Mean-Variance Optimization in View of Parameter Estimation

Erwartungswert-Varianz-Optimierung mit
Blick auf Schätzung der Parameter

a Bachelor's Thesis

by MARK-OLIVER WOLF

supervised by
Prof. Dr. Jörn Saß

May 16, 2020

Department of Mathematics

Contents

1 Einleitung

[On a stock market, a possible investor wants to maximize the amount of money he gains from buying assets and selling them at a later time. As we are not able to predict the behavior of the stock market, stochastic models are important for trying to solve this problem. We decide to model it by assuming a distribution that matches the returns of our assets, further using past data to estimate the specific parameters of it. If we are then able to fix the distribution, we can make the objectively best choice of assets we chose to buy and how many of them. Here objectively best choice means that given a specific utility function best describing our needs, our choice of assets will on average deliver the highest amount of utility. But as these parameters have to be estimated and the estimates depend on past data, the estimation has an included risk which we have to take into consideration. We call the assets we hold a portfolio, and the method deciding what portfolio to buy a portfolio rule.
For a long time portfolios were chosen by 'proven in action' methods, which only use basic mathematical methods for their computation. One of them is the standard plug-in portfolio rule. As can easily be simulated, this rule scores poorly compared to other rules. In this thesis, we will study different plug-in rules in a mathematically accurate way and derive the 'best' plug-in rule for different situations. Further we will prove that the often used standard plug-in rule is strictly dominated by other rules and consecutively is worse in every situation. To get these results, we expand the argumentation of Raymond Kan and Guofu Zhou in their paper 'Optimal Portfolio Choice with Parameter Uncertainty' [**kan_and_zhou**]. We finish the thesis by simulating different portfolio rules on a dataset created by parameters of real stock market data, which will confirm the earlier results.] This indroduction is already done

2 Problemstellung

Bevor wir uns genauer mit dem der Problemstellung der Optimierung von Topologie und Hyperparametern von Convolutional Neural Networks beschäftigen folgte zuerst eine kleine Einführung in neuronale Netze. Um besser verstehen zu können wofür neuronale Netze verwendet werden betrachten wir zunächst ein weitverbreitetes Standardproblem im Bereich des machinellen Lernens, dass wir später auch mit unserem Algorithmus lösen können:

2.1 Handschrifterkennung

Eine Aufgabe für die sich neuronale Netzte hervorragend eignen ist 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 lernfähige Methoden einen Algorithmus zur Erkennung schreiben wollen, so wird dieser sehr kompliziert. Neuronale Netzte 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 60.000 dieser Bildern (Trainingsdaten) Muster zu erkennen. Bewertet wird danach, wie gut der Algorithmus die 10.000 weiteren vom Algorithmus noch nicht gesehenen Zahlen (Testdaten) klassifiziert.

2.2 Klassifikationsprobleme

Die allgemeine Problemstellung die ein Neural Network löst ist die folgende:
Gegeben sei eine Zielfunktion $f : \mathcal{H} \to \mathcal{G}$.
Da diese Funktion aber nicht bekannt ist soll sie nun möglichst gut approximiert werden. Hierzu muss aus einer Menge von Funktionen $f_p : \mathcal{H} \to \mathcal{G}$ mit $p \in \Theta$ aus dem Parameterraum, diejenige ausgewählt werden, die am f möglichst gut 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 vorallem Klasifaktionsprobleme interessant, also (z.B. der Fall das $\mathcal{G}$ endlich ist und alle verschiedenen Klassen enthält. 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 *classiication 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 $(t_i, f(t_i)) \in \mathcal{G} \times \mathcal{H}$ $i \in \{1, \cdots, m\}$ mit korrekt klassifizierten Punkten t_i.
Die Accuracy acc ist nun definiert als:

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

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

Um nun geeignete Parameter aus Θ zu finden, ohne das wir die Testdaten verwenden dürfen, gibt es einen Trainingsdatensatz mit n Daten
$(x_i, f(x_i)) \in \mathbf{G} \times \mathbf{H}\ i \in \{1, \cdots, n\}$, die ebenfalls bereits richitg klassifiziert sind. Aktuelle Methoden des machinellen Lernens optimieren nun die *Accuracy* auf den Trainingsdaten und hoffen, dass dadurch auch die *Accuracy* auf den Testdaten gut wird, also das neurale Netz gut generalisiert hat.

