technische universität
dortmund

Bachelorarbeit

Code Generierung und Training eines CGP Klassifikators zur handschriftlichen Ziffernerkennung

Björn Piepenbrink

27. August 2018

Betreuer:
Prof. Dr. Günter Rudolph
Roman Kalkreuth

Fakultät für Informatik
Algorithm Engineering (LS 11)
Technische Universität Dortmund
http://ls11-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Erkennung von Handschriften ist ein Problem im Bereich des maschinellen Lernens. Gerade die Erkennung von handschriftlichen Ziffern kann genutzt werden, um die Vorgehensweise von verschiedenen Verfahren zu demonstrieren. Es gibt bereits etliche Verfahren, um handschriftliche Ziffern zu erkennen. [5] In dieser Arbeit wird Cartesian Genetic Programming (CGP) [1] genutzt, um dieser Problemstellung nachzugehen. CGP wurde bis zum Zeitpunkt der Verfassung dieser Arbeit noch nie für die Erkennung von handschriflichen Ziffern benutzt. Diese Arbeit beschäftigt sich mit dem Training von CGP auf einem Datensatz von handschriftlichen Ziffern. Die Ziffern sollen erkannt und korrekt klassifiziert werden. Außerdem soll geprüft werden, inwiefern CGP diese Aufgabe lösen kann. Des Weiteren wird versucht Schwachstellen bei der Klassifizierung zu erkennen und es soll auf Möglichkeiten, diese zu beheben, eingegangen werden. In dieser Arbeit wird ein eigens vom Autor geschriebenes Framework zur Nutzung von CGP vorgestellt und genutzt. Dieses ermöglicht, neben dem Training auch die Generierung eines Klassifikators aus den Resultaten von CGP. Diesem Klassifikator ist es möglich, mit geringem Aufwand die Ergebnisse auf neuen Daten umzusetzen und diese zu klassifizieren. In dieser Arbeit werden die verschiedenen Parameter von CGP auf ihre problembezogene Funktionsfähigkeit geprüft und ausgewertet. Daraufhin wird Embedded Cartesian Genetic Programming (ECGP) [2] verwendet. Dies ist eine Erweiterung von CGP, welche zusätzliche Funktionalität bereitstellt. Vom Autor wurde dafür ein weiteres Framework entwickelt, welches die Nutzung von ECGP ermöglicht. Mithilfe dessen wurden weitere Versuche durchgeführt, um das gegebene Problem zu bearbeiten. Die Ergebnisse des zweiten Frameworks werden anschließend ebenfalls geprüft und ausgewertet, um sie mit den Ergebnissen von CGP zu vergleichen.

Diese Arbeit soll einen ersten Ansatz zur Problemlösung bieten und Frameworks bereitstellen, die eine Weiterarbeit an dem Problem ermöglichen.

[1] Im Folgenden wird Cartesian Genetic Programming immer mit CGP abgekürzt.
[2] Im Folgenden wird Embedded Cartesian Genetic Programming immer mit ECGP abgekürzt.

1.1 Hintergrund

CGP ist eine Form von evolutionären Algorithmen, welche ein Programm als Graphenstruktur darstellen. [11] Es wurde 1999 von Miller erfunden. [8] [10] CGP codiert seine Graphenstruktur als eine Reihe von Integern, welche angeben, aus welchen Knoten und Verbindungen der Graph besteht. Außerdem geben diese an, wie das codierte Programm aus Inputs Outputs berechnet.

CGP ist ein relativ unerforschter Bereich von evolutionären Algortihmen. Gerade bei der Bildverarbeitung wurde CGP eher zur Bildtransformation eingesetzt. [16] Zur Klassifikation von Bildern existiert eine überschaubare Anzahl von Forschungsmaterial. So haben Leitner, Harding, Förster und Schmidhuber CGP genutzt, um Mars-Terrain zu klassifizieren. [6]

Zur Klassifikation von handschriftlichen Ziffern zu dem in dieser Arbeit verwendeten Datensatz liegen viele verschiedene Verfahren vor. [5] Mit CGP wurde bisher noch nicht versucht, handschriftliche Ziffern zu klassifizieren, weshalb in dieser Arbeit ein erster Ansatz zur Erreichung dieses Ziels beschrieben wird. Weitere Ziele sind es, die Robustheit von CGP und ECGP herauszuarbeiten und eventuelle Schwächen dieser Ansätze aufzuzeigen.

1.2 Motivation

Cartesian Genetic Programming ermöglicht es, effizient und flexibel Probleme des maschinellen Lernens zu lösen. [8] Für die Klassifikation von Handschriften wurde es bisher nicht benutzt und so erschien es sinnvoll, CGP anhand dieses Problems zu testen.

Im Zuge dieser Arbeit wurde eine eigens in Java implementierte Version von CGP genutzt. Die Implementierungen von CGP in Java beschränken sich auf eine Implementierung für das Java Evolutionary Computing Toolkit (ECJ) [3]. [7] Diese Implementierung wurde von Oranchak (2008) entwickelt und baut auf ECJ auf und heißt CGP for ECJ. [14] ECJ ist die quelloffenste Implementation mit den meisten Möglichkeiten von evolutionären Algorithmen und wird von Miller als Java-Implementation vorgeschlagen. [9] Es ist sehr umfassend, hoch flexibel und extrem modifizierbar, weshalb die Einarbeitungszeit entsprechend lang ist. Es herrschen kohärente Probleme zwischen ECJ und CGP for ECJ. ECJ entwickelt sich stetig weiter. Im Gegensatz dazu ist CGP for ECJ bereits mehrere Jahre alt, weshalb sich mehrere Probleme ergeben. CGP for ECJ ist nicht ohne großen Aufwand nutzbar, da es erst auf den neuesten Stand von ECJ angepasst werden muss. Ebenfalls ist es auf die von ECJ gegebenen Vorgaben beschränkt.

Um eine spezifisch für CGP gestaltete problemfreie Anwendung zu nutzen, wurde deshalb Piepenbrink-Cartesian-Genetic-Programming (PCGP) entwickelt. PCGP hat den Anspruch benutzerfreundlicher zu sein als ECJ. Es sollen dem Nutzer nur die Parame-

[3] Im Folgenden wird Java Evolutionary Computing Toolkit immer mit ECJ abgekürzt.

ter zur Wahl gestellt werden, die aus CGP bekannt sind, sodass er sich nicht erst in die Implementierung einarbeiten muss. Des Weiteren soll es möglich sein, die Werte, die von Interesse sind, leicht auszulesen. Außerdem ermöglichte eine eigene Implementierung eine leichtere Erstellung eines Klassifikators zur handschriftlichen Ziffernerkennung.

Um PCGP umfangreicher zu gestalten und die Ansätze von ECGP mitzuverwenden, wurde im weiteren Verlauf der Arbeit eine erweiterte Version von PCGP entwickelt. Diese erweiterte Version heißt Piepenbrink-Embedded-Cartesian-Genetic-Programming (PECGP). Embedded Cartesian Genetic Programming stellt eine Erweiterung von CGP dar und wurde erstmalig in Java implementiert. [18] Es wurde implementiert, um seine Anwendung auf das Problem der handschriftlichen Ziffernerkennung zu testen.

1.3 Der Datensatz

Um ein CGP-Netzwerk zu trainieren, werden viele Testbeispiele benötigt. In dieser Arbeit wurde als Datensatz der MNIST-Datensatz gewählt.

Der MNIST-Datensatz ist ein Datensatz aus handgeschriebenen Ziffern von 0 bis 9. [5] Er besteht aus einem Trainings-Datensatz von 60.000 Beispielen und einem Test-Datensatz von 10.000 Beispielen. Jede Ziffer ist in einem 28x28 Pixel großem Bild gespeichert. Die Ziffern in diesen Bildern wurden zentriert und liegen als Grauwert-Bilder vor. [5] Jedes Pixel trägt einen Wert von 0 bis 255, der den Schwarzanteil ausdrückt, wobei 0 den geringsten Schwarzanteil darstellt.

Dieser Datensatz wurde gewählt, da in dieser Arbeit nur ein einfacher Fokus auf Vorbearbeitung und Formatierung gelegt werden soll. Der MNIST-Datensatz bietet sich dafür an, da außerdem alle Bilder in derselben Größe vorliegen. Mit 60.000 Bildern besitzt der MNIST-Datensatz eine angemesse Größe, um CGP darauf zu testen. Des Weiteren wird MNIST oft verwendet, um verschiedene Klassifikationsverfahren auf Bildern zu testen. [5]

Aufgeteilt sind die Ziffern in dem Datensatz wie in Abbildung 1.1 angegeben. Des Weiteren sind in der Abbildung 1.1 einige Beispielziffern [17] des Datensatzes ergänzt.

Der Datensatz wurde im CSV-Format verwendet, da so die Daten einfacher eingelesen werden konnten. Dabei wurde die von Redmon bereitgestellte umgewandelte Version genutzt. [15]

Das MNIST-Klassifikationsproblem, welches in dieser Arbeit behandelt wird, ist das Problem zu einem Bild, die richtige Klassifikation anzugeben. So bekommt ein Programm ein Bild übergeben, woraufhin erkannt werden soll, welche Zahl auf dem Bild vorhanden ist. Ein Programm soll alle 60.000 Bilder korrekt erkennen.

Ziffer	Trainings-Daten	Test-Daten
0:	5923	980
1:	6742	1135
2:	5958	1032
3:	6131	1010
4:	5842	982
5:	5421	892
6:	5918	958
7:	6265	1028
8:	5851	974
9:	5949	1009

[17]

Abbildung 1.1: Anzahl der einzelnen Ziffern in den genutzen Datensätzen und Beispielbilder

2 Compute Server
AMD Opteron 6276 2,3 GHz
2* 32 CPUs
2* 128 GB RAM
Ubuntu 16.04

Abbildung 1.2: Spezifikationen des Batch-Systems des Lehrstuhls 11 an der TU Dortmund

1.4 Aufbau der Arbeit

Diese Arbeit beginnt zunächst mit einer Beschreibung von CGP, um den Einstieg in diese Arbeit zu vereinfachen. In dieser Beschreibung werden zuerst evolutionäre Algorithmen beschrieben, um anschließend auf die Besonderheiten einzugehen, die CGP ausmachen. So wird die Genotyp-Struktur, die Punkt-Mutation und die Auswertung von CGP-Programmen beschrieben, welche ebenfalls in PCGP verwendet wurden.

Darauf aufbauend wird PCGP vorgestellt. Für jeden Bestandteil werden seine Funktionsweise und seine Eingliederung im Gesamtkontext von PCGP beschrieben. Mithilfe von Klassendiagrammen soll das Verständnis von PCGP gewährleistet werden.

Im nächsten Abschnitt werden die Resultate von PCGP vorgestellt. Es werden verschiedenste Parameter getestet, inwiefern sie sich auf das gegebene Problem anwenden lassen. Ebenfalls wird auf Probleme bei der Klassifizierung eingegangen und versucht, diese, soweit es geht, zu beheben. Das Ziel ist die Anwendbarkeit von CGP für die Klassifzierung von handschriftlichen Ziffern zu zeigen. Mithilfe von verschiedenen Statistiken zu jedem Versuch werden die Ergebnisse analysiert, um die Korrektheit der Klassifikation zu erhöhen. Am Ende des Abschnittes werden die erzielten Ergebnisse zusammengefasst und miteinander verglichen.

Darauffolgend wird ECGP vorgestellt. Es wird auf die Besonderheiten eingegangen, die ECGP von CGP unterscheiden. Ist ein ausreichendes Verständnis von ECGP vorhanden, wird die Implementierung von ECGP vorgestellt. In der Beschreibung von PECGP wird genauer auf die einzelnen Bestandteile, des Frameworks, eingegangen.

Daraufhin werden die Ergebnisse vorgestellt, die aus der Verwendung von PECGP resultieren. Es werden verschiedene Parameter verändert und zu jeder Änderung Statistiken vorgestellt. Diese werden analysiert, um besser Parameterkombinationen zu finden. Am Ende des Abschnittes werden die Ergebnisse von PECGP mit denen von PCGP verglichen.

Für das Testen der verschiedenen Parameter von PCGP und PECGP wurde das batch-System des Lehrstuhls 11 von der TU-Dortmund genutzt [2], dessen Spezifikationen in Abbildung1.2 gelistet sind.

Am Schluss werden alle Ergebnisse zusammengefasst. Es wird beschrieben, inwiefern sich CGP auf das Problem der handschriftlichen Ziffernerkennung anwenden lässt und ob es

vorteilhaft ist ECGP zu verwenden. Ebenfalls wird ein Ausblick auf weitere Möglichkeiten gegeben, die Parameter weiter zu optimieren.

Kapitel 2

Cartesian Genetic Programming

CGP ist eine Form von evolutionären Algorithmen. Sie wurde von Julian F. Miller entwickelt. In CGP werden Programme als azyklische, zweidimensionale Graphen dargestellt. [12]

Zu Beginn des Kapitels werden zunächst evolutionäre Algorithmen kurz erklärt. Anschließend werden Besonderheiten und Unterschiede von CGP zu anderen evolutionären Algorithmen beschrieben. Es wird erläutert, wie CGP funktioniert und wie es das Prinzip der evolutionären Algorithmen umsetzt. Auch wird erklärt, wie CGP gespeichert und verarbeitet wird und welche Algorithmen es nutzt, um zu einer Lösung zu gelangen.

2.1 Evolutionäre Algorithmen

Evolutionäre Algorithmen sind Optimierungsverfahren für verschiedenste Problemstellungen. Es sind Programme, die sich über Generationen hinweg, weiterentwickeln, um gewünschte Ausgaben zu erreichen. Sie basieren auf dem Prinzip der nätürlichen Selektion von Charles Darwin. [13]

Ein Programm in einem evolutionären Algorithmus wird auch Individuum genannt. Diese Individuen erhalten Inputs und berechnen daraus einen Output. Außerdem besitzt jedes Individuum eine Fitness, welche beschreibt, wie gut ein Individuum eine angegeben Problemstellung lösen kann. Demnach können Individuen mit einer besseren Fitness, gegebene Problemstellungen besser lösen.

Zu Beginn eines evolutionären Algorithmus wird eine Grundpopulation aus verschiedenen Individuen erstellt. Dann wird mithilfe der Fitness-Funktion berechnet, wie fit diese Individuen sind. Aufbauend darauf, werden von dieser ersten Generation die Eltern für die nächste Generation ausgewählt. Mithilfe verschiedener Variationsverfahren entstehen aus diesen Eltern Nachkommen. Diese Variationsverfahren unterscheiden sich in vielen Algorithmen, sind aber meist eine Form von Mutation und Rekombination. Nachdem die Fitness für die Nachkommen berechnet wurde, werden anhand von Selektionsverfahren

verschiedene Individuen ausgewählt, die in die nächste Generation übergehen. In dieser nächsten Generation werden wieder die Eltern ausgewählt. Der evolutionäre Algorithmus arbeitet so lange bis ein Abbruchkriterium erfüllt ist. [13]

Typische Abbruchkriterien sind in der Regel, eine maximale Generationsanzahl oder ein Fitness-Wert. Die Angabe einer maximalen Generationsanzahl ermöglicht es, die Laufzeit eines Programms zu begrenzen. Ein typischer Fitness-Wert als Abbruchkriterium ist der perfekte Fitness-Wert, dieser sagt aus, dass ein Individuum zur gegebenen Problemstellung nicht bessere Ergebnisse ausgeben kann.

So kann mithilfe eines evolutionären Algorithmusses eine Annäherung zu einer Problemstellung gefunden werden. Da die Fitness der Individuen im Generationsübergang eine entscheidende Rolle spielt, überleben eher Individuen mit einem guten Fitness-Wert. Durch Variationsverfahren wird der Fitness-Wert geändert, weshalb Individuen mit einer besseren Fitness entstehen können.

So ist es möglich, mithilfe dieses Verfahrens Lösungen für Problemstellungen zu finden.

2.2 CGP-Grundlagen

In CGP wird ein Individuum als ein azyklischer, zweidimensionaler Graph dargestellt. Dieser Graph besteht aus Knoten, die die Funktionsweise des Individuums definieren. Ein Knoten bestimmt aus Eingaben eine Ausgabe, welche von anderen Knoten weiter verwendet werden kann. Individuen erhalten eine gewisse Anzahl an Inputs, welche mithilfe der Knoten eine Anzahl von Outputs berechnen. Der Output eines Individuums wird als Adresse gespeichert, welche bestimmt, woher der Output des Individuums genommen wird. [12]

Ein Knoten besteht aus mehreren Verbindungs-Genen und einem Funktions-Gen.

Das Funktions-Gen ist eine Adresse für eine Funktion. Es bestimmt, was für eine Operation der Knoten auf seinen Eingaben auszuführen hat. Die Funktionsadressen sind in einer Funktionstabelle hinterlegt. Sie enthält die Funktionen und die dazugehörigen Adressen, die einem Individuum zur Verfügung stehen. [12]

Die Verbindungs-Gene geben an, woher der Knoten seine Eingaben bekommt. Eingaben können die Input-Adressen des Programms sowie die Adressen von anderen Knoten sein. Demnach kann ein Knoten seine Eingabe direkt von einem Input erhalten oder die Ausgabe von anderen Knoten als Eingabe nutzen. Alle Knoten haben immer die gleiche Anzahl an Verbindungsgenen. Diese wird durch die Funktion vorgegeben, die in der Funktionstabelle die meisten Eingaben besitzt. So kann für jeden Knoten eine Ausgabe bestimmt werden, sofern ein Input vorliegt. [12]

Die Output-Adressen geben an, woher der Graph seinen Output nehmen soll. Als Output-Adresse kann ein Knoten oder ein Input gewählt werden. Die Eingabe des Outputs

wird daraus bestimmt, was der adressierte Knoten ausgibt oder was im adressierten Input steht. Sie führen keine Operationen aus, sondern reichen ihre Eingabe nach außen weiter.

Der Graph in CGP liegt in einer zweidimensionalen Form vor mit s Spalten und r Reihen. Die Gesamtanzahl n der Knoten in einem CGP-Graphen ergibt sich aus $n = s * r$. [12]

Graphen in CGP sind immer feed-forward. Die Knoten können sich nur zu Eingaben des Programms oder zu Knoten, die in den Spalten vor ihnen liegen, verbinden. So werden Zyklen im Programm vermieden. Der Nutzer kann die feed-forward-Eigenschaft des Graphen durch einen levels-back-Parameter beeinflussen. Der levels-back-Parameter kontrolliert die Verbundenheit des Graphen, indem er den Knoten eine maximale Anzahl an Spalten gibt, zu denen sie sich verbinden können. Bei einem levels-back-Wert von i können sich die Knoten nur zu den Knoten in den i Spalten vor ihnen verbinden. Jeder Knoten kann trotzdem einen Input als Eingabeadresse wählen.

Des Weiteren nutzt CGP meistens nur Mutation. [12] Es gibt Ansätze von CGP die Rekombination benutzen, jedoch wird hauptsächlich Mutation genutzt, weshalb in dieser Arbeit nur Mutation verwendet wird.

Da ein Nutzer die Anzahl der Knoten eines CGP-Graphen bestimmen kann und diese sich im weiteren Verlauf nicht verändern, leidet CGP nicht unter Genotyp-Wachstum und ein Programm hat immer dieselbe Maximalgröße.

2.3 Genotyp

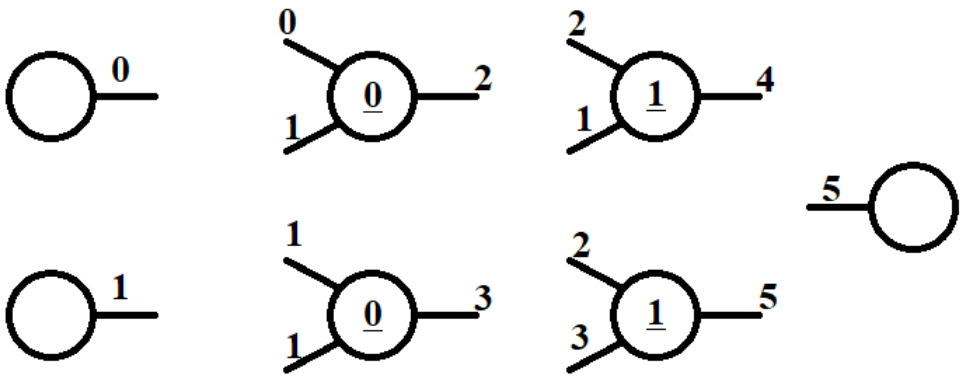
Der Genotyp beschreibt, wie ein CGP-Programm codiert ist. Er besteht aus einer Reihe von Genen, welche als Integer dargestellt werden. Integer beschreiben den zuvor beschriebenen CGP-Graphen. Dabei wird ein Knoten durch mehrere Integer repräsentiert. Diese bestimmen die Funktions- und Adress-Gene des Knoten. Am Schluss des Genotyps sind die Output-Adressen eines Individuums codiert.

2.3.1 Genotyp-Struktur

Funktionadressen sind von der Funktionstabelle vorgegeben. Diese Funktionadressen werden nur von Funktions-Genen genutzt. Die ersten Adressen im Genotyp werden von den Inputs reserviert. So wären bei einer Input-Anzahl von x die ersten 0 - (x-1) Adressen für den Input reserviert. Die Adressen für einen Knoten ergeben sich abhängig von seiner Position im Genotyp. So hätte der j-te Knoten im Genotyp bei x-vielen Inputs die Adresse (x-1)+j. [12]

2.3.1 Beispiel. Um dies zu verdeutlichen, werden wir anhand eines Beispiel-Programms die Ausgabe bestimmen und den Genotyp herleiten. Die Funktionstabelle, die wir nutzen

wollen, besteht aus 2 Funktionen mit den folgenden Adressen: 0: + & 1: - Gegeben sei folgender Graph:



Dieser Graph stellt ein Programm mit zwei Eingaben und einer Ausgabe dar. Dem Programm stehen die Funktionen Addition (+) und Subtraktion (-) bereit. Die Adressen für diese Funktionen stehen in der Funktionstabelle und sind mit einem Unterstrich gekennzeichnet. So ist zum Beispiel die Additions-Funktion mit der Adresse 0 gekennzeichnet. Die Adressen für die beiden Eingaben des Programms sind mit 0 und 1 gekennzeichnet.

Der Knoten 2 besitzt die Verbindungsgene 0 und 1 und die Funktionsadresse 0. Er addiert beide Eingaben zusammen und stellt das Ergebnis unter der Adresse 2 bereit. Knoten 3 nutzt die Eingabe unter der Adresse 0 zweimal und addiert diese. Es ist möglich, dieselbe Eingabe mehmals zu verwenden. Einschränkungen ergeben sich, da ein Knoten seine Eingaben nur von Knoten nehmen darf, welche vor ihm liegen oder von den Inputs, die im Programm enthalten sind. Des Weiteren besitzt Knoten 5 die Verbindungsgene 2 und 3. Diese stellen die bereits zuvor besprochenen Knoten dar. Aus den Ausgaben dieser Knoten wird die Funktion unter der Adresse 1 ausgeführt.

Das letzte Gen gibt an, woher das Programm seine Eingabe bekommt, die als Programmausgabe ausgegeben wird. In diesem Fall wird das Ergebnis des Knoten 5 als Ausgabe ausgegeben. Demnach wird die Funktions $(x_0 + x_1) - (x_1 + x_1)$ vom Graph codiert.

Wenn für jeden Knoten die Funktionsadresse und Verbindungsadresse notiert ist und die Outputadresse angegeben wird, ergibt sich daraus der Genotyp des Graphen. So würde der Knoten 2 im Genotyp wie folgt aussehen: 0 0 1.

Wird dies auf jeden Knoten angewendet und am Ende des Genotyps noch die Outputadresse (hier = 5) beigefügt, ergibt sich folgender Genotyp:

0 0 1 0 1 1 1 2 3 1 2 1 5.

CGP liegt immer in dieser Genotyp-Form vor. Um aus diesem Genotypen eine Ausgabe zu erhalten, muss er decodiert werden. Das Programm, was beim Decodieren entsteht, ist der Phänotyp.

2.3.2 Genotyp-Phänotyp-Mapping

Der Phänotyp ensteht aus dem Genotypen, indem dieser decodiert wird und diesen dadurch für die Output-Berechnung vorbereitet. Dieses wird das Genotyp-Phänotyp-Mäpping genannt. Der Genotyp hat eine vorher festgelegte Länge, welche der Nutzer festlegt, indem er die Anzahl der Knoten und die Anzahl der Outputs bestimmt. Die Länge des Genotyps kann sich nicht mehr ändern, da die in CGP eingefügten Knoten nicht mehr entfernt werden können. Auch können keine neuen Knoten hinzugefügt werden. Es ist möglich, dass der Phänotyp eine kürzere Länge aufweist, da bei der Output-Berechnung einige Knoten ignoriert werden können, welche dann im Phänotypen nicht erwähnt werden. Ignorierte Knoten werden vom Output und von Knoten die vom Output adressiert werden, nicht adressiert. Diese Knoten werden nicht aktive Knoten genannt. [12]

Im obigen Beispiel wurde der Knoten 4 für die Berechnung des Outputs nicht benötigt. Knoten, die bei der Berechnung ausgelassen werden können, nennen sich non-codinggenes (nicht-codierende-Gene). Sie sind jedoch wichtig für die Mutation, da sie nach einer Mutation wieder zu aktiven Knoten werden können und deshalb später für die Output Berechnung von Bedeutung sein können.

Beim Decodieren des Genotyps wird rekursiv vorgegangen. [12]

Zuerst werden die Output-Gene betrachtet, woraufhin die Verbindungen der Output-Gene zurückverfolgt werden und alle Knoten auf diesem Weg als aktiv markiert werden. So werden nicht-codierende-Gene als nicht aktiv markiert und können bei der Berechnung ausgelassen werden.

Nachdem der gesamte Genotyp durchlaufen wurde und alle aktiven Knoten markiert wurden, kann die eigentliche Berechnung beginnen. Dabei werden für einen gegebenen Input die Ergebnisse für alle aktiven Knoten berechnet und der Output des Programms ausgegeben. So kann bei der Fitnessberechnung schnell ein Programmoutput bestimmt werden, ohne dass nicht-codierende-Gene mit berechnet werden. [12]

2.3.2 Beispiel. Sei der Genotyp aus unserem vorherigen Beispiel gegeben:

0 0 1 0 1 1 1 2 3 1 2 1 5

Der entsprechende Phänotyp sieht dann wie folgt aus:

0 0 1 0 1 1 1 2 1 5.

Da der Knoten 4 nie vom Output adressiert wird, ist er im Phänotypen nicht aufgelistet.

2.4 Mutation

Um ein Individuum zufällig zu verändern und vielleicht eine bessere Fitness zu erhalten, gibt es Mutation. Mutation ändert zufällige Teile des Genotypen. CGP benutzt im Gegensatz zu vielen anderen evolutionären Algorithmen nur die Punkt-Mutation, um Individuen

zu verändern. Da CGP nur Punkt-Mutation verwendet, wird diese im Folgenden Abschnitt nur mit Mutation beschrieben. [12]

In der Mutation wird eine zufällige Genotyp-Position gewählt. Das Gen unter dieser Position wird daraufhin in einen anderen zulässigen Wert geändert. So kann sich zum Beispiel der Output eines Individuums ändern und das Individuum möglicherweise sogar eine größere Fitness erzielen.

Die Werte, die ein Gen bei einer Mutation annehmen kann, sind begrenzt. Die Begrenzungen, die beschreiben, welche Werte erlaubt sind, nennen sich allelische Begrenzungen (Allelic Constraints). [12] Wird ein Funktions-Gen gewählt, ist jede mögliche Adresse aus der Funktionstabelle ein gültiger Wert. Sollte ein Verbindungs-Gen gewählt werden, kann die Verbindung zu einem Input oder zu einem anderen Knoten geändert werden (unter Beachtung des levels-back-Parameters). Wenn eines der Output-Gene gewählt wird, kann die Verbindung wie bei einem Verbindungs-Gen geändert werden. [12]

2.4.1 Beispiel. Wir werden im Folgenden unseren vorher definierten Genotyp mutieren, um die Effekte der Mutation zu zeigen. Hierbei werden wir zunächst ein Funktions-Gen, dann ein Verbindungs-Gen und am Schluss ein Output-Gen mutieren.

Genotyp: $\underline{0}$ 0 1 $\underline{0}$ 1 1 $\underline{1}$ 2 3 $\underline{1}$ 2 1 5.

Wird eine der Positionen mit Funktions-Genen zur Mutation ausgewählt (Positionen 1,4 ,7 und 10), kann sie in diesem Beispiel zwei verschiedene Werte annehmen. Wird Position 7 ausgewählt, kann sie entweder bei der Adresse $\underline{1}$ bleiben oder zu $\underline{0}$ wechseln, da dies alle möglichen Adressen in unserer Funktionstabelle sind. Wenn wir die Adresse zu $\underline{0}$ wechseln, so ergibt sich der Genotyp: $\underline{0}$ 0 1 $\underline{0}$ 1 1 $\underline{0}$ 2 3 $\underline{1}$ 2 1 5.

Wählen wir nun die 11. Position zur Mutation aus. Diese Position enthält ein Verbindungs-Gen und gehört zum Knoten 5. Sie kann also die Werte: 0,1,2,3 annehmen, von denen wir die Adresse 0 wählen. Es ergibt sich der Genotyp: $\underline{0}$ 0 1 $\underline{0}$ 1 1 $\underline{0}$ 2 3 $\underline{1}$ 0 1 5.

Mutieren wir nun noch das Output-Gen zum Wert 4 (möglich sind alle Adressen von 0 bis 5), ergibt sich der Genotyp: $\underline{0}$ 0 1 $\underline{0}$ 1 1 $\underline{0}$ 2 3 $\underline{1}$ 0 1 4. Dieser codiert nun nach 3 Mutationen die Funktion: $(x_0 + x_1) + (x_1 + x_1)$. Oder gekürzt: $x_0 + 3x_1$.

Um die Anzahl der Änderungen, die während einer Mutation vorgenommen werden, zu kontrollieren, kann der Nutzer eine Mutationsrate angeben. Üblicherweise wird diese als Prozentanteil der gesamten Gene angegeben. Indem die Anzahl aller Gene mit dem Prozentsatz multipliziert wird, ergibt sich die Anzahl der Mutationen. In unserem Beispiel wurde eine Mutationsrate von 3/13 gewählt.

2.5 Evolutionärer Algorithmus

In CGP wird meistens eine $(\mu + \lambda)$-Strategie verwendet. Üblicherweise ist dies eine (1+4)-Strategie. [12]

Zu Beginn des Algorithmus wird eine Population aus zufälligen Individuen erstellt. Im Anschluss daran werden diese ausgewertet und deren Fitness bestimmt. Das Individuum mit der besten Fitness wird zum Elternteil befördert. Ein Kind dieses Elternteils wird erstellt, indem der Elternteil per Mutation verändert wird. Um vier Kinder zu erstellen, wird der Prozess vier separate Male ausgeführt. Aus diesen vier Kindern wird das fitteste bestimmt und zum Elternteil befördert. Der Algorithmus verfährt weiter, indem er das Elternteil mutiert, um Kinder zu erschaffen und wieder eine neue Generation zu erstellen.

Es gibt zwei Kriterien, welche verhindern, dass eine neue Generation gebildet wird. Zum einen kann der Benutzer ein Generationslimit angeben, dass vom Algorithmus nicht überschritten werden kann. Außerdem kann der Benutzer eine ausreichende Fitness definieren. Wird diese von einem Individuum erreicht, wird keine neue Generation mehr gebildet. Normalerweise ist dies eine perfekte Fitness, sie kann aber auch verringert werden, wenn die Ergebnisse ausreichend sind.

Der (1+4)-Evolutions-Algorithmus: [12]

1. Erstelle eine Population der Größe 5 mit zufällig erstellten Individuen
2. Generations-Nummer g = 0
3. Berechne für jedes Individuum der Population die Fitness
4. Wähle das beste Individuum als Elternteil aus
5. Mutiere das Elternteil 4 mal, um 4 Kinder zu generieren
6. Erstelle eine neue Generation aus dem Elternteil und den 4 Kindern
7. g++
8. 3-6 bis die maximale Generationsanzahl erreicht wurde oder eine akzeptable Fitness erreicht wurde

In CGP wird ein Individuum das Elternteil der nächsten Generation, wenn es eine gleich gute oder eine bessere Fitness besitzt. Dies ist das Prinzip der "Neutralen Mutation". Es hat sich gezeigt, dass dieses Prinzip hilfreich ist, um schneller oder überhaupt eine Lösung zu finden [12]. Es sorgt dafür, dass das überlebende Individuum eine Art von Variation erfährt und nicht stagniert.

2.5.1 Beispiel. Im Folgenden wird der (1+4)-Evolutions-Algorithmus an einem Beispiel erläutert.

Sei eine Startpopulation mit fünf Individuen gegeben. Und sei außerdem ein größerer Fitnesswert besser als ein kleinerer.

Die Individuen besitzen folgende Fitnesswerte:
I : 10, II : 7, III : 13, IV : 9, V : 12

Nun geht das beste Individuum III (mit Fitness 13) in die nächste Generation über und generiert Kinder.

