Künstliche Intelligenz

Finn Harbeke

18. Oktober, 2018

Contents

1 Intro

1.1 KI

1.2 Meine Projekte

2 Snake

2.1 Idee und Ziel

Mein Ziel bei dem ersten Teil meiner Arbeit ist es, verschiedene Arten von Evolutionen zu vergleichen. Diese Evolutionen sollen eine möglichst gute Künstliche Intelligenz erschaffen, die das Game "Snake" spielt. Wichtig ist, dass bei solchen künstlichen Evolutionen das Individuum, also eine bestimmte künstliche Intelligenz, nicht selbst lernt. Es reagiert auf eine Situation immer gleich. Der eigentliche Prozess des Lernens geschieht über mehrere Generationen, wie wir es von der evolutionären Anpassung kennen. Es gibt eine Gruppe von Individuen mit DNA, die versuchen, eine bestimmte Aufgabe zu erfüllen. Im nächsten Schritt wird die Gruppe anhand ihrer Leistungen selektioniert. Und die Besseren machen Kinder, die "gute" DNA von ihren Eltern erben. Dabei sind die 3 Faktoren, die die Effizienz einer Evolution vorrangig beeinflussen die Selektion, das "Crossover" die Art des DNA-Mischens und die Mutation. Der Hauptfokus der Arbeit liegt im Vergleich der Evolutionsmechanismen. Die 3 anderen Faktoren vergleiche ich nur, damit beim Vergleich gute Bedingungen herrschen. Die eine der zwei Evolutionsprinzipien, die ich vergleiche, ist von der Natur inspiriert und die andere ist die in Machine Learning herkömmliche Methode.

2.2 Natur-Evolution und Computer-Evolution

Die Begriffe Natur- und Computer-Evolution sind keine allgemein bekannte Fachbegriffe, sondern zwei von mir gewählte Ausdrücke. In der Folge verwende ich diese beiden Ausdrücke für zwei verschiedene Mechanismen der Evolution. Wie das Wort "Natur-Evolution" impliziert, habe ich diese Methode von Vorgängen unserer Natur abgeschaut und werde sie anhand der menschlichen Fortpflanzung erklären. Der Mensch hat 46 Chromosomen, 23 von der Mutter und 23 vom Vater, das heisst er hat jedes Chromosom doppelt. Bei jedem einzelnen Gen setzt sich das stärkere durch, es ist das dominante. Unabhängig davon, ob bei einem Chromosom mehrheitlich die Gene des Vaters oder der Mutter benutzt wurden, ist die Chance beider Chromosome aller 23 Paare gleich gross, ans Kind weitergegeben zu werden.

Bei der "Computer-Evolution" hingegen wird bei der "Zeugung" jedes Gen entweder von der Mutter oder vom Vaters vererbt, dh sie arbeitete nur mit 23 Chromosomen, die jeweils aus Genen beider Elternteile bestehen. Somit verlieren die Chromosome ihre Bedeutung.

Im Folgenden erkläre ich diese Prinzipien im Detail mit einer bildlichen Darstellung.

2.2.1 Beispiele

Anhand der folgenden Beispiele zeige ich die Unterschiede der Systeme der Natur- und der Computer-Evolution detailliert auf.

Ein Chromosom mit einem Gen

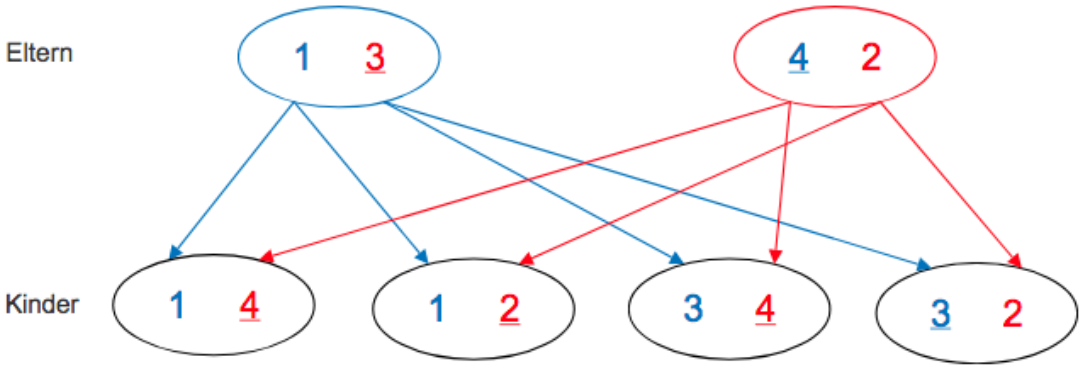


Figure 1: Natur-Evolution

X: ein Gen mit dem Wert X, X: Gen des Mutterchromosoms, X: Gen des Vaterchromosoms, X: dominantes Gen

Dies ist ein Beispiel von Crossover, das gemäss der Natur-Evolution funktioniert, mit Lebewesen, die ein Chromosomenpaar mit je einem Gen haben. Bei der Natur-Evolution sieht man, dass jedes Individuum zwei Exemplare von diesem Chromosom hat, das des Vaters und das der Mutter. Hier ist jeweils die grössere Zahl das dominante, in der Natur wäre es das, welches mehr Proteine, z.B. Pigmente, herstellt. Bei dieser Architektur sind jeweils 4 Möglichkeiten von Zweierpaaren möglich, für jedes Chromosom der Mutter zwei des Vaters. Aber nur drei verschiedene Gene können dominant werden, da die "1" nie dominant wird. Trotzdem haben alle vier dieser "eingenigen" Chromosomen die gleiche Wahrscheinlichkeit, an die dritte Generation weitergegeben zu werden.

Die Computer-Evolution funktioniert simpler. Jedes Individuum hat immer nur eine Version eines Gens in der eigenen Erbinformation gespeichert, so gibt es nur ein Auswahl zwischen zwei Genen und somit auch nur zwei Möglichkeiten von Kindern.

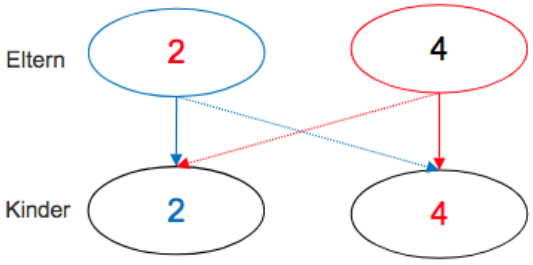


Figure 2: Computer-Evolution

X: ein Gen mit dem Wert X, X: Gen des Mutterchromosoms, X: Gen des Vaterchromosoms

Ein Chromosom mit zwei Genen

Um weitere Unterschiede zwischen Natur-Evolution und Computer-Evolution aufzuzeigen, folgt nun ein etwas kompliziertes Beispiel von Lebewesen mit einem Chromosom, das zwei Gene beinhält. Hier wird die Rolle der Chromosomen auch sichtbar. In den folgenden Abbildungen ist jeweils links des mittleren senkrechten Strichs das erste Gen und rechts das zweite Gen. Bei der Natur-Evolution sind die blauen oberen Gene das Vaterchromosom und die roten unteren das Mutterchromosom.

Es gilt zu beachten, dass die Gene, die im vorherigen Beispiel jeweils nebeneinder standen, nun übereinander dargestellt sind.

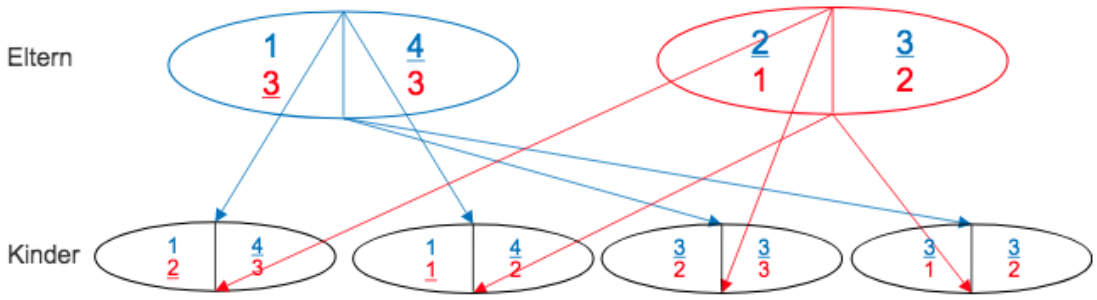


Figure 3: Natur-Evolution 2

X: ein Gen mit dem Wert X, X: Gen des Mutterchromosoms, X: Gen des Vaterchromosoms, X: dominantes Gen

Die Natur-Evolution bringt analog zum vorherigen Beispiel wieder 4 verschiedene Möglichkeiten hervor. Die Ursache dafür ist, dass die 50/50 Auswahl, welches Gen ein Elternteil seinem Kind mitgibt, sich auf die Chromosome bezieht. In diesem Beispiel entsprechen die zwei Gene einem einzigen Chromosom, vergrössert sich die Anzahl "Mischmöglichkeiten" nicht. Die Entscheidung, welche Gene dominant werden, findet im Kind statt, so können in Kindern, die verschiedene Chromosomen geerbt haben, trotzdem die gleichen Gene aktiv sein. In der vorigen Abbildung haben die zwei rechten Individuen das gleiche Chromosom vom Vater, aber nicht das gleiche von der Mutter geerbt. Sie haben exakt die gleichen dominanten Gene (3 & 3), obwohl sie nicht die gleiche Erbinformation haben.

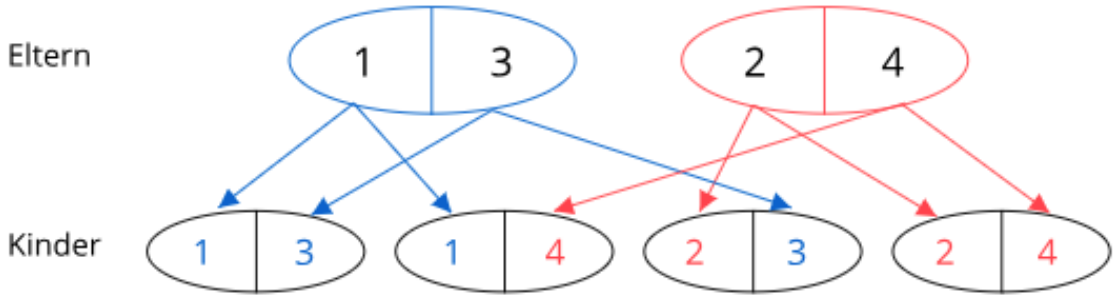


Figure 4: Computer-Evolution 2

X: ein Gen mit dem Wert X, X: Gen des Mutterchromosoms, X: Gen des Vaterchromosoms

Die Computer-Evolution hat ihre Anzahl an neuen DNA-Kombinationen im Gegensatz zur Natur-Evolution verdoppelt, weil es für jedes Gen exakt zwei Möglichkeiten gibt.

Folgerung

Daraus folgt, dass diese Kombinationsmöglichkeiten bei der Computer-Evolution von der Anzahl Gene abhängt und bei der Natur-Evolution von der Anzahl Chromosome. Die Kombinationsmöglichkeiten dieser beiden Mechanismen können mit den folgenden zwei Formeln ausgedrückt werden:

$k = \#Kombinationen$

$c = \#Chromosome$
$g = \#Gene$

$k_{Natur-Evolution} = 4^c$

$k_{Computer-Evolution} = 2^g$

Folglich gibt es bei uns Menschen 4^{23} verschiedene Kombinationen, die ein einzelnes Paar zeugen kann. Zu beachten ist dabei, dass sehr viele unserer Gene (fast) gleich sind und somit Überschneidungen vorkommen. Zudem wird die sichtbare Anzahl nochmals verkleinert, weil bei jeder Kombination jeweils das stärkere Gen benutzt wird und somit die dominante, also erkennbare Erbinformation weniger divers sein kann, als die geerbte DNA. Jedoch werden diese Informationen von dem Moment der Zeugung an von äusseren Faktoren beeinflusst, was zu (beinahe) unendlicher Diversität führt. Bei simulierter Evolution ist das nicht der Fall.

