technische universität
dortmund

Bachelorarbeit

Evolution Strategies als Trainingsmethode für neuronale Netze

Jan Kemming
März 2018

Gutachter:
Prof. Dr. Katharina Morik
Sebastian Buschjäger

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
http://www-ai.cs.uni-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Künstliche neuronale Netze (ANN) sind ein beliebtes Mittel zur Modellierung aktueller Probleme aus den verschiedensten Anwendungsbereichen. Sie sind besonders geeignet in Fällen, wo die komplexen Zusammenhänge zwischen den vorliegenden Daten und dem gewünschten Ergebnis für Menschen nur schwer erkennbar sind. Beispiele dafür sind die Gesichtserkennung in Bildern [17], die Vorhersage von Sonnenstrahlung [13], sowie die Erkennung von Hautkrebs [4].

Das begründet sich darin, dass zur Modellierung eines Problems mit ANN lediglich eine Menge von Trainingsdaten zum Lernen zur Verfügung stehen muss. Damit kann das Netzwerk darauf trainiert werden zu erkennen, welche Merkmale für die Ausgabe besonders relevant sind und in welchen Zusammenhängen sie stehen. Es muss also nicht beispielsweise auf das eventuell unvollständige und nur schwer zu formalisierende Wissen von Experten aus dem jeweiligen Fachgebiet zurückgegriffen werden. Das Training von ANN dauert in der Regel jedoch lange und ist sehr rechenintensiv. Kleinere Geräte oder eingebettete Systeme sind daher dafür meist nicht geeignet. Es wird deshalb gewöhnlich auf besonders leistungsstarke Computer oder spezialisierte Hardware zurückgegriffen. Dadurch entstehen hohe Kosten für Anschaffung, Stromverbrauch und Kühlung.

Wegen der immer weiter voranschreitenden Einbindung von elektronischen Geräten in unser Leben erscheint hier eine Nutzung von ANN jedoch immer sinnvoller. Die Anwendungsmöglichkeiten dabei sind vielfältig. Besonders mobile Geräte besitzen oft eine Vielzahl an Sensoren, deren Daten auf verschiedenste Arten genutzt werden könnten, um uns im Alltag zu unterstützen.

1.2 Ziel

Im Rahmen dieser Arbeit soll eine Methode vorgestellt werden, mit der ANN auch auf weniger spezialisierten Geräten effektiv trainiert und verwendet werden können. Die wichtigste Einschränkung solcher Geräte besteht meist in ihrer begrenzten Rechenleistung. Um die bestehenden Ressourcen in dieser Hinsicht besser nutzen zu können, werden *Evolution Strategies* (ES) eingesetzt. Diese erlauben es, im Gegensatz zu anderen gängigen Methoden, den Trainingsprozess relativ einfach zu parallelisieren. Anstatt in jedem Schritt nur ein einziges Trainingsbeispiel zu bearbeiten, können mehrere gleichzeitig ablaufende Prozesse unabhängig voneinander lernen. Die Ergebnisse daraus können dann zusammengeführt und zur Optimierung genutzt werden.

Außerdem ist noch der begrenzte zur Verfügung stehende Speicherplatz auf vielen Geräten ein Problem. Der Speicherbedarf wird daher verringert, indem statt klassischen ANN mit reellwertigen Gewichten, binäre neuronale Netze verwendet werden. Deren Gewichte sind auf nur zwei mögliche Werte begrenzt. Binäre Gewichte vereinfachen zudem stark die Rechenoperationen, die zur Berechnung der Ausgabe eines ANN durchgeführt werden müssen. Somit wird ein weiterer Geschwindigkeitszuwachs erreicht.

Insgesamt soll also ein Algorithmus zum Training von ANN entwickelt, der besonders zum Einsatz in Umgebungen mit begrenzten Ressourcen an Rechenleistung und Speicherplatz geeignet ist. In der Praxis könnte ein ANN so zum Beispiel für ein spezifisches Problem auf einem Computer mit mehreren Rechenkernen, aber ohne spezielle Hardware wie GPUs trainiert werden. Diese Art Umgebung ist heute bereits auf vielen Anwendersystemen vorzufinden. Denkbar wäre auch ein solches Vortrainieren des neuronalen Netzes mit anschließender Nutzung auf mobilen Geräten wie Smartphones oder Tablets. Hier könnte das Training sogar beispielsweise als Reaktion auf Nutzereingaben noch weiter fortgeführt werden.

1.3 Aufbau

Die Arbeit ist folgendermaßen aufgebaut. Zuerst werden in Kapitel 2 die Grundlagen zu maschinellem Lernen, sowie künstlichen neuronalen Netzen und *Evolution Strategies* erläutert. Dann werden in Kapitel 3 einige thematisch relevante Arbeiten vorgestellt. In Kapitel 4 wird die Problemstellung noch einmal herausgearbeitet und die Implementierung des Algorithmus erklärt. Kapitel 5 beinhaltet eine Beschreibung der durchgeführten Versuche und die Auswertung der Ergebnisse. Zuletzt wird in Kapitel 6 noch ein abschließendes Fazit aus den Ergebnissen gezogen.

Kapitel 2

Grundlagen

2.1 Maschinelles Lernen

Maschinelles Lernen ist der Prozess bei dem Computer möglichst selbstständig algorithmisch, anhand von Beispielen, Muster und Zusammenhänge in Daten erkennen [14]. Im Allgemeinen wird versucht, für eine Menge an Eingabedaten Regeln zu finden, die diese Daten möglichst gut beschreiben. Auf dieser Grundlage können dann auch Aussagen über bisher unbekannte Eingabedaten getroffen werden.

Um diese Regeln festzuhalten, sie zu optimieren und anzuwenden, werden sie mithilfe von Modellen repräsentiert. Beispiele dafür sind unter Anderem künstliche neuronale Netze (Abschnitt 2.2), *Support Vector Machines* [2] und *Naive Bayes Classifiers* [11].

Ein Programm welches maschinelles Lernen nutzt, versucht also die Parameter eines Modells durch das Lernen von Trainingsbeispielen für ein bestimmtes Problem zu optimieren. Beim überwachten Lernen bestehen die Trainingsbeispiele dabei im Gegensatz zum unüberwachten Lernen nicht nur aus den Eingaben, sondern beinhalten zusätzlich noch die tatsächlich gewünschte Ausgabe des Modells.

Formal kann ein solches Modell also als Funktion f_θ betrachtet werden, welche die Menge der möglichen Eingaben $\mathcal{X}$ in Abhängigkeit ihrer Parameter θ auf die Menge der möglichen Ausgaben $\mathcal{Y}$ abbildet.

$$f_\theta : \mathcal{X} \to \mathcal{Y} \tag{2.1}$$

Die Trainingsbeispiele sind daher eine Menge von Paaren aus Modelleingaben mit den zugehörigen korrekten Ausgaben.

$$(x, \hat{y}) \in \mathcal{X} \times \mathcal{Y} \tag{2.2}$$

Das Ziel der Optimierung ist es, eine Kosten- oder Fehlerfunktion zu minimieren. Diese gibt für die aus der Eingabe x eines Trainingsbeispiel resultierende Ausgabe y ein Maß für die Abweichung von der korrekten Ausgabe $\hat{y}$ an.

$$E : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R} \tag{2.3}$$

Das Programm muss also für die Trainingsbeispiele den Fehler des Modells mit den aktuellen Parametern berechnen und diese dann so anpassen, dass der Fehler sich verkleinert. Dadurch kommt es dann zu einer Verbesserung der ausgegebenen Problemlösungen.

2.2 Künstliche neuronale Netze

Künstliche neuronale Netze (ANN) sind ein Modell des maschinellen Lernens, welche in ihrem Aufbau dem menschlichen Gehirn nachempfunden sind [6]. Sie bestehen in Anlehnung an dieses aus einer Menge von in Schichten angeordneten künstlichen Neuronen, welche durch gerichtete und gewichtete Kanten miteinander verbunden sind. Die folgende Abbildung zeigt ein einfaches neuronales Netz mit drei Schichten.

