Machbarkeitsstudie des L-BFGS Verfahrens für das Training von Deep Learning Problemen

Bachelorarbeit

vorgelegt von **Ilona Shonia**

im Rahmen des Bachelor-Studiengangs Wirtschaftsmathematik

am 17. Oktober 2018

am Mathematischen Institut der

Universität zu Köln

Erstgutachter: Prof. Dr. Axel Klawonn

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Künstliche Intelligenz (KI) ist eines der blühenden Gebiete in vielen praktischen Anwendungen und in der Forschung. Viele Probleme, die gut mit mathematischen Formeln darstellbar sind, sind schwer für das menschlichen Gehirn zu lösen, aber relativ einfach für einen Computer. Im Gegensatz dazu gibt es Probleme, die für Menschen einfach intuitiv zu lösen sind, aber schwer formal beschreibbar sind. Dazu gehört z.B. die Erkennung von gesprochenen Wörtern oder die Erkennung von Mustern in Bildern, welche wahre Herausforderung für Computer sind [6, S. 1]. Maschinelles Lernen (ML) ist ein Teilgebiet der KI, das sich mit Lösungsmöglichkeiten von genau solchen intuitiven Problemen beschäftigt. Der Ansatz dabei ist aus Erfahrung zu lernen, indem Wissen aus Rohdaten generiert wird. Die erfolgreichen ML Algorithmen sind hierarchisch aufgebaut und verwenden sehr viele Abstraktionsschichten, so dass man auch häufig von Deep Learning redet.

Auch wenn die meisten der heute angewendeten Deep Learning Algorithmen bereits in den 1980er Jahren bekannt waren [6, S. 22], war die verfügbare Hardware für die meisten wissenschaftlichen Anwendungen lange Zeit nicht schnell genug. Mit der Zunahme der zur Verfügung stehenden Trainingsdaten und dem Durchbruch der Rechenkapazität moderner Computer durch GPUs, erleben Deep Learning Methoden, wie die künstlichen neuronalen Netze, in vielen praktischen Anwendungsfällen eine unglaubliche Renaissance.

Im Trainingsprozess von neuronalen Netzen werden gradientbasierte Optimierungsalgorithmen angewendet. Dabei werden, aufgrund der schieren Größe der Datensätze, die Gradienten für gewöhnlich aber nicht vom ganzen Zielfunktional gebildet, sondern immer nur von einem zufällig gewählten Teil der Summanden. Dies gibt dem berechneten Gradienten einen stochastischen Charakter und die entsprechende Modifikation des einfachsten Gradientenabstiegsverfahrens (gradient descent, Cauchy, 1847) wird entsprechend Stochastic Gradient Descent (SGD) genannt. Abwandlungen dieses Verfahrens, wie z.B. der auch in dieser Arbeit untersuchte Adam Algorithmus [8], erfreuen sich große Beliebtheit bei ML-Anwendern.

Trotz der Beliebtheit des einfachen Gradientenabstiegsverfahrens, stellt sich die Frage, ob nicht

anspruchsvollere Verfahren noch bessere Ergebnisse liefern können. So ist bekannt aus der Optimierungstheorie, dass das einfache Gradientenabstiegsverfahren nur eine lineare Konvergenzgeschwindigkeit besitzt [13, Theorem 3.3], und dass für superlineare Konvergenz die Suchrichtung mit Hilfe von Informationen aus der inversen Hesse Matrix angepasst werden muss [13, Theorem 3.6]. Im Falle des Newton-Verfahrens, wo die volle inverse Hesse Matrix verwendet wird, erhält man sogar quadratische Konvergenz [13, Theorem 3.7]. Jedoch ist das Berechnen und Invertieren der Hesse Matrix für viele Probleme sehr rechenaufwendig. Es gibt aber Methoden aus der Quasi-Newton Klasse, die versuchen die Hesse Matrix bzw. ihre Inverse durch die Änderungen im Gradienten von Schritt zu Schritt zu approximieren, um den Rechen- und Speicheraufwand zu minimieren. Eine solche Methode ist das L-BFGS Verfahren [13, S. 226]. Im Hinblick auf die stochastische Auswertung der Gradienten stellt sich dabei die Frage, inwiefern die stochastischen Fluktuationen im Gradienten überhaupt eine sinnvolle Approximation der inversen Hesse Matrix zulassen, und damit, ob sich das L-BFGS Verfahren überhaupt für das Training von Deep Learning Problemen eignet?

Um diese Frage beantworten zu können, wurde für diese Arbeit eine Recherche von aktuellen Forschungsergebnissen über die Anwendbarkeit der L-BFGS Methode für Deep Learning Probleme durchgeführt. Es wurden zwei für stochastische Fälle modifizierte Varianten des Verfahrens - das robuste Multi-Batch L-BFGS [1] und das Progressive-Batching L-BFGS Verfahren [2] - ausgesucht und die numerischen Ergebnisse untersucht.

Als Anwendungsproblem wurde die Klassifizierung von Bildern auf Basis des MNIST-Datensatzes der handgeschriebenen Ziffern gewählt. Der MNIST-Datensatz wird in der Forschung an neuronalen Netzen oft eingesetzt, um bei neuen Ideen zu prüfen, ob sie besser abschneiden, als bereits bestehende Verfahren.

1 Grundlagen

1.1 Maschinelles Lernen

Das Ziel des maschinellen Lernens ist es, Wissen aus Erfahrung zu generieren. Diese Generierung des Wissens kann erfolgen, indem Lernalgorithmen ein komplexes Modell aus gegebenen Beispielen entwickeln. Das Modell, und damit die automatisch erworbene Wissensrepräsentation, kann anschließend auf neue, potenziell unbekannte Daten derselben Art angewendet werden.
Maschinelles Lernen ist im Wesentlichen eine Form der angewandten Statistik mit verstärktem Schwerpunkt auf der Nutzung von Computern zur statistischen Abschätzung komplizierter Funktionen und abgeschwächten Fokus im Bereich von Testen von Konfidenzintervallen um diese Funktionen [6, S. 95].
Eine prägnante Definition eines maschinellen Lernalgorithmus wurde in [10] von T. Mitchell vorgestellt: "Ein Computerprogramm lernt aus der Erfahrung $\boldsymbol{E}$ hinsichtlich eine Klasse von Aufgaben $\boldsymbol{T}$ und Leistungsmaß $\boldsymbol{P}$, wenn seine Leistung bei Aufgaben in $\boldsymbol{T}$, gemessen durch $\boldsymbol{P}$, sich mit der Erfahrung $\boldsymbol{E}$ verbessert."
Um diese Definition besser zu verstehen, bringen wir ein leichtes Beispiel über ein Programm, dass das Brettspiel "Dame"lernt. Hierbei ist der Prozess des Spielens die Aufgabe $\boldsymbol{T}$, die Prozentzahl der gewonnen Spiele das Leistungsmaß $\boldsymbol{P}$ und die Spiele, die das Programm gegen sich selbst gespielt hat, die Erfahrung $\boldsymbol{E}$.
Immer, wenn Prozesse zu kompliziert sind, um sie analytisch zu beschreiben, aber genügend viele Beispiele – etwa Sensordaten, Bilder oder Texte – verfügbar sind, bietet sich maschinelles Lernen an. Die Kollektion von Beispielen wird als Datensatz bezeichnet und die einzelne Beispiele als Datenpunkte.
Maschinelle Lernalgorithmen bestehen aus zwei Phasen:

- Training- oder Lernphase: Gegeben sei ein Trainingsdatensatz und wir suchen eine Abbildung zwischen Input-Output Paare mithilfe der Optimierung.

- Testphase: Die gelernte Abbildung wird auf einen neuen Datensatz - den Testdatensatz -

angewendet.

Die meisten maschinelle Lernalgorithmen können in zwei Kategorien eingeteilt werden: das überwachte Lernen (*supervised learning*) und das unüberwachte Lernen (*unsupervised learning*) [6, S. 101]. Auf andere Arten des Lernens wird hier nicht eingegangen.

Das unüberwachte Lernen

Im unüberwachten Lernen erhält der Lernalgorithmus lediglich die Input-Werte. Das Ziel ist die Struktur der Daten zu lernen. Es geht also um die Entdeckung der Information, wobei die Grundwahrheit (*ground truth*) nicht vorhanden ist.

Das überwachte Lernen

Beim überwachten Lernen handelt es sich um ein vom Menschen gesteuertes Lernen. Dabei wird ein System mit Input-Output Paaren trainiert und so ein Wissen über diese Daten gewonnen. Dieses Wissen wird dann auf bisher unbekannte Daten angewendet, um so Voraussagen über zukünftige Daten zu treffen. Insbesondere soll eine mathematische Funktion gelernt werden, die einem gegebenen Input ein Output mit hoher Wahrscheinlichkeit zuordnet [12, S. 63].
Das überwachte Lernen teilt sich in zwei Unterkategorien: Klassifizierung und Regression. Einteilung der Daten in bestimmten Klassen wird als Klassifizierung bezeichnet. Bei der Regression sind die Ausgabewerte, im Gegensatz zur Klassifizierung, kontinuierlich.

Diese Arbeit beschäftigt sich mit dem Problem der Bildklassifizierung, das als eine Aufgabe der Einteilung von Bildern in die vordefinierten Klassen definiert werden kann. Da dieses Problem unter dem überwachten Lernen fällt, konzentrieren wir uns in den kommenden Abschnitten nur auf das überwachte Lernen und führen dafür relevante Begriffe und ein mathematisches Modell ein.

1.2 Ein maschinelles Lernmodell

Im überwachten Lernen haben wir das folgende mathematische Modell: Gegeben sei ein Input x aus dem Raum $\mathcal{X}$ der Trainingsdaten und ein Output y aus dem Raum $\mathcal{Y}$ der dazu korrespondierenden Labels. Die Paare $\{(x_i, y_i)\},\ i = 1, ..., n$ stellen einen Datensatz aus n Datenpunkten dar. Das Ziel des Lernens ist es, für jeden neuen Input x eine Vorhersage für den entsprechenden Output y zu treffen. Im einfachsten Fall kann dies durch direkte Konstruktion einer geeigneten Funktion

$h(x) = y$ erfolgen [4, S. 2]. In der Machine Learning Literatur wird eine solche Funktion als Hypothesefunktion bezeichnet. Da jedoch in typischen ML-Aufgaben recht wenig über die Hypothesefunktion bekannt ist, versucht man diese aus einer möglichst großen parametrisierten Funktionenklasse zu wählen. Um diese Parametrisierung zu kennzeichnen, bezeichnen wir die Hypothesefunktion als $h_\vartheta(x)$, wobei ϑ die Modellparameter sind.
Die durch die initiale Hypothesefunktion $h_\vartheta(x)$ gemachten Vorhersagen stimmen zunächst natürlich nicht mit der Grundwahrheit überein. Vielmehr versucht man die Parameter ϑ der Hypothesefunktion solange zu optimieren, bis $h_\vartheta(x) \approx y$ gilt. Grundvoraussetzung für die Optimierung ist eine Fehlerfunktion ℓ, die den Unterschied zwischen der Grundwahrheit y und der von dem Modell gemachten Vorhersage $h_\vartheta(x)$ misst.
Eine einfache solche Fehlerfunktion ist die mittlere quadratische Abweichung MSE (Mean Squared Error), gegeben durch

$$\ell = \frac{1}{n} \sum_{i=1}^{n} \left(h_\vartheta(x_i) - y_i \right)^2 . \tag{1.1}$$

Die Wahl der Fehlerfunktion hängt jedoch stark von dem gegebenen Problem und der gewählten Lösungsmethode ab.

1.3 Künstliche neuronale Netze

Künstliche neuronale Netze (kNN) liefern eine allgemeine, praktische und robuste Methode zur Approximation von reellwertigen, diskreten und vektorwertigen Funktionen. Das Lernen mit künstlichen neuronalen Netzen ist fehlertolerant in der Trainingsdatensatz. Für bestimmte Arten von Problemen, wie z.B. das Lernen der Interpretation komplexer realer Sensordaten, gehörten künstliche neuronale Netze zu den effektivsten bekannten Lernmethoden in den 90-er Jahren. So hat sich beispielsweise der Backpropagation-Algorithmus bei vielen praktischen Problemen wie der Erkennung von handschriftlichen Ziffern (LeCun et al. 1989), der Erkennung von gesprochenen Wörtern (Lang et al. 1990) und der Gesichtskennung (Cottrell 1990) überraschend bewährt [10, S. 81].
Die Darstellung im unteren Abschnitt folgt dem Kapitel 4 und 5 aus [14].

Struktur der neuronalen Netzen

Nach der in [14, S. 38] gegebener Definition ist kNN ein gerichteter Graph $G = (U, C)$, dessen Knoten $u \in U$ als Neuronen und Kanten $c \in C$ als Verbindungen bezeichnet werden. Die Menge der Kanten U besteht ihrerseits aus den disjunkten Mengen der Input-Neuronen U_{in},

Output-Neuronen U_{out} und verdeckten Neuronen U_{hidden}. Input-Neuronen sind dabei solche, die nur ausgehende Kanten besitzen und Output-Neuronen solche mit nur eingehenden Kanten. Die verdeckten Neuronen bilden den Rest.
Jede Verbindung $(u, v) \in C$ ist mit einem Parameter $\vartheta_{uv} \in \mathbb{R}$ gewichtet. Jedem Neuron $u \in U$ sind zwei Werte zugeordnet: der Netzwerkinput in_u und der Netzwerkoutput out_u, wobei

$$in_u = \sum_{v \in U} \vartheta_{vu} out_v.$$

Jedes Neuron $u \in U$ besitzt einen Bias-Wert b_u und eine Aktivierungsfunktion f^u_{act}, die aus Bias- und Input-Werten den Output out_u berechnet. Häufig ist die Aktivierungsfunktion für alle Neuronen gleich, so dass der Index u entfällt.
Falls der zum Netzwerk gehörender Graph azyklisch ist, d.h. er enthält keine gerichtete Kreise und keine der Neuronen ist mit sich selbst verbunden, haben wir ein feedforward-Netz, ansonsten ein rekurrentes Netz. Wir werden in dieser Arbeit nur feedforward-Netze behandeln.