III : 13, I' : 12, II' : 9, III' : 13, IV' : 7

Ein Kind des Individuums hat nun dieselbe Fitness, wie sein Elternteil. Aufgrund des Prinzips der neutralen Mutation wird dieses nun in die nächste Generation voranschreiten und wieder Kinder generieren.

Kapitel 3

PCGP

In diesem Kapitel wird das vom Autor entworfene Programm vorgestellt und beschrieben, mit dem sich CGP für den MNIST-Datensatz ausführen lässt.

3.1 Grundlegendes

Piepenbrink-Cartesian-Genetic-Programming (oder kurz: PCGP) ist ein von Björn Piepenbrink entworfenes Programm, welches eigens für den MNIST-Datensatz entworfen wurde. Es soll die Daten des Datensatzes einlesen und lernen, sie korrekt zu klassifizieren. PCGP wurde entwickelt, um die Effekte eines CGP-Netzwerkes genau nachvollziehen zu können und jeden Schritt nach Belieben ändern zu können. Außerdem ist PCGP komplett konfigurierbar und erlaubt es dem Nutzer, viele Parameter nach seinem Belieben zu verändern. Es wurde in Java 8 entwickelt.

PCGP wurde komplett von Björn Piepenbrink implementiert und nutzt lediglich die Werkzeuge, die einem mit dem Java-Development-Kit zur Verfügung gestellt werden, und JavaPoet [1]. JavaPoet ist eine Schnittstelle um Java-.class-Dateien zu erstellen. In PCGP wird JavaPoet benutzt, um aus einem Individuum einen Klassifikator zu erstellen. Dieser wird als eigene Klasse gespeichert und funktioniert autonom.

PCGP arbeitet mit einem eindimensionalen CGP-Netzwerk, bestehend aus einer Reihe von Knoten. CGP wird heutzutage hauptsächlich nur in eindimensionaler Darstellung genutzt. Es nutzt dabei eine $(\mu + \lambda)$-Strategie, um bessere Individuen zu erhalten. Die Knoten sind hierbei als eigene Klassen gespeichert und enthalten drei Integer, die ihre Funktionsadresse und ihre Inputadressen bestimmen. In PCGP werden nur Funktionen mit zwei Inputs verwendet. Die Fitness eines Individuums wird als Anzahl der falsch klassifizierten Ziffern beschrieben. Individuen mit einem geringeren Fitness-Wert besitzen dementsprechend eine höhere Fitness und können das Klassifizierungsproblem besser lösen.

Im folgenden Teil dieses Kapitels, werden zunächst die Anforderungen an PCGP vorgestellt, woraufhin die Grundstruktur erläutert wird. Anschließend werden die einzelnen

Teile des Programmes behandelt und beschrieben, sodass der Leser Einblicke in die Funktionsweise von PCGP gewinnt.

3.1.1 Anforderungen

PCGP hat den Anspruch höchst konfigurierbar zu sein. Es soll für jeden Nutzer verständlich sein, sodass nach einer kurzen Einarbeitung volle Funktionalität gewährleistet sein soll. Das Framework ist gut kommentiert, sodass PCGP zu keiner Zeit unverständlich ist. Der Anspruch an das Framework ist, dass Individuen in angemessener Zeit mutiert werden, ihnen eine Fitness zugewiesen wird und sie miteinander vergleicht. Es soll komplett einsichtig sein, sodass zu einem Individuum alle Informationen vorliegen, welche der Nutzer benötigt. Das Hauptziel von PCGP ist es, dem Leser einen Einblick in CGP zu gewähren, damit genaue Abläufe verständlicher gemacht werden. PCGP soll keine grafische Oberfläche anbieten, da dies nichts zum Kernthema dieser Arbeit beitragen würde.

3.1.2 Grundlegende Struktur

Der CGPNetHandler-Klasse werden alle Argumente übergeben, die für die Ausführung des CGP-Netzwerkes nötig sind. Sie erhält die Nummer des Funktions-Set, das benutzt werden soll. Außerdem werden dieser Klasse weitere Parameter übergeben, die für die Nutzung des Netzwerkes von Nöten sind. Diese sind:

1. Mutations-Rate
2. Anzahl der Knoten
3. maximale Generationsanzahl
4. Anzahl der Outputs des Netzwerkes
5. Höhe des levels-back-Parameters
6. $(\mu + \lambda)$-Strategie

Des Weiteren werden dem CGPNetHandler die Input-Ziffern übergeben. Diese enthalten das Bild der Ziffer und deren Klassifikation, welche dem CGPNetHandler als eine Liste von Digit-Objekten übergeben wird. Mithilfe der Network-Klasse werden die übergebenen Parameter genutzt, um ein CGP-Netzwerk zu erstellen.

In der Network-Klasse geschieht der eigentliche $(\mu + \lambda)$-Algorithmus. Sie enthält Individuen, welche als Individual-Objekte vorliegen. Individual-Objekte besitzen eine Liste aus Node-Objekten, welche ihre Knoten darstellen. Sie werden zu Beginn zufällig in der Network-Klasse anhand der übergebenen Parameter erstellt. Die Network-Klasse führt dann den $(\mu + \lambda)$-Algorithmus aus. Anschließend wird dem CGPNetHandler das beste Individuum des Netzwerkes zurückgegeben.

In PCGP werden nur levels-back-Werte von eins oder größer angenommen. Wird ein anderer Wert angegeben, so wird der levels-Back-Parameter nicht berücksichtigt.

3.2 Hilfsklassen

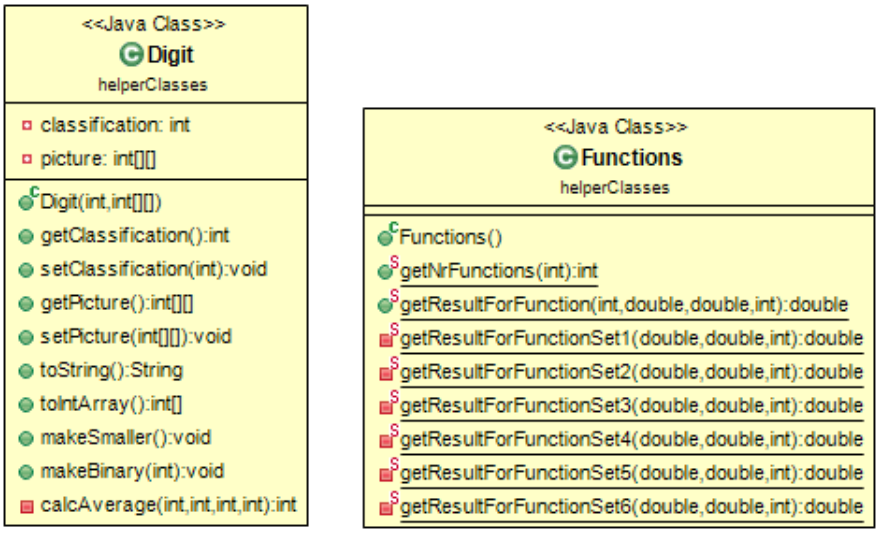


Abbildung 3.1: Klassendiagramm Hilfsklassen

Im Folgenden werden zwei Klassen beschrieben, die benötigt sind, um PCGP auszuführen. Sie stellen die problembezogenen Input-Daten und die Funktionstabelle dar.

3.2.1 Digit

Die Digit-Klasse enthält eine Ziffer aus dem MNIST-Datensatz und deren Klassifikation. Die Klassifikation wird als Integer im Bereich [0 , 9] gespeichert. Das Bild der Ziffer wird als zweidimensionales Integer-Array gespeichert. Dabei steht jeweils ein Wert in diesem 2D-Array für ein Pixel des Bildes. Neben einem Konstruktor und Getter- und Setter-Klassen für die Attribute besitzt die Digit-Klasse mehrere Methoden, die mit dem gespeicherten Bild arbeiten.

Die toString-Methode ermöglicht es, die Klassifikation sowie das Bild als String auszugeben, um eventuell Fehler zu erkennen. Um das gespeicherte Bild als eindimensionales Array auszugeben, wird die toIntArray-Methode benötigt. Dabei werden die Reihen des Bildes nacheinander aufgelistet. Dies ist nötig, da PCGP mit einer eindimensionalen Eingabe-Struktur arbeitet und nur dazu ein Output berechnet werden kann.

Ebenfalls besitzt die Klasse zwei Methoden, um die Darstellung des Bildes zu vereinfachen.

So verringert die makeSmaller-Methode die Größe des Bildes, in dieser ein Bild erzeugt wird, welches nur ein Viertel so groß ist wie das ursprüngliche Bild. Dies geschieht,

indem für vier in einem Quadrat nebeneinanderliegende Pixel der Mittelwert ihrer Inhalte berechnet wird. Anschließend wird der Mittelwert in einem neuen Bild entsprechend eingesetzt. So ergibt sich ein kleineres Bild, was die Eingabe für ein PCGP-Netzwerk wesentlich verringern kann. Die makeBinary-Methode macht aus einem Grauwert-Bild ein Schwarz-Weiß-Bild. Sie verlangt ein Schwellwert als Parameter, welcher bestimmt, ob der Wert eines Pixels des Bildes eine null oder eins annimmt. Wenn der Wert größer als der Schwellwert ist, wird eine eins auf dem Pixel eingetragen. Andernfalls wird auf dem entsprechenden Pixel eine null eingetragen. Diese Methode dient dazu, das Bild noch weiter zu vereinfachen.

3.2.2 Functions

Die Functions-Klasse stellt die Funktions-Adressen Tabelle von CGP dar. Sie enthält die Funktionen, die unter den jeweiligen Adressen zu erreichen sind. Alle Funktions-Adressen beginnen bei eins und enden bei der Anzahl der im FunktionenSet enthaltenen Funktionen. Es werden verschiedene Funktions-Sets benutzt, welche alle eine Anzahl von Funktionen mit zwei Inputs besitzen.

Mit der getNrFunctions-Methode lässt sich die Anzahl der Funktionen des aktuellen Funktionen-Sets zurückgeben. Dies ist nötig, da verschiedene Funktions-Sets verschiedene Anzahlen an Funktionen besitzen. Da die Funktionsadressen maximal die Adresse der letzten Funktion annehmen können, ist es nötig, diese abfragen zu können. Die Methode getResultForFunction berechnet einen Wert, der ausgegeben werden soll. Dafür wird das zu benutzende Funktions-Set, die beiden Inputs und die Funktions-Adresse benötigt. Diese ruft dann, abhängig vom benutzten Funktions-Set, die Funktion auf, die den Output berechnet.

Die Funktions-Sets sind in der Functions-Klasse als Methoden codiert, welche aus zwei Inputs und einer Funktions-Adresse den jeweiligen Output ausgeben.

Das erste Funktions-Set wurde mit den vier grundlegenden mathematischen Operationen eigens entworfen. Das zweite und sechste Funktions-Set enthalten verschiedene Funktionen. [11] Funktions-Set 2 arbeitet mit einer konstanten von 255. Hingegen arbeitet das sechste mit einer Konstanten von eins. Das dritte und vierte Funktions-Set nutzen low-level-image-processing-Funktionen, um mit Bildern zu arbeiten. [16] [3] Diese wurden in anderen Arbeiten für einfache Bildverarbeitung genutzt. Das fünfte ist ein Funktions-Set mit logischen-Operationen für boolsche Eingaben. Die Funktions-Sets sind in Abbildung 3.2 und 3.3 aufgelistet.

Funktions-Set 1

Funktion	Beschreibung
ADD	Addiert die zwei Inputs
SUB	Subtrahiert den zweiten vom ersten Input
MULT	Multipliziert beide Inputs
DIVIDE	Dividiert den ersten Input mit dem zweiten (wenn der zweite Input null ist wird null zurückgegeben)

Funktions-Set 2 und 6

Funktion	Beschreibung
ADD	Addiert die zwei Inputs
SUB	Subtrahiert den zweiten vom ersten Input
MULT	Multipliziert beide Inputs
DIVIDE	Dividiert den ersten Input mit dem zweiten (wenn der zweite Input null ist wird null zurückgegeben)
ADD CONST	Addiert eine Konstante zum ersten Input
SUB CONST	Subtrahiert eine Konstante vom ersten Input
DIV CONST	Dividiert den ersten Input mit einer Konstante
SQRT	Gibt die Wurzel des ersten Inputs zurück
POW	Berechnet die Potenz der Inputs Dabei ist der erste Input die Basis und der zweite der Exponent (ist das Ergebnis undefiniert, wird eins zurückgegeben)
SQUARE	Quadriert den ersten Input
COS	Berechnet den Kosinus des ersten Inputs
SIN	Berechnet den Sinus des ersten Inputs
NOP	Gibt den ersten Input zurück
CONST	Gibt die jeweilige Konstante zurück
ABS	Berechnet den Betrag des ersten Inputs
MIN	Gibt den kleineren der beiden Inputs zurück
MAX	Gibt den größeren der beiden Inputs zurück
LOG2	Berechnet den Logarithmus des ersten Inputs (zur Basis 2)
ROUND	Rundet den ersten Input
FRAC	Gibt den Bruchteil des ersten Inputs zurück
RECIPRICAL	Gibt (1 / ersten Input) zurück
RSQRT	Gibt (1 / Wurzel des ersten Inputs) zurück

Konstante bei Funktions-Set 2 = 255

Konstante bei Funktions-Set 6 = 1

Abbildung 3.2: Verwendete Funktions-Sets

Funktions-Set 3

Funktion	Beschreibung
CONSTANT	Gibt 255 zurück
IDENTITY	Gibt den ersten Input zurück
INVERSION	Subtrahiert den ersten Input von 255
BITWISE OR	Verodert den ersten und zweiten Input in Bitdarstellung
BITWISE AND	Verundet den ersten und zweiten Input in Bitdarstellung
BITWISE NAND	Verundet den ersten und zweiten Input in Bitdarstellung und negiert diesen dann
BITWISE OR 2	Verodert den inversen ersten Input und den zweiten Input in Bitdarstellung
BITWISE XOR	Führt den XOR-Operator auf beiden Inputs in Bitdarstellung aus
RIGHTSHIFT ONE	Verschiebt den ersten Input in Bitdarstellung um einen nach rechts
RIGHTSHIFT TWO	Verschiebt den ersten Input in Bitdarstellung um zwei nach rechts
SWAP	Verschiebt erst beide Inputs um 4 und verundet diese dann (in Bitdarstellung)
ADD SAT	Addiert erst beide Inputs gibt dann den kleineren Wert von dem Ergebnis oder 255 zurück
ADD	Addiert beide Inputs
AVERAGE	Berechnet erst den ADD SAT- Oprator und teilt das Ergebnis durch zwei
MAX	Gibt den größeren der beiden Inputs zurück
MIN	Gitb den kleineren der beiden Inputs zurück

Funktions-Set 4

Funktion	Beschreibung
BITWISE OR	Verodert den ersten und zweiten Input in Bitdarstellung
BITWISE AND	Verundet den ersten und zweiten Input in Bitdarstellung
BITWISE XOR	Führt den XOR-Operator auf beiden Inputs in Bitdarstellung aus
ADD SAT	Addiert erst beide Inputs gibt dann den kleineren Wert von dem Ergebnis oder 255 zurück
ADD	Addiert beide Inputs
AVERAGE	Berechnet erst den ADD SAT- Oprator und teilt das Ergebnis durch zwei
MAX	Gibt den größeren der beiden Inputs zurück
MIN	Gitb den kleineren der beiden Inputs zurück

Funktions-Set 5

Inputs sollen hierbei in Binärdarstellung vorliegen

Funktion	Beschreibung
AND	Verundet beide Inputs
OR	Verodert beide Inputs
NOT	Negiert den ersten Input

Abbildung 3.3: Verwendete Funktions-Sets

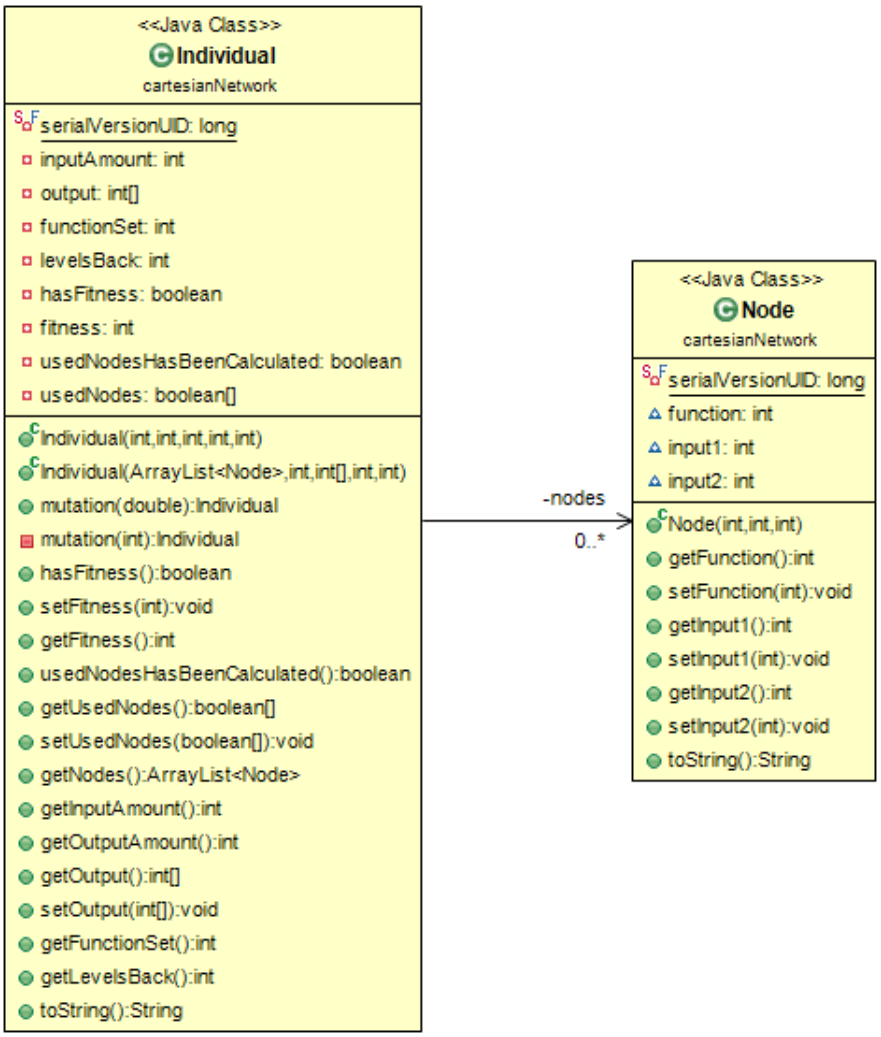


Abbildung 3.4: Klassendiagramm Hilfsklassen

3.3 Das Netzwerk

Im Folgenden werden alle Klassen beschrieben, die die Grundstruktur des CGP-Graphen bilden. Diese Klassen codieren ein Individuum mit Knoten und das Netzwerk, dass diese umfasst.

3.3.1 Node

Ein Node-Objekt repräsentiert einen Knoten in einem Individuum. Es enthält drei Integer, die die Adressen des Knoten beschreiben. Ein Integer enthält die Funktionsadresse (function) und die anderen beiden (input1 und input2) enthalten die jeweiligen Eingaben des Knoten. Da in PCGP alle Funktionen nur zwei Inputs haben, sind die Anzahl der Eingabe-Adressen eines Knoten vorgegeben.

Abgesehen von einem Konstruktor enthält die Klasse die benötigten Getter und Setter. Diese werden benötigt, da auf die Adressen des Knoten voller Zugriff nötig ist. Bei der Outputberechnung müssen sie gelesen und bei der Mutation beschrieben werden. Des Weiteren enthält die Node-Klasse eine Methode, um eine Node als String auszugeben.

3.3.2 Individual

Die Individual-Klasse repräsentiert ein Individuum aus CGP.

Ein Individual-Objekt besitzt eine Liste von Knoten (nodes), welche den Knoten-Graphen darstellen, der ein Individuum in CGP codiert. Da PCGP mit einer eindimensionalen Graphenstruktur arbeitet, werden die Node-Objekte in einer Liste abgespeichert. Die Anzahl der Inputs des Graphen werden auch im Individual unter dem Namen inputAmount gespeichert. Dies ist nötig, damit die ersten Adressen für den Input reserviert werden können.

Die Adressen in einem Individual beginnen mit den Adressen: 0 - (inputAmount-1) und sind die Adressen für alle Inputs. Anschließend beginnen die Adressen der Knoten und besetzen die Positionen: inputAmount - (node.size() -1). Dabei stellt nodes.size() die Länge der Liste der Knoten dar, also die Anzahl von Knoten, die eine Individual-Objekt enthält. Diese Begrenzungen für die Adressen sind im Individual nicht explizit gespeichert, da sie aus der Anzahl der Inputs und Knoten hergeleitet werden können.

Output-Adressen sind im Individual-Objekt direkt gespeichert. Die zu benutzende Funktions-Tabelle ist als Integer gespeichert, mit welcher sich, unter Zuhilfenahme von der Functions-Klasse, die Funktionstabelle und die möglichen Funktions-Adressen bestimmen lassen. Außerdem ist der levels-back-Parameter ebenfalls im Individuum gespeichert, da dieser wichtig für die Konnektivität der einzelnen Knoten ist. Für die spätere Fitness- und Output-Berechnung sind im Individuum der Fitness-Wert und die aktiven-Knoten gespeichert. Ob ein Knoten aktiv ist, wird in einem boolean-Array gespeichert, das für jeden Knoten enthält, ob dieser aktiv ist oder nicht. Dazu sind booleans gespeichert, die angeben, ob diese Werte schon gespeichert wurden.

Konstruktoren

Die Individual-Klasse besitzt zwei verschiedene Konstruktoren. Der erste Kontruktor erlaubt es, ein zufälliges Individuum unter Einhaltung der übergebenen Werte zu konstruieren. Ihm werden die Anzahl der zu generienden Nodes, die Input- und Output-Anzahl sowie das zu benutzende Funktions-Set und der levels-back-Parameter übergeben. Darauf aufbauend wird ein Individuum mit zufälligen Werten erstellt.

Zuerst werden für jede Node eine zufällige Funktions-Adresse und zwei zufällige Eingabe-Adressen bestimmt. Zulässige Werte für die Funktions-Tabelle liegen im Bereich: [1, höchste-Funktions-Adresse]. Die höchste Funktions-Adresse wird hierbei, abhängig vom verwen-

deten Funktions-Set mit der Klasse Functions bestimmt. Die Eingabe-Adressen für jede Node sind hierbei vom levels-back-Parameter abhängig. Wurde kein oder ein unzulässiger levels-Back Parameter angegeben, so wird dieser nicht bei der zufälligen Adresswahl berücksichtigt.

Zulässige Werte für die Eingabe-Adressen sind von der Position der Node im Genotyp abhängig, weshalb sich die zulässigen Werte im Bereich: [0, inputAmount+ i) befinden. Wobei gilt: i = Position des Node-Objektes in der Liste der Node-Objekte (beginnend mit null). Dadurch wird gewährleistet, dass ein Knoten nur die vor ihm liegenden Knoten und Inputs adressieren kann.

Wird dagegen ein levels-back-Parameter angegeben, sind die zulässigen Eingabe-Adressen, aus denen ein zufälliger Wert gewählt werden darf, erst zu berechnen. Levels-back-Parameter werden erst ab einem Wert von eins oder größer verarbeitet. Da mit negativen Werten nicht umgegangen werden soll und ein Wert von 0 bedeuten würde, dass sich Nodes nur zu Input-Adressen verbinden könnten. Die zulässigen Werte für eine Eingabe-Adresse liegen dann in den Bereichen: [0 , inputAmount -1] und [inputAmount + i - levelsBack , inputAmount+ i - 1], wobei: i = Position des Node-Objekts in der Liste der Node-Objekte (beginnend mit null). Wird aus diesen ein zufälliger Wert ausgewählt und als Eingabe-Adresse zugewiesen, ergibt sich ein Problem. Bei Nodes, die die ersten levels-back-vielen Stellen in der Node-Liste einnehmen, hätten die letzten Input-Adressen eine erhöhte Wahrscheinlichkeit als Eingabe-Adresse gewählt zu werden.

3.3.1 Beispiel. Um diese erhöhte Wahrscheinlichkeit für manche Inputs zu erläutern, wird dies hier an einem Beispiel erläutert. Sei eine Input-Anzahl von fünf Inputs gegeben und eine Liste von vier Knoten. Die Inputs belegen hierbei die ersten Adressen: $\{0, 1, 2, 3, 4\}$. Die Knoten belegen demnach die Adressen: $\{5, 6, 7, 8\}$. Sei außerdem ein levels-back-Wert von zwei gegeben.

Nun wollen wir eine zufällige Eingabe-Adresse für den Knoten bestimmen, der unter der Adresse 6 erreichbar ist. Nehmen wir einen zufälligen Wert aus den Bereichen [0 , inputAmount -1] und [inputAmount + i - levelsBack , inputAmount+ i - 1]]. Hierbei ist i=1, da der Knoten unter der Adresse 6 an der zweiten Position in der Liste steht und die erste Position mit null beginnt.

Für den ersten Bereich wären also die Werte $\{0, 1, 2, 3, 4\}$ möglich und für den zweiten die Werte $\{4, 5\}$. Würde aus diesen beiden Bereichen eine zufällige Adresse ausgewählt, so hätte die Adresse 4 eine erhöhte Wahrscheinlichkeit ausgewählt zu werden. Da diese in beiden Bereichen vorkommt, müssten die Bereiche eine leere Schnittmenge besitzen, um für jeden Wert die gleiche Wahrscheinlichkeit zu besitzen. Es sollte also nur noch zwischen den Adressen 0 bis 5 ausgewählt werden, wobei jede Adresse eine Wahrscheinlichkeit von 1/5 besitzt.

Um dieses Problem zu lösen werden wir zunächst den levels-back-Wert abhängig von der Position eines Node-Objektes einschränken. Der levels-back-Wert soll für eine einzelne Node nie höher sein als die Position dieser Node in der Liste von Node-Objekten. Wäre der levels-back-Wert höher, so könnten durch ihn fälschlicherweise Inputs adressiert werden. Deswegen wurde für jede Node ein temporärer levels-back-max-Wert erstellt. Dieser wird auf die Position der Node herabgesetzt, wenn der levels-back-Wert größer ist als die Position der Node. Nun können wir eine zufällige Adresse aus dem Bereich [0 , inputAmount + levels-back-max) wählen. Wird ein Wert im Bereich [0 , inputAmount -1] gewählt, wird die Eingabe-Adresse auf den jeweiligen Input gesetzt. Wird jedoch ein Wert im Bereich [inputAmount , inputAmount + levels-back-max) gewählt, wird zuerst die Eingabe-Adresse berechnet. Zunächst wird der zufällige Wert unabhängig von der Anzahl der Inputs gemacht, sodass er im Bereich [0, levels-back-max) liegen kann. Dann wird von der Position der momentanen Node dieser Wert abgezogen, um die Adresse der adressierten Node zu bekommen. Sei x der zufällig gewählte Wert im Bereich [0, levels-back-max) und i die Position des Node-Objektes. Die Eingabe-Adresse der Node bestimmt sich wie folgt: Eingabe-Adresse = $inputAmount + i - x - 1$.

Nachdem für eine Node die Funktions-Adresse sowie beide Eingabe-Adressen bestimmt wurden, wird sie zur Liste der Node-Objekte hinzugefügt. Wird dies so oft ausgeführt, wie durch die Anzahl der zu generienden Nodes vorgegeben, besitzt das Individuum so viel Nodes wie gewollt. Danach müssen die Output-Adressen zufällig gewählt werden. Diese werden wie die Eingabe-Adressen für Nodes bestimmt. Zuletzt wird gespeichert, ob Fitness und aktive Knoten berechnet wurden, um ein vorzeitiges Zugreifen darauf zu verhindern, wenn diese noch nicht vorliegen.

Der zweite Konstruktor der Individual-Klasse dient dazu, ein Individual-Objekt mit spezifischen-Werten zu erstellen. Ihm werden die Liste von Node-Objekten, die Anzahl der Inputs, die Output-Adressen, das zu benutzende Funktions-Set und der levels-back-parameter direkt übergeben. Hier wird außerdem gespeichert, dass Fitness und aktive Knoten nicht berechnet wurden, da diese, wenn nötig, noch im Nachhinein ergänzt werden können.

Mutation

Die Individual-Klasse in PCGP besitzt eine Methode zur Mutation. Wird die mutation-Methode auf einem Individual-Objekt aufgerufen, wird ein neues Individual erstellt, welches aus dem ersten durch CGP-Punkt-Mutation mutiert ist. Es gibt zwei Varianten dieser mutation-Methode.

Die erste Variante erhält einen Prozentsatz von Genen, die mutiert werden sollen. Daraus wird die genaue Anzahl der Mutationen errechnet, indem die gegebene Mutationsrate mit der Anzahl der Gene multipliziert wird. Die Anzahl der Gene in PCGP ergibt sich

aus $Knotenanzahl * 3 + Outputanzahl$, da jede Node in PCGP drei Gene enthält. Anschließend wird die zweite Variante der Methode aufgerufen, die mit der genauen Anzahl der Mutationen arbeitet. Da PCGP, um Verwirrungen zu vermeiden, nur mit einer Prozentanzahl als Mutations-Rate arbeitet, ist die zweite Variante außerhalb der Klasse nicht aufrufbar.

Die wahre Mutation erfolgt dann in der zweiten Methode, die mit der genauen Anzahl der Mutationen arbeitet. Sie erstellt zuerst ein neues Individual-Objekt, was die gleichen Werte, wie das aktuelle Individual-Objekt besitzt. Anschließend werden auf dem neu erstellten Individual-Objekt die Gene mutiert. Zum Schluss wird dieses Individuum wieder zurückgegeben. Um einen identischen Nachkommen zu erzeugen, werden zunächst die Node-Liste und die Output-Adressen kopiert. Danach wird ein Nachkommen mit den gegebenen Werten erzeugt. Dieser wird dann mutiert. Es wird so oft eine Punkt-Mutation ausgeführt, wie es der übergebene Wert vorgibt.

Eine Punkt-Mutation beginnt damit, dass eine zufällige Stelle g im Genotypen gewählt wird. In PCGP wird also eine Position g im Bereich [0, $Knotenanzahl*3+Outputanzahl$) gewählt.

Wenn $g \geq Knotenanzahl*3$, wurde ein Output-Gen zur Mutation gewählt. Das genaue Output-Gen, das mutiert werden soll, ergibt sich aus der Formel: $g - Knotenanzahl * 3$. Daraufhin wird eine zufällige Adresse bestimmt, die von dem Output-Gen adressiert wird. Diese zufällige Output-Adresse wird, wie bereits im Konstruktor beschrieben, gewählt. Danach wird der Output des neu erstellten Individual-Objektes auf diesen Wert gesetzt.

Wenn $gmod3 = 0$, wurde ein Funktions-Gen zur Mutation gewählt, da jedes dritte Gen im Genotypen ein Funktions-Gen ist. Für dieses gewälte Gen muss eine zufällige Funktionsadresse gewählt werden. Dies geschieht wie im Konstruktor, indem eine Adresse zwischen 1 und der von der Functions-Klasse maximalen Funktions-Adresse gewählt wird. Um die genaue Position des Gens herauszufinden, muss die Position g durch drei geteilt werden, da jeder Knoten aus drei Genen besteht.

In allen anderen Fällen wurde ein Verbindungs-Gen zur Mutation ausgewählt. Die genaue Position des Knoten, dessen Verbindungs-Gen geändert werden soll, ergibt sich aus $g/3$. Wenn $gmod3 = 1$ gilt, soll die erste Eingabe zufällig gewählt werden. Bei $gmod3 = 2$ wird die zweite Eingabe verändert. Der jeweiligen Eingabe wird dann eine zufällige Adresse zugewiesen. Diese wird, wie vorher beschrieben, bestimmt.

Wird dies, so oft wie angegeben, durchgeführt, ist die Mutation des erstellten Individual-Objekts vollendet. Dieses wird dann zurückgegeben.

Restliche Methoden

Abgesehen von den üblichen Getter-Methoden der Individual-Klasse, die die jeweiligen Werte des Objektes zurückgeben, besitzt die Individual-Klasse einige besondere Getter-

Methoden. So geben die Methoden, die die Fitness oder die aktiven Knoten zurückgeben, einen Fehler aus, falls auf eine Fitness oder aktive Knoten zugegriffen wird, wenn diese noch nicht berechnet wurden. Es ist erst möglich auf diese zuzugreifen, wenn sie einem Individuum zugewiesen wurden. Bei der Fitness- und Output-Berechnung erfolgt diese Zuweisung. Anschließend kann auf die Werte zugegriffen werden.

Ebenfalls besitzt die Individual-Klasse eine toString-Methode, die es ermöglicht das komplette Individuum als String auszugeben, damit die Ausgabe von einem Individuum nachvollzogen werden kann.