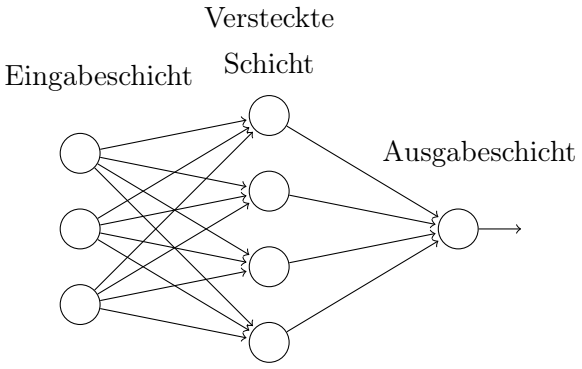


Abbildung 2.1: Aufbau eines künstlichen neuronalen Netzes

An der Eingabeschicht kann jeweils die Eingabe für das vom Netzwerk zu lösende Problem angelegt werden. Diese wird dann über eine beliebige Anzahl versteckter Schichten entlang der Kanten durch das Netzwerk propagiert. Gibt es mehrere versteckte Schichten spricht man von einem tiefen neuronalen Netz. Die berechnete Problemlösung liegt zum Schluss an der Ausgabeschicht an.

Die künstlichen Neuronen in jeder Schicht gewichten dabei die Eingaben ihrer eingehenden Kanten mit den entsprechenden Kantengewichten und bilden die Summe der Ergebnisse. Diese Summe wird in eine Aktivierungsfunktion eingesetzt, welche bestimmt, ob bzw. wie stark das Neuron aktiviert wird. Der Aufbau eines künstlichen Neurons ist in Abbildung 2.2 dargestellt. Je stärker ein Neuron aktiviert wird, desto größer ist sein Einfluss auf die Neuronen in der darauffolgenden Schicht. Meistens werden für die Aktivierungsfunktion nicht-lineare Funktionen wie die Sigmoid-Funktion (Formel 2.4) verwendet, damit auch nicht-lineare Zusammenhänge von Netzwerkeingabe und Netzwerkausgabe modelliert werden können.

$$\varphi(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

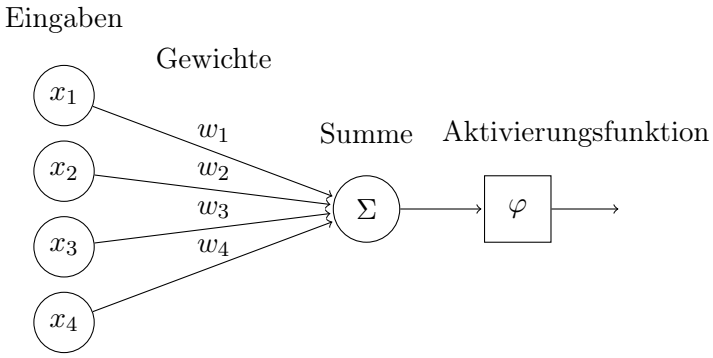


Abbildung 2.2: Aufbau eines künstlichen Neurons

Seien also $x = (x_1...x_m)$ die Ausgaben der Neuronen der vorhergehenden Schicht, w_{jk} das Gewicht der Kante, welche das j-te Neuron der vorhergehenden Schicht mit dem k-ten Neuron der aktuellen Schicht verbindet, und φ die Aktivierungsfunktion. Dann lässt sich die Ausgabe y_k des k-ten Neurons der aktuellen Schicht durch die folgenden Funktionen darstellen.

$$y_k = \varphi(u_k) \tag{2.5}$$

$$u_k = \sum_{j=1}^{m} w_{jk} x_j \tag{2.6}$$

Ausschlaggebend für eine korrekte Ausgabe sind also die Kantengewichte. Optimiert man diese für Eingaben eines bestimmten Problems, dann kann ein solches ANN für Eingaben zu dem Problem mit relativ geringem Aufwand mögliche Lösungen berechnen. Eine verbreitete Methode für diese Optimierung ist der in Abschnitt 2.2.1 näher erklärte *backpropagation* Algorithmus.

2.2.1 Backpropagation

Backpropagation ist ein Algorithmus zum trainieren von tiefen neuronalen Netzen beim überwachten Lernen [8]. Der Algorithmus optimiert also die Gewichte in einem neuronalen Netz schrittweise mithilfe von Trainingseingaben und den dazugehörigen gewünschten Ausgaben. In jeder Iteration wird dabei eine Trainingseingabe in das Netzwerk eingegeben und im ersten Schritt vorwärts durch das Netzwerk bis zur Ausgabeschicht propagiert. Ausgehend von einer Fehlerfunktion, wie beispielsweise der *squared error* Funktion (Formel 2.7), wird dann ein Gradientenabstieg entlang derselben durchgeführt, sodass der Fehler minimiert wird. Der Faktor $\frac{1}{2}$ vereinfacht dabei lediglich eine spätere Ableitung.

$$E(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{2.7}$$

Der Gradientenabstieg basiert auf der Idee, dass sich der Wert der Fehlerfunktion verringert, wenn man die Gewichte in Richtung des negativen Gradienten der Fehlerfunktion anpasst. Man folgt also der negativen Steigung der Funktion um ein Minimum zu erreichen.

Die Gewichte in jeder Schicht werden dabei jeweils mithilfe des zurückpropagierten Fehlers aus der vorhergehenden Schicht angepasst. Formal ergeben sich am Beispiel der Sigmoid Aktivierungsfunktion und squared-error Fehlerfunktion die folgenden Berechnungen.

Sei $u_j^{(l)}$ die Summe der Eingaben von Neuron j (Formel 2.6) in Schicht l, $y_j^{(l)}$ bzw. $\hat{y}_j^{(l)}$ die Ausgabe bzw. korrekte Ausgabe des Neurons, $w_{jk}^{(l)}$ das Gewicht der Kante, welche das j-te Neuron in Schicht l mit dem k-ten Neuron der nachfolgenden Schicht $l+1$ verbindet und φ die Aktivierungsfunktion. Dann lässt sich die Ableitung der Fehlerfunktion (Formel 2.8) in Abhängigkeit vom Fehlersignal (Formel 2.9) schreiben.

$$\frac{\partial E}{\partial w_{jk}^{(l)}} = y_j^{(l)} \cdot \delta_k^{(l)} \tag{2.8}$$

$$\delta_j^{(l)} = \begin{cases} \varphi'(x_j^{(l)}) \frac{\partial E}{\partial y_j^{(l)}} = \varphi(1 - \varphi(x_j^{(l)})) \cdot -(y_j^{(l)} - \hat{y}_j^{(l)}) & \textit{in der Ausgabeschicht} \\ \varphi'(x_j^{(l)}) \frac{\partial E}{\partial y_j^{(l)}} = \varphi(1 - \varphi(x_j^{(l)})) \sum\limits_k \delta_k^{(l+1)} w_{jk}^{(l+1)} & \textit{sonst} \end{cases} \tag{2.9}$$

Zur Berechnung der neuen Gewichte wird also die abgeleitete Fehlerfunktion mit der Lernrate α gewichtet und dann von den alten Gewichten subtrahiert (Formel 2.10). Die Lernrate steuert dabei die Geschwindigkeit des Gradientenabstiegs.

$$w_{jk}^l = w_{jk}^l - \alpha \frac{\partial E}{\partial w_{jk}^l} \tag{2.10}$$

2.2.2 Binäre neuronale Netze

Binäre neuronale Netze (BNN) sind neuronale Netze deren Gewichte auf lediglich zwei mögliche Werte beschränkt sind. Einige Varianten davon wurden bereits in Kapitel 3 vorgestellt.

Die starke Reduzierung der Genauigkeit der Parameter hat den Vorteil, dass der Speicherbedarf für ein neuronales Netz mit binären Gewichten deutlich geringer als der eines vergleichbaren Netzes mit reellwertigen Gewichten. Ein einzelnes Bit pro Gewicht reicht damit für die Parametrisierung aus. Die genauen Größenunterschiede wurden in [16] am Beispiel von VGG-19 [19], ResNet-18 [7] und AlexNet [9] ermittelt.