2.2.2 Hypothese

Wenn es mehr als doppelt so viele Gene wie Chromosome hat, was üblich ist, gibt es bei der Computer-Evolution weit mehr und somit auch mehr erfolgreiche Kombinationsmöglichkeiten. Daraus schliesse ich, dass die Computer-Evolution sehr schnell Fortschritte machen wird. Auch die Natur-Evolution hat ihre Vorteile, erstens ist doppelt so viel Erbinformation im Spiel, wenn die Gruppengrösse die gleiche ist und zweitens werden langfristig mehr verschiedene Gene im Umlauf sein. Ich denke, die Diversität spielt in der Evolution eine entscheidende Rolle. Deshalb stelle ich folgende Hypothese auf: Die durchschnitliche Fitness aller Schlangen der Natur-Evolution wird nach mehreren Generationen die der Computer-Evolution einholen und überholen.

Dies wäre eine befriedigende Bekräftigung der Evolutionsprinzipien unserer Natur. Dazu ist zu ergänzen, dass die Mechanismen der Computer-Evolution in der Natur gar nicht funktionieren würden, da die DNA eines Kindes nicht aus zwei Eltern-DNAs zusammengestellt werden kann wie mit einem Baukasten. Denn Spermium wie auch Eizelle müssen jeweils eine vollständige DNA beinhalten. Würden sie, unwissend welche Bauteile der Andere mitbringt, nur einzelne Bausteine mitbringen, könnten Leerstellen und ungewollte zweifache Ausführungen eines Elements vorkommen.

2.3 Genetischer Algorithmus

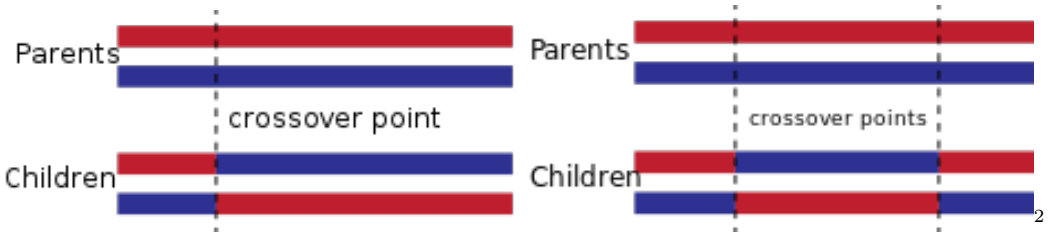
Ein genetischer Algorithmus, meist Genetic Algorithm genannt, besteht aus drei essentiellen Prozessen der Evolution, welche an einer Population[1] ausgeübt werden. Diese sind “natürliche Selektion”, “Crossover” und “Mutation”.

Die natürliche Selektion ist einer der heikelsten Schritte, da sie den grössten Einfluss haben kann, bzw. ihr Einfluss am meisten zwischen positiv und negativ schwanken kann. Denn in der natürlichen Selektion muss definiert werden, was das Ziel eines genetischen Algorithmus ist, dh. es muss eine gewisse Fitness verteilt werden können, der die Leistung eines Individuums darstellt. Hier wird der grösste Unterschied zur realen Evolution ersichtlich, während sich dort eine Population konstant fortpflanzt und Nachwuchs kriegt, wird beim normalen genetischen Algorithmus eine Generation in einem Moment durch eine neue ersetzt. In der echten natürlichen Selektion ist es so, dass die, die länger leben, tendenziell mehr Kinder kriegen und somit das bessere Genmaterial häufiger wird. Beim normalen Genetischen Algorithmus versucht jedoch jedes Individuum die Aufgabe zu bewältigen und die Wahrscheinlichkeit, ein Kind zu zeugen, ist jeweils proportional zur Fitness eines Individuums. Das heisst einer, der zehnmal so gut war wie ein anderer hat auch eine zehnmal so grosse Wahrscheinlichkeit, bei der Zeugung des Nachwuchses mitzuwirken. *Und*

[1] Gruppe von Individuen einer Spezie, die untereinander die Möglichkeit zu Fortpflanzung haben. https://de.wikipedia.org/wiki/Population_(Biologie), aufgerufen am 21.10.2018

jedes Mal, wenn diese Fortpflanzung im genetischen Algorithmus stattfindet, wird danach die ganze Elterngeneration mit der neuen Generation ausgewechselt. Zusammenfassend muss die Funktion der natürlichen Selektion, die solche Scores berechnet, wirklich aufs Ziel hinsteuern, was je nach Aufgabenstellung verschieden schwierig ist.

Das Crossover hingegen ist zwar sehr grundlegend, ihre Art aber meistens weniger entscheidend, was das Endresultat angeht. Zum einen gibt es die Möglichkeit beim DNA-strang eine, zwei oder mehr Schnittstellen zu wählen und jeweils die Informationen links einer Schnittstelle vom einen, die rechts vom anderen Elternteil zu nehmen. Diese Schnittstellen sind an zufälligen Punkten im Strang



[2]

Die andere verbreitete Methode, die auch die von mir verwendete sein wird, wird uniform crossover genannt. Bei ihr wird, wie ich schon bei der Computer-Evolution erklärt habe, bei jedem einzelnen Gen zwischen den Elternteilen entschieden. Dabei ist die Wahrscheinlichkeit in den meisten Fällen je 50%.

Der letzte sehr wichtige Schritt ist die Mutation, bei jedem neuen Nachwuchs gibt es eine gewisse Wahrscheinlichkeit, dass ein Gen zufällig mutiert. Diese Wahrscheinlichkeit wird oft auch mutation rate genannt. Wenn sie zu klein ist, wird kaum neues Genmaterial geschaffen und die Population wird sehr schnell zu kleiner Diversität streben. Ausserdem, kann es möglicherweise sein, dass in einer ersten Generation schlicht nicht dass gesuchte Genmaterial vorhanden ist, dann nützt noch so viel Crossover und natürliche Selektion nichts, um an diese zu gelangen. Durch Mutation kann aber immer neue Erbinformation geschaffen werden. Andererseits ist es auch nicht wünschenswert, dass die Mutationsrate zu gross ist, weil dann alle Fortschritte der natürlichen Selektion und des Crossovers immer wieder durch all die Mutationen zunichte gemacht werden.

Deshalb ist es wichtig vorallem die natürliche Selektion und die Mutationsrate im Gleichgewicht zu halten. Eine zu exponentielle Score-Funktion der natürlichen Selektion würde das in einer Population vorhandene Genmaterial sehr schnell dezimieren, bis nur noch alle Individuen fast die gleiche Erbinformation haben. Eine kleine Mutationsrate würde das sogar noch unterstützen. Im anderen Extrem wäre die Score-Funktion sehr flach und der Unterschied der Scores eines schlechten und eines guten Individuumswäre so gering, dass es die natürliche Selektion sozusagen nicht stattfinden würde. Oder wie gesagt, könnte eine zu grosse Mutationsrate den Fortschritt zwischen zwei Generationen jedes Mal rückgängig machen. Mit diesen und mehr Faktoren und mehr werde ich im nächsten Kapitel auch noch etwas experimentieren, um eine möglichst gute,Ausgangslage für den Vergleich zu schaffen.

[2] https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm) aufgerufen am 21.10.2018

2.3.1 Beispiel Shakespeare-Affen

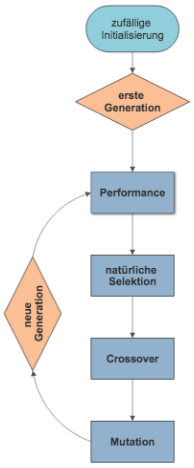


Figure 5: genetischer Algorithmus

Bei diesem Beispiel gibt es ein ganz klares Ziel: Schreibe den Satz "to be or not to be that is the question" Die Metapher handelt von ganz vielem Affen, die auf Schreibmaschinen rumtippen. Die Frage ist, wie lange es dauert, bis ein Affe zufälligerweise den Shakespeare-Satz schreibt. Um einen Satz, der 39 Zeichen lang ist, zu schreiben gibt es $27^{39} = 27392739$ Möglichkeiten. Bei jedem Zeichen gibt es, 26 Möglichkeiten für die Buchstaben und noch eine für einen Abstand. Das heisst es ginge ewig. Doch mit einem genetischen Algorithmus geht das viel schneller. Man initialisiert eine Population von beispielsweise 1000 Shakespeare-Affen, ihre DNA sind 39 Zeichen lange Listen. In diesem Beispiel gibt es keine wirkliche Performance die ihr Können anders als die DNA widerspiegelt. Man könnte aber sagen, die Performance ist das Tippen auf der Schreibmaschine. Wichtig ist, dass ein einzelner Affe jedes Mal das Gleiche schreiben würde, da es in seiner DNA geschrieben steht. Dann folgt die natürliche Selektion, bzw die Verteilung von Scores an alle Affen. Hier könnte man schlicht die Anzahl richtiger Buchstaben nehmen.

Hier ein Beispiel:
AffeXY: "ocxjgnfx dzjoscjfgcvizdtplgwro twtodmnf"
Lösung: "to be or not to be that is the question"

Der Score ist 2, da an der neunten und and der neuntletzten Stelle ein Abstand ist. Alle anderen Zeichen sind falsch. Es folgt die eigentliche natürliche Selektion, bei der jeweils zwei Eltern ausgewählt werden, bis die neue Generation die gleiche Grösse hat wie die alte. Die Wahrscheinlichkeit, das ein Affe ausgewählt wird, um ein Kind zu zeugen, ist proportional zu seinem Score.

Mathematisch sieht das folgendermassen aus:

a = Liste aller Affen, a_x = ein Affe aus der Liste an Stelle x (0-basiert),
S_{a_x} = Score des Affens a_x und
P_{a_x} = Wahrscheinlichkeit des Affens a_x, für die Fortpflanzung ausgewählt zu werden
$P_{a_x} = \frac{S_{a_x}}{\sum_{i=0}^{999} S_{a_i}}$

In Python:

```
def probability(x):
    s = sum([monkey.score for monkey in monkeys])
    return monkeys[x]/s
```

Dann folgt das Crossover von jeweils zwei selektierten Geschlechtern. Wichtig! Hier gibt es keine Geschlechter. Hier wird uniform crossover benutzt:

```
parent1:   "rbnfemtfqgrbe  ygdhuxywvyjvqecizhhfwwdz"
parent2:   "ankixhhognzqxfverrh fvq caqgnidbrdy izo"
child:     "ankfxhhfqnzqxf yrrhuxyq yavqnidbhhywwzo"
```

Zum Schluss wird bei allen entstandenen neuen Affen noch mutiert, was in der echten Welt etwas grausam tönt, ist ganz harmlos. Bei einer Wahrscheinlichkeit von beispielsweise 1% wird ein Buchstabe in der DNA zufällig ausgewechselt, dh in unserem Beispiel würde es sogar nur bei 2 von 5 Affen eine Mutation geben.
child before mutation: "ankfxhhfqnzqxf yrrhuxyq yavqnidbhhywwzo"
child after mutation: "ankfxhhfqnzqxf yrrhuxyq pavqnidbhhywwzo"

Somit entsteht eine neue Generation, der wieder verschiedene Scores zugeteilt werden und die dann eine 3. Generation macht. So geht es weiter und weiter. Bei diesem Beispiel kann man

aufhören sobald die Lösung gefunden worden ist, da es eine richtige Lösung gibt, sonst macht man es einfach bis zufriedenstellende Resultate aufkommen.