3.3.3 Network

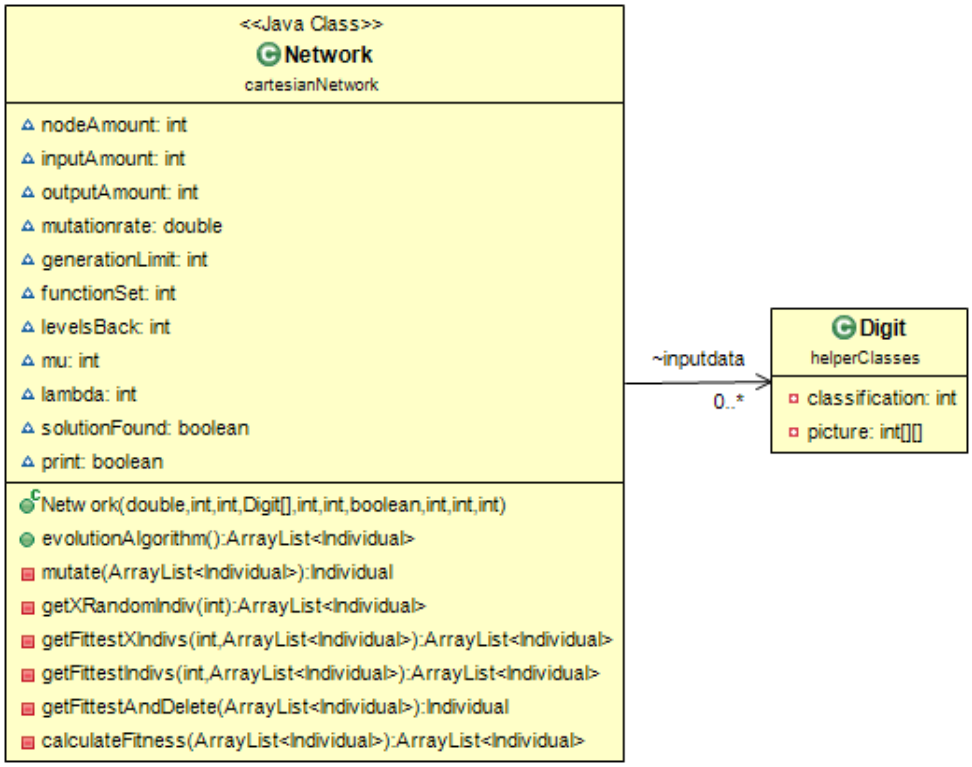


Abbildung 3.5: Klassendiagramm Network

Die Network-Klasse ist verantwortlich für den $(\mu + \lambda)$-Algorithmus. Sie führt den Algorithmus aus und sorgt dafür, die besten Individuen einer Generation auszuwählen. Ihr werden alle wichtigen Argumente übergeben, damit der evolutionäre Algorithmus ausgeführt werden kann.

Attribute und Konstruktor

Die Network-Klasse besitzt alle Parameter, die für die Ausführung von PCGP nötig sind. Diese Parameter umfassen: Mutations-Rate, Anzahl der Knoten, Generations-Limit,

Input-Daten, Output-Anzahl, Funktions-Set, levels-back-Parameter und μ und λ. Außerdem kann angegeben werden, ob während eines Laufes Ausgaben in die Konsole getätigt werden dürfen.

Zusätzlich zu der Speicherung dieser Parameter wird im Konstruktor gespeichert, wie viele Inputs ein Individuum erhält. Dies wird bestimmt, indem das erste von den übergebenen Digit-Objekten zu einem Integer-Array umgeformt wird. Die Länge dieses Arrays gibt die Anzahl der Inputs an. Des Weiteren wird ein boolean gespeichert, welcher auf true gesetzt wird, wenn der Algorithmus eine perfekte Lösung findet. Dies sorgt dafür, dass der Algorithmus, nachdem er eine Lösung gefunden hat, nicht weiter arbeitet.

Der evolutionäre Algorithmus

Die evolutionAlgorithm-Methode führt den $(\mu+\lambda)$-Algorithmus für die angegebenen Werte aus.

Sie beginnt, indem sie mithilfe der getXRandomIndiv-Methode $(\mu + \lambda)$-viele zufällige Individuen erstellt. Mithilfe der getFittesXIndivs-Methoden werden dann die μ Fittesten ausgewählt und als Elternteile gespeichert. Anschließend beginnt die Schleife des $(\mu + \lambda)$-Algorithmus, welche so lange läuft, bis eine perfekte Lösung gefunden wurde oder das Generationslimit erreicht wurde.

Zu Beginn der Schleife wird eine neue Liste von Individual-Objekten erstellt, diese stellt die momentane Generation dar. Zuerst werden die μ-vielen Elternteile hinzugefügt, um anschließend mithilfe der mutate-Methode λ-viele Individual-Objekte als Nachkommen zu erzeugen. Daraufhin werden diese der Generation hinzugefügt, worauf die μ-Fittesten ausgewählt werden und die momentane Generationsnummer erhöht wird. Danach beginnt die Schleife von neuem, bis ein Abbruchkriterium erfüllt wird. Ist die Schleife beendet, werden die μ Fittesten Idividual-Objekte zurückgegeben.

Methoden zur Ausführung des evolutionären Algorithmus

Der $(\mu + \lambda)$-Algorithmus nutzt verschiedene Methoden, um Individuen zu mutieren oder um die Fittesten auszuwählen.

Die mutate-Methode wählt ein zufälliges Individual-Objekt aus μ-vielen aus und mutiert es, dafür nutzt sie die mutation-Methode des ausgewählten Individual-Objektes. Indem $(\mu + \lambda)$ mal der ersten Individual-Konstruktor aufgerufen wird, werden $(\mu + \lambda)$-viele zufällige Individual-Objekte erstellt.

Für die übergebene Liste aus Individual-Objekten wird die Fitness mit der calculateFitness-Methode berechnet. Wenn für alle Individual-Objekte der Fitness-Wert berechnet wurde, wird mithilfe der getFittestIndivs-Methode die besten der $(\mu + \lambda)$-vielen zurückgegeben.

Mit dieser Methode werden die $(\mu + \lambda)$-vielen besten dieser Individuen berechnet. Dies geschieht, indem die Individual-Objekte mit den geringsten Fitness-Werten ausgewählt werden.

Die Fitness wird in der calculateFitness-Methode berechnet.

Damit diese Berechnung schneller vonstatten geht, wird für jedes Individual-Objekt ein Thread erstellt, in welchem die Fitness mit dem Klasse FitnessCalculator berechnet wird. Wenn ein Individual-Objekt bereits einen Fitness-Wert besitzt, wird dieser nicht neu berechnet. Wurden alle Fitness-Werte berechnet, werden diese den Individual-Objekten zugewiesen, welche keine Fitness besitzen. Zum Schluss wird die Liste der Individual-Objekte zurückgegeben.

3.4 Berechnung

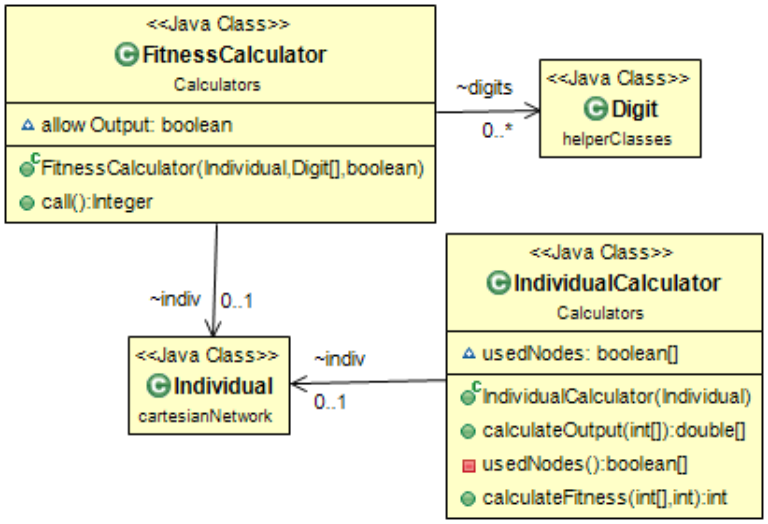


Abbildung 3.6: Klassendiagramm Calculators

Die Klassen Fitness- und Individual-Calculator sind für die Berechnung von Fitness und Outputs in PCGP zuständig. In diesem Abschnitt werden beide Klassen beschrieben und genauer auf ihre Funktionsweise eingegangen.

Fitness in PCGP ist auf das MNIST-Klassifikationsproblem eingestellt. Die Fitness in PCGP drückt aus, wie viele der Klassifikationen falsch gelöst wurden.

3.4.1 FitnessCalculator

Der Fitness Calculator berechnet die Fitness eines Individual-Objektes für alle Objekte einer übergebenen Digit-Liste. Beides wird im Konstruktor übergeben.

Die einzige Methode des FitnessCalculator ist die call-Methode, welche in Threads aufgerufen wird, um die Laufzeit zu verringern. In ihr wird für alle Objekte der Digit-Liste

die Fitness des Individual-Objektes berechnet. Um die gesamte Fitness des Individuums für die übergebene Liste zu bestimmen, werden diese Werte addiert und anschließend wieder zurückgegeben.

Die Fitness wird mithilfe der IndividualCalculator-Klasse bestimmt. Diese gibt für eine korrekt ausgeführte Klassifikation eine 0 zurück, für falsch ausgeführte Klassifikationen wird eine 1 zurückgegeben. So kann sich für den Testdatensatz des MNIST-Datensatzes eine Zahl zwischen 0 und 60.000 als Fitness des Individuums ergeben. Eine Fitness von 60.000 bedeutet, dass alle Bilder falsch klassifiziert wurden, da der Datensatz 60.000 Bilder enthält.

3.4.2 IndividualCalculator

Die IndividualCalculator-Klasse übernimmt die Aufgabe, die Fitness eines Individuums für einen einzelnen Input zu berechnen. Bevor sie dies jedoch tun kann, müssen die aktiven Knoten des Individuums bestimmt werden. Dafür wird im Konstruktor, der ein Individual-Objekt übergeben bekommt, die Liste von aktiven Knoten mithilfe der usedNodes-Methode bestimmt. Danach kann für jeden Input die Fitness mithilfe der CalculateFitness-Methode zurückgegeben werden. Diese bestimmt die Fitness des Individuums für den übergebenen Wert.

Berechnung der aktiven Knoten

Bevor die Berechnung des Outputs des Individual-Objektes geschieht, werden die aktiven Knoten bestimmt. Dies dient dazu, dass nur Knoten, die für die Outputberechnung nötig sind, berechnet werden. So wird zusätzlicher Rechenaufwand gespart und die Laufzeit verringert.

Um zu speichern, welche Node-Objekte des Individuums aktiv sind, wird ein boolean-Array angelegt, welches für jedes Node-Objekt des Individual-Objektes einen Wert enthält. Dieser gibt an, ob der jeweilige Knoten aktiv ist und wird mit allen Werten auf false initialisiert. Zu Beginn werden alle Output-Adressen des Individual-Objektes durchlaufen. Für jede Output-Adresse wird geprüft, ob sie eine Node referenziert, falls sie das nicht tut, wird ein Input adressiert und es ist nichts zu markieren. Wenn eine Node referenziert wird, wird die entsprechende Stelle im boolean-Array auf true gesetzt.

Wurden alle Output-Adressen durchlaufen, wird mit den restlichen Nodes begonnen. Die Liste aus Node-Objekten wird, beginnend von hinten, durchlaufen. Ist eine Node als aktiv markiert, werden Nodes, die als Eingaben addressiert sind, ebenfalls als aktiv markiert. Zuletzt wird das boolean-Array im Individual-Objekt gespeichert, damit dies nicht erneut berechnet werden muss.

Outputberechnung

Die calculateOutput-Methode berechnet zu einem Input, welcher als Argument übergeben wurde, den Output des Individual-Objektes. Um die Ergebnisse der einzelnen Nodes zu speichern und manche Nodes nicht mehrfach zu betrachten, wird ein Ausgabe-Array angelegt. Dieses ist genauso lang wie die Anzahl der Node-Objekt-Liste des Individuums. Es enthält an jeder Stelle den Output der Node zu dem gegebenen Input, sofern diese Node aktiv ist.

Zu Beginn wird die Liste von Node-Objekten durchlaufen. Wenn eine Node aktiv ist, wird für sie der Input berechnet. Dafür nutzt sie entweder die übergebenen Inputs oder andere Nodes, deren Ergebnisse in dem erstellten Array gespeichert werden. Hat sie alle nötigen Eingaben, wird mithilfe der Functions-Klasse die Ausgabe berechnet. Anschließend wird die Ausgabe an der entsprechenden Stelle im Ausgabe-Array gespeichert. Dadurch werden zum einen nur aktive Nodes betrachtet und zum anderen wird jede Node nur einmal betrachtet.

Wurden für alle Nodes die Ausgaben berechnet, werden die Output-Adressen des Individuums durchlaufen und für jede Output-Adresse wird der Wert, den sie adressiert, gespeichert und ausgegeben. Dadurch wird der Output eines Individual-Objektes für einen übergebenen Input erhalten.

Fitnessberechnung

Die Fitnessberechnung mit der calculateFitness-Methode bestimmt für einen gegebenen Input die Fitness des Individuums. Sie nutzt die calculateOutput-Methode, um zu Beginn den Output für den gegebenen Input zu erhalten. Da PCGP auf einem Klassifizierungsproblem arbeitet, wird in der calculateFitness-Methode der Output des Individual-Objektes mit der Klassifizierung des Inputs verglichen. In PCGP wird auf zwei verschiedene Arten mit verschiedener Output-Anzahl umgegangen.

Hat das Individual-Objekt nur einen Output, so wird die Klassifizierung mit dem Output verglichen. Sind diese gleich und die Klassifizierung somit korrekt, wird eine null zurückgegeben. Wenn diese unterschiedlich sind, wird eine eins zurückgegeben.

Besitzt das Individual-Objekt vier Outputs, so wird die Zahl als Binärzahl interpretiert. Entspricht der Wert der Binärzahl mit der gewünschten Klassifizierung, so wird eine null und andernfalls eine eins als Fitness zurückgegeben.

3.5 Rahmenklassen

Die Klassen, die in diesem Abschnitt besprochen werden, sind nicht Teil von CGP. Sie dienen in PCGP der Bedienbarkeit und der Interpretation der Ergebnisse. Der CGPNetHand-

ler erleichtert die Bedienbarkeit von PCGP, wobei der StatisticsCalculator die Ergebnisse eines Laufes strukturiert darstellt.

3.5.1 CGPNetHandler

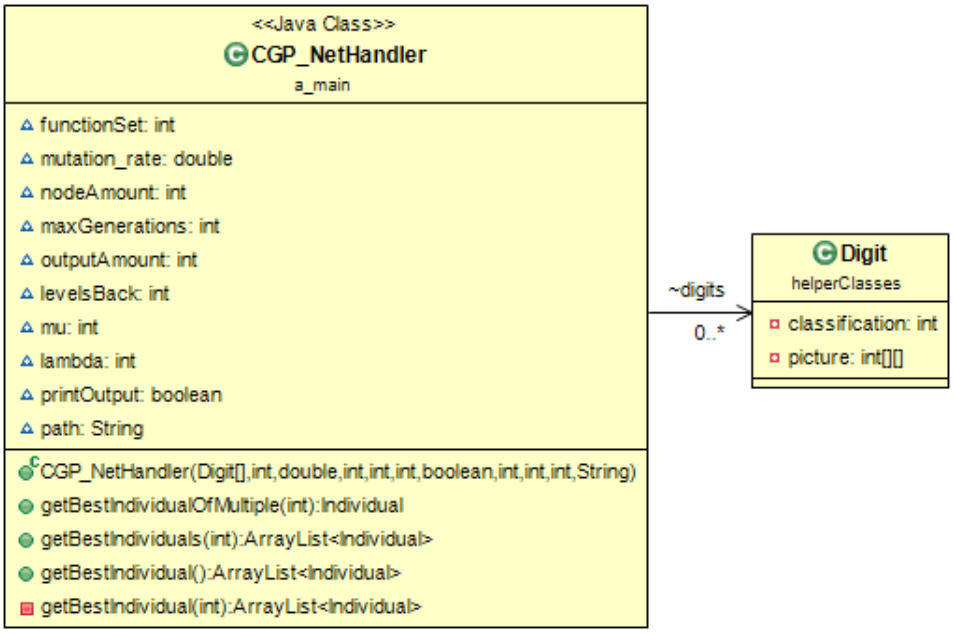


Abbildung 3.7: Klassendiagramm CGPNetHandler

Die CGPNetHandler-Klasse erleichtert die Bedienung von PCGP. Sie bekommt im Konstruktor alle wichtigen Argumente zur Ausführung von PCGP überreicht, die schon in Abschnitt 3.3.3 besprochen wurden. Zusätzlich kann ihr ein Dateipfad übergeben werden, der bestimmt, wo die besten Individual-Objekte gespeichert werden. Sie besitzt drei verschiedene Arten von Methoden, welche alle dafür zuständig sind, mit PCGP das beste oder mehrere beste Individual-Objekte zurückzugeben.

Die GetBestIndividual-Methode führt PCGP für die dem Konstruktor übergebenen Argumente aus und speichert die von der Network-Klasse zurückgegebenen Individual-Objekte ab.

Hingegen führt die GetBestIndividuals-Methode die BetBestIndividual-Methode beliebig oft aus. Sie bekommt die Anzahl der Durchläufe als Argument übergeben. Für jeden Durchlauf werden die fittesten Individual-Objekte, die von der GetBestIndividual-Methode zurückgegeben werden, in einer Liste gespeichert. Daraufhin werden für diese mithilfe des StatisticsCalculator verschiedene Statistiken berechnet, damit die Ergebnisse besser interpretiert werden können. Diese Statistiken werden unter dem übergebenen Pfad abgespeichert, damit später darauf zugegriffen werden kann. Zum Schluss gibt die Methode die Liste aller Individual-Objekte zurück, welche gespeichert wurden.

Die getBestIndividualOfMultiple-Methode ruft die vorher beschriebene Methode auf. Sie durchläuft die Liste der zurückgegebenen Individual-Objekte und gibt das Objekt mit dem besten Fitness-Wert zurück. So wird das beste Individuum aus mehreren Läufen von PCGP zurückgegeben.

3.5.2 StatisticsCalculator

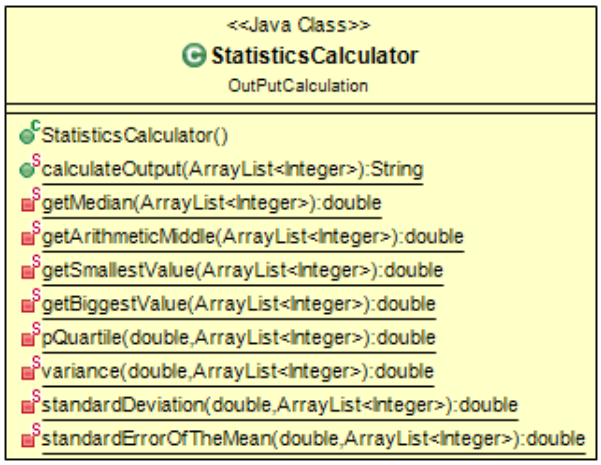


Abbildung 3.8: Klassendiagramm Statistiken

Mit der StatisticsCalculator-Klasse wird die Liste von übergebenen Integern zu aussagekräftigen Statistiken berechnet. Sie wird von der CGPNetHandler-Klasse verwendet, um zu einer Anzahl von Läufen Statistiken zu allen Läufen auszugeben. Zuerst werden alle Statistiken berechnet, um sie anschließend als String zur weiteren Verarbeitung zurückzugeben.

Die berechneten Statistiken umfassen: Median, arihtmetisches Mittel, Standard-Fehler des arithmetischen Mittels, größter und kleinster Wert, oberes und unteres Quartil und die Standardabweichung.

3.6 Speichern und Laden

Im folgendem Abschnitt werden die Klassen (CSVReader, SaveAndLoadIndividuals) beschrieben, welche zur Datenspeicherung und zum Laden der Daten genutzt werden.

Die CSVReader-Klasse dient dazu den MNIST-Datensatz einzulesen. Entweder liest sie den Trainings- oder den Test-Datensatz ein, was abhängig davon ist, welche Methode aufgerufen wurde. Des Weiteren speichert sie die eingelesenen Bilder und Klassifikationen in Digit-Objekten, um diese als Liste zurückzugeben.

Die SaveAndLoadIndividuals-Klasse ist dafür zuständig, Individual-Objekte zu speichern und zu laden. Mit den saveIndividual-Methoden speichert die Klasse ein oder mehrere Individual-Objekte. Zum Laden eines Individual-Objektes wird die readIndividual-

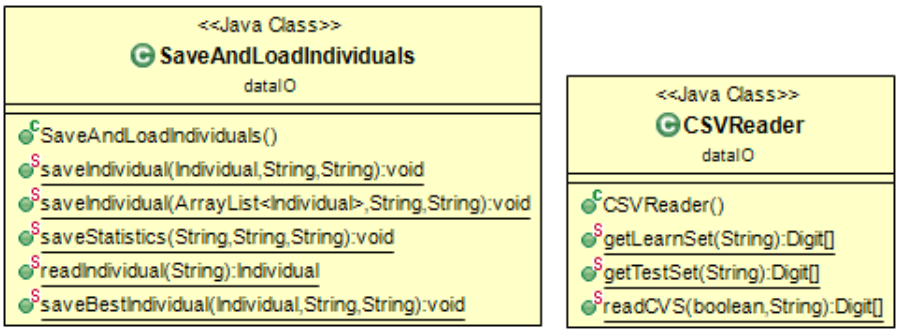


Abbildung 3.9: Klassendiagramm Speichern und Laden

Methode verwendet. Soll mit der saveBestIndividual-Methode ein Individuum gespeichert werden, wird zuerst geprüft, ob das unter dem Pfad gespeicherte Objekt eine größere Fitness besitzt. So wird sichergestellt, dass nur das Individual mit der besten Fitness abgespeichert wird.

Außerdem besitzt die SaveAndLoadIndividual-Klasse noch die saveStatistics-Methode, die einen String unter dem gegebenen Pfad abspeichert. PCGP nutzt diese, um die Statistiken eines Durchlaufes zu speichern.

3.7 Klassifikator

Mit dem CalssificatorClassGenerator kann aus einem Individuum eine Klassifikator-Klasse erstellt werden. Eine Klassifikator-Klasse kann eigenständig genutzt werden, um die Arbeitsweise eines Individuums zu nachzuvollziehen. Diese arbeitet auf einem Input genauso wie das Individuum, aus dem sie erstellt wurde.

Im Folgenden wird zuerst die ClassificatorClassGenerator-Klasse vorgestellt und anschließend wird ein Beispiel-Classificator vorgestellt.

3.7.1 ClassifitcatorClassGenerator

Die ClassificatorClassGenerator-Klasse nutzt die JavaPoet-API, um Klassen zu erstellen. Als Vorlage wurde der Beispiel-Code, welcher die Verwendung von JavaPoet erklärt, verwendet. [1]

Um einen Klassifikator aus einem Individuum zu generieren, kann entweder ein Individual-Objekt oder ein Pfad, auf dem ein Individuum gespeichert ist, der Methode generateCode übergeben werden. Außerdem wird der Pfad benötigt, auf dem der generierte Klassifikator gespeichert werden soll. Zu Beginn wird der Konstruktor erstellt, welcher ein Array generiert, in dem alle Outputs der Knoten gespeichert werden. Dieser ist genauso lang, wie die Anzahl der Knoten in dem ursprünglichen Individuum.

Wurde der Konstruktor erstellt, wird mit der Erstellung der restlichen Methoden begonnen. Abhängig vom verwendeten Funktions-Set des Individuum wird eine Methode er-

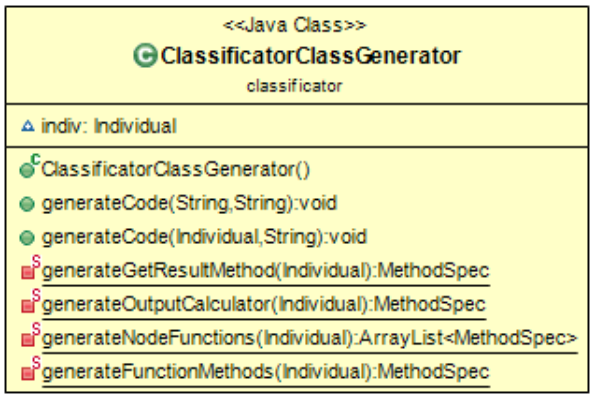


Abbildung 3.10: Klassendiagramm Klassifikator-Generator

stellt, die dieses widerspiegelt. Sie enthält alle Funktionen des Funktionen-Sets und kann aus zwei Eingaben und einer Funktions-Nummer eine Ausgabe erzeugen. Diese Methode wird von den verschiedenen Node-Methoden verwendet. Für jede aktive Node des Individuums wird im Klassifikator eine Node-Methode erstellt, die diese aktive Node repräsentiert. Sie holt sich alle benötigten Eingaben entweder aus dem Input oder den bereits berechneten Funktionen. Daraufhin speichert sie ihre Ausgabe in dem Array, in dem alle Outputs gespeichert werden. So wird, wenn eine Node-Methode aufgerufen wird, ihre Ausgabe berechnet und gespeichert. Für das Aufrufen der verschiedenen Node-Methoden ist die outputCalculator-Methode zuständig. Sie ruft jede einzelne Node-Methode in der Reihenfolge auf, in der die Node-Objekte im Genotypen auftreten. Damit ist sichergestellt, dass Nodes, die im Genotyp weiter hinten liegen, die Ausgaben vorheriger Nodes bereitstehen.

Zuletzt wird die getResult-Methode erstellt. Diese soll zu einem Input den jeweiligen Output ausgeben. Die Methode beginnt damit, anhand des Inputs alle Outputs mithilfe der outputCalculator-Methode zu berechnen. Danach wird ein Ausgabe-Array erstellt, welches so lang ist, wie die Anzahl der Outputs des Individuums. Gefüllt wird dieses, indem für jede Adresse der entsprechende Output aus dem gespeicherten Output-Array geholt wird. Dieses Ausgabe-Array wird zum Schluss als Ergebnis des Individuums ausgegeben.

3.7.2 Beispiel-Klassifikator

In diesem Abschnitt wird ein Beispiel-Klassifikator vorgestellt. Der Code des Klassifikator ist in Abbildung 3.12 aufgezeigt. Dieser Klassifikator wurde aus einem Individuum mit einem Fitness-Wert von 42652 auf dem Trainingsdatensatz erstellt. Er benutzt das Funktions-Set 2. Er besitzt sieben aktive Knoten, die als Methoden in der Klasse aufgelistet sind. Die Methode function, die das Funktions-Set enthält, ist im Beispiel eingeklappt. Aus diesem Klassifikator ist erkennbar, dass der erste aktive Knoten den zweiten und den 190ten Input nutzt und auf ihnen die Operation ROUND ausführt.

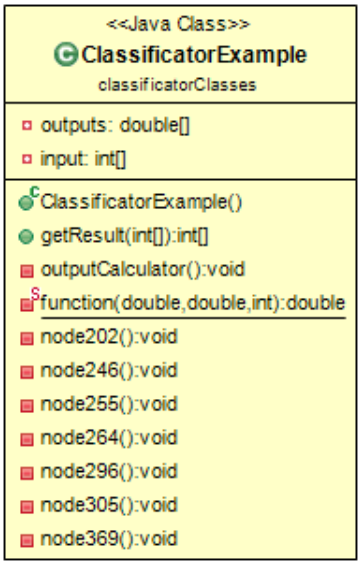


Abbildung 3.11: Klassendiagramm Klassifikator

```
 3 public final class ClassificatorExample {
 4     private double[] outputs;
 5
 6     private int[] input;
 7
 8     public ClassificatorExample() {
 9         this.outputs = new double[1196];
10     }
11
12     public int[] getResult(int[] input) {
13         this.input = input;
14
15         outputCalculator();
16
17         double[] output = new double[1];
18         output[0] = outputs[369];
19
20         // cast back to int array
21         int[] toReturn = new int[output.length];
22         for (int i = 0; i < output.length; i++) {
23             toReturn[i] = (int) Math.round(output[i]);
24         }
25         return toReturn;
26     }
27
28     private void outputCalculator() {
29         node202();
30         node246();
31         node255();
32         node264();
33         node296();
34         node305();
35         node369();
36     }
37
38     private static double function(double in1, double in2, int functionNr) {
91
92     private void node202() {
93         double node1 = (double) input[189];
94         double node2 = (double) input[1];
95         outputs[202] = (function(node1, node2, 18));
96     }
97
98     private void node246() {
99         double node1 = (double) input[105];
100        double node2 = (double) input[102];
101        outputs[246] = (function(node1, node2, 4));
102    }
103
104    private void node255() {
105        double node1 = (double) input[188];
106        double node2 = (double) input[3];
107        outputs[255] = (function(node1, node2, 14));
108    }
```

Abbildung 3.12: Code eines Beispiel-Classificators

Kapitel 4

Ergebnisse PCGP

In diesem Kapitel wird PCGP verwendet, welches im vorherigen Kapitel vorgestellt wurde. Ziel von PCGP ist es, das Problem der handschriftlichen Ziffernerkennung zu lösen. Zuerst werden die getroffenen Vorüberlegungen beschrieben, worauf im Anschluss die Testläufe vorgestellt werden. Um die Interpretation der Ergebnisse nachvollziehen zu können, wurden nach jedem Testlauf verschiedene Statistiken aufgelistet. Diese Statisiken umfassen Median, arithmetisches Mittel, Standardfehler des arithmetischen Mittels, Spannweite, oberes und unteres Quartil sowie die Standardabweichung. Die ausgewählten Statistiken dienen als experimentelles Setup, mit dessen Hilfe sich möglichst viele Aussagen über die durchgeführten Tests treffen lassen sollen.

Mit dem Ziel, dass am Ende dieses Abschnittes die Funktionsweisen einzelner Parameter klarer geworden sind, werden Probleme, welche aufgetreten sind, erläutert. Des Weiteren soll aufgezeigt werden, inwiefern sich CGP eignet, um das gegebene Klassifikationsproblem zu lösen.

4.1 Vorüberlegungen

Vor Beginn der ersten Tests können bereits Schwierigkeiten ausgemacht werden, die bei der Klassifikation von handschriftlichen Ziffern des MNIST-Datensatzes auftreten könnten. So treten bei dem verwendeten Datensatz Übergangsschwierigkeiten auf, die einen Übergang zu besseren Fitness-Werten erschweren. Ein Individuum, das nur eine konstante Zahl zwischen null und neun ausgibt, hat bereits eine Genauigkeit von etwa 10%, da jede Ziffer in dem Datensatz zu ungefähr 10% vorkommt. So würde ein Individuum, welches nur eine Zahl, zum Beispiel die Zahl Eins raten würde, maximal 6.742 Bilder richtig klassifizieren, da die Zahl Eins in dem Trainingsdatensatz 6.742 mal vorkommt. Insgesamt hat der Traingsdatensatz 60.000 Bilder, weshalb hier eine Fitness von 53.258 erreicht werden würde.

4.1.1 Beispiel. Um diesen Sachverhalt genauer zu veranschaulichen, wird er im Folgendem an einem Beispiel erklärt:

Wir betrachten ein Individuum, welches beispielsweise nur einen Pixel des Eingabe-Bildes betrachtet. Ergibt sich ein Grauwert von 0, welches bedeuten würde, dass dort das Bild an dieser Stelle weiß wäre, wird von dem Individuum die Zahl Eins klassifiziert. Wird jedoch ein höherer Grauwert erreicht, wird festgelegt, dass dieses Bild eine Acht darstellt. Damit ist sichergestellt, dass ein gewählter Pixel, der wahrscheinlich nicht von einer Eins belegt wird, wahrscheinlich von einer Acht belegt wird. So errät das Individuum bereits zwei Zahlen wahrscheinlich richtig.

Diese Tatsache erschwert den Übergang zu höheren Fitness-Werten, da Individuen, die mehr Zahlen erkennen sollen, einen größeren Teil des Bildes betrachten müssen. Um in die nächste Generation übertreten zu können, müssen diese Individuen mehr Bilder richtig klassifizieren, als die Individuen, welche nur einen Pixel betrachten. Individuen, welche nur einen Pixel des Bildes betrachten, können wie grade beschrieben, eine Erfolgsrate von ungefähr 20% besitzen. Demnach müssen Individuen, welche einen größeren Teil des Bildes betrachten, diese Anfangshürde übersteigen. Ein Individuum, ist erst dann besser als sein Elternteil, wenn es mindestens ein Bild mehr richtig klassifiziert. Damit mehr Bilder richtig klassifiziert werden, sind viele Veränderungen nötig. Zum einen müssen mehr Knoten benutzt werden, damit ein größerer Teil des Bildes erkannt wird. Außerdem müssen diese Eingaben richtig verarbeitet werden, um das Bild richtig zu klassifizieren. Diese Veränderungen erfolgen nicht gezielt, sondern per Zufall, weshalb blind mutiert wird, da sonst keine Änderung am Fitness-Wert geschehen würde.

