

Implementierung eines Neuroevolutionsalgorithmus zur Optimierung von Topologie und Hyperparametern eines künstlichen neuronalen Netzes

Ein genetischer Algorithmus zur Automatisierung
von Trainingsprozessen.

Fachpraktikumsbericht

von JUSTUS WILL

25. Mai 2020

Inhaltsverzeichnis

1	Einleitung	3
2	Problemstellung	4
2.1	Handschrifterkennung	4
2.2	Klassifikationsprobleme	4
2.3	Neuronale Netze	5
2.3.1	Neuronen	5
2.3.2	Fully Connected Layer	6
2.3.3	Convolutional Layer	7
2.3.4	Pooling Layer	8
2.4	Parameter und Training	9
2.5	Hyperparametersuche	9
2.6	bisherige Ansätze	10
2.6.1	Grid Search	10
2.6.2	Random Search	10
2.7	NEAT	11
3	convNEAT	12
3.1	Representation	12
3.2	Mutation	14
3.3	Selektion	15
3.4	Crossover	16
3.5	Clustering	17
3.5.1	Ähnlichkeit	18
3.5.2	Clusteralgorithmus	19
3.6	Eliten und Training	20
4	Ergebnisse	23
4.1	MNIST	25
5	Ausblick	27
	Referenzen	28

1 Einleitung

Der größer werdende Einfluss von künstlicher Intelligenz und maschinellem Lernen ist in unserer Gesellschaft nicht mehr zu leugnen. In vielen Bereichen haben sich sogenannte künstliche neuronale Netzwerke als vielseitiges Mittel zur Findung von Mustern und Lösung von Problemen erwiesen. Unter anderem in Spracherkennung, Bilderkennung, Übersetzung, Prognose und Vorhersage von Zeitreihen, Suchmaschinen oder in medizinischen Systemen werden alltäglich Lösungen eingesetzt, die auf einem gut trainiertem neuronalem Netz basieren. Eine Untergruppe der neuronalen Netze sind die *Convolutional Neural Networks*, welche die neuronalen Netze um Faltungsoperationen erweitern. Besonders bei Bild und Textverarbeitung sind diese Netze State-of-the-Art. Hierbei muss für jede neue Aufgabe ein neues Netz angepasst und auf großen Datenmengen trainiert werden. Geeignete Hyperparameter zu finden ist oft ein schwieriges Problem, das in der Regel von Experten übernommen werden muss. Durch systematisches Ausprobieren und Testen optimieren diese auf Basis ihrer Expertise und Erfahrung per Hand die Parameter der Netze. Der entwickelte Algorithmus *convNEAT* automatisiert diesen Prozess. Durch einen genetischen Evolutionsalgorithmus werden gute Kandidaten generiert, selektiert, trainiert und weiter verbessert. Dies geschieht problemunabhängig und ermöglicht mit genug Rechenleistung automatisch auf eine gleiches, wenn nicht sogar besserer Ergebnis zu kommen, als wenn die Hyperparametersuche mit anderen Mitteln geschieht.

2 Problemstellung

Bevor wir uns genauer mit dem der Problemstellung der Optimierung von Topologie und Hyperparametern von neuronalen Netzen (*Neural Networks*) beschäftigen folgte zuerst eine kleine Einführung in neuronale Netze. Um besser verstehen zu können wofür neuronale Netze verwendet werden betrachten wir zunächst beispielhaft ein weitverbreitetes *Benchmarkproblem* im Bereich des maschinellen Lernens, dass später mit unserem Algorithmus gelöst werden kann.

2.1 Handschrifterkennung

Eine Aufgabe für die sich neuronale Netze hervorragend eignen ist die Handschrifterkennung. Das Problem besteht darin eine Zahl, die vorher noch nicht gesehen wurde, nur anhand eines Bildes richtig zu klassifizieren, also auszugeben, was für eine Zahl abgebildet ist. Für Menschen ist diese Aufgabe mühelos lösbar, aber würde man ohne lernbasierte Methoden einen Algorithmus zur Erkennung schreiben wollen, so wäre dies sehr kompliziert. Neuronale Netze bieten hier eine simple und einfache Lösung.

Der MNIST Datensatz [Y L98] besteht aus 70.000 Schwarz-Weiß-Bildern von Zahlen in einer Auflösung von 28x28 Pixeln. Die Aufgabe besteht darin mit Hilfe von 60.000 dieser Bildern (Trainingsdaten) einen Klassifizierer zu erstellen. Bewertet wird dieser danach, wie gut der Algorithmus die 10.000 weiteren vom Algorithmus noch nicht gesehenen Zahlen (Testdaten) klassifiziert.



Abbildung 1: Beispiele aus dem Datensatz MNIST [Y L98]

2.2 Klassifikationsprobleme

Die allgemeine Problemstellung die ein *Neural Network* lösen kann ist die folgende:

Gegeben sei eine Zielfunktion $f : \mathcal{H} \rightarrow \mathcal{G}$.

Da diese Funktion aber nicht bekannt ist, soll sie möglichst gut approximiert werden.

Hierzu muss aus einer Menge von Funktionen $f_p : \mathcal{H} \rightarrow \mathcal{G}$ mit $p \in \Theta$ aus dem Parameterraum, diejenige ausgewählt werden, die f am besten approximiert. In der Praxis ist es nicht leicht zu sagen, was es bedeutet, dass f_p eine gute Approximation für f ist. Für unsere Anwendung sind vorrangig Klassifikationsprobleme interessant, hier enthält \mathcal{G} alle verschiedenen Klassen und ist meistens endlich. Um in diesem Fall die Güte quantifizieren zu können, und um verschiedene Funktionen f_p vergleichen zu können, definieren wir die Genauigkeit auf den Testdaten, die aus dem Englischen *classification accuracy* auch kurz als *Accuracy* bezeichnet wird:

Definition 2.1 (Classification accuracy).

Zu der zu approximierenden Funktion f betrachten wir eine Menge von m Testdaten

$(t_i, f(t_i)) \in \mathcal{G} \times \mathcal{H} \ i \in \{1, \dots, m\}$ mit korrekt klassifizierten Punkten t_i .
Die Accuracy acc ist nun definiert als:

$$acc(f_p) = \frac{1}{m} \sum_{k=1}^m \delta_{f(t_i) f_p(t_i)} \text{ wobei } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{sonst} \end{cases}$$

Die Accuracy gibt also an welcher Anteil der Testdaten korrekt klassifiziert wird.

Um geeignete Parameter aus Θ zu finden, ohne das wir die Testdaten verwenden dürfen, gibt es einen Trainingsdatensatz mit n Daten $(x_i, f(x_i)) \in \mathbf{G} \times \mathbf{H} \ i \in \{1, \dots, n\}$, die ebenfalls bereits richtig klassifiziert sind. Aktuelle Methoden des maschinellen Lernens optimieren nun die Accuracy auf den Trainingsdaten, was auch die Accuracy auf den Testdaten verbessert. Letzteres gibt nach dem Training an, wie gut der Klassifizierer neuen Daten vorhersagen kann und dient als Vergleichsmetrik zwischen mehreren Klassifizierern.

Wenn wir erneut das Klassifikationsproblem MNIST betrachten ergeben sich insgesamt z.B:

$\mathcal{H} = \mathbb{R}^{28 \times 28}$, $\mathcal{G} = \{0, \dots, 9\}$ sowie $n = 60.000$ Trainingsdaten und $m = 10.000$ Testdaten.

2.3 Neuronale Netze

2.3.1 Neuronen

Um verstehen zu können was *neurale Netze* (im Verlauf auch kürzer *Netze* genannt) sind, schauen wir uns zuerst den Grundbaustein an, aus denen sie bestehen, die (*künstlichen*) *Neuronen*. Ein Neuron ist eine kleine Einheit die eine Menge Inputs x_1, \dots, x_n erhält und daraus einen Output y berechnet. Graphisch lässt sich das folgendermaßen vorstellen:

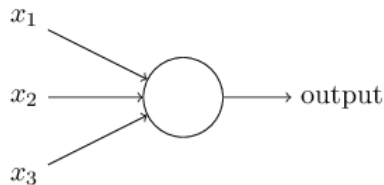


Abbildung 2: Ein Neuron [Nie19]

Definition 2.2 (Neuron).

Mathematisch gesehen handelt es sich bei einem Neuron um eine Funktion $y = g(\mathbf{x}) = \sigma(\sum_{j=1}^n \omega_j x_j + b)$ wobei σ die Aktivierungsfunktion des Neurons ist.