Wenn wir erneut das Klasifaktionsproblem MNIST betrachten ergibt sich insgesamt z.B: $\mathcal{H} = \mathbb{R}^{28\times 28}$, $\mathcal{G} = \{0, \cdots, 9\}$ sowie $n = 60.000$ Trainingsdaten und $m = 10.000$ Testdaten

2.3 Neuronale Netze

2.3.1 Neuronen

Um verstehen zu können was neurale Netzte sind schauen wir uns zuerst den Grundbaustein an, aus denen Sie bestehen, die (künstlichen) Neuronen. Ein Neuron ist eine kleine Einheit die beliebig viele Inputs $x_1, \cdots, x_n$ erhält und daraus einen Output y errechnet. Grafisch lässt sich das folgendermasßen vorstellen:

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 beinflussen, was genau das Neuron berechnen kann 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 $\mathbf{x}$. Wie σ aussieht beeinflusst wie gut sich ein Netzt trainieren lässt und kann ausgewählt werden. 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*

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

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

$$ReLU(z) = \max(0, z)$$

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

2.3.2 Fully Connected Layer

Ein neuronales Netzt besteht nun 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 Ouputs der vorherigen Layer enthält. Jedes g_i enthält immernoch seine eigenen Parameter $\omega_1^i, \cdots, \omega_n^i$ und b^i.
In normalen neuronalen Netzten gibt es nur sogenannte *Fully Connected 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 mit Funktionen $g_1, \cdots, g_m$ hat also die oben angesprochene Funktion
$g(\mathbf{x}) = (g_1(\mathbf{x}), \cdots, g_m(\mathbf{x}))^T$
Der Input der ersten Layer von Neutronen ist $\mathbf{x} \in \mathcal{H}$. Die einzelnen Komponenten von $\mathbf{x}$ bilden ebenfalls eine *Layer*, die sogeannte *Input Layer*. Die letze *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. $\|\cdot\|_{sup}$ 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:

2.3.3 Convolutional Layer

Da sich das Hauptaugenmerk unsere Algorithmus auf Klasifaktionsproblemen in der Bilderkennung liegt, betrachten wir zudem die dort üblichen *Convolutional Neural Networks*. Dises erweitern die Idee der neuronale Netze um eine weitere Art zwei *Layer* zu verbinden - die Faltung, im Englischen *Convolution*. Anders als bei den vollständig verbunden *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 Abildung 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

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

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

$$g(\mathbf{X})_{ij} = g_{ij}(\mathbf{X}) = \sigma(\sum_{k_1=0}^{\hat{m}-1}\sum_{k_2=0}^{\hat{n}-1} K^{ij}_{k_1k_2}\mathbf{X}_{i+k_1,j+k_2})$$

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 Ouptut generiert. Dieser Vorgang ist hier noch einmal dargestellt:
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.

2.3.4 Pooling Layer

In *Convolutional Neural Networks* gibt es noch eine zweite neue Art *Layer* zu verbinden, die *Pooling Layer*. Sie funktionieren sehr ähnlich zu der *Convolutional Layer* gibt es aber keinen Faltungskern mit Gewichten, sondern die räumlich nahen Inputs werden mit einer anderen einfachen Funktion verknüpft:

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

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

2.4 Parameter

Wir haben nun gesehen, das 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 Testgenauigkeit (*Accuracy*) zu wählen. Das geschieht in dem Lernverfahren von gewissen Anfangsparametern p_0 iterativ verbessert werden. Diesen iterative Vorgang nennt man "trainieren".
Dazu werden auf dem Gradientenabstieg (*Gradiant Descent*) basierende Methdoden verwenden, die eine Kostenfunktion 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 gradiant descent* (*SGD*) oder den Optimizier *ADAM*. Alle Optimizer müssen mit verschiedenen Hyperparametern wie der *learning rate* eingestellt werden, die z.b. angibt wie schnell/fein sich die Werte verbessern.
Ohne zu viel auf diese Hyperparameter eingehen zu wollen ist klar, das 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*).