Die Steigerung der allgemeinen Fitness der Population geschieht nicht zwingend kontinuierlich. Es werden Sprünge vorhanden sein, wenn ein neues fitteres Individuum mehr Bilder klassifizieren kann, als sein Elternteil. Dies ist der Fall, wenn ein Individuum eine Zahl mehr erkennt als sein Elternteil, wodurch dass Individuum eine circa 10%-ige bessere Klassifikationsrate als sein Elternteil besitzt. Des Weiteren ist, wenn das Elternteil nur wenige aktive Knoten nutzt, die Wahrscheinlichkeit einen Knoten zu wählen, welcher aktiv ist, sehr gering, da der Großteil des Elternteils nur aus nicht-aktiven Knoten besteht. Das Prinzip der neutralen Mutation macht es leichter für Populationen sich zu verändern, da die Elternteile nicht stagnieren, jedoch ist das Problem des erschwerten Überganges zu besseren Fintess-Werten immer noch vorhanden.

4.1.2 Beispiel. Um dies verständlicher zu machen, wird dies im folgendem anhand eines Beispiels erklärt: Wir nehmen uns ein Individuum, dass das MNIST-Klassifikationsproblem lösen soll als Elternteil. Das Individuum besitzt keine aktiven Knoten. Der Output des Individuums ist lediglich ein weitergereichter Pixel-Wert des Input-Bildes.

Nehmen wir an, dass das Individuum den Wert eines Pixels weiterreicht, der in einer Ecke des Bildes liegt. Dieser Pixel wird sehr wahrscheinlich immer den Grauwert null be-

sitzen, da die Ziffern im Bild zentriert sind. Wird also immer der Wert null ausgegeben besitzt das Individuum eine ungefähre 10%ige Erfolgswahrscheinlichkeit in der Klassifikation, da in circa 10% der Fälle eine Null auf dem Bild ist. Das Elternteil besitzt also eine Fitness von ungefähr 54.000.

Um daraus ein fitteres Individuum zu erschaffen, sind mehrere Veränderungen notwendig. Zum einen muss das Individuum so mutiert werden, dass es einen anderen Knoten zur Ausgabeberechnung nutzt und nicht einen Input. Es muss also das Output-Gen mutiert werden. Des Weiteren muss der nun referenzierte Knoten für eine größere Erfolgswahrscheinlichkeit sorgen. Da der Knoten, der nun von der Ausgabe genutzt wird, vorher nicht aktiv war, besitzt dieser zufällige Eingaben und wurde nicht fitnessgünstig mutiert.

Dieser Knoten muss, ohne vorher mit dem Fitness-Wert in Verbindung gebracht worden zu sein, eine Fitness ausgeben, die größer als die des Elternteiles ist. Da das Elternteil jedoch schon eine Fitness von ungefähr 54.000 besitzt, sind die Anforderungen an das Kind, welche übertroffen werden müssen, hoch. Resultiert aus dem neu gewählten Knoten eine Fitnessverschlechterung wird dieses Individuum verworfen. Da der Knoten, wie bereits erwähnt, zufällig mutiert ist, ist dies sehr wahrscheinlich. Wenn der Knoten eine bessere Fitness besitzt als sein Elternteil, wird er als neues Elternteil gewählt, was zum Beispiel der Fall ist, wenn er eine Zahl mehr erkennt und demnach eine Fitness von ungefähr 48.000 besitzt (20% Klassifikationsgenauigkeit). Da dies jedoch eher zufällig geschieht ist die Wahrscheinlichkeit hierfür eher gering.

Daraus kann geschlussfolgert werden, dass in diesem Problem, zumindest dann, wenn die Berechung innerhalb eines Individuums mit sehr wenigen Knoten vollführt wird, sich die Fitness eher in Sprüngen verändert.

Dieses Problem erschwert es, eine Lösung zu finden. Außerdem entsteht ein weiteres Problem, welches die Lösungsfindung erschwert.

Wie bereits in Abschnitt 1.3 erwähnt, besteht der Datensatz aus Bildern mit 28x28 Pixeln, was 784 Inputs sind, die ein Individuum verarbeiten muss. Diese hohe Anzahl von Inputs erschweren es einem Indivdiuum, die Pixel zu wählen, die für die Klassifikation von Bedeutung sind.

Jeder Knoten und alle Outputs besitzen die Wahrscheinlichkeit einen Input als Eingabeadresse zu wählen. Mit einer so hohen Anzahl an Inputs ist die Wahrscheinlichkeit, einen Input zu wählen, viel höher. Somit ist die Wahrscheinlichkeit geringer, weitere Knoten für die Eingabeverarbeitung zu wählen, was die Problemlösung erschwert.

Es wird versucht, eine möglichst optimale Fitness zu erzielen. Da die Möglichkeiten CGP zu varrieren sehr groß sind, wird hier versucht, den vorerst besten Wert für einzelne Parameter zu finden. Es soll für jeden Parameter geprüft werden, welche Wirkung er auf die Fitness-Entwicklung hat und inwiefern dieser hilft, ein besseres Ergebnis zu bekommen. Da diese Arbeit nicht alle Varianten abdecken kann, werden im Folgenden einzelne

Node-Anzahl = 1000; Max Generation = 2000; Mutationsrate = 3%;
Levels-Back = -1; Strategie = (1+4); OutputAnzahl = 1;

Funktions-Set:	1	2	3	4
Median	47327,5	47670	48916,5	53292
Arithm. Mittel	47576,56	47411,63	49331	52324,03
Standardfehler	394,05	284,92	421,78	365,83
Kleinster Wert	45265	43356	45589	47415
Größter Wert	54058	51075	54058	54058
Unteres Quartil	45921	47446	47772	51457
Oberes Quartil	48004	48335	49456	53825
Standardabw.	2158,29	1560,59	2310,17	2003,72

Abbildung 4.1: Ergebnisse der Funktions-Sets-Versuche

Parameter nach der Vorgabe von Julian Miller genutzt [18]. Anschließend wird geprüft, wie diese Parameter verändert werden können, um eine bessere problembezogene Fitness zu erhalten.

4.2 Testen ohne vorherige Bildbearbeitung

In diesem Abschnitt werden die durchgeführten Läufe von PCGP behandelt. Zu jedem Versuch werden Überlegungen gemacht und Schlussfolgerungen gezogen, um die dargestellten Statistiken auszuwerten. Zu jedem einzelnen Versuch wurden 60 Durchläufe durchgeführt, um eine Aussagekraft zu gewährleisten.

4.2.1 Testen der verschiedenen Funktions-Sets

Zu Beginn des Testens wurde sich die Frage gestellt, inwiefern welche Funktions-Sets für das Klassifikationsproblem von Nutzen sind. Es wurden die ersten vier Funktions-Sets getestet, die in Abbildung 3.2 aufgelistet sind.

Diese vier verschiedenen Funktions-Sets sollen nun auf ihre Anwendbarkeit auf das gegebene Problem geprüft werden. Als Grundlage wurde eine Knoten-Anzahl von 1000 Knoten gewählt, da so mehr Knoten als Inputs vorhanden sind.

Des Weiteren wurde eine maximale Generationsanzahl von 2000 gewählt, da diese nicht zu groß sein sollte, um möglichst viele separate Durchläufe zu ermöglichen. Sie sollte jedoch nicht zu klein ausfallen, damit die Generation genug Zeit besitzt, ein Individuum zu enthalten, welches bessere Fitness-Werte erreicht. Ebenfalls wurde eine (1+4)-Strategie gewählt, da diese Standard in CGP ist und nach Miller [18] wurde eine Mutations-Rate von 3% gewählt. Ein levels-back-Parameter wurde nicht festgelegt.

Die Durchläufe mit den verschiedenen Funktions-Sets ergaben die Ergebnisse in 4.1.

Es ist erkennbar, dass das Funktions-Set 4 gegenüber den anderen Funktions-Sets schlechtere Ergebnisse erzielt. Mit einer durchschnittlichen Fitness mit etwas über 52.000 ist dieser Ansatz kaum besser als ein zufälliges Raten der Klassifikation. Die anderen Funktions-Sets schneiden wesentlich besser ab, weshalb Funktions-Set 4 nicht weiter betrachtet wird. Funktions-Set 3 schneidet im Vergleich zu Funktions-Set 4 besser ab, im Vergleich zu Funktions-Set 1 und 2 jedoch schlechter, weshalb dieses im weiteren Verlauf ebenfalls nicht weiter betrachtet wird.

Die Ergebnisse der Funktions-Sets 1 und 2 liegen sehr nah beieinander. So zeigt sich aus den Ergebnissen des Funktions-Sets 1, dass sich einfache mathematische Operationen positiv auf die Fitness dieses Problems auswirken. Mit der Inklusion der Funktionen, die im Funktions-Set 2 zusätzlich dazu genommen werden, ergibt sich eine weitere positive Wirkung. Das Funktions-Set 2 erzielte insgesamt den besten Fitness-Wert. Aufgrund der geringeren Streuung und der kleineren Minimal-Werte wird im Folgenden mit dem Funktions-Set 2 weitergearbeitet.

Für eine bessere Fitness könnten andere Funktions-Sets nützlicher sein. Da jedoch zu CGP nicht viele Arbeiten existieren, ist die Wahl auf vorher definierte Funktions-Sets beschränkt. Um eventuell bessere Fitness-Werte zu erhalten, könnte ein eigens für das Problem entworfenes Funktions-Set von Nutzen sein. Da die Erstellung eines solchen Funktions-Sets sehr aufwendig ist, wird das beste Funktions-Set aus den vorher getesteten Funktions-Sets genutzt. Die Erstellung eines neuen Funktions-Sets entspricht nicht dem Ziel dieser Arbeit. Das Ziel dieser Arbeit soll es sein, die ersten Schritte für die Problemlösung zu ebnen.

4.2.2 Testen der verschiedenen Mutations-Raten und Knotenanzahl

Die Mutations-Rate ist ein weiterer wichtiger Faktor, welcher zu verbesserten Ergebnissen führen kann. Ist die Mutationsrate zu klein, sind die Änderungen am Genotypen pro Generation zu gering und es werden keine bedeutenden Änderungen pro Generation erzielt. Fällt die Rate zu groß aus, vollführt sich die Evolution mehr und mehr zufällig, wodurch kaum zielgerichtete Lösungen gefunden werden können. Der Vorteil einer hohen Mutations-Rate ist, dass lokale Übergangshürden, wie sie in Abschnitt 4.1 beschrieben werden, überwunden werden können.

Im Folgenden wurden verschiedene Mutations-Raten ausführlich getestet, um zu prüfen, ob und inwiefern sich diese auf die Fitness ausüben. Es wurden Mutations-Raten von 1% bis 20% getestet. Die weiteren Parameter wurden vom vorherigen Versuch übernommen und es wurde das Funktions-Set 2 gewählt. Die Ergebnisse sind in 4.2 aufgelistet.

In diesen Versuchen wird deutlich, dass die Mutations-Rate keinen großen Einfluss auf den Fitnesswert besitzt. Alle Versuche erreichen ähnliche Ergebnisse. Ebenfalls weichen die Ergebnisse nicht sonderlich voneinander ab. Um genauere Aussagen treffen zu können

Funktions-Set = 2; Node-Anzahl = 1000; Max Generation = 2000;
Levels-Back = -1; Strategie = (1+4); OutputAnzahl = 1;

Mutationsrate:	1%	2%	3%	4%	5%	6%	8%
Median	48373,5	47783,5	47711,5	47589	47602	47195	47570
Arithm. Mittel	48415,84	47857,7	47847,73	47152,27	47269,1	46263,6	47267,07
Standardfehler	205,16	205,77	220,02	269,47	232,6	335,63	216,63
Kleinster Wert	46557	44732	45416	43179	43619	42183	43921
Größter Wert	51231	51201	51154	49377	48822	48727	48341
Unteres Quartil	47735	47589	47434	47200	47356	45079	47048
Oberes Quartil	48563	48383	48319	48260	48124	47525	48131
Standardabw.	1123,71	1127,04	1205,11	1475,96	1274,01	1838,32	1186,55

Mutationsrate:	10%	12%	14%	16%	18%	20%
Median	47488	47339	47470,5	47405	47408	47248,5
Arithm. Mittel	46700,77	46444,57	46951,17	46641,23	46721,7	46774,07
Standardfehler	330,47	319,29	178,37	261,63	250,49	358,98
Kleinster Wert	41302	42818	45159	42032	43313	42627
Größter Wert	48360	48316	48316	48316	48335	50220
Unteres Quartil	46075	45824	45958	45982	45966	45485
Oberes Quartil	47676	47589	47552	47552	47552	48225
Standardabw.	1810,06	1748,8	976,98	1433,01	1371,96	1966,2

Abbildung 4.2: Ergebnisse der Mutations-Raten-Versuche

Funktions-Set = 2; Max Generation = 2000; Mutationsrate = 6%;
Levels-Back = -1; Strategie = (1+4); OutputAnzahl = 1;

Node-Anzahl:	1000	100	50
Median	47195	48319	48340,5
Arithm. Mittel	46263,6	48758,87	49267,1
Standardfehler	335,63	290,23	331,41
Kleinster Wert	42183	46059	47552
Größter Wert	47827	54058	54062
Unteres Quartil	45079	47827	48319
Oberes Quartil	47525	48520	51075
Standardabw.	1838,32	1589,63	1815,23

Abbildung 4.3: Ergebnisse der Knoten-Anzahl-Versuche

müssten zahlreiche Versuche gemacht werden. Insgesamt zeigt sich, dass die Mutations-Rate in diesem Stadium nicht von Bedeutung ist. Gründe dafür könnten sein, dass zum einen ein schnelles lokales Maximum erreicht werden kann, welches schwer zu überwinden ist. Zum anderen kann der Genotyp nur sehr wenige aktive Knoten enthalten, weshalb die Mutation keine aktiven Knoten beeinflusst. Wie bereits in Abschnitt 4.1 dargestellt wurde, waren diese Probleme zu erwarten. Eine höhere Mutations-Rate, um diese Maxima zu überwinden und aktive Knoten zu mutieren, würde jedoch eher zu zufälligem Raten führen. Des Weiteren könnte auch die Anzahl der Eingabe-Adressen ein Problem sein, da diese die ersten 784 Adressen belegen.

Um diesen Problemen vorzubeugen, wurde zunächst die Anzahl der zu benutzenden Knoten getestet. Diese darf nicht zu gering sein, da sonst der Genotyp keine komplexen Operationen durchführen kann. Sie darf jedoch auch nicht zu groß sein, da sonst die Ausführungszeit unter der hohen Knotenanzahl leidet. Es wurden Knoten-Anzahlen von 1000, 100 und 50 getestet. Des Weiteren wurde mit einer Mutations-Rate von 6% getestet, da mit dieser die besten Ergebnisse erreicht wurden. Die Ergebnisse sind in Abbildung 4.3 aufgelistet.

Es ist erkennbar, dass mit einer höheren Knotenanzahl bessere Fitness-Werte erzielt werden. Dies lässt sich damit erklären, dass bei mehreren Knoten mehr Möglichkeiten existieren, um komplexe Berechnungen durchzuführen. Durch die erhöhte Knotenanzahl ist es wahrscheinlicher, dass Knoten, welche im Genotypen später auftreten, einen anderen Knoten als Eingabe wählen. Die Wahrscheinlichkeit einen Input als Eingabe zu nutzen, sinkt je weiter der Knoten im Genotypen von der Eingabe entfernt liegt.

Bei diesen Tests fällt weiterhin auf, dass Individuen mit einer geringen Knoten-Anzahl meistens keinen oder nur einen aktiven Knoten besitzen. So war bei Individuen, welche 1000 Knoten besitzen, oft kein aktiver Knoten vorhanden. Häufig war nur ein Knoten

Funktions-Set = 2; Node-Anzahl = 1000; Max Generation = 2000;
Mutationsrate = 6%; Strategie = (1+4); OutputAnzahl = 1;

Levels-Back:	-1	5	300	784
Median	46483,5	54058	48199	48905
Arithm. Mittel	45964,43	5267,33	47977	49464,03
Standardfehler	380,28	373,15	319,25	268,04
Kleinster Wert	41183	47827	43432	47538
Größter Wert	49377	54062	51194	51207
Unteres Quartil	44380	51090	47453	48255
Oberes Quartil	47542	54058	48909	51090
Standardabw.	2082,9	2043,85	1748,62	1468,12

Abbildung 4.4: Ergebnisse der levels-back-Versuche

aktiv, selten waren zwei Knoten aktiv (0,2% aktive Knoten). Dabei ist zu beobachten, dass Individuen mit mehr aktiven Knoten generell eine bessere Fitness besitzen, als Individuen mit weniger. So sind die Individuen mit 1000 Knoten und 2 aktiven Knoten die Individuen, die die besten Fitness-Werte besitzen. Weshalb im Folgenden versucht wird, die Anzahl der aktiven Knoten zu erhöhen, damit eine wahrscheinlich bessere Fitness erreicht wird.

4.2.3 Testen, um Konnektivität zu erhöhen

Um die Konnektivität von Knoten zu beeinflussen, wird der levels-back-Parameter benötigt, welcher bereits in Abschnitt 2.2 vorgestellt wurde. Die Funktion des levels-back-Parameter ist es, dafür zu sorgen, dass Knoten keine anderen Knoten verwenden, welche im Genotypen zu weit von ihnen entfernt sind. In den nachfolgenden Tests wurde versucht, durch Änderungen des levels-back-Parameters prozentual mehr aktive Knoten zu erreichen. Ziel dabei war es, dass möglichst viele aktive Knoten benutzt werden.

Getestet wurde dies mit den zuvor benutzten Werten, wobei nur der levels-back-Parameter verändert wurde. Am Ende soll geschaut werden, welche Auswirkungen dies auf die Fitness und die aktiven Knoten hat. Die Ergebnisse sind in Abbildung 4.4 aufgelistet.

Für die Vergleichbarkeit wurde ein Versuch ohne levels-back-Parameter durchgeführt. Um eine möglichst große Spannweite abzudecken, wurden die restlichen Werte zwischen fünf, 300 und 784 gewählt. Dabei wurde 784 als größter Wert gewählt, da ein Eingabe-Bild genauso viele Werte enthält.

Bereits am Anfang lassen sich Beobachtungen machen. So sinkt die durchschnittliche Fitness, je größer der levels-back-Parameter ist. Je niedriger der levels-back-Parameter ist, desto wahrscheinlicher ist es, einen Knoten mit einem Input zu verbinden. Demnach hat ein Knoten mit mindestens fünf Knoten vor ihm, einem levels-back-Parameter von fünf

Funktions-Set = 2; Node-Anzahl = 1000; Max Generation = 8000;
Mutationsrate = 6%; Strategie = (1+4); OutputAnzahl = 1;

Levels-Back:	-1	200	300	500
Median	44595	47481	46809,5	47489,5
Arithm. Mittel	45124,17	46669,9	45966,43	47520,8
Standardfehler	435,3	362,78	340	117,21
Kleinster Wert	41178	42406	41108	45866
Größter Wert	48316	51075	48301	48324
Unteres Quartil	43093	45637	44392	47421
Oberes Quartil	47534	47872	47538	48255
Standardabw.	2384,21	1987,04	1862,24	642

Abbildung 4.5: Ergebnisse der 2ten levels-back-Versuche

und mit 784 Inputs nur eine Chance von 5/789 = 0,6337%, um sich mit einem anderen Knoten zu verbinden. Ein solcher Knoten mit einem levels-back Parameter von 784 hätte bei einer ausreichenden Entfernung zu den Input-Adressen eine Chance von 50%, um einen Input zu wählen. Individuen mit geringen levels-back-Werten können schwieriger mehrere Knoten zur Verarbeitung nutzen. Dies spiegelt sich auch in der Anzahl der benutzten Knoten wider. Bei allen Versuchen, die mit einem levels-back-Parameter arbeiten, waren keine oder nur ein Knoten aktiv. Bei einem Versuch, welcher keinen levels-back-Parameter nutzt, waren teils bis zu zwei Knoten aktiv.

Um das Verhalten, dass sich levels-back-Parameter negativ auf die Fitness ausüben, weiter zu testen, wurde die Anzahl der maximalen Generationen auf 8000 erhöht. Dadurch konnten sich Strukturen über einen längeren Zeitraum entwickeln. Dabei wurde einmal ein Test ohne levels-back-Parameter ausgeführt. Die anderen Versuche wurden mit Werten von 200, 300 und 500 getestet.

Ein Wert von 784 wurde nicht getestet, da der Sinn des levels-back-Parameters darin besteht, die Konnektivität der Knoten zu beeinflussen. Bei einem Wert von 784 ist dies bei einer Knotenanzahl von 1000 kaum möglich, da mit solch einem Wert erst ab dem 784. Knoten Veränderungen bei der Konnektivität auftreten. Des Weiteren wurde auf sehr kleine levels-back-Werte verzichtet, da die Chance einen anderen Knoten als Eingabe zu wählen, zu gering ist. Die resultierenden Ergebnisse sind in 4.5 aufgelistet. Dieser Test wurde für jedes einzelne Ergebnis nur 30 anstatt 60 mal ausgeführt, da sich mit einer erhöhten Generationsanzahl auch die Dauer der Durchläufe erhöhte.

Aus diesen Ergebnissen kann das Resultat geschlossen werden, dass ohne ein Benutzen des levels-back-Parameters die durchschnittlichen Fitness-Werte am geringsten sind. Außerdem ist die Anzahl der aktiven Knoten am höchsten. Ebenfalls hat sich ergeben, dass

Funktions-Set = 2; Node-Anzahl = 1000; Max Generation = 8000;
Mutationsrate = 6%; Strategie = (1+4); OutputAnzahl = 1;

Levels-Back:	-1	200	300	500
Median	5,5	3	3	2
Arithm. Mittel	5,97	3,27	3,4	1,9
Standardfehler	0,56	0,26	0,19	0,07
Kleinster Wert	1	1	2	1
Größter Wert	11	6	7	3
Unteres Quartil	3	2	3	2
Oberes Quartil	8	4	4	2
Standardabw.	3,06	1,44	1,04	0,4

Abbildung 4.6: Anzahl der aktiven Knoten im 2ten levels-back-Versuch

ein levels-back-Parameter von 300 die besten Ergebnisse erzielte im Vergleich zu 200 oder 500. Die besten Ergebnisse wurden wieder ohne levels-back-Parameter erzielt.

Gezeigt hat sich außerdem, dass sich mehr Generationen positiv auf die Fitness auswirken. So wurden bei 2.000 Generationen schlechtere durchschnittliche Werte erzielt im Vergleich zu 8.000 Generationen. Überraschend war, dass sich die kleinsten Werte nicht verringern. So wurde eine Fitness von 41183 bei einem levels-back-Parameter und 2.000 Generationen erreicht. Bei 8.000 Generationen sank der Wert letztlich auf 41178, wobei dort wesentlich mehr Knoten genutzt wurden. Die Anzahl der aktiven Knoten nach 8.000 Generationen können der Abbildung 4.6 entnommen werden. Dabei wurden für jeden Versuch die Anzahl der aktiven Knoten und nicht die Fitness-Werte aufgelistet.

Der Abbildung kann entnommen werden, dass mit keinem levels-back-Parameter mehr Knoten genutzt werden als bei anderen Tests, bei denen levels-back-Parameter genutzt wurden. Demnach bestätigt sich, dass der levels-back-Parameter das Problem vergrößert, dass öfter Inputs als andere Knoten verwendet werden. In den folgenden Tests wurde der levels-back-Parameters nicht mehr genutzt.

In den Versuchen wird eine Schwachstelle von CGP deutlich. So hat CGP bei einer hohen Anzahl von Eingaben Schwierigkeiten, komplexe Berechnungen mit aktiven Knoten durchzuführen. Die große Anzahl von Eingabe-Adressen sorgt dafür, dass sich Knoten eher mit Eingaben als mit anderen Knoten verbinden. Das sorgt dafür, dass erst mit einer großen Knotenanzahl gewährleistet wird, dass das Individuum mehr Knoten zur Verarbeitung nutzt. Diese Tatsache wird durch die am Anfang beschriebenen Schwierigkeiten, mehr Knoten zur Verarbeitung zu nutzen, weiter verstärkt. Daraus kann das Fazit gezogen werden, dass es weiterhin erschwert blieb, bessere Fitness-Werte zu erzielen.

Damit diesem Problem vorgebeugt wird, wurden, wie im nächsten Abschnitt beschrieben, die Eingabebilder vorher geringfügig bearbeitet.

4.3 Testen mit vorheriger Bildbearbeitung

In diesem Abschnitt werden die Auswirkungen der Parameter von CGP weiterhin untersucht. Vorher wurde jedes Bild einer Bearbeitung unterzogen. Dabei sollte geprüft werden, ob eine vorherige Bildbearbeitung die Fitness-Werte von CGP verbessert. Vorgegangen wurde dabei wie folgt: Zu Beginn wurden die Auswirkungen der vorherigen Bildbearbeitung untersucht. Mit den daraus gewonnenen Ergebnissen wurde geprüft, ob der levels-back-Parameter mit einer geringeren Zahl von Inputs doch eine problembezogene Nützlichkeit besitzt. Zum Schluss wurden die vorherigen Ergebnisse genutzt, um möglichst niedrige Fitness-Werte zu erzielen.

Dabei wurden zwei verschiedene Verfahren genutzt, um die Eingabebilder zu vereinfachen. Bei dem ersten Verfahren wird das Bild verkleinert, indem aus vier in einem Quadrat angeordneten, nebeneinanderliegenden Pixeln das arithmetische Mittel aus dem Grauwert berechnet wird. Anschließend werden diese vier Pixel zu einem zusammengefasst, dem dann das zuvor berechnete arithmetische Mittel zugewiesen wird. Wird dieser Prozess für alle Pixel durchgeführt, wird ein kleineres Bild mit 25% der Größe des vorherigen erzeugt. Dieses kleinere Bild soll es ermöglichen, dass sich Knoten innerhalb des Genotypen besser untereinander verbinden und weniger die Inputs des Individuums nutzen. Bei dem zweiten Verfahren wird das Bild in ein Schwarz-Weiß-Bild überführt. Dafür wird, bevor über das komplette Bild gelaufen wird, ein Treshold-Wert festgelegt, welcher als Schwellwert dient. Anschließend wird für jeden Pixel geprüft, ob er einen höheren Wert als den Treshold besitzt. Wenn der Wert größer ist, wird der Wert des Pixels auf Eins gesetzt, wenn nicht auf Null.

4.3.1 Testen der Bildbearbeitung

In diesem Abschnitt wird geprüft, ob und inwiefern sich die vorherige Bildbearbeitung auf die Fitness derr Individuen auswirkt. Zu Beginn werden dafür alle Bilder einmalig, wie zuvor beschrieben, verkleinert, sodass den Individuen weniger Eingaben überreicht werden. Die kleineren Bilder nehmen die ersten 196 Adressen in einem Individuum ein und nicht wie bisher 784 Adressen. So soll das Problem von CGP, mit großen Eingaben umzugehen, teilweise gelöst werden, da die Anzahl der Eingaben verringert wird.

Im ersten Versuch wurden die verkleinerten Bilder mit den vorher genutzten Parametern getestet. Die Mutations-Rate wurde hierbei auf 10% erhöht, um mehr aktive Gene zu mutieren. Die Versuche des ersten Testes wurden in Abbildung 4.7 aufgelistet. Die Tests wurden für eine Generationsanzahl von 2000, 4000 und 8000 ausgeführt, um zu sehen inwiefern sich die durchschnittlichen Fitness-Werte verbessern.

In diesen Tests ist gut zu erkennen, inwiefern eine größere Anzahl an maximalen Generationen die Fitness beeinflusst. Sind die Eingabebilder kleiner, sind die Auswirkungen auf die Fitness größer. Werden die Tests 4.4 und 4.5 mit keinem levels-back-Parameter mit

Bildbearbeitung: 1x kleiner
Funktions-Set = 2; Node-Anzahl = 1000;
Mutationsrate = 10%; Strategie = (1+4); OutputAnzahl = 1;

Max Gen.:	2000	4000	8000
Median	46945,5	45345,5	42507
Arithm. Mittel	46269,9	45130,63	42706,33
Standardfehler	339,4	339,09	461,65
Kleinster Wert	40377	40610	38301
Größter Wert	47896	47550	47464
Unteres Quartil	45633	43698	40382
Oberes Quartil	47550	47082	44320
Standardabw.	1858,96	1857,29	2528,58

Abbildung 4.7: Ergebnisse von einmaliger Bildverkleinerung

denen der kleineren Eingabebildern verglichen, lassen sich Verbesserungen erkennen. Diese Unterschiede sind bei einer Generationzahl von 2.000 kaum zu erkennen. Jedoch sind schon bei einer Generationszahl von 4.000 die Ergebnisse mit kleineren Eingabebildern fast so gut wie die Ergebnisse mit größeren Eingabebildern und einer Generationszahl von 8.000. Demnach haben kleinere Eingabebilder mit 8.000 Generationen die besten Ergebnisse erzielt. Daraus kann der Schluss gezogen werden, dass kleinere Bilder eine bessere Fitness ermöglichen. Ebenfalls ist bei kleineren Eingabe-Bildern die Wahrscheinlichkeit größer, dass ein Knoten einen anderen nutzt. Bei 8.000 Generationen wurden bis zu 32 aktive Knoten verwendet, was sich positiv auf die allgemeine Fitness auswirkt.

Anschließend wurde versucht, die Bilder weiter zu vereinfachen, um möglicherweise bessere Ergebnisse zu erzielen. Dazu wurden wieder zwei Versuche gestartet. Dabei sind beide Versuche mit einer maximalen Generationsanzahl von 4.000 gestartet und nutzen dieselben Parameter wie bei den Versuchen in 4.7. Bei diesen Versuchen wurde das Bildbearbeitungsverfahren geändert. So wurde bei dem ersten Versuch das Bild zweimal verkleinert, was zur Folge hat, dass das Bild aus 49 Pixeln besteht. Bei diesem Versuch soll geprüft werden, ob dieses Bild von CGP in dieser Form noch erkannt wird. Bei dem zweiten Versuch wird das Bild zweimal verkleinert und dann mit einem Treshold von 50 in ein Schwarz-Weis-Bild übergeführt. Die Ergebnisse dieser beiden Versuche sind in Abbildung 4.8 aufgelistet.

Werden die Ergebnisse des Versuchs I mit den Ergebnissen aus 4.7 mit ebenfalls 4.000 Generationen verglichen, wird deutlich, dass sich die Ergebnisse verschlechtert haben. Daraus kann der Schluss gezogen werden, dass trotz der geringeren Anzahl von Eingaben, die Bilder schlechter erkannt werden. Die Ergebnisse aus Versuch II machen deutlich, dass ein Schwarz-Weiß-Bild mit einem Treshold von 50 kaum noch erkennbar ist. Trotz

Funktions-Set = 2; Node-Anzahl = 1000; Max Generation = 4000;
Mutationsrate = 10%; Strategie = (1+4); OutputAnzahl = 1;

Versuch:	I	II
Median	45505	51676,5
Arithm. Mittel	44708,63	51832,37
Standardfehler	423,95	77,25
Kleinster Wert	39736	51281
Größter Wert	47381	52654
Unteres Quartil	43096	51548
Oberes Quartil	46777	52045
Standardabw.	2322,08	423,09

Abbildung 4.8: Ergebnisse von verschiedenen Bildbearbeitungsverfahren

der Unerkennbarkeit der Bilder konnte jedoch bestätigt werden, dass mit einer geringeren Anzahl von Inputs mehr Knoten aktiv sind. In beiden Versuchen waren mindestens drei Knoten aktiv, wobei meistens zwischen 30 und 40 Knoten aktiv waren. Die höchste Anzahl aktiver Knoten wurde in Versuch I mit maximal 73 aktiven Knoten erzielt. In Versuch II waren maximal 57 Knoten aktiv. Demnach kann bestätigt werden, dass bei einer kleineren Eingabe mehr aktive Knoten vorliegen. Dies unterstreicht das Problem in CGP mit großen Eingaben effektiv zu arbeiten. Jedoch ist es nicht effektiv, das Bild mit den beschriebenen Verfahren weiter zu verkleinern, da dadurch das Bild unerkennbar wird, weshalb in den darauffolgenden Tests die Bilder nur einmalig verkleinert wurden.

Aufbauend auf den vorherigen Ergebnissen wurde getestet, ob ein einmalig verkleinertes Bild, welches in ein Schwarz-Weiß-Bild umgewandelt wurde, erkennbar bleibt. Da das Bild in binärer Darstellung vorliegt, wurden zwei neue Funktions-Sets eingeführt, welche besser mit dieser Darstellung umgehen können. Das Funktions-Set 5 nutzt die binäre Darstellung des Bildes aus und nutzt dabei einfache logische Operationen. Es werden Verundung, Veroderung und Negation genutzt, um zu einem Bild die Klassifikation zu bestimmen. Funktions-Set 6 ist eine Umwandlung von Funktions-Set 2, um auf einem Bild in Schwarz-Weiß-Darstellung besser zu arbeiten. So wurde die Konstante in allen Funktionen von 255 auf Eins reduziert und die Potenz-Funktion entfernt. Diese beiden Funktions-Sets wurden mit denselben Parametern wie in 4.7 getestet. Es wurde eine maximale Generationsanzahl von 8.000 gewählt. Die Ergebnisse sind in 4.9 aufgelistet.