2.4 Neuronale Netzwerke

Neuronale Netzwerke sind etwas sehr sehr Grundlegendes in der Welt von Machine Learning. Es ist eine Methode um mit einer gewissen Anzahl Inputs (Liste von Zahlen) zu rechnen und daraus eine gewisse Anzahl von Outputs zu machen. Ein einfaches Beispiel wäre, anhand von Länge und Breite der Blütenblätter die Blumensorte zu bestimmen. So ein Neuronales Netzwerk besteht jeweils aus mehreren Layers, die alle aus einigen Neuronen bestehen. Die Neuronen einer Layer sind durch Synapsen mit den Neuronen der nächsten Layer verbunden. Dies ist von unserem Gehirn inspiriert, welches auch mit Neuronen, die über Synapsen verbunden sind, funktioniert, nur ohne Layers und viel komplizierter.

Ersichtlicher wird das mit zwei Beispielen von Neuronalen Netzwerken, welche Gleichheit erkennen sollten. Sie haben zwei Inputs, je eine 1 oder eine 0 und haben die Aufgabe, 1 ausrechnen, wenn beide Inputs gleich sind und 0 wenn sie verschieden sind. Zuerst zeige ich ein Beispiel eines sehr gut funktionierenden Netzwerks dann eines mit zufällig initialistierten Werten. Anhand dieser beiden Beispiele werde ich die inneren Prozesse eines Neuronalen Netzwerkes erklären.

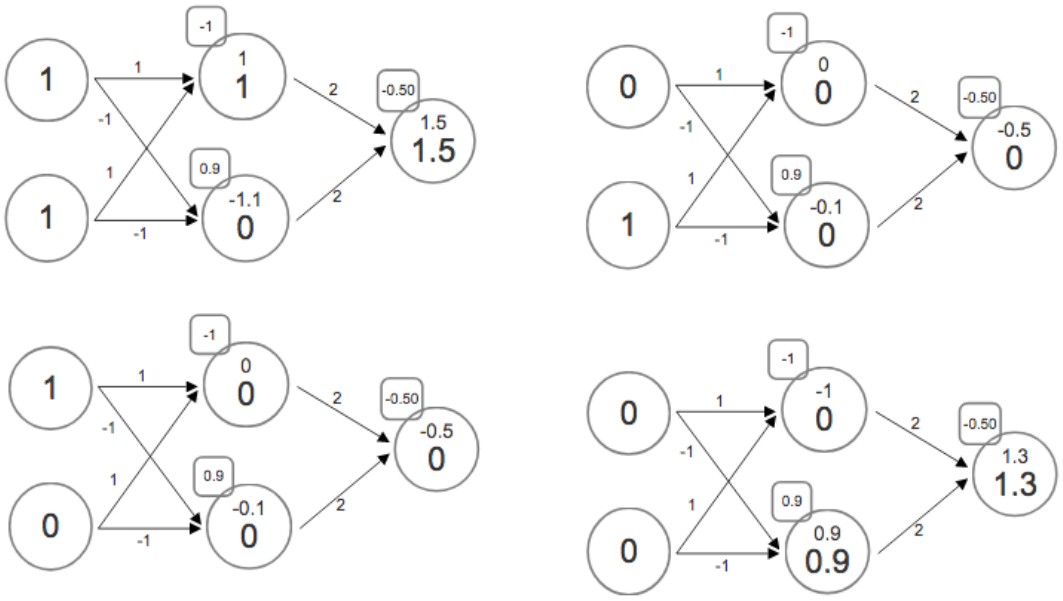


Figure 6: funktionierendes Gleichheits-NN

Jedes Input-Neuron hat eine Verbindung zu allen Neuronen der nächsten Layer, die "hidden layer", diese sind mit einem Gewicht versehen. Das Zwischen Resultat eines Neurons der Hidden Layer ist die Summe aller Inputs jeweils mit ihrem Gewicht multipliziert plus die "bias" des Neurons, die oben links zu sehen ist. An diesem Resultat wird noch eine sogenannte "activation function" angewendtet. Oftmals ist das eine Funktion wie Sigmoid oder aber eine nicht stetige wie "Rectified Linear Units" oder ReLU. Hier wird ReLU benutzt, welche eigentlich einfach negative Zahlen in Nullen umwandelt und positive so lässt. Activation-Functions sind sehr wichtig da sie die Linearität der Rechnungen beseitigen, ohne sie wären alles nur lineare Operationen (Multip-

likation und Addition), doch ohne Linearität können komplexere Verhältnisse ausgedrückt werden. Die Resultate der Hidden-Layer werden dann wieder mit Gewichten multipliziert und zu einer Bias addiert. Das geht noch einmal durch die ReLU-Funktion und schon hat man den Output des neuronalen Netzwerkes.

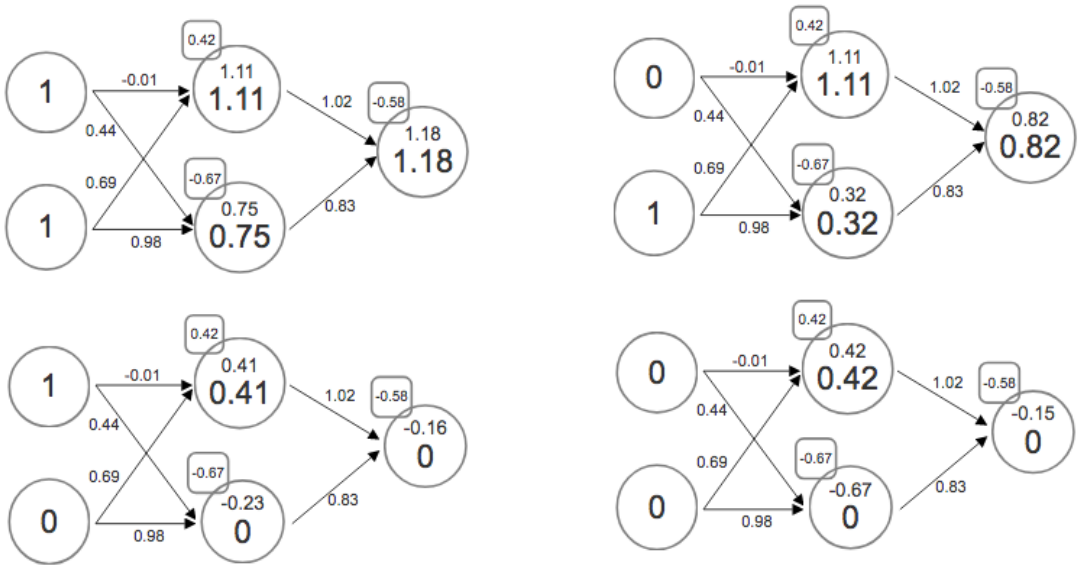


Figure 7: Zufällig initialisiertes, nicht funktionierendes Gleichheits-NN

Das ist der Rechenweg für das Zwischenresultat des oberen Hidden-Layer-Neuron des zufällig initialisierten Netzwerks beim Inputs-Vektor $\begin{pmatrix} 1 & 1 \end{pmatrix}$:

$$ReLU(1 * -0.0053 + 1 * 0.6943 + 0.4179) = 1.1069$$

Doch für das ganze Netzwerk schreibt man lieber in Matrix-Multiplikation:

$$ReLU(input * weights_{ih} + bias_h) = hiddenOutput$$
$$ReLU(\begin{pmatrix} 1 & 1 \end{pmatrix} * \begin{pmatrix} -0.0053 & 0.4379 \\ 0.6943 & 0.9846 \end{pmatrix} + \begin{pmatrix} 0.4179 & -0.6687 \end{pmatrix}) = \begin{pmatrix} 1.1122 & 0.3159 \end{pmatrix}$$
$$ReLU(hiddenOutput * weights_{ho} + bias_o) = Output$$
$$ReLU(\begin{pmatrix} 1.1122 & 0.3159 \end{pmatrix} * \begin{pmatrix} 1.0212 \\ 0.8301 \end{pmatrix} + \begin{pmatrix} -0.5795 \end{pmatrix}) = \begin{pmatrix} 1.1766 \end{pmatrix}$$

Sehr spannend ist es ausserdem, wenn man den Denkmechanismus solcher Netzwerke genauer anschaut. Meistens ist es äusserst komplex und nicht zu verstehen, doch beim *and*-Problem kann man die einzelnen Schritte erkennen. Wenn man sich das funktionierende Netzwerk anschaut, sieht man, das obere Hidden-Neuron gibt nur einen positiven Output, wenn der Input $\begin{pmatrix} 1 & 1 \end{pmatrix}$ ist. Das heisst, es hält sozusagen Ausschau nach dem 1 1 Muster. Das untere Hidden-Neuron reagiert analog nur auf den $\begin{pmatrix} 0 & 0 \end{pmatrix}$ Input, sonst ist es 0. Der Outputput zeigt dann lediglich an, ob eins der Hidden-Neuronen ein Signal gibt. So simpel ist das *and*-Problem gelöst.

Neuronale Netzwerke sind für Machine Learning deshalb so wichtig, weil sehr, sehr komplexe Verhältnisse zwischen Input-Vektoren und Output-Vektoren verkörpern können. Denn die Anzahl der Layers wie auch die Anzahl Neuronen in den Layers können unglaubliche Dimensionen annehmen. Das passiert dann vorallem im Bereich von Machine Learning, der Deep Learning

genannt, der heisst so, weil die Netzwerke so "tief" sind, dh unzählige grosse Hidden-Layers haben. Deep Neural Networks, werden zum Beispiel bei der Bildererkennung benutzt. Doch zu beachten ist, dass Neuronale Netzwerke allein noch nichts mit dem Prozess des Lernens zu tun haben. Sie sind lediglich komplizierte Funktionen. Um zu lernen, können dann verschiedene Techniken benutzt werden, die Gewichte und die Biases gezielt verändern.

2.5 Prozess und Entwicklung

Dieses Kapitel geht darum wie ich mein Snake-Projekt aufgebaut und so weit programmiert habe bis es für den schlussendlichen Vergleich bereit war. Es ist das einzige, in welchem ich etwas genauer auf meinen Code eingehe. Zuerst erläutere ich einzelne Teile der Programmierung des eigentlichen Games, dann schauen wir uns das Neuronale Netzwerk der Schlangen an, welche selbst spielen, dh die, die dann Bestandteil des genetischen Algorithmus sein werden. Anschliessend, zeige ich, wie einige der Funktionen des genetischen Algorithmus in JavaScript, einer webbasierten Programmiersprache aussehen. Zum Schluss analysiere ich noch verschiedene Einstellungen, wie die Mutationsrate und verschiedene Arten der natürlichen Selektion, um das Projekt für den Vergleich der Natur- und der Computer-Evolution mäglichst schön vorzubereiten.