ω_j und b sind dabei die Gewichte und der Bias des Neurons, diese beiden beeinflussen, was genau das Neuron berechnet und müssen trainiert werden. Sie sind also Teil des Parameterraums Θ eines neuronalen Netzes. Sobald sie einmal festgelegt wurden ändern Sie sich

jedoch nicht mehr, sind also unabhängig von \mathbf{x} . Welches σ gewählt wird beeinflusst, wie sich ein Netz beim Training verhält und wird problemspezifisch angepasst. Es handelt sich um den ersten Hyperparameter, der beeinflusst wie die Funktion f_p für ein festes $p \in \Theta$ aussieht. Dazu später mehr. In der Praxis werden zwei Aktivierungsfunktionen häufig verwendet, Die Sigmoid- und die ReLU-Aktivierungsfunktion:

Definition 2.3. Die Sigmoidfunktion ist definiert als

$$\text{sig}(z) := \frac{1}{1 + e^{-z}}$$

Definition 2.4. Eine weitere Funktion ist die ReLU-Funktion:

$$\text{ReLU}(z) = \max(0, z)$$

Beide Funktionen haben Eigenschaften die sie zu guten Kandidaten für Aktivierungsfunktionen machen, da sie das schnelle und effektive Trainieren von neuronalen Netzen ermöglichen.

2.3.2 Fully Connected Layer

Ein neuronales Netz besteht aus sogenannten *Layern* von Neuronen. Das sind Schichten die viele Neuronen enthalten, die nicht untereinander, aber mit den Neuronen der benachbarten Schichten, bzw *Layern* verbunden sind.

Eine *Layer* fasst also die einzelnen Funktionen der m Neuronen g_i zu einer großen Funktion g zusammen. g operiert auf einem Vektor \mathbf{X} , der als Komponenten alle Outputs der vorherigen Layer enthält. Jedes g_i enthält immernoch seine eigenen Parameter $\omega_1^i, \dots, \omega_n^i$ und b^i .

In normalen neuronalen Netzen gibt es nur sogenannte *Fully Connected Layer* oder *Dense Layer*, das sind *Layer*, bei denen der Output jedes Neurons einer der Inputs jedes Neurons in der nächsten *Layer* ist. Die beiden *Layer* sind also vollständig miteinander verbunden. Eine *Fully Connected Layer* mit m Neuronen und Funktionen g_1, \dots, g_m hat also die oben angesprochene Funktion

$$g(\mathbf{X}) = (g_1(\mathbf{X}), \dots, g_m(\mathbf{X}))^T$$

Der Input der ersten Layer von Neuronen ist $\mathbf{x} \in \mathcal{H}$. Die einzelnen Komponenten von \mathbf{x} bilden ebenfalls eine *Layer*, die sogenannte *Input Layer*. Die letzte *Layer* aus Neuronen heißt auch *Output Layer*, da ihr Output die Komponenten von $f_p(\mathbf{x}) \in \mathcal{G}$ darstellt. Jede anderen *Layer* von Neuronen heißt *hidden (versteckte) Layer*. Schon mit nur einer *hidden Layer* kann man jede stetige Funktion z.B. bzgl. $\|\cdot\|_{sup}$ mit steigender Anzahl an Neuronen beliebig gut approximieren. Dieses *Universalitätstheorem* erklärt, warum sich neuronale Netze in so unterschiedlichen Problemen anwenden lassen.

Im Fall unseres MNIST Datensatzes wäre z.B. ein einfaches Netzwerk mit einer *hidden Layer* denkbar:

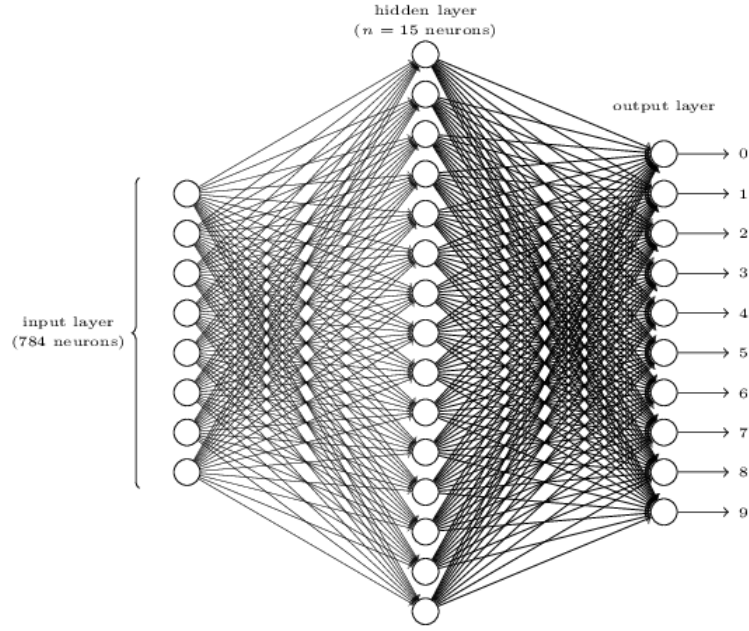


Abbildung 3: Teilausschnitt aus einem neuronalen Netz [Nie19]

2.3.3 Convolutional Layer

Da sich das Hauptaugenmerk unseres Algorithmus auf Klasifikationsproblemen in der Bilderkennung liegt, betrachten wir zudem die dort üblichen *Convolutional Neural Networks*. Diese erweitern die Idee der neuronalen Netze um eine weitere Art zwei *Layer* zu verbinden - die Faltung, im Englischen *Convolution*. Anders als bei den vollständig verbundenen *Layern* werden nur die räumlich lokalen Nachbarn zusammengefasst. Das heißt, der Output eines Neurons ist nicht für jedes Neuron der nächsten *Layer* ein Input, sondern nur für manche (räumlich nahe). Außerdem hat nicht jedes Neuron seine eigenen Gewichte ω_j sondern es gibt einen Faltungskern (*Kernel*), der bestimmt, wie die einzelnen Inputs gewichtet werden. Wir betrachten nur die zweidimensionale Faltung, dafür müssen die einzelnen *Layer* nicht eindimensional wie in 3 sondern zweidimensional angeordnet werden.

Sei $\mathbf{X} \in \mathbf{R}^{m \times n}$ der Input der *Layer*. Im Gegensatz zu einer *Fully Connected Layer* wo

$$g(\mathbf{X})_{ij} = g_{ij}(\mathbf{X}) = \sigma\left(\sum_{k_1=1}^m \sum_{k_2=1}^n \omega_{k_1 k_2}^{ij} \mathbf{X}_{k_1 k_2} + b^{ij}\right)$$

gelten würde, gibt es nun einen Faltungskern $K \in \mathbf{R}^{\hat{m} \times \hat{n}}$ und es gilt

$$g(\mathbf{X})_{ij} = g_{ij}(\mathbf{X}) = \sigma\left(\sum_{k_1=1}^{\hat{m}} \sum_{k_2=1}^{\hat{n}} K_{k_1 k_2}^{ij} \mathbf{X}_{i+k_1-1, j+k_2-1}\right)$$

Oft wird zudem $\sigma(z) := z$ gewählt, bzw. kein σ verwendet. Intuitiv lässt sich die Faltung so interpretieren, dass der Faltungskern über den Input läuft und an jeder Stelle einen Output generiert. Dieser Vorgang ist hier noch einmal dargestellt:

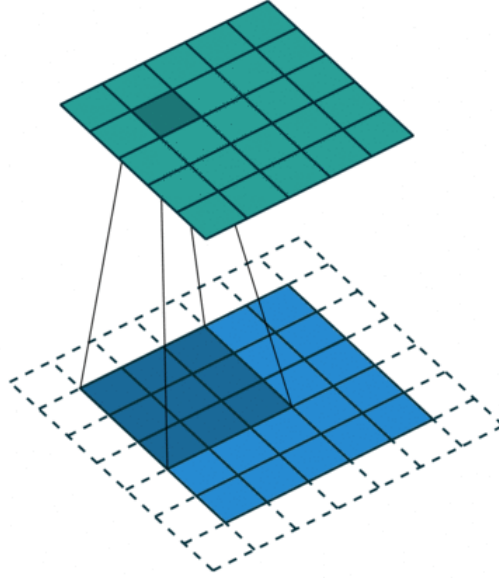


Abbildung 4: grafische Darstellung einer Faltung mit Faltungskern $K \in \mathbf{R}^{3 \times 3}$ [Prö17]

Es fällt zudem auf, dass die Größe von $g(\mathbf{X})$ nun nicht mehr beliebig gewählt werden kann, wie bei der *Fully Connected Layer*, stattdessen ist die Größe durch die Faltung eindeutig bestimmt. Es gilt $g(\mathbf{X}) \in \mathbf{R}^{m' \times n'}$ mit $m' = m - \hat{m} + 1$ und $n' = n - \hat{n} + 1$. Die Gewichte K des Faltungskerns werden trainiert und sind Teil des Parameterraums Θ während \hat{m} und \hat{n} Hyperparameter sind. Zudem gibt es noch weitere Hyperparameter wie *stride* oder *padding*. Für mehr Informationen siehe [Prö17].

