietet Einblick in die Grundlagen von **Genetischen Algorithmen** im Zusammenhang mit **neuronalen Netzen** und der **Cross Entropy Method**. 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 Methode zum GA.

### 2.1. Genetische Algorithmen

Ein genetischer Algorithmus, im folgenden als *GA* abgekürzt, ist ein Optimierungsverfahren, der an von der natürlichen Selektion und Evolution inspiriert ist. Wir stellen uns beispielsweise eine Gruppe Antilopen und einen Geparden vor.

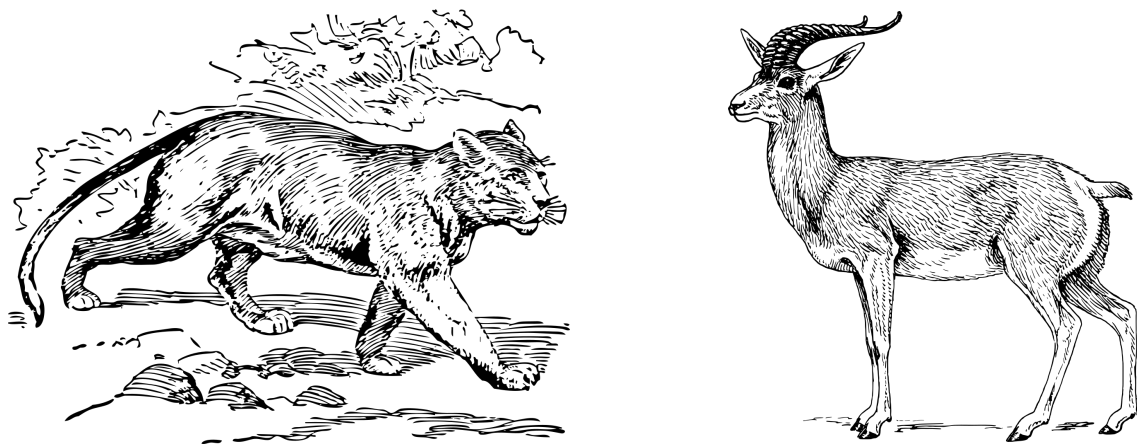


Abbildung 2.1.: Illustration von einem Geparden und einer Antilope

Sei unser Gepard durch seine Geschwindigkeit den Antilopen überlegen, dann wird die Antilopenherde über Zeit in ihrer Anzahl sinken. Dabei werden die langsamen Antilopen dem Geparden erlegen und die Schnelleren überleben. Dieser Schritt wird als **Selektion** bezeichnet. Diese werden sich fortpflanzen und mit hoher Wahrscheinlichkeit Antilopen-Babys 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 Antilope, oder auch **Individuum** genannt, hat eine eigene Fitness, die es aber bei Geburt noch nicht weiß, da sie noch nie vor einem Geparden weglaufen musste. Erst nachdem sie einmal erfolgreich entwischt ist, können wir uns vorstellen wie was ihre Fitness ist.

## 2. Definitionen

Ganz selten wird ein Antilopen-Baby geboren das ein klein bisschen längere Beine hat als alle anderen, dabei hat solche Beine keiner in dem gesamten Herde. Sie erlauben ihm schneller zu Laufen, aber leider hat diese Ausprägung nicht nur Vorteile, da die Standhaftigkeit darunter leidet. Dieser unerwartete Veränderung in den Kindern heißt **Mutation**.

Fassen wir zusammen, nachdem jede Antilope die nach dem Raubkatzenangriff überlebt und sich fortgepflanzt hat, bekommen wir hoffentlich wieder eine vollzählige Herde, das wir **Population** nennen. Nach all diesen Schritten fängt der Kampf um das Überleben wieder an und geht solange, bis sich entweder Antilopen 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 zu der Frage, wie wir ihn umsetzen.

### 2.1.1. Individuen

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

Höchstgeschwindigkeit	$50 \frac{km}{h}$
Beinlänge	$47 cm$
Gewicht	$43 kg$
Alter in Jahren	12

Tabelle 2.1.: Kodierung einer Antelope

Die Aufgabe von unserem GA ist ein oder mehrere Individuen zu finden die es schaffen vor dem Geparden wegzulaufen. Da wir aber nicht wissen ob die vorgeschlagene Kodierung gut oder schlecht ist, müssen wir Antilopen mit zufälligen Ausprägungen 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.

Ausprägung	Minimaler Wert	Maximaler Wert
Höchstgeschwindigkeit	$40 \frac{km}{h}$	$80 \frac{km}{h}$
Beinlänge	$40 cm$	$65 cm$
Gewicht	$39 kg$	$63 kg$
Alter in Jahren	2	20

Tabelle 2.2.: Grenzen für die Kodierung

Für die Kodierung unseres Individuums eignet sich ein Array von Zahlen. Um die gesamte Population darzustellen würde ein 2-dimensionales Array reichen, wobei jedes Element ein einzelnes Individuum darstellt. (evtl Grafik)

### 2.1.2. Simulation

Nachdem wir unsere Population an Antilopen erstellt haben, müssen wir sie in eine Simulationsumgebung schicken, die ihre Fitness misst. In unserem Beispiel müssten wir eine Physiks simulation mit einem Geparden programmieren, die nach einer Zeitspanne zurückgibt welche Antilopen überlebt haben.