Multilayer-Perzeptronen

Eine der bekanntesten Arten von kNN sind die multilayer-Perzeptronen (MLPs). Sie besitzen eine strikte Struktur und bestehen aus Input-, Output- und eventuell mehreren verdeckten Schichten (*layer*). In der Input-Schicht befinden sich alle Input-Neuronen $U_{in} = U_0$ und in der Output-Schicht alle Output-Neuronen $U_{out} = U_N$. Die verdeckten Neuronen sind entsprechend der Schichten in $U_1, U_2, \ldots, U_{N-1}$ eingeteilt.
Der Netzoutput von jeder Schicht verdeckter oder Output-Neuronen berechnet sich als eine mit f_{act} modifizierte gewichtete Summe der Ausgänge eingegangener Neuronen plus Bias b_u d.h. für ein Neuron u mit $u \in U_{hidden}$ oder $u \in U_{out}$ gilt:

$$out_u = f_{act}\left(\sum_{v \in V_u} \vartheta_{vu} out_v + b_u\right),$$

wobei V_u die Menge aller Vorgänger von u ist, d.h. in V_u befinden sich alle Neuronen aus der Schicht davor, die mit u verbunden sind.
Die strikte Schichtstruktur von MLPs erlaubt uns die Netzstruktur in Matrixschreibweise darzustellen. Seien $U_i = \{v_1, \ldots, v_m\}$ und $U_{i+1} = \{u_1, \ldots, u_p\}$ zwei aufeinanderfolgende Schichten

und $\Theta^{(i)} \in \mathbb{R}^{p\times m}$ die Matrix aus den entsprechenden Gewichten, erweitert um die Bias-Werte

$$\Theta^{(i)} = \begin{pmatrix} \vartheta_{u_1v_1} & \dots & \vartheta_{u_1v_m} & b_{u_1} \\ \vartheta_{u_2v_1} & \dots & \vartheta_{u_2v_m} & b_{u_2} \\ \vdots & \vdots & \vdots & \vdots \\ \vartheta_{u_pv_1} & \dots & \vartheta_{u_pv_m} & b_{u_p} \end{pmatrix}.$$

Es wird $\vartheta_{u_jv_k} = 0$ gesetzt, falls es keine Verbindung zwischen den Neuronen u_j und v_k existiert. Diese Schreibweise erlaubt uns den Netzoutput der Schicht U_{i+1} als

$$out_{U_{i+1}} = (out_{v_1}, \dots, out_{v_m})^T = f_{act}\left(\Theta^{(i)}\begin{pmatrix} out_{U_i} \\ 1 \end{pmatrix}\right)$$

darzustellen. Die Aktivierungsfunktion f_{act} wirkt dabei Elementweise auf die Einträge.
Die Hypothesefunktion ist für mehrere Output-Neuronen vektorwertig und gegeben durch

$$h_\vartheta = out_{U_N}.$$

Im Hinblick auf die Optimierung, sei noch angemerkt, dass der Index ϑ hier auch die Abhängigkeit der Hypothesefunktion von den Bias-Parametern miteinbezieht.

Aktivierungsfunktion

Ein typisches Beispiel für eine Aktivierungsfunktion ist die Sigmoid-Funktion

$$f_{act}(x) = \frac{1}{1+\mathrm{e}^{-x}}, \tag{1.2}$$

die in dieser Arbeit auch ausschließlich betrachtet wird.

Fehlerfunktion

Simard *et. al.* haben in ihrem Paper [16] festgestellt, das für das Training der Klassifizierungsprobleme mit neuronalen Netzen als Fehlerfunktion die Kreuzentropie zu schnelleren Training und verbesserten Generalisierung führt als die MSE (1.1). Die Kreuzentropie für eine Klasse ist gegeben durch

$$\ell = -\frac{1}{n}\sum_{i=1}^{n}\left\{y_i \ln h_\vartheta(x_i) + (1-y_i)\ln\left(1-h_\vartheta(x_i)\right)\right\}, \tag{1.3}$$

wobei $y_i \in \{0,1\}$, also x_i ist Teil der Klasse oder nicht, angenommen werden.
Für eine vektorwertige Hypothesefunktion für mehrere Klassen ist die Kreuzentropie gegeben durch

$$\ell = -\frac{1}{n}\sum_{j}\sum_{i=1}^{n}\left\{y_{i,j} \ln h_{\vartheta,j}(x_i) + (1-y_{i,j})\ln\left(1-h_{\vartheta,j}(x_i)\right)\right\}.$$

Training

Im Trainingsprozess der neuronalen Netzen wird versucht die Parameter ϑ so anzupassen, dass die geeignet gewählte Fehlerfunktion minimiert wird. Man versucht also die Zielfunktion $J(\vartheta) = \ell$ durch geeignete Wahl der Parameter ϑ zu optimieren. Dies geschieht unter Anwendung vom Backpropagation-Algorithmus und gradientenbasierter Optimierungsmethoden, die wir im kommenden Abschnitt diskutieren werden. Dazu sei an dieser Stelle darauf hingewiesen, dass die Zielfunktion $J(\vartheta)$ beliebig oft stetig differenzierbar ist. Im Hinblick auf die stochastische Auswertung von Gradienten definieren wir außerdem $J_i(\vartheta)$ als den Summanden der Zielfunktion, der nur von Datenpunkt (x_i, y_i) anhängt. Es gilt also

$$J(\vartheta) = \sum_{i=1}^{n} J_i(\vartheta).$$

2 Nichtlineare Optimierung

Im Falle von neuronalen Netzen ist die Zielfunktion, die wir optimieren wollen, nicht linear und i.A. auch nicht konvex. Eine Klasse von Algorithmen für die Optimierung einer nichtlinearen Funktion sind die iterativen Abstiegsmethoden. Diese fangen mit der initialen Schätzung ϑ_0 des Optimums an und generieren eine Folge von verbesserten Schätzungen, bis die hoffentlich optimale Lösung erreicht wird. Generiert wird also eine Folge $\{\vartheta_k\},\ k \geq 0$, mit der Eigenschaft

$$J(\vartheta_{k+1}) < J(\vartheta_k), \tag{2.1}$$

wobei $J(\vartheta)$ die durch Modellparameter $\vartheta \in \mathbb{R}^d$ parametrisierte und zu optimierende Zielfunktion ist. Die verwendete Strategie, um von einer Iterierten zu einer anderen zu wechseln, unterscheidet die Methoden voneinander. Die meisten Algorithmen verwenden den Zielfunktionswert, Nebenbedingungen und eventuell die erste oder zweite Ableitung der Zielfunkion. Manche Methoden akkumulieren Information aus früheren Iterationen, während andere nur lokale Informationen aus dem aktuellen Punkt benutzen. Im folgenden stellen wir einige diesen Methoden vor.

2.1 Gradientenbasierte Optimierung

Die Abstiegsbedingung (2.1) alleine reicht nicht aus, um garantieren zu können, dass die Iterationsfolge gegen ein lokales Minimum konvergiert. Hierfür sind stärkere Bedingungen erforderlich. Sobald sich die Iterierte ϑ_k in einer solchen Umgebung befindet, liefern die Abstiegsmethoden normalerweise ein schnell konvergentes lokales Verfahren [11, S. 3].
Die gradientenbasierte Optimierung ist in vielen Bereichen der Forschung und Ingenieurwissenschaften von zentraler praktischer Bedeutung. Wenn die Zielfunktion differenzierbar ist, sind die gradientenbasierte Optimierungsalgorithmen effiziente Methoden, da die Berechnungen von den partiellen Ableitungen erster Ordnung bezüglich aller Veränderlichen von der gleichen Rechenkomplexität ist, wie die Auswertung der Zielfunktion selbst [8, S. 1]. Daher ist das Gradientenverfahren (oder das Verfahren des steilsten Abstiegs) die einfachste Methode zur schrittweisen Minimierung einer Zielfunktion und mit Abstand der am häufigsten angewandte Algorithmus zur Optimierung

neuronaler Netze.

Die Iterationsvorschrift für das Gradientenverfahren lautet

$$\vartheta_{k+1} = \vartheta_k - \alpha \nabla_{\vartheta_k} J(\vartheta_k), \tag{2.2}$$

d.h. wir aktualisieren die Parameter in die entgegengesetzte Richtung der Steigung. Der Parameter α gibt die Größe der Schritte an, die wir nehmen, um ein (lokales) Minimum zu erreichen und wird im maschinellen Lernen als Lernrate bezeichnet.

Es gibt drei Varianten des Gradientenverfahrens, die sich darin unterscheiden, wie viele Daten wir zur Berechnung der Steigung der Zielfunktion benutzen: das (Batch-)Gradientenverfahren[1], das stochastische Gradientenverfahren und das Mini-Batch Gradientenverfahren. Mit der Wahl der Größe der Datenmenge machen wir einen Kompromiss (*trade-off*) zwischen der Genauigkeit des Gradienten und damit der Parameteraktualisierung, und der Laufzeit des Aktualisierungsprozesses. Im unteren Abschnitt werden diese drei Verfahren vorgestellt.

2.1.1 Das (Batch-)Gradientenverfahren

Im (Batch-)Gradientenverfahren wird in jeder Iteration der Gradient des gesamten Datensatzes ausgewertet, d.h. in jedem Schritt gilt es

$$\vartheta_{k+1} = \vartheta_k - \alpha \nabla_{\vartheta_k} J(\vartheta_k; x, y) \tag{2.3}$$

zu berechnen. Somit unterscheidet sich die Methode nicht von dem klassischen Gradientenverfahren.

2.1.2 Das stochastische Gradientenverfahren

Die Verwendung des kompletten Datensatzes in jeder Iteration verlangsamt den Optimierungsprozess drastisch. In vielen Anwendungsfällen verfügt man außerdem nicht über den gesamten Datensatz: die Daten können permanent einfließen oder der Datensatz überschreitet die Speicherkapazität des Rechners.

Um die oben genannten Fälle abzudecken, betrachtet man stochastische Zielfunktionen. So setzen sich beispielsweise viele Funktionen wie in Kapitel 1 erläutert, aus einer Summe von Teilfunktionen zusammen, die an verschiedenen Teilmengen von Daten ausgewertet werden. In den anderen Fällen wird Rauschen in die Zielfunktion künstlich eingebracht (z.B. Regularisierung mit Dropout), um die Modelle, die durch diese Funktionen dargestellt sind, besser generalisierbar

[1] Als Batch bezeichnet man ein Bündel aus Datenpunkten.

zu gestalten. In solchen Fällen, wo die Zielfunktion also stochastisch ist, benötigt man effiziente stochastische Optimierungstechniken [8, S. 1].
Eine einfache solche Methode ist das stochastische Gradientenverfahren (SGD). Im Gegensatz zu dem (Batch-)Gradientenverfahren nutzt SGD in jeder Iteration nur den Datenpunkt x_k mit zugehörigem Label y_k zur Aktualisierung der Parameter:

$$\vartheta_{k+1} = \vartheta_k - \alpha \nabla J_k(\vartheta_k). \tag{2.4}$$

Dies bringt Geschwindigkeitsvorteile und ist in den Fällen gut anwendbar, in denen Daten sozusagen „online“ einfließen, verursacht aber Probleme für die Konvergenz, denn es werden häufige Aktualisierungen mit einer hohen Varianz durchgeführt [15, S. 2].

2.1.3 Das Mini-Batch Gradientenverfahren

Das Mini-Batch Gradientenverfahren ist ein Kompromiss aus den beiden obigen Varianten und führt eine Aktualisierung für jedes Mini-Batch der Größe n aus dem Datensatz durch:

$$\vartheta_{k+1} = \vartheta_k - \alpha \nabla \sum_{i \in S_k} J_i(\vartheta_k), \tag{2.5}$$

wobei S_k das zur Iteration k gehörende Batch ist. Auf diese Weise kann man die Varianz in der Parameteraktualisierung reduzieren und somit eine stabilere Konvergenz erreichen. Außerdem macht dieser Ansatz es möglich, die state-of-the-art Deep-Learning Bibliotheken anzuwenden, welche Gradienten auf Mini-Batches der Datenmenge effizient berechnen.
In der Praxis ist es üblich, die Größe vom Batch S_k zwischen 50 und 256 zu wählen [15, S. 3]. Dieses Verfahren wird häufig verwendet für das Training von Neuronalen Netzen [15, S. 3]. Obwohl hier Mini-Batches statt einzelne Datenpunkte zur Parameteraktualisierung benutzt werden, wird diese Methode der Einfachheit halber als stochastisches Gradientenverfahren (SGD) bezeichnet. Das Mini-Batch Gradientenverfahren kann jedoch keine gute Konvergenz garantieren [15, S. 3] und führt zu einigen Herausforderungen, die im Paper [15] aufgelistet werden:

- Die Wahl einer passenden Lernrate kann zu einer schwierigen Aufgabe werden. Eine zu klein gewählte Lernrate verursacht zu langsame Konvergenz. Wird sie jedoch zu groß gewählt, kann dies die Konvergenz verhindern (Fluktuation um das Minimum) oder gar zur Divergenz des Verfahrens führen.

- Man kann die Lernrate variabel halten: die im maschinellen Lernen als *Annealing* bekannte Prozedur reduziert die Lernrate während des Trainings nach einem vorher definierten

Zeitplan. Eine andere Möglichkeit ist das Verkleinern der Lernrate, falls eine untere Grenze (Schwellwert) an Differenz in Zielfunktionswerten zwischen den Epochen unterschritten wird. In den beiden Fällen muss die Änderungsstrategie im Voraus definiert werden und kann sich daher nicht an die Eigenschaften eines Datensatzes anpassen.

- Falls die Daten dünnbesetzt sind und die Features sehr unterschiedliche Frequenzen besitzen, ist es empfehlenswert, alle Parameter nicht im gleichen Maß zu aktualisieren, sondern größere Aktualisierungen für selten auftretende Features durchzuführen.

