UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Varianten von DPSGD

Variants of DPSGD

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Marcel Pflaeging

ausgegeben und betreut von
Prof. Dr. Esfandiar Mohammadi

Lübeck, den 28. April 2023

Zusammenfassung

Da in der heutigen Zeit viele auf großen Datensätze trainierte Algorithmen wie Neuronale Netze zum Lösen von Problemen verwendet werden, ist es wichtig, diese Trainingsdaten zu schützen. Dabei reicht es nicht, diese geheim zu halten, da es möglich ist, Daten aus dem fertig trainierten Netzwerk zu extrahieren. Differential Privacy (DP) ist ein wohlbekannter und beweisbarer Standard, um den Schutz dieser Daten zu garantieren. DPSGD ist eine auf Differential Privacy erfüllende Version des Trainingsalgorithmus Stochastic Gradient Descent (SGD). Dabei werden erst alle Datenpunkte anhand einer Clipping-Bound in ihrem Einfluss auf das Update beschränkt und schließlich Noise, der mit dem größtmöglichen Einfluss eines Datenpunktes skaliert ist, hinzugefügt. Dieser Ansatz sorgt für eine Verschlechterung des Trainings, wobei dies oft an zu viel Noise liegt.
Das Hauptziel dieser Arbeit ist es, zu prüfen, inwiefern die Möglichkeiten von DPSGDs Clipping-Bound bisher ausgenutzt werden. Dafür werden zwei Ansätze vorgestellt, um vermeintlich effektivere Updates auf Basis des DPSGD-Algorithmus zu finden. Der erste Ansatz versucht durch Sortierung Batches mit Gradienten gleicher Länge zu finden, um dann anhand dieser Längen die Clipping-Bound zu wählen. Dabei kann ohne Betrachtung von Differential Privacy gezeigt werden, dass die Sortierung einen positiven Einfluss auf das Training hat. Dies kann durch eine an den Gradientenlängen orientierte Clipping-Bound weiter verbessert werden. Diese Arbeit stellt hierbei keine fertige DP-Implementierung des Algorithmus vor, sondern soll zukünftige Forschung in diese Richtung motivieren.
Außerdem wurde ein an DPSGD orientierter Algorithmus entworfen, der frühzeitiger auf Veränderungen der Loss-Landschaft reagiert, jedoch nicht DP erfüllt. Mit diesem wurde untersucht, ob DPSGD innerhalb der Clipping-Bound optimale Updateschritte unternimmt. Die vorliegende Arbeit zeigt, dass der neue Algorithmus keinen Vorteil gegenüber DPSGD hat.

Abstract

As many algorithms trained on large data sets, such as neural networks, are used to solve problems in today's world, it is important to protect this training data. Keeping it secret is not enough, as it is possible to extract data from the final trained network. Differential Privacy (DP) is a well-known and provable standard for protecting this data. DPSGD is a version of the Stochastic Gradient Descent (SGD) training algorithm that satisfies Differential Privacy. It first constrains the influence of all data points on the update using a clipping bound and then adds noise scaled by the largest possible influence of a data point. This approach leads to training degradation, often due to too much noise.
The main goal of this paper is to investigate to what extent the possibilities of the clipping bound of DPSGD have been exploited so far. For this purpose, two approaches to finding supposedly more effective updates based on the DPSGD algorithm are presented.
The first approach tries to find batches with gradients of the same length by sorting, and then selects the clipping bound based on these lengths. Without considering differential privacy, it can be shown that sorting has a positive impact on training. This can be further improved by a clipping bound based on gradient lengths. This work does not present a ready DP implementation of the algorithm, but is intended to motivate future research in this direction.
In addition, a DPSGD oriented algorithm was developed that responds earlier to changes in the loss landscape, but does not satisfy DP. This was used to investigate whether DPSGD takes optimal update steps within the clipping bound. The present work shows that the new algorithm has no advantage over DPSGD.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 28. April 2023

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1 Einleitung

Künstliche Intelligenz, kurz KI, hat in den letzten Jahren in vielen Bereichen des öffentlichen Lebens Einzug gehalten. Als KI werden dabei Algorithmen bezeichnet, die anhand von im Training erlernten Parametern neue Informationen verarbeiten können. Diese Algorithmen eignen sich für sehr viele Anwendungen wie Bild- und Spracherkennung, autonome Fortbewegung oder Roboterinteraktion. Insbesondere die Medizin profitiert von KI als Werkzeug.
Eines der erfolgreichsten Optimierungsverfahren zum Training von KI ist Stochastic Gradient Descent, kurz SGD.
Für das Training dieser Algorithmen werden viele Daten benötigt und große Firmen sammeln deshalb Daten und klassifizieren und analysieren diese. Unter diesen gesammelten Daten sind gerade im medizinischen Bereich hochgradig sensible Informationen. Daher schränken berechtigte Datenschutzbedenken die Entwicklung und Veröffentlichung solche sehr leistungsfähigen Systeme ein. Es wurde allerdings gezeigt, dass Privacy-Angriffe schon anhand des veröffentlichten Modells Trainingsdaten extrahieren können.
In einer Reihe von Arbeiten wurde außerdem gezeigt, wie sich solche Privacy-Angriffe verhindern lassen. In einer der Arbeiten wurde Differential Privacy vorgestellt, welches inzwischen ein gängiges Maß für beweisbare Privacy-Versprechen ist. Auf Differential Privacy aufbauend wurde eine Annäherung des SGD-Algorithmus namens DPSGD entwickelt.
SGD bestimmt für jeden Datenpunkt seine Richtungsvektoren, die sogenannten Gradienten. Diese zeigen die Änderungen des Systems an, um das Ergebnis zu optimieren. Diese Gradienten werden gemittelt und das System verändert seine Position bezüglich dieses Updateschrittes.
DPSGD baut beim Schutz gegen Privacy-Angriffe auf zwei Konzepte auf. Zum einen wird das Training durch Addition von vorsichtig gewählten Zufallswerten, dem Noise, gestört. Zum anderen wird der Einfluss eines einzelnen Datenpunkts mittels eines Beschränkungsradiuses, auch Clipping-Bound genannt, beschränkt. Dabei wird der Noise anhand der Clipping-Bound skaliert, sodass dieser in etwa den gleichen Einfluss wie ein Datenpunkt hat. Durch diese Maßnahmen wird jedoch das Training verschlechtert.
Vorherige Untersuchungen haben gezeigt, dass die Beschränkung mittels der Clipping-Bound dabei die größte Schwachstelle ist, denn es treten folgende relevante Probleme auf. Einerseits lohnt es sich aufgrund der Skalierung des Noises mit der Clipping-Bound nur

dann diese groß zu wählen, wenn auch die Gradienten entsprechend groß sind. Andererseits ist das Beschränken in DPSGD nicht sehr geschickt, wodurch das gewählte Update nicht optimal ist.

Im Rahmen dieser Arbeit untersuche ich zwei Hypothesen, mit denen vielleicht die beiden oben genannten Probleme von DPSGD zu lösen sind.

In der ersten Hypothese (im Folgenden H1) wird davon ausgegangen, dass es DPSGD effizienter machen kann, wenn man die Datenpunkte eines Batches mittels Sortierung der Gradienten nach ihrer Länge wählt.

Die zweite Hypothese (im Folgenden H2) besagt, dass innerhalb der Clipping-Bound effektivere Updates als von DPSGD gefunden werden können.

Beitrag meiner Arbeit zur Klärung der Hypothesen:
Zur Untersuchung von H1 wurden die Datenpunkte nach der Länge ihrer Gradienten bezüglich der aktuellen Position des Systems sortiert. Aus diesen wurden die längsten gewählt und zur Berechnung des nächsten Updateschrittes verwendet. Zusätzlich wurde untersucht, ob eine Anpassung der Clipping-Bound an die Gradientenlängen des Updateschritts zu effizienteren Updates ohne zu starken Noise führt. Die Sortierung und Anpassung der Clipping-Bound wurden unter optimalen Bedingungen untersucht, die keine Differential Privacy erfüllen. Dabei kamen eine perfekte Sortierung nach Länge und eine Datenpunkt abhängige Wahl der dynamischen Clipping-Bound zum Einsatz. Die Ergebnisse zeigen, dass Sortieren mit Beibehaltung der Clipping-Bound geringe positiven Auswirkungen auf das Training hat. Dies wird durch eine dynamische Anpassung der Clipping-Bound gesteigert. Dabei sind die Privacy-Kosten jedoch zu hoch für eine Nutzung in der Praxis.

Für H2 habe ich ein neues Verfahren entwickelt, um möglichst effektive Updateschritte innerhalb der Clipping-Bound zu finden. Dieses Verfahren ermittelt mittels eines virtuellen Zwischenschrittes die optimale Richtung des Updates. Dabei soll geprüft werden, inwiefern die neuen Updateschritte das Training beeinflussen, und sich zu DPSGD unterscheiden. Aus den Ergebnissen ist abzuleiten, dass das neue Verfahren die Effizienz des Trainings im Vergleich zu DPSGD nicht steigert. Die Untersuchung zeigt außerdem, dass die neu gewählten Update-Schritte nahezu identisch mit den von DPSGD gewählten sind. Daraus wird geschlossen, dass DPSGD schon eine sehr gute Position innerhalb der Clipping-Bound ermittelt.

1.1 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen von Differential Privacy sowie dem Algorithmus DPSGD erklärt. Darauf aufbauend folgt in Kapitel 3 die Problemstellung die in der vorliegenden Arbeit untersucht wird. Kapitel 4 stellt den in H1 formulierten Ansatz dar und untersucht dabei, inwiefern sich das Training verändert. Kapitel 5 untersucht H2 und stellt dabei die erreichten Erkenntnisse dar. Kapitel 6 gibt einen abschließenden Überblick über das untersuchte Themengebiet. Im Anhang ist das Material zusammengestellt, das die für die Arbeit ausgewählten Abbildungen und Tabellen unterstützt, aber keine neuen Erkenntnisse liefert. Jedem dieser Materialien geht der Buchstabe A voran.

2 Hintergründe

Um die in der Einleitung formulierten Hypothesen zu untersuchen, werden im Folgenden die Grundlagen von KI, der Differential Privacy und des darauf aufbauenden DPSGD erklärt. Des Weiteren werden die verwendeten Tools vorgestellt. Abschließend wird ein Überblick über verwandte Arbeiten gegeben.

2.1 Deep Learning

Als Künstliche Intelligenz bezeichnet man meist sogenannte Neuronale Netze, welche mittels Gewichten, auch Parameter genannt, Zusammenhänge abbilden können. Diese Netze bestehen aus mehreren Schichten mathematischer Funktionen, die Informationen aus dem Input extrahieren. Deep Learning ist eine spezielle Art dieser Netze, bei der besonders viele miteinander verbundene Schichten vorhanden sind. Dabei können verschiedene Arten der Schichten unterschiedliche Funktionen erfüllen und Informationen extrahieren. Die Parameter, welche in den verschiedenen Schichten genutzt vorkommen, werden mittels Trainings auf einem Datensatz bestimmt.

Die verbreitetsten Trainingsarten sind *unsupervised learning* und *supervised learning*. Beim *unsupervised learning* ist das Ziel, Zusammenhänge und Strukturen innerhalb des Datensatzes zu finden. *Supervised learning* arbeitet auf einem Datensatz mit Labels, die Zuordnungen der Datenpunkte zu verschiedenen Klassen bieten. Das Modell soll dabei die Zusammenhänge zwischen den Datenpunkten als Input und den Labels als Output lernen. In den Abläufen dieser Arbeit wurde mittels *supervised learning* trainiert.

Ein im Umfeld Neuronaler Netze sehr bekannter Trainingsalgorithmus ist Mini-Batch Stochastic Gradient Descent, kurz SGD. SGD ist dabei sehr effektiv und das verbreitetste Optimierungsverfahren im Bereich der Neuronalen Netze. Er optimiert ein Netz mit Parametern θ bezüglich einer Loss-Funktion $\mathcal{L}(\theta)$. Eine Lossfunktion stellt den Abstand zwischen der Ausgabe des aktuellen Netzes und den Labels dar. In jedem Trainingsschritt wird dabei ein zufälliger Batch B aus den Trainingsdaten gezogen und dessen gemittelter Gradient durch $\boldsymbol{g}_B = \frac{1}{|B|} \sum_{x \in B} \nabla_\theta \mathcal{L}(\theta, x)$ bestimmt. Daraufhin werden die Parameter θ in Richtung der negierten Gradienten $-\boldsymbol{g}_B$ zu einem lokalen Minimum geupdatet [ACG+].