Die Komplexität der Simulation hat einen großen Einfluss darauf ob der GA eine Lösung finden kann. (*Grafik*)

### 2.1.3. Selektion

Nachdem die Simulation vorbei ist, bekommen wir eine Population von Antilopen die jeweils überlebt hat oder nicht. Da die Fitness ist in diesem Fall binär ist, sortieren wir die Herde aller Antilopen absteigend, wobei das Überleben natürlich mehr wert ist.

Nach dem Sortieren müssen wir ein prozentualen Betrag wählen, wieviele Eltern wir aus der Population wählen. In unserer Implementierung nennen wir diesen Parameter  $\alpha$ . (*Grafik*)

### 2.1.4. Kreuzung

Die guten Individuen wurden ausgewählt und können sich fortpflanzen. Dafür nehmen wir jeweils zwei Individuen und nehmen zufällig die Ausprägungen von jeweils dem Vater und der Mutter.

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

Tabelle 2.3.: Kodierung des Vaters

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

Tabelle 2.4.: Kodierung der Mutter

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

Tabelle 2.5.: Kodierung vom Kind Nr.1

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

Tabelle 2.6.: Kodierung vom Kind Nr.2

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

## 2. Definitionen

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

Diese Art und Weise zwei Individuen zu kreuzen nennt sich **n-point crossover**, weil wir die Kodierung an undefiniert vielen Stellen unterbrechen und wieder zusammensetzen. Es gibt noch andere Kreuzungsmethoden die 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 das Alter in Zusammenhang mit der Höchstgeschwindigkeit steht, weil ältere Tiere nicht mehr die Leistung bringen können die sie mal gebracht haben. Wenn nun ein Kind gezeugt wird, dass in unserem Beispiel ein hohes Alter vererbt, bringt ihm die Höchstgeschwindigkeit nichts mehr. Deshalb wäre es besser, wenn diese Ausprägungen zusammen übernommen werden, weil dadurch eine höhere Fitness garantiert werden kann. Kreuzungsmethoden die die Kodierung nicht oft aufspalten verletzen diese Eigenschaft seltener als *n-point crossover*.

Es ist zu beachten dass je nach Implementierung nur eins der beiden Kinder weiter verwendet wird, weil dann die Repopulation einfach einfacher ist, die Varianz nicht zu stark gesenkt wird und keinerlei Information verloren geht, weil die Eltern in der Population 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  $\beta$ .

Ausprägungen
62 $\frac{km}{h}$
42 <i>cm</i>
51 <i>kg</i>
8

Tabelle 2.7.: Kodierung vom Kind Nr.2

Ausprägungen
62 $\frac{km}{h}$
42 <i>cm</i>
45 <i>kg</i>
8

Tabelle 2.8.: Mutierte Kodierung vom Kind Nr.2

Der Python Code sieht hier folgendermaßen aus:

```
kinder = [k1, k2...]
beta   = 0.1
for i in range(kinder.length):
    for j in range(kodierung.length):
        if (r > beta):
            kinder[i][j] = sampleNewFrom(kodierung[j].range)
```

Dieser Schritt ist wichtig damit man die Möglichkeit hat aus lokalen Fitnessminimas rauszukommen, weil es schnell passieren kann dass sich über Generationen gleichwertige Ausprägungen weiterverbreiten, da sie die derzeit beste Lösung vorschlagen. Ohne Mutation würde der GA zu dieser Lösung konvergieren ohne Bessere in Betracht zu ziehen.

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 Simulation neu gestartet werden kann. Wir stellen das Problem wieder an einem Beispiel dar.

```
population = [i1,i2,...]           # population.length = 10
alpha      = 0.4
eltern     = selection(population, alpha) # eltern.length = 4
kinder     = crossover(eltern)         # kinder.length = 4
beta       = 0.1
mutkinder  = mutation(kinder, beta)    # mutkinder.length = 4

newpopulation = eltern + mutkinder    # newpopulation.length = 8
```

Man kann einfach erkennen dass uns zwei Individuen zum Neustart der Simulation fehlen. Dieses Problem kann man auf viele Weisen angehen, die ihre 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 inherent Neue. Der Nachteil ist jedoch die gesenkte Varianz in der Population und die erhöhte Wahrscheinlichkeit zum Feststecken in einem lokalen Fitnessminima.

## 2. Definitionen

### **Nicht selektiere Individuen nachfüllen**

Man kann die nicht benutzen Individuen aus der vorherigen Population zum Auffüllen benutzen, was sich aber nur dadurch begründen würde, wenn die Chance besteht dass sie in der erneuten Simulation besser abschneiden als bisher. Ansonsten nehmen sie einen Platz einem potenziell besseren Individuum weg.