- Eine weitere zentrale Herausforderung bei der Minimierung von hochgradig nicht-konvexen Fehlerfunktionen (typisch für neuronale Netze) ist es, dass sie sich in ihren zahlreichen suboptimalen lokalen Minima verfangen. Im Paper [5] wird argumentiert, dass die Schwierigkeit nicht von lokalen Minima, sondern von Sattelpunkten verursacht werden. Diese Sattelpunkte sind normalerweise von einem Plateau mit dem gleichen Fehler umgeben, was es für SGD schwierig macht aus dem Punkt zu entkommen, da die Gradienten in allen Richtungen nahe Nullpunkt sind.

Um die oben genannten Herausforderungen zu bewältigen, sind in Deep Learning einige optimierte Varianten von SGD entstanden. Eine davon, die in der Praxis gute Leistungen liefert, ist unter dem Namen Adam bekannt und wird im unteren Abschnitt detailliert beschrieben.

2.1.4 Adam - der Goldstandard in Deep Learning

Adam ist eine der effizienten SGD Methoden für das Trainieren von Deep Learning Problemen. Als gradientenbasierte Methode benötigt sie nur den Gradienten der Zielfunktion und somit ist die Anforderung an den Speicherplatz gering. Die Methode berechnet individuelle adaptive Schrittweiten für verschiedene Parameter aus Schätzungen der ersten und zweiten Momente des Gradienten - daher auch der Name Adam (adaptive Momentenschätzung). Adam bringt mehrere Vorteile, darunter:

- Die Größe der Parameteraktualisierung ist invariant zur Neuskalierung des Gradienten.

- Die Schrittweiten sind approximativ durch Schrittweiten-Hyperparameter begrenzt.

- Performt gut mit dünnbesetzten Gradienten.

- Setzt das Annealing auf natürliche Weise durch.

Sei $J(\vartheta)$ eine stochastische Funktion, die differenzierbar bezüglich der Parameter ϑ ist. Der Algorithmus aktualisiert exponentielle durchschnittliche Bewegung des Gradienten und des quadrierten Gradienten. Die Hyperparameter $\beta_1, \beta_2 \in [0, 1)$ kontrollieren deren exponentielle Zerfallsraten (*decay rates*). Die durchschnittlichen Bewegungen selber sind die Approximationen der ersten und zweiten Momenten des Gradienten. Der resultierende Algorithmus in Pseudocode ist unter Algorithmus 1 angegeben.

Algorithmus 1 Adam

Input: Zielfunktion $J(\vartheta)$, Startwert $\vartheta_0, \beta_1, \beta_2 \in [0, 1)$, Schrittweite α

Initialisiere: Vektoren für die ersten und zweiten Momente: $m_0 \leftarrow 0$, $v_0 \leftarrow 0$; Iterationszähler: $k \leftarrow 0$

while Abbruchbedingung nicht erfüllt **do**

$k \leftarrow k + 1$

$g_t \leftarrow \nabla J_{k-1}(\vartheta_{k-1})$

Aktualisiere verzerrte approximierte Momente an den ersten und den zweiten Momenten des Gradienten: $m_k \leftarrow \beta_1 m_{k-1} + (1 - \beta_1) g_t$, $v_k \leftarrow \beta_2 v_{k-1} + (1 - \beta_2) g_t^2$

Berechne korrigierte Approximationen an den Momenten an den ersten und den zweiten Momenten des Gradienten: $\hat{m}_k \leftarrow m_k / (1 - \beta_1^k)$, $\hat{v}_k \leftarrow v_k / (1 - \beta_2^k)$

$\vartheta_k \leftarrow \vartheta_{k-1} - \alpha \hat{m}_k / (\sqrt{\hat{v}_k} + \varepsilon)$

end while

Hierbei bezeichnet g_t^2 die elementweise Quadrierung des Gradienten, d.h. $g_t^2 = g_t \odot g_t$. Die im Paper [8] empfohlene Wahl der Parameter ist $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ und $\varepsilon = 10^{-8}$.

2.2 Methoden zweiter Ordnung

Im Gegensatz zu den Methoden erster Ordnung, die nur den Gradienten zur Optimierung benutzen (wie das Gradientenverfahren), verwenden die Methoden der zweiten Ordnung zweite Ableitungen, um mehr Information zu gewinnen und somit den Optimierungsprozess zu verbessern. Die bekannteste solche Methode ist das Newton-Verfahren.

2.2.1 Das Newton-Verfahren

Das Newton-Verfahren spielt eine zentrale Rolle in der Entwicklung der numerischen Techniken zur Optimierung. Einer der Gründe für seine hohe Bedeutung ist es, dass die Methode ganz

natürlich aus der Betrachtung einer Taylor-Approximation an der Funktion entsteht. Wegen seiner Einfachheit und seiner breiten Anwendbarkeit bleibt Newtons Methode ein wichtiges Werkzeug zur Lösung vieler Optimierungsprobleme. Tatsächlich können die meisten der gegenwärtigen praktischen Verfahren zur Optimierung (siehe z. B. das Quasi-Newton-Verfahren aus Abschnitt 2.2.2) als Variationen des Newton-Verfahrens betrachtet werden. Es ist daher wichtig, Newtons Methode als einen eigenständigen Algorithmus und als eine Schlüsseleinführung zu den neuesten Ideen in diesem Bereich zu verstehen.

Algorithmus 2 Allgemeine Newton-Methode für Optimierung

Input: Zielfunktion $J(\vartheta)$, Startwert ϑ_0

$k \leftarrow 0$

while Abbruchbedingung nicht erfüllt **do**

Berechne den Gradienten ∇J_k und die inverse Hesse-Matrix $H_k^{-1} = \nabla^2 J_k^{-1}$

Setze $\vartheta_{k+1} = \vartheta_k - H_k^{-1} \nabla J_k$

$k \leftarrow k + 1$

end while

Das Newton-Verfahren weist in einer Umgebung des lokalen Minimums quadratische Konvergenz auf, benötigt aber die Berechnung zweiter (partieller) Ableitungen, was sehr aufwendig ist. Deswegen wird es in numerischer Optimierung versucht die partiellen Ableitungen zu approximieren. Diese Vorgehensweise liefert eine neue Klasse der Algorithmen - die Quasi-Newton Methoden.

2.2.2 Das Quasi-Newton-Verfahren

Der Abschnitt 2.2.2 - 2.2.4 folgt der Darstellung der Kapitel 8 und 9 aus [13]. Die Quasi-Newton-Methoden vermeiden die direkte Berechnung der Hesse-Matrix und versuchen eine Approximation dieser mithilfe der Information erster Ordnung zu liefern. Dafür benötigen sie, wie das Gradientenverfahren, nur den Gradienten der Zielfunktion in jeder Iteration. Mit der Messung der Änderungen in den Gradienten und in dem Datenpunkt konstruieren sie ein Modell, das gut genug ist, um idealerweise eine super-lineare Konvergenz zu garantieren. Für komplexe Optimierungsprobleme ist die Verbesserung gegenüber dem Gradientenverfahren sehr groß. Da hier die Berechnungen von zweiten Ableitungen nicht mehr nötig sind, sind Quasi-Newton-Verfahren effizienter als das klassische Newton-Verfahren [13, S. 193-194].
Der allgemeine Quasi-Newton-Algorithmus hat die unten dargestellte Form. Die verschiedene Techniken zur Berechnung der approximierten Hesse-Matrix legen die Arten der Quasi-Newton

Algorithmus 3 Allgemeine Quasi-Newton Methode

Input: Startwert ϑ_0, Konvergenz-rate $\varepsilon > 0$, Approximation an der inversen Hesse-Matrix B_0

$k \leftarrow 0$

while $\|\nabla J_k\| > \varepsilon$ **do**

Berechne die Suchrichtung $p_k = -B_k \nabla J_k$

Setze $\vartheta_{k+1} = \vartheta_k + \alpha_k p_k$, wobei α_k die Lösung der als lineare Suche bekannte Minimierungsaufgabe $argmin_{\alpha>0} J(\vartheta_k + \alpha_k p_k)$ ist

Definiere $s_k = \vartheta_{k+1} - \vartheta_k$ und $y_k = \nabla J_{k+1} - \nabla J_k$

Berechne B_{k+1}

$k \leftarrow k + 1$

end while

Methode fest. Im kommenden Abschnitt wird das bekannteste Quasi-Newton Verfahren vorgestellt.

2.2.3 Das Broyden-Fletcher-Goldfarb-Shanno (BFGS) Verfahren

Die BFGS Methode wurde von den vier Mathematikern Broyden, Fletcher, Goldfarb und Shanno unabhängig voneinander und fast zeitgleich entdeckt. Für die Konstruktion der Methode wird das quadratische Modell der Zielfunktion J mit dem Iterationswert ϑ_k zum Zeitpunkt k verwendet:

$$m_k(p) = J_k + \nabla J_k^T p + \frac{1}{2} p^T H_k p. \tag{2.6}$$

H_k ist hier eine $d \times d$ symmetrische, positiv definite Matrix, die in jeder Iteration aufdatiert wird. Der Wert und der Gradient dieses Modells bei $p = 0$ beträgt J_k bzw. ∇J_k. Der Minimierer dieses konvexen quadratischen Modells ist $p_k = -H_k^{-1} \nabla J_k$ und wird als Suchrichtung für die neue Iteration benutzt:

$$\vartheta_{k+1} = \vartheta_k + \alpha_k p_k, \tag{2.7}$$

wobei die Schrittweite α_k so gewählt ist, dass sie die Wolfe-Bedingungen erfüllt:

$$J(\vartheta_k + \alpha_k p_k) \leq J(\vartheta_k) + c_1 \alpha_k \nabla J_k^T p_k \tag{2.8}$$

$$\nabla J(\vartheta_k + \alpha_k p_k)^T p_k \geq c_2 \nabla J_k^T p_k \tag{2.9}$$

mit $0 < c_1 < c_2 < 1$.

Anstatt H_k in jeder Iteration neu zu Berechnen, hat Davidon vorgeschlagen, sie in einer einfache Weise aufzudatieren, um die im letzten Schritt gemessene Krümmung zu berücksichtigen [13,

S. 194]. Angenommen, wir haben die nächste Iteration ϑ_{k+1} generiert und wollen ein neues quadratisches Modell der Form

$$m_{k+1}(p) = J_{k+1} + \nabla J_{k+1}^T p + \frac{1}{2} p^T H_{k+1} p \tag{2.10}$$

konstruieren. Basierend auf dem Wissen aus dem letzten Schritt, sollen wir nun sinnvolle Bedingungen an H_{k+1} stellen. Einer solcher Bedingungen ist, dass der Gradient von m_{k+1} dem Gradienten der Zielfunktion J in der letzten zwei Iterationen ϑ_k und ϑ_{k+1} entsprechen soll, d.h.

- ∇m_{k+1} soll ∇J_k entsprechen
- ∇m_{k+1} soll ∇J_{k+1} entsprechen

Die zweite Bedingung ist für $p = 0$ genau erfüllt, da $\nabla m_{k+1} = \nabla J_{k+1} + H_{k+1} p$. Die erste Bedingung kann mathematisch wie folgt formuliert werden:

$$\nabla m_{k+1}(-\alpha_k p_k) = \nabla J_{k+1} - \alpha_k H_{k+1} p_k \stackrel{!}{=} \nabla J_k \tag{2.11}$$

Somit erhalten wir $H_{k+1}\alpha_k p_k = \nabla J_{k+1} - \nabla J_k$ und mit $s_k = \vartheta_{k+1} - \vartheta_k = \alpha_k p_k$ und $y_k = \nabla J_{k+1} - \nabla J_k$ erhalten wir die sogenannte Sekantengleichung:

$$H_{k+1} s_k = y_k. \tag{2.12}$$

Die Sekantengleichung 2.12 erfordert also, dass die positiv definite, symmetrische Matrix H_{k+1} s_k nach y_k abbildet. Dies ist nur möglich, wenn s_k und y_k die Krümmungsbedingung $s_k^T y_k > 0$ erfüllen[2]. Für stark konvexe Funktionen ist die Krümmungsbedingung für beliebige ϑ_{k+1} und ϑ_k erfüllt. Für nicht konvexe Zielfunktionen benötigt man zusätzliche Voraussetzungen bei der Bestimmung der Schrittweite α, die in [13, S. 195] nachgelesen werden können. Wenn die Krümmungsbedingung erfüllt ist, hat die Sekantengleichung (2.12) immer eine Lösung. Eine solcher Lösungen kann mithilfe der symmetrischen Rang-2 Aktualisierung

$$H_{k+1} = H_k + \gamma u u^T + \delta v v^T, \quad u, v \in \mathbb{R}^n, \quad \gamma, \delta \in \mathbb{R} \tag{2.13}$$

berechnet werden. Mit Einsetzen dieser Gleichung in (2.12) erhalten wir die Gleichung

$$H_k s_k + \gamma u u^T s_k + \delta v v^T s_k = y_k \tag{2.14}$$

mit der Lösung

$$u = y_k, \quad v = H_k s_k, \quad \gamma = \frac{1}{y_k^T s_k} \quad \text{und} \quad \delta = -\frac{1}{s_k^T H_k s_k}.$$

[2] Multipliziere (2.12) mit s_k^T von links und nutze, dass H_{k+1} positiv definit ist.

Einsetzen dieser Lösung in (2.13) liefert die BFGS Aktualisierungsformel für die Hesse-Matrix

$$H_{k+1} = H_k + \frac{y_k y_k^T}{s_k^T y_k} - \frac{(H_k s_k)(H_k s_k)^T}{s_k^T H_k s_k}. \tag{2.15}$$

Anstatt eine Approximation der Hesse-Matrix zu berechnen, approximiert man, um das häufige Lösen eines Gleichungssystems zu sparen, die inverse Hesse-Matrix. Mit Anwendung der Sherman-Morrison-Woodbury Formel:

$$\left(A + UV^T\right)^{-1} = A^{-1} - A^{-1}UC^{-1}V^T A^{-1}, \tag{2.16}$$

wobei $C = I + V^T A^{-1} U$, $C \in \mathbb{R}^{2\times 2}$, $U = [u_1, u_2]$ und $V = [v_1, v_2]$, erhalten wir eine Aktualisierungsformel für die approximierte inverse Hesse-Matrix $B_{k+1} = H_{k+1}^{-1}$:

$$B_{k+1} = B_k - \frac{B_k y_k s_k^T + s_k y_k^T B_k}{s_k^T y_k} + \frac{s_k s_k^T}{s_k^T y_k}\left(1 + \frac{y_k^T B_k y_k}{s_k^T y_k}\right).$$

Äquivalent dazu ist

$$B_{k+1} = \left(I - \rho_k s_k y_k^T\right) B_k \left(I - \rho_k y_k s_k^T\right) + \rho_k s_k s_k^T, \tag{2.17}$$

wobei $\rho_k = \dfrac{1}{y_k^T s_k}$ bezeichnet. Der resultierende Algorithmus ist unten angegeben.