2.2 Differential Privacy

2.2.1 Differential Privacy

Ein trainiertes Neuronales Netz enthält viele Informationen über die verwendeten Trainingsdaten. Es wurde gezeigt, dass aus einem trainierten Netz Rückschlüsse auf die verwendeten Datenpunkte im Trainingsdatensatz gezogen werden können [SSSS]. Diese Trainingsdaten können aus einem öffentlichen Datensatz sein oder hochgradig sensible Daten aus dem Medizinbereich. Bei einem solchen Datensatz müssen daher einzelne Datenpunkte geschützt sein, sodass nicht auf diese zurückgeschlossen werden kann. Um die in Neuronalen Netzen und anderen datenabhängigen Algorithmen verwendeten Daten zu schützen, wurde Differential Privacy entwickelt. Dies bietet einen starken Standard für ein beweisbares Sicherheitsversprechen [ACG+].
Differential Privacy ist dabei auf benachbarten Datensätzen definiert.

Definition 2.1 (Benachbarte Datensätze [DR]). Zwei Datensätze D_0, D_1 gelten als benachbart, wenn sie sich nur in einem Datenpunkt p unterscheiden. Dabei gilt $\|D_1 - D_2\|_1 = 1$, wobei $\|\cdot\|_1$ die l_1-Distanz beschreibt.

Insbesondere gilt diese Eigenschaft, wenn ein Datenpunkt aus einem von zwei identischen Datensätzen entfernt wurde. Dieses veränderte Element p wird auch Challenge-Element genannt.

Definition 2.2 (Approximate Differential privacy [DR]). Ein randomisierter Mechanismus $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ mit der Domäne $\mathcal{D}$ und Wertebereich $\mathcal{R}$ heißt (ϵ, δ)-differentially private, wenn für jede beliebigen Datensätze $D_0, D_1 \in \mathcal{D}$ und für jede Untermenge von Ergebnissen $\mathcal{S} \subseteq \mathcal{R}$ folgende Gleichung gilt.

$$\Pr[\mathcal{M}(d) \in S] \leq e^{\epsilon} \Pr[\mathcal{M}(d') \in S] + \delta$$

Der Parameter δ ist der originalen Definition hinzugefügt worden und beschreibt die Möglichkeit, dass die ϵ-Differential Privacy mit Wahrscheinlichkeit δ bricht. Es ist wünschenswert, dass $\delta < \frac{1}{|D|}$ gilt.

Definition 2.3 (Pure Differential privacy [DR]). Sei $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ ein randomisierter Mechanismus mit der Domäne $\mathcal{D}$ und Wertebereich $\mathcal{R}$ heißt purely-differentially private, wenn für jede beliebigen Datensätze $D_0, D_1 \in \mathcal{D}$ und für jede Untermenge von Ergebnissen $\mathcal{S} \subseteq \mathcal{R}$ der Mechanismus $(\epsilon, \delta = 0)$-differentially private ist.

Die Idee hinter Differential Privacy ist es, den Einfluss eines Datenpunktes so gering zu halten, dass ein Angreifer nicht entscheiden kann, ob dieser im Training enthalten war

oder nicht. Dies kann auch, wie in Abbildung 2.1, als Kryptographisches Spiel dargestellt werden, bei dem der Angreifer mit einem Orakel kommuniziert. Der Angreifer darf zwei Datensätze D_1, D_2 wählen und schickt diese dem Orakel. Das Orakel macht einen Münzwurf und wählt einen der beiden Datensätze. Auf diesen wird der Mechanismus $\mathcal{M}(D_b)$ angewendet und das Ergebnis dem Angreifer geschickt. Der Angreifer hat gewonnen, wenn er aus dem Ergebnis bestimmen kann, welche der Datensätze D_1, D_2 verwendet wurde. Die Wahrscheinlichkeit für den Angreifer richtig zu raten ist dabei durch e^ϵ beschränkt.

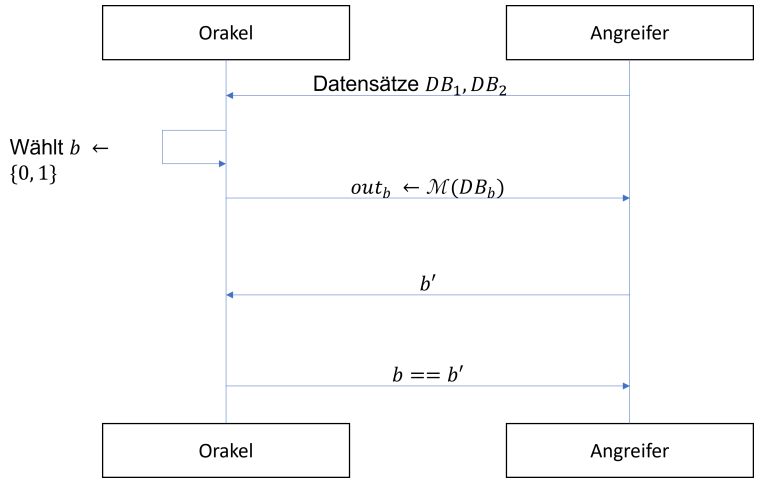

Abbildung 2.1: Zeichnung Differential Privacy als Angreifer-Orakel-Spiel

2.2.2 Differentially Private SGD

Abadi et al. [ACG+] beschreiben in ihrem Paper einen Algorithmus, um das bekannte SGD kompatibel mit Differential Privacy zu machen. Dieser sogenannte DPSGD ist als Algorithmus 1 dargestellt und führt das Training wie folgt durch.
Es werden in jeder Epoche disjunkte Untermengen der Größe L gleichverteilt aus dem Datensatz gewählt. Diese Untermenge bildet einen sogenannten Batch. Für jeden dieser Datenpunkte wird anhand der aktuellen Parameter der Gradient gezogen, indem die Ableitung der Lossfunktion mit den Datenpunkten bestimmt wird. Die Gradienten beschreiben die Steigung der Loss-Funktion und können als Richtungsvektoren für die aktuelle Position des Netzes verstanden werden. Die Gradienten ziehen das Netz dabei in Richtung des geringsten Losses. Die Gradienten werden auf eine gemeinsame Maximallänge gekürzt. Dies geschieht anhand der Clipping-Bound C und wird im Weiteren als Clippen bezeichnet. Durch dieses Clippen haben alle Gradienten gleich viel maximalen Einfluss, welcher abgeschätzt werden kann und genutzt wird, um den Noise zu skalieren. Ohne Clippen

Algorithm 1: Differentially private SGD (Outline)

Input: Examples $\{x_1, ..., x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C.
Initialize θ_0 randomly
for $t \in [T]$ **do**
    Take a random sample L_t with sampling probability L/N
    Compute gradient
    For each $i \in L_t$, compute $\boldsymbol{g}_t(x_i) \leftarrow \nabla\theta_t\mathcal{L}(\theta_t, x_i)$
    Clip gradient
    $\bar{\boldsymbol{g}}_t(x_i) \leftarrow \boldsymbol{g}_t(x_i)/\max(1, \frac{\|g_t(x_i)\|_2}{C})$
    Add noise
    $\tilde{\boldsymbol{g}}_t \leftarrow \frac{1}{L}(\sum_i \bar{\boldsymbol{g}}_t(x_i) + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I}))$
    Descent
    $\theta_{t+1} \leftarrow \theta_t - \eta_t\tilde{\boldsymbol{g}}_t$
Output θ_T and compute the overall privacy cost (ϵ, δ) using a privacy accounting method.

wäre der Einfluss eines Datenpunktes beliebig. Da über den ganzen Batch ein mittlerer Gradient gebildet wird, könnte ohne Clippen ein besonders langer Gradient sehr kurze Gradienten überstimmen und den mittleren Gradienten dadurch dominieren. Durch das Clippen ist der Einfluss jedes einzelnen Datenpunkts durch C begrenzt. Abbildung 2.2 zeigt das Cippen der Gradienten und die resultierenden Mittleren Gradienten abhängig von dem Challenge-Element. Auf den gemittelten Gradienten wird Noise hinzugefügt, der in etwa so viel Einfluss hat, wie ein einzelner Datenpunkt. Dies gelingt durch das Skalieren des Noises auf C. Die Parameter θ werden nun wie in SGD anhand des gemittelten Gradienten angepasst.

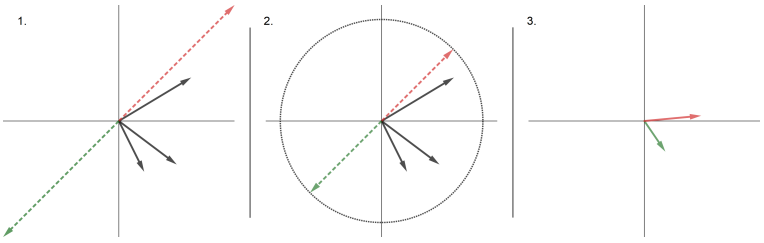

Abbildung 2.2: Darstellung des DPSGD-Algorithmus mittels Clipping und Bildung des Durchschnitts. Die Batchsize beträgt $L = 4$, wobei der Batch drei Gradienten (schwarz und ein Challenge Element enthält (grün oder rot).
1. Berechnung der Gradienten, 2. Clippen der Gradienten anhand des durch de Clipping-Bound C aufgespannten Kreises, 3. Berechnung der Durchschnittsvektoren resultierend aus den Möglichkeiten des Challenge-Elements

Definition 2.4 (Sensitivity [DR]). Die maximale Distanz; die ein Datenpunkt bezüglich des Mechanismus $\mathcal{M}$ verändern kann; ist die *Sensitivity* $S_{\mathcal{M}}$.

Definition 2.5 (L2-Sensitivity [DR]). Die l_2-Sensitivity eines Funktion $f : \mathbb{N}^{|\mathcal{X}|} \to \mathbb{R}^k$ ist wie folgt definiert:

$$\Delta_2(f) = \max_{x,y \in \mathbb{N}^{|\mathcal{X}|}, \|x-y\|_1=1} \|f(x) - f(y)\|_2$$

Definition 2.6 (Gaußscher-Mechanismus [ACG$^+$]). Mit $\mathcal{N}(0, S_f^2 \cdot \sigma^2)$ als Normal- oder Gauß-Verteilung mit Durchschnitt $\mu = 0$ und Standardabweichung $S_f \cdot \sigma$ sowie S_f als die Sensitivität der Funktion f ist der Gaußsche Mechanismus durch

$$\mathcal{M}(x) \triangleq f(x) + \mathcal{N}(0, S_f^2 \cdot \sigma^2)$$

definiert.

Der Gaußsche Mechanismus erreicht DP mit $\delta \geq \frac{4}{5} e^{-\frac{(\sigma\epsilon)^2}{2}} \Leftrightarrow \sigma = \frac{\sqrt{2 \ln \frac{1.25}{\delta}}}{\epsilon}$ [ACG$^+$].
Abadi et al. [ACG$^+$] zeigen und beweisen weiter eine engere Schranke für das Privacy Accounting bezüglich des Algorithmus DPSGD.

Theorem 2.7 (Privacy-Schranke zu DPSGD [ACG$^+$]). *Es existieren zwei Konstanten c_1, c_2, so dass bei gegebener Sampling-Probability $q = \frac{L}{N}$ und der Zahl der Schritte T, sodass für jedes $\epsilon < c_1 q^2 T$ und $\delta > 0$, DPSGD (ϵ, δ)-differentially private gilt:*

$$\sigma \geq c_2 \frac{q\sqrt{T \log(\frac{1}{\delta})}}{\epsilon}$$

Der Beweis ist von Abadi et al. [ACG$^+$] in ihrem Paper gegeben.

2.3 Adaptive Learningrate

Über die Jahre haben sich verschiedene Techniken zur Verbesserung des Trainings entwickelt. Dabei ist es im Kontext des Trainings von Neuronalen Netzen üblich, keine feste Learningrate zu verwenden. Eine solche erschwert es, dass das Training konvergiert. Stattdessen eignet sich besser eine Learningrate, die zu Beginn hoch bleibt und gegen Ende des Trainings schnell abfallend ist. Diese Art der Learningrate verbessert das Training im Gegensatz zu einer fixierten Variante insbesondere in den späteren Epochen zusätzlich. Für das Training in dieser Arbeit wurde die Learningrate basierend auf einer Cosinusfunktion angepasst. Dabei startet sie bei der initial gegebenen Learningrate und nimmt zur letzten Epoche hin entsprechend ab.

$$\mathrm{lr}_{dyn}(E) = \mathrm{lr}_0 * 0.5 * (1 + \cos(\pi * \frac{E}{E_{max}})) \tag{2.1}$$

Die Berechnung entspricht dabei Gleichung (2.1). Dabei ist E der aktuelle Index der Epoche, lr_0 die initiale Learningrate und E_{max} die Anzahl an Epochen des Trainings. Zusätzlich sind in Abbildung A.1 beispielhafte Verläufe für verschiedene initiale Learningrates über 30 Epochen dargestellt.