Ein weiterer Vorteil ist, dass sich die Rechenoperationen, die häufig während des Trainings und der Berechnung der Ausgabe eines neuronalen Netzes verwendet werden, deutlich

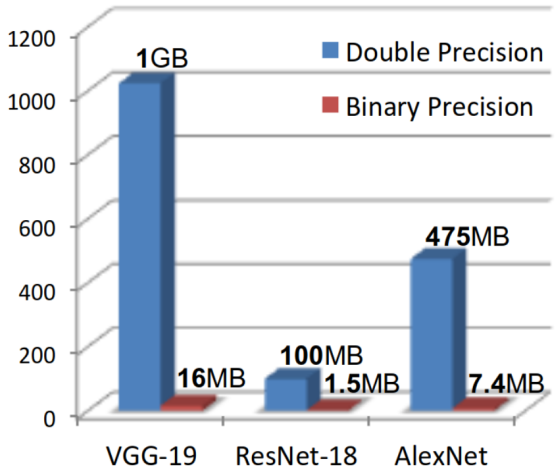


Abbildung 2.3: Abnahme des Speicherbedarfs von VGG-19, ResNet-18 und AlexNet durch Verwendung von binären Gewichten [16]

vereinfachen. Anstatt wie in Formel 2.6 und 2.9 in jedem Schritt eine hohe Anzahl an Multiplikation mit Fließkommazahlen durchzuführen, reicht dann eine einfache Aufsummierung der entsprechenden Werte.

2.3 Evolution Strategies

Evolution Strategies (ES) sind eine Klasse von genetischen Algorithmen, welche auf der Idee des natürlichen Phänomens der Evolution basieren [1]. Ihr Ziel ist es, die Parameter einer Zielfunktion durch simulierte Evolution zu optimieren.

Die Parameter y_k werden dazu jeweils als Teil von Individuen a_k in Populationen betrachtet. Das Einsetzen einer der Parameter in die Zielfunktion ergibt die Fitness $F(y_k)$ des Individuums. Eine besondere Eigenschaft von ES, durch die sie sich von anderen genetischen Algorithmen unterscheiden, ist die Fähigkeit zur Selbstadaption. Das bedeutet, dass ein Individuum nicht nur die für die Zielfunktion relevanten Parameter enthält, sondern zusätzlich auch Strategieparameter s_k, welche die Evolution beeinflussen. Insgesamt ist ein Individuum also Tupel $a_k = (y_k, s_k, F(y_k))$ definiert.

Ein ES Algorithmus hat immer eine Elternpopulation und eine Kinderpopulation. Die einzelnen Iterationen werden daher auch Generationen genannt. In jeder Generation werden aus der Elternpopulation neue Kinder generiert, die zu Eltern der nächsten Generation werden können. Die verschiedenen ES Variationen unterscheiden sich unter Anderem in der Größe dieser beiden Populationen. Die gängige Notation um einen ES zu beschreiben ist $(\mu/\rho^{+}_{,}\lambda)$. μ ist dabei die Größe der Elternpopulation und λ die Größe der Kinderpopulation. ρ gibt an, wie viele Individuen aus der Elternpopulation jeweils an der Erzeugung eines Individuums in der Kinderpopulation beteiligt sind. Das $+$ bzw. , gibt die Art der

Selektion an. Bei +-Selektion können die Individuen für die neue Elternpopulation sowohl aus der Kinderpopulation, als auch aus der aktuellen Elternpopulation entnommen werden. ,-Selektion beschränkt die Auswahl hingegen auf die Kinderpopulation. Der allgemeine Ablauf von ES wird in Algorithmus 1 mit ,-Selektion gezeigt.

Algorithmus 1 : $(\mu/\rho, \lambda)$ Evolution Strategies

initialisiere Generation $g \leftarrow 0$;
initialisiere Elternpopulation $\mathfrak{P}_p^{(0)} \leftarrow \{(y_m^{(0)}, s_m^{(0)}, F(y_m^{(0)}) \mid m = 1...\mu)\}$;
repeat
  for $l = 1...\lambda$ **do**
    wähle zufällige Eltern $(y_1^{(g)}, s_1^{(g)}, F(y_1^{(g)}))...(y_\rho^{(g)}, s_\rho^{(g)}, F(y_\rho^{(g)}))$ aus $\mathfrak{P}_p^{(0)}$;
    $s_l \leftarrow$ rekombiniere $s_1^{(g)}...s_\rho^{(g)}$;
    $y_l \leftarrow$ rekombiniere $y_1^{(g)}...y_\rho^{(g)}$;
    $\tilde{s}_l \leftarrow$ mutiere s_l;
    $\tilde{y}_l \leftarrow$ mutiere y_l mit $\tilde{s}_l$;
    $\tilde{F}_l \leftarrow F(\tilde{y}_l)$;
  end
  $\mathfrak{P}_o^{(g)} \leftarrow \{(\tilde{y}_l, \tilde{s}_l, \tilde{F}_l) \mid l = 1...\lambda)\}$;
  $\mathfrak{P}_p^{(g+1)} \leftarrow$ selektiere μ neue Eltern aus $\mathfrak{P}_o^{(g)}$;
  $g \leftarrow g + 1$
until *Stop-Kriterium erfüllt*;

Der Algorithmus verwendet dabei einen Selektionsoperator, einen Mutationsoperator und einen Rekombinationsoperator, die festlegen, wie die entsprechende Operation durchgeführt wird. Der Selektionsoperator wählt normalerweise, je nach Art der Selektion (, oder +), aus den entsprechenden Populationen die Individuen mit der jeweils höchsten Fitness aus. Der Mutationsoperator verändert Individuen zufällig und ist abhängig von den zu mutierenden Parametern. Sind die beispielsweise reellwertig, dann könnte die Mutation als Addition eines normalverteilten Vektors $\epsilon \sim \mathcal{N}(\sigma, I)$ implementiert werden. Die Stärke der Mutation σ wäre dann Teil der Strategieparameter s_l des Individuums. Der Rekombinationsoperator erzeugt ein Kind durch Kombination der Eltern. Die Standardrekombinationsoperatoren für ES sind *discreet recombination* und *intermediate recombination*. Discreet recombination wählt für jeden Parameter des Kindes zufällig den entsprechenden Parameter von einem Elternteil. Intermediate recombination setzt die Parameter dagegen auf den Mittelwert der Parameter der Eltern.

2.4 Natural Evolution Strategies

Natural Evolution Strategies (NES) sind eine Familie von ES, welche Schrittweise eine Wahrscheinlichkeitsverteilung (WV) für eine Fitnessfunktion optimieren, indem sie den

sogenannten *natural gradient* ihrer Parameter approximieren [20]. Die WV wird auch Suchverteilung genannt, da in ihr nach den möglichen Problemlösungen gesucht wird. Zunächst wird dazu in jedem Optimierungsschritt aus der WV eine Menge Punkten generiert, an denen jeweils die Fitnessfunktion ausgewertet wird. Anhand der Ergebnisse davon wird dann der Gradient der Parameter in Richtung der höheren erwarteten Fitness approximiert. Dieser einfache Gradient wird dann bezüglich der Unsicherheit der Parameteränderung normalisiert, um den natural gradient zu erhalten. Zum Schluss wird ein Gradientenaufstieg entlang des natural gradients durchgeführt.

Als erstes wird also eine Approximation für den Gradienten der erwarteten Fitness benötigt. Für eine WV mit Dichte $\pi(z|\theta)$ und Fitnessfunktion $f(z)$, lässt sich die erwartete Fitness insgesamt folgendermaßen schreiben.

$$J(\theta) = \mathbb{E}_\theta[f(z)] = \int f(z)\pi(z|\theta)dz \tag{2.11}$$

Daraus ergibt sich der Gradient der erwarteten Fitness.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int f(z)\pi(z|\theta)dz \\
&= \int f(z)\nabla_\theta \pi(z|\theta)dz \\
&= \int f(z)\nabla_\theta \pi(z|\theta)\frac{\pi(z|\theta)}{\pi(z|\theta)}dz \\
&= \int [f(z)\nabla_\theta \log \pi(z|\theta)]\pi(z|\theta)dz \\
&= \mathbb{E}_\theta[f(z)\nabla_\theta \log \pi(z|\theta)]
\end{aligned} \tag{2.12}$$

Dieser kann dann mit Beispielpunkten $z = (z_1 ... z_\lambda)$ approximiert werden.

$$\nabla_\theta J(\theta) \approx \frac{1}{\lambda}\sum_{k=1}^{\lambda} f(z_k)\pi(z_k|\theta)dz \tag{2.13}$$

Würde man für den Gradientenaufstieg einfach nur ∇J folgen, dann wären die Parameter der neuen Verteilung je nach Schrittgröße ϵ ähnlich den alten Parametern. Selbst ein kleiner Schritt im Parameterraum kann jedoch große Auswirkung auf das Aussehen der Verteilung haben. Der natural gradient versucht das auszugleichen, indem er den entsprechenden Einfluss der Parameteränderung auf die Verteilung mit einbezieht. Ein Maß für die Ähnlichkeit bzw. den Abstand zweier WV mit Parametern θ und θ' ist die Kullback-Leibler Divergenz $D(\theta \mid \theta')$ [10].