Algorithmus 4 BFGS Methode

Input: Startwert ϑ_0, Toleranz $\varepsilon > 0$, Approximation der inversen Hesse-Matrix B_0

$k \leftarrow 0$

while $\|\nabla J_k\| > \varepsilon$ **do**

Berechne die Suchrichtung $p_k = -B_k \nabla J_k$

Setze $\vartheta_{k+1} = \vartheta_k + \alpha_k p_k$, wobei α_k die Wolfe-Bedingungen (2.8) erfüllt

Definiere $s_k = \vartheta_{k+1} - \vartheta_k$ und $y_k = \nabla J_{k+1} - \nabla J_k$

Berechne B_{k+1} wie in (2.17)

$k \leftarrow k + 1$

end while

2.2.4 Das Limited-Memory BFGS (L-BFGS) Verfahren

Das Quasi-Newton Verfahren aus Kapitel 2.2.3 ist nicht direkt auf größere Optimierungsprobleme anwendbar, da die Approximationen an der Hesse-Matrix oder inversen Hesse-Matrix gewöhnlich dicht sind. Die Speicher- und Rechenanforderungen wachsen proportional zu d^2 und werden übermäßig groß für ein großes d [13, S. 223], wobei d die Dimension des Problems ist. Jedoch ist es möglich die BFGS-Methode auf verschiedenen Arten zu modifizieren bzw. zu erweitern, sodass diese für größere Probleme geeignet werden.
Einer der derartigen Ansätze - die Limited-Memory Quasi-Newton Methode - modifiziert die Techniken aus dem vorherigen Kapitel um eine Approximation der inversen Hesse-Matrix zu erhalten, die kompakt in nur wenigen Vektoren der Länge d gespeichert werden kann. Diese Methoden sind robust, benötigen keinen großen Rechenaufwand und sind einfach zu implementieren, konvergieren jedoch langsam [13, S. 223]. Quasi-Newton-Methoden mit begrenztem Speicherplatz sind nützlich bei der Lösung großer Probleme, deren Hesse-Matrizen nicht mit vertretbarem Aufwand berechnet werden können oder die zu dicht sind, um leicht manipuliert werden zu können. Diese Methoden ermöglichen kompakte Approximationen von Hesse-Matrizen: Anstatt vollständige $d \times d$ Annäherungen zu speichern, speichern sie nur wenige Vektoren der Länge d, die die Approximationen implizit darstellen. Trotz dieser geringen Speicheranforderungen, ergeben sie oft eine akzeptable Konvergenzrate. Wir werden uns hauptsächlich auf einen Algorithmus konzentrieren, der als L-BFGS Methode bekannt ist (Nocedal, 1980) und wie der Name schon verrät, auf der Aktualisierungsformel von BFGS basiert.
Die Grundidee dieser Methode ist es, Krümmungsinformationen nur aus den neuesten Iterationen zu verwenden, um die Approximation an der Hesse-Matrix zu konstruieren. Krümmungsinformationen aus früheren Iterationen, die für das tatsächliche Verhalten der Hesse-Matrix in der aktuellen Iteration weniger relevant sind, werden zum Ziel der Speichereinsparung verworfen. Um das Speicherproblem zu umgehen, speichern wir nun implizit eine modifizierte Version von B_k, indem wir eine nur bestimmte Anzahl m der Vektorpaare $\{s_k, y_k\}$ berücksichtigen. Nachdem in einer Iteration s_k und y_k neu berechnet wurden und das Gedächtnis der Methode voll ist, wird das älteste Vektorpaar aus dem Gedächtnis gelöscht und durch das neue Paar $\{s_k, y_k\}$ ersetzt, das aus dem aktuellen Schritt erhalten wurde. Auf diese Weise enthält die Menge von Vektorpaaren $\{s_k, y_k\}$ Krümmungsinformationen aus den m letzten Iterationen.
Die praktische Erfahrung hat gezeigt, dass eine geringe Größe von m (zwischen 3 und 20) oft zu zufriedenstellenden Ergebnissen führen kann [13, S. 227]. Abgesehen von der modifizierten Matrix-Aktualisierungsstrategie ist die Implementierung der L-BFGS Methode identisch mit der

Standard-BFGS-Methode. Insbesondere kann die gleiche Strategie zur Liniensuche verwendet werden. Wir betrachten die bereits bekannte BFGS Aktualisierungsformel für die Approximation der inversen Hesse-Matrix:

$$B_{k+1} = U_k^T B_k U_k + \rho_k s_k s_k^T \quad \text{mit} \quad U_k = \left(I - \rho_k y_k s_k^T\right) \quad \text{und} \quad \rho_k = \frac{1}{y_k^T s_k} \tag{2.18}$$

Für die ersten k Iterationen gilt:

$$\begin{aligned}
B_1 &= U_0^T B_0 U_0 + \rho_0 s_0 s_0^T \\
B_2 &= U_1^T B_1 U_1 + \rho_1 s_1 s_1^T \\
&= U_1^T \left(U_0^T B_0 U_0 + \rho_0 s_0 s_0^T\right) U_1 + \rho_1 s_1 s_1^T \\
&= \left(U_0 U_1\right)^T B_0 \left(U_0 U_1\right) + \rho_0 \left(U_1^T s_0\right) \left(U_1^T s_0\right)^T + \rho_1 s_1 s_1^T \\
&... \\
B_k &= \left(U_0 \cdots U_{k-1}\right)^T B_0 \left(U_0 \cdots U_{k-1}\right) + \rho_0 \left(\left(U_0 \cdots U_{k-1}\right)^T s_0\right) \left(\left(U_0 \cdots U_{k-1}\right)^T s_0\right)^T \\
&\quad + ... + \rho_{k-2} \left(U_{k-1}^T s_{k-2}\right) \left(U_{k-1}^T s_{k-2}\right)^T + \rho_{k-1} s_{k-1} s_{k-1}^T
\end{aligned}$$

Wir sehen also, dass jedes B_k aus B_0 und Vektorpaaren $\{s_k, y_k\}$, $j = 0, 1, ..., k-1$ berechnet werden kann. Hieraus können wir die L-BFGS Formel mit dem Speicherparameter m herleiten:

$$B_k = \left(U_{k-m_k} ... U_{k-1}\right)^T B_0^{(k)} \left(U_{k-m_k} ... U_{k-1}\right) + \sum_{j=k-m_k}^{k-1} \rho_j \left(\left(U_{j+1} ... U_{k-1}\right)^T s_j\right) \left(\left(U_{j+1} ... U_{k-1}\right)^T s_j\right)^T, \tag{2.19}$$

wobei $m_k = min\left(k, m\right)$ gesetzt wird. Aus diesem Ausdruck können wir die im Algorithmus 5 gegebene rekursive Prozedur ableiten, um das Produkt $B_k \nabla J_k$ für die Abstiegsrichtung effizient zu berechnen [13, S. 225]. Der resultierende Algorithmus ist im Algorithmus 6 aufgeführt.

Algorithmus 5 L-BFGS 2-Schleifen Rekursion

$q \leftarrow \nabla J_k$

for $i = k-1, k-2, ..., k-m$ **do**

$\alpha_i \leftarrow \rho_i s_i^T q; \quad q \leftarrow q - \alpha_i y_i$

end for

$r \leftarrow B_k^{(0)} q$

for $i = k-m, k-m+1, ..., k-1$ **do**

$\beta \leftarrow \rho_i y_i^T r; \quad r \leftarrow r + s_i \left(\alpha_i - \beta\right)$

end for

STOP mit $r = B_k \nabla J_k$

Algorithmus 6 L-BFGS Methode

Input: Startwert ϑ_0, ganze Zahl $m > 0$

$k \leftarrow 0$

while Abbruchbedingung nicht erfüllt **do**

Wähle $B_k^{(0)}$, z.B. als $\frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} I$

Berechne die Suchrichtung $p_k = -B_k \nabla J_k$ aus Algorithmus 5

Setze $\vartheta_{k+1} = \vartheta_k + \alpha_k p_k$, wobei α_k die Wolfe-Bedingungen erfüllt

if $k > m$ **then**

Lösche das Vektorpaar $\{s_{k-m}, y_{k-m}\}$ aus dem Speicher

end if

Berechne und speichere $s_k = \vartheta_{k+1} - \vartheta_k$ und $y_k = \nabla J_{k+1} - \nabla J_k$

$k \leftarrow k + 1$

end while

2.3 Stochastische Quasi-Newton-Verfahren für maschinelles Lernen

Zwar gilt die L-BFGS Methode als state-of-the-art Methode in der numerischen Optimierung, aber wenn wir uns im stochastischen Modus befinden und mit Echtzeitdaten zu tun haben, können mehrere Probleme auftreten. Diese Probleme wurden bereits bei den stochastischen Methoden erster Ordnung erwähnt. Oft können wir den Gradienten und die Approximation an der Hesse-Matrix nicht direkt berechnen, da die Daten in den Online-Modus einfließen. In den anderen Fällen reicht die Rechenkapazität der Computer nicht aus, Gradienten von den gesamten Datenmengen zu berechnen. Eine Modifizierung der Optimierungsmethoden zum stochastischen Modus ist also notwendig.

2.3.1 Das robuste Multi-Batch L-BFGS Verfahren

In diesem Abschnitt diskutieren wir das im Paper [1] vorgestellte Verfahren.
Im überwachten Lernen versucht man die erwartete Risikofunktion

$$R(\vartheta) = \mathbb{E}\left[f(\vartheta; x, y)\right] = \int_{\Omega} f(\vartheta; x, y) \mathbb{P}(x, y) \tag{2.20}$$

zu minimieren, wobei (x, y) die Input-Output Paare, Ω ein Raum von Input-Output Paaren versehen mit der Wahrscheinlichkeitsverteilung $\mathbb{P}(x, y)$ und $f : \mathbb{R}^d \to \mathbb{R}$ eine Komposition von

Hypothese- h und Fehlerfunktion ℓ ist, d.h. $f = f(\vartheta; x_i, y_i) = \ell(h_\vartheta(x_i); y_i)$. Da die Wahrscheinlichkeitsverteilung $\mathbb{P}$ normalerweise nicht bekannt ist, approximiert man (2.20) mit der empirischen Risikofunktion

$$F(\vartheta) = \frac{1}{n}\sum_{i=1}^{n} f(\vartheta; x_i, y_i) \stackrel{\text{def}}{=} \sum_{i=1}^{n} f_i(\vartheta), \tag{2.21}$$

wobei $(x_i, y_i),\ i = 1, ..., n$ die Trainingsmuster bezeichnen, die in der Literatur auch Datenpunkte oder Samples genannt werden. Das Problem des Trainings besteht nun darin, eine optimale Wahl der Parameter $\vartheta \in \mathbb{R}^d$ zu treffen, sodass (2.21) minimiert wird. Wir suchen also die Lösung des Minimierungsproblems

$$\min_{\vartheta \in \mathbb{R}^d} F(\vartheta) = \sum_{i=1}^{n} f_i(\vartheta). \tag{2.22}$$

In einem vollen Batch-Ansatz wendet man eine gradientenbasierte Methode auf dieses deterministische Optimierungsproblem (2.22) an. Eine beliebte solche Methode ist die im Kapitel 2.2.4 eingeführte L-BFGS Methode. Wenn aber n zu groß ist, möchte man die Auswertung von der Funktion F und dem Gradienten ∇F auf dem Datensatz vermeiden. Stattdessen implementiert man einen multi-batch Ansatz der L-BFGS Methode. In diesem Ansatz wird nicht die gesamte Trainingsmenge $T = \{(x_i, y_i),\ i = 1, ..., n\}$ in jeder Iteration benutzt, sondern eine Untermenge der Trainingsmenge wird angewandt, um den Gradienten auszuwerten. Der Datensatz wird zufällig in eine gewisse Anzahl von Batches unterteilt (z.B. 10, 50 oder 100) und die Minimierung wird in jeder Iteration bezüglich dieser verschiedenen Batches durchgeführt. In der k-ten Iteration wählt also der Algorithmus eine Teilmenge $S_k \subset \{1, ..., n\}$, berechnet

$$F^{S_k}(\vartheta_k) = \frac{1}{|S_k|}\sum_{i \in S_k} f_i(\vartheta_k), \quad g_k^{S_k} = \nabla F^{S_k}(\vartheta_k) = \frac{1}{|S_k|}\sum_{i \in S_k} \nabla f_i(\vartheta_k) \tag{2.23}$$

und macht einen Schritt in die Richtung $-B_k g_k^{S_k}$, wobei B_k eine Approximation an $\nabla^2 F(\vartheta_k)^{-1}$ ist. Da der Ansatz eine stochastische Approximation vom Gradienten benutzt, kann er nicht mehr als deterministisch betrachtet werden. Die freie Änderung des Samples S_k in jeder Iteration gibt diesem Ansatz Flexibilität bei der Implementierung und ist vorteilhaft für den Lernprozess. Hier wird S_k als Sample der Trainingspunkte bezeichnet, aber was in S_k gemeint ist, ist die Indexmenge der Trainingspunkte. Wir müssen jedoch diesen Ansatz von der klassischen SGD-Methode unterscheiden, die kleine Mini-Batches und rauschhaltige Annäherungen an den Gradienten verwendet. Der hier vorgestellte Algorithmus operiert mit viel größeren Batches, so dass gelieferte Funktions- und Gradientenauswertungen vorteilhaft sind. Dies führt zu Gradienten mit relativ geringer Varianz und begründet die Verwendung eines Verfahrens zweiter Ordnung, wie L-BFGS Methode.