2.4 Verwendete Tools

Der verwendete Code wurde in Python geschrieben. Dabei kam insbesondere das Machine Learning Framework *Pytorch* [PGM+] zum Einsatz. Pytorch ist eine Bibliothek die viele Funktionen zum Erstellen und Verändern Neuronaler Netze bietet. Insbesondere Matrixoperationen und Gradientenberechnungen sind hier schnell und einfach möglich. Um die neuen Implementierungen mit aktuellen DPSGD-Implementierungen zu vergleichen, wurde das auf *Pytorch* basierende DPSGD-Framework *Opacus* genutzt. [YSS+]. Dieses bietet optimierte DPSGD-Berechnungen durch geringe Veränderung bestehenden *Pytorch*-Codes.

2.5 Toolbox

Für die Experimente wurde von mir eine Sammlung an Werkzeugen geschrieben, die das Training und die Auswertung der Neuronalen Netze erleichtert. Unter anderem können damit Netze geklont werden und diese mit verschiedenen Algorithmen trainiert werden. Dadurch kann man sehen, wo genau sich die Algorithmen im Trainingverlauf unterscheiden.
Der verwendete Code ist auf
`https://git.its.uni-luebeck.de/theses/bachelor-marcel_pflaeging-dpsgd`
zu finden.

2.6 Verwandte Arbeiten

Da DPSGD eine geringere Performanz als SGD bietet, wurden bereits verschiedene Ansätze der Optimierung untersucht. Dabei konnten meist gute Fortschritte erreicht werden, die jedoch nicht an die Effektivität von SGD herankamen. Im Folgenden stelle ich einige wichtige Arbeiten und Ideen exemplarisch vor.
Pichapati et al. [PSY+] stellen eine neue Variante des DPSGD-Algorithmus vor, die beweisbar weniger Noise hinzufügt. Dabei wird für jede Dimension mit optimierter Clipping-

Bound geclippt und es wird Noise auf den Gradienten addiert. Da weniger Noise in den Updateschritt einfließt, sind die Updateschritte effektiver.
Sehr vielversprechende Ansätze sind von Yu et al. [YZC+] und Hu et al. [HSW+] vorgestellt worden. Dabei werden die Trainingsparameter des Netzes als eine große Matrix dargestellt und mittels der Low-Rank-Approximation auf eine geringere Dimensionalität transformiert. Die Gradienten werden dabei nur noch auf den kleineren Matrizen und nicht auf den ursprünglichen Parametern berechnet. Die entstehenden Gradienten sind von einer sehr viel geringeren Dimensionalität, was auch den Noise reduziert. Des Weiteren führen die geringeren Dimensionen zu effektiveren Updates.
Andrew et al. [ATMR] stellen ein Verfahren vor, um die Clipping-Bound adaptiv an die Gradientenlängen anzupassen. Dadurch muss diese nicht auf einen festen Wert gewählt sein, wodurch der Noise eher den tatsächlichen Gradientenlängen entspricht und nicht viel zu groß sein kann. Um die Clipping-Bound zu bestimmen, werden Quantil-Werte, die mittels DP-Techniken bestimmt wurden, verwendet.
Fu et al. [FCL] stellen ein Verfahren vor, das auf Simulated Annealing, einem Optimierungsverfahren der lokalen Suche, basiert. Simulated Annealing führt dabei nur Updateschritte durch, wenn dadurch das Netz besser wird. Wenn dies nicht der Fall ist, wird der Schritt nur mit einer bestimmten Wahrscheinlichkeit getätigt. Da die von DPSGD berechneten Updateschritte nicht immer optimal sind, und sich dadurch die Qualität des Netzes wieder reduziert, ist das hilfreich. Durch die Annealing Technik wird ein solcher Schritt in weniger Fällen akzeptiert, wodurch der Weg zum Optimum des Netzes direkter sein soll. Es können durch die Reduzierung der durchgeführten Schritte außerdem geringere Privacy-Kosten erreicht werden.
In vorherigen Arbeiten konnte gezeigt werden, dass die Reduzierung auf einen geringeren Dimensionsraum die Updates auf wesentliche Aspekte begrenzt, sowie den Noise verringert. Auch wurden Techniken zur Anpassung der Clipping-Bound an Gradientenlängen des Batch vorgestellt.
Diese Arbeit wählt einen anderen Ansatz, indem jeder Batch mit möglichst aussagekräftigen und ähnlich langen Gradienten gefüllt wird. Auf diesen Batches soll die Clipping-Bound möglichst ideal gewählt werden.
Weiter wurde in vorherigen Arbeiten betrachtet, dass die von DPSGD berechneten Updateschritte nicht unbedingt optimal sind. In dieser Arbeit wird daher eine alternative Berechnung des Updateschrittes untersucht, um zu prüfen, ob ein effektiverer Weg des Netzes ermöglicht werden kann.

3 Problemstellung

Wie im Kapitel 2.2.2 ausgeführt, werden die Batches für DPSGD zufällig gewählt und die Gradienten auf die Clipping-Bound C geclippt. Dabei sind die Gradienten, auch Richtungsvektoren genannt, meist alle unterschiedlicher Länge. Wenn ein besonders langer Vektor auf ein im Vergleich kleines C geclippt wird, geht dabei auch viel an Information verloren. Dieser Wegfall von Information kann dazu führen, dass das Netz bei gleicher Zahl von Trainingsschritten weniger leistungsfähig ist. Es wird der mittlere Gradient, auch Durchschnittsvektor genannt, gebildet. Zusätzlich wird Noise hinzugefügt, der abhängig vom gewählten C skaliert. Da der Noise über das C skaliert wird, kann dieser dem maximalen Einfluss eines Datenpunktes auf den Durchschnittsvektor entsprechen. Aus diesem Grund hängt die DP-Eigenschaft nicht von dem Wert der Clipping-Bound ab und diese kann beliebig im Training verändert werden. Martin et al. [ACG$^+$] schlagen für diesen Wert den Median über die Gradientenlängen vor. Da die Längem im Batch allerdings stark variieren und der Noise anhand von C skaliert wird, können kleine Gradienten weiterhin zu viel Noise erhalten. Dies sorgt bei einem zu großen C dafür, dass der Noise zu viel Einfluss hat. Ein daraus resultierender Updateschritt bietet daher wenig Informationsgewinn für das Neuronale Netz. Durch den Median wird dieses Problem etwas verringert, da er nicht anfällig für Ausreißer ist.
Die Ergebnisse des Trainings mit DPSGD sind dabei aber deutlich schlechter als mit SGD. Dabei liegt dies zum Großteil an dem Noise, der zu viel in das Update eingeht.
Die in der Einleitung formulierten Hypothesen werden hier nochmal aufgegriffen und spezifiziert.
Die erste Hypothese (H1) geht davon aus, dass für DPSGD effizientere Updateschritte ermöglicht werden, wenn die Batches aus den Datenpunkten mit den längsten Gradientenlängen bestehen. Diese Auswahl kann durch Sortierung nach den Gradientenlängen erreicht werden.
Die Hypothese stützt sich dabei auf die Vermutung, dass bei großem C das Update ebenfalls größer ist und mehr Informationsgehalt bietet. Mit der Sortierung kann gewährleistet werden, dass die Gradienten ähnlich groß sind. So können alle Gradienten mehr Informationen einbringen, ohne dass ein Gradient den Durchschnittsvektor zu sehr beeinflusst und kleinere überstimmt. Weiter kann die Clipping-Bound C kann bei gleichen Längen innerhalb eines Batches gut angepasst werden, sodass der Noise nicht zu hoch skaliert und das Ergebnis zu stark beeinträchtigt wird.

Die Batches für das Training geeignet zu wählen, wurde bisher in der Forschung wenig betrachtet. Dabei ist Sortierung zur Auswahl geeignet, um zum einen ähnlich große, als auch möglichst lange Gradienten zu erhalten. Jedoch kann Sortierung nicht direkt mit Rücksicht auf Differential Privacy durchgeführt werden. Daher verspricht ein positives Ergebnis der Hypothese einen neuen Ansatz für weitere Forschung, jedoch noch keine fertige Differential Privacy erhaltende Implementierung.
Die zweite Hypothese (H2) beschreibt, dass innerhalb der Clipping-Bound, d. h. der möglichen Aktualisierungen, effizientere Aktualisierungen als die von DPSGD gewählten gefunden werden können.
Da die Clipping-Bound die Gradienten kürzt, sind besonders große Gradienten weniger aussagekräftig als zuvor. Daher kann das Update weniger stark ausgeprägt sein. Betrachtet man ein ähnliches Update, das bereits auf die kommenden Veränderungen reagiert, sollte ein sich dabei die Effektivität erhöhen. Damit könnte gezeigt werden, dass die Berechnung des DPSGD-Updateschritts optimiert werden sollte, sodass ähnliche Vektoren betrachtet werden. Gilt dies nicht, bedeutet es, dass DPSGD schon bestmögliche Updates vorschlägt und Optimierungen nicht in ähnliche Update gefunden werden.
Können die Hypothesen nachgewiesen werden, so wird ein effektiveres Update für DPSGD ermöglicht. Dabei ist das Hauptziel der Forschung in diesem Bereich, den Unterschied in der Performanz zwischen SGD und DPSGD möglichst gering werden zu lassen, um starke Neuronale Netze zu ermöglichen ohne Abzüge an Privacy zu machen.

4 SortedDPSGD

In diesem Kapitel untersuche ich die Hypothese H1 und werde nach der Beschreibung der Versuchsdurchführung zeigen, dass SortedDPSGD im Vergleich zu den Privacy-Kosten einen zu geringen Vorteil gegenüber DPSGD erreicht. Daran anschließend stelle ich weitere Forschungsansätze für SortedDPSGD vor.

4.1 Idee

Wie in Kapitel 3 erläutert, liegt das Hauptproblem von DPSGD in der Wahl der Clipping-Bound, da diese entweder einen Informationsverlust oder zu großen Einfluss von Noise nach sich zieht. Um dies zu verbessern, wird Folgendes angenommen:

Annahme 1 (Batchzusammenstellung). *Werden die Gradienten ihrer Länge entsprechend sortiert, kann man durch Wahl der längsten Gradienten einen Batch mit ähnlich langen Gradienten zusammenstellen.*

Annahme 2 (Größte Gradienten). *Zur Wahl eines Batches sind die längsten Gradienten besonders effektiv, da diese besonders relevante und dringende Updates enthalten.*

Annahme 3 (Dynamische Clipping-Bound). *Wird die Clipping-Bound C an die Gradientenlängen innerhalb eines Batches angepasst, so führt das zu einem erfolgreicheren Training. Dies spiegelt sich in einem kleineren Loss nach Ende des Trainings wider.*

Da der Noise mit C skaliert wird, hat dieser so die gleiche Länge wie ein Datenpunkt im Batch und kann nicht zu viel Einfluss nehmen.
Um das zu gewährleisten, wurde anstatt der zufälligen Wahl eines Batches eine Vorsortierung mit Wahl der längsten Gradienten für den Batch in das Training eingebaut. Dabei werden zunächst für alle Samples die Längen ihrer Gradienten beim aktuellen Zustand des Netzes bestimmt. Diese werden dann der Länge nach sortiert. Aus dieser Sortierung werden die längsten Gradienten gewählt.
Zum Verbessern der Clipping-Bound schlagen Martin et al. [ACG$^+$] eine Learningrate vor, die sich beispielsweise an dem Median der Längen im Batch orientiert. Da durch die Sortierung die Batches jeweils sehr ähnliche Gradientenlängen enthalten, kann eine so gewählte Clipping-Bound zu effektiveren Updates durch weniger Noise führen.