### **Neue Individuen erstellen**

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

## **2.2. Neuroevolution**

“Neuroevolution beschäftigt sich mit der Verknüpfung von Genetischen Algorithmen und Neuronalen Netzen”

### **2.2.1. Neuronale Netze**

“Neuronale Netze sind eine riesige Gleichung”

“Formel von Machine Learning zeigen”

“Backpropagation wird angesprochen aber nicht ausführlich erklärt”

### **Dense Ebene**

“Kleines Beispielnetz aufmalen und evtl durchrechnen”

### **LSTM Ebene**

“Kleines Beispielnetz aufmalen und evtl durchrechnen”

### **Softmax Ebene**

“Erklärung bieten wieso es sowas gibt und was für eine Formel angewendet wird”

### **2.2.2. Verbindung mit genetischen Algorithmen**

“Die Verknüpfung findet in der Kodierung von den Individuen statt - wir nehmen eine (naive) Darstellung von allen Gewichten” “Problematik -> Folgerung zu DCT”

## **2.3. Diskrete Cosinus Transformation - DCT**

“Der Suchraum kann durch die sog. DCT auf beliebige Dimensionalität eingeschränkt werden, wenn bestimmte Annahmen getroffen werden können”

“Fouriertransformationen machen folgendes...”



“Es gibt eine Inverse die aus einem n-stelligen liste eine m-stellige macht, wo die Zahlen korelliert sind (Beispiele)”

### 2.3.1. Kodierung des Suchraums

“Die Kodierung besteht aus eine 20-stelligen Liste wie im Paper (Cosyne), wo damit 2k Gewichte entwickelt wurden”

## 2.4. Cooperative Synapsen Neuroevolution - CoSyNE

“CoSyNE wurde vom Prof.Dr.Schmidhuber an der ETH Zürich entwickelt und hat damit sehr viele anderen Algorithmen in den Schatten gestellt”

“Methodik, ist wie GA bloß mit einer Aktion mehr die statt spielbare ‘Policies’, nur im Koeffizientraum entwickle”

“Dies erlaubt eine (zitat) Kooperative Entwicklung von Koeffizienten für die nachfolgenden Inv.DCT”

“Kein Crossover, geringe Mutation”

“Beispiele vom Erfolg”

### 2.4.1. Permutation

“Wir transponieren, shuffeln und transponieren zurück”

## 2.5. Cross Entropy

“Cross Entropy ist eine andere Möglichkeit um die Individuen darzustellen, anstatt von Zahlen, hab ich nun pro Coeffizient eine Normalverteilung habe mit Mean und STD”  
(cite boer07)

### 2.5.1. Normalverteilung

“Was ist eine Normalverteilung”

“Wie programmiere ich eine Normalverteilung selber, Box-Muller, Randomness, Haskell-code Beispiel”



### 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.

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 hat 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 Fragement von RoboCup2D, genannt **Half Field Offense**.

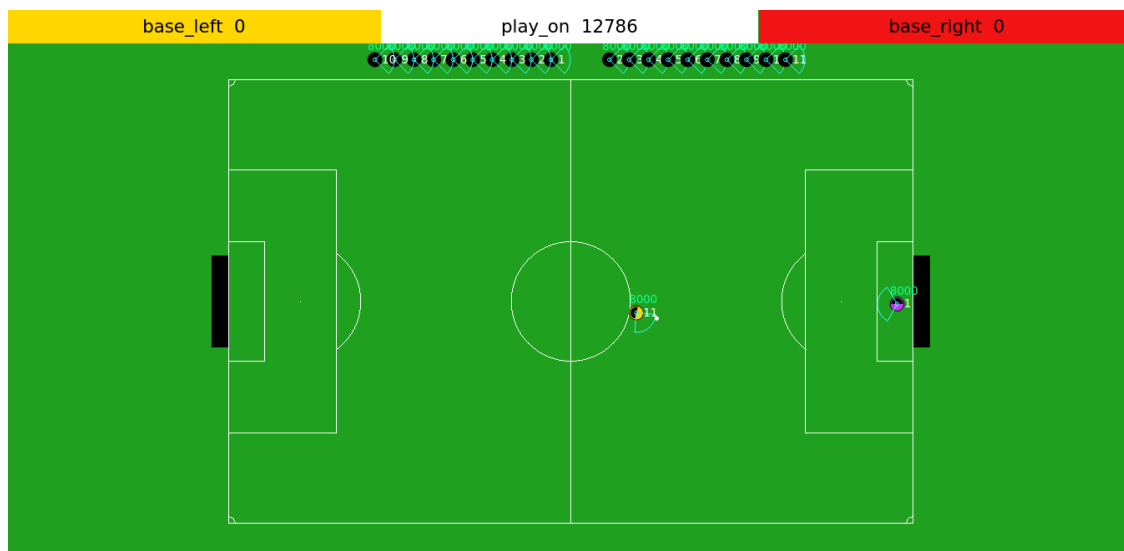


Abbildung 3.1.: Screenshot von dem gesamten Spielfeld von RoboCup2D

### 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 potenziell eine Wiederverwendbarkeit der Agenten, wenn eine vollständige Mannschaft aufgebaut wird.