2.5.1 Code Beispiele

Zuallererst musste ich das Spiel Snake programmieren. Um diesen Prozess zu veranschaulichen, zeige ich am besten die `action()` Funktion einer Schlange, welche aber noch nicht "intelligent"ist, und erkläre ihn. Text nach "`//`" und zwischen "`/*`" und "`*/`" ist jeweils ein Kommentar, dh er wird nicht ausgeführt:

```
class Snake {
    /*
    Das ist eine sogenannte Klasse, also eine Art eines Objekts,
    die alle Informationen eines Snake Games enthalt.
    Wichtige Variabeln sind:
        this.pos, die Position des Kopfes der Schlange.
        this.gameover, ob die Schlange tot ist.
        this.dir, die Richtung, in welche die Schlange gerade geht.
        this.growing, wie viele Blöcke die Schlange noch zu wachsen hat.
        this.tail, die Positionen aller Blöcke des Schwanzes

    ihre Funktionen sind:
        this.show(x, y, w, h), sie zeigt das Spiel an der Position x, y
            mit Breite w und Höhe h an.
        this.changeDir(dir), sie ändert die Richtung der Schlange zu "dir".
        this.action(), die Funktion die jeden "step" benutzt wird, folgt gleich.
        this.updateTail(), sie zieht den Schwanz der Schlange nach
        this.newFruit(), generiert eine neue Frucht an einer leeren Stelle
        this.isWall(x, y), ob der Block an der Stelle (x, y) eine Wand ist.
        this.isSnake(x, y), ob der Block an der Stelle (x, y) zur Schlange gehört.
        this.isFruit(x, y), ob der Block an der Stelle (x, y) eine Frucht ist.
    */

    ... // Definition anderer Funktionen

    action() {
        /*
        Diese Funktion wird jeden Step ausgeführt, sie bewegt die Schlange,
        prüft, ob sie gestorben ist und lässt die Schlange Früchte essen.
        */
```

```
        if (!this.gameover) { // "!" bedeutet "nicht"
            // Folgendes wird nur ausgeführt, wenn die Schlange am Leben ist.

            this.pos.add(this.dir); // Position der Schlange wird aktualisiert

            // Überprüfung, ob die Schlange gestorben ist:
            // Nur, wenn der Block, an welchem der Kopf nun ist,
            // zuvor Wand oder Teil der Schlange war.
            if (this.isSnake(this.pos.x, this.pos.y)
                || this.isWall(this.pos.x, this.pos.y)) {
                // "||" bedeutet "oder"
                this.gameover = true; // Schlange stirbt hiermit offiziell!
                return; // Funktion abbrechen.
            }

            // Die Schlange hat "uberlebt

            // Schwanz wird "nachgezogen",
            // also jeder Block rutscht eins nach vorn.
            this.updateTail();
            // Falls die Schlange noch am wachsen (this.growing > 0) ist,
            // wächst sie in updateTail().

            // Da sie jetzt gewachsen ist, falls nötig,
            // hat sie nun nur noch eins weniger zu wachsen.
            if (this.growing > 0) this.growing--;

            // Falls die Schlange an die Stelle der Frucht kam
            if (this.isFruit(this.pos.x, this.pos.y)) {
                // Nun soll sie 4 Blöcke wachsen.
                this.growing += 4;
                // Und eine neue Frucht soll zufällig generiert werden.
                this.newFruit();
            }


        }

    }

    ... // Definition anderer Funktionen
}
```

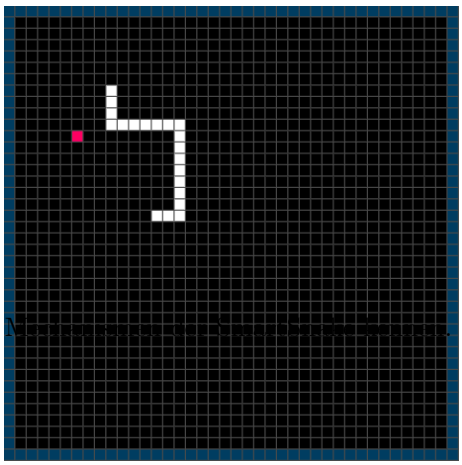


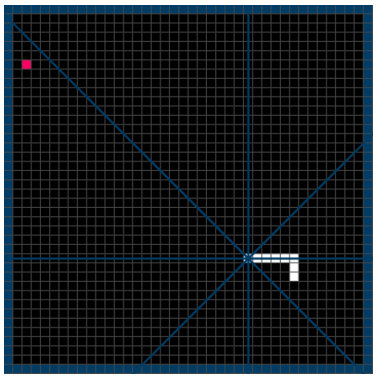
Figure 8: `show(x, y, w, h)` der Klasse `Snake`

Damit man das auch noch bildlich sieht, ist hier ein Resultat der `this.show(x, y, w, h)` Funktion:

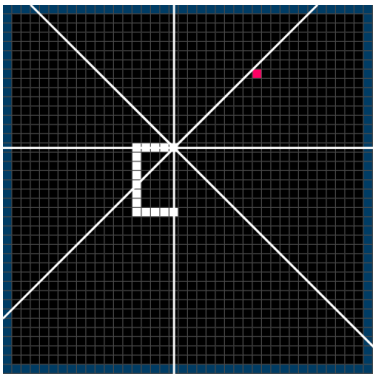
Doch das ist nur eine Schlange, die von uns Menschen gespielt werden kann. Wie sieht es aus, wenn man der Schlange die Kontrolle überlässt? Dafür müssen wir uns die `think()` Funktion der `SmartSnake()` Klasse, welche eine Unterklasse der `Snake` Klasse ist, genauer anschauen. Doch für das müssen wir zuerst noch andere

2.5.2 Struktur des neuronalen Netzwerks

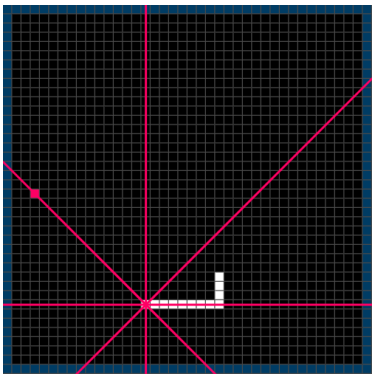
Die SmartSnake besitzt als DNA die Werte eines Neuronalen Netzwerks. Dieses Netzwerk hat die Struktur (24, 16, 4), dh sie hat 24 Inputs, eine Hidden-Layer mit 16 Neuronen und 4 Outputs. Die vier Outputs sind Werte, die jeweils den Drang der Schlange, in eine bestimmte Richtung zu gehen, numerisch darstellen. Und zwar in folgender Reihen folge: $\begin{pmatrix} up & right & down & left \end{pmatrix}$. Wäre der Output also $\begin{pmatrix} 1 & 0 & -1 & 2 \end{pmatrix}$, würde die Schlange somit ausrechnen, am liebsten in die letzte Richtung, also nach links, zu gehen. Die 24 Inputs bestehen aus 3 Achtern-Sets, jedes Achter-Set besteht aus acht Richtungen, in welche die Schlange sehen kann, alle 45°. Sie sieht Wandblöcke, Schlangenblöcke und ob eine Frucht zu sehen ist. Diesen drei "Sichtarten" entsprechen die Achter-Sets. Bei der Wand- und der Schlangensicht ist der Input jeweils 1 / Distanz, sodass die nächsten Wände und Schlangenblöcke die grössten Werte abgeben. Bei der Fruchtsicht ist der Input immer 0, ausser eine Frucht ist in Sicht, dann ist er 1. Die Achter-Sets gehen jeweils von Richtung zwölf um den Uhrzeigersinn alle Richtungen ab.



Bei dieser Situation ist der vierte Input der grösste, weil die Wand nach unten am nächsten ist. Es folgt rechts (dritter Input), da die rechte Wand nur ein Block weiter entfernt ist als die untere. Unten links und unten rechts haben den gleichen Wert.



Bei der Schlangensicht sind in diesem Fall alle Inputs ausser links, unten links und unten, gleich Null. Der Input der linken Richtung ist 1, da schon der erste Block ein Schlangenblock ist.



Ausser dem Input von oben links sind alle Null, da nur oben links eine Frucht in Sicht ist.

2.5.3 2. Code Beispiele

```
class SmartSnake extends Snake {

    ... // andere Funktionen wie getInput()

    think() {
        // In der getInput() Funktion werden die oben genannten 24 Inputs berechnet.
        let input = this.getInput();

        // Der eigentliche "think" Prozess, nämlich der Berechnung
        // des Outputs des Neuronalen Netzwerks.
        // Das erste Chromosom der DNA (this.dna.chromosomes[0])
        // ist das Neuronale Netzwerk.
        let output = this.dna.chromosomes[0].predict(input);
```

```
        // Hier wird der Index des grössten Wertes gefunden
        let ind = output.findIndex((e) => e == max(output));

        // Dann folgt lediglich die Umformung dieses Indexes
        // in changeDir(dir) Befehle:
        switch (ind) {
            case 0:
                this.changeDir("up");
                break;

            case 1:
                this.changeDir("right");
                break;

            case 2:
                this.changeDir("down");
                break;

            case 3:
                this.changeDir("left");
                break;

            default:
                // Zu diesem Punkt kommt es nie.
                console.log("Error: No output max?");
                break;

        }
    }

    ... // andere Funktionen
}
```

Die Funktion predict(input) eines neuronalen Netzwerkes ist genau das, was im Kapitel "Neuronale Netzwerke" besprochen wird. Die Matrix der Gewichte zwischen Input- und Hidden-Layer wird mit dem Input-Vektor multipliziert, das Ergebnis wird zu den Hidden-Biases addiert und dessen Ergebnis wird von der Activation-Funktion umgerechnet. Dann wird der dabei entstandene Vektor der Länge 8 mit der Matrix der zweiten Gewichte multipliziert, zu der zweiten Bias addiert und nochmal aktiviert. Für die Richtigkeit des Resultats sind natürlich die Gewichte, weights, und die biases. Um zu lernen müssen genau dies auch verändert werden. Dies geschieht in meinem Beispiel dank des genetischen Algorithmus, der die guten Schlangen benützt, um eine neue Generation von Schlangen zu züchten, welche, dank besserer DNA, bzw besserer Netzwerke, durchschnittlich auch bessere Resultate erzielen. Wichtig ist dabei, den guten Schlangen klar höhere Chancen zu geben, aber trotzdem nicht nur mit den besten 5 Schlangen neue züchten, sodass all das wertvolle Erbmaterial im Nu verloren geht und nach wenigen Generationen alle Schlangen identische Netzwerke haben. Dies wird mit sogenannter fitnessproportionaler Selektion geschafft. Diese Selektion ist eine Methode, die eine Schlange mit einer, wie der Name sagt, zu ihrer Fitness proportionalen Chance auswählt, um Kinder machen zu dürfen. Die Fitness ist dabei der Score, von dem wir im Kapitel "genetische Algorithmen" gesprochen haben. Bei meinen Schlangen wird diese Fitness aus der Lebenszeit und der Länge einer Schlange berechnet. Eine Schlange die einen Score von 300 hat, wird also mit doppelter Wahrscheinlichkeit ausgewählt, um ihr Genmaterial an die nächste Generation weiterzugeben, als eine mit dem Score 150. Dieses Verfahren haben wir auch schon bei dein Shakespeare-Affen des Kapitels "genetische Algorithmen" benutzt. Und so sieht das in meinem Code aus:

```
class SmartSnake extends Snake {

class Population {
    /*
    Die Klasse Population besteht hauptsächlich aus einer Liste
    von Individuen, in diesem Fall Schlangen, welche Bestand eines
    genetischen Algorithmus sind. Mit ihr werden alle Prozesse eines
    GA wie Selektion, Crossover und Mutation ausgeführt.
        this.pop ist die Liste ihrer Individuen.
    */

    ... // andere Funktionen

    fitnessProportionalSelection(count) {
        /*
        Wenn die Funktion fitnessProportionalSelection benutzt wird,
        wurde die Fitness aller Schlangen schon berechnet. Der Output
        der Funktion ist eine Liste von selektierten Schlangen
        der Länge count. In dieser Liste können einzelne (sehr gute)
        Schlangen auch mehrmals vorkommen.
        */

        // Zuerst wird die Summe aller Fitnessen berechnet
        var sumFit = 0;
        for (var i = 0; i < this.size; i++) {
            sumFit += this.pop[i].fitness;
        }

        // Dann wird die ganze Population in absteigender Reihenfolge
        // nach der Fitness sortiert.
        this.pop.sort((a, b) => b.fitness - a.fitness);

        /*
        Als nächstes kriegt jede Schlange die neue Variable accFit,
        welche die Summe der Fitnessen aller besseren Schlangen und
        ihrer eigenen Fitness beträgt.
        */
        var accFit = 0;
        for (var i = 0; i < this.size; i++) {
            accFit += this.pop[i].fitness;
            this.pop[i].accFit = accFit;
        }

        /*
        Nun werden die Schlangen ausgewählt, die für das Crossover
        benutzt werden.
        */
        pool = []; // leere Liste pool (= Mating Pool)
        for (let _ = 0; _ < count; _++) {
            // Heisst so viel wie: "Folgenden Block count mal wiederholen!".

            // random() = zufällige Zahl zwischen 0 und 1
            rand = random() * sumFit;

            var i = 0;
            // Durch die Population gehen, bis die accFit einer
            // Schlange grösser als rand ist.
```

```
            while (this.pop[i].accFit < rand) {
                i += 1;
            }
            pool.push(this.pop[i]);
        }
        return pool; // pool hat jetzt die Länge count
    }

    ... // andere Funktionen
}
```

Der Prozess der Auswahl anhand der Hilfsvariable `accFit` kann sich auch bildlich vorgestellt werden. Alle Schlangen werden nach der Fitness in absteigender Reihenfolge sortiert. Jede Schlange hat nun einem Kiste mit einer Breite, die gleich ist wie ihre Fitness. Somit entspricht der Abstand des rechten Randes einer bestimmte "Schlangenkiste" zum linken Rand der linksten "Schlangenkiste" der `accFit` der betroffenen Schlange. Dann wird ein Dart auf die Kisten geworfen, dieser Dart trifft überall, von links bis rechts, mit der gleichen Wahrscheinlichkeit. Die Schlange der getroffenen Schlangenkiste wird ausgewählt. So simpel.