Eine robuste Implementierung der stochastischen L-BFGS-Methode basiert auf der folgenden Beobachtung: die Schwierigkeit, die durch die Verwendung verschiedener Samples S_k in jeder Iteration entstehen, kann umgegangen werden, falls aufeinanderfolgende Samples S_k und S_{k+1} eine Überlappung

$$O_k = S_k \cap S_{k+1} \neq \emptyset$$

besitzen. Dann kann man stabile Quasi-Newton Aktualisierungen durchführen, indem man die Differenz in Gradienten basierend auf diese Überlappungen berechnet und in der L-BFGS Akstualisierungformel das Korrekturpaar (s_k, y_k) mit

$$y_{k+1} = g_{k+1}^{O_k} - g_k^{O_k}, \quad s_{k+1} = \vartheta_{k+1} - \vartheta_k \tag{2.24}$$

benutzt. Wenn der Überlappung O_k nicht zu klein ist, ist y_k eine gute Näherung an die Krümmung der Zielfunktion F entlang der letzten Verschiebung und führt zu einem produktiven Quasi-Newton-Schritt. Diese Beobachtung basiert auf einer wichtigen Eigenschaft Newton-ähnlicher Methoden, nämlich, dass es viel mehr Freiheit bei der Wahl einer Approximation an der Hesse-Matrix gibt, als bei der Berechnung des Gradienten [3]. Genauer gesagt, kann ein kleineres Sample O_k zur Aktualisierung der approximativen inversen Hesse-Matrix B_k verwendet werden, als das Sample S_k zur Berechnung des Batch-Gradienten $g_k^{S_k}$ zum Definieren der Suchrichtung $-B_k g_k^{S_k}$. Einziger Unterschied zur klassischen L-BFSG Methode ist hier also die Auswertung der Zielfunktion und dem stochastischen Gradienten auf der Überlappungen aufeinanderfolgender Samples S_k und S_{k+1}

$$F^{O_k}(\vartheta_k) = \frac{1}{|O_k|} \sum_{i \in O_k} f_i(\vartheta_k), \quad g_k^{O_k} = \nabla F^{O_k}(\vartheta_k) = \frac{1}{|O_k|} \sum_{i \in O_k} \nabla f_i(\vartheta_k) \tag{2.25}$$

und Benutzung dieser zur Aktualisierung der Korrekturpaare (s_k, y_k).
Ein Pseudocode der vorgestellten Multi-Batch L-BFGS Methode ist im Algorithmus unten aufgeführt und hängt von zwei zusätzlichen Parameter ab: r bezeichnet den Anteil von Samples in dem Datensatz, der zum Ausrechnen des stochastischen Gradienten verwendet wird und o bezeichnet die Länge der Überlappung zwischen aufeinanderfolgenden Samples.

Sample-Generierung: Multi-Batch Sampling

Im Multi-Batch-Sampling können mehrere Strategien verwendet werden, mit der einzigen Einschränkung, dass sich die aufeinanderfolgenden Samples S_k und S_{k+1} bis zu einem gewissen Grad überschneiden sollten. Zwei mögliche Sampling-Stategien sind: (i) Überlappungen O_k werden bei der Sample-Generation erzwungen, (ii) die Überlappungsmenge O_k wird als Sub-Sample von der

Algorithmus 7 Multi-Batch L-BFGS Methode (MB-LBFGS)

Input: Startwert ϑ_0, Trainingsdatensatz $T = \{(x_i, y_i),\ i = 1, ..., n\}$, Gedächtnisparameter $m \in \mathbb{Z}_{>0}$, Batchgröße r, Größe der Überlappung $o \in (0, 1)$

$k \leftarrow 0$

Generiere initiales Batch S_0

for $k = 0, 1, 2, ...$ **do**

Berechne die Suchrichtung $p_k = B_k g_k^{S_k}$

Wähle eine Schrittweite $\alpha_k > 0$

Berechne $\vartheta_{k+1} = \vartheta_k + \alpha_k p_k$

Generiere nächstes Batch S_{k+1}

Berechne die Krümmungspaare $s_{k+1} = \vartheta_{k+1} - \vartheta_k$ und $y_{k+1} = g_{k+1}^{O_k} - g_k^{O_k}$

Ersetze das älteste Paar (s_i, y_i) durch (s_{k+1}, y_{k+1}) falls bereits m Paare gespeichert, ansonsten füge das neue Paar einfach hinzu

end for

Sample S_k gewählt.

In der ersten Strategie wird zunächst eine initiale Teilmenge S_0 des Datensatzes ausgewählt. Jede nächste Teilmenge ist der Form $S_k = \{O_{k-1}, N_k, O_k\}$, wobei O_{k-1} bzw. O_k die Überlappungen mit S_{k-1} bzw. S_{k+1} sind und N_k ist der Rest des Batches S_k. Nach jeder Epoche werden die Daten gemischt und das Prozedere wiederholt. Diese Strategie hat den Vorteil, dass man keine zusätzliche Rechnung zur Auswertung von $g_k^{O_k}$ und $g_{k+1}^{O_k}$ braucht, jedoch sind die Batches S_k nicht stochastisch unabhängig.

Die zweite Strategie ist einfacher und benötigt wenig Kontrolle. In jeder Iteration k ist das Batch S_k generiert durch zufälliges Auswählen der $|S_k|$ Elemente aus $\{1, ..., n\}$. Die Überlappung O_k wird dann durch wiederholtes zufälliges Auswählen von $|O_k|$ Elemente aus S_k erzeugt. Dieses Prozedur wird Subsampling genannt. Sie ist leicht aufwendiger, da $g_{k+1}^{O_k}$ eine zusätzliche Rechnung benötigt, aber wenn die Überlappung O_k klein ist, ist die Rechenaufwand gering. Dafür wird hier als Vorteil stochastische Unabhängigkeit erreicht.

2.3.2 Das Progressive-Batching L-BFGS Verfahren

Das Progressive-Batching L-BFGS Verfahren gehört ebenfalls zu den stochastischen quasi-Newton Methoden. Die Methode wurde von Bollapagada *et. al.* im Paper [2] vorgestellt. Diese Version der stochastischen L-BFGS Methode kombiniert drei algorithmische Komponente, die in der letzten

Zeit Aufmerksamkeit in der Literatur erhalten haben: das progressive Batching, die adaptive Wahl der Schrittweite und die stabile Quasi-Newton-Aktualisierung.
Anders als im MB-LBFGS Verfahren, wird hier die Batchgröße variabel gehalten. Angefangen wird mit einer kleinen Batchgröße, um die Laufzeit pro Iteration niedrig zu halten. Immer, wenn auf diesem Batch berechneter stochastischer Gradient nicht akkurat genug ist, um sinnvolle quadratische Modelle zu konstruieren und zuverlässige lineare Suche durchzuführen, wird die Batchgröße erhöht.

IPQN-Test

Die Bestimmung der Größe der Erhöhung ist eine herausfordernde Aufgabe und wird mithilfe des Inneres-Produkt-Quasi-Newton Tests (IPQN-Test) geliefert. Sei $g_k^{S_k}$ der stochastische Gradient ausgewertet auf dem Batch $S_k \subset \{1, ..., n\}$ (definiert wie in (2.23)) und B_k die Approximation der inversen Hesse Matrix. Um den IPQN-Test herzuleiten, verlangen wir, dass die stochastische Quasi-Newton Suchrichtung $d_k = -B_k g_k^{S_k}$ einen spitzen Winkel mit der wahren Quasi-Newton Suchrichtung bildet $-B_k \nabla F(\vartheta_k)$ bildet, d.h.

$$\mathbb{E}\left[(B_k \nabla F(\vartheta_k))^T \left(B_k g_k^{S_k}\right)\right] = \|B_k \nabla F(\vartheta_k)\|^2. \tag{2.26}$$

Wir wollen die Varianz der Menge kontrollieren, indem wir die Batchgröße $|S_k|$ so wählen, dass die folgende Ungleichung erfüllt ist:

$$\mathbb{E}\left[\left((H_k \nabla F(\vartheta_k))^T \left(H_k g_k^{S_k}\right) - \|H_k \nabla F(\vartheta_k)\|^2\right)^2\right] \leq \theta^2 \|H_k \nabla F(\vartheta_k)\|^4. \tag{2.27}$$

Ersetzt man nun den wahren erwarteten Gradienten und die wahre Varianz durch Sample Gradient und Sample Varianz, erhält man den IPQN Test:

$$\frac{Var_{i \in S_k^v}\left((g_k^i)^T B_k^2 g_k^{S_k}\right)}{|S_k|} \leq \theta^2 \|B_k g_k^{S_k}\|^4, \tag{2.28}$$

wobei $g_k^i = \nabla F_i(\vartheta_k)$ und $S_k^v \subseteq S_k$ ist. Der Varianzterm ist definiert durch

$$\frac{\sum_{i \in S_k^v}\left((g_k^i)^T B_k^2 g_k^{S_k} - \|B_k g_k^{S_k}\|)^2\right)^2}{|S_k^v| - 1}. \tag{2.29}$$

Wenn (2.28) nicht erfüllt ist, erhöhen wir die Batchgröße von $|S_k|$ auf $|\overline{S}_k|$, so dass $\|B_k g_k^{S_k}\| \cong \|B_k g_k^{\overline{S}_k}\|$ ist. Eine Umformung von (2.28) liefert dann eine Abschätzung für die Größe des neuen Batches:

$$|\overline{S}_k| \geq \frac{Var_{i \in S_k^v}\left((g_k^i)^T B_k^2 g_k^{S_k}\right)}{\theta^2 \|B_k g_k^{S_k}\|^4} =: b_k. \tag{2.30}$$

Stochastische lineare Suche

Stochastische lineare Suche ist selten in der Praxis angewandt, da sie eine Entscheidung basierend auf den verzerrten Funktionswert auf dem Sample S_k machen muss. Jedoch wird hier eine Formel für die Berechnung der initialen Schrittweite α_k vorgestellt, die mit großer Wahrscheinlichkeit garantiert, dass der Zielfunktionswert sich verringert. Somit wird in jeder Iteration

$$\alpha_k = \left(1 + \frac{Var_{i \in S_k^v}\left\{g_k^i\right\}}{|S_k| \| g_k^{S_k} \|^2}\right)^{-1} \tag{2.31}$$

berechnet, wobei

$$Var_{i \in S_k^v}\left\{g_k^i\right\} = \frac{1}{|S_k - 1|} \sum_{i \in S_k^v} \| g_k^i - g_k^{S_k} \|^2 \tag{2.32}$$

und $S_k^v \subseteq S_k$ gilt. Die Herleitung dieser Formel kann in [2, S. 4] nachgelesen werden.
Mit dieser Initialisierung der Schrittweite wird dann die lineare Suche mit Backtracking Algorithmus [13, S. 41] durchgeführt, d.h. die Schrittweite wird solange verkleinert, bis die Armijo-Bedingung (2.8) erfüllt ist.

Mit dem IPQN-Test und der stochastischen linearen Suche, zusammen mit der Überlappungsstrategie aus dem Abschnitt 2.3.1 zur Berechnung der stochastischen Gradienten, erhalten wir den Progressive-Batching L-BFGS Algorithmus.

Algorithmus 8 Progressive-Batch L-BFGS Methode (PB-LBFGS)

Input: Startwert ϑ_0, initiale Batchgröße $|S_0|$, ganze Zahl $m > 0$

$k \leftarrow 0$

while Abbruchbedingung nicht erfüllt **do**

Wähle S_k mit der Batchgröße $|S_k|$

if Bedingung (2.28) nicht erfüllt **then**

Berechne b_k wie in (2.30) und erweitere S_k auf die Summe b_k Samples aus $S^+ \subseteq \{1, ..., N\} \setminus S_k$

end if

Berechne den stoch. Gradienten $g_k^{S_k}$ und die Suchrichtung p_k mit dem Algorithmus 5

Berechne α_k mit (2.31) und setze $\alpha_k = \alpha_k/2$, solange Armijo-Bedingung nicht erfüllt wird

Setze $\vartheta_{k+1} = \vartheta_k + \alpha_k p_k$

Berechne $y_k = g_{k+1}^{O_k} - g_k^{O_k}, \quad s_{k+1} = \theta_{k+1} - \theta_k$

if $y_k^T s_k > \varepsilon \|s_k\|^2$ **then**

if Anzahl der gespeicherten Vektorpaare (y_j, s_j) überschreitet m **then**

Lösche das älteste Vektorpaar (y_j, s_j)

end if

Speichere das neue Krümmungspaar (y_k, s_k)

end if

Setze $k \leftarrow k + 1, \quad |S_k| = |S_{k-1}|$

end while

3 Numerische Experimente

In diesem Abschnitt vorgestellte numerische Ergebnisse wurden in der Programmiersprache Python programmiert. Python ist eine sehr zugängliche Programmiersprache. Sie hat sich zur beliebtesten Programmiersprache für die Datenwissenschaften entwickelt, weil sie es erlaubt, die Ideen schnell zu notieren und Konzepte direkt umzusetzen. Um effiziente Berechnungen in Python zu implementieren, wurden weitere Bibliotheken wie NumPy benutzt, die die rechenintensive mathematische Operationen, wie z.B. die Matrizenmultiplikation, außerhalb von Python in einer performanterer Programmiersprache durchführt.

3.1 Experimente auf dem MNIST-Datensatz

Der MNIST Datensatz besteht aus 50 000 Trainings- und 10 000 Testbildern von handgeschriebenen Ziffern. Die Pixelgröße der einzelnen Bilder beträgt 28×28. MNIST wurde von LeCun und Cortes [9] entwickelt, um eine Datenbank für hochvariante, handgeschriebene Ziffern für Klassifizierungen und Tests statistischer Methoden zu verwenden. Ziel war es, eine aus der Realität entnommene dichte Menge von verschiedenen Handschriften in der Datenbank unterzubringen. Die 60 000 Trainings- und Testdaten stammen von 250 verschiedenen Personen. Alle Ziffern in MNIST wurden in ihrer Größe normalisiert und zentriert. Die Auflösung aller Bilder ist gleich. Das Entwicklungsziel des Formates liegt in der Aufwandsminimierung des Formatierens und Auslesens der Datensätze. Die Bilddaten sind in Graustufen angegeben. Das bedeutet, dass die Pixel einen Wertebereich von 0 bis 255 haben, wobei 0 für Weiß und 255 für Schwarz steht. Ein Ausschnitt aus dem MNIST Datensatz ist in Abbildung unten zu sehen.