4.2 Motivation

Um H1 zu stützen, wurden zunächst erste Tests durchgeführt. Dafür wurden zwei identische Netze A und B des Modells *MNIST_Net* (vergleiche A.1) erzeugt. Diese beiden Netze wurden mittels DPSGD auf dem Datenset MNIST [Den] trainiert. MNIST, das hochdimensionale Daten in Form von 28 × 28 Pixel großen graustufigen Bildern bereitstellt, wird oft für Benchmarks genutzt. Es enthält 60000 Trainingsdatenpunkte sowie 10000 Testdatenpunkte. Die enthaltenen Bilder sind handschriftliche Ziffern von 0 bis 9, weshalb die Daten in 10 Klassen unterteilt werden. Ein entsprechendes Netz soll die verschiedenen Ziffern unterscheiden und eine abgebildete Ziffer korrekt klassifizieren. Es existieren viele Vergleichswerte für evaluierte Netze und es ist trotz seiner hochdimensionalen Daten gut für Neuronale Netze lernbar.

Im Training waren die Hyperparameter wie folgt gesetzt:
Es wird eine Batchsize $B = \sqrt{N}$ vorgeschlagen [ACG+]. Für MNIST mit $N = 60000$ folgt $BS = \sqrt{60000} \approx 250$. Für die Learningrate wurde die bereits vorgestellte mittels Cosinus adaptive Variante verwendet. Zum Training wurden 30 Epochen verwendet, da nach diesen DPSGD nur noch sehr geringe Steigerung erreicht. Für DPSGD wurde der Noise-Multiplier auf den Wert 1.1 gesetzt, der von der Opacus-Implementierung vorgeschlagen wird [YSS+].
Als Delta wird vom Originalpaper [ACG+] $\delta = 10^{-5}$ vorgeschlagen. Das aus $\sigma = 1.1$ und $\delta = 10^{-5}$ resultierende ϵ beträgt 1.769.

Bei dem Netz A wurden die Batches zufällig gezogen, während bei Netz B die Batches durch Sortierung der Gradienten gewählt wurden. Die Funktion topk wählt dabei die k Datenpunkte mit den derzeit längsten Gradienten. Diese Funktion ist nicht DP. Im Training wurde die Clipping-Bound dynamisch abhängig von dem Median der Gradientenlänge des Batches angepasst. Dies ist ebenfalls nicht DP, gibt aber eine gute Abschätzung für den optimalen Fall. Zeigt sich der Ansatz für non-DP als ungeeignet, hat er auch keinen Vorteil in DP-Settings. Da bei DPSGD Noise hinzugefügt wird und dieser einen Lauf mehr oder weniger beeinflussen kann, wurden mehrere Läufe betrachtet und deren Ergebnisse gemittelt.

Abbildung 4.1 zeigt den Verlauf des Trainings beider Algorithmen mit einer festgesetzten Clipping-Bound.

Dabei fällt auf, dass DPSGD und SortedDPSGD bei der geringsten Learningrate von 0.01 einen fast identischen Verlauf des Losses haben, jedoch die Accuracy sehr voneinander abweicht. Die SortedDPSGD-Variante mit Learningrate 0.1 hat in der ersten Epoche einen stark angestiegenen Loss, der daraufhin wieder abfällt. Dabei sinkt der Loss weiter ab als bei DPSGD. Auch die letztendliche Accuracy ist besser als bei DPSGD. Ähnlich ist es auch

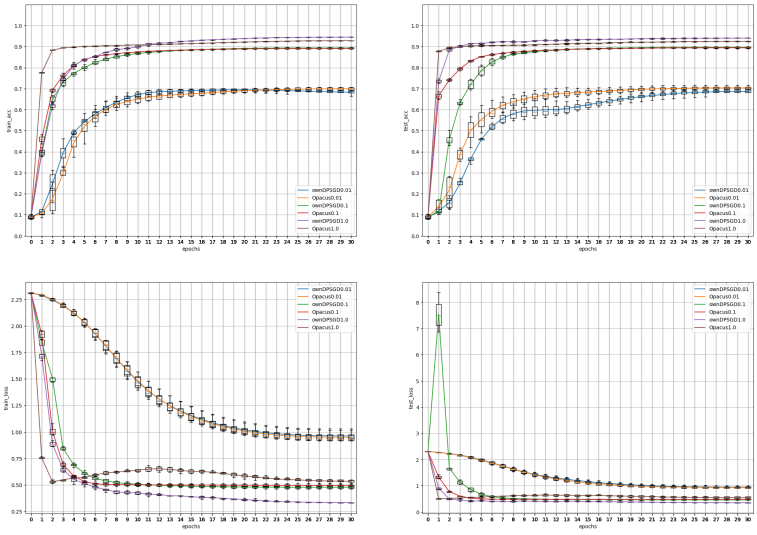

Abbildung 4.1: Verläufe von DPSGD mit $C = 1$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 30 Epochen auf dem MNIST-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.769$

bei der höchsten Learningrate 1.0. Dort sinkt bei SortedDPSGD jedoch von Beginn an der Loss rapide und liefert somit bessere Werte als DPSGD.
Die resultierenden Werte sind in Tabelle 4.1 zu sehen.

Tabelle 4.1: Ergebnisse von DPSGD mit $C = 1$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 30 Epochen auf dem MNIST-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.769$

Variante	Loss	Acc
DPSGD0.01	0.93	70.37
DPSGD0.1	0.48	89.46
DPSGD1.0	0.55	92.51
SortedDPSGD0.01	0.95	68.69
SortedDPSGD0.1	0.46	89.69
SortedDPSGD1.0	0.36	94.07

SortedDPSGD liefert sehr ähnliche Accuracy Werte, die bei höherer Learningrate besser sind als von DPSGD. Der Loss ist dabei auch sehr ähnlich.
In der Abbildung 4.2 sieht man die Accuracy und den Loss über den Trainingsverlauf. Trainiert wurde mit dem gewöhnlichen DPSGD-Algorithmus der Opacus-Implementierung [YSS+], sowie DPSGD mit Sortierung und dynamischer Clipping-Bound. Für beide Algorithmen wurden als initiale Learningrates $1.0, 0.1$ und 0.01 gewählt.
Es ist zu sehen, dass bei der geringen Learningrate 0.01 der Anstieg der Trainingsaccu-

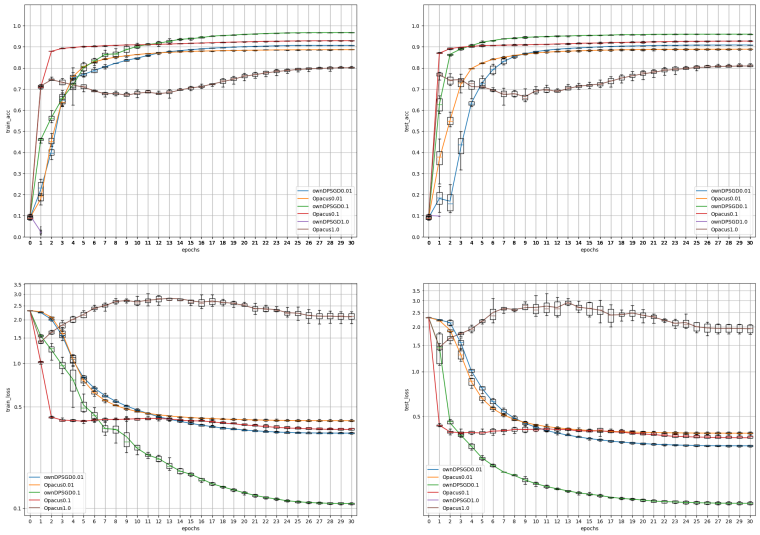

Abbildung 4.2: Verläufe von DPSGD mit $C = 7.5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median über 30 Epochen auf dem MNIST-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.769$

racy von SortedDPSGD erst später einsetzt. Dies spiegelt sich auch im Loss wider, der zu Beginn langsamer sinkt, um dann stärker zu fallen als bei DPSGD. Der Durchlauf mit Learningrate 0.1 hat bei beiden Algorithmen einen ähnlichen Verlauf. Die Accuracy von SortedDPSGD flacht dabei weniger schnell als DPSGD ab und erreicht bereits in der dritten Epoche die Endaccuracy des vergleichbaren DPSGD-Trainings. Bei der größten Learningrate 1.0 zeigt sich, dass das entsprechende DPSGD-Training zwar eine akzeptable Accuracy erreicht, jedoch den Loss wenig reduziert hat. Die SortedDPSGD-Variante (lila) hingegen konnte ab der zweiten Epoche nicht weiter fortgesetzt werden, da das Programm mit einem NaN-Fehler abstürzt.

Tabelle 4.2: Ergebnisse von DPSGD mit $C = 7.5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median über 30 Epochen auf dem MNIST-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.769$

Variante	Loss	Acc
DPSGD0.01	0.39	88.87
DPSGD0.1	0.36	92.71
DPSGD1.0	1.94	80.89
SortedDPSGD0.01	0.32	90.86
SortedDPSGD0.1	0.13	96.0

Wie der Tabelle 4.2 zu entnehmen ist, ist der beste Loss von SortedDPSGD weit unter den Werten von DPSGD. Für DPSGD wurde eine Hyperparametersuche für C durchgeführt.

Dabei hat sich ergeben, dass der Loss und die Accuracy bei Learningrate 0.1 und $C = 7.5$ bestmögliche Werte sind. Es wird also mit SortedDPSGD das bestmögliche Ergebnis von DPSGD übertroffen.

Es ist damit klar, dass sich auf einem einfachen Datenset wie MNIST diese Methode im optimalen Setting ohne DP als wirksam erweist. Daher werden weitere Tests auf einem komplexeren Datensatz durchgeführt.

4.3 Versuchsdurchführung

In der Motivation konnte anhand eines Beispiels gezeigt werden, dass das neue Verfahren einen Vorteil gegenüber DPSGD hat. Da die Datenpunkte aus MNIST [Den] eine simple Struktur haben, wird darauf aufbauend das neue Verfahren in einem schwierigeren Setting getestet.

Es werden zwei identische Netze erzeugt und mittels DPSGD diesmal auf CIFAR10 [KHo] trainiert. CIFAR10 stellt Bilder der Auflösung 32 × 32 Pixel bereit, die jeweils 3 Farbkanäle besitzen. Dieser Datensatz eignet sich gut zur Evaluation von sehr hochdimensionalen Daten. CIFAR10 stellt besonders realistische Daten zur Verfügung. Die verwendeten Bilder sind dabei aus öffentlichen Quellen des Internets bezogen und auf eine geeignete Auflösung herunterskaliert worden.

Als Netz wird das Convolutional Neural Network, kurz CNN, *Conv_CIFAR_Net*, dessen Aufbau in Listing A.2 dargestellt ist, aus der Opacus-Implementierung [con] verwendet. Dies ist ein Standardnetzwerk, das oft zum Vergleich von Benchmarks verwendet wird.

Im weiteren Training waren die Hyperparameter wie folgt gesetzt:
Da CIFAR10 50000 Datenpunkte bereitstellt, wird eine Batchsize von $\sqrt{50000} \approx 200$ verwendet. Die Learningrate ist dabei adaptiv, basierend auf einer Cosinusfunktion. Dabei wurde zwischen den initialen Learningrates $0.01, 0.1$ und 1.0 variiert. Zum Training wurden 30 Epochen verwendet, welche für DPSGD ausreichen, um stabile Werte zu erzielen. Für DPSGD wurde der Noise-Multiplier σ auf den Wert 1.1 gesetzt. Dies ist meistens ausreichend und wird im Standardfall von der Opacus-Implementierung [YSS$^+$] vorgeschlagen. Bei dieser Trainingskonfiguration ergibt sich mit $\delta = 10^{-5}$ der Wert $\epsilon = 1.729$.

Die Auswahl der Batches wird mit zwei Sortierverfahren ohne Betrachtung von DP evaluiert. Des Weiteren wird die Clipping-Bound mit unterschiedlichen Methoden dynamisch angepasst. Es wird zuerst die Sortierung unabhängig von der dynamischen Clipping-Bound untersucht und anschließend die Kombination beider Ansätze.

4.3.1 Die Sortierfunktionen

Es wurden zwei unterschiedliche Sortierfunktionen untersucht. Dabei sollte der optimale und ein etwas geringerer Fall dargestellt werden. Dafür kamen topk-Sortierung und Quantile-Sortierung zum Einsatz.
Topk ist eine vollständige Sortierung, bei der die k Datenpunkte mit den größten Gradienten den Batch füllen. Dies ist ohne Berücksichtigung von DP und zeigt den optimalen Fall einer Sortierung.