Das Funktions-Set 5 schnitt in diesen Versuchen schlecht ab. So zeigte sich, dass simple logische Operatoren so nicht für die Bild-Klassifikation nutzbar sind. Hingegen zeigte der Versuch mit Funktions-Set 6 bessere Ergebnisse. Der Test mit diesem Funktions-Set übertraf alle zuvor durchgeführten Tests. Daraus konnte für den weiteren Verlauf geschlussfolgert werden, dass es vorteilhaft ist, nach der Verkleinerung der Eingabebilder, diese

Bildbearbeitung: 1x kleiner & in Schwarz-Weiß
Node-Anzahl = 1000; Max Generation = 8000; Mutationsrate = 10%;
Levels-Back = -1; Strategie = (1+4); OutputAnzahl = 1;

Funktions-Set:	5	6
Median	47398	41534,5
Arithm. Mittel	47430,83	41673,97
Standardfehler	11,31	568,94
Kleinster Wert	47379	36379
Größter Wert	47539	47539
Unteres Quartil	47389	39795
Oberes Quartil	47427	43508
Standardabw.	61,94	3116,2

Abbildung 4.9: Ergebnisse von Funktions-Sets bei kleineren Bildern in Schwarz-Weiß

in Schwarz-Weiß-Bilder zu überführen. Im Folgenden wurde darauf aufbauend versucht, bessere Fitness-Werte zu erreichen.

4.3.2 Testen des levels-back-Parameters bei vorheriger Bildbearbeitung

In diesem Abschnitt wenden wir uns erneut dem levels-back-Parameter zu. Es sollte geprüft werden, ob er bei der kleineren Eingabemenge eine positive Wirkung besitzt und nicht wie zuvor die Fitness-Werte verschlechtert.

Genutzt wurden dieselben Parameter wie zuvor, nur wurde die maximale Generationsanzahl auf 10.000 erhöht. Es wurden verschiedene levels-back-Parameter auf ihre Wirkung auf die durchschnittliche Fitness hin betrachtet. Die Ergebnisse der Tests sind in 4.10 augelistet.

Dabei konnte erneut gezeigt werden, dass die Anwendung von levels-back-Parameter sich negativ auf die Fitness-Entwicklung auswirkt. Damit wird erneut die Annahme bestätigt, dass durch die Anwendung von levels-back-Parametern die Berechnung von großen Eingaben erschwert wird. Dies lässt sich dadurch erklären, dass bei der Nutzung von levels-back-Parametern die Chance höher ist, dass sich ein Knoten mit einem Input statt mit einem anderen Knoten verbindet. Weshalb die Schwäche von CGP mit großen Eingaben umzugehen, durch dieses Parameter verstärkt wird.

4.3.3 Abschließende Tests

In diesem Abschnitt wird auf die Erkenntnisse aus den vorherigen Abschnitten eingegangen, um möglichst optimale Läufe in PCGP zu starten. Dabei wurden zu Beginn weitere verschiedene Parameter getestet, woraufhin Tests mit möglichst vielen Generationen gestartet wurden, um eine möglichst optimale Lösung zu erreichen.

Bildbearbeitung: 1x kleiner & in Schwarz-Weiß
Funktions-Set = 6; Node-Anzahl = 1000; Max Generation = 10000;
Mutationsrate = 10%; Strategie = (1+4); OutputAnzahl = 1;

Levels-Back:	-1	150	300	500
Median	40414,5	42235	43718,5	44712
Arithm. Mittel	41017,53	43032,4	44099,3	44630,57
Standardfehler	537,28	561,13	373,91	257,57
Kleinster Wert	36118	38149	39775	40653
Größter Wert	47539	47539	47539	47425
Unteres Quartil	38896	41036	42633	44712
Oberes Quartil	42179	46158	45360	45198
Standardabw.	2942,83	3073,41	2048	1410,8

Abbildung 4.10: Ergebnisse von levels-back-Parametern bei vorheriger Bildbearbeitung

Bildbearbeitung: 1x kleiner & in Schwarz-Weiß
Funktions-Set = 6; Node-Anzahl = 1000; Max Generation = 10000;
Levels-Back = -1; Strategie = (1+4); OutputAnzahl = 1;

Mutationsrate:	10%	6%	3%	5%	7%
Median	40414,5	39570,5	43271,5	40819	41242,5
Arithm. Mittel	41017,53	40546	42840,6	41242	41087,4
Standardfehler	537,28	640,82	566,12	637,5	534,41
Kleinster Wert	36118	32774	35261	35062	34458
Größter Wert	47539	47539	47890	47539	47539
Unteres Quartil	38896	37952	41154	39089	39427
Oberes Quartil	42179	43064	44861	43219	42633
Standardabw.	2942,83	3509,89	3100,74	3491,73	2927,07

Abbildung 4.11: Ergebnisse Mutationsraten bei vorheriger Bildbearbeitung

Ursprünglich wurde die Mutations-Rate mit 2.000 Generation ohne vorherige Bildbearbeitung getestet. Hinzu kommt, dass die im ersten Versuch (s. Abbildung 4.2) gemachten Ergebnisse nah beieinander lagen, weshalb die Mutations-Raten erneut getestet wurden. Dabei wurden Mutations-Raten von 6% und 10% getestet. Anschließend wurden, aufgrund dieser Ergebnisse Mutationsraten von 3%, 5% und 7% getestet. Die Ergebnisse dieser Versuche sind in 4.11 aufgelistet.

In diesen Versuchen ist aufgefallen, dass die Ergebnisse der Mutationsraten diesmal wesentlich unterschiedlicher ausgefallen sind. So war eine Mutationsrate von 6% wesentlich besser als eine Mutationsrate von 10%. Wegen dieser Ergebnisse wurden weitere Mutationsraten getestet, welche im Bereich um sechs liegen, um sicherzustellen, dass eine

Bildbearbeitung: 1x kleiner & in Schwarz-Weiß
Funktions-Set = 6; Node-Anzahl = 1000; Max Generation = 10000;
Mutationsrate = 6%; Levels-Back = -1; OutputAnzahl = 1;

Strategie:	(1+4)	(2+8)
Median	39570,5	39623,5
Arithm. Mittel	40546	39640,73
Standardfehler	640,82	401,46
Kleinster Wert	32774	34458
Größter Wert	47539	47539
Unteres Quartil	37952	37257
Oberes Quartil	43064	411660
Standardabw.	3509,89	3109,66

Abbildung 4.12: Ergebnisse für verschiedene Strategien

Mutationsrate von 6% die besten Ergebnisse für die dann folgenden Versuche war. Dabei konnte festgestellt werden, dass eine 6%ige Mutationsrate die besten Ergebnisse erzielte. Deshalb wurde anschließend nur noch mit einer Mutationsrate von 6% gearbeitet.

Neben einer (1+4)-Strategie sind noch andere Strategien möglich. Im Folgenden soll beispielhaft eine (2+8)-Strategie getestet werden. Bei einer (2+8)-Strategie werden anstatt einem Elternteil, zwei Elternteile in die nächste Generation überführt. Des Weiteren beträgt bei dieser Strategie die Größe einer Generation zehn anstatt fünf Individuen, wie es bei einer (1+4)-Strategie üblich ist. Die Ergebnisse sind in 4.12 ersichtlich.

Es zeigte sich, dass beide Strategien ähnliche Ergebnisse erzielten. Eine (2+8)-Strategie ermöglichte mehr Varietät in der Generation, da die zwei besten Elternteile und nicht nur eins in die nächste Generation übernommen wurde. Ebenfalls zeigt das Ergebnis, dass die zusätzliche Varietät keine besseren Fitness-Werte erzielt. Die Laufzeit einer (2+8)-Strategie ist ungefähr doppelt so lang wie die Laufzeit einer (1+4)-Strategie, da doppelt so viele Individuen ausgewertet werden müssen, weshalb in den darauffolgenden Versuchen eine (1+4)-Strategie angewandt wurde.

Da eine höhere maximale Generationszahl zu besserern Ergebnissen führt, wurde in den darauffolgenden Tests die maximale Generationsanzahl erhöht. Weiterhin wurden die besten Parameter aus den vorherigen Versuchen genutzt, um möglichst gute Ergebnisse zu erhalten. Die durchgeführten Versuche sind in Abbildung 4.13 einzusehen.

Bildbearbeitung: 1x kleiner & in Schwarz-Weiß
Funktions-Set = 6; Node-Anzahl = 1000; Mutationsrate = 6%;
Strategie = (1+4); Levels-Back = -1; OutputAnzahl = 1;

Max Generation:	20.000	30.000	100.000
Median	38417,5	37560	35582,5
Arithm. Mittel	38573,73	38038,6	35892
Standardfehler	456,98	455,46	473,15
Kleinster Wert	34441	33368	29965
Größter Wert	44861	43484	41657
Unteres Quartil	36911	36312	33992
Oberes Quartil	39952	40206	38029
Standardabw.	2508,98	2494,66	2591,54

Abbildung 4.13: Ergebnisse bei mehr Generationen

4.4 PCGP - Fazit und Schlussfolgerungen

Zusammenfassend zeigt sich, dass PCGP zu diesem Zeitpunkt nicht gut genutzt werden konnte, um die Klassifikation des MNIST-Datensatzes erfolgreich zu bewältigen. Die Gründe dafür wurden aus mehreren Versuchen ersichtlich.

CGP ermöglicht es, ein Funktions-Set zu wählen, dass auf das Problem zugeschnitten ist, wodurch etwaiges Vorwissen zu einer besseren Lösung beitragen kann. Die Wahl eines auf das Problem zugeschnittenen Funktions-Sets könnte die erreichten Fitness-Werte stark verbessern, jedoch können auch mit der Wahl eines vorgegeben Funktions-Sets Rückschlüsse auf die Anwendbarkeit von CGP getroffen werden.

So zeigte sich, dass CGP bei einer großen Anzahl von Eingaben Schwierigkeiten hat, mehrere aktive Knoten zu besitzen. Bei großen Eingaben werden mit höherer Wahrscheinlichkeit Input-Werte als Eingaben der Knoten und Outputs genutzt. Dies wirkt sich negativ auf die Fitness-Entwicklung aus, da komplexe Berechnungen mit mehreren Knoten seltener oder gar nicht auftreten, weshalb es zu weniger oder keinen aktiven Knoten kommt. Begünstigt wird dieses Problem von der Tatsache, dass es bei der Klassifikation der Daten schwierig ist, mehrere Knoten zu nutzen, da bereits mit wenig aktiven Knoten eine Klassifikationsrate von 20% erreicht werden kann.

Die Wirksamkeit des levels-back-Parameters hinsichtlich dieses Problems wurde deutlich. Bei der Nutzung eines levels-back-Wertes wird die Wahrscheinlichkeit geringer, dass sich komplexe Strukturen bilden können, da mit der Nutzung für einen Knoten die Wahrscheinlichkeit steigt, einen Input statt einen anderen Knoten zu referenzieren. Werden große Eingaben verwendet, wirkt sich das negativ aus, da fast nie andere Knoten genutzt werden und im Genotypen meist nur wenige Knoten aktiv sind.

Positiv konnte aufgezeigt werden, dass durch eine Bildbearbeitung, die die Eingabe-Bilder vereinfacht, bessere Fitness-Werte erreicht wurden. So wurden mithilfe von vorheriger Bildbearbeitung die durschnittlich erreichten Fitness-Werte verringert. Ebenfalls konnte eine optimale Mutationsrate von 6% ermittelt werden.

Insgesamt konnte keine Lösung für das Klassifikationsproblem gefunden werden. Jedoch wurde die Effektivität einzelner Parameter aufgezeigt und bearbeitet. Die besten Fitness-Werte konnten von 43356 in Versuch 4.1 zu 29965 4.13 gesenkt werden, dies entspricht einer Klassifikationsrate von mehr als 50%. Mit weiteren Verbesserungen an Parametern, Bildbearbeitung und Funktions-Sets könnte CGP effektiver werden, um dieses Problem zu lösen. So könnte durch die Entwicklung eines problem-spezifischen Funktions-Sets die durschnittliche Fitness stark verringert werden. Des Weiteren wäre es möglich durch aufwendigere Bildbearbeitungsverfahren spezifische Bildmerkmale hervorzuheben, die es erleichtern, diese zu klassifizieren. Mit diesen Verfahren könnten die restlichen Parameter genau festgelegt werden, sodass diese sich besser für die vorherigen Verfahren eignen. Außerdem könnte dazu die maximale Generationsanzahl erhöht werden, um bessere Ergebnisse zu erzielen.

Neben diesen Verbesserungsmöglichkeiten könnten Erweiterungen von CGP genutzt werden. So bietet das Embedded Cartesian Genetic Programming (ECGP) eine Erweiterung von CGP an, mit derer sich die Ergebnisse eventuell verbessern ließen. Da es im Rahmen der Arbeit nicht möglich ist, all diese Varianten zu testen, wurde sich dafür entschieden, ECGP zunutzen, da ECGP qualitativ bessere Ergebnisse trotz längerer Laufzeit erzielen soll. [18]

Kapitel 5

Embedded Cartesian Genetic Programming

Embedded Cartesian Genetic Programming (ECGP) ist eine Erweiterung von CGP, welche im Folgenden erklärt werden soll. [18] CGP wird dabei um Module erweitert, die im Genotypen auftreten können. Ein Modul ist ein kleines CGP-Programm, welches als Funktion in ECGP genutzt werden kann. Module enstehen in ECGP während der Evolution aus dem Genotypen. Module besitzen eigene Mutationen, die sich an den ursprünglichen Mutationen von CGP anlehnen. So enstehen in ECGP eigens erstellte Funktionen. [11] ECGP nutzt, wie CGP einen $(\mu + \lambda)$- Evolutions-Algorithmus. [18]

Zu Beginn wird erläutert wie die grundlegende Struktur des Genotypen verändert werden muss, um die Anwendung von Modulen zu ermöglichen. Daraufhin werden Module vorgestellt und ihre Aufbauweise sowie ihre Verbindung mit dem Genotypen anhand von Beispielen erläutert. Danach wird die Mutation in ECGP behandelt. Außerdem wird besprochen, wie Module erstellt und wieder gelöscht werden. Am Ende wird auf die Modul-Mutation eingegangen, welche dafür sorgt, dass Module mutiert werden können.

Auf die genaue Berechnung von Outputs wird nicht gesondert eingegangen, da diese sich nicht groß von CGP unterscheidet. Bei beiden werden erst alle aktiven Knoten markiert und dann für diese ein Output berechnet.

5.1 Genotyp-Änderungen

Im Folgenden wird auf die Änderungen eingegangen, die an dem CGP-Genotypen vorgenommen werden müssen, damit der ECGP-Genoytp entstehen kann.

Jedes Individuum in ECGP besitzt eine Modulliste, in der alle Module enthalten sind, die das Individuum besitzt. Um Module und somit selbst erstellte Funktionen möglich zu machen, muss der CGP-Genotyp leicht verändert werden. So werden Gene im Genotypen nicht mehr durch einen einzelnen Integer dargestellt, weshalb jedes Gen nun aus einem

Paar von zwei Integern besteht und demnach mehr Information enthält. Die Anzahl der Gene pro Knoten wird nicht verändert. Die Codierung eines Gens mit zwei Integern ist nötig, da Knoten mehrere Outputs und einen verschiedenen Typ besitzen können. [18]

5.1.1 Beispiel. Um dieses Prinzip anschaulich zu erklären, wird diese Veränderung an einem Beipiel erläutert. Dieses Beispiel orientiert sich an [18].

Dabei sehen wir uns folgenden Knoten in CGP an:

3 7 2

Dieser Knoten nutzt die Funktion unter der Adresse 3 und als Eingaben die Werte, welche unter den Adressen 7 und 2 vorliegen. In ECGP sieht dieser Knoten nun wie folgt aus:

3:0 7:0 2:0

Dabei stehen die Adressen für genau dieselben Eingaben und Funktionen wie bisher. Welche Bedeutung der zweite Integer in einem Gen besitzt, wird im Folgenden erklärt.

Ein Input-Gen besteht aus zwei Integern, wobei der erste wie bereits in CGP die Adresse angibt, von welchem die Eingabe des Knoten genommen wird. Der zweite Integer gibt die Nummer des Outputs an der von dem adressierten Knoten benutzt werden soll. Dieser zweite Integer ist nötig, da Knoten mehrere Outputs besitzen können. Bei dem Funktions-Gen nimmt der erste Integer dieselbe Funktion ein wie in CGP. Der zweite Integer gibt den Typ des momentanen Knoten an.

Knotentypen ermöglichen es, Knoten in drei verschiedene Typen aufzuteilen: [18]

- Knoten-Typ 0 : Der Knoten nutzt eine primitive Funktion. Diese sind in der Funktions-Tabelle enthalten.
- Knoten-Typ 1 : Der Knoten nutzt ein Modul, welches eine originale Sektion aus dem Genotypen nutzt.
- Knoten-Typ 2 : Der Knoten nutzt ein Modul, welches eine kopierte Sektion aus dem Genotypen nutzt (ein erneut genutzes Modul).

Die Typen dieser Knoten müssen angegeben sein, da viele Operatoren abhängig von ihrem Knoten-Typen auf die Knoten wirken. Knoten des Types 0 verhalten sich wie die Knoten aus CGP. Sie besitzen Eingabe-Adressen, eine Adresse zu einer Funktion in der Funktions-Tabelle und eine Ausgabe. Knoten des Types 1 und 2 sind hingegen verschieden. Sie nutzen beide Module als Funktionen. Der zweite Integer gibt die Nummer des Moduls an, welches genutzt werden soll. Dies ist ähnlich zu einer Funktions-Adresse, nur wird die Modul-Nummer nicht in der Funktions-Liste aufgelistet, sondern in der Modulliste. [18] Die Unterschiede der Knotentypen 1 und 2 werden in den folgenden Abschnitten genauer erläutert.

Durch die Nutzung von Modulen besitzt ein Genotyp in ECGP nicht immer dieselbe Länge. In ECGP ist die Genotyp-Länge variabel. Sie reduziert sich, wenn aus einem Teil des Genotypen ein Modul erzeugt wird. Die Genotypen-Länge erhöht sich, wenn ein Modul wieder in den Genotypen eingefügt wird. Weiterhin kann sich die Genotyp-Länge verändern, wenn ein Knoten ein Modul referenziert oder sich ein Modul mutiert. Auf die genauen Auswirkungen wird in den folgenden Abschnitten eingegangen.

5.2 Module

Ein Modul ist ein Abschnitt aus dem Genotypen des ursprünglichen ECGP-Genotypen. [18] Das Modul besitzt dieselben Eigenschaften wie der ECGP-Genotyp. In einem Modul können keine weiteren Module enthalten sein. Die Anzahl der Knoten eines Moduls ist von Anfang an festgelegt und kann nicht mehr verändert werden. Ein Modul besitzt Inputs und Outputs, deren Anzahl sich durch Modul-Mutation verändern kann.

Ein Modul-Genotyp ist eine Liste von Integern, welche in drei Abschnitte eingeteilt ist. Der erste Abschnitt befindet sich am Beginn des Genotypen und wird Modul-Header genannt und enthält vier Integer. Die Integer codieren in folgender Reihenfolge: Modulnummer, Anzahl an Inputs, Anzahl an Knoten innerhalb des Moduls sowie Anzahl an Outputs. Die Modul-Nummer gibt an, unter welcher Adresse das Modul referenziert werden kann. [18]

Der zweite Abschnitt codiert die Knoten, die im Modul enthalten sind. Sie werden auf dieselbe Art und Weise codiert, wie die Knoten im ECGP-Genotyp. Der dritte Abschnitt am Ende des Genotypen codiert die Output-Adressen. Diese geben wie in CGP an, von welchen Knoten der Output als Ausgabe genutzt wird. [18]

ECGP erlaubt es nicht, Module innerhalb von anderen Modulen zu verwenden. Sie nutzen nur primitive Funktionen aus der Funktionstabelle. Ein Modul wird erzeugt, indem der Compress-Operator angewandt wird. [18] Dieser fasst eine zufällige Anzahl von Knoten zu einem Modul zusammen. Die genaue Funktionsweise des Compress-Operators wird in Abschnitt 5.4.2 erläutert. Um ein Modul wieder in den Genotypen einzusetzen, ist der Expand-Operator nötig. [18] Dieser arbeitet, indem er die Operation des Compress-Operators rückgängig macht. Die genaue Funktionsweise des Expand-Operators wird in Abschnitt 5.4.3 erläutert. Ein Modul kann sich im Genotypen replizieren, wenn ein anderer Knoten sein Funktions-Gen so mutiert, dass er ein Modul referenziert. [18] Dieser Knoten erhält dann den Knoten-Typ 2, da er ein Modul referenziert, was nicht durch ihn erstellt wurde. Die bisher nicht festgelegten Inputs des Knotens werden zufällig gewählt.

- Phase 1: Punkt-Mutation (ähnlich zur Mutation in CGP)
- Phase 2: Compress (zur Modulerstellung)
- Phase 3: Expand (zur Modulexpansion)
- Phase 4: Modul-Mutation (Mutation auf einzelnen Modulen)
- Phase 5: Löschung unbenutzter Module

Abbildung 5.1: Mutationsphasen in ECGP

5.3 Knoten-Typen

Wie in Abschnitt 5.1 dargestellt, kann ein Knoten drei verschiedene Typen besitzen. Die Unterscheidung von Knoten des Types 1 und des Types 2 ist notwendig, damit die Größe des Genotypes sich nicht stark verändert. [18]

Knoten des Types 2 können nicht vom Exand-Operator beeinflusst werden. Dies verhindert, dass die Größe des Genotypes immens ansteigt, wenn Knoten ein Modul referenzieren, welches durch den Expand-Operator immer wieder in den Genotyp eingefügt werden könnte. Mithilfe dieser Restriktion ist es nur Knoten mit Knoten-Typ 1 möglich, vom Expand-Operator beeinflusst zu werden und Knoten in den Genotyp einzufügen. [18]

Des Weiteren können die Funktionen von Knoten des Types 1 nicht geändert werden. Dies verhindert, dass Knoten, die vorher eine originale Sektion des Genotypes referenzierten, nicht eine andere Funktion oder ein anderes Modul referenzieren und den Knoten-Typ wechseln. Da Knoten den Typ 1 nur durch den Compress-Operator erhalten, wäre das Modul, welches vorher referenziert wurde, nicht wieder in den Genotyp einfügbar. So könnte es passieren, dass die Größe des Genotypes immer kleiner wird. Damit dies nicht geschieht, sind Knoten des Types 1 von der Mutation des Funktions-Genes ausgenommen. [18]

5.4 Mutation

ECGP nutzt wie CGP einen $(\mu + \lambda)$-Evolutions-Algorithmus. [18] Die Besonderheit bei diesem Algorithmus in ECGP ist die Mutation. Sie gestaltet sich aufwendiger und komplizierter als in CGP und wird deshalb in diesem Kapital ausführlich behandelt.

Die Mutation in ECGP besteht aus fünf verschiedenen Phasen, welche in Abbildung 5.1 ersichtlich sind.

Jede Phase besitzt eine gewisse Wahrscheinlichkeit ausgeführt zu werden. Diese wird ECGP zu Beginn überreicht. Die für die Ausführung von ECGP benötigten Parameter sind in 5.2 aufgelistet.

- Funktions-Set: das zu nutzende Funktions-Set
- Mutations-Rate: die Rate der Punkt-Mutation
- Knoten-Anzahl: die ursprüngliche Anzahl der Knoten eines Individuums
- max. Generationsanzahl: die maximal zu erreichende Generation bevor der Durchlauf abgebrochen wird
- levels-back-Parameter: für die Verbindungseigenschaften der Knoten
- Output-Anzahl: die Anzahl der Outputs eines Individuums
- max. Modul-Größe: die maximale Modulgröße
- $(\mu + \lambda)$: die zu verwendende Strategie für den evolutionären Algorithmus
- Compress-Wahrscheinlichkeit: die Wahrscheinlichkeit des Compress-Operators & Expand-Operators
- Modul-Punkt-Mutations-Wahrscheinlichkeit: die Wahrscheinlichkeit der Modul-Punkt-Mutation
- Add-Input-Wahrscheinlichkeit: die Wahrscheinlichkeit, einen Input in einem Modul hinzuzufügen oder zu entfernen
- Add-Output-Wahrscheinlichkeit: die Wahrscheinlichkeit, einen Output in einem Modul hinzuzufügen oder zu entfernen

Abbildung 5.2: Parameter in ECGP

5.4.1 Punkt-Mutation

Die Punkt-Mutation in ECGP besitzt dieselbe Vorgehensweise wie in CGP. Anhand der Mutationsrate wird bestimmt, wie viele Gene mutiert werden sollen. Danach werden so oft wie vorher festgelegt, Gene ausgewählt und mutiert. Bei ECGP ergeben sich einige Besonderheiten, auf welche geachtet werden muss.

Es gibt zwei unterschiedliche Fälle der Mutation abhängig davon, was für ein Gen für die Mutation ausgewählt wurde. So ist die Vorgehensweise davon abhängig, ob ein Funktions-Gen oder ein Verbindungs-Gen gewählt wurde. [18]

Funktions-Gen-Mutation

Ein Funktions-Gen beschreibt, welche Operation ein Knoten auf seinen Eingaben ausführt. Da in ECGP auch Module von dem Funktions-Gen referenziert werden können, ergeben sich Besonderheiten, die auftreten, wenn ein solches Gen zur Mutation ausgewählt wird.

Wenn ein Knoten des Types 1 zur Mutation ausgewählt wird, erfolgt keine Operation, da Knoten des Types 1 immun gegenüber der Funktions-Gen-Mutation sind. Wurde kein Knoten des Types 1 gewählt, muss eine zufällige Funktion ausgewählt werden. Mögliche Werte dieser zufälligen Wahl sind alle Adressen des vorgegebenen Funktions-Sets und alle Modulnummern aus der Modulliste des Individuums.

Wird eine primitive Funktion als neue Funktion festgelegt, wird die Funktions-Adresse entsprechend gesetzt und der Knoten-Typ auf 0 gesetzt. Besaß der Knoten vorher mehr Inputs als die primitive Funktion braucht, werden diese verworfen.

Wird als neue Funktion ein Modul gewählt, wird die Funktions-Adresse auf die Modulnummer und der Knoten-Typ auf 2 gesetzt. Es werden alle vorher existierenden Inputs des Knoten übernommen. Besaß der Knoten zuvor mehr Inputs als nötig, werden diese verworfen. Besitzt der Knoten weniger Inputs als zuvor, werden ihm die restlichen Inputs zufällig zugewiesen.

Verbindungs-Gen-Mutation

Ein Verbindungs-Gen beschreibt, woher ein Knoten seine Eingaben oder ein Individuum seine Ausgaben bezieht. Da in ECGP ein Knoten mehrere Outputs besitzen kann, beeinflusst dies die zufällige Wahl einer Adresse.

Wird ein Verbindungs-Gen zur Mutation ausgewählt, muss zunächst ein Input oder ein Knoten gewählt werden, mit dem sich das Individuum verbinden kann. Wurde ein Input als Verbindungsadresse gewählt, wird seine Adresse angegeben. Außerdem wird der Output 0 gewählt, da Inputs nur einen möglichen Wert besitzen.

Wird hingegen ein anderer Knoten als Verbindung ausgewählt, muss zuerst auf den Knoten-Typen geachtet werden. Ist der Knoten von Typ 0, wird wie bei der Verbindung zu einem Input die Adresse und der erste (und einzige) Output angegeben. Besitzt der

Knoten Typ 1 oder 2, wird zwar die Adresse angegeben, aber die Output-Nummer wird zufällig aus allen möglichen Outputs bestimmt.

5.4.2 Compress

Nachdem die Punkt-Mutation durchgeführt wurde, werden nun durch den Compress-Operator die Module erstellt. [18] Der Compress-Operator wird nur zu der übergebenen Wahrscheinlichkeit ausgeführt.

Zu Beginn des Compress-Operators werden zunächst zwei zufällige Positionen im Genotypen gewählt. Die Knoten, die sich zwischen diesen beiden Positionen befinden, sollen zu einem Modul zusammengefasst werden. Die Punkte müssen verschieden sein, damit die Mindestgröße von zwei Knoten für Module erfüllt ist. Außerdem dürfen sie maximal so weit voneinander entfernt sein, wie durch den max.-Modulgröße-Parameter vorgegeben. Besitzt ein Knoten innerhalb dieses gewählten Bereiches den Knoten-Typ 1 oder 2 bricht der Operator ab und es wird keine Operation ausgeführt. So wird verhindert, dass innerhalb von Modulen weitere Module referenziert werden können.

Sind nur Knoten des Types 0 zwischen den beiden Punkten vorhanden, wird aus diesen ein Modul kreiert. Dafür werden die gewählten Knoten aus dem Genotyp entfernt und ein neuer Knoten eingefügt. Dieser neue Knoten referenziert das neu kreierte Modul. Die Modulnummer wird mithilfe der Modulliste bestimmt, da sie einzigartig unter allen Modulen des Individuum sein muss, damit eindeutig bestimmt werden kann, welches Modul ein Knoten referenziert.

Die Anzahl der Inputs des Moduls bestimmt sich aus allen Verbindungen von Knoten im Modul zu Adressen vor dem Modul. Für jede dieser Verbindungen wird ein Modul-Input erstellt. Die ursprünglichen Inputs werden dem Knoten, der im Genotyp das Modul referenziert, als Verbindungs-Gen eingetragen. Die Knoten im Modul, welche eine Referenz in den Genotyp besaßen, werden so verändert, dass sie den Modul-Input referenzieren.

Die Anzahl der Outputs bestimmt sich aus der Anzahl der Verbindungen von Knoten des Genotypen zu Knoten, die im Modul enthalten sind. Für jede dieser Verbindungen wird in dem Modul ein Output hinzugefügt, dieser referenziert den jeweiligen referenzierten Knoten. Für den Knoten im Genotypen, der einen Knoten in dem Modul referenziert, wird die Verbindungs-Adresse geändert, sodass er den jeweiligen Output des Knoten adressiert, der das Modul im Genotyp repräsentiert.

Wurde das Modul, wie beschrieben, erstellt, müssen alle Verbindungsadressen des Individuums geändert werden, da Knoten aus dem Genotypen entfernt wurden, weshalb sich die Verbindungsadressen verschoben haben.

5.4.1 Beispiel. Im Folgenden wollen wir die Erstellung eines Moduls an einem vorgegebenen Genotypen beispielhaft durchführen. Dafür wird zuerst ein Genotyp vorgestellt, aus

dem anschließend ein Modul erzeugt werden soll. Nach Erstellung des Moduls wird der veränderte Genotyp vorgestellt, um Unterschiede hervorzuheben.

Der Genotyp, welcher verändert werden soll, besitzt drei Inputs, acht Knoten und einen Output. Die Input-Adressen sind demnach im Bereich [0, 2] und die Adressen der Knoten liegen zu Beginn im Bereich [3, 10]. Wir nutzen ein Funktions-Set mit drei Funktionen. Die Funktionen und Funktions-Adressen sind wie folgt verteilt: 0: +; 1: -; 2: *.

Um einzelne Knoten besser voneinander zu unterscheiden, wird hier jeder Knoten von den anderen abgegrenzt und seine Adresse unter ihm gelistet. Unter O ist dabei der Output gelistet. Der Genotyp besitzt die Form:

$|_3$ 0:0 0:0 2:0 $|_4$ 2:0 3:0 0:0 $|_5$ 0:0 1:0 2:0 $|_6$ 1:0 5:0 3:0 $|_7$ 0:0 5:0 6:0

$|_8$ 2:0 0:0 0:0 $|_9$ 2:0 6:0 7:0 $|_{10}$ 0:0 7:0 3:0 $|_O$ 9:0

Seien als Eingaben x_0, x_1 und x_2 gegeben. Mit der Outputadresse 9:0 wird von dem Genotypen die Funktion $((x_1+x_2)-(x_0+x_2))*((x_1+x_2)+((x_1+x_2)-(x_0+x_2))$ codiert. Gekürzt ergibt sich daraus die Funktion $(x_1 - x_0) * (x_2 + 2x_1 - x_0)$.

Aus diesem Genotypen wollen wir nun ein Modul erzeugen. Dem Modul wird eine Modul-Nummer von 0 zugewiesen, da noch kein anderes Modul existiert. Es soll die Knoten 5 bis 8 umfassen, woraus sich eine Knoten-Anzahl von vier Knoten innerhalb des Modules ergibt.

Die Anzahl der Inputs des Moduls ergibt sich aus der Anzahl von Verbindungen der innerhalb des Moduls liegenden Knoten, welche sich mit den außerhalb liegenden Knoten verbunden haben. Die Knoten 5 und 8 referenzieren jeweils zwei Inputs. Der Knoten 6 adressiert den Knoten 3, welcher nicht innerhalb des zu generierenden Moduls liegt. Daraus ergibt sich eine Anzahl von fünf Inputs. Die ersten beiden Inputs des Moduls sind die Adressen, die vom Knoten 5 referenziert werden. Der dritte Input des Moduls wird von Knoten 6 vorgegeben und die letzen beiden von Knoten 8.