Der natural gradient soll somit das folgende Optimierungsproblem lösen.

$$\begin{aligned}
&\max_{\delta\theta} J(\theta + \delta\theta) \approx J(\theta) + \delta\theta^T \nabla_\theta J, \\
&mit\ D(\theta + \delta\theta||\theta) = \epsilon
\end{aligned} \tag{2.14}$$

Für $\delta\theta \to 0$ ergibt sich daraus

$$D(\theta + \delta\theta||\theta) = \frac{1}{2}\delta\theta^T F(\theta)\delta\theta, \tag{2.15}$$

wobei

$$\begin{aligned} F =& \int \pi(z|\theta)\nabla_\theta \log \pi(z|\theta)\nabla_\theta \log \pi(z|\theta)^T dz \\ =& \mathbb{E}[\nabla_\theta \log \pi(z|\theta)\nabla_\theta \log \pi(z|\theta)^T] \end{aligned} \tag{2.16}$$

die Fisher Matrix für die Verteilung ist. Diese kann wie folgt approximiert werden.

$$F \approx \frac{1}{\lambda}\sum_{k=1}^{\lambda} \nabla_\theta \log \pi(z_k|\theta)\nabla_\theta \log \pi(z_k|\theta)^T \tag{2.17}$$

Der natural gradient ergibt sich dann aus der Skalierung der erwarteten Fitness mit dem Inversen der Fisher Matrix.

$$\tilde{\nabla}_\theta J = F^{-1}\nabla_\theta J(\theta) \tag{2.18}$$

Das Inverse der Fisher-Information stellt dabei eine untere Schranke für die Varianz der Parameter dar. Ist also die Unsicherheit in einer Dimension groß, dann ist der natural gradient an dieser Stelle entsprechend klein und umgekehrt. Dadurch ist die Anpassung in jeder Iteration abhängig von dem Einfluss, den eine entsprechende Parameteränderung auf die resultierende WV hat. So wird erreicht, dass die mit den neuen Parametern generierten Lösungen, unabhängig von der Parametrisierung der WV, immer relativ ähnlich zu denen aus der vorherigen WV sind. Algorithmus 2 zeigt den gesamten Ablauf von NES.

Algorithmus 2 : Natural Evolution Strategies

Input : f, θ

repeat
  for $k = 1...\lambda$ **do**
    generiere $z_k \sim \pi(\cdot|\theta)$;
    berechne Fitness $f(z_k)$;
    berechne Score $\nabla_\theta \log \pi(z_k|\theta)$;
  end
  $\nabla_\theta J \leftarrow \frac{1}{\lambda}\sum_{k=1}^{\lambda} \nabla_\theta \log \pi(z_k|\theta) \cdot f(z_k)$;
  $F \leftarrow \frac{1}{\lambda}\sum_{k=1}^{\lambda} \nabla_\theta \log \pi(z_k|\theta)\nabla_\theta \log \pi(z_k|\theta)^T$;
  $\theta \leftarrow \theta + \eta \cdot F^{-1}\nabla_\theta J$;
until *Stop-Kriterium erfüllt*;

Kapitel 3

Verwandte Arbeiten

Im Folgenden werden einige bisherige Arbeiten vorgestellt, die sich mit der Parallelisierung von *Evolution Strategies* (ES) und neuronalen Netzen mit binären Gewichten befassen.

Salimans et al. haben die Parallelisierbarkeit von ES im Zusammenhang mit *Reinforcement Learning* untersucht [18]. Die Grundidee ist, dass die Generierung von möglichen Parametern auf mehrere hundert unabhängige Prozesse verteilt wird. Jeder dieser Prozesse verändert dabei die aktuellen Parameter für sich durch die Addition von zufällig generiertem Rauschen. Die so entstehenden neuen Parameter werden dann getestet und der entsprechende Ertrag berechnet. Wenn das für alle Prozesse geschehen ist, werden die Erträge mit allen parallel abgelaufenen Prozessen geteilt. Die Prozesse kennen dabei den Wert mit dem die Zufallsgeneratoren der anderen Prozesse initialisiert wurden. Es kann so jeder Prozess das von den anderen Prozessen generierte Rauschen rekonstruieren. Zusammen mit dem empfangenen Ertrag wird dann beides zur Anpassung der Parameter genutzt, sodass der Ertrag maximiert wird. Dazu wird jedes Rauschen mit dem entsprechenden Ertrag gewichtet und der Durchschnitt über die Anzahl der Prozesse zu den aktuellen Parametern addiert. Experimentell wurde für dieses Vorgehen gezeigt, dass die benötigte Rechenzeit linear mit der Anzahl der verwendeten Rechenkerne skaliert. Dies gilt selbst noch bei über 1000 verteilten Prozessen.

Die von Courbariaux et al. entwickelte Methode *BinaryConnect* ist eine Abwandlung des *backpropagation* Algorithmus, bei der die Gewichte während des Vorwärts- und Rückwärtsschrittes auf die Werte $+1$ und -1 beschränkt sind [3]. Versuche haben gezeigt, dass neuronale Netze so deutlich schneller, aber ohne großen Verlust an Genauigkeit der Ausgabe trainiert werden können. Die echten, reellwertigen Gewichte werden hierbei jedoch weiterhin abgespeichert und vor jeder Iteration erneut auf binäre Gewichte reduziert. Nach jedem Durchlauf werden dann die echten Gewichte mit den auf Grundlage der binären Gewichte berechneten Gradienten angepasst. So konnten die Autoren auf verschiedenen Datensätzen Fehlerraten erzielen, welche vergleichbar sind mit Ergebnissen von verbreiteten Methoden wie *Maxout Networks* [5] und *Deeply-Supervised Nets* [12].

Rastegari et al. stellen in [16] zwei Varianten von *convolutional neural networks* (CNN) vor, welche an verschiedenen Stellen binäre Werte verwenden. In *Binary-Weight-Networks* werden alle Gewichte mit binären Werten approximiert. Für die Berechnungen sind so nur noch Additionen und Subtraktionen, aber keine Multiplikationen mehr notwendig. Das resultiert in einer ungefähren Verdopplung der Geschwindigkeit. Als Erweiterung dazu sind bei *XNOR-Networks* sowohl die Eingaben, als auch die Gewichte binär. Damit können die *convolutions* effizient in Form von XNOR Operationen und dem Zählen von Bits implementiert werden. Die geringere Genauigkeit der Werte hat dabei in Versuchen keinen großen Einfluss auf die Ergebnisse gehabt. XNOR-Networks erreichen so einen ungefähren Geschwindigkeitszuwachs um das 58-fache gegenüber klassischen CNN.

Kapitel 4

Ansatz

4.1 Problemstellung

Die derzeitige Standardmethode zum trainieren von ANN ist *Backpropagation* (Abschnitt 2.2.1). Der Algorithmus ist zwar sehr effektiv, dafür aber auch sehr rechenaufwändig. Um gute Ergebnisse zu erzielen muss eine große Menge an Trainingsbeispielen gelernt werden. In jeder Iteration müssen dabei eine hohe Anzahl an Additionen und Multiplikationen durchgeführt werden.

In der Praxis werden diese oft in Form von vielen großen Matrixmultiplikationen implementiert. Das ermöglicht die Nutzung von spezialisierter Hardware, wie GPUs oder FPGAs, die Matrixmultiplikationen besonders schnell und effizient durchführen können. Die für das Training benötigte Zeit kann so zwar stark reduziert werden, jedoch ist die Nutzung von spezialisierter Hardware längst nicht in allen Umgebungen möglich. Gerade kleinere mobile Geräte und eingebettete Systeme haben oft nicht die nötigen Ressourcen an Energie für Betrieb und Kühlung, sowie Raum zur Verfügung.