Eine weitere Idee war die Sortierung anhand der sogenannten Quantile, um möglichst gleich große Gradienten zu erhalten. Quantilesort ist in Algorithmus 2 dargestellt und berechnet als Threshold die Quantile-Grenzen, also einen Wert unter dem ein bestimmter Prozentanteil aller Längen liegen. In diesem Fall das 0.75-Quantile auch Quartile $Q_{0.75}$ genannt. Werden nun alle Längen bezüglich dieses Thresholds sortiert, können aus dem oberen Quantile Datenpunkte zufällig gezogen werden, um einen Batch zu füllen. Wählt man den Threshold des Quantils nun DP mittels des expQ-Algorithmus [DFM$^+$], so ist dies dennoch nicht DP. Nach der Wahl des Thresholds fallen für jeden Datenpunkt, der mit diesem Threshold verglichen wird, Privacy-Kosten an.

Algorithm 2: Quantilesort

Input: Datenpunkte $D = \{d_1, ..., d_N\}$, Längen der Gradienten $L = \{l_1, ..., l_N\}$, Batchsize L
$T \leftarrow$ quantile$(0.75, L)$ // compute Threshold
$Q \leftarrow \{i \in [1...N] : d_i | l_i >= T\}$ //Compute set with all elements that are above threshold
Use most elements with greater length than Threshold
if $|S| > L$ **then**
    Take a random sample S with sampling probability $\frac{L}{|Q|}$
else
    Take a random sample Q' with sampling probability $\frac{|Q|-L}{N}$
    $S \leftarrow Q \cup Q'$
Output $S \subseteq D$

4.3.2 Sortierung und Dynamische Clipping-Bound

Um einen geeigneten Wert für die Clipping-Bound zu finden, wird sich an den Gradientenlängen innerhalb des Batches orientiert. Die Clipping-Bound hängt dadurch jedoch direkt von den Längen im Batch ab. Dieses Verfahren ist daher nicht DP. Die Clipping-Bound C ist ein zu veröffentlichender Parameter, der Leakage über die längsten Längen

besitzt, d. h. es ist möglich Informationen über die zugrundeliegenden Daten zu extrahieren.
Sind die Batches durch Sortierung gewählt, so sollten ihre Längen ähnlich groß sein. Es kann jedoch auch der Fall auftreten, dass dem nicht so ist. Wenn einige sehr lange Gradienten im Batch sind, können diese den Durchschnitt und damit die Clipping-Bound stark anheben. Ist damit die Clipping-Bound zu groß gewählt, gehen in den Updateschritt insbesondere starker Noise und einige wenige Ausreißer ein. Daher ist der Durchschnitt ungeeignet für die dynamische Anpassung der Clipping-Bound.
Eine Möglichkeit, die dynamische Clipping-Bound zu wählen, ist der Median der Gradientenlängen des Batches. Der Median eignet sich hierbei besser als der Durchschnitt, da der Median sich nicht so leicht durch Ausreißer beeinflussen lässt. Dadurch soll die Clipping-Bound besser den tatsächlichen Gradientenlängen entsprechen.
Des Weiteren wurde als Clipping-Bound das Minimum der Gradientenlängen innerhalb des Batches betrachtet. Dies verspricht, dass alle Gradienten nach dem Clippen dieselbe Länge haben und genau gleich viel in den Durchschnittsvektor eingehen. Es soll damit verhindert werden, dass lange Gradienten die kürzeren überstimmen.

4.4 Ergebnisse

Die hier dargestellten Ergebnisse sollen einen Einblick in das neu vorgestellte Verfahren SortedDPSGD liefern. Dabei wird zuerst auf die durch Sortierung entstandenen Resultate eingegangen. Da unter bestimmten Umständen Abstürze des Programms verzeichnet wurden, folgt eine Untersuchung der Ursache. Daraufhin werden die Experimente mit dynamischer Clipping-Bound vorgestellt und ein Überblick über die DP-erhaltende Variante von SortedDPSGD gegeben. Dabei sind weitere Ergebnisse, die ähnliche Daten enthalten, im Anhang zu finden.
Da für das Sortieren alle Datenpunkte betrachtet werden und das für jeden Batch, ist der Overhead sehr groß. Bei größeren Batchgrößen werden auch weniger Sortierdurchgänge benötigt. Bei Batchgröße 200 erhöhte sich die Laufzeit für eine Epoche um den Faktor 50 bis 75.

4.4.1 Sortierung mit fester Clipping-Bound

Es wurde geprüft, wie sich DPSGD und SGD mit Sortierung und ohne Anpassen der Clipping-Bound verhalten. Dabei wurde die topk-Sortierung verwendet, um den bestmöglichen Fall darzustellen.
Wie in Abbildung 4.3 zu sehen ist, sind die Endergebnisse der Läufe alle sehr ähnlich. Die Variante mit Sortierung ist in allen Durchläufen ca. 2 Prozentpunkte schlechter. Bei der

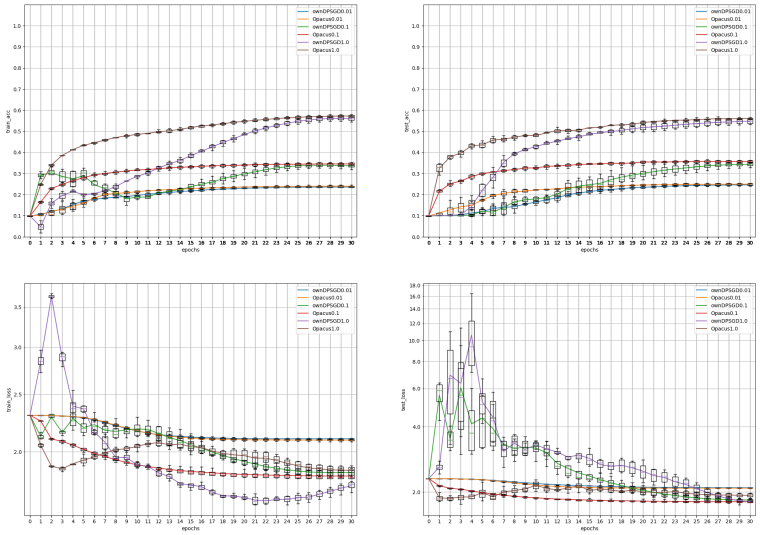

Abbildung 4.3: Verläufe von DPSGD mit $C = 1$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

geringsten Learningrate ist das Endergebnis fast identisch.
Die Varianten mit Sortierung haben in den ersten Epochen nahezu keine Steigerung der Accuracy, während der Loss steigt. Ungefähr ab der 5. Epoche zeigt sich eine Verbesserung. Der hohe Loss sinkt und nähert sich den Werten von DPSGD an.
Es stellt sich im Training heraus, dass sich die Performanz von SortedDPSGD sehr verzögert verbessert. Daher reichen 30 Epochen nicht mehr aus, um den Verlauf geeignet darzustellen. Um die Werte zu vergleichen, werden zusätzlich Durchläufe mit 50 Epochen untersucht.
In Abbildung 4.4 ist zu sehen, dass bei der größeren Clipping-Bound von $C = 5.0$ SortedDPSGD über 50 Epochen eine höhere Accuracy und einen geringeren Loss erreichen kann als DPSGD. Dabei verbessert sich SortedDPSGD weiterhin, während DPSGD seine Werte nahezu konstant hält. Nach den Werten in Tabelle 4.3 ist der Loss von SortedDPSGD bei Learningrate 0.1 sehr viel geringer.
Tabelle 4.4 zeigt die für das *Conv_CIFAR10_Net* mit festgelegter initialer Learningrate beste Clipping-Bound. Diese wurde bezüglich der Minimierung des Losses über eine Hyperparametersuche gefunden.
In Abbildung 4.5 und 4.6 sind die Verläufe auf Learningrate 0.1 mit $C = 9.5$ sowie Learningrate 0.01 und $C = 10$ zu sehen.
In beiden ist der erreichte Loss von SortedDPSGD deutlich niedriger als der von DPSGD. Bei der größeren Learningrate 0.1 hat SortedDPSGD trotz des geringeren Losses eine schwächere Accuracy. Im Beispiel mit der geringeren Learningrate konnte SortedDPSGD

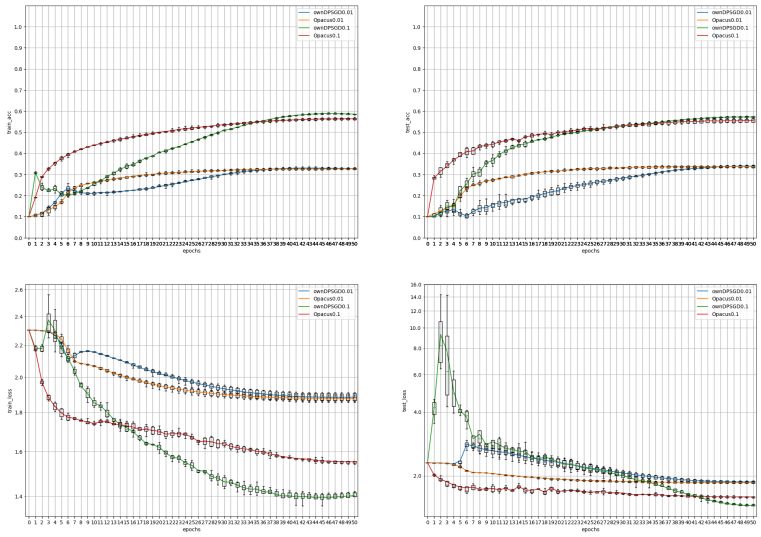

Abbildung 4.4: Verläufe von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

Tabelle 4.3: Ergebnisse von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

Variante	Loss	Acc
DPSGD0.01	1.86	33.78
DPSGD0.1	1.59	55.52
SortedDPSGD0.01	1.87	33.94
SortedDPSGD0.1	1.45	57.24

Tabelle 4.4: Die durch Hyperparametersuche idealen Werte C für eine gegebene initiale Learningrate für DPSDG

LR	C
1.0	1
0.1	9.5
0.01	10

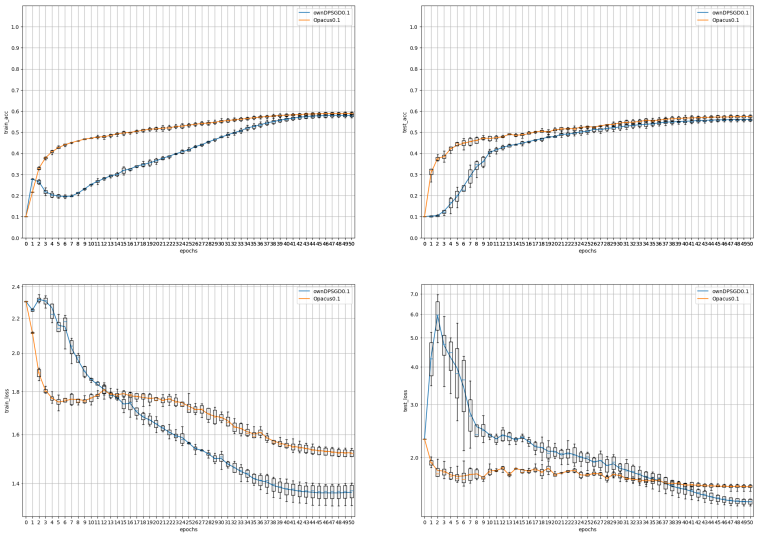

Abbildung 4.5: Verläufe von DPSGD mit $C = 9.5$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

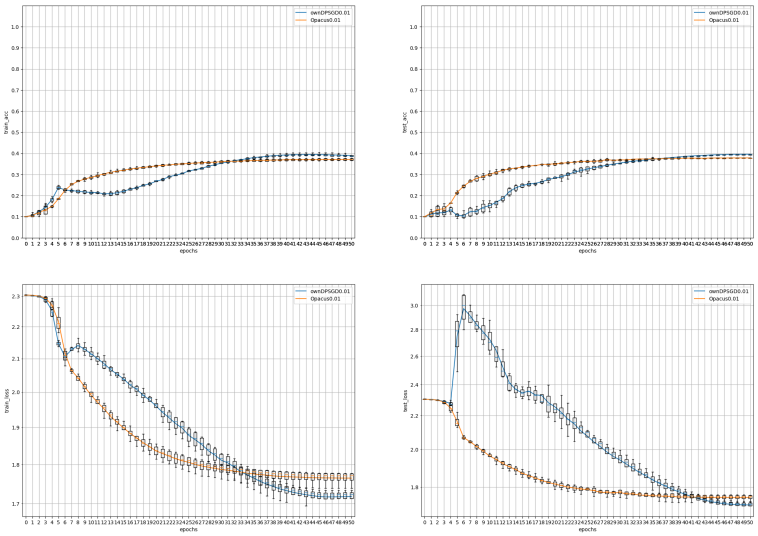

Abbildung 4.6: Verläufe von DPSGD mit $C = 10$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

sowohl Loss als auch Accuracy im Vergleich zu DPSGD verbessern.
Es fällt auf, dass bei den Läufen mit klein gewählter Learningrate DPSGD und die sortierte Variante sehr ähnliche Verläufe und Werte haben. Der plötzliche Anstieg im Loss zu Beginn des Trainings ist auch geringer als bei größerer Learningrate.
Wird nur Sortieren verwendet, so unterscheiden sich die Resultate gering von gewöhnlichem DPSGD. Dies liegt daran, dass jeder Datenpunkt weiterhin mit der fixierten Clipping-Bound geclippt wird. Dadurch können Ausreißer oder Batches mit besonders langen Gradienten keinen größeren Einfluss nehmen. Die sortierende Variante war dennoch in vielen Fällen bezüglich des Losses performanter. Tabelle 4.5 zeigt die Ergebnisse der Sortierung mit fester Clipping-Bound auf dem für die jeweilige Learningrate optimal gewählten Parameter C.