Die Anzahl der Outputs ergibt sich aus der Anzahl der Knoten, die Knoten innerhalb des Modules referenzieren. So referenzieren die Knoten 9 und 10 andere Knoten innerhalb des Modules. Da 9 zwei und 10 einen Knoten referenziert, ergibt sich eine Anzahl von drei Outputs.

Mit diesen Überlegungen können wir den Modul-Header beschreiben. Der Modul Header sieht wie folgt aus: 0 5 4 3.

Um das Modul vollständig zu beschreiben, fehlen die Knoten. Diese werden aus dem Genotypen übernommen. Da das Modul fünf Inputs besitzt, fangen die Adressen der Knoten bei 5 an. Referenziert ein Knoten jedoch einen Input oder einen anderen Knoten außerhalb des Moduls, wird nun der Modul-Input referenziert. Die Outputs ergeben sich dabei aus den vorher von außerhalb referenzierten Knoten. Daraus ergibt sich die folgende Darstellung des Moduls: Der Header wird dabei durch H hervorgehoben und der Modul-Output durch o.

$|_H$ 0 5 4 3 $|_5$ 0:0 0:0 1:0 $|_6$ 1:0 5:0 2:0 $|_7$ 0:0 5:0 6:0 $|_8$ 2:0 3:0 4:0 $|_o$ 6:0 7:0 7:0

Aus dem Genotyp wurden die Knoten entfernt und unter der Adresse 5 wird ein Knoten eingefügt, welcher das Modul referenziert. Als Funktions-Gen besitzt er nun die Adresse 0:1. Dabei ist die 0 die Modul-Nummer und die 1 der Knoten-Typ, welcher angibt, dass ein Modul und nicht eine Funktion genutzt werden soll. Dieser Knoten besitzt fünf Inputs, wobei jeder Input für einen Input des Moduls steht. Mit dem Einsetzen dieses Knotens würde das Individuum folgendermaßen aussehen:

$|_3$ 0:0 0:0 2:0 $|_4$ 2:0 3:0 0:0 $|_5$ 0:1 1:0 2:0 3:0 0:0 0:0 $|_6$ 2:0 5:0 5:1 $|_7$ 0:0 5:2 3:0 $|_O$ 9:0

Dabei wurden die Knoten des Moduls entfernt, ein neuer Knoten, der das Modul referenziert eingefügt und die restlichen Knoten des Genotypen aufgeschoben, damit keine Zwischenräume in den Adressen entstehen.

5.4.3 Expand

In diesem Abschnitt wird der Expand-Operator behandelt. Dieser wird nach dem Compress-Operator durchgeführt und dient dazu, Module wieder in den Genotypen einzugliedern. [18]

Die Wahrscheinlichkeit, dass der Express-Operator durchgeführt werden soll, ist die doppelte Wahrscheinlichkeit der compress-Wahrscheinlichkeit. Dies wird benötigt, damit nur Module vorhanden bleiben, welche nützlich für die Fitness sind. [18]

Eine Änderung an der Modulliste wird vom Expand-Operator nicht ausgeführt. Er gliedert ein Modul in den Genotypen ein, jedoch bleibt das Modul noch in der Modulliste vorhanden. Dies geschieht, da andere Knoten des Typs 2 das Modul noch referenzieren könnten. Ein Löschen des Moduls wäre in diesem Falle also schädlich.

Zu Beginn der Expand-Operation wird ein zufälliger Knoten mit Knoten-Typ 1 aus dem Genotyp ausgewählt. Das Modul, welches dieser Knoten referenziert, soll im Folgenden in den Genotypen eingefügt werden. Dazu werden die Knoten, die im Modul enthalten sind hinter dem referenzierenden Knoten in den Genotyp eingefügt. Danach wird der Knoten, der zuvor das Modul referenzierte, gelöscht. Die Eingabe-Adressen, die dieser Knoten besaß, müssen zwischengespeichert werden, da diese im weiteren Verlauf benötigt werden. Bei den eingefügten Knoten, die zuvor im Modul enthalten waren, müssen die Eingabe-Adressen geändert werden. Haben sie einen anderen Knoten im Modul adressiert, muss die neue Adresse an die entsprechende Stelle im Genotypen angepasst werden. Wurde hingegen ein Input im Modul adressiert, ist die neue Eingabe-Adresse des Knotens die entsprechende Adresse des entfernten Knotens. So wird sichergestellt, dass alle Knoten richtig in den Genotyp eingefügt werden.

Danach müssen alle Knoten, die hinter dem ursprünglich entfernten Knoten standen, geprüft werden. Wenn diese den gelöschten Knoten referenziert haben, muss die neue Adresse angepasst werden. Die neue Adresse ergibt sich aus der Position des referenzierten Knotens und des jeweiligen Outputs im Modul. Haben diese Knoten einen Knoten

referenziert, der hinter dem eingefügten Modul liegt, müssen ihre Eingabe-Adressen an die neue Länge des Genotypens angepasst werden, damit die Funktionalität des Genotypens bestehen bleibt.

Für ein Beispiel dieses Anwendungsfalls kann das Beispiel 5.4.1 rückwärts betrachtet werden, da der Expand-Operator die Effekte eines Compress-Operators rückgängig macht.

Durch die Anwendung der Expand-Funktion kann die Einschränkung des levels-back-Parameter-Wertes umgangen werden. Sei ein Individuum mit einem levels-back-Wert von 1 gegeben. Wir betrachten einen Knoten in diesem Individuum, welcher einen Knoten vor sich referenziert, der ein Modul als Operator nutzt. Wird dieses entsprechende Modul expandiert, kann es auftreten, dass der zuerst genannte Knoten einen Knoten referenziert, der maximal so weit entfernt ist, wie die maximale Modul-Größe es zulässt. So kann der levels-back-Wert überschritten werden.

5.4.4 Modul-Mutation

Die vierte ECGP-Mutationsphase ist die Modulmutation. Sie mutiert zu einer gewissen Wahrscheinlichkeit jedes Modul und ermöglicht es so, Varietät bei Modulen zu schaffen. Sie wird nacheinander für jedes Modul ausgeführt. [18] Diese Phase beginnt, nachdem der Expand-Operator seine Arbeit vollendet hat. Sie besteht aus 3 verschiedenen Teilen.

Im Folgenden wird auf die verschiedenen Modul-Mutationen eingegangen und ihre Vorgehensweise erläutert.

Punkt-Mutation

Die Punkt-Mutation ist die erste der Mutationen innerhalb eines Moduls. Ihr wird keine Mutationsrate übergeben, sondern nur eine Modul-Punkt-Mutations-Wahrscheinlichkeit. Wird diese erfüllt, wird innerhalb des Moduls nur ein einziges Gen verändert. Dieses zu mutierende Gen wird zufällig ausgewählt. [18]

Wurde ein Funktions-Gen ausgewählt, kann die Funktion zu einer beliebigen anderen primitiven Funktion geändert werden. Ein Modul kann nicht als Funktion gewählt werden, da geschachtelte Module nicht erlaubt sind. Wird ein Verbindungs-Gen eines Knoten des Moduls gewählt, kann sich der Knoten zu Inputs des Moduls oder zu anderen Knoten, die im Genotyp vor ihm liegen, verbinden. Ist das zu mutierende Gen ein Output-Gen des Moduls, kann als neue Output-Adresse innerhalb des Moduls nur ein Knoten gewählt werden. Würde sich der Modul-Output zu einem Modul-Input verbinden, wäre das Modul zwecklos und dies ist nicht erlaubt.

Input-Mutation

Die Input-Mutation eines Moduls erfolgt nach der Modul-Punkt-Mutation. Sie besitzt die Wahrscheinlichkeit, einem Modul einen Input hinzuzufügen oder einen zu entfernen. Die

Wahrscheinlichkeit einen Input hinzuzufügen, wird vom Nutzer festgelegt. Einen Input zu entfernen, ist immer doppelt so wahrscheinlich wie einen Input hinzuzufügen. So wird gewährleistet, dass Module nicht zu großem Wachstum unterliegen.

Soll ein Input hinzugefügt werden, wird die Anzahl der Inputs im Modul erhöht. [18] Diese kann nicht größer sein als die doppelte Anzahl der Knoten innerhalb des Moduls. Hat ein Modul die maximale Anzahl an Inputs erreicht, kann kein weiterer hinzugefügt werden. Wird jedoch ein Input hinzugefügt, wird er an der letzten Stelle der Input-Adressen eingefügt. Die Anzahl an Inputs innerhalb des Module-Headers wird um eins inkrementiert. Dabei müssen alle Adressen von Knoten innerhalb des Moduls um eins erhöht werden, wenn sie einen anderen Knoten referenzieren. Dies ist der Fall, da die Knoten-Adressen sich um einen erhöhen, weil eine neue Input-Adresse eingefügt wurde. Die Adressen von allen Modul-Output-Genen müssen erhöht werden, da ein Modul-Output immer nur einen Knoten im Modul referenziert. Für alle Knoten, die dieses mutierte Modul referenzieren, wird außerdem ein neuer zufälliger Input zugewiesen.

Nachdem dies geschehen ist, kann zu der doppelten Wahrscheinlichkeit ein Input entfernt werden. Wenn geplant ist, dies zu tun, sollte zuerst geprüft werden, ob die Anzahl der Inputs größer als zwei ist, da ein Modul welches zwei Inputs besitzt die Anzahl an Inputs nicht weiter verringern kann, weshalb in diesem Fall, keine Operation ausgeführt werden kann. [18] Trifft dies nicht zu, wird zu Beginn die Anzahl an Inputs innerhalb des Module-Headers um eins dekrementiert. Anschließend wird die letzte Input-Adresse innerhalb des Moduls gelöscht. Die betroffenen Adressen der Knoten und die Adressen der Outputs werden um eins dekrementiert, damit sie weiterhin valide Adressen enthalten. Daraufhin wird für jeden Knoten im Genotyp des Individuums der letzte Input gelöscht, falls dieser das betroffene Modul referenziert.

Output-Mutation

Die Output-Mutation erfolgt direkt nach der Input-Mutation und ist ihr sehr ähnlich. Der Nutzer legt zu Beginn von ECGP eine Wahrscheinlichkeit fest, mit der ein Output hinzugefügt werden soll. Die Wahrscheinlichkeit einen Output zu entfernen, ist doppelt so hoch wie die einen hinzuzufügen.

Es wird kein neuer Output hinzugefügt, wenn das Individuum bereits so viele Outputs besitzt, wie es Knoten enthält. [18] Wenn ein Output hinzugefügt werden soll, wird zunächst im Modul-Header die Anzahl der Outputs erhöht. Danach wird eine zufällige Adresse eines Knoten innerhalb des Moduls gewählt und als letzer Output des Moduls ans Ende des Modul-Genotypes hinzugefügt.

Danach kann ein Output abhängig von der Wahrscheinlichkeit gelöscht werden. Das Modul kann dabei niemals weniger als einen Output besitzen. [18] Zunächst wird dafür ein zufälliger Output des Moduls gewählt, welcher daraufhin gelöscht wird. Danach muss

der Genotyp des Individuums durchlaufen und geprüft werden, ob dieser gelöschte Output referenziert wird. Hat ein Knoten einen gelöschten Output referenziert, wird seine Referenz geändert, sodass ein Output des Moduls adressiert wird, welcher existiert.

Nach dieser Output-Mutation ist die Modul-Mutation abgeschlossen.

5.4.5 Modul-Löschung

Nach der Modul-Mutation erfolgt die letzte Phase (Löschung unbenutzter Module), um die Mutation abzuschließen. Hierbei werden alle Module, die von keinem Knoten adressiert werden, aus der Modulliste gelöscht. [18] Dafür wird der Genotyp des Individuums durchlaufen und geprüft, welche Module referenziert werden. Es werden dabei auch Module als benutzt markiert, wenn diese durch einen nicht-aktiven Knoten referenziert werden. Da nicht-aktive Knoten in einem darauffolgenden Schritt immer noch aktiv werden könnten.

Nachdem alle unbenutzten Module aus der Modulliste gelöscht wurden, ist die Mutation eines Individuums in ECGP abgeschlossen.

Kapitel 6

PECGP

Piepenbrink Embedded Cartesian Genetic Programming (PECGP) ist eine komplett eigenständig, vom Autor entworfene Implementierung von ECGP. In diesem Kapitel wird das entworfene Programm vorgestellt und seine Ausführung beschrieben.

Dafür werden zu Beginn grundlegende Eigenschaften zu PECGP vorgestellt. Anschließend wird auf die genutzten Hilfsklassen eingegangen. Daraufhin wird das ECGP-Netzwerk, welches in PECGP codiert wird, beschrieben. Danach werden die Klasse und deren Funktionsweise beschrieben, welche für die Mutation von Individuen und für deren Auswertung zuständig sind. Zum Schluss werden einige Rahmenklassen beschrieben, mit denen es erleichtert wird, ein ECGP-Programm zu starten und zu nutzen.

6.1 Grundlegendes

PECGP soll das Klassifikationsproblem mithilfe des ECGP-Ansatzes bearbeiten. Es wurde für den MNIST-Datensatz entworfen, kann jedoch auch für andere Klassifikationsprobleme genutzt werden. Es wurde entwickelt, da zuvor keine Implementierung von ECGP in Java existierte. Dadurch wurde eine Möglichkeit geschaffen, ECGP auf den MNIST-Datensatz zu testen und mit CGP zu vergleichen.

PECGP nutzt das Java-Develompment-Kit und wurde aufbauend auf PCGP programmiert. Es besitzt mehr Komplexität und überarbeitet viele der zuvor definierten Klassen. Es soll als ein Framework für weitere Versuche in ECGP dienen und den Einstieg sowie die Anwendbarkeit erleichtern. PECGP arbeitet mit einem eindimensionalen ECGP-Netzwerk und einer $(\mu + \lambda)$-Strategie, um den evolutionären Algorithmus auszuführen. In PECGP besitzen primitive Funktionen eine Anzahl von zwei Inputs und die Anzahl der falsch klassifizierten Inputs wird als Fitness beschrieben.

PECGP soll alle Funktionen bieten die CGP und ECGP beschreiben. Es soll leicht verständlich und möglichst einfach bedienbar sein. Des Weiteren soll in PECGP durch Parallelität die Laufzeit möglichst gering gehalten werden. Das Framework soll eine Ein-

sicht in ECGP erlauben, die es leicht macht, das zu Grunde liegende Konzept zu verstehen und weiterzuverwenden. Eine graphische Oberfläche ist nicht angeboten, da dies das Thema dieser Arbeit verfehlt.

PECGP lässt sich mit der ECGP_NetHandler-Klasse verwenden. Sie enthält alle Parameter, die für die Ausführung notwendig sind. Die verwendeten Parameter sind in Abbildung 6.1 aufgelistet.

Dabei sind die Inputs, die ein PECGP-Netzwerk erhält, eine Liste von Input-Klassen, welche genauer in 6.2 beschrieben werden. Die Input-Klassen sollen von einem PECGP-Netwerk möglichst gut klassifiziert werden. Dafür wird die Evolutions-Klasse benutzt, welche den $(\mu + \lambda)$-Algorithmus beschreibt und ausführt. Die verwendeten Klassen und deren Aufbau und Funktion werden im Rest dieses Kapitels erläutert.

Viele Klassen wurden bereits in der Beschreibung von PCGP erläutert. Da einige Klassen verändert wurden, werden diese Klassen hier kurz beschrieben.

6.2 Hilfsklassen

In diesem Abschnitt werden verschiedene Klassen beschrieben, die wichtig sind, um PECGP auszuführen. Zuerst werden dafür die Klassen beschrieben, die als Inputs für die einzelnen Individuen benötigt werden. Danach wird die Functions-Klasse beschrieben, welche die Funktionen für ein Netzwerk bereitstellt.

Die Klassen Input und InputWithClassification sind beides abstrakte Klassen. Diese dienen dazu, neue Klassen zu erstellen, mit denen PECGP arbeitet. So stellt die Klasse Input eine Methode bereit, die es ermöglicht, den Input eines Objektes als Integer-Array auszugeben. So kann auch zu unbekannten Objekten, welche die Klasse erweitern, ein Output berechnet werden.

Die InputWithClassification-Klasse ist eine abstrakte Klasse, die die Klasse Input um eine Methode erweitert, die die Klassifikation des Objektes als Integer ausgibt. Dies ermöglicht es, dem PECGP-Netzwerk zu unbekannten Listen von Objekten Fitness-Werte zu berechnen, da so die Klassifikation mit dem Output eines Individuums verglichen werden kann.

Die Digit-Klasse erweitert die InputWihtClassification-Klasse, um die benötigten Angaben für das MNIST-Klassifikationsproblem bereitzustellen. Sie enthält ein Bild, das als zweidimensionales Array von Integern gespeichert ist und einen Integer, der beschreibt, welche Zahl auf diesem Bild zu sehen ist. Sie ermöglicht es, das Bild als ein eindimensionales-Array auszugeben, das von PECGP als Eingabe verwendet werden kann. Des Weiteren besitzt die Klasse zwei Methoden, die es ermöglichen, das dargestellte Bild zu vereinfachen. Diese beiden Methoden wurden in 3.2.1 genauer beschrieben.

Die Functions-Klasse beschreibt die Funktions-Sets, die PECGP vorliegen. Die Funktions-Sets, die in ECGP genutzt werden, sind die gleichen Funktions-Sets, die auch in PCGP

- path: Der Dateipfad, unter dem die besten Individuen der Generation gespeichert werden.
- print: Gibt an, ob während der Berechnung Ausgaben in die Konsole erlaubt sind.
- maxAllowedSecondsForMutation: Gibt eine Obergrenze für die Dauer der Mutation an.
- diffCalc: Gibt an, ob die Fitness von Individuen durch Vergleiche vorberechnet werden soll.
- inputs: Die Inputs, die das PECGP-Netzwerk verarbeiten soll.
- nodeAmount: Start-Anzahl der Knoten eines Individuums in PECGP.
- outputAmount: Anzahl der Outputs eines Individuums.
- functionSet: Das von Individuen zu benutzende Funktions-Set.
- maxModuleSize: Anzahl der Knoten, die ein Modul maximal enthalten darf.
- maxAllowedModules: Anzahl der maximal erlaubten gleichzeitigen Module
- generationLimit: Maximale Anzahl an Generationen bevor die Berechnung abgebrochen wird
- mutation_rate: Mutations-Rate für die Punkt-Mutation von Individuen
- levelsback: Der zu beachtende levels-back-Parameter (-1 wenn levels-back nicht verwendet werden soll)
- mu: Um die $(\mu + \lambda)$-Strategie zu bestimmen.
- lambda: Um die $(\mu + \lambda)$-Strategie zu bestimmen.
- compress_probability: Wahrscheinlichkeit für Ausführung des Compress-Operators
- modulePointMutation_probability: Wahrscheinlichkeit für Ausführung der Modul-Punkt-Mutation
- addInput_probability: Wahrscheinlichkeit, einen Input zu einem Modul hinzuzufügen
- addOutput_probability: Wahrscheinlichkeit, einen Output zu einem Modul hinzuzufügen.

Abbildung 6.1: Parameter in PECGP

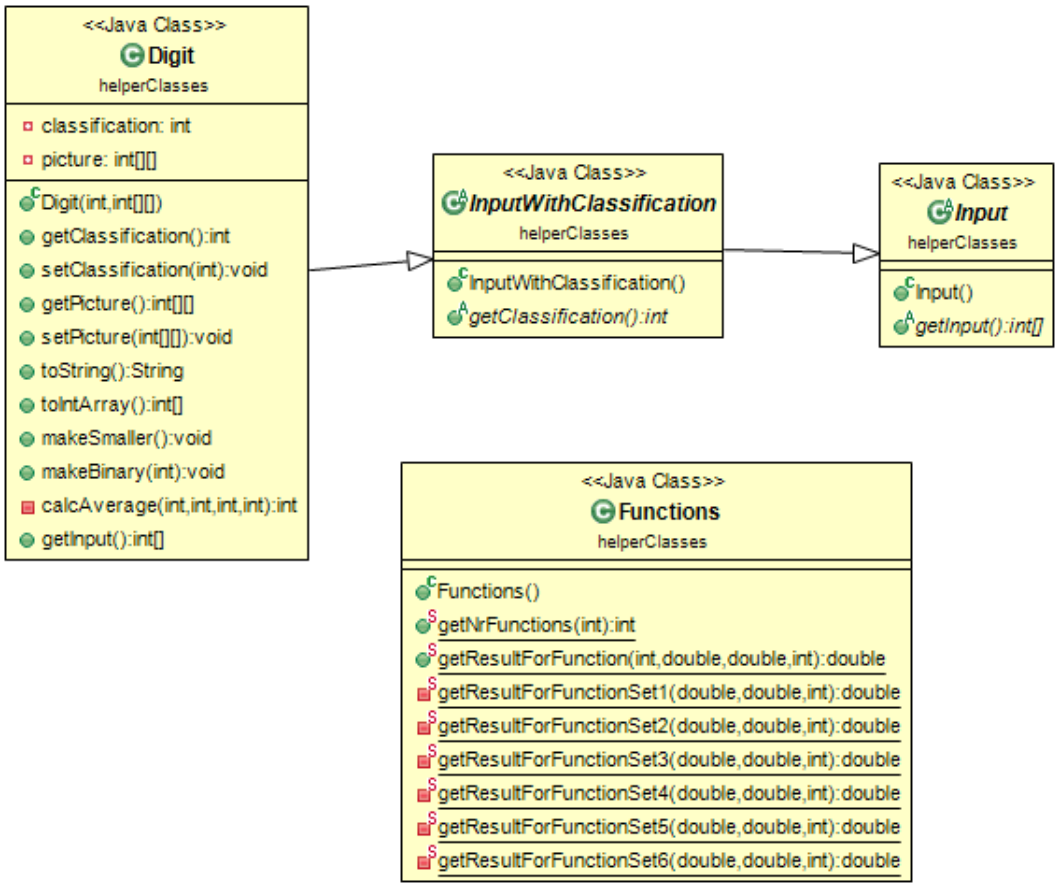


Abbildung 6.2: Klassendiagramm Digit und Functions

genutzt werden. (s. 3.2.2) Die Klasse wird in der Outputberechnung eines Individuums genutzt, welche im Abschnitt 6.5 genauer beschrieben wird.

6.3 Das Netzwerk

Die Klassen, die in diesem Abschnitt beschrieben werden, stellen die Grundstruktur von PECGP dar. Sie beschreiben ein Individuum und seine Bestandteile. Die Evolution-Klasse vollführt den eigentlichen evolutionären Algorithmus und gibt die fittesten Individuen aus.

6.3.1 NodeECGP

Die NodeECGP-Klasse stellt einen ECGP-Knoten dar. Sie enthält Attribute, die Funktions-Gen und Verbindungs-Gene beschreiben.

Das Funktions-Gen ist als Integer-Array gespeichert. An erster Stelle steht die Funktionsadresse bzw. die Modulnummer und an zweiter Stelle wird der Knoten-Typ gelistet. Die Verbindungs-Gene sind als Liste von Integer-Arrays gespeichert. Dabei besteht jedes Integer-Array aus zwei Positionen. Auf der ersten Position ist die Knoten-Adresse gespei-

chert, die von diesem Knoten referenziert wird. An zweiter Stelle ist der Knoten-Output gespeichert, der von dem referenzierten Knoten als Input an die Funktion/das Modul weitergereicht wird.

Des Weiteren enthält diese Klasse eine copy-Methode, die eine Kopie des Knoten erstellt. Diese wird später von anderen Funktionen gebraucht, wenn sie einen Teil eines Genotyps kopieren wollen.

6.3.2 Module

Die Klasse Module repräsentiert ein ECGP-Modul. Es enthält eine Liste von NodeECGP-Modulen, die die Knoten des Moduls darstellen und eine Liste von Integer-Array für die Output-Adressen. Eine Integer-Array-Output-Adresse besteht dabei aus zwei Stellen. An erster Stelle steht die Adresse und an zweiter der jeweilige Output von dem referenzierten Knoten.

Der Modul-Header enthält in PECGP zwei Werte. Es werden die Modulnummer (Module-Identifier) an erster Stelle und die Anzahl von Modul-Inputs an zweiter Stelle gespeichert. Die Anzahl der Knoten und der Outputs werden nicht separat gespeichert, da diese über die Längen der jeweiligen Listen zurückgegeben werden können.

Des Weiteren enthält die Module-Klasse eine Möglichkeit, die aktiven Knoten des Moduls zu speichern. Dies dient dazu, den Output eines Moduls nach einmaliger Berechnung der aktiven Knoten schneller zu berechnen. Ebenfalls wird gespeichert, ob die aktiven Knoten im Modul bereits berechnet wurden. Auch enthält diese Klasse ein copy-Methode, um Module kopieren zu können, ohne Änderungen am ursprünglichen Objekt vornehmen zu müssen.

6.3.3 ModuleList

Die Klasse ModuleList implementiert eine Modul-Liste. Sie ist dafür zuständig, die Liste und die Modulnummern (Module-Identifier) zu verwalten.

Sie enthält eine Methode, die einen von der Modulliste unbenutzten Module-Identifier zurückgibt. So kann sichergestellt werden, dass zwei Module niemals die gleiche Modulnummer besitzen. Ebenfalls ermöglicht sie es, mit einem gegebenen Modul-Identifier das entsprechende Modul zu erhalten oder es aus der Modulliste zu löschen.

Die ModuleList-Klasse enthält eine copy-Methode, mit der die Modulliste kopiert wird. Dies dient dazu, um Kindern eines Elternteiles dieselbe Modulliste zuzuweisen, ohne dass diese ungewollt von den Kindern modifiziert wird.

6.3.4 Individual

Die Individual-Klasse stellt ein ECGP-Individuum dar. Sie enthält alle Werte, die für ein Individuum bedeutend sind.

Eine Liste von NodeECGP-Objekten, die im Individuum gespeichert werden, repräsentieren den Genotypen des Individuums. Des Weiteren werden die Anzahl der In- und Outputs in dieser Klasse gespeichert. Die Output-Adressen werden als zwei-dimensionales Array dargestellt. So wird dadurch eine Liste von zwei-elementigen Arrays repräsentiert, die die Adressen der einzelnen Outputs codieren. An erster Stelle eines Outputs ist hierbei die jeweilige Adresse und an zweiter Stelle der jeweilige Output gespeichert.

Um die Mutationsrate eines Individuums richtig zu bestimmen, muss bekannt sein, wie groß der Genotyp ist. Da sich in ECGP die Genotyp-Größe ständig ändert, wird sie für jedes Individual-Objekt festgelegt. Die benutzte Modulliste wird als Module-List und die maximale Größe der Liste wird als Integer gepeichert. Auch enthält ein Individual-Objekt Attribute für das zu verwendende Funktions-Set und den levels-back-Parameter. Die Fitness und die aktiven Knoten eines Individuums werden als Attribute aufgeführt. Aus diese beiden Werte kann zugegriffen werden, wenn sie berechnet und dem Individual-Objekt zugewiesen wurden. Die Adressen in einem Individuum sind wie in PCGP verteilt.

Die Individual-Klasse enthält zwei Konstruktoren. Der eine Konstruktor erstellt ein Individual-Objekt mit den übergebenen Parametern. Diese umfassen eine Liste von NodeECGP-Objekten, die In- und Output-Anzahl, die Output-Adressen, das zu verwendende Funktions-Set sowie die ModuleList, die Größe des Genotypes, die maximale Modul-Größe und den levels-back-Parameter. Dieser erste Konstruktor wird bei der Mutation von Individual-Objekten genutzt, da die zu übergebenden Werte bereits bekannt sind.

Der zweite Konstruktor ermöglicht es, ein zufälliges Individuum zu erstellen. Er wird am Anfang des Algorithmus benutzt, um eine Anzahl von Individuen zu erstellen, die die erste Generation bildet. Damit dies möglich ist, müssen ihm die ursprüngliche Knoten-Anzahl, mit denen das Objekt initialisiert werden soll, die In- und Output-Anzahlen, das zu nutzende Funktions-Set sowie die maximale Modul-Größe und der levels-back-Parameter übergeben werden. Damit einem Individuum eine bereits zu Beginn vorgefertigte Modul-Liste übergeben werden kann, wird diese hier als Parameter übergeben. Zu Beginn des Konstruktors werden die übergebenen Werte als Attribute festgelegt und es können die zufälligen Knoten kreiert werden. Die Funktions-Adressen eines Knoten werden bei dieser zufälligen Erstellung aus der Liste der primitiven Funktionen gewählt, sodass jeder Knoten einen Knoten-Typen von 0 besitzt. Die restlichen Verbindungs-Adressen werden wie in Abschnitt 3.3.2 zufällig erstellt.

Neben den üblichen Getter- und Setter-Methoden besitzt die Individual-Klasse eine Methode, um die Größe ihres eigenen Genotypen zu errechnen. Diese wird genutzt, um die Mutations-Anzahl genau berechnen zu können. Sie läuft durch den gesamten Genotypen und addiert die Anzahl aller Gene aufeinander. Des Weiteren besitzt sie eine copy-Methode, um ein identisches Individuum zu erschaffen.

Um ein Individuum abspeichern zu können, implementieren die Klasse Individual sowie alle von ihm benutzten Klassen (NodeECGP, Module, ModuleList) das Interface Serializable.

6.3.5 Evolution

Die Evolutions-Klasse ist für den $(\mu + \lambda)$-Algorithmus zuständig. Dafür werden ihr alle nötigen Parameter übergeben. Eine Liste dieser Parameter kann in 6.1 gefunden werden.

Die getXRandomIndiv-Methode erstellt so oft wie nötig zufällige Individuen. Um aus einer Anzahl von Individuen die Individuen mit der geringsten Fitness zu erhalten, wird innerhalb der Evolution-Klasse die getXFittestIndivs-Methode aufgerufen. Diese nutzt die Evaluation-Klasse (s. 6.5.1), um ein Ergebnis zu erhalten.

Um ein Individual-Objekt zu mutieren, wird die mutate-Methode genutzt. Diese erstellt aus Individuen mutierte Kinder und wird in 6.4 kurz erläutert. Da Mutation in ECGP viel aufwendiger ist als in CGP, besteht in PECGP die Möglichkeit die Mutation in Threads zu berechnen. Dies dient dazu, mithilfe von Parallelität die Laufzeit bei großen Populationen zu verringern. Diese Option ist standardmäßig in der Evolution-Klasse deaktiviert, da bei einer Populationsgröße von fünf die Laufzeit durch Mutations-Threading erhöht wird. Sie kann jedoch leicht verändert werden, falls eine Mutation in Threads vorteilhafter ist.

Mithilfe der Methode EvolutionAlgorithm-Methode kann der $(\mu+\lambda)$-Algorithmus ausgeführt werden.

6.4 Mutation

In diesem Abschnitt wird erläutert wie genau PECGP die Mutation von Individual-Objekten handhabt und wie genau diese arbeiten und funktionieren. Aufgerufen wird die Mutation über die CallableMutator-Klasse. Dieser Schritt ist nötig, da es dadurch möglich wird die Mutation per Threading zu verarbeiten.

Der CallableMutator-Klasse werden alle für die Mutation benötigten Parameter im Konstruktor übergeben. Diese umfassen das Individual-Objekt, das mutiert werden soll, die Anzahl an Punkt-Mutationen, die Wahrscheinlichkeit für den Compress-Operator, sowie die Wahrscheinlichkeiten für den Modul-Punkt-Mutations-, Input-Hinzufügen-, und Output-Hinzufügen- -Operator. Desweiteren wird ihr die Anzahl der maximal erlaubten Module überreicht. Wurden all diese Parameter übergeben, wird mithilfe der call-Methode das Individuum mutiert. Diese reicht alle übergebenen Parameter an die Mutator-Klasse weiter, welche für die Mutation zuständig ist. Die call-Methode gibt immer eine eins zurück, außer es ist ein Fehler aufgetreten. Das übergebene Individual-Objekt ist nach Aufruf der Methode mutiert.

In dem restlichen Abschnitt werden die Klassen, die die Mutation ausführen, weiter beschrieben.

6.4.1 Mutator

Die Mutator-Klasse ist für die Punkt-Mutation des übergebenen Individual-Objektes zuständig. Sie ruft außerdem die Compressor, Expander und ModuleMutator-Klassen auf, um die anderen Mutationen eines Individuums auszuführen.

Mithilfe der Methode mutate, der alle für die Mutation wichtigen Parameter übergeben werden, wird das Individuum mutiert. Zu Beginn wird dabei mithilfe der mutateOffspring die Punkt-Mutation ausgeführt. Danach wird mit der Bestimmung von Zufallszahlen bestimmt, ob der Compress- oder der Expand-Operator ausgeführt werden soll. Nachdem diese ihre Ausführung beendet haben, wird die Module-Mutation mithilfe der ModuleMutator-Klasse ausgeführt. Nach Abschluss der Mutation werden mithilfe der deleteUnusedModules-Methode alle unbenutzten Module gelöscht und aus der Modulliste entfernt. Anschließend wird für das Individuum die Größe des Genotyps berechnet, da sich während einer Mutation die Genotyp-Größe verändern kann.