2.3.4 Pooling Layer

In *Convolutional Neural Networks* gibt es noch eine dritte Art zwei *Layer* zu verbinden, die *Pooling Layer*. Sie ähnelt der *Convolutional Layer*, statt mit einem Faltungskern werden räumlich nahen Inputs mit einer anderen einfachen Funktion verknüpft:

$$\begin{aligned} g(\mathbf{X})_{ij} &= g_{ij}(\mathbf{X}) \\ &= \text{pool}(\mathbf{X}_{i,j}, \dots, \mathbf{X}_{i,j+\hat{n}-1}, \mathbf{X}_{i+1,j}, \dots, \mathbf{X}_{i+1,j+\hat{n}-1}, \dots, \mathbf{X}_{i+\hat{m}-1,j}, \dots, \mathbf{X}_{i+\hat{m}-1,j+\hat{n}-1}) \end{aligned}$$

Dies ist meistens das Maximum oder der Durchschnitt. Man spricht von *Maximum Pooling Layer* oder *Average Pooling Layer*.

2.4 Parameter und Training

Wir haben nun gesehen, dass neuronale Netze eine Funktion f_p darstellen, die von Gewichten und Bias, also den Parametern $p \in \Theta$ abhängt. Diese Parameter sind die trainierbaren Parameter des neuronalen Netzes und unterscheiden sich so von den nicht trainierbaren Hyperparametern. Das Ziel besteht nun p möglichst gut bezüglich der Trainingsgenauigkeit (*Accuracy*) zu wählen. Das geschieht mit iterativen Lernverfahren, die ausgehend von initialen Anfangsparametern immer bessere p erzeugen. Diesen iterative Vorgang nennt man "trainieren".

Dazu werden oft auf dem Gradientenabstieg (*Gradient Descent*) basierende Methoden verwendet, die eine Kostenfunktion über p minimieren, die angibt wie gut f_p die Trainingsdaten vorhersagen kann. Für das Lernen bieten aktuelle Bibliotheken bereits vorgefertigte Optimierer, wie z.B. in *PyTorch* den *Stochastic gradient descent* (*SGD*) oder den Optimierer *ADAM*.

Alle Optimierer müssen mit verschiedenen Hyperparametern wie einer *learning rate* eingestellt werden, die beispielsweise angibt wie schnell/fein sich p verbessert.

Ohne zu viel auf diese Hyperparameter eingehen zu wollen ist klar, dass die Auswahl dieser Hyperparameter ein gewisses Verständnis des Optimierers voraussetzt und einen großen Einfluss auf die Trainingsgeschwindigkeit und Güte des Ergebnisses hat.

Zusätzlich gibt es noch weitere nicht trainierbare Hyperparameter, darunter unter anderem die Anzahl und Art von *Layern* und welche *Layer* untereinander verbunden sind. Man spricht hier oft von der Topologie des Netzes. Des Weiteren gibt es noch die Hyperparameter der einzelnen Layern wie die Anzahl der Neuronen (bei einer *Fully Connected Layer*) oder die Größe des Faltungskerns (bei einer *Convolutional Layer*).

All diese Hyperparameter des Netzes ändern Θ und wie f_p für ein festes $p \in \Theta$ aussieht.

2.5 Hyperparametersuche

Insgesamt gibt es also jede Menge Hyperparameter, die gewählt werden müssen und ein Vielzahl von möglichen Topologien. Ohne gründliches Wissen über neuronale Netze ist es für den Einsteiger sehr schwer geeignete Hyperparameter zu finden. Erfahrungen aus der Praxis zeigen, in welchen Situationen welche Werte sinnvoll sind. Mit genügend Erfahrung kann man strukturiert verschiedene Hyperparameter austesten. Diese durch Erfahrung gewonnen Erkenntnisse und weitere Ratschläge finden sind zum Beispiel in dem Paper "Practical Recommendations for Gradient-Based Training of Deep Architectures" von Yoshua Bengio [Ben12].

Einem Anwender, der sich mit neuronalen Netzen nicht gut auskennt, sollte es aber im besten Fall erspart werden, sich so tief in die Materie einlesen zu müssen. Obwohl moderne Bibliotheken wie *PyTorch* mit bereits implementierten Methoden und Klassen viele lästige und komplizierte Aufgaben bereits abnehmen, und so auch Nicht-Experten das Experimentieren mit neuronalen Netzen ermöglicht wird, wäre es wünschenswert auch die Wahl der Hyperparameter zu automatisieren.

Das Ziel ist es, dass der Anwender nur deklarativ die Trainings- und Testdaten angibt und ein Algorithmus automatisch das passende Netz auswählt und es trainiert, so dass mit genügend Rechenzeit auch ohne Expertise ein gutes Resultat entsteht. Auch wenn man schon alles über neuronale Netze weiß ist dies trotzdem erstrebenswert, da so mehr Zeit für wichtigere Aufgaben bleibt, während die immer besser werdenden Computer die Rechenarbeit übernehmen.

Diesen Algorithmus zu entwickeln war meine Aufgabe im Fachpraktikum und das Resultat ist der Algorithmus *convNEAT*.

2.6 bisherige Ansätze

Aufgrund der Bedeutenheit der Hyperparametersuche ist es nicht verwunderlich, dass schon viele Anstrengungen in diese Richtung unternommen werden. Wir betrachten nun ein paar klassische Ansätze der Hyperparametersuche, wie sie auch in [Ben12] beschrieben werden. Allen Ansätzen ist es gleich, dass viele Netze nacheinander trainiert werden müssen. Danach wählt man das beste aus allen betrachtenden Netzen.

2.6.1 Grid Search

Für jeden Parameter der gewählt werden soll kann man ein Intervall angeben, in dem nach dem optimalen Wert des Parameter gesucht werden soll. So kann man den Suchraum definieren, in dem man nach den besten Hyperparametern suchen möchte.

Die Idee des *Grid Search* ist es nun für jedes dieser Wertintervalle eine Liste diskreter Werte auszuwählen. Für jede Kombination aus möglichen Werten, lässt man nun einmal das entstehende Netz trainieren, bis schlussendlich die besten Hyperparameter gefunden wurden.

Durch geschicktes Design kann *Grid Search* noch verbessert werden, z.B. indem man bei der Auswahl der Werte aus dem Wertintervall Wissen einfließen lässt, wie z.B. dass sich die *learning rate* des Optimizers für ähnliche Größenordnungen auch ähnlich verhält und man deshalb besser logarithmisch linear Werte auswählt also z.B. [0.1, 0.01, 0.001, 0.0001]. Auch kompliziertere Erweiterungen wie geschachtelte Suchen mit immer höherer Auflösung oder portionsweise nur ein paar Hyperparameter auf einmal zu testen und dafür mehrere Tests durchzuführen, kann die Güte und Geschwindigkeit des Verfahrens verbessern. Für weitere Details siehe [Ben12].

Trotzdem bleibt *Grid Search* sehr rechenaufwändig, da die Anzahl an Tests exponentiell in der Anzahl der Hyperparameter ist.

2.6.2 Random Search

Im Gegensatz zur *Grid Search*, die systematisch den Suchraum absucht kann durch die zufällige Wahl von Hyperparametern erstaunlicherweise viel schneller und sogar bessere Ergebnisse erzielt werden. [J B12] Für jeden Parameter wird eine Verteilung angegeben, meistens eine Gleichverteilung über das logarithmische Wertintervall (siehe 2.6.1) oder eine multinomiale Verteilung bei diskreten Hyperparametern.

2.7 NEAT

Einen ganz anderen Ansatz verfolgen sogenannte genetische Algorithmen. Sie basieren auf den drei genetischen Operatoren *Mutation*, *Selektion* und *Crossover*. Ähnlich zur Evolution in der Natur wird eine Population verwaltet. Die einzelnen Individuen können mutieren, durch evolutionären Druck aussortiert werden und aus zwei Individuen können Nachkommen generiert werden, die Teile der Gene der Eltern besitzen.

Für die Hyperparameterwahl ist besonders das Paper "Evolving Neural Networks through Augmenting Topologies"[K S02] aus dem Jahr 2002 hervorzuheben, das den Grundstein für alle vergleichbaren Methoden gelegt hat. Die Idee ist simpel. Um eine bestmögliche Netztopologie zu finden, kann man mit einem möglichst kleinen, einfachem Netz anfangen und durch Mutation und Crossover neue immer größere Netze erzeugen, die immer besser werden. Sobald die entstehenden Netze nicht mehr besser werden kann man aufhören.

Ein Nachteil an NEAT ist jedoch, dass der Algorithmus in einer Zeit entwickelt wurde, als die Computer noch nicht so viele Möglichkeiten hatten wie heutzutage und außerdem seitdem viele Fortschritte im Bereich des maschinellen Lernens gemacht wurden, die bei *NEAT* nicht berücksichtigt werden konnten. In *NEAT* werden einzelne Verbindungen und Neuronen erzeugt und mutiert. *NEAT* ist nicht darauf ausgelegt moderne Netze mit mehreren Tausend Neuronen zu erzeugen, sondern nur höchstens ein paar Hundert. Das ist für aktuelle Zwecke nicht mehr ausreichend. Außerdem sieht *NEAT* vor, dass auch die Parameter in Θ durch Evolution trainiert werden. Hierfür stehen mittlerweile viel bessere und performantere Methoden wie *SGD* oder *ADAM* zur Verfügung. 2.4

3 convNEAT

NEAT liefert die Grundidee für den entwickelten Algorithmus.