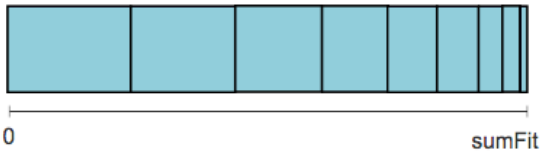


Figure 9: Darstellung zur Variable `accFit` und fitnessproportionaler Selektion

Das Crossover bespreche ich hier nicht, da es schon im Kapitel "Natur- und Computer-Evolution" genau behandelt wurde.

Die Mutation ist wiederum sehr einfach, bei jeder Schlange wird über alle Gene gegangen und bei einer Wahrscheinlichkeit von beispielsweise 1%, bzw der Mutationsrate, wird das Gen etwas verändert.

2.5.4 Vergleich der verschiedenen Faktoren

Wie ich im Kapitel "genetische Algorithmen" angesprochen habe, gibt es sehr viele Dinge, die die Effizienz eines genetischen Algorithmus beeinflussen können. Die einfachen dieser Faktoren sind die Mutation Rate und die Populationsgrösse. Grosse Populationsgrösse heisst automatisch eine grosse Auswahl an Erbmaterial und somit eine bessere Ausgangslage für die Evolution. Der einzige Nachteil einer sehr grossen Population ist die benötigte Rechenleistung. Denn normalerweise möchte man ja einen effizienten Algorithmus im Bezug auf die Zeit, nicht die Generationen. Doch unser Ziel ist weniger die Zeiteffizienz als die Generationeneffizienz, da es um die evolutionären Prozesse geht, die wir vergleichen wollen und diese in der echten Welt wichtiger ist. Ausserdem ist die Ziet, die es braucht Zufälle und Scores auszurechnen, in der echten Welt gleich null. Mutationrate ist etwas weniger eindimensional. Wird sie zu klein, tendiert die Population zur Kongruenz, also der Homogenisierung aller Individuen. Ist sie jedoch zu gross, wird jeglicher Fortschritt der natürlichen Selektion rückgängig gemacht, da zu viel DNA mutiert. Dann gibt es noch die Fitness-Funktion, die einen sehr grossen Einfluss auf die Evolution haben kann. Denn wenn die Funktion zu flach ist, dann erfüllt die fitnessproportionale Selektion ihren Sinn nicht mehr, da die besseren Schlangen kaum eine höhere Chance als der Rest haben. Ist sie wiederum zu exponentiell, kann es schnell vorkommen, dass eine ganze neue Generation nur aus einer handvoll Schlangen gezüchtet. Deshalb ist es ein ziemlich schwieriger Balanceakt, eine gute Fitness-Funktion für eine komplexe Sache wie die Leistung einer Schlange zu finden. Letztens spielt in meinem Projekt der Aufbau des neuronalen Netzwerkes eine grosse Rolle. Es kann zu wenig komplex sein, um die Situation angemessen zu einer Richtung zuzuteilen oder auch so gross, dass der Output fast nicht mehr vom ursprünglichen Input abhängt. Ausserdem führen grössere neuronale Netzwerke auch schneller zu Speicherproblemen. Ich vergleiche nun alle diese Faktoren, indem ich für jede Art Einstellung den genetischen Algorithmus 15 Generationen lasse.

Zuerst erkläre ich alle diese Einstellungen:

- Netzwerke:
  1. Net 1 (Spalten 1 - 3): Ein Netzwerk der Struktur (24, 8, 4) (24 Inputs, 8 Neuronen der Hidden Layer und 4 Ouputs)
  2. Net 2 (Spalten 4 - 6): Ein Netzwerk der Struktur (24, 8, 8, 4)
  3. Net 3 (Spalten 7 - 9): Ein Netzwerk der Struktur (24, 16, 4)
- Mutation Rates:
  1. 0.002mr (Spalten 1, 4, 7): Mutation Rate = 0.2 Prozent
  2. 0.01mr (Spalten 2, 5, 8): Mutation Rate = 1 Prozent
  3. 0.05mr (Spalten 3, 6, 9): Mutation Rate = 5 Prozent
- Populationsgrössen:
  1. 1000P (Zeilen 1 - 5): Grösse der Population = 1000

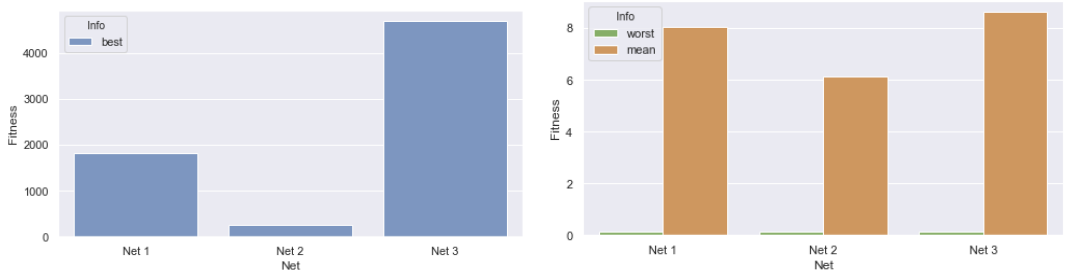
2. 5000P (Zeilen 6 - 10): Grösse der Population = 5000
3. 10000P (Zeilen 11 - 15): Grösse der Population = 10000

- Fitness-Funktionen:

1. 1cf (Zeilen 1, 6, 11): erste Fitness-Funktion (mehr später)
2. 2cf (Zeilen 2, 7, 12): zweite Fitness-Funktion
3. 3cf (Zeilen 3, 8, 13): dritte Fitness-Funktion
4. 4cf (Zeilen 4, 9, 14): vierte Fitness-Funktion
5. 5cf (Zeilen 5, 10, 15): fünfte Fitness-Funktion

Am Auffälligsten sind wahrscheinlich die verschiedenfarbigen und verschieden satten Hintergründe der Diagramme. Doch zuerst muss ich die eigentlichen Diagramme erklären, damit diese ebenfalls Sinn ergeben. Die X-Achse sind jeweils die 15 Generationen eines Versuches. Die Y-Achse hat die Einheit des Scores der Fitness-Funktion. Die blaue Linie zeigt jeweils die besten Scores einer einzelnen Generation. Die orange den Durchschnitt der Generation und die grüne den schlechtesten Score. Dementsprechend zeigt ein Diagramm den Verlauf der Leistungen einer Population über die Generationen. Diese Leistung wird anhand der Besten, des Schnitts sowie den Schlechtesten aufgezeigt. Die Farben dienen der Unterscheidung der Fitness-Funktionen, denn ein Score von 200 bei der zweiten Funktion ist nicht das gleiche wie ein Score von 200 bei der fünften Fitness-Funktion. Die Sättigkeit des Hintergründe zeigt das Verhältnis zwischen eigenem Maximalscore und dem Maximalscore aller Versüche dieser Fitness-Funktion. Die rötesten Diagramme sind also die Versüche mit den höchsten Scores aller Versüche auf Zeilen 1, 6 und 11.

Doch nun zur Analyse, gut ersichtlich ist, dass die mittleren 3 Spalten meistens weniger satt sind, als links und rechts. Folglich ist das Net 2, also die Struktur (24, 8, 8, 4) nicht so gut wie die beiden Anderen. Die Unterschiede zwischen denn verschiedenen Netzwerk-Strukturen sind mit einem etwas anderen Test noch besser ersichtlich. Ich lasse eine riesige Population von 80000 Schlangen zufällig initialisieren und das Game spielen und schaue dann direkt die Resultate an, ohne Evolution stattfinden zu lassen. Somit ist das Glück, dass beim Net 3 vielleicht einfach eine bessere Anfangspopulation vorhanden war, auszuschliessen.



Da die besten Fitnessen der Population so viel höher sind als der Schnitt, ist es besser die besten gesondert zu betrachten. Nun ist definitv zu erkennen, dass das Net 3 die anderen übertrifft, denn nicht nur der beste Score sondern auch der Schnitt ist dort jeweils am höchsten.

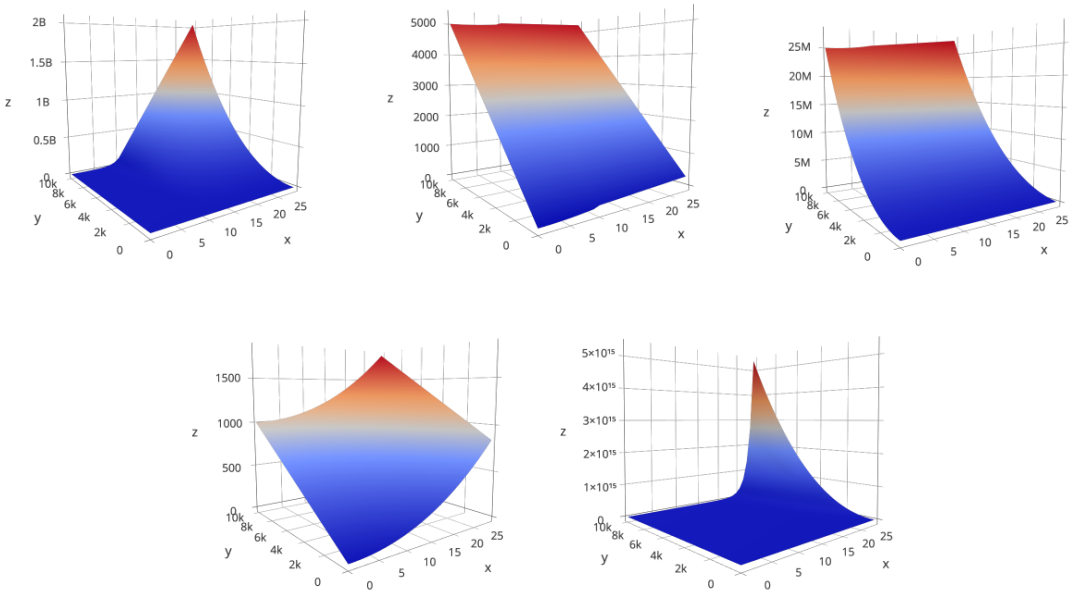
Um die Verschiedenheiten der verschiedenen Fitness-Funktionen zu verstehen, müssen wir uns zuerst die Funktionen anschauen:
pow(x, y) ist x^y.

```
def cf1(lifetime, length):
    if length < 10:
        return lifetime*lifetime*pow(2, length)/1000
    else:
        return lifetime*lifetime*pow(2,10)*(length-9)/1000
```

```
def cf2(lifetime, length):
    if length < 10:
        return lifetime * 0.5 + length * length
    else:
        return 9 * 9 + (length - 9) * 8 + 100 + (lifetime - 100) * 0.5
def cf3(lifetime, length):
    if length < 10:
        res = lifetime * 0.5 + length * length
    else:
        res = lifetime * 0.5 + 81 + (length-9) * 10
    return res * res
def cf4(lifetime, length):
    return lifetime * 0.1 + length * length

def cf5(lifetime, length):
    return (lifetime * lifetime * 0.1 * pow(2, length))
```

Wenn man das visualisiert, sieht das so aus:
Die X-Achse ist jeweils die Länge der Schlange, die Y-Achse die Lebenszeit und die Z-Achse die Fitness.