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 geignete Hyperparamter zu finden. In der Praxis hat die Erfahrung gezeigt, dass manche Dinge bessern funktionieren als andere. Mit genügend Erfahrung kann man strukturiert verschiedene Hyperparameter austesten. Diese durch Erfahrung gewonnen Erkenntnisse 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 neuronale 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 Klassem viele lästige und komplizierte Aufgaben bereits abnehmen, und so auch Nicht-Experten das Experimentieren mit neuronalen Netzen ermöglichen wäre es wünschenswert auch die Wahl der Hyperparameter zu automatisieren.

Das Ziel ist es, dass ein 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 ensteht. 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

2.6 bisherige Ansätze

Aufgrund der Bedeutendheit 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, das 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 kan man ein Intervall angeben, in dem nach dem 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 Werteintervalle ein paar 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 Werteintervall Wissen einfließen lässt, wie z.B. das 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 geschatelte Suchen mit immer höherer Auflösung oder 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 können durch zufällige Wahl von Hyperparametern erstaunlicherweise viel schneller und soagar bessere Ergebnisse erzielt werden. [J B12] Für jeden Parameter wird eine Verteilung angeben, meistens eine Gleichverteilung über das logaritmische Werteintervall (siehe 2.6.1) oder eine multinomiale Verteilung bei diskreten Hyperparametern.

2.7 NEAT

Einen ganz anderen Ansatz verfolgen sogenannte genetische Algorithmen. Besonders hervorzuheben ist hier das Paper "Evolving Neural Networks through Augmenting Topologies" [K S02] aus dem Jahr 2002, 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 Netz anfangen und durch Mutation und Crossover neue immer größere Netze erzeugen, die immer besser werden. Sobald die enstehenden Netzt nicht mehr besser werden kann man aufhören.
Ein Nachteil an NEAT ist aber, dass der Algortihmus in einer Zeit entwickelt wurde, als die Computer noch nicht so viele Möglichkeiten hatten wie heutzutage und außerdem zeitdem viele Fortschritte im Bereich des machinellen 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 aktuell Zwecke nicht mehr ausreichend. Außerdem sieht *NEAT* vor das 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 uns die Grundidee für unseren Algorithmus.
Wir übertragen das Konzept in die Moderne, in dem wir nicht mit einzelnen Neuron arbeiten, sondern als Grundeinheit direkt 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 Klasifaktionsprobleme in der Bildverarbeitung wie den MNIST Datensatz 2.1 besser lösen.
Es gibt bereits ähnliche Versuche die Ideen von *NEAT* auf *Convolutional Neural Networks* wie etwa *EXACT* von T. Desell [Des17] oder einen Ansatz von Y Sun et al [YY19]. Beide Ansätze haben aber Nachteile gegenüber *convNEAT* zeigen aber das 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 leichter Erweiterbarkeit und Anpassbarkeit. Durch modernes *Clustering* können bessere Ergebnisse erzielt werden und durch die Evolution von allen Hyperparametern inklusive den Hyperparametern der Optimierers werden dem Anwender alle Entscheidungen abgenommen.
Bevor wir auf die genaueren Details von convNEAT eingehen, werden wir zuerst ein paar andere Grundlegende Probleme ansprechen.