Wir übertragen das Konzept in die Moderne, in dem wir nicht mit einzelnen Neuron arbeiten, sondern als Grundeinheit ganze *Layer* von Neuronen betrachten und auf und zwischen diesen Mutationsoperationen definieren.

Das Training auf den Trainingsdaten überlassen wir einem moderneren Optimierer wie in 2.4. Zusätzlich betrachten wir nicht nur neurale Netze sondern auch *Convolutional Neural Networks* die unserem Algorithmus den Namen *convNEAT* verleihen. So können wir besonders Klasifikationsprobleme in der Bildverarbeitung wie den MNIST Datensatz 2.1 besser lösen.

Es gibt bereits ähnliche Versuche die Ideen von *NEAT* auf *Convolutional Neural Networks* zu übertragen, wie etwa *EXACT* von T. Desell [Des17] oder *EvoCNN* von Y. Sun et al [YY19]. Beide Ansätze haben Nachteile gegenüber *convNEAT*, zeigen aber, dass ein genetischer Ansatz durchaus zielführend sein kann.

Sie verwenden keine Kapselung der Netze in Species wie bei *NEAT* und erlauben beide nur begrenzte *Convolutional Neural Networks*. *ConvNEAT* bietet eine größere Flexibilität und mehr Möglichkeiten für beliebige *feedforward Convolutional Neural Networks* auch mit *Pooling Layern*, sowie leichte Erweiterbarkeit und Anpassbarkeit. Durch modernes *Clustering* können bessere Ergebnisse erzielt werden und durch die Evolution von allen Hyperparametern inklusive den Hyperparametern des Optimierers werden dem Anwender alle schwierigen Entscheidungen abgenommen.

Die Grundidee von *convNEAT* ist im folgenden kurz als Pseudocode dargestellt.

Algorithmus 1 *convNEAT*

```
population ← Initialisiere die Startpopulation mit einfachen Netzen
while durchschnittliche Accuracy verbessert sich do
    Trainiere alle Netze aus population und bestimme ihre Accuracy
    parents ← Selektiere Paare von Netzen aus population
    population ← Generiere mit Crossover neue Netze aus allen Paaren in parents
    Mutiere alle Netze aus population
end while
```

Bevor wir auf weitere Details von *convNEAT* eingehen, werden wir zuerst ein grundlegendes Probleme ansprechen.

3.1 Representation

Da wir mit einem evolutionären Algorithmus arbeiten, müssen wir eine geeignete genetische Representation finden. Diese Kodierung des Netzes als *Genom*, bestehend aus mehreren Genen, bestimmt wie das Netz aussieht, und muss deshalb alle Hyperparameter enthält die uns interessieren. Neben der Anzahl und der Größe der *Layer*, so wie

deren Hyperparameter, gehören aber auch alle anderen Hyperparameter, z.B. die des Optimierers dazu.

Die Representation bestimmt schlussendlich welche Netze gebildet werden können, also auch den Suchraum in dem gesucht werden muss. Dieser Suchraum kann dank des evolutionären Ansatzes viel größer sein als bei einer *Grid Search* oder *Random Search* 2.6. *EvoCNN* verwendet z.B. eine Liste von *Layern* variabler Länge um Netze darzustellen. Der Suchraum ist beschränkt auf diejenigen *Convolutional Neural Networks*, die erst eine Reihe *Convolutional Layer* und danach eine Reihe von *Fully Connected Layer* aufweisen. *convNEAT* schränkt den Suchraum nicht so weit ein, sondern lässt alle möglichen *feed-forward Convolutional Neural Networks* zu. Gerade für schwierige Probleme und Netzen mit vielen *Layern* reichen einfache Netze wie in *EvoCNN* nicht aus [al15].

convNEAT verwendet eine Kodierung, die an die Kodierung in *NEAT* angelehnt ist. *NEAT* kodiert neuronale Netze als Graph, wobei die Neuronen die Knoten und die Verbindungen mit Gewichten die Kanten sind. Natürlich muss dieser Ansatz angepasst werden, trotzdem kann man sich jedes *Convolutional Neural Net*, also auch jedes gewöhnliche neuronale Netz als einen Graphen vorstellen:

Die Knoten sind die einzelnen *Layer* und die Verbindungen zwischen den einzelnen *Layern* werden über die Kanten beschreiben. Wie wir bereits in 2.3 gesehen haben hängt letzteres davon ab, um was es sich bei der hinteren *Layer* handelt. Diese Information, ob *Fully Connected Layer*, *Convolutional Layer* oder *Pooling Layer*, ist zusammen mit allen zugehörigen Hyperparametern in den Kanten gespeichert.

Wie bereits erwähnt ist die Anzahl der Neuronen in den Knoten nicht immer frei wählbar, sondern nur falls es sich bei der eingehenden Kante um eine *Fully Connected Layer* handelt. Aus diesem Grund wird die Größe nicht in den Knoten kodiert. Stattdessen wird die Größe des Inputs durch das Netz propagiert und in jeder Kante bearbeitet. Für *Fully Connected Layer* wird gespeichert wie groß die Änderung der Größe ist, also zum Beispiel, dass die nächste *Layer* 20 mehr Neuronen enthält als ihr Vorgänger. Es können auch inaktive Kantengene in Genom gespeichert werden, die einfach ignoriert werden.

Wenn wir beliebige gerichtete Graphen zulassen stoßen wir auf zwei Probleme: Es kann zu Kreisen kommen, so dass das Durchpropagieren des Inputs nicht mehr funktioniert. Diesem Problem können wir entgegenwirken, in dem wir beim Hinzufügen einer Kante darauf achten, dass keine Kreise entstehen.

Außerdem ist es möglich, dass ein Knoten zwei eingehende Kanten besitzt, die verbunden werden müssen. In diesem Fall müssen durch Hochskalierung, Runterskalierung, einer Mischung aus beiden oder durch das Hinzufügen von Nullen die Outputs auf die selbe Größe gebracht und konkateniert werden. Wie genau das passiert ist ein weiterer Hyperparameter der im Knoten kodiert wird.

Für maximale Flexibilität besonders auf die Anwendung der Bildanalyse bezogen, sind alle *Layer* im späteren Netz, dreidimensional. Neben Höhe und Breite gibt es noch verschiedene Channels, in denen z.B. im Input Gelb-, Rot- und Blauanteile kodiert werden könnten. Bei MNIST bleibt der Input dann z.B. $\mathbf{X} \in \mathbb{R}^{1 \times 28 \times 28}$. Das sorgt dafür, dass im Vergleich zu z.B. *EXACT* viel weniger Knoten gebraucht werden, weil viel mehr Faltungen kompakt repräsentiert werden können.

Neben der Topologie wird auch noch kodiert welcher Optimierer gerade mit welchen

Hyperparametern verwendet wird.

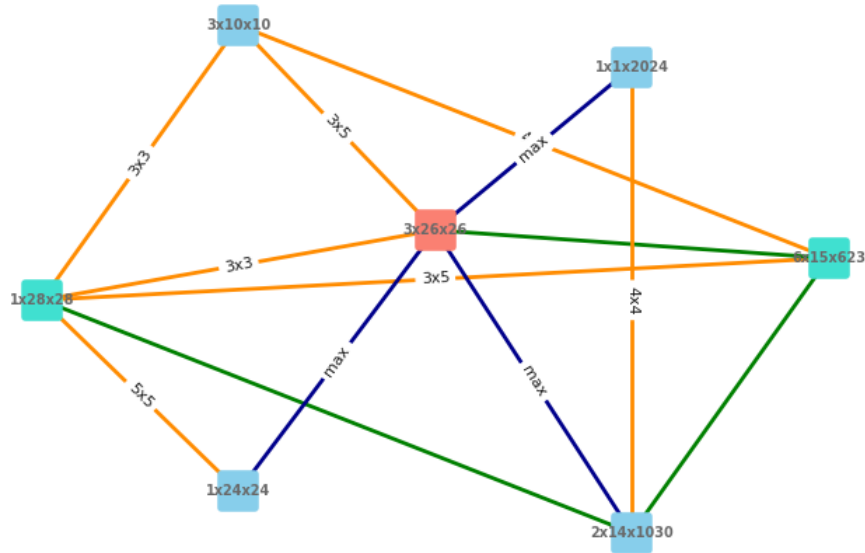


Abbildung 5: Ein Beispiel für ein komplexeres Netz das kodiert werden kann. Die Farben der Kanten geben ihren Typ an, z.B. steht orange für eine *Convolution Layer*

3.2 Mutation

Die Möglichkeit Genome zu mutieren bildet die Basis jedes genetischen Algorithmuses, so können neue Netze erzeugt werden, die den bisherigen Netzen ähneln. Unvorteilhafte Mutationen können in der *Selektion* aussortiert werden, gute Mutationen bleiben erhalten. Welche Mutationen entstehen ist zufällig, manche haben jedoch höhere Wahrscheinlichkeit als andere. Die Wahrscheinlichkeiten wurden durch viele Testläufe so angepasst, dass Sie sinnvoll sind. Diese Hyper-Hyperparameter von *convNEAT* müssen nur einmal eingestellt werden und sind problemunabhängig. Trotzdem wäre es denkbar auch eine adaptive Veränderung der Wahrscheinlichkeiten einzubauen, um diejenigen Mutationen häufiger auftreten zu lassen, die häufiger zu guten Netzen führen. *convNEAT* bietet folgende Mutationen:

Aktivieren und Deaktivieren von Genen

Es besteht die Möglichkeit Gene zu deaktivieren und wieder zu reaktivieren. Deaktivierte Gene spielen keine Rolle mehr für das entstehende Netz. Deaktivierte

Gene können auch durch den *Crossover* 3.4 entstehen. Beim Deaktivieren muss sichergestellt sein, dass es noch einen Pfad vom Input zum Output gibt.