Tabelle 4.5: Ergebnisse von DPSGD mit für die Learningrate idealem C und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Variante	Loss	Acc
DPSGD0.01_C-10	1.85	34.27
DPSGD0.1_C-9.5	1.56	56.09
DPSGD1.0_C-1	1.93	56.04
SortedDPSGD0.01_C-10	1.87	33.77
SortedDPSGD0.1_C-9.5	1.47	53.23
SortedDPSGD1.0_C-1	1.8	54.75

In Abbildung 4.7 ist zu sehen, wie sich die Cosinus-Ähnlichkeit der aufeinanderfolgenden Updateschritte innerhalb der Epochen entwickelt.

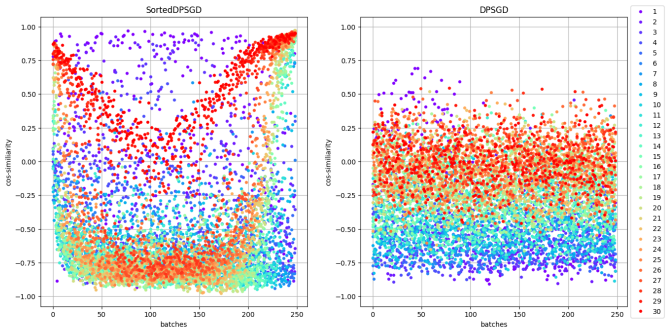

Abbildung 4.7: Die Cosinus-Ähnlichkeiten aufeinanderfolgender Updateschritte in SortedDPSGD mit fester Clipping-Bound $C = 9.5$ und DPSGD Training über 30 Epochen auf dem CIFAR10-Datensatz, Learingrate 0.1

In den ersten Epochen ist die Cosinus-Ähnlichkeit der Updateschritte meist zwischen -1

und 1 alternierend. In den späteren Epochen bildet sich eine Art "Badewannenkurve" heraus, die sehr ähnliche Updateschritte zu Beginn und am Ende der Epoche darstellt. In der Mitte der Epochen widersprechen sich noch fast alle Updateschritte. In der letzten Epoche hat sich die "Badewannenkurve" zu einem V-förmigen Verlauf entwickelt. Dabei sind die Updates über die gesamte Epoche meist ähnlich, nur in der Mitte der Epoche dreht sich die Richtung mehrfach. Dies ist der Wendepunkt, an dem die Ausreißer nicht vorwiegend die Batches bilden.
Bei einer kleineren Learningrate sorgt die Sortierung auch in den ersten Epochen für eine V-Form des Verlaufs. Ab der siebten Epoche ändert sich die Form des Verlaufs zu einer "Badewannenkurve". Dies ist der Wendepunkt, an dem der Loss wieder sinkt. Dies bleibt so bis zu den letzten Epochen. In der letzten Epoche entsteht erneut eine V-Kurve.

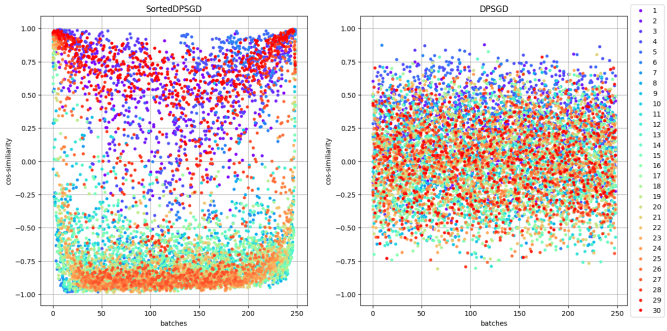

Abbildung 4.8: Die Cosinus-Ähnlichkeiten aufeinanderfolgender Updateschritte in SortedDPSGD mit fester Clipping-Bound $C = 10$ und DPSGD Training über 30 Epochen auf dem CIFAR10-Datensatz, Learingrate 0.01

Interessant ist, dass die Learningrate auch Auswirkungen auf die Cosinus-Ähnlichkeit innerhalb von DPSGD hat. Vergleicht man Abbildung 4.7 mit 4.8, so ist gut zu sehen, dass DPSGD bei geringerer Learningrate eine größere Streuung hat. Dabei sind die Werte in den ersten Epochen meist über 0 und bilden in den späteren Epochen eine Verteilung mit Erwartungswert 0. Bei Learningrate 0.1 hingegen sind die Werte der ersten Epochen eher unter 0 und nähern sich in den späteren Epochen einer Verteilung um die 0 herum an.

4.4.2 Programmabstürze

Wie in Abbildung 4.2 zu sehen ist, stürzte das Programm bei der hohen Learningrate von 1.0 nach der ersten Epoche ab. Dies geschah während des Trainings mit dynamischer Clipping-Bound, die abhängig von den Längen im Batch gewählt wurde.
Um den Absturz zu untersuchen, wurde der SGD-Algorithmus mit Sortierung getestet. Der SGD-Algorithmus lässt sich dabei informell als Variante des DPSGD-Algorithmus

mit Clipping-Bound $C = \infty$ und Noise-Multiplier $\sigma = 0$ verstehen. Auf dem MNIST-Datensatz führte SGD mit Sortierung bei einer hohen Learningrate von 1.0 oftmals zum Absturz.

Auf dem komplexeren Datenset CIFAR10 stürzt SGD mit Sortierung selbst bei der kleineren Learningrate 0.1 regelmäßig ab. Abbildung 4.9 zeigt, dass das Training nur mit Learningrate 0.01 erfolgreich verlief. Deren Loss und die Accuracy haben jedoch keinen sehr regelmäßigen Verlauf.

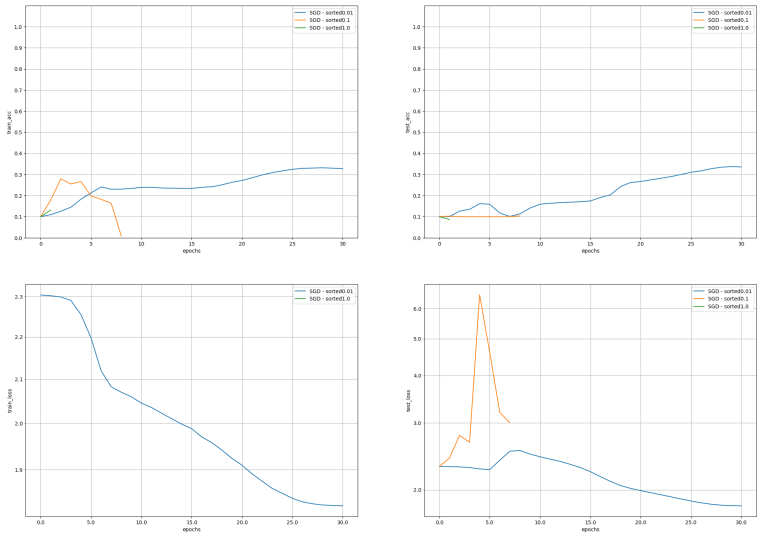

Abbildung 4.9: Verläufe von SGD mit topk-Sortierung über 30 Epochen auf dem CIFAR10-Datensatz

Der Absturz des Programms wird durch einen ein NaN-Fehler erzeugt, der durch zu große Werte bei der Loss Berechnung entstand.

SortedDPSGD hatte nur bei unbeschränkter Clipping-Bound, wie in der Motivation verwendet, Programmabstürze. Dies liegt daran, dass – ähnlich wie bei SGD – mit Sortierung die unbeschränkte Clipping-Bound den Gradienten erlaubt unbegrenzt zu wachsen. Durch den mit der stetig ansteigenden Clipping-Bound skalierten Noise wird das Problem in SortedDPSGD noch verstärkt.

Erklärt werden die Programmabstürze durch die Auswirkungen der Sortierung, die immer die längsten Gradienten wählt. Zu Beginn sind die Parameter zufällig initialisiert. Deshalb haben einige Datenpunkte zufällig die längsten Gradienten. Dabei haben diese Datenpunkte jedoch meist eine ähnliche Klasse, da bei ähnlichen Datenpunkten auch die Gradienten ähnlich sind. Daher haben die Gradienten einer Klasse mit den größten Längen auch den größten Einfluss.

Durch Abbildung 4.10 lässt sich der Ablauf der ersten Epochen nachvollziehen.

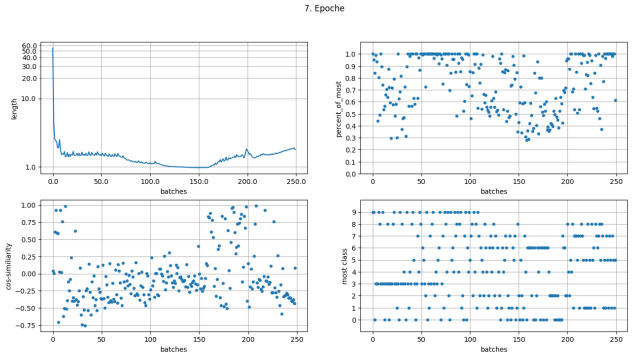

Abbildung 4.10: Die Daten der 7. Epoche des in Epoche 8 abgestürzten SGD-Trainings mit topk-Sortierung, Learningrate 0.1. Dargestellt sind die mittlere Gradientenlänge der Batches, der Anteil der dominanten Klasse eines Batches, die Cosinus-Ähnlichkeit zum letzten Update und die dominante Klasse eines Batches

Da Datenpunkte einer Klasse meist sehr ähnlich sind, sind die Batches in den ersten Epochen häufiger mit Datenpunkten derselben Klasse gefüllt. Es können mehrfach Batches derselben Klasse hintereinander gewählt werden, da die Klasse mit den längsten Gradienten nicht nach einem Update von einer anderen abgelöst werden muss. Daraufhin hat eine andere Klasse einen hohen Loss und füllt den Batch. Dabei muss es nicht eine einzige Klasse sein, sondern der Batch kann auch aus mehreren Klassen bestehen, die den letzten Updates widersprechen. Die geschieht mehrfach im Wechsel, sodass vorwiegend Batches aus einer Klasse gewählt werden, immer wieder unterbrochen von Batches, die mit mehr unterschiedlichen Klassen gefüllt sind. Die resultierenden Updates sind meist orthogonal oder entgegengesetzt zueinander.
In Abbildung 4.11 ist zu sehen, wie die letzten Schritte vor dem Absturz waren. Zu Beachten ist, dass bei dem Wechsel von Epoche 7 auf 8 die Gradientenlängen von 1.7 auf 6 ansteigen. Der Wechsel ist in Batch 2 auf 3, dort steigen die Längen von 12 auf 120. Des Weiteren sind die Batches vorwiegend mit Klassen gefüllt, die in der vorherigen Epoche lange nicht mehr dominant waren. Die ersten beiden Batches enthalten am häufigsten Datenpunkte der Klasse 0. Nach den beiden Updates sind die Datenpunkte der Klasse 9 mit den längsten Gradienten. Nach diesen Updates hat die Klasse 7, welche am Ende der letzten Epoche häufig oft vorkam, die längsten Gradienten. Ab diesem Punkt steigen die Längen exponentiell an, wobei die Updates fast orthogonal sind. Da die Klasse 7 die Batches mehrfach gefüllt hat, sind daraufhin Batches mit einem geringeren Anteil der dominanten Klasse gewählt. Deren Updates sind erneut orthogonal, wodurch die Längen im nächsten Batch gegen unendlich gehen und bei den Berechnungen mit ∞ als Ergebnis NaN zurückgegeben wird. NaN steht für Not-a-Number und sagt aus, dass das Ergebnis