Die zweite und vierte Fitness-Funktion scheinen ziemlich flach zu sein. Dies sieht man auch bei den Evolutionsresultaten. Erstens ist im Laufe der Generationen kein Fortschritt erkennbar, was auf die mangelnde Überlegenheit der besseren Schlangen zurückzuführen ist. Dh die besseren Schlangen, die eigentlich ihr Erbmaterial vererben sollten, können es nicht genug vererben, da die Selektion sie kaum bevorzugt. Ausserdem führen alle Versuche zu sehr ähnlichen Resultaten.
Die fünfte Fitness-Funktion ist definitv zu exponentiell, da bei ihr jeweils vereinzelte Versuche (durch Glück) einen im Vergleich wahnsinnig hohen Score zustande bringen und dieser mit gleicher DNA, die offensichtlich vererbt worden sein muss, oft nicht nochmals möglich ist.
Das Problem der dritten Funktion ist hauptsächlich, dass es die Lebenszeit zu gewichtig zählt. Das führt zu Schlangen, die endlos im Kreis drehen. Und da ich für Trainingszwecke Schlangen nach einer zu langen Zeit, keine Frucht zu essen, sterben lasse, führt das zu keinen guten Resultaten. Man fragt sich dann vielleicht, wieso ich überhaupt die Lebenszeit in die Fitness miteinbeziehe, weil doch eigentlich nur die Länge zählt. Wenn ich sie nicht mit einbeziehen würde, würden die

anfangs so schlechten, zufällig initialisierten Schlangen alle eine Fitness von Null haben. Wenn aber doch einige eine Frucht gegessen haben, ist das zu einem grossen Teil pures Glück, weil die Frucht gerade auf dem Weg lag.
Nun bleibt noch die erste, bei ihr ist die Steigung grundsätzlich schon ziemlich passend. Wenn man die Evolutionsresultate anschaut, wird das auch bestätigt. Die Schlangen verbessern sich richtig, da die blauen Graphen oft kontinuierlich steigen. Das ist bei den anderen Fitness-Funktionen oft nicht der Fall. Ausserdem hat es diese vereinzelten Spitzen nicht ganz so oft wie etwa Funktion 3 und 5. Diese Spitzen würden darauf hindeuten, dass die Funktion zu exponentiell ist. Das scheint sie nicht zu sein.

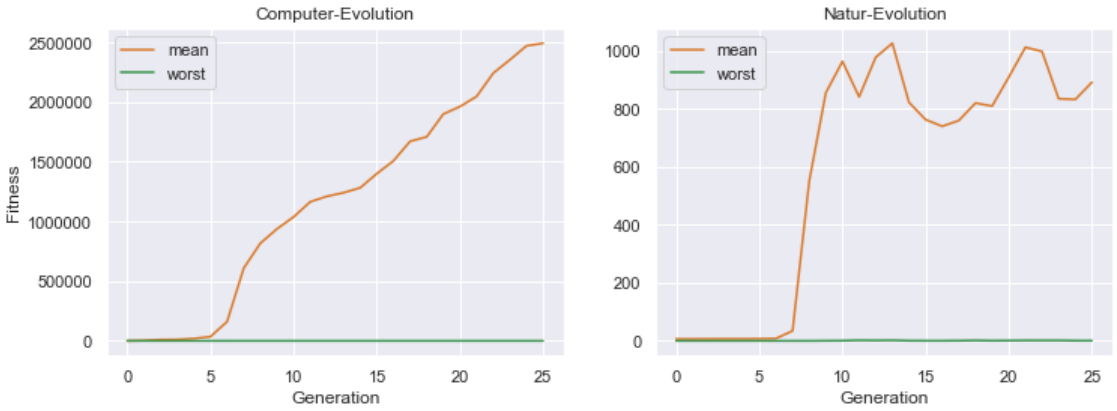
Bei den Mutationsraten sieht man vorallem in den letzten drei Spalten, wie die 5%ige die besten Resultate bringt. Doch in den Vergleichen werden auch relativ kleine Populationsgrössen benutzt. Deshalb werde ich beim Endexperiment trotzdem eine Mutationsrate von einem Prozent benutzen, da ich dann auch grössere Populationen benutzen werde und somit trotzdem genug Mutation vorhanden ist.

Offensichtlich ist eine Population von nur 1000 Schlangen zu klein, da dort klar die schlechtesten Resultate erzielt wurden. Zwischen 5000 und 10000 sieht man zwar fast keinen Unterschied, aber ich werde sowieso noch etwas mehr benutzen, da ich dann im Vergleich zwischen Natur-Evolution und Computer-Evolution nur zweimal Evolution laufen lassen muss und nicht wie in diesem Quervergleich 135 Mal für die 5 verschiedenen Fitness-Funktionen, 3 Populationsgrössen, 3 Mutationsraten und 3 Netzwerkstrukturen.

Zusammenfassend entscheide ich mich also für eine 1%ige Mutationsrate, die erste Fitness-Funktion, das Net 3 und eine 25000 grosse Population. 25000 nehme ich schlicht, weil es das Maximum ist, das noch keine Speicherprobleme mit sich bringt. Mit diesen Einstellungen denke ich, dass der Vergleich zwischen Natur-Evolution und Computer-Evolution gute Konditionen hat und zu den am besten brauchbaren Resultaten führen wird.

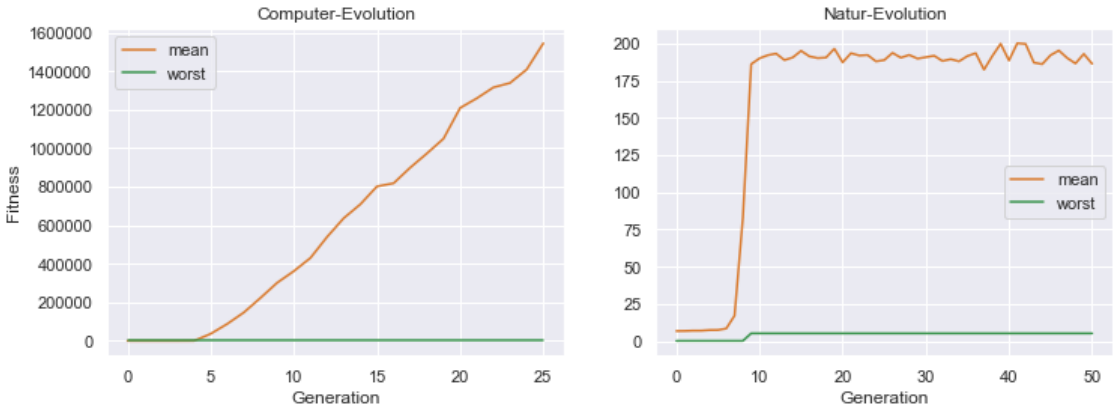
2.6 Vergleich Natur- und Computer-Evolution

In diesem Kapitel komme ich zum Endziel dieses Teils meiner Maturaarbeit. Der Vergleich zwischen der Natur-Evolution und der Computer-Evolution. Für das lasse ich den Genetischen Algorithmus zweimal laufen. Einmal haben die Schlangen die DNA-Architektur, die ich für die Natur-Evolution definiert habe, mit den doppelten Chromosomen und einem Crossover, das analog zur echten Welt funktioniert. Beim zweiten Mal ist dann alles wie bei einem herkömmlichen genetischen Algorithmus, also der Computer-Evolution.



Sehr spannende Resultate, welche genaue Auseinandersetzung benötigen, um verstanden zu werden. Zuerst rein objektive Beobachtungen: Die maximale Fitness, die die Computer-Evolution erreichte, ist 338 Millionen während die Natur-Evolution auf höchstens 5.1 Millionen kam. Die Computer-Evolution in Generation 7 66 Prozent der maximalen Fitness und die Natur-Evolution in Generation 10 82 Prozent ihres Rekords. Natur- wie auch Computer-Evolution steigerten ihre besten Scores nach den ersten Spitzen nur noch wenig. Der Durchschnitt erreichte bei der Computer-Evolution den Spitzenwert von knapp 2.5 Millionen in Generation 25. Schon in der dreizehnten Generation gipfelte der Durchschnitt der Natur-Evolution bei einer Fitness von 1026. Der Schnitt der Computer-Evolution verschlechterte sich kein einziges Mal. Die Natur-Evolution verschlechterte sich im Schnitt einige Male und machte ab der zehnten Generation mehr oder weniger keinen Fortschritt.

Offensichtlich hat die Natur-Evolution einiges schlechter abgeschnitten. Doch anhand weiterer Daten habe ich bemerkt, dass bei der Computer-Evolution etwas aussergewöhnliches passiert ist. In der nullten Generation, bzw. die die zufällig initialisiert wurde, gab es eine Schlange die für die zufällige Initialisierung wahnsinnig gut abgeschnitten hat. Sie hatte eine Fitness 412000, was extrem hoch ist. Ganz allein hat sie den Durchschnitt von 8.4 auf knapp 25 angehoben. Somit entstand eine seltsame erste gezüchtete Generation. 11000 Kopien der besten Schlange entstanden, da als Vater wie auch als Mutter diese ausgewählt wurde, ja das ist hier möglich. Dazu ist bei weiteren 11000 Schlangen nur ein Elternteil die gute, was nur noch 3000 Schlangen übrig lässt, die nicht von der Mega-Schlange abstammen. Spätestens in der zweiten und dritten gezüchteten Generation wird dieses "Geschlecht" die ganze Population übernommen haben. Deshalb habe ich das Experiment wiederholt. Da ich in meiner Hypothese an die langfristige Überlegenheit der Natur-Evolution glaubte, habe ich diese nun doppelt solange laufen lassen. Ziemlich ähnliche Resultate sind dabei herausgekommen:



Zuerst wieder zu den Zahlen: Dieses Mal erreichte die Computer-Evolution eine maximale Fitness von fast 180 Millionen, der Rekord der Natur-Evolution ist 16300. Im Gegensatz zum ersten Durchlauf sieht man jetzt besser die stetige Steigerung der Topscores der Computer-Evolution. Bei der Natur-Evolution ist die Maximal-Fitness wieder ab 10. Generation stehen geblieben, und zwar bei klar tieferen Resultaten als im ersten Durchlauf. Den höchsten Durchschnitt erreichte die Natur-Evolution diesmal eher am Schluss, nämlich in der 41. Generation, sie hatte in der 10. Generation jedoch schon 95% dessen und blieb danach wie gesagt stehen. Die Computer-Evolution hingegen hat sich wie beim ersten Mal nicht einmal im Schnitt verschlechtert, vielmehr hat sie sich stetig zu höheren Durchschnitten emporgearbeitet.