In unserer Implementierung haben wir lediglich ein 1v1 Szenario, also ein Angreifer gegen ein Torwart. Diese sieht jedoch explizit eine nahtlose Skalierung auf ein 4vs4 Szenario vor, sodass weitere Parametrisierung ohne viel Aufwand ausprobiert werden können.

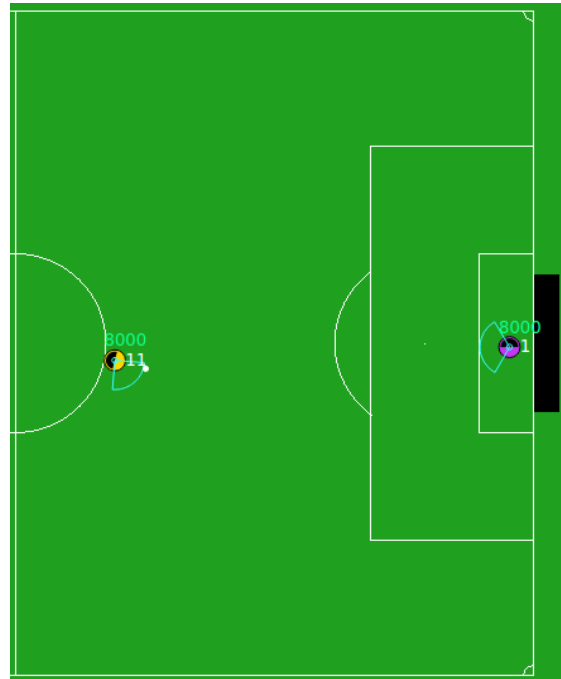


Abbildung 3.2.: Screenshot von dem Spielfeld für den Subtask HFO

Im Folgenden wird die Domäne samt Zustandsraum und Aktionen erklärt, sowie ihre 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 ist lediglich in der Dimensionalität, da man aus dem Low Level State den High Level State ableiten kann. Die Zustandsräume werden durch folgende Formeln aufgespannt:

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

$$\text{High Level State} := 10 + 6T + 3O$$

$$\text{Low Level State} := 58 + 8T + 8O$$

In unserem 1v1 High Level Setting haben wir damit 13 Zustandsparameter. Vier von diesen Parametern gehören zu dem Torwart, aber da seine Position implizit durch andere Ausprägungen gegeben ist, werden sie nicht beachtet. Redundante Information würde den Suchraum unnötig aufblähen und die Suche verlängern. Folgende 9 Zustände wurden bereitgestellt:

Zustandsbeschreibung	Wertebereich	Kontinuierlich	Boole'sch
x Koordinaten	$[-1, +1]$	X	
y Koordinaten	$[-1, +1]$	X	
Sichtrichtung	$[-1, +1]$	X	
Nähe zum Ball	$[-1, +1]$	X	
Winkel zum Ball	$[-1, +1]$	X	
Kann eine Ballaktion ausgeführt werden	$[-1, +1]$		X
Winkel zum Mittelpunkt des Tors	$[-1, +1]$	X	
Größte offene Winkel zwischen Torwart und Torpfosten	$[-1, +1]$	X	

Tabelle 3.1.: Zustandsraum von HFO 1vs1

(Muss hier eine Erklärung wie der Zustand kodiert war hin, also Normalisierung der Winkel?

Wäre dann eigentlich abschreiben ab 15.1.1 von <https://github.com/LARG/HFO/blob/master/doc/manual.pdf>)

### 3.1.2. Aktionsraum

Es gibt 8 parametrisierte und 6 nicht parametrisierte Aktionen. Wir haben die Algorithmen über 5 der 6 Aktionen ohne zusätzlichen Argumente trainiert. Die Aktion *CATCH* ist für Angreifer illegal und wurde deshalb weggelassen. Die folgende Aufzählung beschreibt alle Aktionen: (Genauere Erklärung von benutzen Aktionen kommt noch)

#### Parametrisierte

- Dash(power, degrees)
- Turn(degrees)
- Tackle(degrees)
- Kick(power, degrees)
- Kick\_To(x-coords, y-coords, speed)
- Move\_To(x-coords, y-coords)
- Dribble\_To(x-coords, y-coords)

#### Nicht parametrisierte

- Move
- Shoot
- Dribble
- Intercept
- Catch
- No-Op

### 3. Umsetzung in RoboCup2D

Jedes Spiel hatte eine maximale Zeit die in Frames aufgeteilt war und jeder Agent wird zu jedem Frame gefragt ob er eine neue Aktion ausführen will. Wenn ein Timeout von einem festen Zeitabstand kommt, wird pauschal die No-Op Aktion ausgeführt.