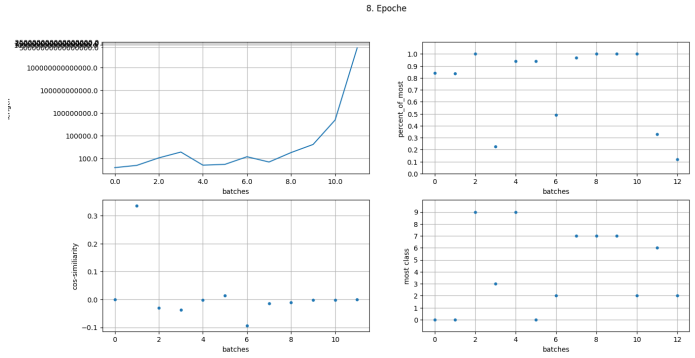

Abbildung 4.11: Die Daten der 8. Epoche des in Epoche 8 abgestürzten SGD-Trainings mit topk-Sortierung, Learningrate 0.1. Dargestellt sind die mittlere Gradientenlänge der Batches, der Anteil der dominanten Klasse eines Batches, die Cosinus-Ähnlichkeit zum letzten Update und die dominante Klasse eines Batches

einer Berechnung keinem mathematisch definierten Wert entspricht. Da mit NaN nicht weitergearbeitet werden kann, stürzt das gesamte Programm ab.
Es zeigt sich, dass Datenpunkte einer Klasse meistens im selben Batch liegen, wodurch Datenpunkte anderer Klassen diesen widersprechen. Durch dieses Verhalten steigt der Loss und die Gradientenlängen insbesondere zu Beginn einer Epoche.
Abbildung 4.12 zeigt, dass in SGD, wo die Batches vollkommen zufällig gezogen werden, der Batch gleichverteilt gefüllt ist. Dadurch kann sich das Netz in verschiedenen Dimension durch alle Gradienten anpassen und wird in den folgenden Batches auch nicht wieder in die entgegengesetzte Richtung gezogen.
Da SGD mit Sortierung ebenfalls abstürzt, wird die Vermutung gestützt, dass die Gradientenlänge durch die Sortierung ansteigt und schließlich zu hohe Werte annimmt. Dabei verursachen insbesondere die Ausreißer zu Beginn einer Epoche große Schwankungen. Diese bestehen meistens aus einer Klasse, die in der vorherigen Epoche lange nicht dominant war und daher Datenpunkte mit hohem Loss enthält. Dadurch wird das Netz stark verändert und es verlängern sich auch die Gradienten der anderen Klassen.
Um dieses Verhalten auszugleichen, kann versucht werden, die Batches weiterhin durch Sortierung zu wählen, dabei allerdings eine künstliche Diversität der Klassen einzuhalten. Damit soll verhindert werden, dass Batches fast vollständig durch nur eine Klasse gefüllt sind.
Die Experimente mit dieser künstlichen Diversität bei Sortierung zeigen allerdings keinen Effekt. SGD stürzt bei den hohen Learningrates von 1.0 und 0.1 weiterhin regelmäßig ab. Dabei ist der Nutzen der Sortierung mit Diversität sehr gering, da weiterhin ähnliche Datenpunkte gewählt werden. Diese sind zwar aus unterschiedlichen Klassen, jedoch

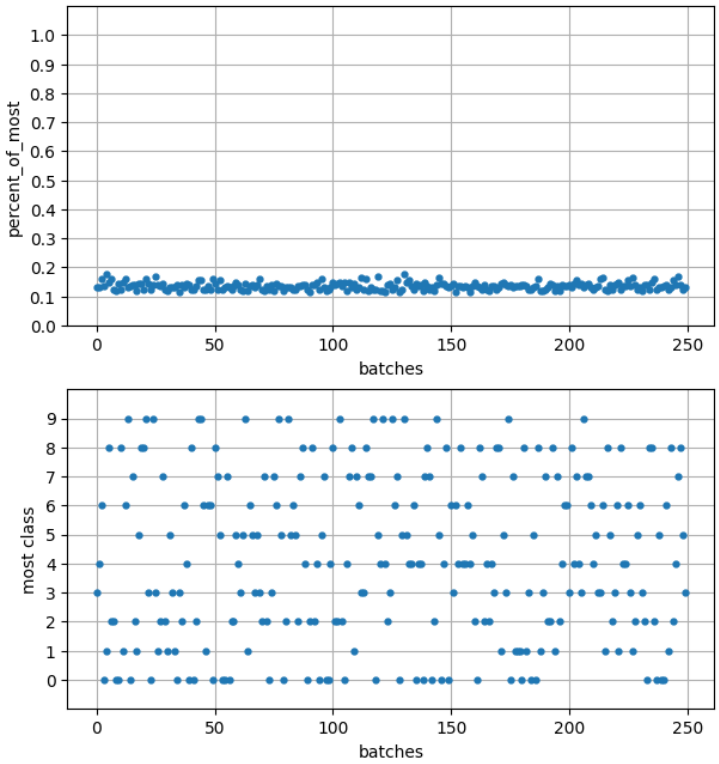

Abbildung 4.12: Der Anteil der Klassen in den Batches bei gewöhnlicher zufälliger Wahl der Batches in SGD

ziehen zum einen die Gradienten anderer Klassen mit hohem Loss das Netz in ähnliche Richtungen, zum anderen können Gradienten mit kurzen Gradientenlängen überstimmt werden.

4.4.3 SortedDPSGD und Dynamische Clipping-Bound

Bei den Versuchen auf dem Datensatz CIFAR10 zeigen sich viele Abstürze des Programms. Diese passieren aufgrund eines NaN-Fehlers, der während der Loss-Berechnung auftritt. Wie zuvor gesehen, führt es zu Abstürzen, wenn die Gradienten ähnlich wie bei SGD unbeschränkt sind. Bei der dynamischen Anpassung der Clipping-Bound wird diese abhängig von den längsten Gradienten gewählt. Dies entspricht einer sehr geringen Beschränkung. Deshalb kann die Länge der Gradienten immer weiter steigen.
Um dem entgegenzuwirken, wird die dynamische Clipping-Bound so gewählt, dass diese nicht zu stark wachsen kann. Dieses Bounding mittels C_{max} beschränkt die dynamische Clipping-Bound C_{dyn} nach oben und wird mittels $C_{new} = \min(C_{dyn}, C_{max})$ gewählt.

Es wird als obere Begrenzung der dynamischen Clipping-Bound von SortedDPSGD die in DPSGD genutzte Clipping-Bound C verwendet. Die beiden Algorithmen werden anhand ihres Trainings miteinander verglichen.
Tabelle 4.6 zeigt die Endresultate des Trainings bei $C = 5$ für die initialen Learningrates $1.0, 0.1$ und 0.01.

Tabelle 4.6: Ergebnisse von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median mit Maximum C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Variante	Loss	Acc
DPSGD0.01	1.95	31.15
DPSGD0.1	1.62	50.95
DPSGD1.0	2.4	17.29
SortedDPSGD0.01	1.99	29.68
SortedDPSGD0.1	1.57	51.4
SortedDPSGD1.0	2.19	35.54

Dabei fällt auf, dass das Training von SortedDPSGD bei den Learningrates 1.0 und 0.1 bessere Performance-Werte erreicht. Der Loss der SortedDPSGD-Variante mit initialer Learningrate 1.0 steigt zwischendurch auf sehr hohe Werte und sinkt erst ab Epoche 20 wieder ab (vergleiche Abbildung 4.13). Interessant ist dabei, dass der endgültige Loss mit 2.19 unter den Loss 2.4 von DPSGD fällt. DPSGD gelingt es bei der hohen Learningrate kaum, im Verlauf des Trainings den Loss oder die Accuracy zu verbessern, während SortedDPSGD

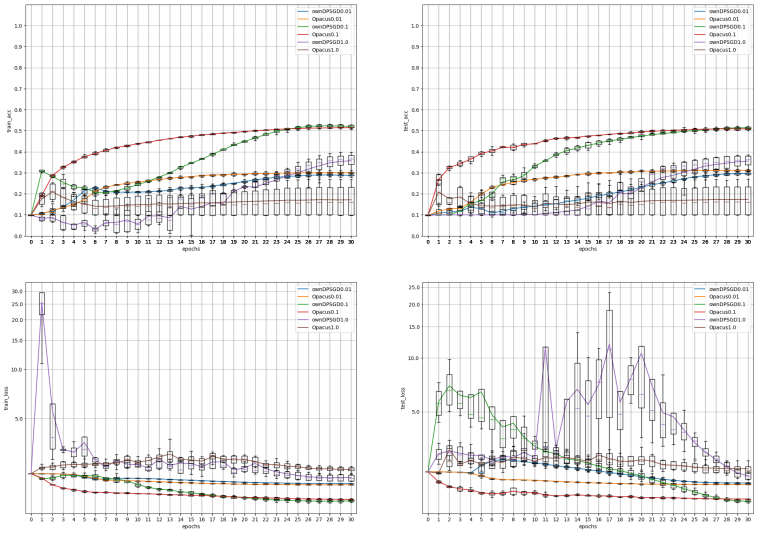

Abbildung 4.13: Verläufe von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median mit Maximum C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

am Ende des Trainings einen deutlichen Unterschied zeigt (siehe Abbildung 4.13).

Abbildung 4.14 zeigt den Verlauf des Trainings über 50 Epochen. Dabei ist der Peak in den ersten Epochen des Losses von SortedDPSGD bei dynamischer Clipping-Bound geringer als im Training mit fester Clipping-Bound. Die Endwerte von SortedDPSGD sind ebenfalls etwas besser als bei fester Clipping-Bound (vergleiche im Anhang Tabelle A.2).

Tabelle 4.7: Ergebisse von SortedDPSGD mit Quantilesort und topk-Sortierung mit dynamischer Clipping-Bound durch Min mit Maximum C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.769$

Variante	Loss	Acc
Quantilesort0.01_C-10	1.89	32.73
Quantilesort0.1_C-9.5	1.43	54.89
Quantilesort1.0_C-1	1.75	55.96
topk-sort0.01_C-10	1.9	31.98
topk-sort0.1_C-9.5	1.45	52.58
topk-sort1.0_C-1	1.79	54.79

Die Tabelle 4.7 zeigt die Endresultate der topk-Sortierung gegen Quantilesort über 30 Epochen für das jeweils beste C. Sie zeigt außerdem, dass eine weniger korrekte Sortierung das Ergebnis nicht unbedingt negativ beeinflusst.

Beim Vergleich der Verläufe in den Abbildungen 4.13 und 4.15 ist zu sehen, dass der

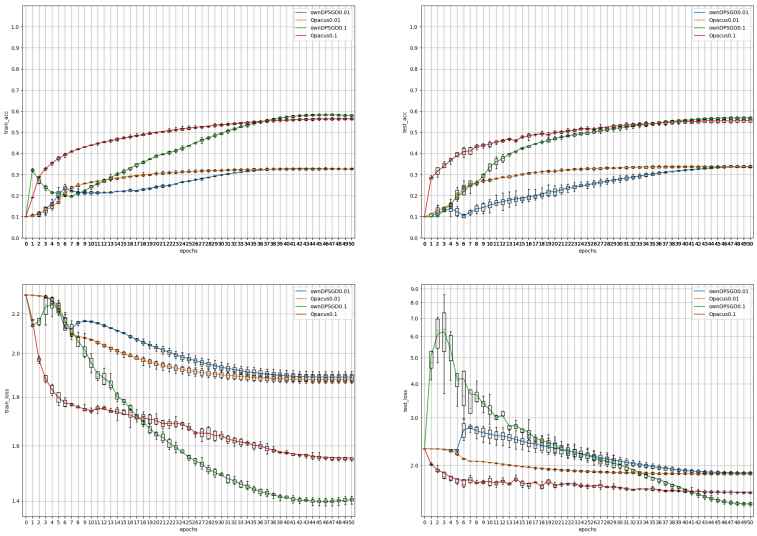

Abbildung 4.14: Verläufe von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median mit Maximum C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

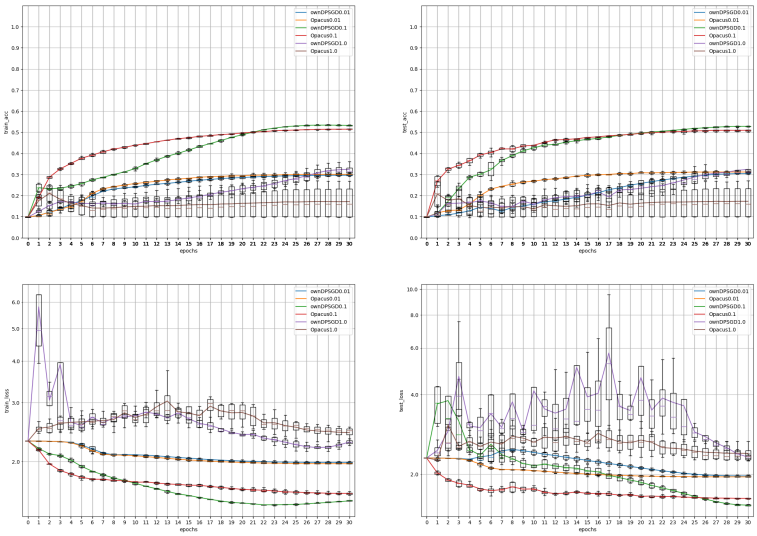

Abbildung 4.15: Verläufe von DPSGD mit $C = 5$ und SortedDPSGD mit Quantilesort und dynamischer Clipping-Bound durch Median mit Maximum C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

Anstieg des Losses bei Verwendung des Quantilesorts geringer ist als bei der topk-Sortierung.