Kante aufteilen

Es besteht die Möglichkeit eine bestehende Kante aufzuteilen. Dabei entsteht ein neuer Knoten zwischen zwei existierenden Knoten. Die ursprüngliche Verbindungskante wird deaktiviert und zwei neue Kanten eingefügt. Die erste ist eine Kopie der alten Kante. Die zweite neue Kante ist eines zufälligen Typs. Manche Kanten sind hier wahrscheinlicher, z.B. ist hinter einer *Convolutional Layer* eine neue *Pooling Layer* am wahrscheinlichsten.

Kante einfügen

Zwischen zwei Knoten kann eine Kante eingefügt werden. Wie beim Aufteilen einer Kante ist die Art der Kante zufällig. Damit keine Kreise entstehen können, wird für jeden Knoten seine Tiefe im Netz gespeichert. Eine neue Kante zeigt dann immer auf die tiefere Kante, beliebige Pfade durchlaufen immer Knoten in echt absteigender Tiefe, Kreise sind unmöglich.

Optimierer

Welcher Optimierer verwendet wird und alle seine Hyperparameter können ebenfalls mutiert werden. Dies kann unter anderem die *learning rate* oder der *weight decay* sein.

Kante mutieren

Alle Kantentypen können ihre Hyperparameter mutieren, jede Mutation hat seine typspezifische Wahrscheinlichkeit. Unter anderem sind folgende Mutationen möglich:

Fully Connected Layer

Die Aktivierungsfunktion und die Größenänderung können mutieren.

Convolutional Layer oder Pooling Layer

Die Weite, Höhe und Tiefe des Kernels sowie weitere Parameter wie unter anderem *padding* oder *stride* können mutieren.

3.3 Selektion

Wichtig für jeden genetischen Algorithmus ist der Operator der *Selektion*. Die besten Netze werden beibehalten und die schlechtesten, z.B. diejenigen die durch eine unvorteilhafte Mutation entstanden sind, sollten nicht weiter überleben. Ein gutes Maß für die Güte eines Netzes ist die *Accuracy* auf den Testdaten. Da beim öfteren Vergleich der *Accuracy* auf den Testdaten, aber die Gefahr des *Overfittings* besteht, also die Chance, dass das Netz das Rauschen der Testdaten lernt und auf neuen Daten keine guten Vorhersagen macht, wird ein Teil der Trainingsdaten nicht zum Trainieren verwendet sondern als Validierungsdaten zurückgehalten. Bei der Selektion kann dann die *Accuracy* auf den Validierungsdaten bestimmt und verglichen werden. Die Testdaten werden nur ganz am

Ende verwendet um die Güte des finalen Ergebnisses aus *convNEAT* zu überprüfen.

Damit Netze die schon länger trainiert haben keinen Vorteil gegenüber frisch mutierten Netzen haben gibt es für jede trainierte Epoche eine Strafe. Die Strafe ist linear im logarithmischen Klassifikationsfehler.

Definition 3.1 (Log classification error).

Der logarithmische Klassifikationsfehler eines Netzes mit Accuracy a ist definiert als

$$\text{logerr}(a) = \log_{10}(1 - a)$$

Für ein Genome mit Accuracy a und n trainierten Epochen kann man einen modifizierten score berechnen mittels

$$\text{score}(a, p) = 1 - 10^{\text{logerr}(a) + n * \text{decay}}$$

Denn so gilt:

$$\text{logerr}(\text{score}(a, p)) = \text{logerr}(a) + n * \text{decay}$$

Basierend auf diesem angepasstem *score* können jetzt Paare aus Genomen gebildet werden die im nächsten Schritt, dem *Crossover* kombiniert werden können. Es ist auch möglich, dass ein Paar aus zweimal dem gleichen Genom besteht. Dann wird der *Crossover* übersprungen und es kommt nur zur *Mutation* des Genoms. Es gibt viele Möglichkeiten diese Selektion durchzuführen. In *convNEAT* sind mehrere Methoden implementiert.

Von einfachen Methoden, wie die simple Auswahl aller besten Genome (*cutoff*), oder der zufälligen Wahl, bei der jedes Genom eine Wahrscheinlichkeit proportional zu seinem *score* hat (*fps*), bis hin zu komplizierteren Methoden wie der *Tournament selection* (*ts*) bei der zufällig eine Gruppe von k Genomen ausgewählt wird, von denen das Beste ausgewählt wird, oder der *stochastic universal sampling* (*sus*) [Bak87].

Welche Methode die beste ist, muss mit vielen Testläufen auf verschiedenen Problemen ausgetestet werden, da meine Rechenzeit aber limitiert war, konnte ich nur einige wenige Tests auf dem MNIST Datensatz durchführen. Hier schien es, als ob *stochastic universal sampling* eine stabile und effektive Methode der Selektion ist.

Wichtig ist das die Selektion auch ein paar schlechtere Netze überleben lässt. Denn vielleicht sind manche Netze nicht lange genug trainiert worden und zeigen ihr volles Potenzial nach etwas mehr Zeit. *stochastic universal sampling* besitzt diese Eigenschaft. Um eine Intuition für die verschiedenen Selektionsfunktion zu bekommen sind in 1 die durchschnittliche Auswahlwahrscheinlichkeit einiger Verfahren für ein Beispielpopulation $x_i, i \in \{1, \dots, 15\}$ mit $\text{score}(x_i) = i^2$ angegeben. Man kann gut sehen, wie *stochastic universal sampling* bessere Werte bevorzugt, aber auch schlechtere Werte noch berücksichtigt.

3.4 Crossover

Eine weitere Operation in genetischen Algorithmen ist der *Crossover*. Er erzeugt aus zwei Genomen ein neues Genom. Dieses neue Genome erhält kann Gene beider Eltern enthalten. *convNEAT* verwendet, wie bei *EXACT*, die Parameter *more fit parent crossover rate*

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
<i>cutoff</i>	0	0	0	0	0	0	0	0	0	17	17	17	17	17	17
<i>fps</i>	0	0	1	1	2	3	4	5	7	8	10	12	13	15	17
<i>ts</i>	0	0	0	1	2	4	5	7	8	10	11	12	13	13	13
<i>sus</i>	0	0	1	1	2	3	4	5	6	8	10	12	13	16	18

Tabelle 1: Einige Selektionsfunktionen zu $score(x_i) = i^2$, alle Werte in %

und *less fit parent crossover rate*. Diese beiden beschreiben wie viel Prozent des jeweiligen Genomes aktiviert bleiben soll. Gene (hier vor allem die Kanten), die beispielsweise nur im besseren Genom (mit höherem *score* 3.3) vorkommen, werden mit Wahrscheinlichkeit gleich der *more fit parent crossover rate* aktiviert übernommen. Alle anderen Gene werden deaktiviert insofern das möglich ist.

Da die Eltern zwei beliebige Graphen sein können, ist es schwierig zu sagen, welche Gene gleich sind und wie die ungleichen Gene kombiniert werden sollen. Abhilfe schafft das Konzept der *historical markers* aus *NEAT*. Jedes Gen bekommt eine eindeutige Zahl, die *innovation number* zugeordnet. Wenn durch Mutation neue Kanten oder Knoten entstehen, werden ihnen neue Zahlen zugeordnet. Möchte man zwei Genome vergleichen, kann man sich die *Marker* anschauen und sieht sofort wie viele Genome gleich sind und wie unterschiedlich (im biologischen Sinne) die Genome sind.

Ein simples Beispiel für einen Crossover findet sich in in 6

3.5 Clustering

Mit den obigen genetischen Operatoren ist bereits das Grundgerüst für *convNEAT* entworfen. Theoretisch werden die Netze immer komplizierter und dadurch auch besser. In der Praxis tritt man jedoch auf ein Problem, das schon bei der Entwicklung von *NEAT* bekannt war. Wird ein neues Netz generiert, z.B. durch *Crossover* so kann es in der Praxis oft nicht lange überleben, es wird durch die Selektion schnell wieder aussortiert. Das neue Netz braucht nämlich einige Zeit um durch Training, oder weitere Mutationen langsam besser zu werden, bis es kompetitiv genug ist, um mit den aktuell besten Netzen verglichen werden zu können, die schon über viele Generationen optimiert wurden.