#### 3.1.3. Einschränkungen

Diese Domäne hat viele Einschränkungen wenn man sie mit herkömmlichen Machine Learning Tasks vergleicht (*Vergleich Moonrover, Roboterarm etc.*). Zum einen erlaubt sie uns wegen der Implementierung nicht in die Zukunft zu propagieren und zu schauen wie gut eine Entscheidung ist. Wir haben eine Simulation die erst nachdem ein Spiel fertig ist ein Fitnesssignal sendet und wir daraufhin abzuleiten müssen ob die lange Aktionsketten die wir ausgeführt haben uns zum Erfolg führten. Diese Eigenschaft nennt sich **sparse Fitness** und findet sich in Beispielen wie (*Zitat*)

*(Simulation based learning)*

*(Kontinuierlicher Zustandsraum, hohe Abstraktion)*

## 3.2. Implementierung der Algorithmen

Der ausführliche Aufbau der Algorithmen wird näher im Appendix erklärt, hier schauen wir uns die Parametrisierung grobe Funktionsweise an. Die Simulation kann in die folgenden drei Teile unterteilt werden.

### Simulationsserver

Der Simulationsserver ist in C++ geschrieben und wurde 1-zu-1 aus [cite HFO] übernommen. Er wird durch Flags beim Starten parametrisiert.

### Agenten

Die Agenten sind in Python geschrieben und stellen eine Erweiterung von einem der Beispielskripte dar [cite HFO]. Diese Prozesse werden auch mit eigenen Kommandozeilenparametern aufgerufen.

### Koordinator

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

### Simulation

Jedes Team hat pro Generation 25 Spiele gespielt und die gesamte Simulation bestand aus insgesamt 375000 Spielen. Die Episodenzeit wurde auf 500 Echtzeitsekunden beschränkt, da ansonsten die simulierte Zeit pro Spiel nicht praktikabel war.

Für alle Simulationen galten die folgenden Rahmenbedingungen:

Generationen	300
Populationsgröße	50
Teamepisoden	25
Episodenzeit	500s
Ball nicht berührt	50s
$\alpha$	0.25
$\beta$	0.10

### 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

Set von allen Aktionen  $X := \{\text{Move, Shoot, Dribble, Intercept, No-Op}\}$

$$\forall x \in X : P(x) \geq 0$$

$$\sum_{x \in X} P(x) = 1$$

#### Kreuzung

Die Kreuzung wurde auf zwei verschiedenen Arten umgesetzt, wobei sie im Vergleich an der vollständige Simulation weder neueartige Lösungen entwickelt haben, noch die Konvergenzzeit beeinflusst wurde.

#### Generator

Die erste Methode kam aus der Idee wie man mit einer absehbaren Laufzeit eine Wahrscheinlichkeitsverteilung über  $n$  Aktionen erstellt. Dafür werden  $n - 1$  zufällige Zahlen erstellt, als Liste verpackt, sortiert und jeweils eine 0 von vorne und eine 100 am Ende angehängt.

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

Anschließend wird diese Liste dupliziert und um ein Element nach rechts verschoben und paarweise voneinander abgezogen.

```
> let l1 = [0, 15, 38, 55, 87, 100]
> drop 1 l1
```

### 3. Umsetzung in RoboCup2D

```
[15,38,55,87,100]
> let l2 = it
> {-
  [15, 38, 55, 87, 100]
- [ 0, 15, 38, 55, 87, 100]
= [15, 23, 17, 32, 13]
-}
> zipWith (-) l2 l1
[15, 23, 17, 32, 13]
> sum it
100
```

Damit haben wir eine Wahrscheinlichkeitsverteilung über 5 Aktionen und können uns sicher sein dass sie aufsummiert immer 100 ergibt. Die Kreuzung von zwei solcher Individuen wurde mit den jeweiligen Listen umgesetzt, aus denen sie generiert wurden. Dafür wurde elementweise der Durchschnitt berechnet und daraus entsteht dann eine neue Generatorliste aus der sich die Verteilung berechnen lässt.

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

#### Normalisierung

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

Seien  $\mathcal{A}, \mathcal{B}$  die diskreten Wahrscheinlichkeitsverteilungen die verknüpft werden sollen:

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

#### Mutation

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

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

#### Generator

Wir erstellen teilen das  $\delta$  in  $n - 1$  Teile auf, fügen eine 0 von vorne und 100 von hinten hinzu und verknüpfen es analog wie in der Kreuzung mit dem Ausgangsgenerator. Diesmal



müssen wir jedoch die Zahlen per Hand auf den Bereich von 0 – 100 begrenzen.

```
> let delta = 100
> splitDelta delta 4
[-25, +25, +25, -25]
> let mutGen = 0 : it ++ [100]
> let child = [0, 14, 31, 49, 75, 100]
> {-
  [0, -25, +25, +25, -25, 100]
+ [0, 14, 31, 49, 75, 100]
= [0, -11, 56, 74, 50, 200]
min 0
  [0, 0, 56, 74, 50, 200]
max 100
  [0, 0, 56, 74, 50, 100]
sort
  [0, 0, 50, 56, 74, 100]
-}
> sort $ zipWith (((max 0 . min 100) .) . (+)) child mutGen
[0,0,50,56,74,100]
```

Aus diesem Generator kann wieder eine Wahrscheinlichkeitsverteilung erstellt werden.