Abbildung 3.1: Der MNIST Datensatz

Quelle: `https://en.wikipedia.org/wiki/MNIST_database`

3.1.1 Das neuronale Netzmodell

Das verwendete MLP hat nur einen verdeckten Layer bestehend aus 25 Neuronen. Im Input-Layer befinden sich 784 Neuronen, da diese Anzahl mit der Anzahl der sogenannten Features (hier: Anzahl der Pixeln $28 \cdot 28 = 784$) übereinstimmen soll. Im Output-Layer haben wir 10 Neuronen für die Ziffer $0, ...9$. Außerdem wird zu dem Input- und verdeckten Layer jeweils ein Bias Unit hinzugefügt. Somit erhalten wir insgesamt $(784 + 1) \cdot 25 + (25 + 1) \cdot 10 = 19\,885$ trainierbare Parameter im Netz.

Für jede in dieser Arbeit getestete Optimierungsmethode erfolgt die Initialisierung der Parameter aus einer CSV-Datei mit den Werten aus dem Intervall $[0, 0.12)$. Diese Werte wurden einmal zufällig erzeugt und gespeichert. Somit wird versucht für jeden Algorithmus gleiche Anfangsbedingungen zu schaffen.

Als Aktivierungsfunktion wurde die Sigmoid-Funktion gewählt und die verwendete Fehlerfunktion ist die Kreuzentropie. Zu Regularisierungszwecken wird der Fehlerfunktion ein *Weight Decay* Term mit dem Koeffizient $\lambda = 0.1$ hinzugefügt. Der gesamte Fehler sieht also wie folgt aus:

$$J^*(\vartheta) = J(\vartheta) + \lambda \left(\sum_{i,j} \left(\Theta_{i,j}^{(1)*} \right)^2 + \sum_{i,j} \left(\Theta_{i,j}^{(2)*} \right)^2 \right), \tag{3.1}$$

wobei $\Theta^{(1)*}$ bzw. $\Theta^{(2)*}$ die Matrizen aus Gewichten zwischen Input- und verdecktem btw. verdektem und Output-Layer sind, jedoch im Vergleich zu $\Theta^{(0)}$ und $\Theta^{(1)}$ nicht die Bias-Terme enthalten.

3.1.2 Numerische Ergebnisse

Optimale Wahl der Überlappungsgröße

Wie bereits erwähnt, realisieren die stochastischen Quasi-Newton Methoden MB-LBFGS und PB-LBFGS eine Überlappungsstrategie und garantieren somit, dass die Durchschnittsmenge zweier aufeinanderfolgenden Batches nicht leer ist. Jedoch, wie groß muss man die Überlappungen wählen bei einer gegebenen Batchgröße? Um diese Frage zu beantworten, vergleichen wir verschiedene Batch- und Überlappungsgrößen für die beiden Methoden. Die in den Abbildung 3.2 und 3.3 dargestellten Ergebnisse zeigen, dass bei einer Batchgröße von 256 bzw. 512 für das MB-LBFGS bzw. PB-LBFGS Verfahren jeweils die Überlappungröße von 20% eine optimale Wahl ist.

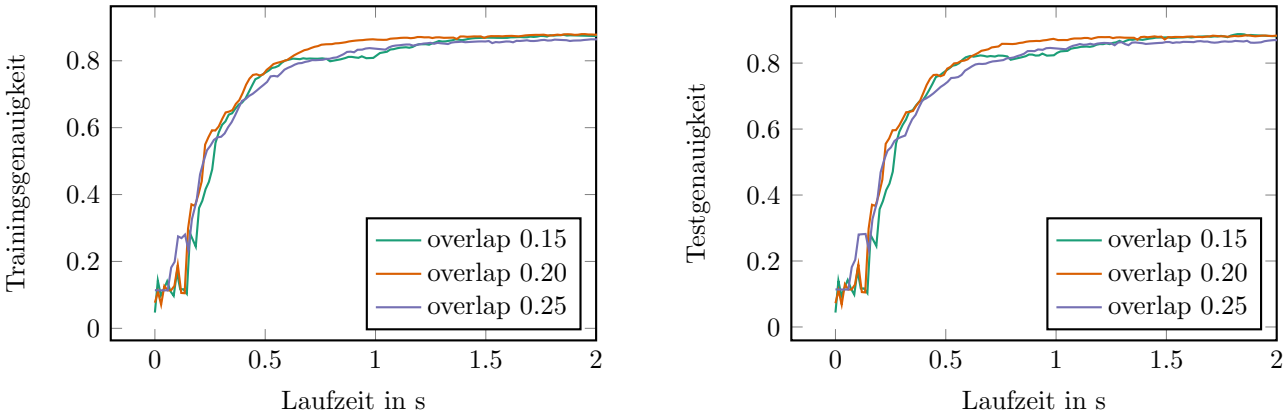

Abbildung 3.2: MB-LBFGS; Auswirkung der Überlappungsgröße auf die Rechenzeit

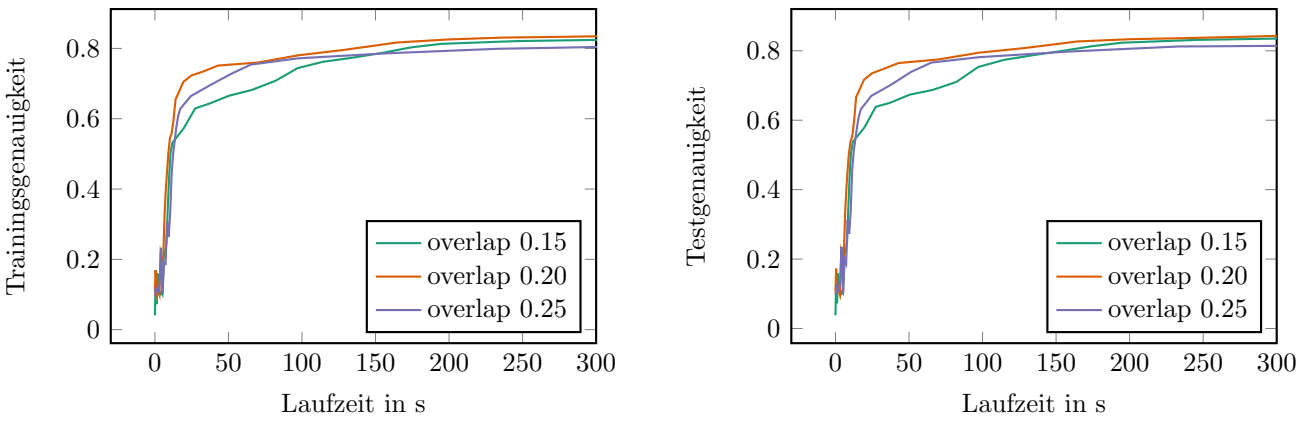

Abbildung 3.3: PB-LBFGS; Auswirkung der Überlappungsgröße auf die Rechenzeit

Insgesamt ist bei einer kleineren Batchgröße eine höhere Überlappungsgröße zu empfehlen, damit die Überlappung nicht zu wenige Samples enthält und nicht zu einem sehr verzerrten

stochastischen Gradienten führt. Dies bestätigen die entsprechende Grafiken, die im Anhang zu finden sind.

Die Rolle der linearen Suche

MB-LBFGS Verfahren kann man mit einer festen oder durch die lineare Suche modifizierten Schrittweite realisieren. Die Abbildung 3.4 zeigt genau die Auswirkung der Anwendung der linearen Suche auf das Verfahren. Für eine Batchgröße von 256 mit der Überlappungsgröße von 30% und den Startwert 0.30 für die Schrittweite sehen wir, wie wichtig die lineare Suche für das stabile Verhalten des Verfahrens ist: mit LS wird eine durchgehende Testgenauigkeit von über 89% erreicht, wohingegen das gleiche Verfahren ohne LS oft abstürzt und stark fluktuiert. Auf der rechte Seite der Grafik ist zu sehen, wie oft in der linearen Suche die Armijo-Bedingung (2.8) nicht erfüllt ist und die zu groß gesetzte Schrittweite verringert wird, um somit einen feineren Schritt in die Richtung des Minimums zu schaffen.

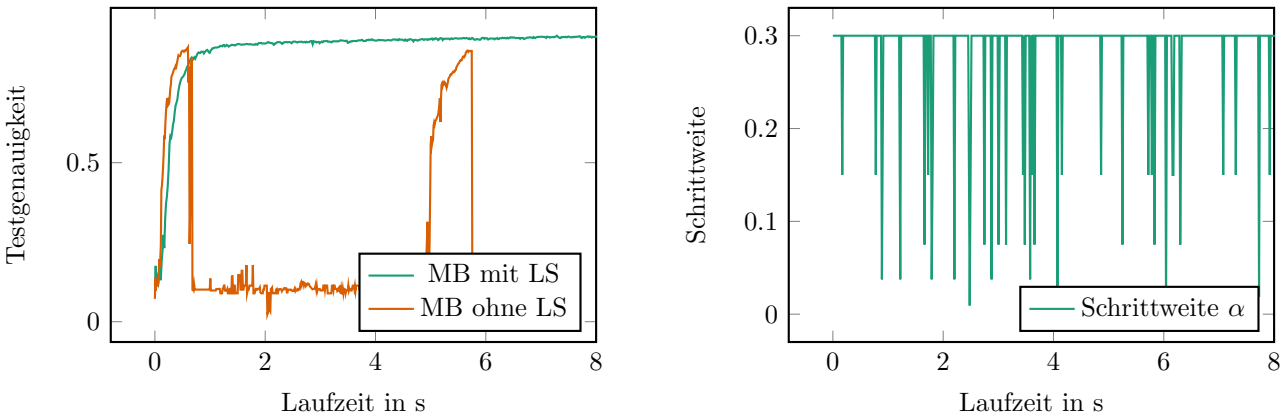

Abbildung 3.4: MB-LBFGS; Auswirkung der linearer Suche (LS)

Das PB-LBFGS Verfahren realisiert eine stochastische lineare Suche mit dem Startwert, der mithilfe (2.31) berechnet wird. Die linke Seite der Abbildung 3.5 zeigt, wie groß dieser Startwert gesetzt wird (grün) und welchen Wert der Algorithmus tatsächlich für die Schrittweite einnimmt (orange). Zu bemerken ist, dass ab der 19. Iteration die von der stochastischen linearen Suche berechnete Startschrittweite (2.31) direkt die Armijo-Bedingung (2.8) erfüllt und nicht weiter verkleinert wird.
In der rechte Seiten der Abbildung 3.5 ist die Batcherhöhung pro Iteration abgebildet, die wir als b_h bezeichnen.

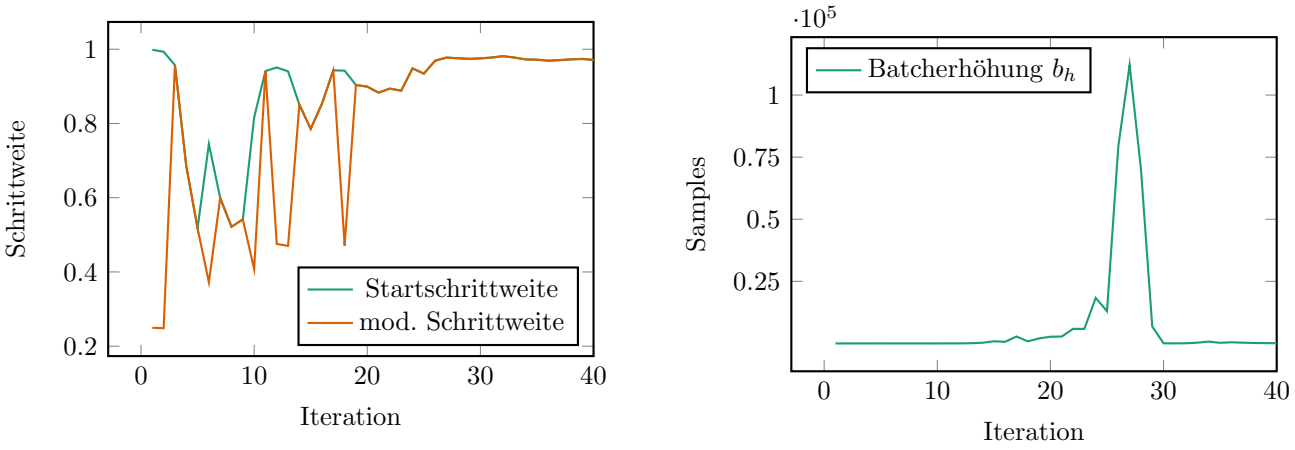

Abbildung 3.5: PB-LBFGS; links: Veränderung der Schrittweite, rechts: Veränderung der Batchgröße

Dabei geht im PB-LBFGS Verfahren die Batcherhöhung sogar so weit, dass ein einzelner Batch ab Iteration 27 fast den kompletten Datensatz umfasst.

Vergleich der Algorithmen

Der Vergleich von Trainings- und Testgenauigkeiten der Algorithmen bezüglich der Iterationen und der Gradientenauswertungen sind recht ähnlich, wie es in den entsprechenden Abbildungen 3.6 und 3.7 zu sehen ist. Die beiden Methoden erster Ordnung SGD und Adam machen lediglich eine Gradientenauswertung pro Iteration. Daher verändern sich deren Grafiken nicht in den beiden Abbildungen. MB-LBFGS bzw. PB-LBFGS machen 4 bzw. $4+1$ (falls die Batchgröße erhöht wird, wertet die Methode den Gradienten auf dem neuen Batch aus, daher $+1$) Gradientenauswertungen pro Iteration. Somit verfügen sie über mehr Information an das Krümmungsverhalten der Zielfunktion und machen bessere Schritte in die Richtung des lokalen Minimums. Deswegen steigen die Grafiken dieser zwei Verfahren stärker an, wenn wir die in Abbildung 3.6 dargestellten Genauigkeiten pro Iterationen betrachten. Zur besseren Übersichtlichkeit wurden für das PB-LBFGS Verfahren jeweils nur 40 Iterationen bzw. Gradientenauswertungen geplottet.