Zudem ist das zurückpropagierte Fehlersignal in jeder Schicht immer vom Fehlersignal aus den zuvor bearbeiteten Schichten abhängig (Formel 2.9). Eine Parallelisierung der Gradientenberechnung ist somit nur schwer zu realisieren. Viele der Geräte die Verbrauchern heutzutage zur Verfügung stehen besitzen aber bereits mehrere Rechenkerne und sind für parallele Berechnungen geeignet. Dazu gehören zum Beispiel Smartphones, Tablets oder Smartwatches. Ein großer Teil der ohnehin stark begrenzten Rechenleistung, die auf diesen Geräten zur Verfügung steht, bleibt so ungenutzt.

Auch der Speicherverbauch von ANN kann auf vielen Geräten ein Problem darstellen. Selbst kleine ANN können schon mehrere zehntausend Gewichte benötigen, deren Anzahl exponentiell mit der Größe der Eingabe und der Anzahl der Neuronen in jeder Schicht steigt. Die Nutzbarkeit auf kleinen und mobilen Geräten wird dadurch noch weiter eingeschränkt.

4.2 Algorithmus

Der Algorithmus versucht durch die Kombination der Vorteile von ES und BNN die genannten Probleme zu minimieren. Die Gewichte sollen dabei auf eine Weise optimiert werden, die auch bei alleiniger Verwendung von CPUs noch schnell und akkurat ist. Zudem soll das fertig optimierte Netz einen geringen Speicherplatzverbauch aufweisen, damit es auch auf den erwähnten ressourcenschwachen Geräten verwendet werden kann.

Zudem verwendet der Algorithmus NES um das Training mit den Trainingsbeispielen zu parallelisieren. Der Ablauf ist damit im Grunde sehr ähnlich zu Algorithmus 2. Anstatt klassische ANN zu verwenden, werden jedoch BNN trainiert. So wird der Speicherverbrauch verringert und die Auswertung beschleunigt. NES bieten sich deshalb besonders an, da gerade wegen der binären Gewichten schon bei kleinen Veränderungen der Parameter ein deutlich anderes Verhalten des ANN zu erwarten ist.

Um binäre Gewichte für ein BNN zu generieren wird eine entsprechende Wahrscheinlichkeitsverteilung benötigt. Diese ist als eine Art multivariate Verteilung aus Bernoulli Zufallsvariablen $Y = (Y_1, ..., Y_d)$ definiert. Die Parameter $\theta = (\theta_1 ... \theta_d)$ sind jeweils die Erfolgswahrscheinlichkeiten der entsprechenden Zufallsvariable. Ein positives Ergebnis bedeutet dabei ein Gewicht von $+1$ und ein negatives Ergebnis ein Gewicht von -1. Bei der Auswertung muss ein Neuron die Eingabe so also nur noch für ein Gewicht von $+1$ addieren und für ein Gewicht von -1 subtrahieren. Zu Beginn werden die Erfolgswahrscheinlichkeiten auf zufällige Werte zischen 0 und 1 initialisiert, sodass es am Anfang ungefähr gleich viele $+1$ und -1 Gewichte gibt. Da die Zufallsvariablen unabhängig voneinander sind ergibt sich die Dichtefunktion einfach durch Multiplikation der Dichten der einzelnen Zufallsvariablen.

$$\pi(x|\theta) = \prod_{i=1}^{d} \theta_i^{x_i} (1 - \theta_i)^{1-x_i} \tag{4.1}$$

Um die später benötigte Ableitung der Dichtefunktion zu vereinfachen wird erst ihr Logarithmus berechnet.

$$\begin{aligned} log(\pi(x|\theta)) &= log(\prod_{i=1}^{d} \theta_i^{x_i} (1 - \theta_i)^{1-x_i}) \\ &= \sum_{i=1}^{d} log(\theta_i^{x_i} (1 - \theta_i)^{1-x_i}) \\ &= \sum_{i=1}^{d} log(\theta_i^{x_i}) + log((1 - \theta_i)^{1-x_i}) \\ &= \sum_{i=1}^{d} x_i log(\theta_i) + (1 - x_i) log(1 - \theta_i) \end{aligned} \tag{4.2}$$

$$\begin{aligned}\nabla_{\theta_i} log(\pi(x|\theta)) &= \nabla_{\theta_i} \sum_{j=1}^{d} x_j log(\theta_j) + (1 - x_j) log(1 - \theta_j) \\ &= \sum_{j=1}^{d} x_j \nabla_{\theta_i} log(\theta_j) + (1 - x_j) \nabla_{\theta_i} log(1 - \theta_j) \\ &= x_i \nabla_{\theta_i} log(\theta_j) + (1 - x_i) \nabla_{\theta_i} log(1 - \theta_j) \\ &= \frac{x_i}{\theta i} - \frac{1 - x_i}{1 - \theta_i}\end{aligned} \tag{4.3}$$

In jedem Schritt werden gleichzeitig in mehreren Prozessen aus der Verteilung Gewichte für das BNN erzeugt. Um den Gradientenaufstieg durchzuführen muss zuerst die Fitness des BNN für die Gewichte approximiert werden. Dazu wird der Fehler mit einigen Trainingsbeispielen evaluiert. Die Anzahl der dabei verwendeten Trainingsbeispiele λ ist ein Parameter des Algorithmus. Für ein BNN mit Gewichten w und Trainingseingaben $x = (x_1...x_\lambda)$ mit korrekten Ausgaben $\hat{y} = (\hat{y}_1...\hat{y}_\lambda)$ ist der durchschnittliche Fehler analog zu Formel 2.7 in Abbhängigkeit von den entsprechenden Ausgaben $y = (y_1...y_\lambda)$ folgender.

$$E(y, \hat{y}) = \frac{1}{\lambda} \sum_{1}^{\lambda} (y - \hat{y})^2 \tag{4.4}$$

Durch Negation wird aus dem Fehler die zu maximierende Fitnessfunktion.

$$f(y, \hat{y}) = -E(y, \hat{y}) \tag{4.5}$$

Außerdem berechnet jeder Prozess noch den Gradienten der Dichtefunktion für seine Parameter (Formel 4.3). Nach dem das für alle Prozesse geschehen ist, wird die Fisher Matrix wie in Formel 2.17 angegeben approximiert und dann invertiert. Daraufhin kann der natural gradient mit Formel 2.18 berechnet werden.

Schluessendlich wird für den eigentlichen Gradientenaufstieg entlang des natural gradients, dieser mit der Lernrate η multipliziert, die einen weiteren Parameter des Algorithmus darstellt. Das Ergebnis wird zu den aktuellen Parametern addiert. Insgesamt sieht der resultierende Algorithmus dann folgendermaßen aus.

Algorithmus 3 : Parallelisiertes NES für binäre neuronale Netze

Input : Anzahl der Gewichte des BNN d, Trainingseingaben $x_1...x_m$ mit Lösungen $\hat{y}_1...\hat{y}_m$, Lernrate η, Anzahl der Prozesse n, Anzahl Iterationen pro Prozess λ

initialisiere $\theta_i \leftarrow$ zufällig aus $[0, 1]$ für $i = 1...d$;
repeat
  for *Prozess* $k = 1...n$ **do**
    generiere Gewichte $w_k \sim \pi(\cdot|\theta)$;
    $e_k \leftarrow 0$;
    for $l = 1...\lambda$ **do**
      wähle Trainingseingabe x_r für zufälliges r aus $1...m$;
      $y_l \leftarrow$ Ausgabe des BNN für Eingabe x_r;
      $e_k \leftarrow e_k + (x_r - \hat{y}_r)^2$
    end
    $f(w_k) \leftarrow -\frac{1}{\lambda} e_k$;
    berechne $\nabla_\theta \log \pi(w_k|\theta)$;
  end
  $\nabla_\theta J \leftarrow \frac{1}{\lambda} \sum_{k=1}^{\lambda} \nabla_\theta \log \pi(w_k|\theta) \cdot f(w_k)$;
  $F \leftarrow \frac{1}{\lambda} \sum_{k=1}^{\lambda} \nabla_\theta \log \pi(w_k|\theta) \nabla_\theta \log \pi(w_k|\theta)^T$;
  $\theta \leftarrow \theta + \eta \cdot F^{-1} \nabla_\theta J$;
until *Stop-Kriterium erfüllt*;

Output : θ

4.3 Implementierung

Der Algorithmus wurde in C++ implementiert. Er nutzt Eigen (http://eigen.tuxfamily.org) für die Matrix- und Vektorberechnungen. Eigen ist eine Bibliothek für C++, die sehr effiziente Matrixoperationen- und Algorithmen implementiert. Es werden Matrizen und Vektoren mit beliebigen Größen unterstützt. Eigen ist hauptsächlich für die Nutzung auf CPUs optimiert und profitiert stark von speziellen Prozessorinstruktionen, die für die Berechnungen ausgenutzt werden können. Gerade für die in dieser Arbeit angestrebten Anwendungsfälle erscheint Eigen aufgrund dieser Eigenschaften sehr geeignet.