#### Normalisierung

Bei der Lösung mit der Normalisierung generieren wir uns wieder die Liste aus dem  $\delta$ , summieren sie elementweise mit der Verteilung, überprüfen ob die Grenzen von  $[0, 100]$  überschritten wurden und normalisieren sie wie in der Kreuzung.

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

Damit bekommen wir wieder eine veränderte diskrete Verteilung zurück.

#### 3.2.2. Cross Entropy mit DCT

“Parametrisierung”

#### 3.2.3. Neuroevolution mit DCT

“Parametrisierung”

### 3.2.4. CoSyNE mit DCT

“Parametrisierung”

## 3.3. 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. Die Simulationen wurden auf einem Laptop mit einem Intel i5 mit 2.9GHz und 4GB Arbeitsspeicher ausgeführt.

Nachdem wir pro Algorithmus die besten 5 Individuen ermitteln, lassen wir sie jeweils 10000 Spiele spielen, um die erfasste Fitness auf ihre Stabilität zu testen.

Sei  $F_{Entwicklung}$  die entwickelte Fitness und  $F_{Test}$  die neu getestete Fitness. Die Stabilität wird danach gemessen wie gering die Abweichung von  $F_{Test}$  zu  $F_{Entwicklung}$  ist. Je kleiner die Abweichung, umso stabiler und sicherer spielt das Individuum.

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

### 3.3.1. 1v1

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.

### Wahrscheinlichkeitsverteilung

Die Wahrscheinlichkeitsverteilung war der erste naive 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%

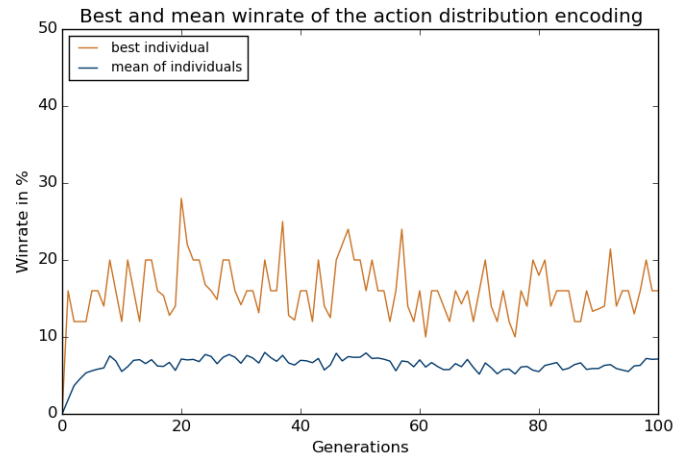


Abbildung 3.3.: Fitness Graph für die Wahrscheinlichkeitsverteilung

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 zu Gunsten des Torwarts beendet.

Die maximale erreichte Fitness beträgt 28% und schwankt im Bereich von [10, 25]. Leider stellt sich heraus dass die besten fünf Werte Ausreißer waren und keinesfalls die durchschnittliche Gewinnwahrscheinlichkeit darstellen.

	Trained Fitness	Tested Fitness	Error
Nr.1	28.00%	6.82%	75.64%
Nr.2	25.00%	5.54%	77.84%
Nr.3	24.00%	5.40%	77.50%
Nr.4	24.00%	6.33%	73.62%
Nr.5	22.00%	6.65%	69.78%
Mean	<b>24.60%</b>	<b>6.15%</b>	<b>74.88%</b>

Tabelle 3.2.: Stabilität der besten 5 Wahrscheinlichkeitsverteilungs Individuen

Die durchschnittliche Gewinnwahrscheinlichkeit liegt bei 6.15% und wenn man den Spieler beobachtet kann man sich beim besten Willen nicht erklären, wie er überhaupt schafft Tore zu schießen, da er meistens versucht ins Tor zu laufen während ihm der Ball abgenommen wird. Es ist aber auch nicht verwunderlich, da der Spieler weder weiß wo der Torwart ist, ob er den Ball hat, noch wo er sich auf dem Spielfeld befindet.

([Link zum Video](#))

### 3. Umsetzung in RoboCup2D

#### Cross Entropy

Die Cross Entropy Methode hat sehr interessante Ergebnisse produziert, da sie durchschnittlich eine 4% bessere Fitness hat, die Stabilität jedoch um 3% schlechter ist, als wie die Wahrscheinlichkeitsverteilung.