Eine Lösung für das Problem ist wie *EvoCNN* eine Selektion auszuwählen, die nur sehr geringen *Selektionsdruck* ausübt, also auch schwache Netze oft überleben lässt. Da aber mit Senkung des *Selektionsdruck* auch die Geschwindigkeit sinkt, mit der Fortschritte gemacht werden können, verwendet *convNEAT* stattdessen das Konzept der *Artentrennung* (im Englischen *Speciation*), im weiteren auch mit *Clustering* bezeichnet.

Ähnliche Netze mit ähnlicher Topologie werden in einzelne *Arten* (*Cluster*) zusammengefasst, so dass Netze im gleichen Cluster möglichst ähnlich und in verschiedenen Cluster möglichst verschieden sind. Die Selektion erfolgt jetzt nur innerhalb den einzelnen Clustern, so dass es dauerhaft Cluster von Netzen geben kann, die sich von dem besten Cluster unterscheiden. Während das beste Cluster weiter optimiert wird gibt *convNEAT* auch anderen Ansätzen die Chance weiter erforscht und optimiert zu werden.

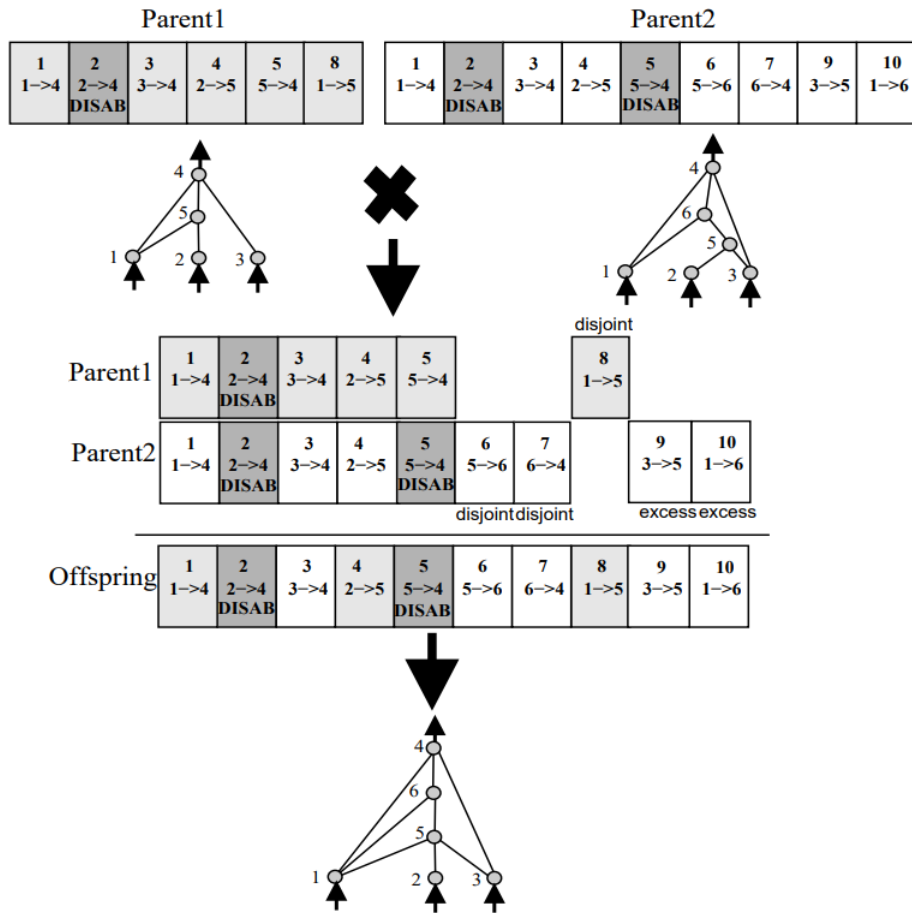


Abbildung 6: Ein Beispiel für Crossover zweier Graphen aus [K S02]

Das Zusammenspiel der einzelnen Cluster untereinander muss genau geregelt sein. *conv-NEAT* sorgt dafür, dass die besten Cluster größer werden können, um mehr Rechenzeit in die Optimierung der aktuell besten Lösung zu investieren, aber gleichzeitig andere Cluster nicht zu klein werden, auch wenn sie schlechter sind. Cluster die sich nicht weiter verbessern, weil die Netztopologie einfach nicht passend für das Problem ist, werden nach einer gewissen Zeit gelöscht.

Damit *Clustering* funktionieren kann brauchen wir zwei Dinge: Eine Abstandsfunktion und einen Clusteralgorithmus.

3.5.1 Ähnlichkeit

Das Konzept von "ähnlichen" Genomen muss nun mathematisch definiert werden. Für jede Art von Gen wird eine Ähnlichkeitsfunktion $sim(x, y) \in [0, 1]$ definiert, die angibt wie unterschiedlich die Hyperparameter sind. $sim(x, y) = 0$ bedeutet, dass die Ausprägungen

x und y des Gens gleich sind. Je näher $\text{sim}(x, y)$ der 1 ist, desto verschiedener sind die beiden Ausprägungen.

Definition 3.2 (Distanzmetrik). *convNEAT verwendet folgenden Abstand:*

$$\text{dist} = c_0 * S + c_1 * D + c_2 * E + c_3 * T + c_4 * K + c_5 * X$$

wobei S der Unterschied in den gemeinsamen Kanten ist, D und E zusammen die Anzahl der Gene die nur in einem der beiden Genome vorkommt, T ist der Unterschied der verwendeten Optimierer, K der Unterschied in den Knoten und X der Unterschied in der Anzahl der trainierten Epochen.

Diese Abstandsfunktion sorgt dafür, dass alle Unterschiede eine festlegbare Rolle spielen. Die Parameter c wurden durch Testläufe optimiert. Am wichtigsten sind S , D und E .

3.5.2 Clusteralgorithmus

Das Problem, eine Menge von n Daten $\mathbf{X} \in \mathcal{A}^n$ in k Cluster einzuteilen lässt sich als Optimierungsproblem darstellen. Zu minimieren ist der Abstand der Mitglieder der Clusters S_i zum jeweiligen Clustermittelpunkt μ_i :

Definition 3.3.

$$J := \sum_{i=1}^k \sum_{x_j \in S_i} \text{dist}(x_j, \mu_i)^2$$

Für den Fall $\mathcal{A} = \mathbb{R}^m$ gibt es hervorragende Algorithmen wie *k-Means* [Llo57], um das Optimierungsproblem approximativ sehr schnell zu lösen. *k-Means* ist ein iteratives Verfahren, dass ausgehen von Start-Clustermittelpunkten μ_i^0 immer bessere μ_i findet, in dem es den Mittelpunkt der aktuellen zu Cluster S_i gehörenden Datenpunkte zum neuen μ_i macht bis dieser Prozess konvergiert.

Leider lässt sich *k-Means* nicht auf unser Problem anwenden, da der Suchraum kein Vektorraum mit einer Basis ist, der Begriff eines Mittelpunktes ist nicht definiert. Wir dürfen lediglich mit der Metrik *dist* arbeiten.

convNEAT verwendet deshalb eine stark modifizierte Variante des *k-Medoid* Algorithmus. Im Gegensatz zum Mittelpunkt wählen wir den "*Median*", in unserem Fall das Genom im Cluster, dass den geringsten Abstand zu allen anderen Clustermittgliedern hat, also

$$\text{med}(z) := \sum_{x_j \in S_i} \text{dist}(x_j, z)$$

minimiert.

k-Medoid kann deshalb so umgebaut werden, dass es nur von *dist* abhängt. Einige Dinge sind noch zu beachten, z.B. darf kein Cluster zu klein werden, da sonst keine sinnvolle Selektion stattfinden kann. Damit man messen kann wie sich die einzelnen Netze eines Clusters über die Zeit entwickeln muss zudem eine Art Konsistenz erhalten bleiben. Ist das Beste Cluster S_k für ein festes k dann sollte nach dem erneuten Clustern wieder viele

Netze $x_j \in S_k$ im neuen Cluster S'_k sein. Diese Konsistenz ist bei *k-Medoid* nicht gegeben. Die Güte des Ergebnis im Bezug auf J hängt stark von den Start-Clustermittelpunkten μ_i^0 ab. *k-Medoid* wird deshalb üblicherweise mit verschiedenen kompliziert gewählten μ_i^0 mehrmals ausgeführt. *convNEAT* verzichtet auf eine komplizierte Initialisierung und verwendet stattdessen den Median der schon aus der letzten Generationen bekannten Cluster als Startwert. Die Einbuße in der Funktion J können durch die entstehende Konsistenz gerechtfertigt werden.