Die Methode mutateOffspring ist dabei für die Punkt-Mutation zuständig. Zu Beginn wird eine zufällige Zahl aus dem Bereich [0, Genotyp-Größe) gewählt. Zuerst wird geprüft, ob ein Output-Gen mutiert werden soll. Ist dies der Fall, wird dies mit der changeOutputGeneOfNode-Methode durchgeführt. Wird hingegen kein Output-Gen ausgewählt, muss der Genotyp durchlaufen werden, bis das zu mutierende Gen gefunden ist. Dies geschieht, indem für jeden Knoten, der durchlaufen wird, die Anzahl an Genen des Knotens von dem zu Anfang zufällig gewählten Wert abgezogen wird. Ergibt sich dabei ein negativer Wert, ist der aktuelle Knoten der, der mutiert werden soll. Um das genaue Gen rauszufinden, welches mutiert werden soll, wird dabei die vorher abgezogene Summe wieder addiert und geprüft, welchen Wert sie annimmt. Ist der neue Wert eine null, soll das Funktions-Gen mutiert werden, welches mithilfe der changeFunctionsGeneOfNode-Methode geschieht. Andernfalls wird über den Wert bestimmt, welcher von den Input-Genen des Knoten mutiert werden soll, wofür die changeInputGeneOfNode-Methode zuständig ist.

6.4.1 Beispiel. Das vorher beschriebene Verfahren, um das zu mutierende Gen zu finden, wird im Folgenden an einem Beispiel weiter erläutert, um es einfacher verständlich zu machen.

Gegeben sei folgender Genotyp:

$|_3$ 0:0 0:0 2:0 $|_4$ 2:0 3:0 0:0 $|_5$ 0:1 1:0 2:0 3:0 0:0 0:0 $|_6$ 2:0 5:0 5:1 $|_7$ 0:0 5:2 3:0 $|_O$ 9:0

Dieser Genotyp besitzt eine Genotyp-Größe von 19. Ein zu mutierendes Gen g wird also aus dem Bereich [0,19) bzw. [0, 18] gewählt. Nehmen wir an, dass ein Wert von g=12 gewählt wurde. Wir wollen nun das richtige Gen, welches mutiert werden soll, bestimmen.

Nun wird der Genotyp durchlaufen und wir wollen die Größe jedes Knotens von g abziehen. Beginnen wir mit dem ersten Knoten, welcher 3 Gene besitzt und wir erhalten g = 12-3 = 9. Für den zweiten Knoten ergibt sich dann g = 9-3 = 6. Der dritte Knoten

besitzt eine Größe von 6 Knoten und folgt daraus: g = 6-6 = 0. Da noch kein Wert der kleiner als Null ist, erreicht wurde, wird mit dem nächsten Knoten fortgefahren. Wird seine Größe auch noch abgezogen, ergibt sich ein Wert von g = 0-3 = -3. Da nun ein Wert erreicht wurde, der kleiner als 0 ist, wird die Größe des Knotens wieder addiert und es ergibt sich der Wert g=0. Daraus folgt, dass das Funktions-Gen des vierten Knotens mutiert werden soll, welches im Folgenden markiert ist:

$|_3$ 0:0 0:0 2:0 $|_4$ 2:0 3:0 0:0 $|_5$ 0:1 1:0 2:0 3:0 0:0 0:0 $|_6$ **2:0** 5:0 5:1 $|_7$ 0:0 5:2 3:0 $|_O$ 9:0

Würde beispielsweise ein zufälliger Wert von 9 gewählt werden, würde das folgende Gen mutiert werden:

$|_3$ 0:0 0:0 2:0 $|_4$ 2:0 3:0 0:0 $|_5$ 0:1 1:0 2:0 **3:0** 0:0 0:0 $|_6$ 2:0 5:0 5:1 $|_7$ 0:0 5:2 3:0 $|_O$ 9:0

Funktions-Mutation

Die changeFunctionsGeneOfNode-Methode ist dafür zuständig, das Funktions-Gen eines Knotens zu mutieren. Zuerst wird geprüft, ob der Knoten, zu dem das Funktions-Gen gehört, den Knoten-Typ 1 besitzt.

Anschließend wird ein zufälliges Funktions-Gen ausgewählt. Dieses kann eine Funktionsadresse oder eine Modulnummer sein. Wurde eine Funktionsadresse gewählt, werden die ersten beiden Inputs des Knotens übernommen und alle anderen werden verworfen und der Knoten-Typ auf 0 gesetzt. Wurde hingegen eine Modulnummer gewählt, wird zunächst der Knoten-Typ auf 2 gesetzt und die Modulnummer entsprechend in das Funktions-Gen eingetragen. Dann werden möglichst viele der vorherigen Inputs des Knotens übernommen. Besaß er vorher zu viele Input-Adressen werden die restlichen verworfen. Hatte er zu wenige Input-Adressen, werden die restlichen Adressen mit Berücksichtigung auf den levels-back-Parameter zufällig zugewiesen.

Wurde der Knoten erfolgreich mutiert, muss der Rest des Genotyps noch durchlaufen werden. Referenziert eine Adresse einen Output des Knoten, der nicht mehr vorhanden ist, wird ein zulässiger zufälliger Output-Wert gewählt.

Input und Output-Mutation

Die changeInputGeneOfNode-Methode ist dafür zuständig, ein gewähltes Verbindungs-Gen zu mutieren. Zuerst wird unter Beachtung des levels-back-Parameters ein zufälliger Knoten oder Input gewählt, zu dem sich der Knoten verbinden soll. Wurde ein Input gewählt, so verbindet sich der Knoten mit diesem. Wurde hingegen ein anderer Knoten gewählt, muss geprüft werden, ob dieser ein Modul referenziert. Wird ein Modul referenziert, wird ein zufälliger Output dieses Moduls als Input gewählt und in dem Knoten gespeichert.

Die changeOutputGeneOfNode-Methode funktioniert ähnlich wie die zuvor besprochene. Sie wählt für den Output, der mutiert werden soll, einen neuen zufälligen Input aus. Dies geschieht mit dem zuvor beschriebenen Verfahren.

Löschung unbenutzter Module

Die Methode deleteUnusedModules wird am Ende der Mutation aufgerufen und löscht alle Module, die in dem Individuum nicht genutzt werden. Zu Beginn wird dafür eine Liste erstellt, die zu jeder Modulnummer die Anzahl ihrer Vorkommen im Gentoyp enthält. Anschließend wird der Genotyp durchlaufen und für jede angetroffene Modulnummer deren entsprechende Vorkommensanzahl erhöht. Danach werden alle Module, die nicht im Gentoypen auftauchen, aus der Modulliste gelöscht.

6.4.2 Compressor

Die Compressor-Klasse ist für die Erstellung von Modulen mithilfe des Compressor-Operator zuständig.

Die compress-Methode wird von der Mutator-Klasse nach der Punkt-Mutation aufgerufen und soll ein Modul innerhalb des Genotyps erstellen. Dafür wird zu Beginn geprüft, ob die Modulliste bereits ihre maximale Größe erreicht hat. Ist dies nicht der Fall, beginnt die Methode damit, zwei zufällige Punkte in dem Genotyp auszuwählen. Ist zwischen diesen Punkten ein oder mehr Knoten mit den Typen 1 oder 2 vorhanden, vollführt die Methode keine weiteren Operationen und bricht ab. Andernfalls wird mithilfe der createModule-Methode ein Modul erstellt. Dieses Modul umfasst alle Knoten zwischen den beiden ausgewählten Punkten.

So werden zu Beginn alle betroffenen Knoten zwischen den ausgewählten Punkten kopiert und einer Liste von Knoten, die später das Modul bilden, hinzugefügt. Danach werden die Anzahl der Inputs des Moduls berechnet. Die Adressen der Knoten des Moduls werden entsprechend angepasst. Die alten Inputs der Knoten des Moduls, welche ersetzt wurden, werden gespeichert, da diese später benötigt werden.

Danach werden die Output-Adressen des Moduls bestimmt. Außerdem werden die Adressen späterer Knoten geändert, damit diese an die Änderung des Adressraumes angepasst werden.

Anschließend wird das Modul erzeugt und die entsprechenden Knoten aus dem Genotypen entfernt. Der Knoten an der ersten Position wird nicht gelöscht. Es wird so verändert, dass er das Modul referenziert und die vorher gespeicherten Inputs benutzt. Das neu erstellte Modul wird zurückgegeben, wo es innerhalb der compress-Methode zu der Modulliste des Individuums hinzugefügt wird.

6.4.3 Expander

Die Expander-Klasse ist dafür zuständig, ein Modul wieder in den Genotyp einzuführen. Die expand-Methode wird von der Mutator-Klasse aufgerufen.

Zu Beginn der Methode wird der Genotyp durchlaufen und eine Liste aller Knoten des Types 1 erstellt, da nur Knoten mit dem Typ 1 expandiert werden können. Existiert kein solcher Knoten bricht die Methode ab, ansonsten wird ein zufälliger Typ-1-Knoten gewählt. Danach werden die Knoten aus dem Modul kopiert, da keine Änderungen an dem Modul durchgeführt werden sollen.

Die Adressen der kopierten Knoten werden abhängig von den Modul-Inputs verändert. Danach wird der Knoten, der expandiert werden soll, gelöscht und an der entsprechenden Stelle werden alle Knoten, die aus dem Modul kopiert wurden, eingefügt. Zuletzt müssen alle Verbindungs-Adressen von Knoten betrachtet werden, die hinter dem zuletzt eingefügten Knoten liegen und entsprechend verändert werden. Für die Output-Adressen des Individuums muss dies ebenfalls durchgeführt werden, damit die Expansion des Moduls vollendet ist.

6.4.4 ModuleMutator

Die ModuleMutator-Klasse ist für die Modul-Mutation zuständig. Sie kümmert sich um die Modul-Punkt-Mutation und die Input-Hinzufügen-, Input-Entfernen-, Output-Hinzufügen- und Output-Entfernen-Operatoren. Die moduleMutation-Methode wird von der Mutator-Klasse aufgerufen. Diese führt für jedes Modul aus der Modulliste des Individuums die mutate-Methode aus.

Die mutate-Methode betimmt, welche Operatoren auf einem Modul ausgeführt werden. Zu Beginn wird mit der übergebenen Wahrscheinlichkeit berechnet, ob die Modul-Punkt-Mutation ausgeführt werden soll, welche mit der pointMutation-Methode ausgeführt wird. Danach wird berechnet, ob der Input-Hinzufügen-Operator ausgeführt werden soll und ob der Input-Entfernen-Operator ausgeführt werden soll. Die addInput-Methode ist dabei für den ersten Operator zuständig und die removeInput-Methode für den Zweiten. Zuletzt werden die Wahrscheinlichkeiten für die addOutput- und removeOutput-Methoden bestimmt.

Modul-Punkt-Mutation

Die pointMutation-Methode ist für die Modul-Punkt-Mutation zuständig. Dafür wird zu Beginn ein zufälliges Gen aus dem Modul ausgewählt. Da ein Modul nur Knoten des Types 0 enthalten kann und diese drei Gene enthalten, ergibt sich die Modul-Genotyp-Größe, indem die Anzahl der enthaltenen Knoten mit drei multipliziert und die Anzahl an Outputs addiert wird. Ein Output-Gen zu mutieren geschieht, indem ein zufälliger Knoten des Moduls ausgewählt wird. Wurde hingegen ein Funktions-Gen eines Knoten

zur Mutation gewählt, wird dem entsprechenden Knoten eine zufällige primitve Funktion zugewiesen. Wurde ein Verbindungs-Gen eines Knoten gewählt, wird dem Knoten ein zufälliger Input oder anderer Knoten als Inputadresse zugewiesen.

Input hinzufügen / entfernen

Die addInput-Methode fügt einem Modul einen Input hinzu. Besitzt das Modul bereits so viele Inputs wie die doppelte Anzahl an Knoten, führt diese Methode keine Operation aus. Ist dies nicht der Fall, wird dem Modul ein Input hinzugefügt, welcher an die restlichen Inputs angehangen wird. So müssen die Adressen aller Knoten und Outputs innerhalb des Moduls, die einen Knoten referenzieren, inkrementiert werden. Des Weiteren müssen im Modul-Header die Anzahl der Inputs erhöht werden und alle Knoten im Individuum durchlaufen werden. Besitzt ein Knoten innerhalb des Individuums das veränderte Modul als Funktionsadresse, muss ihm ein neuer zufälliger Input zugewiesen werden.

Die removeInput-Methode entfernt einen Input aus dem Modul. Sie entfernt nur einen Input, wenn das Modul mehr als zwei Inputs besitzt. Zuerst verringert sie die Anzahl der Inputs in dem jeweiligen Modul-Header. Danach wird ein zufälliger Input des Moduls ausgewählt, welcher gelöscht werden soll. Alle Adressen innerhalb des Moduls die diesen Input und alle späteren Inputs und Knoten referenzieren, werden daraufhin dekrementiert. Zuletzt muss der gesamte Genotyp des Individuums durchlaufen werden. Für Knoten, die das mutierte Modul referenzieren, wird der entsprechende Input gelöscht.

Output hinzufügen / entfernen

Die addOutput-Methode ist dafür zuständig, einen Output zu den Outputs des Moduls hinzuzufügen. Besitzt das Modul bereits so viele Outputs wie Knoten, wird keine Operation durchgeführt. Da Outputs eines Moduls keine Inputs refenzieren dürfen, wird ein zufälliger Knoten gewählt. Anschließend wird ein neuer Output, der diesen zufälligen Knoten referenziert, erstellt und an die Liste der Outputs des Individuums angehängt.

Die removeOutput-Methode entfernt einen zufälligen Output des Moduls. Zuerst wird dabei geprüft, ob das Modul nur einen Output besitzt. Wenn mehr als ein Output vorhanden ist, wird ein zufälliger von diesen gewählt. Dieser wird daraufhin aus der Liste der Modul-Outputs entfernt. Danach wird der Genotyp des Individuums durchlaufen und jede Adresse, die den gelöschten Output des Moduls referenziert, wird geändert, sodass sie einen anderen zufälligen, existierenden Output des Moduls referenziert.

6.5 Evaluation

In diesem Abschnitt werden alle Klassen vorgestellt, die in PCGP für die Berechung des Outputs und der Fitness eines Individuums zuständig sind.

Um die Fitness-Berechnung und auch die Output-Berechnung zu starten, wird die Evaluation-Klasse aufgerufen.

6.5.1 Evaluation

Die Evaluation-Klasse ist für die Fitness- und Output-Berechnung in PECGP zuständig. Sie wird von der Evolution-Klasse zur Fitness-Berechnung aufgerufen. Als Parameter bekommt sie folgendes übergeben:

- Die aufrufende Evolution-Klasse, um weiterzureichen, falls ein perfektes Individuum gefunden wurde.
- Den jeweiligen μ-Parameter. um zu bestimmen, wie viele fitteste Individuen zurückgegeben werden sollen.
- Die momentane Generation, um daraus die fittesten Individuen zu bestimmen.
- Die Liste der Inputs, mit deren Hilfe die Fitness bestimmt wird.
- Die Elternteile der Generation, damit nach dem Prinzip der neutralen Mutation vorgegangen werden kann.
- Einen Boolean-Wert, der angibt, ob die Fitness mithilfe des Vergleiches von aktiven Knoten bestimmt werden soll.

Mithilfe dieser Parameter werden zuerst die Fitness-Werte aller Individuen bestimmt. Dafür wird innerhalb der Evaluation-Klasse die Methode calculateFitnessThreading aufgerufen. Diese bestimmt für jedes Individuum, falls dieses noch keine Fitness besitzt, mithilfe der CallableCalculator-Methode die Fitness. Diese wird mithilfe von Threads bestimmt, welches im Fall des MNIST-Datensatzes eine große Laufzeitverkürzung zur Folge hat.

Die CallableCalculator-Klasse ruft die FitnessCalculator-Klasse auf, um die Fitness zu bestimmen. Damit sie zur Fitnessberechnung die Knoten vergleicht, müssen ihr die Elternteile der Generation überreicht werden. Falls die Fitness auf herrkömmliche Weise berechnet werden soll, wird anstatt der Elternteile ein null-Parameter übergeben. Wie genau die Fitnessberechnung mithilfe des Vergleiches von aktiven Knoten geschieht, wird in Abschnitt 6.5.2 genauer erläutert.

Nach dem alle Fitness-Werte bestimmt wurden, müssen die μ-vielen besten Individuen ausgewählt werden. Dafür wird die getXBest-Methode verwendet, welche die besten Individuen der Generation auswählt. Besitzen mehrere Individuen die gleiche Fitness, werden immer die Kinder anstatt der Eltern ausgewählt. Besitzen jedoch auch mehrere Kinder die gleiche Fitness, wird von ihnen ein zufälliges ausgewählt.

6.5.2 FitnessCalculator

Die FitnessCaculator-Klasse besitzt zwei verschiedene Varianten die Fitness zu berechnen.

Die erste Variante der Fitness-Berechnung ist dieselbe, wie sie in PCGP ausgeführt wird. Diese wird mithilfe der calculateFitnes-Methode berechnet, die als Parameter nur ein Individuum und die Inputs benötigt, die ein Individuum verarbeiten soll.

Die zweite Variante implementiert eine weitere Möglichkeit, der Fitnessberechnung. [12] Bei dieser werden die Genotypen von Kindern und Eltern verglichen. Besitzt ein Kind dabei genau dieselben aktiven Knoten, wird ihm der Fitness-Wert des Elternteils zugewiesen. So werden weitere Fitnessauswertungen vermieden.

Diese zweite Variante wird mit der calculateFitness-Methode aufgerufen, wobei ihr in diesem Fall neben dem Individuum und den Inputs noch die Elternteile übergeben werden. Zuerst werden die benutzten Knoten berechnet, woraufhin die aktiven Knoten der Individuen verglichen werden. Haben Elternteil und Individuum nicht identische aktive Knoten, wird der Fitness-Wert des Individuums wie in der ersten Methode berechnet.

6.5.3 OutputCalculator und ModuleOutputCalculator

Die OutputCalculator-Klasse berechnet für eine Individuum und einen Input den entsprechenden Output des Individuums. Zuerst werden dafür die aktiven Knoten rekursiv berechnet. Danach wird der übergebene Input vom Individuum verarbeitet. Bei PECGP muss auf Knoten, die ein Modul referenzieren, geachtet werden. So wird bei diesen der Output nicht mithilfe der Functions-Klasse, sondern mithilfe der ModuleOuputCalculator-Klasse bestimmt. Die ModuleOutputCalculator-Klasse berechnet zuerst rekursiv die aktiven Knoten des Moduls, woraufhin der übergebene Input verarbeitet wird. So kann zu einem Individuum und einem Input der entsprechende Output berechnet werden.

6.5.4 ComputeOutput

Die ComputeOutput-Methode ist dafür zuständig die Input-Klassifikation mit dem Output des Individuums zu vergleichen. Sie wird von der FitnessCalculator-Klasse aufgerufen, um den vorher berechneten Output zu verarbeiten. Diese besitzt zwei verschiedene Varianten, die von der Output-Anzahl des Individuums abhängig sind. Die erste Variante rundet den Output des Individuums ab und vergleicht ihn mit der Klassifikation. Die zweite Variante geht davon aus, dass im Individuum nur Binär-Werte vorliegen und interpretiert den Output des Individuums als Binärzahl und vergleicht ihn mit der Klassifikation.

6.6 Rahmenklassen

Um die MNIST-Datensätze in CSV-Format einzulesen, kann die CSVReader-Klasse verwendet werden. Sie liest entweder das Trainings- oder das Test-Datenset von MNIST ein

und gibt es als Liste von InputWithClassification-Objekten zurück. Die StatisticsCalculator-Klasse berechnet zu übergebenen Integer-Werten Statistiken und wird in PECGP genutzt, um Statistiken über erreichte Fitness-Werte auszugeben. Die SaveandLoadIndividuals-Klasse ist für das Speichern und Laden verschiedener Individuen zuständig. Diese Klassen wurden bereits in Kapitel 3 vorgestellt und beschrieben.

Die ECGP_NetHandler-Klasse ermöglicht eine leichtere Benutzbarkeit von PECGP. Sie benötigt alle für die Ausführung von PECGP wichtigen Parameter (aufgelistet in Abbildung 6.1). Sie ermöglicht es dann, mit diesen Parametern einen Lauf in PECGP zu starten. Desweiteren ist es möglich, mehrere Läufe hintereinander zu starten und sich dann die Statistiken zu diesen ausgeben zu lassen. Diese Klasse ist, abgesehen von zusätzlichen Parametern, identisch zu der CGP_NetHandler-Klasse, welche in Kapitel 3 bereits näher erklärt wurde.

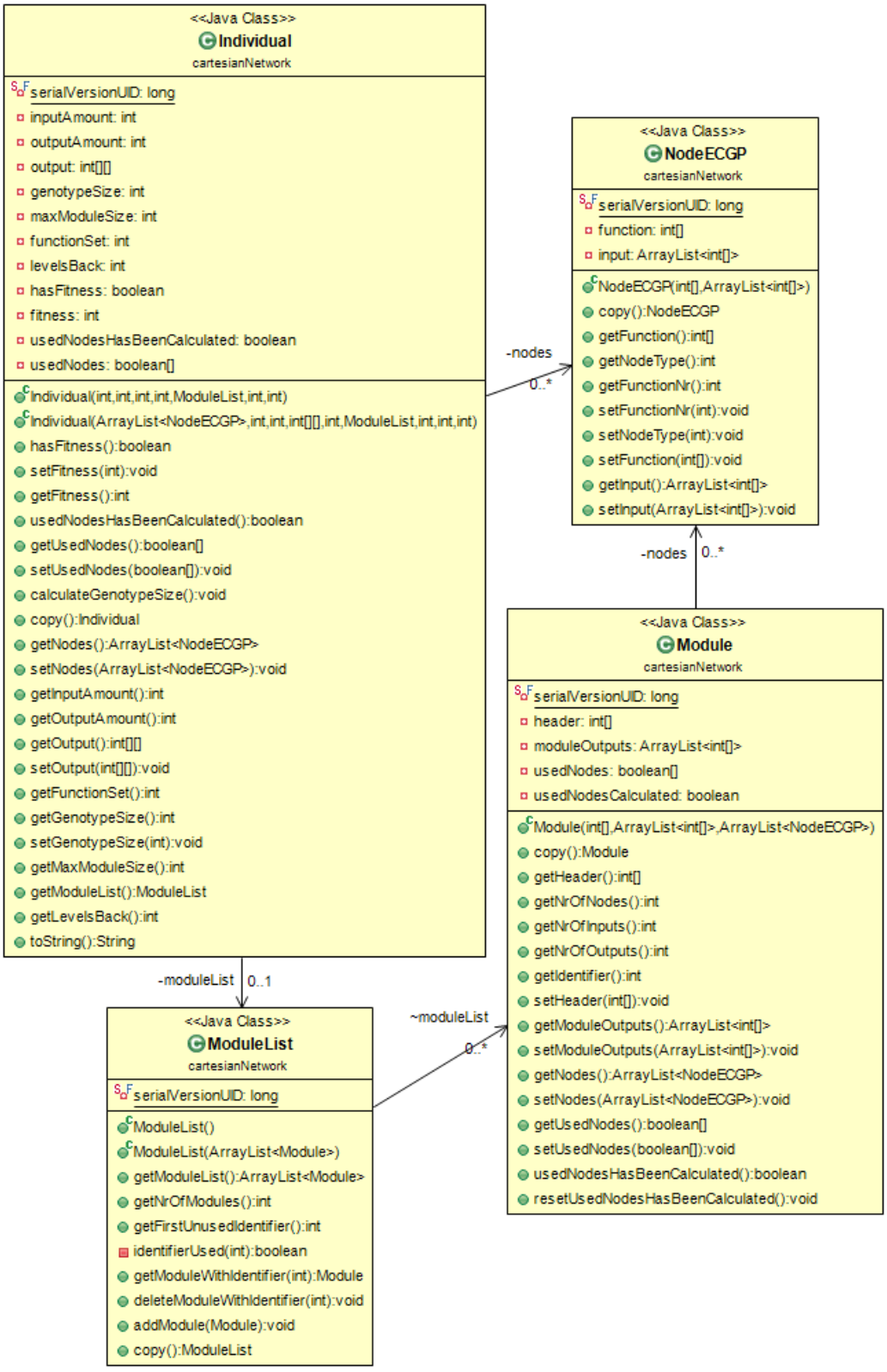


Abbildung 6.3: Klassendiagramm Netzwerk

Abbildung 6.4: Klassendiagramm Evolution

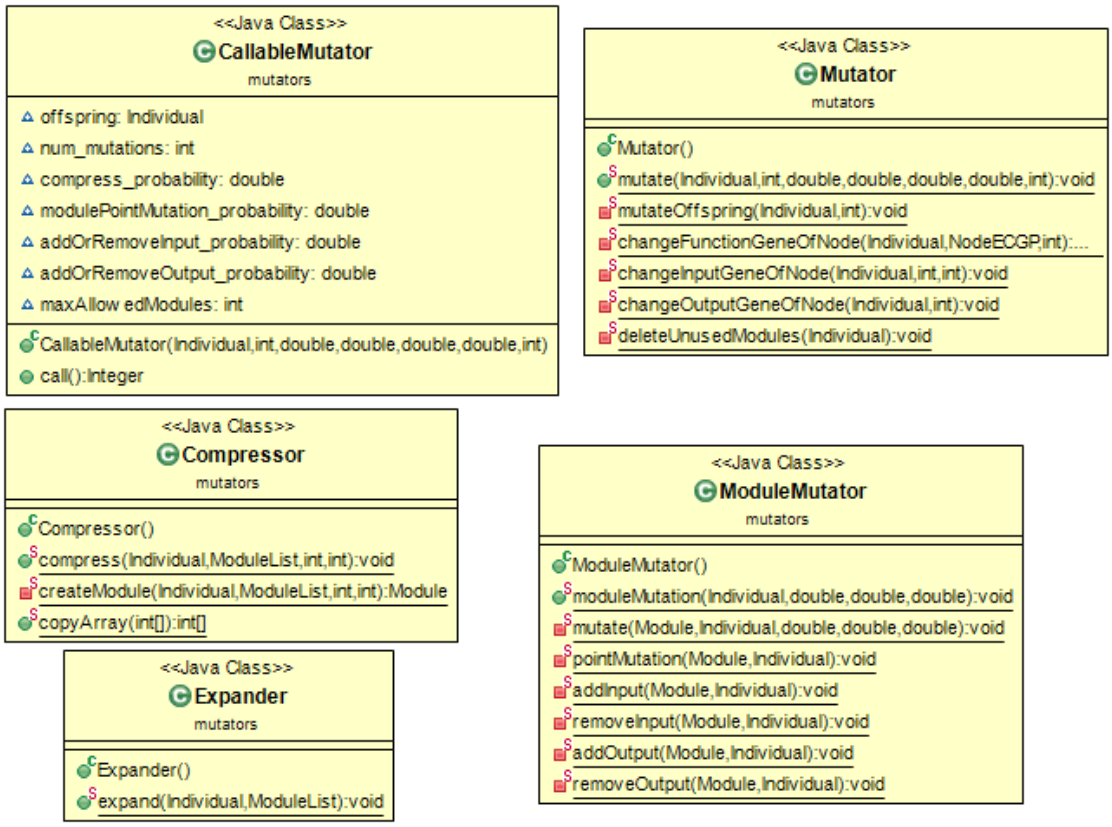


Abbildung 6.5: Klassendiagramm Mutation

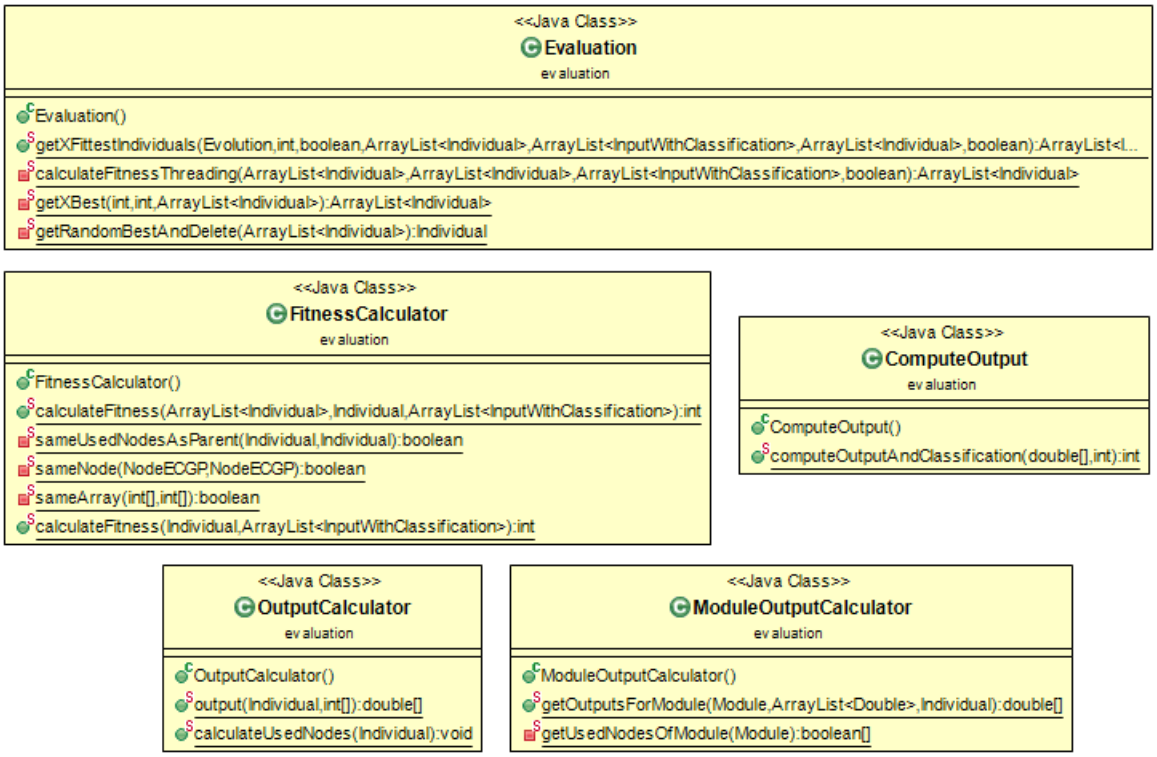


Abbildung 6.6: Klassendiagramm Evaluation

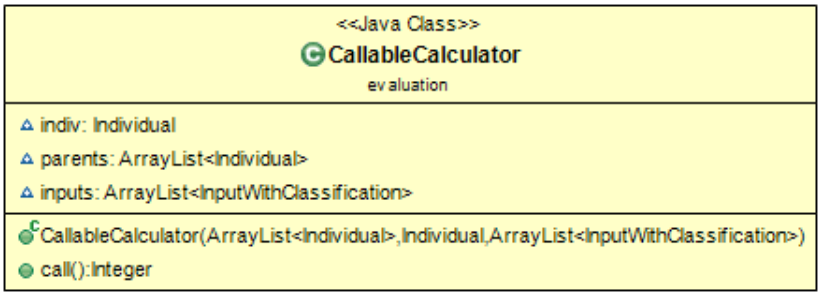


Abbildung 6.7: Klassendiagramm CallableCalculator

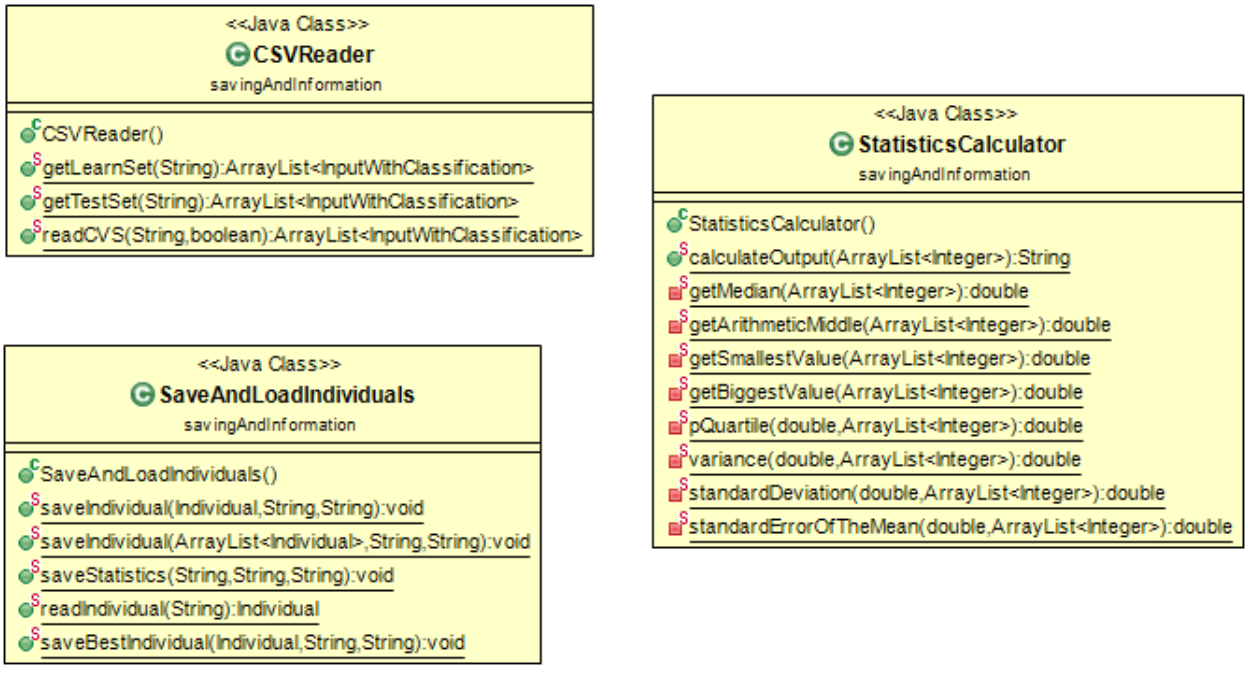


Abbildung 6.8: Klassendiagramm Data-I/O

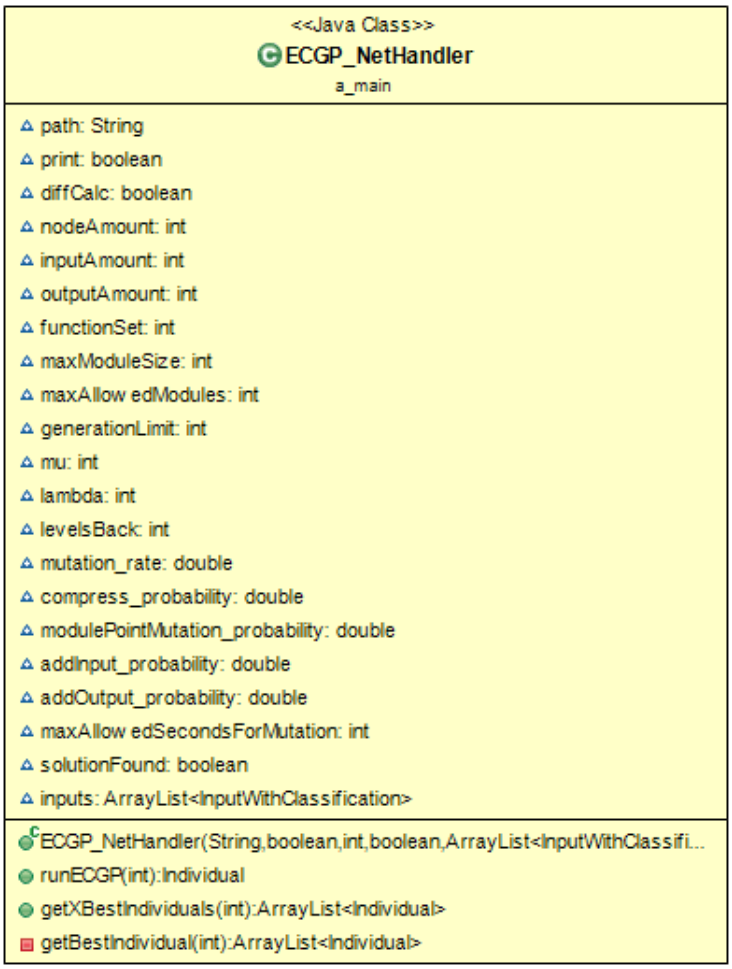


Abbildung 6.9: Klassendiagramm ECGP_NetHandler

Kapitel 7

Ergebnisse PECGP

In diesem Kapitel wird PECGP verwendet. Es wurde im vorherigen Kapitel vorgestellt und wird hier genutzt, um das Problem der handschriftlichen Ziffernerkennung zu lösen. Es werden in diesem Kapitel einzelne Versuche erläutert und analysiert. Die Statistiken, die zu jedem Versuch angegeben werden, sind die gleichen wie bereits in Kapitel 4. In diesem Kapitel wurden alle Tests aufgrund der erhöhten Laufzeit von PECGP zu PCGP 30 mal ausgeführt.