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

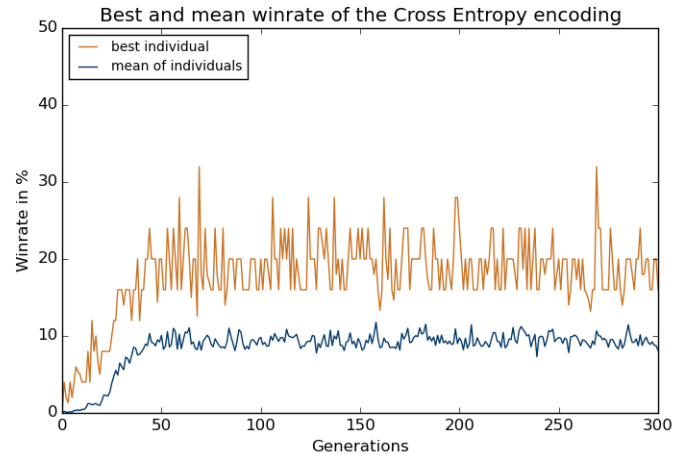


Abbildung 3.4.: Fitness Graph für Cross Entropy

	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 3.3.: 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 wieder dazu dass er öfter ins Aus schießt, ist aber wesentlich interessanter anzuschauen, da er von Spiel zu Spiel unberechenbar ist.

*(Link zum Video)*

## Neuroevolution

Der Ansatz die Gewichte naiv als DCT Koeffizienten darzustellen führe 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.

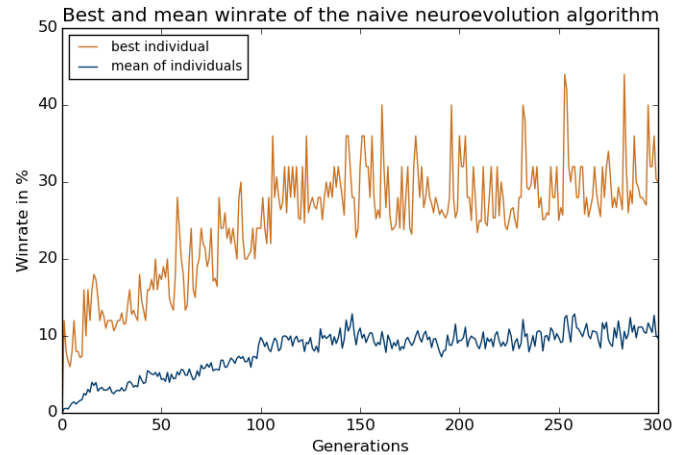


Abbildung 3.5.: Fitness Graph für Cross Entropy

	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 3.4.: Stabilität der besten 5 Neurevolution Individuen

Die stabile Fitness ist bei knapp 20% und damit gewinnen diese Individuen durchschnittliche jedes fünfte Spiel. Die Spielweise von diesem Ansatz könnte man *geplant* erklären, da der Spieler oft zum Tor rennt, kurz vor dem Strafraum stehen bleibt und von Ecke zu Ecke pendelt bis er den Torwart etwas aus dem Tor gelockt hat 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.

(*Link zum Video*)

#### 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.

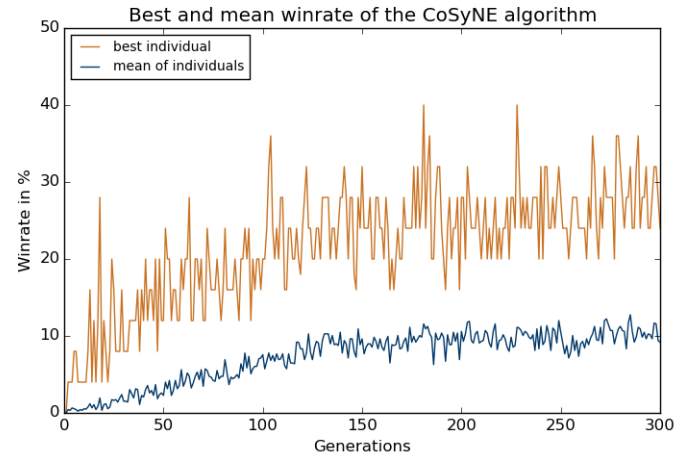


Abbildung 3.6.: Fitness Graph für Cross Entropy

	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 3.5.: 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 erraten, 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 am Anfang des Spiels tritt hier auf gehäuft auf.