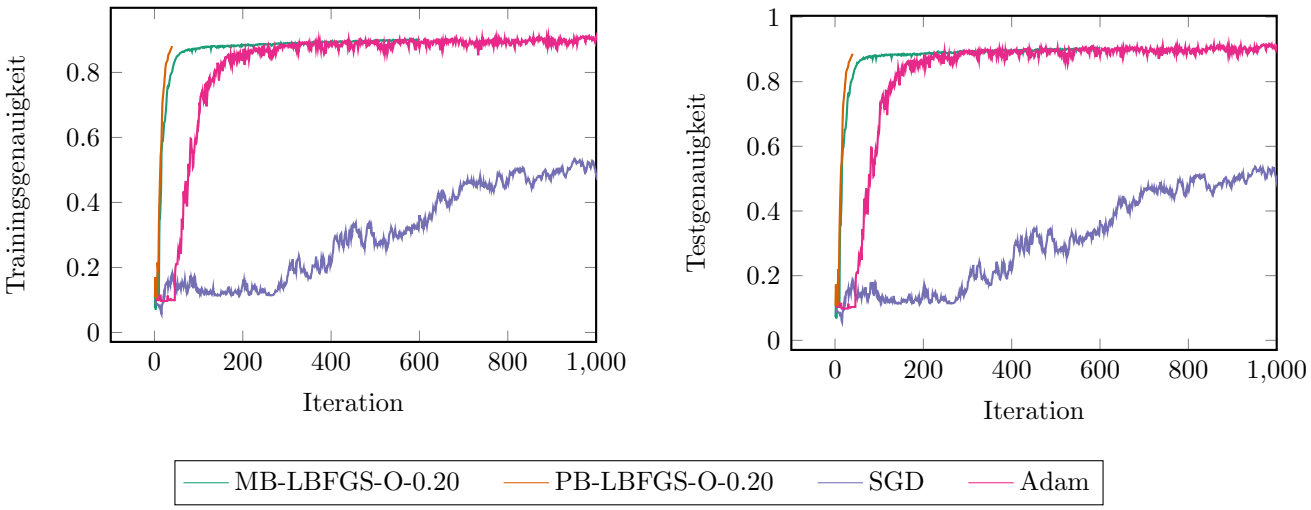

Abbildung 3.6: Vergleich von Trainings- und Testgenauigkeit bezüglich Iterationen

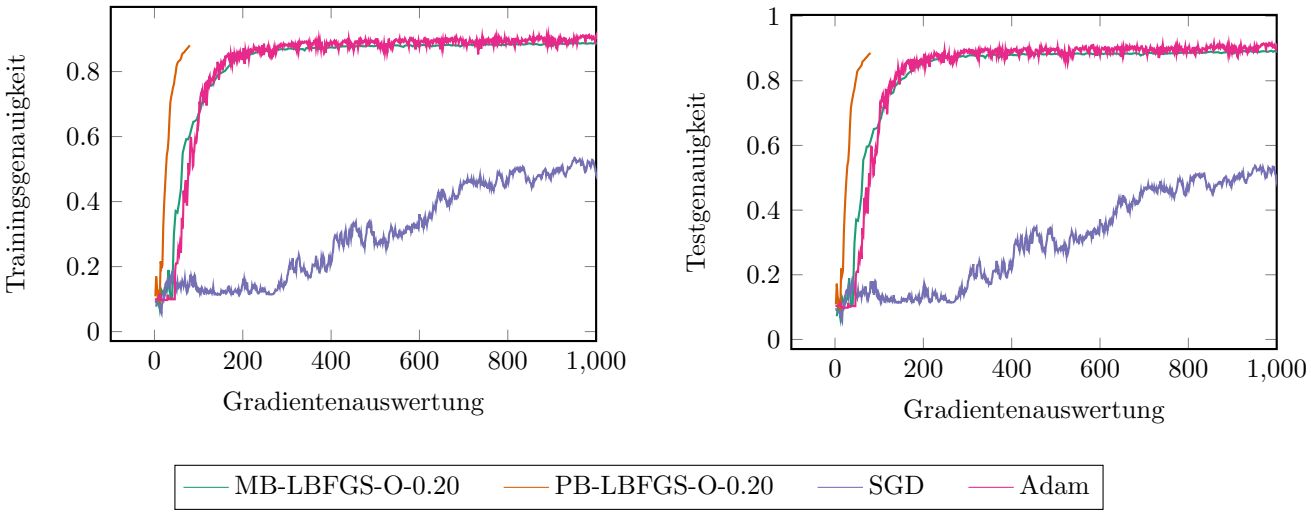

Abbildung 3.7: Vergleich von Trainings- und Testgenauigkeit bezüglich der Anzahl der Gradientenauswertungen

Die Anzahl der Epochen, die ein stochastisches Verfahren durchmacht, hängt von der gewählten Batchgröße ab. Je kleiner die Batchgröße, desto mehr Iterationen braucht ein Verfahren, um durch den gesamten Datensatz durchzugehen und eine Epoche zu vervollständigen. Ein Vergleich der Trainings- und Testgenauigkeiten der Algorithmen bezüglich der Epochen (dargestellt in der Abbildung 3.8) zeigt, dass der Adam Optimierer bereits am Anfang - noch vor der ersten Epoche - fast genauso gute Trainings- und Testgenauigkeit aufweist, wie der MB-LBFGS Optimierer. Der PB-LBFGS und SGD Optimierer sind sogar nach 5 Epochen nicht so gut, wie die Anderen.

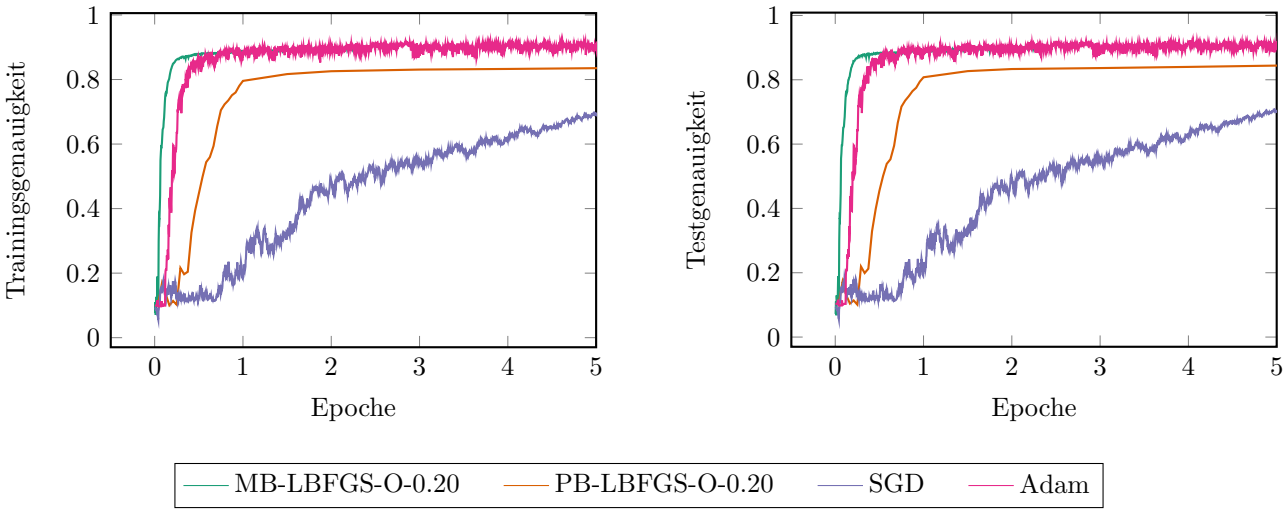

Abbildung 3.8: Vergleich von Trainings- und Testgenauigkeit bezüglich der Anzahl der Epochen

Besonders interessant wird es, wenn wir die Rechenzeit pro Epoche in Betracht nehmen (siehe: Abbildung 3.9). Für eine komplette Epoche benötigt Adam ca. 0.81s mit einer Batchgröße von 128. MB-LBFGS mit der Batchgröße 256 und Überlappungsgröße 20% braucht dafür ca. 4.13s. Insgesamt verlaufen SGD und Adam gleich. Bei MB-LBFGS steigt die Rechenzeit von Epoche zu Epoche etwas stärker, als bei den beiden Verfahren erster Ordnung. Bei PB-LBFGS ist jedoch der Anstieg in der Rechenzeit am stärksten.

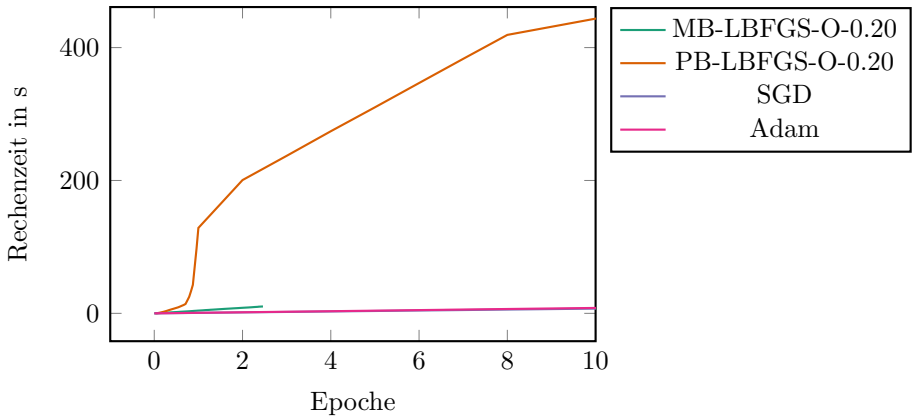

Abbildung 3.9: Vergleich von Rechenzeit der Epochen

Am Anfang wurde für den Adam Optimierer eine Schrittweite von 0.001, wie im Paper [8, S. 2] empfohlen, verwendet. Jedoch, muss hier erwähnt werden, dass bei der Implementierung die Schrittweite konstant gesetzt und nicht adaptiv angepasst wird. Mit dieser Wahl der Schrittweite α performt der Adam Optimierer am Anfang schlechter als der MB-LBFGS. Für die in Abbildung

3.10 dargestellten Ergebnissen wurden SGD und Adam mit Batchgröße 128 und Lernrate 0.001 implementiert. Das MB-LBFGS Verfahren realisiert die Batchgröße 256 mit 30%-ger Überlappung und das PB-LBFGS Verfahren die Batchgröße 512 mit 20%-ger Überlappung. Der Hyperparameter θ beim PB-LBFGS Verfahren ist auf 2.3 gesetzt.

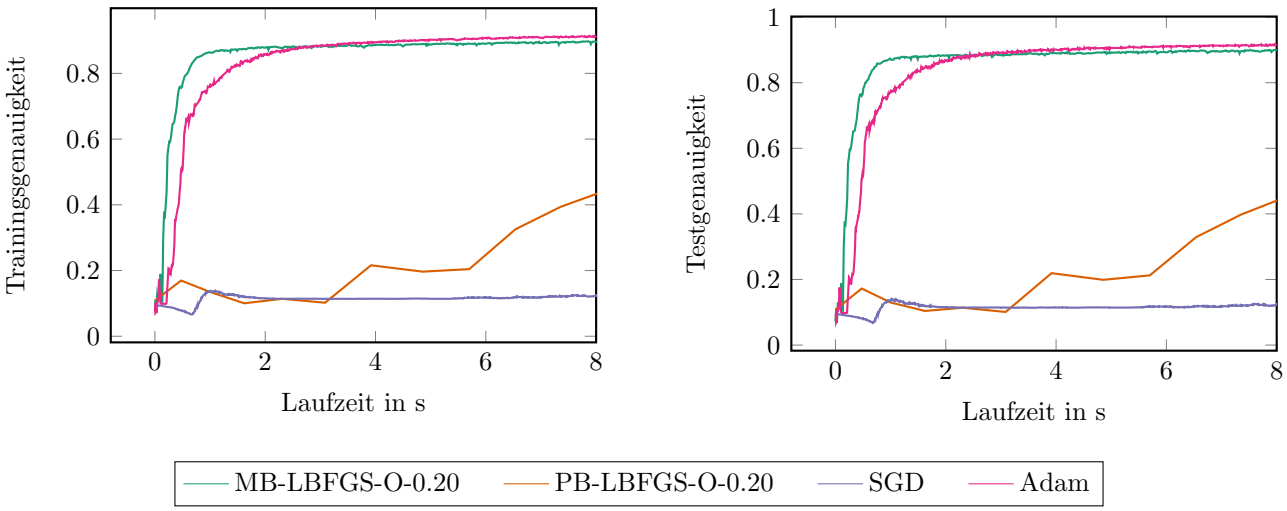

Abbildung 3.10: Vergleich von Trainings- und Testgenauigkeit bezüglich der Laufzeit mit unangepassten (*untuned*) Adam und SGD

Weitere numerische Experimente haben jedoch gezeigt, dass für Adam und SGD eine Schrittweite von 0.02 eine bessere Wahl für das konstruierte neuronale Netz ist[1]. Wie es in der Abbildung 3.11 zu sehen ist, erreicht der Adam Optimierer früher bessere Trainings- und Testgenauigkeit, als die anderen Verfahren. Somit stimmen die numerische Ergebnisse mit den in [1, S. 21] vorgestellten Ergebnissen qualitativ überein.

[1] Genau diese Wahl der Schrittweite wurden in allen Ergebnissen, außer in Abbildung 3.10, realisiert.