Meine Hypothese war, dass die Natur-Evolution nach anfänglicher Führung der Computer-Evolution schlussendlich bessere Resultate erreichte. Diese lag offensichtlich ziemlich gänzlich falsch, da ich das Potenzial der Computer-Evolution ziemlich unterschätzt habe und ich vielleicht zu viel Hoffnung auf unsere Natu gesetzt habe. Im Folgenden werde ich die unerwarteten Resultate erläutern.

Die Natur-Evolution hat doch einige Nachteile, denen ich mir anfangs nicht bewusst war. Zum Beispiel ist die dominante Erbinformation weniger divers. Wegen der Dominanz der stärkeren Genen (grössere Zahl) ist die Wahrscheinlichkeit, dass ein Gen grösser als 0.5 ist, nicht 50%, sondern 75%. Dies beeinflusst die Effizienz der neuronalen Netzwerke, denn wenn alle Gewichte und Biases ähnlicher sind, funktioniert der Denkprozess nicht so gut. Man braucht ja genau ziemlich verschiedene Gewichte um die Inputs unterschiedlich zu behandeln. Das kann man sich auch etwa so vorstellen wie der Unterschied zwischen einer schwarz-weissen Sicht und farbiger Sicht, was aber eher überspitzt ist.

Andererseits ist die Natur-Evolution auch insofern im Nachteil, wie die Leistung einer Schlange gar nicht unbedingt das symbolisiert, was sie vererben kann. Es kann also sein das sich die dominanten Gene des Chromosomenpaars einer bestimmten Schlange perfekt kombinieren. Wenn es nun Kinder macht, vererbt sie jedoch jeweils nur eines ihrer beiden Chromosome. Dieses könnte zusammen mit dem anderen Chromosom, welches das Kind erbt, ganz andere dominante Gene hervorbringen, die gar nicht gut funktionieren. Die Computer-Evolution ist natürlich auch davon abhängig, ob eine gute Paarung gemacht wird, die ein zufriedenstellendes Kind zeugt. Jedoch vererbt eine Schlange, die in der Computer-Evolution eine hohe Fitness hatte, mit Sicherheit auch die Gene, welche sie selbst benutzte. Dies führt zu einer grösseren Verlässlichkeit, das gute Schlangen auch wirklich guten Nachwuchs zeugen.

Im Vorhinein sagte ich, dass die Natur-Evolution von einer doppelt so grossen Menge Erbmaterial profitieren könnte. Doch offensichtlich geht es mehr um die richtige Kombination als, um die richtige Zahl bei einem bestimmten Gen. Die Natur-Evolution hat, was Kombinationspotenzial angeht, einen sehr grossen Nachteil. Wenn man sich die Mutation wegdenkt, hat die Natur-Evolution pro Chromosom genau doppelt so viele Exemplare, wie Schlangen in der Population sind. Weil es sich bei diesem Projekt, um eine DNA mit nur einem einzigen Chromosom, dem neuronalen

Netzwerk, handelt, hat es bei 25000 Schlangen, 50000 Chromosome mit der Information für je ein Netzwerk. Ein einzelnes Individuum hat jeweils zwei solche Exemplare als Chromosomenpaar, woraus sie die dominanten Gene wählt. Somit sind eigentlich $\binom{50000}{2} = 1249975000$ verschiedene Chromosomenpaare und somit Schlangen. Vernachlässigt ist hierbei wie gesagt Mutation und der Verlust grosser Teile an Information über die Generationen, sodass nie alle 1.2 Milliarden Chromosomenpaare realisiert werden. Die Anzahl dieser Möglichkeiten scheint vielleicht gross zu sein, doch wenn man sich die Computer-Evolution anschaut, sieht das ganz anders aus. Für jedes einzelnes Gen gibt es so viel Exemplare wie Schlangen. Dh eigentlich halb so viele wie bei der Natur-Evolution. Doch da bei der Computer-Evolution immer wieder neu Chromosome aus den Elternchromosomen gemischt werden und diese auch vererbt werden (Bei der Natur-Evolution wird auch aus Vater- und Mutterchromosom gemischt, doch sozusagen die Hälfte der Zutaten vererbt, anstatt dem Gemisch), gibt es für jedes Gen 25000 Varianten, die beliebig kombiniert werden können. Weil das Neuronale Netzwerk 468 Parameter, also alle Gewichte und Biases hat, gibt es somit 25000^{468} verschiedene Schlangen, die ohne jegliche Mutation gezüchtet werden könnten. Das ist $1.72e2058$. Die Natur-Evolution hatte $1.25e9$. Das ein riesiger Unterschied. Die Unabhängigkeit von den Chromosomen bringt diesen entscheidenden Vorteil. In meinem hatten die Schlangen zwar auch nur ein einziges Chromosom, was den Nachteil der Natur-Evolution noch verstärkt, doch in unserer Natur haben die meisten Lebewesen unter 100 Chromosomen und vereinzelt bis zu 1300 [3]. Das ist zwar klar mehr als in meinem Projekt, aber es sind natürlich auch viel mehr Gene vorhanden, beispielsweise beim Menschen zählt man zwischen 25 bis über 120 Tausend. Nur schon mit 20000 Genen, übersteigt das die Anzahl Chromosome genug, um bei der Computer-Evolution ein Vielfaches mehr verschiedene Kombinationen zu ermöglichen, als sich bei der Natur-Evolution je möglich ist.

[3](http://www.biologie-lexikon.de/lexikon/chromosomenzahl.php)

3 Fingeralphabet-Erkennung

3.1 Idee und Ziel

Mein zweites grosses Projekt, besteht daraus ein Endprodukt zu erarbeiten, welches auf künstlicher Intelligenz basiert. Ich werde ein sogenanntes Convolutional Neural Network, eine spezielle Form eines neuronalen Netzwerks, programmieren und dieses trainieren, Fotos von Händen in das Fingeralphabet der Deutschschweiz zu kategorisieren. Das Fingeralphabet ist ein Teil der Gebärdensprache und hat für jeden Buchstaben ein Zeichen, sowie für "SCH" und "CH". Mein Ziel ist es also eine künstliche Intelligenz zu erschaffen, die das Fingeralphabet versteht. Das ist ein sogenanntes Kategorisierungsproblem. Die Fotos müssen in 29 Kategorien eingeteilt werden, 26 Buchstaben, "SCH", "CH" und Nichts. Denn die KI muss ja auch wissen, wann gar kein Zeichen gezeigt wird. In diesem Teil, fokusiere ich mich weniger darauf alles für Laien verständlich zu machen, sondern benutze manchmal Fachbegriffe, ohne sie weiter zu erklären, und erwarte somit hin und wieder Vorwissen. Doch die groben Grundzüge werden auch interessierte Laien verstanden haben und wichtig ist vorallem, dass das Endprodukt funktioniert. Denn das ist dann benutzbar ohne jegliches Wissen über Machine Learning. Das Datenset, mit welchem meine künstliche Intelligenz trainiert wird, mache ich selbst. Ich drehe Videos von verschiedenen Händen, die jeweils solche Zeichen formen, mit verschiedenen Hintergründen. Ich berufe mich dabei auf die Vorlage von Sonos, dem schweizerischen Hörbehindertenverband. Die Videos teile ich dann in einzelne Fotos auf. Ich werde am Schluss von jeder Kategorie 3000 Bilder haben. Dh insgesamt 87000 Fotos mit dem Format 320x240. Das ist ein ziemlich kleines Format, aber es reicht um die Hände zu erkennen und grössere Formate würden auch mehr Rechenleistung verlangen. Ich denke, durch das Training einer solchen KI kann man ein sehr praktisches Endprodukt von grossem Nutzen erschaffen. Das Fingeralphabet wird zwar grundsätzlich nur benutzt, um zu buchstabieren und Wörter ohne eigenes Gebärdenzeichen auszudrücken, aber grundsätzlich sind einem Alphabet ja keine Grenzen gesetzt. Ausserdem wäre es ein Projekt einer ganz anderen Dimension, wenn man möchte, dass eine KI alle oder sehr viele Gebärdenzeichen erkennen kann.

3.2 Wie lernt ein neuronales Netzwerk?

Im ersten Teil meiner Maturaarbeit habe ich einen genetischen Algorithmus benutzt, um den Lerneffekt zu erzeugen. Dabei habe ich einfach gute neuronale Netzwerke rausgepickt und neu zusammengestellt, doch nie hat sich eine einzelne Schlange in ihrer Lebenszeit verbessert, alles ging jeweils über die nächste Generation. In den meisten Bereichen von Machine Learning wird das nicht so gehandhabt. Normalerweise werden verschiedene Formen des Gradient Descent benutzt. Gradient Descent ist eine Methode, die für jeden einzelnen Parameter, also die einzelnen Gewichte und Biases, eine Ableitung ausrechnet. Diese Ableitung ist das Verhältnis des jeweiligen Parameters und des Fehlers des Netzwerkes. Und wenn man dieses Verhältnis kennt, kann man den Parameter in eine Richtung ändern, in welcher der Fehler, also der Unterschied zwischen dem Output des Netzwerks und des erwünschten Outputs, kleiner wird. Wenn man diesen Gradient Descent, also das Umherschieben der Parameter, sehr oft wiederholt, nimmt das neuronale Netzwerk langsam die erwünschte Form an. Und so werde ich auch den nächsten Teil meiner Maturarbeit lernen lassen. Ich werde jedoch nicht auf die Details eingehen, da man sich mit diesen als Laie sehr intensiver auseinandersetzen müsste, um es zu verstehen.

3.3 CNN

Convolutional Neural Networks werden hauptsächlich benutzt, um Inputs in Form von Bildern zu verarbeiten, können aber auch an Audio-Inputs und eindimensionalen Inputs angewendet werden. Sie funktionieren eingermassen gleich wie traditionelle Neural Networks. Man gibt ihnen einen gewissen Input und sie geben wiederum Outputs einer bestimmten Anzahl zurück. Sie haben auch solche Layers wie ich beim Kapitel "Neuronale Netzwerke" zeigte, doch die ersten paar Layers sind

speziell. Es sind sogenannte Convolutional Layers, zu Deutsch etwa "faltende Schicht". Convolutional Layers sind sozusagen Filter die über ein Bild geschoben werden. Beziehungsweise besteht eine solche Layer meist aus mehreren Filtern und produziert somit mehrere geefilterte Versionen des Bildes. Ein einzelner Filter hat eine bestimmte Breite und Höhe und besteht aus einer Matrix aus Gewichten. Wenn nun ein Filter, oder auch Kernel, über das Bild gehalten wird, wird jeweils der Pixelwert mit dem Gewicht, das darüber liegt, multipliziert. All diese Produkte werden summiert und das ist dann der Wert des einen Pixel des "gefilterten" Bildes. An dem Resultat einer Convolutional Layer wird wieder eine Activation Function angewendet, analog zu normalen Neural Networks. Dann werden meist auch noch sogenannte Maxpooling Layers benutzt, welche eigentlich lediglich das Resultat zu kleineren Formaten skalieren. Diese 3 Schritte werden einige Male wiederholt. Dann werden die Pixelwerte des Resultat zu einer langen Kette aufgereiht. Diese Kette von Zahlen werden dann von sogenannten Fully Connected Layers, den ganz normalen, die wir im Kapitel "Neuronale Netzwerke" angeschaut haben, weiter verarbeitet. Also muss man sich das so vorstellen, dass ein Input-Bild durch ganz viele spezielle Filter geschickt werden und danach ein normales neuronales Netzwerk folgt.