Das Ziel dieses Kapitels ist es, die einzelnen Parameter von ECGP auf ihre Nützlichkeit in Bezug auf das MNIST-Klassifikationsproblem zu testen. Dabei wird versucht, jeden Parameter durch verschiedenste Tests möglichst zu optimieren. Am Schluss soll sich zeigen, inwiefern sich ECGP nutzen lassen kann, um das gegebene Klassifikationsproblem zu lösen.

7.1 Parameterwahl

Zu Beginn wurden die Parameter festgelegt, mit denen getestet wurde. Da in Kapitel 4 bereits optimale Parameterwerte für CGP bestimmt wurden, wurden diese hier übernommen. Die Funktions-Sets, die in PECGP verwendet werden, sind die gleichen wie in PCGP. Außerdem werden die Bilder vor der Verarbeitung verkleinert und in Schwarz-Weiß-Bilder überführt, da sich gezeigt hat, dass dies vorteilhaft für die Fitness ist.

Die genutzten Parameter, die in allen Tests benutzt wurden, sind in Abbildung 7.1 aufgelistet.

Außerdem wurden alle Tests mit einer Generationsanzahl von 2000 ausgeführt, da in PECGP die einzelnen Durchläufe wesentlich länger brauchen als entsprechende Läufe in PCGP. Dies ist der Fall, da in PECGP auf die variable Genotyp-Größe sowie die größere Mutations-Phase Rücksicht genommen werden muss.

- Funktions-Set = 6
- Mutations-Rate = 6%
- Knoten-Anzahl = 1000
- Levels-Back = -1
- Output-Anzahl = 1
- Strategie = (1+4)

Abbildung 7.1: Parameterwahl für PECGP-Versuche

Generationen = 2000; Modul-Punkt-Mutationswarsch. = 4%; Input-Hinzufügen-Warsch. = 1%; Output-Hinzufügen-Warsch. = 1%; Max.-Modul-Größe = 10;

Compress-Warsch.:	5%	10%	20%	30%	40%	50%
Median	45391,5	45315	44619,2	45049	44712	45512,5
Arithm. Mittel	45987,23	45839,03	44919,1	45911,9	44639,5	46858,37
Standardfehler	614,52	735,24	520,29	638,38	577,06	815,72
Kleinster Wert	40863	39000	40315	40960	38073	40290
Größter Wert	54075	54077	54077	54077	54077	54102
Unteres Quartil	44052	42720	42802	43691	42777	44239
Oberes Quartil	47539	47429	46852	47539	46252	48784
Standardabw.	3365,88	4027,09	2849,72	3496,53	3160,67	4467,86

Abbildung 7.2: Ergebnisse der Compress-Raten-Versuche

7.2 Testen der Compress-Rate

Begonnen wurden die Tests mit dem Testen der Compress-Wahrscheinlichkeit. Es wurden Wahrscheinlichkeiten von 5% bis 50% stichprobenartig getestet. Die restlichen Werte wurden nach den üblichen Belegungen in [18] festgelegt und sind in 7.2 ersichtlich.

Bei diesen Versuchen dienten die Tests mit einer Compress-Wahrscheinlichkeit von 50% als Vergleichs-Versuche. So wurden bei diesen Versuchen keine Module verwendet, weil die Expand-Wahrscheinlichkeit 2*50% = 100% beträgt. Kreierte Module werden direkt wieder in den Genotypen eingefügt, ohne dass sie eine Chance hatten, von anderen Knoten genutzt zu werden. Es zeigte sich, dass die Compress-Rate keinen großen Einfluss auf die Fitness besaß. Jedoch wurden mit einer Mutations-Rate von 20% und 40% die besten Ergebnisse erreicht. So werden bei einer Mutations-Rate von 20% nicht so häufig Module erzeugt wie bei 40%. Jedoch ist bei der höheren Compress-Rate der Druck auf Module höher,

Generationen = 2000; Compress-Wahrscheinlichkeit = 20%; Input-Hinzufügen-Warsch. = 1%; Output-Hinzufügen-Warsch. = 1%; Max.-Modul-Größe = 10;

Modul-Punkt-Mutation:	1%	2%	4%	6%	8%
Median	44518	45014,5	44953,5	44996,5	44762
Arithm. Mittel	45228,43	45565,83	45831,57	45767,87	45350,63
Standardfehler	657,58	535,31	636,01	664,14	398,19
Kleinster Wert	38671	41522	39571	39227	42687
Größter Wert	54078	54077	54078	54077	51407
Unteres Quartil	43162	43888	44053	44192	44364
Oberes Quartil	47409	46158	46979	47442	46283
Standardabw.	3601,73	2932,02	3483,57	3637,65	2180,96

Modul-Punkt-Mutation:	10%	20%	40%	50%
Median	45373	44256,5	44215	43467,5
Arithm. Mittel	4598,66	45375,37	44655,6	43800,37
Standardfehler	768,12	673,89	608,46	564,48
Kleinster Wert	38135	40668	39756	38476
Größter Wert	54077	54077	54077	54075
Unteres Quartil	43556	43094	42768	41672
Oberes Quartil	46852	47271	46292	44749
Standardabw.	4207,16	3691,84	3332,66	3091,81

Abbildung 7.3: Ergebnisse der Modul-Punkt-Mutations-Raten-Versuche

dass sie expandiert werden. Es wurde sich entschieden, mit einer Compress-Rate von 20% weiterzutesten, da dies in [18] vorgeschlagen wurde.

Im Weiteren wurden andere Parameter getestet, mit denen sich die Fitness weiter verbessern lassen kann.

7.3 Testen der Modul-Punkt-Mutations-Wahrscheinlichkeit

In diesem Abschnitt wurden verschiedene Modul-Punkt-Mutations-Wahrscheinlichkeiten getestet. Da eine 4%ige-Wahrscheinlichkeit vorgeschlagen wird, werden Werte, die in der Nähe dieses Wertes liegen, getestet. [18] Die Ergebnisse sind in der ersten Tabelle in Abbildung 7.3 aufgelistet.

Aus diesen Versuchen zeigte sich, dass mit Wahrscheinlichkeiten von bis zu 8% keine großen Unterschiede erreicht werden. So bewegen sich alle Ergebnisse in einem ähnlichen Bereich. Eine mögliche Ursache dafür ist, dass die gewählten Wahrscheinlichkeiten sehr nah beieinander liegen. Im Anschluss daran wurden aus diesem Grund Wahrscheinlichkeiten

getestet, die weiter auseinander liegen. Die Ergebnisse dieser Versuche mit Wahrscheinlichkeiten bis zu 50% sind in der zweiten Tabelle in Abbildung 7.3 aufgelistet.

Aus diesen Versuchen ist erkennbar, dass mit einer höheren Modul-Punkt-Mutations-Wahrscheinlichkeite auch bessere Fitness-Werte einhergehen. So wird mit höheren Werten die Wahrscheinlichkeit erhöht, dass sich Module mutieren. Dies ist vorteilhaft, da die Varietät in Modulen erhöht wird. Mit einer höheren Varietät wird die Chance erhöht, dass Module sich ähnlich zum Genotypen des Individuums eher in Bezug auf die Fitness mutieren, da sie öfter mutiert werden. Möglicherweise wären höhere Wahrscheinlichkeiten besser für die Fitness-Entwicklung von Individuen. In diesem Kapitel soll sich auf die Wirkung anderer Parameter konzentriert werden, um die Wirkungen dieser beurteilen zu können.

In den folgenden Versuchen wurde mit einer Modul-Punkt-Mutations-Wahrscheinlichkeit von 40% weitergetestet. Tests mit einer Wahrscheinlichkeit von 50% waren zu diesem Zeitpunkt noch nicht abgeschlossen. Später in diesem Kapitel werden sie jedoch verwendet. Der nächste Abschnitt befasst sich zunächst mit den Auswirkungen auf die Laufzeit, die mit der Verwendung von PECGP einhergehen und den Möglichkeiten, diese zu beeinflussen.

7.4 Laufzeit

In den durchgeführten Läufen zeigte sich, dass die Laufzeit von PECGP höher als die Laufzeit von PCGP war. Dies liegt an der variablen Genotyp-Größe und der aufwendigeren Mutation. Durch die von PECGP eingeführten Veränderungen wird im allgemeinen eine Lösung in weniger Generationen erreicht [18]. Es wird sich zeigen, ob dies die erhöhte Laufzeit rechtfertigt. Des Weiteren ist die Laufzeit von PECGP nicht so konsistent wie in PCGP. Dies liegt daran, dass die unterschiedliche Genotyp-Größe von PECGP unterschiedliche Laufzeiten hervorbringt, da die Berechung von größeren Genotypen aufwendiger ist.

Des Weiteren wurden bei verschiedenen Versuchen Möglichkeiten getestet, die Laufzeit zu verringern. Es wurden Versuche gestartet, in denen die Mutation in Threads berechnet wurde. So zeigte sich, dass die Verwendung von Threads die Laufzeit erhöhte. Außerdem zeigte sich, dass wenn aktive Knoten von Kindern und Eltern verglichen werden, die Laufzeit ebenfalls ansteigt. Dies könnte daran liegen, dass sich Individuen in einer Generation zu sehr verändern, um noch die gleichen Knoten wie ein Elternteil zu besitzen. So könnte das Vergleichen von Knoten sich positiv auf die Laufzeit auswirken, wenn in einer Generation nicht viele Änderungen im Genotypen auftreten. Dies ist zum Beispiel bei niedrigeren Mutations-Raten der Fall. Da dies jedoch nicht zutrifft, wurde in den folgenden Versuchen die Mutation nicht in Threads ausgeführt und die aktiven Knoten nicht verglichen.

Generationen = 2000; Compress-Wahrscheinlichkeit = 20%; Modul-Punkt-Mutationswarsch. = 40%; Output-Hinzufügen-Warsch. = 1%; Max.-Modul-Größe = 10;

Input-Hinzufügen-Warsch.:	0,5%	1%	2%	5%	10%
Median	44822	44031	44125	43941,5	43147
Arithm. Mittel	44983,13	44709,77	44633,10	45337,27	44254,47
Standardfehler	563,90	647,96	572,31	741,59	624,09
Kleinster Wert	38601	39793	41050	40261	39108
Größter Wert	54077	54970	55261	54077	54069
Unteres Quartil	43326	42541	42711	42238	42251
Oberes Quartil	45873	45746	46046	48345	46208
Standardabw.	3088,60	3549,04	3134,68	4061,85	3418,29

Abbildung 7.4: Ergebnisse der Input-Hinzufügen-Warsch.-Versuche

7.5 Testen weiterer Modul-Wahrscheinlichkeiten

In diesem Abschnitt wurden die Wahrscheinlichkeiten, einen Input oder Output hinzuzufügen, getestet. Begonnen wurde mit der Wahrscheinlichkeit für den Input-Hinzufügen-Wert. Die Tests sind in Abbildung 7.4 ersichtlich.

Es zeigt sich, dass eine höhere Wahrscheinlichkeit, einen Input hinzuzufügen, durchschnittliche bessere Fitness-Werte erzielte. Dies könnte daran liegen, dass Module so mehr Zugang zu anderen Knoten besitzen und der Graph besser verbunden ist. Außerdem könnten zusätzliche Inputs von Modulen, die nicht zur Berechnung beitragen, wahrscheinlicher gelöscht werden. Dies ist jedoch nicht häufig der Fall und es zeigte sich, dass eine Erhöhung der Input-Hinzufügen-Wahrscheinlichkeit eine Erhöhung der Laufzeit bedeutete. So waren Läufe mit einer höheren Wahrscheinlichkeit wesentlich langsamer als andere. Dies liegt daran, dass ein Individuum mit Modulen mit vielen Inputs einen viel größeren Genotypen besitzt als vergleichbare Individuen mit wenigen Modul-Inputs. Diese Vergrößerung des Genotyps macht sich bei einer Wahrscheinlichkeit von 10% bemerkbar. So dauerte ein Lauf mit 10%-Wahrscheinlichkeit meistens mehr als vierfach so lang wie der längste Lauf mit einer 5%igen Wahrscheinlichkeit. Aus diesem Grund wurde für die weiteren Läufe eine Wahrscheinlichkeit von 5% gewählt, da diese unter den restlichen Läufen die beste durchschnittliche Fitness besaß.

Gleichzeitig zu den Tests für eine Input-Hinzufügen-Wahrscheinlichkeit wurden Tests für die Output-Hinzufügen-Wahrscheinlichkeit ausgeführt. Da zu diesem Zeitpunkt keine bessere Wahrscheinlichkeit bekannt war, wurden diese Läufe mit einer Input-Hinzufügen-Wahrscheinlichkeit von 1% ausgeführt. Getestet wurden die gleichen möglichen Wahrscheinlichkeiten wie zuvor. Die Ergebnisse der Versuche sind in 7.5 aufgelistet.

Generationen = 2000; Compress-Wahrscheinlichkeit = 20%; Modul-Punkt-Mutationswarsch. = 40%; Input-Hinzufügen-Warsch. = 1%; Max.-Modul-Größe = 10;

Output-Hinzufügen-Warsch.:	0,5%	1%	2%	5%	10%
Median	45061	43812	44006	44512	44377
Arithm. Mittel	45783,17	44564,30	43888,50	45149,97	45556,07
Standardfehler	635,70	694,66	453,14	694,51	710,47
Kleinster Wert	40031	38132	39662	38563	40797
Größter Wert	54079	54077	48385	54077	54078
Unteres Quartil	43754	42309	41892	42444	43071
Oberes Quartil	47723	45516	45342	47218	46756
Standardabw.	3481,88	3804,82	2481,92	3804,00	3891,39

Abbildung 7.5: Ergebnisse der Output-Hinzufügen-Warsch.-Versuche

Aus den Ergebnissen ist erkennbar, dass eine Änderung der Output-Hinzufügen-Wahrscheinlichkeit keine großen Auswirkungen auf die Fitness-Entwicklung besitzt. Jedoch wird bei höheren Output-Hinzufügen-Wahrscheinlichkeiten nicht die Laufzeit erhöht, wie es zuvor der Fall war. Dies liegt daran, dass bei Hinzufügen und Entfernen eines Outputs die Genotyp-Größe nicht verändert wird. Die besten durchschnittlichen Fitness-Werte werden bei Wahrscheinlichkeiten von 1% und 2% erzielt. Für die abschließenden Tests wurde deshalb eine Wahrscheinlichkeit von 2% gewählt.

7.6 Testen der maximalen Modul-Größe

In diesem Abschnitt wurde der Parameter der maximalen-Modulgröße auf seine Auswirkung auf den Fitness-Wert hin untersucht. Dabei wurden für Input- und Output-Hinzufügen-Wahrscheinlichkeiten beide Male 1% gewählt, da zu Zeiten dieser Versuche die besseren Parameter noch nicht vorlagen. Getestet wurden im Folgenden maximale Modul-Größen im Bereich zwischen fünf und fünfzig, um so einen relativ großen Bereich abzudecken. Die Ergebnisse sind in 7.6 einzusehen.

So zeigt sich das maximale-Modul-Größen von fünf eindeutig bessere Ergebnisse erzielten als höhere Modul-Größen. Dies kann mehrere Gründe haben. Zum einen kann durch kleinere Module sichergestellt werden, dass Module nur die Knoten enthalten, die für die Verarbeitung notwendig sind und nur wenige nicht-aktive Knoten enthalten. Des Weiteren werden durch kleinere maximale-Modul-Größen öfter Module erstellt. Dies ist der Fall, da beim Compress-Operator kleinere Bereiche gewählt werden und so die Wahrscheinlichkeit geringer ist, dass Module des Typen 1 oder 2 im gewählten Bereich liegen. Bei größeren Modul-Größen ist außerdem die Laufzeit länger, da Module mehr Inputs enthalten kön-

Generationen = 2000; Compress-Wahrscheinlickeit = 20%; Modul-Punkt-Mutationswarsch. = 50%; Input-Hinzufügen-Warsch. = 1%; Output-Hinzufügen-Warsch. = 1%;

Max.-Modul-Größe:	5	10	20	35	50
Median	42793	43467,5	45508,5	44951	45321
Arithm. Mittel	43732,33	43800,37	46112,90	45852,52	45295,67
Standardfehler	491,94	564,48	738,02	762,96	567,61
Kleinster Wert	40351	38476	38676	38634	40889
Größter Wert	54074	54075	54075	54077	54077
Unteres Quartil	42115	41672	43173	42821	42797
Oberes Quartil	44987	44749	46827	47440	46130
Standardabw.	2694,94	3091,81	4042,31	4108,64	3108,94

Abbildung 7.6: Ergebnisse der Max.-Modul-Größe-Versuche

nen und so die Genotyp-Größe ansteigt. Aus diesen Gründen wird in den abschließenden Versuchen eine Max.-Modul-Größe von fünf gewählt.

7.7 Abschlusstests

Mit den vorher bestimmten Werten wurden abschließende Tests durchgeführt. Dabei wurden die vorgeschlagenen Parameter von ECGP getestet und mit den optimierten verglichen. [18] Des Weiteren wurde die maximale Generationsanzahl auf 8.000 Generationen erhöht. Außerdem wurde dazu ein Test mit PCGP und 8.000 Generationen gestartet, um die Ergebnisse von CGP und ECGP zu vergleichen. Die Ergebnisse der Versuche sind in 7.7 aufgelistet. Dabei sind die Parameter für CGP in 7.1 aufgelistet, diese werden auch für die ECGP-Läufe genutzt.

7.8 PECGP - Fazit und Schlussfolgerungen

Zusammenfassend zeigt sich, dass ECGP eine gute Erweiterung zu CGP darstellt, um auf das Klassifikationsproblem angewandt zu werden. Jedoch treten Schwierigkeiten auf. Die Läufe mit ECGP besitzen eine längere Laufzeit als die mit CGP, weshalb fraglich ist, ob sich damit schneller bessere Ergebnisse erzielen lassen. So wurden in ECGP mit derselben Generationsanzahl bessere Ergebnisse erzielt, jedoch wurde mehr Zeit benötigt, um diese Ergebnisse zu erhalten. Mit einer weiteren Optimierung der Parameter könnten bessere Fitness-Werte erreicht werden. Ebenfalls wurde eine minimale Gesamtverbesserung der Ergebnisse durch die durchgeführten Tests erzielt. Weiterhin könnte getestet werden, ob ECGP mit einer größeren Generationsanzahl schneller zu Ergebnissen kommt als CGP. Mit ECGP treten insbesondere Schwierigkeiten auf, die auch schon bei CGP auftraten. So

ECGP-1
Generationen = 8000; Compress-Wahrscheinlichkeit = 20%; Modul-Punkt-Mutationswarsch. = 4%; Input-Hinzufügen-Warsch. = 1%; Output-Hinzufügen-Warsch. = 1%; Max.-Modul-Größe = 10;
ECGP-2
Generationen = 8000; Compress-Wahrscheinlichkeit = 20%; Modul-Punkt-Mutationswarsch. = 50%; Input-Hinzufügen-Warsch. = 5%; Output-Hinzufügen-Warsch. = 2%; Max.-Modul-Größe = 5;

Versuch:	CGP	ECGP-1	ECGP-2
Median	42039	40622,5	39699
Arithm. Mittel	41809,33	42324,70	40938,86
Standardfehler	575,25	949,97	896,9
Kleinster Wert	37002	36910	35823
Größter Wert	47539	53786	54077
Unteres Quartil	38837	38417	38823
Oberes Quartil	43844	44456	40869
Standardabw.	3150,76	5203,2	4829,95

Abbildung 7.7: Ergebnisse der abschließenden Tests mit PECGP

könnte die Fitness mit der Erstellung eines Funktionssets wesentlich erhöht werden, wenn dieses auf das Problem zugeschnitten wird. Außerdem kommen viele einzelne Tests nicht über eine Klassifikationsrate von 10%, wie es bei CGP schon der Fall war. Für die weitere Anwendung auf dem MNIST-Datensatz wird deshalb PCGP empfohlen, da mit diesem wesentlich schneller Ergebnisse erreicht wurden.

Kapitel 8

Zusammenfassung und Ausblick

Insgesamt zeigt sich, dass Cartesian Genetic Programming durchaus genutzt werden kann, um das Problem der handschriftlichen Ziffernerkennung auf dem MNIST-Datensatz zu lösen.

Es konnten mithilfe von PCGP Individuen erzielt werden, die mehr als die Hälfte des Datensatzes korrekt klassifizierten. Diese wurden mithilfe einzelner Tests erreicht, die jeden Parameter von CGP auf seine Auswirkungen auf die Fitness prüften. So konnten die Funktions-Sets bestimmt werden, die, ohne auf das Problem zugeschnitten worden zu sein, die beste Fitness ermöglichten. Außerdem zeigte sich, dass eine Veränderung der Mutations-Rate erst bei höherer Generationsanzahl ausschlaggebend ist. Es wurde außerdem herausgefunden, dass eine Erhöhung der maximalen Generationsanzahl und der Anzahl an Knoten innerhalb eines Individuums nützlich ist. Diese erhöhen die Laufzeit, jedoch sind sie vorteilhaft, um bessere Ergebnisse erzielen zu können. Es zeigte sich, dass gerade durch vorherige Bildbearbeitung die Ergebnisse wesentlich verbessert werden konnten. Mithilfe der Bildbearbeitung konnte das Problem vermindert werden, das bei CGP und großen Eingaben auftritt. So verbinden sich viele Knoten in CGP bei großen Eingaben eher mit der Eingabe als mit anderen Knoten. Der levels-back-Parameter verstärkt dieses Problem enorm und es wurde gezeigt, dass dieser bei Anwendung beim MNIST-Klassifikationsproblem die Fitness-Entwicklung behindert. Da im MNIST-Datensatz mit wenigen Knoten Klassifikationsraten von 10 bis 20% erreicht werden können, ist es schwer, komplexere Berechnungen zu erreichen. Dies zeigte sich besonders dadurch, dass in vielen Versuchen der schlechteste Wert fast immer eine Klassifikationsrate von 10% erreichte und diese nicht überschreiten konnte. Mit all diesen Erkenntnissen konnten optimierte Läufe gestartet werden, die eine Klassifikationsrate von mehr als 50% erzielten. Diese könnte weiter verbessert werden, wenn ein Funktions-Set entwickelt wird, welches auf das MNIST-Klassifikationsproblem zugeschnitten ist. Mithilfe dieses und weitere Optimierung der Parameter könnte sehr wahrscheinlich das MNIST-Datenset komplett korrekt klassi-

fiziert werden und Ergebnisse erreicht werden, deren Klassifikationsrate vergleichbar mit Ergebnissen aus [4] sind.

Des Weiteren konnte die Effektivität von Embedded Cartesian Genetic Programming für die Klassifikation der Ziffern des MNIST-Datensatzes gezeigt werden. Dabei wurde ECGP durch die Verwendung von PECGP getestet. Mithilfe von Tests für jeden einzelnen Parameter konnte dessen Wirkung auf Fitness und Laufzeit aufgezeigt werden. Es zeigt sich, dass die Laufzeit von PECGP sehr stark von der Parameterwahl abhängig war. Parameter, die die Größe des Gentoypen vergrößerten, erhöhten die Laufzeit, wie es zum Beispiel beim Input-Hinzufügen-Parameter ersichtlich war. Im Vergleich zu PCGP erreichten Versuche mit gleichen Parametern bessere Ergebnisse, jedoch wird aufgrund der erhöhten Laufzeit von ECGP geraten, zur Klassifikation des MNIST-Datensatzen PCGP zu verwenden. Die Auswirkungen von ECGP könnten jedoch verbessert werden, wenn ein spezialisiertes Funktions-Set genutzt wird und die Parameter weiter optimisiert werden. Des Weiteren konnten die Auswirkungen von ECGP bei einer hohen Generationsanzahl nicht aufgezeigt werden und es bleibt offen, ob die positiven Auswirkungen von ECGP die Erhöhung der Laufzeit der einzelnen Generationen rechtfertigen.

Das Ziel dieser Arbeit, einen ersten Ansatz vorzustellen, den MNIST-Datensatz mithilfe von Cartesian Genetic Programming zu klassifizieren, wurde erreicht. Mithilfe von eigens vom Autor entwickelten Frameworks kann leicht darauf aufgebaut werden, um die Ergebnisse weiter zu verbessern, um das Klassifizierungsproblem zu lösen. Es wurden Probleme aufgezeigt, die bei der Klassifikation entstanden und diese wurden versucht, bestmöglich zu beheben. Auch die Erstellung eines Klassifikators konnte erreicht werden, welcher leicht weiterverwendet werden kann.

Zusammenfassend wurde ein erster Ansatz zur Problemlösung vorgestellt und Frameworks eingeführt, die eine Weiterarbeit an diesem Problem ermöglichen. Die besten Versuche sind der Abbildung 4.13 zu entnehmen.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *JavaPoet*. https://github.com/square/javapoet. [zuletzt aufgerufen am 19.08.2018].

[2] *technische universität dortmund, Lehrstuhl 11*. https://ls11-www.cs.tu-dortmund.de/de/start. [zuletzt aufgerufen am 26.08.2018].

[3] Harding, Simon, Jürgen Leitner und Jürgen Schmidhuber: *Cartesian Genetic Programming for Image Processing*, Seiten 31–44. Springer New York, New York, NY, 2013.

[4] LeCun, Y., L. Bottou, Bengio Y. und Haffner P.: *Gradient-Based Learning Applied to Document Recognition*. Proceedings of the IEEE, 1998.

[5] LeCun, Y., C. Cortes und C.-J.-C. Burges: *MNIST handwritten digit database*. http://yann.lecun.com/exdb/mnist/. [zuletzt aufgerufen am 19.08.2018].

[6] Leitner, J., S. Harding, A. Förster und J. Schmidhuber: *Mars terrain image classification using Cartesian genetic programming*. Conference Paper, Dalle Molle Institute for Artificial Intelligence (IDSIA), SUPSI and Università della Svizzera Italiana (USI), Lugano, Switzerland, 2012.

[7] Luke, S., E.-O. Scott, L. Panait, G. Balan, S. Paus, Z. Skolicki, R. Kicinger, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, A. Desai, A. Chircop, J. Compton, W. Haddon, S. Donelly, B. Jamil, J. Zelibor, E. Kangas, F. Abidi, H. Mooers, J. O'Berine, K.-A. Talukder, S. McKay, J. McDermott, J. Zou, A. Rutherforf, D. Freelan und E. Wei: *ECJ - A Java-based Evolutionary Computation Research System*. https://cs.gmu.edu/ eclab/projects/ecj/. [zuletzt aufgerufen am 19.08.2018].

[8] Miller, J.-F.: *CGP home*. http://www.cartesiangp.co.uk/index.html. [zuletzt aufgerufen am 19.08.2018].

[9] Miller, J.-F.: *CGP Resources*. http://www.cartesiangp.co.uk/resources.html. [zuletzt aufgerufen am 23.08.2018].

[10] Miller, J.-F.: *An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach.* In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, Seiten 1135–1142, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[11] Miller, J.-F. (Herausgeber): *Cartesian Genetic Programming.* Springer-Verlag, Berlin Heidelberg, 2011.

[12] Miller, J.-F.: *Cartesian Genetic Programming.* In: Miller, J.-F. (Herausgeber): *Cartesian Genetic Programming*, Kapitel 2, Seiten 17–34. Springer-Verlag, Berlin Heidelberg, 2011.

[13] Miller, J.-F.: *Introduction to Evolutionary Computation and Genetic Programming.* In: Miller, J.-F. (Herausgeber): *Cartesian Genetic Programming*, Kapitel 1, Seiten 1–16. Springer-Verlag, Berlin Heidelberg, 2011.

[14] Oranchak, D.: *Cartesian Genetic Programming for the Java Evolutionary Computing Toolkit (CGP for ECJ).* http://www.oranchak.com/cgp/doc/. [zuletzt aufgerufen am 19.08.2018].

[15] Redmon, J.-C.: *MNIST in CSV.* https://pjreddie.com/projects/mnist-in-csv/. [zuletzt aufgerufen am 19.08.2018].

[16] Sekanina, L., S.-L. Harding, W. Banzhaf und Kowaliw: *Image Processing and CGP.* In: Miller, J.-F. (Herausgeber): *Cartesian Genetic Programming*, Kapitel 6, Seiten 181–215. Springer-Verlag, Berlin Heidelberg, 2011.

[17] Steppan, J.: *Wikipedia.* https://commons.wikimedia.org/w/index.php?curid=64810040. [zuletzt aufgerufen am 23.08.2018].

[18] Walker, J.-A., J.-F. Miller, P. Kaufmann und Platzner M.: *Problem Decomposition Cartesian Genetic Programming.* In: Miller, J.-F. (Herausgeber): *Cartesian Genetic Programming*, Kapitel 3, Seiten 35–99. Springer-Verlag, Berlin Heidelberg, 2011.

Eidesstattliche Versicherung

Piepenbrink, Björn	183595
Name, Vorname	Matr.-nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Code Generierung und Training eines CGP Klassifikators zur handschriftlichen Ziffernerkennung

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 27. August 2018	
Ort, Datum	Unterschrift

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/ die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Dortmund, den 27. August 2018	
Ort, Datum	Unterschrift