Der entwickelte *consistent bounded k-Medoid* kann aber nur für festes k eine Clusterung finden. Für *convNEAT* ist es jedoch wichtig die Anzahl der Cluster variieren zu lassen, falls durch Mutation und Crossover viele neuartige Netze entstanden sind oder nach der Selektion zu wenige Netze eines Cluster übrig bleiben.

convNEAT ändert adaptiv die Anzahl der Cluster, falls dies nötig ist und achtet trotzdem darauf, dass die Konsistenz der Cluster erhalten bleibt. Ein Beispiel für die Ergebnisse des Clusterings finden sich in Abbildung 7 und 8.

3.6 Eliten und Training

Damit die Netze in der Population überhaupt gute Vorhersagen auf den Validierungsdaten machen können müssen sie auf den Trainingsdaten trainieren, dies geschieht in einer Trainingsphase. In dieser Phase kann jedes Netz eine feste Anzahl von Epochen auf den Daten trainieren. Die Laufzeit von *convNEAT* ist auf diesen Schritt zurückzuführen, denn wie bei der Arbeit mit neuronalen Netzen üblich, kann das Training sehr lange, manchmal Stunden oder Tage lang dauern. Wie schnell *convNEAT* zu einem Ergebnis kommt hängt also nur davon ab wie viele Epochen alle Netze gemeinsam trainiert werden.

Netze die schnellen Fortschritt machen werden belohnt in dem Sie länger trainieren können. Dies sorgt dafür, dass insgesamt schnellerer Fortschritt gemacht wird. Besonders bei komplizierten Problemen ist aber nicht klar, ob die festgelegte Anzahl an Epochen ausreicht um ein gutes Netz zu trainieren. Je nach Hyperparameterwahl dauert es eventuell viel länger. Die Netze hätten so überhaupt keine Zeit gut genug zu werden, um das Problem lösen zu können.

Abhilfe schaffen die sogenannten *Eliten*. Nach jeder Generation werden die besten Netze jedes Clusters automatisch in die nächsten Generation übernommen. Nur der verbleibende Platz in der Population wird durch den Crossover aufgefüllt.

Das hat den Vorteil, dass die besten Netze die Möglichkeit haben über mehrere Generationen so lange zu trainieren, wie sie brauchen um ihr Potenzial auszuschöpfen. Andererseits sorgt der angepasste score in 3.3 dafür, dass Netze die schon länger trainiert haben, nicht bevorzugt werden und auch neue Netze noch die Chance haben sich zu etablieren. Zudem wird die Laufzeit gesenkt. Gute Netze die durch Training nicht mehr zu verbessern sind, bleiben in der Population ohne jede Generation Trainingszeit zu verschwenden.

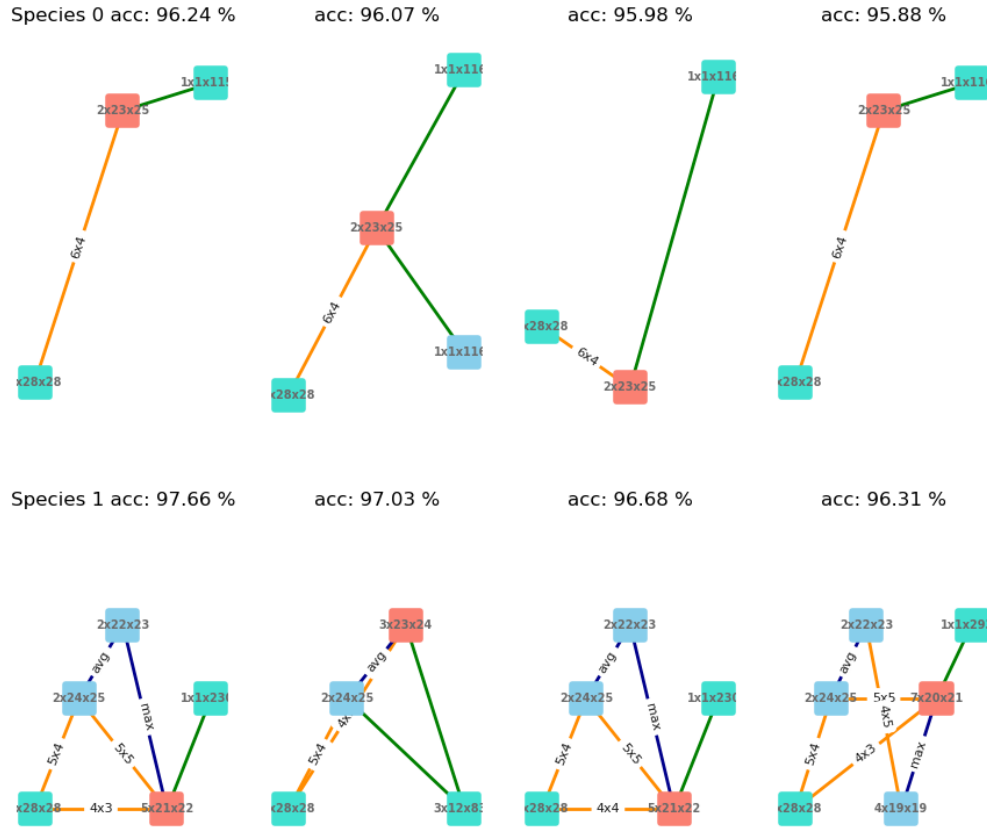


Abbildung 7: Die besten Netze aus zwei Clustern einer Population. Zu sehen ist ein Cluster mit simpleren Netzen mit einer *Convolutional Layer* gefolgt von einer *Fully Connected Layer* sowie Cluster mit komplexeren Netzen.

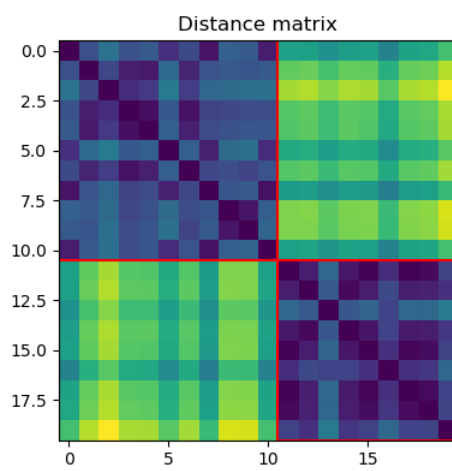


Abbildung 8: Die zu 7 gehörende Distanzmatrix D mit $D_{ij} = \text{dist}(i, j)$. Blaue Farben zeigen eine hohe Ähnlichkeit an. Die rot umrandeten Teilmatrizen sind die clusterinternen Distanzen

4 Ergebnisse

Der entwickelte Algorithmus wurde in *Python* implementiert und verwendet die Bibliothek *PyTorch* um die Netze zu trainieren. Das entwickelte Modul lässt sich sehr einfach ausführen, der Einzige Input ist der Datensatz. Alle Hyperparameter, wie die Anzahl der Genome in der Population kann eingestellt werden, aber auch die Standardwerte funktionieren bereits für die meisten Probleme. Zusätzlich zur detaillierten Ausgabe im Terminal besitzt *convNEAT* eine optionale graphische Oberfläche (siehe 9), die den aktuellen Trainingsfortschritt anzeigt und einen Überblick der Population gibt. Nach jeder Generation werden die besten Netze abgespeichert. Das Training kann dann unterbrochen und später fortgesetzt werden. Die Implementation kann online auf Github abgerufen werden. Die Implementation bietet leichte Erweiterungsmöglichkeiten für andere *Layer*-Typen, Optimierer oder *score* Funktionen.

Agrund der Tatsache, dass *convNEAT* eine vergleichsweise lange Laufzeit besitzt und sehr viele Testläufe gemacht werden müssen um die Hyperparameter einzustellen, blieb nicht mehr genug Zeit *convNEAT* auf verschiedenen Datensätzen zu testen und so die Vielseitigkeit von *convNEAT* zu demonstrieren. Es wurden jeddoch einige Tests auf dem (leider etwas limitierten) MNIST Datensatz durchgeführt:

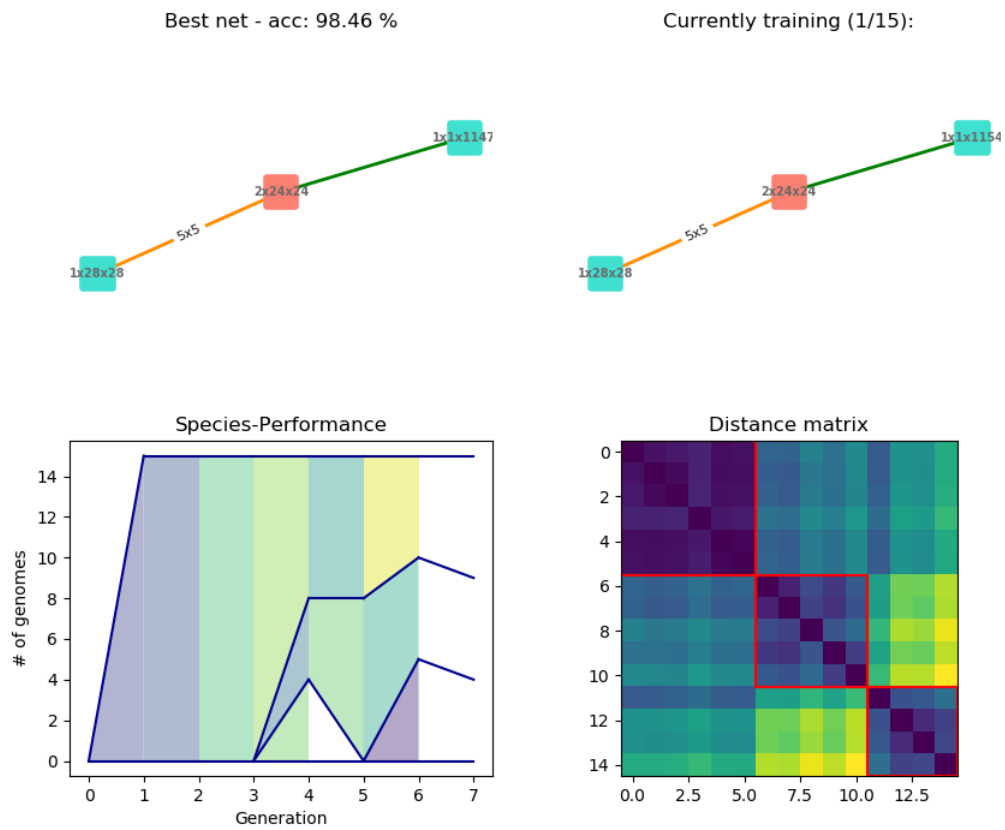


Abbildung 9: Die Oberfläche von *convNEAT*. Neben dem besten und dem aktuell trainierten Netz findet sich hier eine Übersicht über die *Performance* und die *Clustering*. Da MNIST keine sehr komplexer Datensatz ist, konnte auch eine einfache Topologie "gewinnen"

4.1 MNIST

Auf den Standardeinstellung erzeugt *convNEAT* relativ schnell Netze mit einer Accuracy von über 98% (je nach Seed sogar fast bis 99% und erfüllt damit die Ansprüche, die wir an den Algorithmus gestellt haben, automatisch ein passables Ergebnis zu erreichen. Die entstehenden Netze unterscheiden sich von Netzen die ein Mensch erstellen würde, sie sind viel organischer. In den Abbildungen 10 und 11 finden sich das besten Netz eines zufälligen Durchlaufs mit 15 Generation, so wie eine Übersicht über den Verlauf des Trainings.

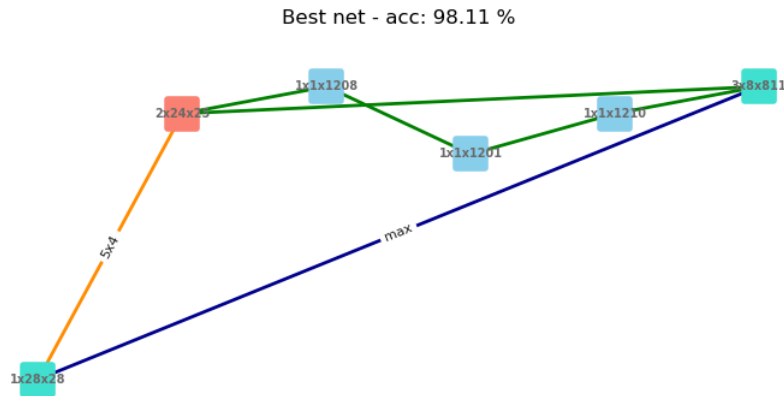


Abbildung 10: Das beste Netz mit einer Accuracy von 98,11%

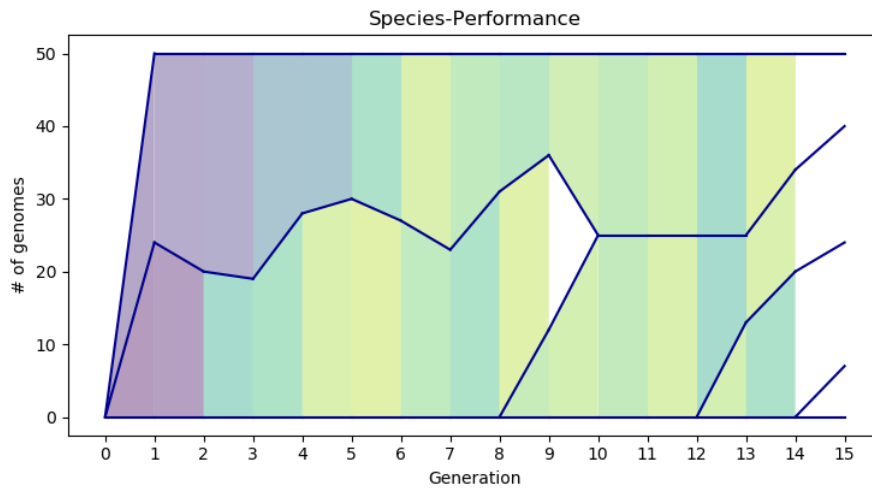


Abbildung 11: Eine grafische Darstellung des Verlaufes. Man erkennt die einzelnen Cluster mit adaptiv angepasster Größe. In der 10. Generation wird ein Cluster gelöscht, dass sich 3 Generationen lang nicht weiter verbessert hat.

Wie in 2.1 nachzulesen ist, ist MNSIT ein bereits "*gelöstes*" Problem, bei dem schon Klassifizierer mit einer *Accuracy* über 99.5% existieren. Das letzte Prozent wird meist erst mit geeignetem Preprocessing der Daten erreicht.

Vergleicht man die Resultate mit denen von *EXACT*, so sieht man, dass die entstehenden Netze wie zu erwarten kompakter sind. *EXACT* lief über einen längeren Zeitraum verteilt auf mehreren Computern nebenläufig während *convNET* schon nach einigen Stunden auf einem normalen Laptop Ergebnisse erzielen kann. Welcher Algorithmus schneller ist, ist trotzdem schlecht abzuschätzen. Die *Accuracy* ist vergleichbar, *EXACT* erreicht in 4 Läufen eine *Accuracy* zwischen 97.58 und 98.32. Auch *EvoCNN* erreicht eine ähnliche *Accuracy* von durchschnittlich 98.72.

5 Ausblick

Wie bereits in 4 diskutiert wäre es interessant *convNEAT* auf weiteren Benchmarkproblemen, wie dem *ImageNet* Datensatz zu testen, die noch nicht "*gelöst*" sind. Hier würde sich zeigen wie flexibel und gut *convNEAT* wirklich ist.

Die Hyperparameter von *convNEAT* könnten zu dem noch intensiver getestet werden und die Performance so noch einmal deutlich zu verbessern. Eine weitere Überlegung ist es die besten Netze der Population "*abstimmen*" zu lassen, so kann ebenfalls eine noch bessere *Accuracy* erreicht werden.

Als letztes wäre es noch möglich die leichte Erweiterbarkeit auszunutzen um auch auf anderen Probleme jenseits von Klassifikationsprobleme zu lösen oder andere neuronale Netze wie *Recurrent Neural Networks* zuzulassen.

Referenzen

- [Llo57] S. Lloyd. „Least square quantization in PCM“. In: *Bell Telephone Laboratories Paper* (1957).
- [Bak87] *Reducing Bias and Inefficiency in the Selection Algorithm*. International Conference on Genetic Algorithms und their Application. 1987.
- [Y L98] C. Burges Y. LeCun C. Cortes. „THE MNIST DATABASE of handwritten digits“. In: (1998). URL: <http://yann.lecun.com/exdb/mnist/>.
- [K S02] R. Miikkulainen K. Stanley. „Evolving Neural Networks through Augmenting Topologies“. In: (2002). URL: <http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf>.
- [Ben12] Y. Bengio. „Practical Recommendations for Gradient-Based Training of Deep Architectures“. In: (16. Sep. 2012). URL: <https://arxiv.org/pdf/1206.5533v2.pdf>.
- [J B12] Y. Bengio J. Bergstra. „Random search for hyper-parameter optimization“. In: *Journal of Machine Learning Research* (2012). URL: <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>.
- [al15] K. He et al. „Deep Residual Learning for Image Recognition“. In: (2015). URL: <https://arxiv.org/pdf/1512.03385.pdf>.
- [Des17] T. Desell. „Large Scale Evolution of Convolutional Neural Networks Using Volunteer Computing“. In: (2017). URL: <https://arxiv.org/pdf/1703.05422.pdf>.
- [Prö17] P. Pröve. „An Introduction to different Types of Convolutions in Deep Learning“. In: *Towards Data Science* (2017). URL: <https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d>.
- [Nie19] M. Nielsen. „Neural Networks and Deep Learning“. In: (2019). URL: <http://neuralnetworksanddeeplearning.com/chap1.html>.
- [YY19] M. Zhang Y. Sun B. Xue und G. Yen. „Evolving Deep Convolutional Neural Networks for Image Classification“. In: (2019). URL: <https://arxiv.org/pdf/1710.10741.pdf>.