Für die Parallelisierung der Prozesse wird OpenMP (http://www.openmp.org) eingesetzt. OpenMP ist eine API für parallele Berechnungen, die mit den meisten üblichen Betriebssystemen und Rechnerarchitekturen kompatibel ist. Die Verwendung ist extrem simpel. Um die Durchläufe einer Schleife parallel ablaufen zu lassen muss lediglich eine einzige Präprozessoranweisung eingegfügt werden. Die Allokation von Threads und Verteilung auf die vorhandenen Prozessoren passiert automatisch. Die Anzahl der zu verwendenden

Threads kann einfach durch eine Umgebungsvariable angegeben werden. OpenMP ist in den meisten gängigen Compilern, und insbesondere dem hier verwendeten GCC, integriert.

Kapitel 5

Experimente

Der Algorithmus wurde zur Durchführung der Versuche in verschiedenen Konfigurationen auf einem Testsystem ausgeführt. Das Testsystem ist mit einem AMD Ryzen 7 1700X Prozessor mit 8 Rechenkernen (16 Threads) und einer Taktfrequenz von 3,8GHz ausgestattet. Außerdem stehen 16GB DDR4 RAM zur Verfügung. Zum Vergleich wurde für drei verschiedene Datensätze jeweils ein neuronales Netz sowohl mit dem in C++ implementierten Algorithmus, als auch mit einer entsprechenden Vergleichsimplementierung in Tensorflow getestet, die backpropagation verwendet. Das trainierte Netzwerk besitzt jeweils zwei versteckte Schichten mit 20 Neuronen.

In diesem Kapitel werden die verwendeten Datensätze beschrieben und die Versuchsergebnisse aufgelistet.

5.1 Datensätze

5.1.1 MAGIC Gamma Telescope

Major Atmospheric Gamma Imaging Cherenkov Telescopes (MAGIC) ist die Bezeichnung für zwei Imaging Atmospheric Cherenkov Teleskope, die sich auf La Palma befinden. Sie beobachten auf indirekte Weise die auf der Erde direkt nur schwer nachweisbare Gammastrahlung aus der Atmosphäre. Beobachtet werden daher die von der Strahlung erzeugten Teilchenschauer. Aus der Verteilung der Teilchen eines beobachteten Teilchenschauers lässt sich die Ursache für die Entstehung der Gammastrahlung ermitteln. Betrachtet werden dazu die Eigenschaften der für gewöhnlich elliptisch geformten Ansammlungen der Teilchen auf Aufnahmen.

Der MAGIC Gamma Telescope Datensatz besteht aus 19020 künstlich generierten Beispielen, welche das Auftreffen von durch atmosphärische Gammastrahlung erzeugten Teilchen auf einem solchen Teleskop simulieren. Die Attribute der Beispiele sind 10 verschiedene Eigenschaften der vom Tekeskop aufgenommenen Bilder. Dazu gehören beispielsweise die

Längen der Achsen der Ellipse, der Winkel zwischen der Hauptachse und dem Ursprung und ein Maß für die Anzahl der insgesamt erfassten Photonen.

Die Beispiele sind je nach Ursache für für die Entstehung des Teilchenschauers in zwei Klassen eingeteilt. Die eine Klasse enthält die Beispiele deren Teilchenschauer wirklich durch Gammastrahlung erzeugt wurden. Die andere Klasse enthält die restlichen durch Hadronen erzeugten Teilchenschauer.

Zur Durchführung der Versuche wurden 15000 Beispiele für das Training verwendet und die restlichen 4020 zum anschließenden Testen.

5.1.2 Covertype

Der Covertype Datensatz enthält insgesamt 581012 Beispiele für die Vorhersage der Art der Bodenbedeckung in verschiedenen Gebieten des Roosevelt National Forest in den USA. Jedes Beispiel repräsentiert eine Zelle mit einer Größe von 30 mal 30 Metern. Ein einzelnes Beispiel enthält 54 Merkmale aus Kartendaten für das jeweilige Gebiet und ist einer von 7 Arten der Bodenbedeckung zugeordnet. Die realen Werte für die Art der Bedeckung wurden vom US Forest Service und US Geological Survey erfasst. Die Merkmale stammen ausschließlich aus Kartendaten und umfassen unter Anderem die Höhe über dem Meeresspiegel, die Stärke des Abhangs, die Distanz zur nächsten Wasseroberfläche und zum nächsten Waldbrandgebiet, den Bergschatten zu verschiedenen Tageszeiten und die vorhandene Bodentypen.

Bei der Auswahl der Gebiete wurde darauf geachtet, dass der menschliche Einfluss möglichst gering ist. Die Art der Bodenbedeckung ist dadurch hauptsächlich ein Ergebnis von natürlichen Prozessen statt und nicht das von Maßnahmen zur Waldbewirtschaftung.

Für die Versuche wurde der Datensatz in 500000 Trainingsbeispiele und 81012 Testbeispiele aufgeteilt.

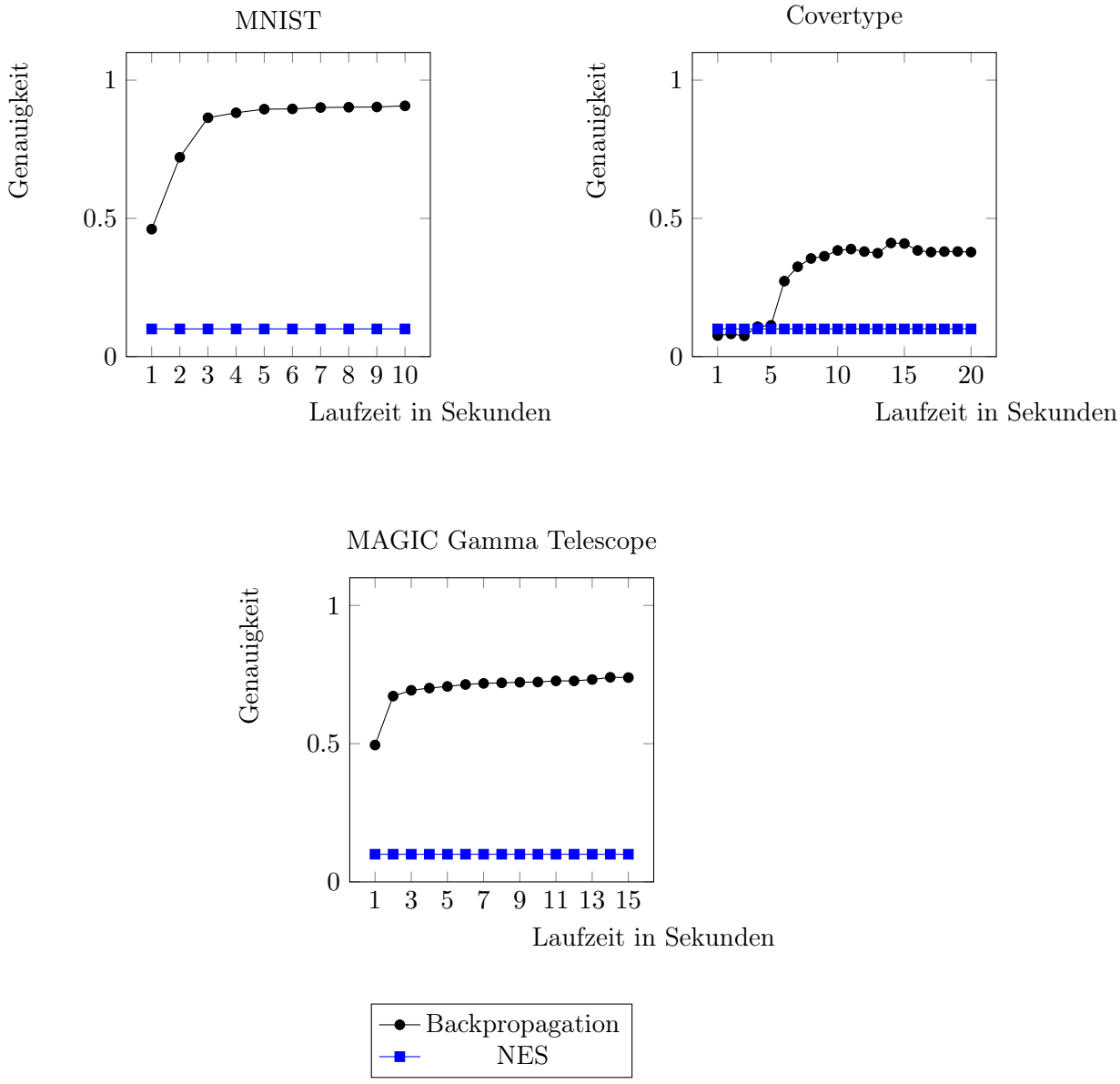
5.1.3 MNIST

Die *Modified National Institute of Standards and Technology* (MNIST) Datenbank beinhaltet schwarz-weiß Bilder von handgeschriebenen Ziffern. Die Bilder stammen je zur Hälfte aus zwei Datensätzen mit von Schülern geschriebenen Ziffern und solchen von Mitarbeitern des *United States Census Bureau*. Insgesamt wurden die Ziffern von ungefähr 250 verschiedenen Personen geschrieben.