*(Link zum Video)*

### 3.3.2. 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 3.6.: Alle Algorithmen gegenübergestellt

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

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)*





## 4. Diskussion

“Im Folgenden bespreche ich die Nützlichkeit der Ergebnisse”

### 4.1. Anwendungsmöglichkeiten

“Wenn starke Einschränkungen bestehen sind, waren diese Algorithmen zielführend”

### 4.2. Coevolutionärer Aspekt

“Die Annahmen für Netze zu treffen ist interessant gewesen, hat anscheinend auch für 30k Gewichte geklappt, sollte man später zum checken”

### 4.3. Ausblick

“Im Folgenden gebe ich meinen Senf zu den Parametrisierungen an und wenn die Welt perfekt wäre, was ich dann machen würde”

#### 4.3.1. Genetische Algorithmen

“Hab nicht an den Parameteren gespielt”

#### 4.3.2. Aufbau des neuronalen Netzes

“Hab nicht verschiedene Netze ausprobiert”

#### 4.3.3. Cross Entropy

“Hab die naivste Implementierung genommen, könnte viel viel besser werden, wenn man Grips reinsteckt”

#### 4.3.4. Aktionsraum

“Hätte bessere Aktionen definieren können, die schlauere Spielzüge erlauben, da ich die Vermutung habe dass die jetzigen ein Hardcap an Qualität bieten”

#### 4.3.5. Multi-Agenten Systeme

“Hatte noch keine Möglichkeit in diese Domäne ausführlich zu testen”

### 4.4. Verwandte Felder

“Folgende Reads sind nice”

#### *4. Diskussion*

##### **4.4.1. Implementierung für OpenAI Gym**

“Update für OpenAI Gym, meine Implementierung wird folgen”

“Keine Multi-Agenten Setting möglich, dsw ist meine Arbeit nicht umsonst gewesen”

##### **4.4.2. Bestärkendes Lernen - Black Box RL**

“Wenn man die Netze als Policy betrachtet ist meine Arbeit eine Anwendung im Reinforcement Learning Bereich gewesen”

##### **4.4.3. Convolutional neuronale Netze und CoSyNE**

“Warum ist die Annahme für homogene Netzstrukturen besser für Convolutional Networks”

“Evtl ist CoSyNE dort richtig geil”

##### **Roboterarm**

“Bilder, Graphen und Parametrisierung zeigen + Link + Zitat”

##### **Rennspiel**

“Bilder, Graphen und Parametrisierung zeigen + Link + Zitat”

# A. Appendix

“Da der Fokus sehr stark auf dem Machine Learning Bereich lag, aber die Domäne viele Tücken hatte und die Implementierung einen Großteil der Zeit in Anspruch genommen hat, wird es hier behandelt”

## A.1. Architektur

“Die Architektur des gesamten Projektes wurde in Haskell geplant, da ich mit externen und unsicheren Implementierungen arbeite”

“Automatische Docs, weil ich gute Kommentare schreibe” “Hat geholfen Fehler schneller zu finden und Codereuse ist super geil gewesen”

“Bild von der Kommunikation zeichnen”

### A.1.1. Haskell Server

“Module”

“Typklassen sind geil”

“Globale Config ist nice”

“Automatische Serialisierung mit Aeson”

### A.1.2. Python Agent

“Module”

“CMD Parser”

“Keras”

### A.1.3. Kommunikation

“Haskell <-> Python: JSON ist von Python nativ als Dict unterstützt” “Python <-> Server: FFI Python to C++ (HFO) + (Zitat)”

### A.1.4. Parallelisierungsmöglichkeiten

“Bottleneck ist die Kommunikation über JSON-Files”

## A.2. Statistik

“Für Cross Entropy hab ich Varianz und STD mit Seed gebraucht, für MonadRandom gabs keine Implementierung, dsw hab ich eine eigne gemacht”

### **A.2.1. Lineares Laufzeit und Speicherkomplexität für Evaluation**

“Folds sind supernice, minimale Erklärung, Links zu Gabriels Blog, Vorzeigen von Effizienz”

### **A.2.2. Stabile Varianzfunktion**

“Catastrophic Cancellation, erste Lösung, zweite Lösung von Gabriel mit E-Mail Austausch”

## **A.3. Problematiken**

“Server ist scheiße, nicht besonders gut dokumentiert”

### **A.3.1. HFO Server**

“Erfolgreicher Durchlauf ist abhängig vom Computer, bzw. von der Leistung”

“Undefinierte Lags zwischen Step 2k-8k”

“Bedienung vom Visualizer ist nicht richtig erklärt”

“Visualizer produziert nicht benutzbare logs, nur das erste Spiel ist ‘abspielbar’, rest ist korrupt ”

“Einloggen von Spielern nimmt sehr viel Zeit in Kauf, dafür muss ein Austausch von Policies on-the-fly passieren”

“Visualzer bricht bei 24k Steps ab, aber ohne ihn funktioniert die Simulation nicht, laggt nur rum”

“Es gibt keine Zeitangaben wie lange ein Agent nicht aktiv sein darf, wenn er keine Entscheidung abgibt wird er ignoriert, dieser Fall sollte behandelt werden für kompetative Benutzung”

### **A.3.2. HFO Python Library**

“Kodierung von Zuständen in denen sich der Agent befindet gibt ein Hexdump zurück, man muss per Hand die ENUMS herausfinden und hardcoden”