3.4 Mein Projekt

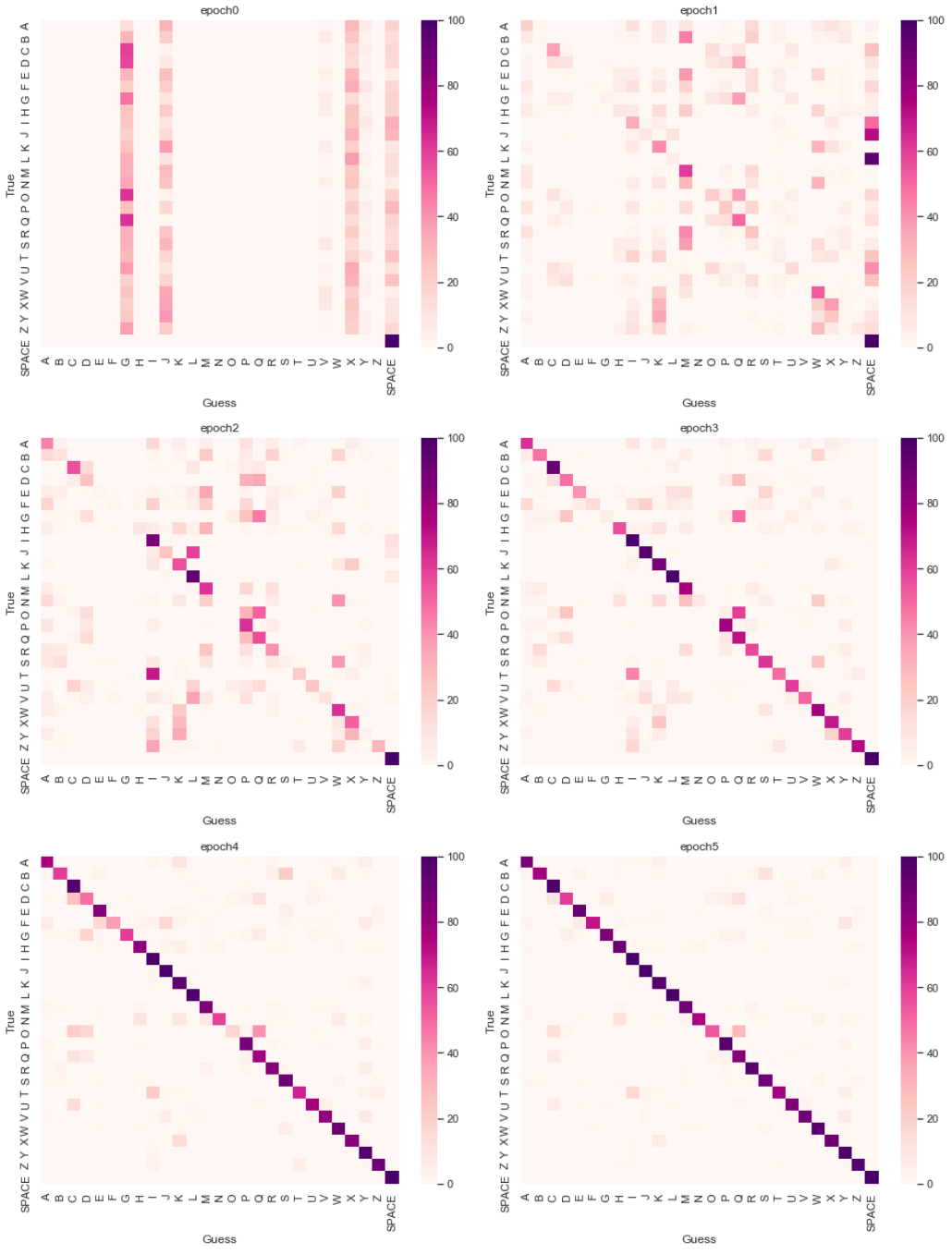
Für mein Bilderkennungsprojekt habe ich zuerst drei phD Studenten der ETH kontaktiert. Denn anfangs war noch unklar, was ich für ein Machine Learning Projekt machen möchte. Ich habe auf der Website Kaggle[4] viele verschiedene Datensets durchsucht. Irgendwann stiess ich auf ein Datenset mit Bildern der American Sign Language. Eine bildliche Sprache in das lateinische Alphabet umwandeln zu können, fand ich eine tolle Vorstellung. Darum habe ich mich dann für das entschieden. Die drei Studenten haben mir dann sehr wichtige Tipps zur Strukturierung des Arbeitsablaufs gegeben.

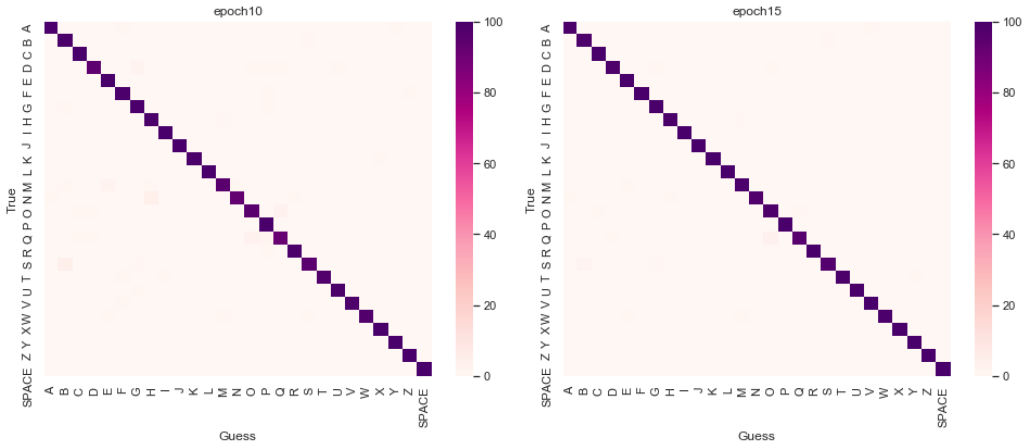
3.4.1 Schwarz-Weiss-Buchstaben

Zuerst machte ich einen Probelauf mit Bildern von lateinischen Buchstaben, welche zufällig gross und zufällig gedreht sind. Diese Bilder sind sehr klein mit einem Format von 32x32 Pixeln. Ich habe die Bilder selbst generiert mit einem simplen Programm. Das sieht beispielsweise so aus:

[4] www.kaggle.com

Dann programmierte ich mit der Machine Learning Bibliothek “pytorch”, der Tipp der ETH-Studenten, ein erstes Convolutional Neural Network. Da die Bilder sehr klein waren, war die gebrauchte Rechenleistung gering und ich konnte es sogar auf meinem eigenen Laptop trainieren. Ich habe pro Buchstaben 2000 Trainings-Bilder generiert und dann noch je 200 für Testzwecke. Denn sehr wichtig beim Training von künstlichen Intelligenzen ist, dass man noch einige Bilder vom Trainingsset separiert. Somit ist das Testen viel realitätsgetreuer, da sie auf unbekannte Bilder getestet werden. Ausserdem könnten die Neuronalen Netzwerke nach zu langem Training "zu gut" sein, auf englisch sagt man "overfitting". Dh sie können die Bilder aus dem Trainingsset so gut, dass sie die sozusagen auswendig können und nicht mehr auf andere Bilder anwendbar sind. Deshalb behält man immer einen Teil des Datensets beiseite, um sie besser testen zu können. Nun zu den Resultaten des CNNs, welches die Schwarz-Weiss-Buchstaben kategorisiert:


epoch0
epoch1
epoch2
epoch3
epoch4
epoch5
True
Guess
SPACE A B C D E F G H I J K L M N O P Q R S T U V W X Y Z SPACE
100
80
60
40
20
0



Auf der Zeile ist jeweils der Buchstabe, der getestet wurde, dh das Ziel für das Netzwerk. Die Summe einer Zeile ist immer Hundert. Denn in einer Zelle steht, wie viel Prozent der getesteten Bilder des Buchstabens der Zeile mit dem Buchstaben der Spalte beantwortet wurde.

Eine Epoche ist der Prozess einmal jedes Bild des Trainingsset zu lernen.

Wie man gut sehen kann, hat das anfangs zufällig initialisierte Neuronale Netzwerk eine Vorliebe für einige der Buchstaben. Egal, was getestet wird, antwortet es sehr häufig mit "G", da es einfach so "geboren" wurde. Doch während des Trainings wird es langsam verbessert und lernt verschiedene Antworten zu geben. Dabei sieht man sehr schön, wie sich die Farbe langsam bei der Diagonalen festigt, weil mehr und mehr die Tests pro Buchstabe 100%ig korrekt beantwortet werden. Von Epoche 10 bis 100 steigert sich die Präzision nur noch von 98.3% zu 99.8%. Das ist eher schlecht als recht. Denn wenn ein neuronales Netzwerk seine Trainingsdaten zu gut kennt, führt das zu dem vorhin genannten Overfitting. Dh es kennt die Bilder des Trainings sozusagen auswendig und ist nicht mehr auf ihm unbekannte Bilder anwendbar. Dies ist möglichst zu vermeiden.

3.4.2 Fingeralphabet

Nach einem erfolgreichen Testlauf mit den schwarzweissen Buchstaben, konnte ich mich der wahren Aufgabe zuwenden. Ursprünglich habe ich ein Datenset der American Sign Language auf Kaggle[5] gefunden. Dieses Datenset war jedoch sehr gleichmässig, dh die Handzeichen wurden stets vor exakt gleichem Hintergrund aufgenommen, nur nicht immer in der Mitte. Deshalb wollte ich selbst ein Datenset erstellen. Für das machte ich Videos des deutschschweizer Fingeralphabets mit meiner Familie. Ich benutzte also drei verschiedene Hände und drehte jeweils die Kamera beim Filmen, sodass sich der Hintergrund stetig etwas verändert. Ausserdem drehte ich an fünf Stellen mit unterschiedlichen Hintergründen. Würde man nur einmal durch das Alphabet gehen, kann ein sehr ungewollter Fehler passieren. Da jeder Buchstabe einige Minuten später gefilmt wird, ist das Sonnenlicht immer etwas schwächer und somit könnte die künstliche Intelligenz die Buchstaben einfach anhand der Helligkeit des Hintergrunds erkennen. Deshalb bin ich auch mehrmals und zu verschiedenen Tageszeiten durch das Alphabet gegangen und habe alle Zeichen gefilmt. Doch eigentlich wollte ich ja Fotos der Handzeichen. Da ich 3000 Bilder jedes Zeichens brauchte, war einzeln fotografieren eigentlich keine Option. Vielmehr habe ich jedes hundert Sekunden lang bei dreissig Bildern pro Sekunde gefilmt und die Resultate mit einem Python-Skript in Bilder des JPEG-Formats umgewandelt. Somit hatte ich mit verhältnismässig wenig Aufwand schnell mein Datenset. Mit 26 Buchstaben, einem SCH, einem CH und einer Kategorie, die nichts symbolisiert, hatte ich $29 * 3000 = 87000$ Bilder. Nun konnte ich mich dem Trainieren des CNN widmen.

Zuerst habe einige Änderungen bei der Struktur des Netzwerks vorgenommen, da das Format der Bilder nun 320x240, und nicht mehr 32x32, ist und jetzt in 29 Kategorien anstatt in 27 zu

[5] https://www.kaggle.com/grassknoted/asl-alphabet

kategorisieren ist. Nach einer ersten Version, die mit dem neuen Datenset funktionierte, habe ich zuerst einfach einen Testlauf gemacht, um zu schauen wie das Netzwerk sich so macht. Jedoch gibt es bei diesem Datenset ein neues Problem, die Rechenleistung meines Laptops reichte lang nicht mehr aus. In einer ganzen Nacht kam er mit dem Training nicht annähernd so weit wie erhofft. Doch die ETH-Studenten wussten natürlich weiter. Ich konnte mich mit SSH und SCP über einen Server mit einem Computer mit einem GPU, einer Grafikkarte, ihrer Abteilung verbinden und von meinem Laptop aus steuern. Das ermöglichte mir auf eine viel höhere Rechenleistung zuzugreifen.

3.5 Schluss

4 Outro