Es gibt 60000 Trainingsbeispiele, sowie 10000 weitere Testbeispiele, die jeweils den Ziffern 0 bis 9 zugeordnet sind. Da die Bilder aus 28x28 Pixeln bestehen, hat ein jedes davon 784 Attribute.

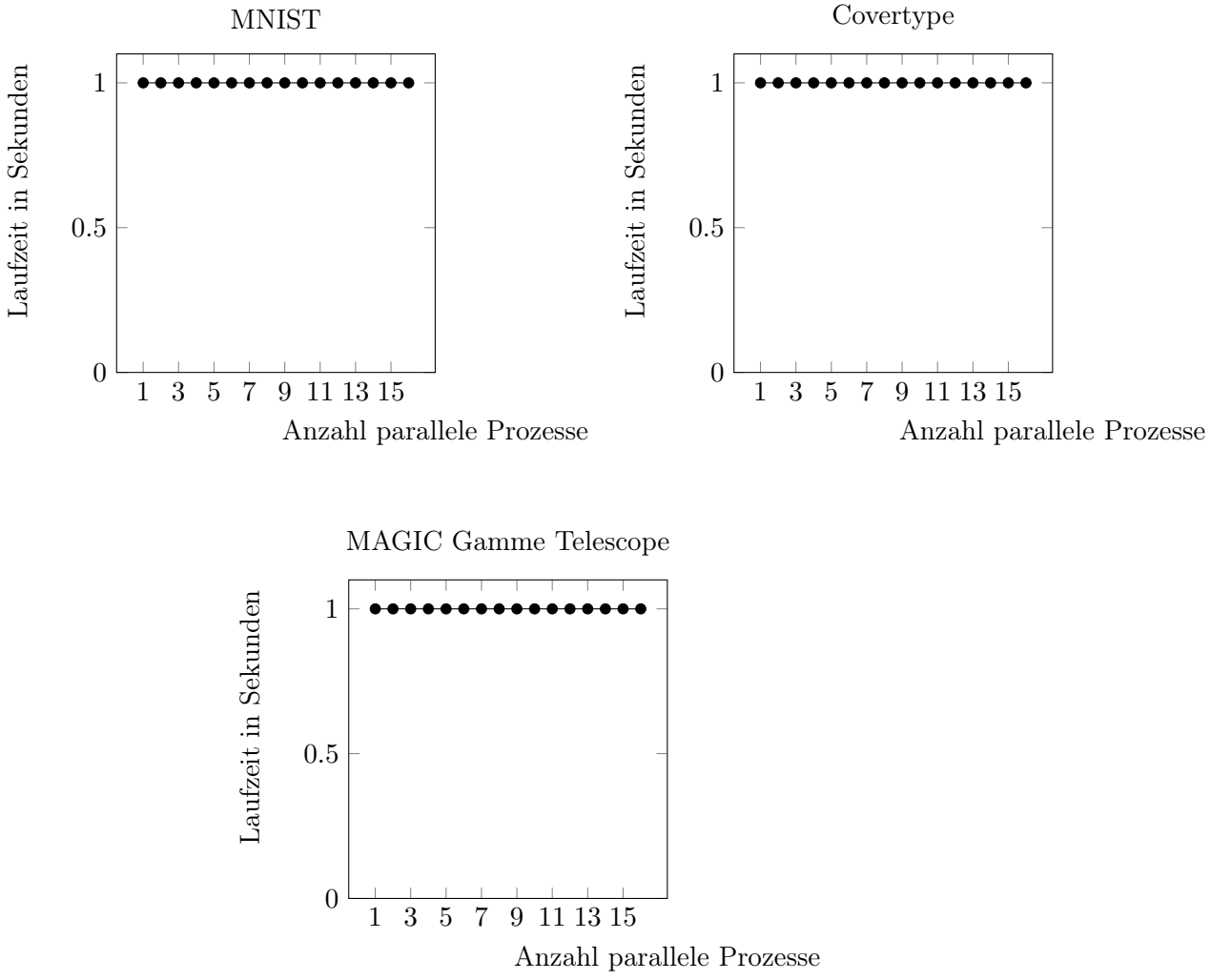
5.2 Vergleich der Genauigkeit

Dieses Experiment vergleicht die Genauigkeit der Ausgaben anhand der genannten Testbeispiele im Verlauf der Trainingszeit. Gezeigt sind für die drei Testdatensätze jeweils die Genauigkeiten der NES Implementierung auf BNN und die der Implementierung mit backpropagation. Das NES Programm nutzt hier bei der Optimierung 8 parallele Prozesse, die jeweils 1000 Trainingsbeispiele zu Ermittlung der Fitness evaluieren. Es wird die Hypothese getestet, dass aufgrund der Parallelisierung für die NES Implementierung ein schnellerer Anstieg der Genauigkeit im Zeitverlauf zu beobachten ist. Die maximal erreichte Genauigkeit sollte jedoch aufgrund der Verwendung von binären Gewichten etwas unter der der Vergleichsimplementierung liegen.



5.3 Test der Skalierbarkeit

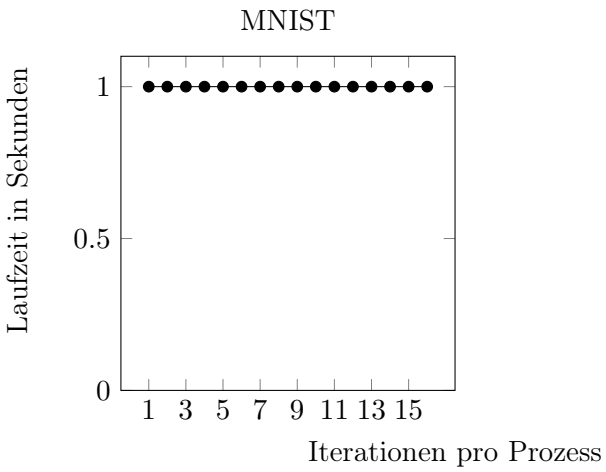
In diesem Experiment wird die Skalierbarkeit der NES Implementierung mit der Anzahl der genutzten parallelen Prozesse verglichen. Gemessen wird jeweils die benötigte Zeit, um die während des vorherigen Experiments ermittelte maximale Genauigkeit zu erreichen. Die hier zu testende Hypothese ist, dass die benötigte Zeit linear mit der Anzahl der parallelen Prozess abnimmt. Für parallelisierte ES wurde dies bereits in [18] gezeigt. Jeder Prozess evaluiert hier für die Fitnessberechnung wieder 1000 zufällige Trainingsbeispiele.