3.1 Representation

Da wir mit einem evolutionären Algorithmus arbeiten, müssen wir eine geieignete genetische Representation finden. Eine Kodierung der Gene eines Netzes in ein Genom bestimmt wie das Netz aussieht und muss deshalb alle Hyperparameter enthält, die uns interessieren. Neben der Anzahl der Größe und *Layer* so wie deren Hyperparameter gehören aber auch alle anderen Hyperparameter z.B. auch die des Optimieres 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. Y Sun et al [YY19] verwenden z.B. Listen variabler Länge um Netze darzustellen. Der Suchraum wird beschränkt auf diejenigen *Convolutional Neural Networks*, die erst eine Reihe *Convolutional Layer* und danach eine Reihe von *Fully Connected Layer* aufweisen. Bei convNEAT wollten wir unseren Suchraum nicht so weit einschränken, sondern alle möglichen *feedforward Convolutional Neural Networks* durchsuchen.
convNEAT verwendet deshalb eine Kodierung die an die Grundidee von *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 das 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 durch das Netz propagiert und in jeder Kante bearbeitet. Für *Fully Connected Layer* wird nur gespeichert wie groß die Änderung der Größe ist, also zum Beispiel, dass die *Fully Connected Layer* 20 mehr Neuronen enthält als der Vorgänger. Es können auch inaktive Kantengene in Genom gespeichert 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 enstehen.
Außerdem ist es möglich das ein Knoten zwei eingehende Kanten besitzt, die nicht zueinander passen. In diesem Fall müssen durch Hochskalierung, Runterskalierung, einer Mischung aus beiden oder durch das Hinzufügen von Nullen die Ouputs auf die selbe Größe und konkatiniert 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 verschiedende Channels, in denen z.B. im Input Gelb, Rot und Blau kodiiert 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 Verleich zu z.B. EXACT [Des17] viel weniger Knoten gebraucht werden, weil viel mehr Faltungen kompakt representiert werden können.
In Abb. finden sich zwei Beispiel für Netze und ihre Representation.
Neben der Topologie wird auch noch kodiert welcher Optimierer gerade mit welchen Hyperparametern verwendet wird.

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. Schlechte Mutationen können in der Selektion aussortiert werden, gute Mutationen bleiben erhalten. Welche Mutation passiert ist zufällig, manche haben jedoch höhere Wahrscheinlichkeit als andere. Die Wahrscheinlihckieten wurden durch viele Testläufe so angepasst, dass Sie sinnvoll sind. Diese Hyper-Hyperparameter müssen jedoch nur einmal bestimmt werden und sind Porblemunabhängig. Trotzdem wäre es denkbar auch eine adaptive Veränderung der Wahrscheinlichkieten einzuabeun, um die Mutationen häufiger auftretten zu lassen, die häufiger zu guten Netzen führen.
convNEAT bietet folgende Mutationen:

Aktivieren und Deaktivieren von Genen

Es gibt 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* entstehen. Beim Deaktivieren muss sichergestellt sein, dass es noch einen Pfad vom Input zum Output gibt, damit noch ein gültiges Netz entsteht.

Kante aufteilen
Es gibt 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 hiner einer *Convolutional Layer* eine neue *Pooling Layer* am wahrscheinlichsten.

Kante einfügen
Zwischen zwei Knoten kann eine Kante eingefügt werden. Wie beim Aufteilen 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 dann immer Knoten in echt absteigender Reihenfolge, 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 Änderung der Größe können mutieren.
>
> **Convolutional Layer doer Pooling Layer**
> Die Weite und 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 auf neuen Daten keine guten Vorhersagen macht, nicht mehr gut generalisiert, wird wie es üblich ist 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 Netzes 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, das heißt:

Definition 3.1 (Log classification error)**.**
Der logarithmische Klassifikationsfehler eines Netzes mit Accuracy a ist definiert als

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

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

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

Denn so gilt:

$$logerr(score(a, p)) = logerr(a) + n * decay$$

3.4 Crossover

Eine wetere Operation in genetischen Algorithmen ist der *Crossover*. Er erzeugt aus zwei Genomen, ein neues Genom. Dieses neue Genome erhält alle Gene der Eltern. Wichtig für *convNEAT* sind, wie bei *EXACT* [Des17], die Parameter *more fit parent crossover rate* und *less fit parent crossover rate*. Diese beiden beschreiben wie viel Prozent des jeweiligen Genomes aktiviert bleiben soll. Gene, also vor allem Kanten, die im besseren (was genau besser hier beduetet unter 3.3) Die anderen Gene werden deaktiviert insofern das möglich ist.

References

[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: (Sept. 16, 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.

[Des17] T. Desell. "Large Scale Evolution of Convolutional Neural Networks Using Volunteer Computing". In: (2017). URL: https://arxiv.org/pdf/1703.05422.pdf.

[YY19] M. Zhang Y. Sun B. Xue and G. Yen. "Evolving Deep Convolutional Neural Networks for Image Classification". In: (2019). URL: https://arxiv.org/pdf/1710.10741.pdf.

Declaration

I, Mark-Oliver Wolf, avouch that I have created this thesis on my own and without help or sources but those explicitly mentioned and that I have marked any and all citations as such.

Kaiserslautern, May 16, 2020

Mark-Oliver Wolf