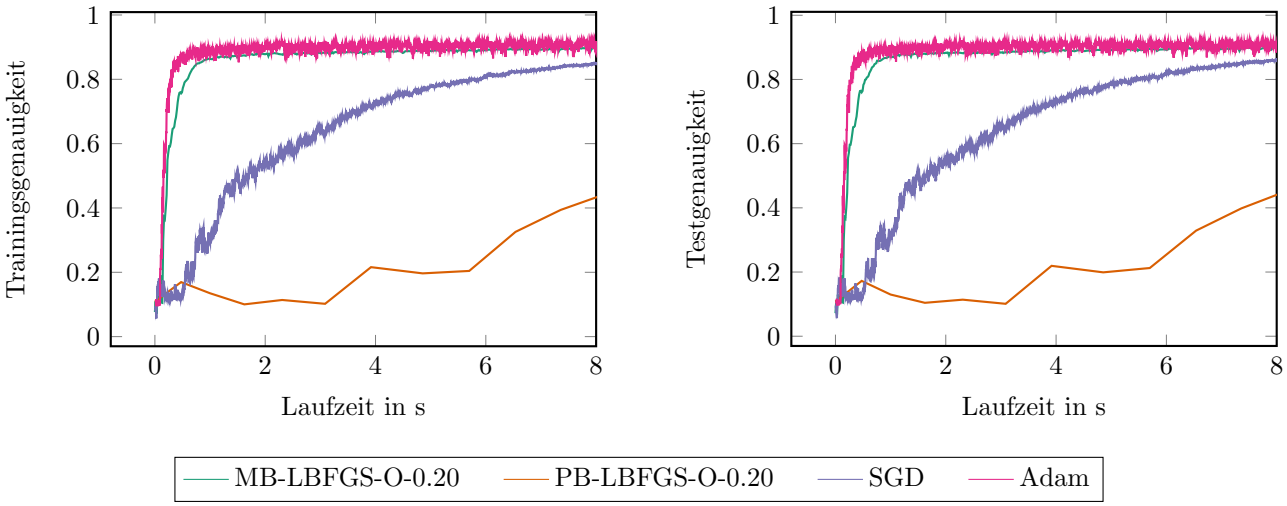

Abbildung 3.11: Vergleich von Trainings- und Testgenauigkeit bezüglich der Laufzeit mit angepassten Adam und SGD

4 Fazit

In dieser Arbeit wurden state-of-the-art numerische Optimierungsalgorithmen auf ein Deep Learning Problem angewendet und die Ergebnisse miteinander verglichen. Das Ziel der Arbeit war es, die Anwendbarkeit der Methoden zweiter Ordnung für die Optimierung von neuronalen Netzen zu testen. Eine der effizientesten Methoden in der Large-Scale Optimierung ist die zu Klasse der Quasi-Newton-Verfahren gehörende L-BFGS Methode [13, S. 223]. Genauer gesagt, wir haben uns für zwei stochastischen Varianten der L-BFGS Methode entschieden: das MB-LBFGS [1] und das PB-LBFGS Verfahren [2]. Beide versuchen mithilfe der Anwendung von stochastischen Gradienten zum Minimum der Zielfunktion zu gelangen. Dafür benötigen sie mehr Rechenaufwand, als die Methoden erster Ordnung, wie das Gradientenabstiegsverfahren [7, S. 39], denn sie versuchen die Information der zweiter Ordnung, d.h. die zweiten partiellen Ableitungen bzw. die Hesse Matrix, zu approximieren.

Die numerische Experimente haben gezeigt, dass auf dieser Weise zwar bessere Suchrichtungen konstruiert werden, aber der zusätzliche Rechenaufwand macht es nicht möglich, die Performance der stochastischen Gradientenverfahren wie Adam [8] zu übertreffen. So benötigt Adam zwar deutlich mehr Iterationen um eine höhere Trainings- und Testgenauigkeit zu erzielen, jedoch ist der Rechenaufwand pro Iteration im Vergleich zu den L-BFGS Versionen so viel geringer, dass sich das vergleichsweise einfache Adam Verfahren eher lohnt. Aus diesem Grund sind in der Optimierung von Deep Learning Problemen mit neuronalen Netzen die Methoden zweiter Ordnung nicht zwangsläufig die beste Wahl.

Man muss aber beachten, dass das konkrete Design des verwendeten neuronalen Netzes und die Wahl der zugehörigen Hyperparameter wichtige Rolle bei der Optimierung spielen. So haben wir im Abschnitt 3.1.2 gesehen, dass für den Adam Optimierer die Schrittweite 0.02 viel bessere Wahl war, als die am Anfang gewählte 0.001.

Für die Zukunft lässt sich jedoch hoffen, dass besser austarierte Versionen des PB-LBFGS Verfahrens, die ähnlich zum Adam Algorithmus die Varianz der Gradienten schätzten, sich durchaus als effiziente Verfahren bewähren werden.

5 Anhang

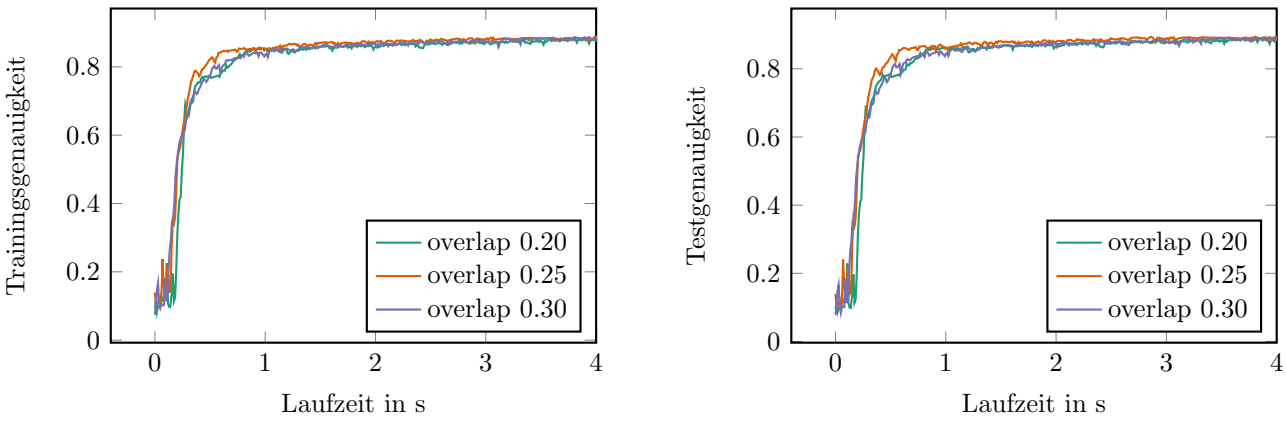

Abbildung 5.1: MB-LBFGS; Auswirkung der Überlappungsgröße auf die Rechenzeit; Batchgröße: 128

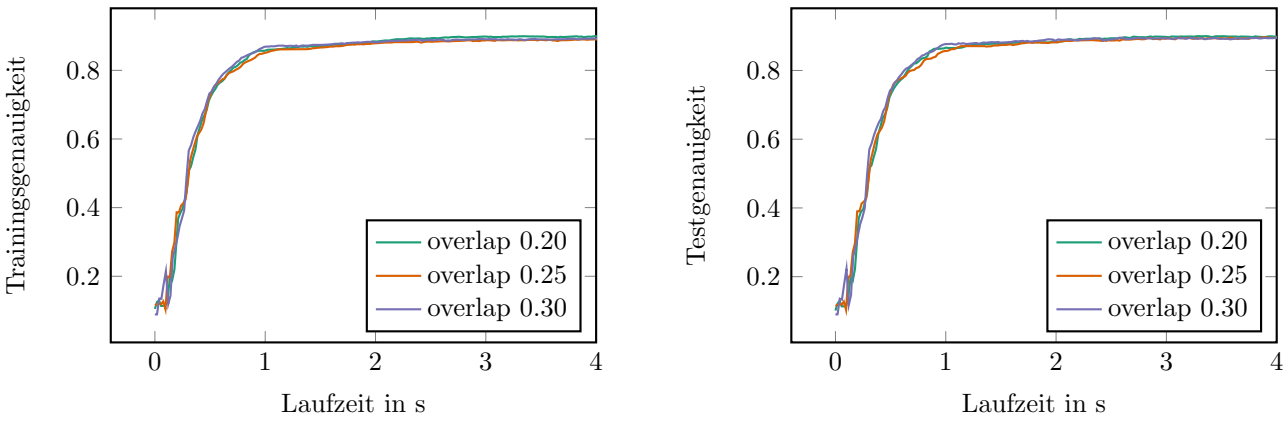

Abbildung 5.2: MB-LBFGS; Auswirkung der Überlappungsgröße auf die Rechenzeit; Batchgröße: 512

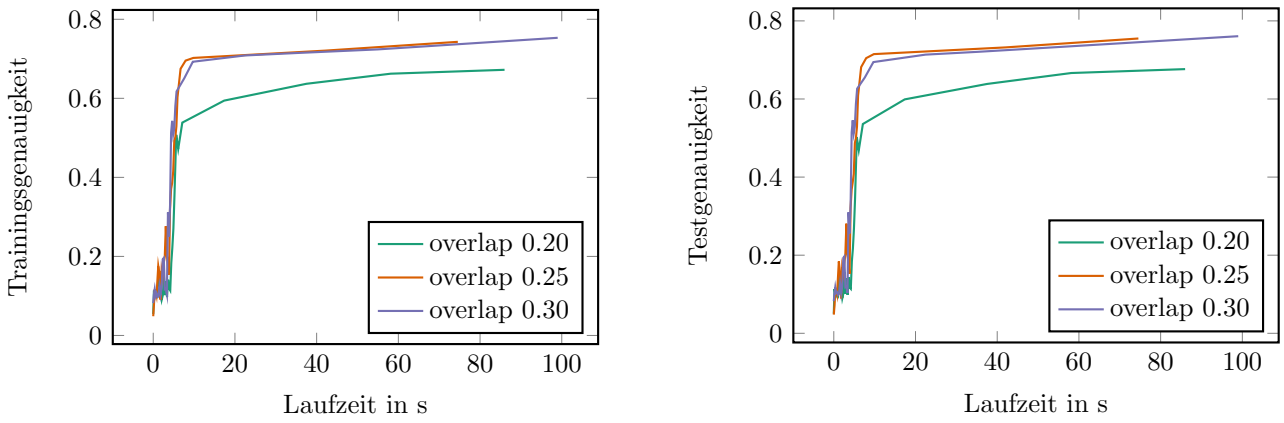

Abbildung 5.3: PB-LBFGS; Auswirkung der Überlappungsgröße auf die Rechenzeit; Batchgröße: 128

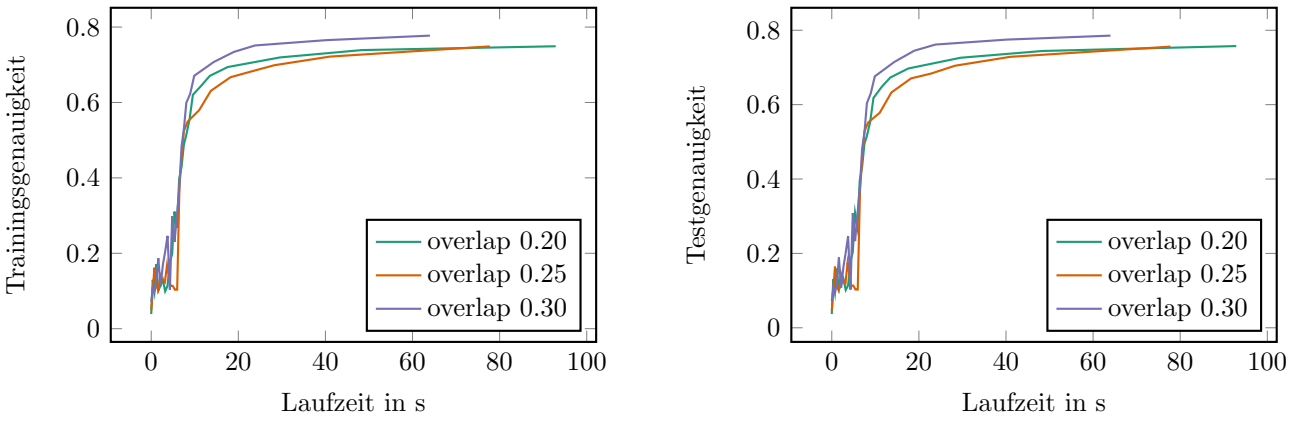

Abbildung 5.4: PB-LBFGS; Auswirkung der Überlappungsgröße auf die Rechenzeit; Batchgröße: 256

Literaturverzeichnis

[1] A. S. Berahas and M. Takáč. A Robust Multi-Batch L-BFGS Method for Machine Learning. ArXiv e-prints, 2017.

[2] R. Bollapragada, D. Mudigere, J. Nocedal, H.-J. M. Shi, and P. T. P. Tang. A Progressive Batching L-BFGS Method for Machine Learning. ArXiv e-prints, 2018.

[3] Richard H. Byrd, Gillian M. Chin, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in unconstrained optimization. SIAM Journal on Optimization, 2011.

[4] Frank E. Curtis and Katya Scheinberg. Optimization Methods for SUpervised Machine Learning: From Linear Models to Deep Learning. ArXiv e-prints, 2017.

[5] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Advances in neural information processing systems, pages 2933–2941, 2014.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. `http://www.deeplearningbook.org`.

[7] C.T. Kelley. Iterative Methods for Optimization. SIAM, 1999.

[8] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. ArXiv e-prints, 2014.

[9] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[10] Tom M. Mitchell. Machine learning. McGraw Hill series in computer science. McGraw-Hill, 1997.

[11] J. J. Moré and D.C. Sorensen. Newton's Method. National Technical Information Service, Argone Illinois 60439, 1982.

[12] Chi Nhan Nguyen and Oliver Zeigermann. Machine Learning kurz und gut. O'Reilly, 2018. https://ebookcentral.proquest.com/lib/ubkoeln/detail.action?docID=5355229.

[13] J. Nocedal and S. J. Wright. Numerical Optimization. Springer, New York, 2nd edition, 2006.

[14] C. Moewes R. Kruse. Computational Intelligence - A Methodologilcal Introduction. Springer-Verlag, 2013.

[15] S. Ruder. An overview of gradient descent optimization algorithms. ArXiv e-prints, 2016.

[16] Patrice Y Simard, Dave Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. IEEE, 2003.

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäßaus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Ort, Datum Unterschrift