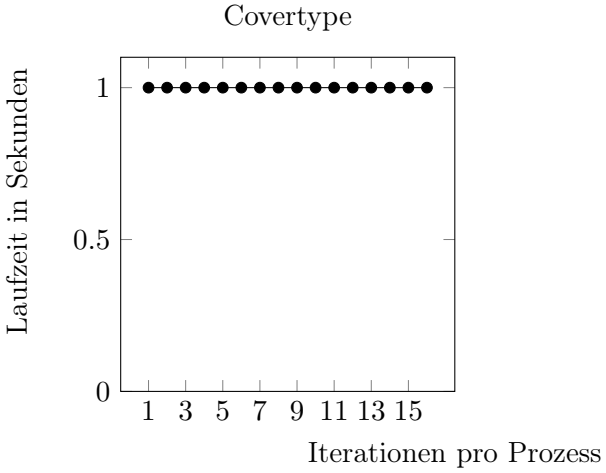


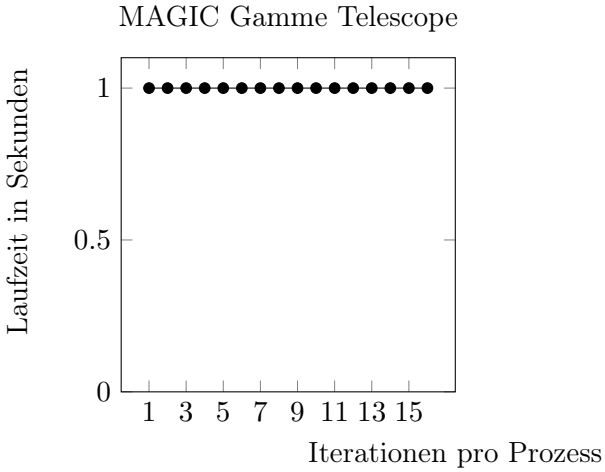
5.4 Test der Anzahl Iterationen pro Prozess

Die Anzahl der Iterationen pro Prozess ist ein Parameter des Algorithmus, der die Genauigkeit und die Laufzeit beeinflussen kann. In jeder Iteration wird ein zufälliges Testbeispiel evaluiert und in die Fehlerberechnung mit einbezogen. Mit der Anzahl der Iterationen steigt also auch die Genauigkeit der durch den jeweiligen Prozess geschätzten Fitness und

damit auch die des approximierten Gradienten. Getestet werden soll die Hypothese, dass es aufgrund des parallelen Ablaufs der einzelnen Prozesse bei einer Erhöhung der Anzahl der Iterationen pro Prozess nur zu einem linearen Zuwachs der benötigten Zeit kommt. Zu sehen ist die Zeit bis zum Erreichen der bereits ermittelten maximalen Genauigkeit in Abhängigkeit von der Anzahl der Iterationen pro Prozess. Die Anzahl der parallelen Prozesse wurde dafür auf 8 festgelegt.







Kapitel 6

Fazit

Aufgrund von Fehlern in der Implementierung der NES für BNN konnten die Experimente leider nicht wie geplant durchgeführt werden. Die beschriebenen Versuche und Hypothesen sollten daher in Zukunft mit einer funktionierenden Implementierung getestet werden.

Es könnten dann auch die Möglichkeiten, die sich in Bezug auf die angestrebte Verwendung von NES bzw. mit NES trainierten BNN auf kleineren und mobilen Geräten ergeben, evaluiert werden. So sollten weitere Experimente zur Verwendung und Training von BNN mit NES direkt auf verschiedenen Anwendergeräten durchgeführt werden.

Wichtig ist dabei vor allem der Speicherbedarf des BNN. Wie schon in Abbildung 2.3 zu sehen, ist dieser im Vergleich zur verfügbaren Speicherkapazität auf vielen modernen Geräten noch sehr gering. Deshalb könnte unter Umständen sogar eine Vergrößerung der Genauigkeit der Gewichte in Betracht gezogen werden, um qualitativ bessere Ergebnisse zu erzielen.

Wenn zusätzlich ausreichend Rechenleistung zur Verfügung steht, besteht außerdem die Möglichkeit eines Trainings direkt auf einem solchen Gerät. Das würde beispielsweise die lokale Verarbeitung der verschiedenen, auf Smartphones verfügbaren, Sensordaten ermöglichen. Außerdem könnten BNN als direkte Reaktion auf Nutzereingaben für verschiedenste Probleme optimiert werden. Sollte ein lokales Training nicht praktikabel sein, ließe sich dieses immer noch im Vorhinein auf einem leistungsstärkeren Gerät ausführen und das fertige BNN auf das eigentliche Gerät übertragen.

Abschließend erscheint es im Zuge der zunehmenden Digitalisierung sinnvoll, in Zukunft auch weitere bereits erforschte Ansätze für die effiziente Anwendung und Optimierung von Modellen des maschinellen Lernens auf elektronischen Geräten des Alltags in Betracht zu ziehen. Um die Entwicklung und Integration von intelligenten Geräten weiter zu verbessern, sollte ferner mit Nachdruck in die Erforschung weiterer effizienter Verfahren für die Verwendung von Methoden des maschinellen Lernens auf diesen Geräten investiert werden.

Anhang A

Anhang

Abbildungsverzeichnis

Liste der Algorithmen

Literaturverzeichnis

[1] Beyer, Hans-Georg und Hans-Paul Schwefel: *Evolution strategies–A comprehensive introduction.* Natural computing, 1(1):3–52, 2002.

[2] Cortes, Corinna und Vladimir Vapnik: *Support-vector networks.* Machine learning, 20(3):273–297, 1995.

[3] Courbariaux, Matthieu, Yoshua Bengio und Jean-Pierre David: *BinaryConnect: Training Deep Neural Networks with binary weights during propagations.* In: *Advances in Neural Information Processing Systems 28*, Seiten 3123–3131. Curran Associates, 2015.

[4] Esteva, Andre, Brett Kuprel, Roberto Novoa, Justin Ko, Susan M Swetter, Helen M Blau und Sebastian Thrun: *Dermatologist-level classification of skin cancer with deep neural networks.* Nature, 542:115–118, 2017.

[5] Goodfellow, Ian J., David Warde-Farley, Mehdi Mirza, Aaron Courville und Yoshua Bengio: *Maxout Networks.* ArXiv e-prints, 2013.

[6] Haykin, Simon: *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1999.

[7] He, Kaiming, Xiangyu Zhang, Shaoqing Ren und Jian Sun: *Deep residual learning for image recognition.* In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, Seiten 770–778, 2016.

[8] Hecht-Nielsen, Robert: *Theory of the Backpropagation Neural Network.* In: *Neural Networks for Perception*, Seiten 65–93. Academic Press, 1992.

[9] Krizhevsky, Alex, Ilya Sutskever und Geoffrey E Hinton: *Imagenet classification with deep convolutional neural networks.* In: *Advances in neural information processing systems*, Seiten 1097–1105, 2012.

[10] Kullback, Solomon und Richard A Leibler: *On information and sufficiency.* The annals of mathematical statistics, 22(1):79–86, 1951.

[11] Langley, Pat, Wayne Iba, Kevin Thompson et al.: *An analysis of Bayesian classifiers.* In: *Aaai*, Band 90, Seiten 223–228, 1992.

[12] Lee, Chen-Yu, Saining Xie, Patrick W. Gallagher, Zhengyou Zhang und Zhuowen Tu: *Deeply-Supervised Nets.* 2014.

[13] Mellit, Adel und Alessandro Massi Pavan: *A 24-h forecast of solar irradiance using artificial neural network: Application for performance prediction of a grid-connected PV plant at Trieste, Italy.* Solar Energy, 84(5):807–821, 2010.

[14] Nasrabadi, Nasser M.: *Pattern recognition and machine learning.* Journal of electronic imaging, 16(4):049901, 2007.

[15] Pascanu, Razvan und Yoshua Bengio: *Revisiting natural gradient for deep networks.* ArXiv e-prints, 2013.

[16] Rastegari, Mohammad, Vicente Ordonez, Joseph Redmon und Ali Farhadi: *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks.* In: *Computer Vision – ECCV 2016*, Seiten 525–542. Springer International Publishing, 2016.

[17] Rowley, Henry A., Shumeet Baluja und Takeo Kanade: *Neural network-based face detection.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 20(1):23–38, 1998.

[18] Salimans, Tim, Jonathan Ho, Xi Chen und Ilya Sutskever: *Evolution Strategies as a Scalable Alternative to Reinforcement Learning.* CoRR, abs/1703.03864, 2017.

[19] Simonyan, Karen und Andrew Zisserman: *Very deep convolutional networks for large-scale image recognition.* ArXiv e-prints, 2014.

[20] Wierstra, Daan, Tom Schaul, Tobias Glasmachers, Yi Sun und Jürgen Schmidhuber: *Natural Evolution Strategies.* ArXiv e-prints, 2011.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 20. März 2018

Jan Kemming