Tabelle 4.8: Ergebnisse von DPSGD mit für die Learningrate idealem C und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median mit Maximum C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

Variante	Loss	Acc
DPSGD0.01_C-10	1.75	37.84
DPSGD0.1_C-9.5	1.6	57.35
DPSGD1.0_C-1	2.04	57.35
SortedDPSGD0.01_C-10	1.75	38.04
SortedDPSGD0.1_C-9.5	1.33	57.92
SortedDPSGD1.0_C-1	1.75	57.77

Tabelle 4.8 zeigt die Performanz von DPSGD und SortedDPSGD auf den für die jeweilige Learningrate idealen Parameter C aus Tabelle 4.4 über 50 Epochen. Dabei erreicht SortedDPSGD in allen Variationen sowohl im Loss als auch in der Accuracy eindeutig bessere Werte als DPSGD.

Vergleicht man die Ergebnisse der durch die Min-Funktion oder den Median gewählten Clipping-Bound, ist zu sehen, dass diese sehr ähnlich sind. Dabei kann die Min-Funktion jedoch einen kleinen Vorteil des Losses im Vergleich zu dem Median verzeichnen (vergleiche Tabelle 4.9).

Tabelle 4.9: Ergebnisse von SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median gegen Min mit Maximum C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Variante	Loss	Acc
SortedDPSGD0.01_median	1.99	29.68
SortedDPSGD0.1_median	1.57	51.4
SortedDPSGD1.0_median	2.19	35.54
SortedDPSGD0.01_min	1.99	29.68
SortedDPSGD0.1_min	1.56	51.16
SortedDPSGD1.0_min	2.08	38.22

Die Cosinus-Ähnlichkeiten über den Trainingsverlauf mit dynamischer Clipping-Bound sind bei gleicher Learningrate nahezu identisch zu den Abbildungen 4.7 und 4.8.

4.5 DP in SortedDPSGD

Es hat sich gezeigt, dass SortedDPSGD im Optimalfall gute Ergebnisse liefern kann. Dieser Optimalfall war allerdings speziell für die Experimente dieser Arbeit gewählt und erfüllt nicht Differential Privacy. Im Folgenden werden Möglichkeiten vorgestellt, die SortedDPSGD mit DP ermöglichen.

Das im Bezug auf Privacy-Kosten teuerste an SortedDPSGD ist die Sortierung. Bei einer Sortierung werden alle Datenpunkte mehrfach betrachtet, wodurch eine sehr große Leakage entsteht. Quantilesort wurde ebenfalls vorgestellt, jedoch erfüllt auch dieser nicht DP.

Um einen Batch mit möglichst geeigneten Datenpunkten zu füllen, kann Rejectionsampling verwendet werden. Rejectionsampling ist keine Sortierung, sondern ein Auswahlverfahren, das anhand einer Bewertungsfunktion eine Auswahl trifft. Dabei werden zufällig Samples ausgewählt und durch eine Bewertungsfunktion bewertet. Ist ein Sample geeignet, wird dieses weitergereicht oder ansonsten verworfen.

Ein Verfahren für Rejectionsampling in DP ist ***Algorithm 1*** *Thresholding with a known threshold* τ [LT]. Dieses Verfahren erlaubt es, geeignete Datenpunkte trotz Einhaltung von DP zu erhalten. Diese Rejectionsampling-Variante bekommt dabei als Eingabe Privacy-Parameter, eine Menge von Datenpunkten und einen Threshold, über dem die Auswahl liegen soll. Dabei werden bis zu einer maximalen Häufigkeit Samples gezogen. Liegt die Auswahl über dem Threshold, so wird sie als Ausgabe gewählt und der Algorithmus terminiert. Wenn sie unterhalb des Thresholds liegt, wird mit einer geringen Wahrscheinlichkeit dennoch gestoppt.

In diesem Setting soll ein Batch so gewählt sein, dass die Längen im Batch über einem Threshold bezüglich aller Längen liegen. Auf diese Weise kann ein Batch gefunden werden, der zwar nicht optimal, aber besser als viele andere ist. Daher würde Rejectionsampling bei jedem Durchlauf einen neuen Batch zufällig verteilt zusammenstellen und bewerten.

Dabei kann der Threshold für die Gradientenlängen des Batches mittels expQ [DFM$^+$] DP-erhaltend bezüglich aller Datenpunkte gewählt werden. Die Schwierigkeit ist es hierbei, den Threshold und die Bewertungsfunktion geeignet zu wählen, sodass die Laufzeit des Programms nicht allzu hoch ist. Die Bewertungsfunktion kann beispielsweise der Median der Gradientenlängen im gesampelten Batch sein. In diesem Beispiel ist der Median über alle Gradientenlängen des Datensatzes als Threshold schnell durch Rejectionsampling zu erreichen, da dies in etwa dem Erwartungswert entspricht. Aufgrund des Erwartungswertes spielt die Batchgröße auch eine Rolle bezüglich des Thresholds. Bei einem großen Batch ist es wahrscheinlich, dass in etwa gleich viel Datenpunkte über wie

auch unter dem Threshold liegen. Allerdings sind bei einem so leicht zu erreichenden Threshold die Datenpunkte des Batches wenig geeignet, um ein Ergebnis ähnlich dem von echter Sortierung zu erreichen. Wird beispielsweise der Wert des Quartils $Q_{0.75}$ der Threshold, so kann das Verfahren deutlich mehr Schritte benötigen, jedoch zeigen sich Ergebnisse ähnlich einer DP-Variante von Quantilesort.
Wenn die Sortierung mit DP arbeitet, muss die dynamische Anpassung der Clipping-Bound ebenfalls angepasst werden. Diese darf dabei nicht von den Längen des gezogenen Batches abhängen.
In dieser Arbeit wurde eine Methode untersucht, die es erlaubt, die Clipping-Bound in jeder Epoche unabhängig von den Gradientenlängen innerhalb der Batches zu setzen. In Abbildung 4.16 sind die mittleren Gradientenlängen innerhalb der Batches bei SGD mit topk-Sortierung dargestellt.

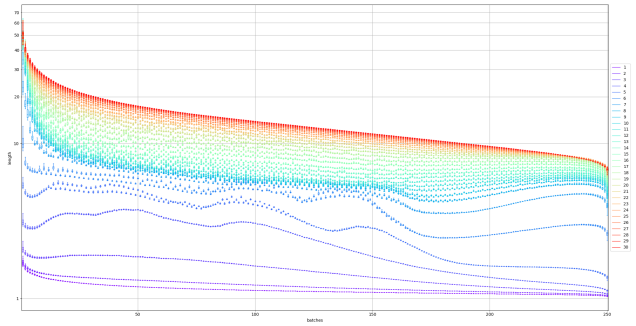

Abbildung 4.16: Die Gradientenlängen der Batches als eingefärbte Boxplots über die Epochen. Aus dem SGD Training mit topk-Sortierung und initialer Learningrate 0.01

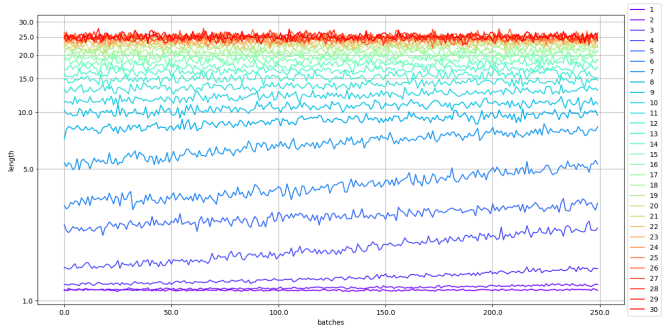

Abbildung 4.17: Die durchschnittlichen Gradientenlängen der Batches über die Epochen. Aus dem SGD Training ohne Sortierung

Es fällt auf, dass die Sortierung einen gleichmäßigen Verlauf der Gradientenlängen über die Batches erreicht. Weiter sind die Boxplots sehr klein dargestellt, was bedeutet, dass die Variation der Gradientenlängen innerhalb eines Batches sehr gering sind. Im Vergleich dazu zeigt Abbildung 4.17 die gemittelten Gradientenlängen der Batches in SGD ohne Sortierung, wobei die Gradientenlängen gleichverteilt um einen Wert für jede Epoche verteilt sind. Sowohl mit Sortierung als auch ohne steigen die Gradientenlängen der Batches mit jeder Epoche an.
Es muss dabei jedoch beachtet werden, dass bei einer weniger effektiven Auswahlstrategie der Batches auch die Längen sich weniger eindeutig verhalten. In Abbildung 4.18 sind die Gradientenlängen dargestellt, die im Training mit Quantilesort in den Batches vertreten waren.

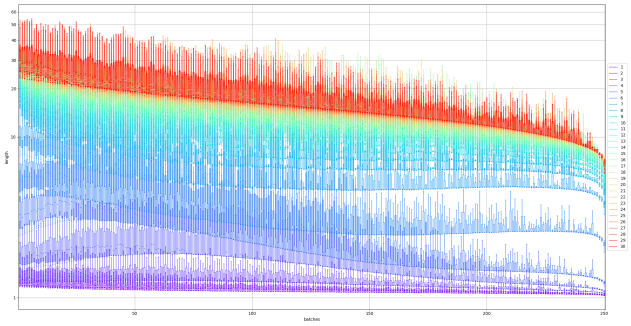

Abbildung 4.18: Die Gradientenlängen der Batches als eingefärbte Boxplots über die Epochen. Aus dem SGD Training mit Quantilesort und initialer Learningrate 0.01

Es zeigt sich, dass eine weniger effektive Sortierung als topk sich direkt auf den Verlauf und die Streuung der Gradientenlängen innerhalb der Batches auswirkt. Bei einer nicht perfekten Sortierung ist der Batch nicht immer vollständig mit den längsten Gradienten gefüllt. Dadurch sind in den folgenden Batches auch Datenpunkte, die sonst bereits in vorherigen Batches enthalten gewesen wären. Da diese auch in den späteren Batches vorkommen, ist der Abstieg weniger gleichmäßig.
Da Sortierung für einen gleichmäßigen Verlauf der Gradientenlängen in den Batches über die Epochen sorgt, kann dies genutzt werden, um die Clipping-Bound anzupassen. Als Alternative zu dieser von den Gradienten abhängig gewählten Clipping-Bound könnte also eine DP-erhaltende Variante die Clipping-Bound anhand einer fest definierten Funktion innerhalb jeder Epoche anpassen. Diese Funktion kann somit unabhängig von den tatsächlichen Gradientenlängen des Trainings die Clipping-Bound trotz Einhaltung der DP-Eigenschaft dynamisch verändern.

Wird eine Funktion mit einem ähnlichen Verlauf der Gradientenlängen wie in Abbildung 4.16 gewählt, so sollte sich die Clipping-Bound ähnlich zu dem tatsächlichen Längenverlauf verhalten. Dabei wird diese jedoch datenunabhängig bestimmt und sorgt für Einhaltung der Differential Privacy.
Eine derartige Funktion ist beispielsweise die Funktion c in Gleichung (4.1).

$$c(B) = A \log(B_{max} + 1 - B) + v_{max} - A \log(B_{max}) \tag{4.1}$$

Dabei ist B der Index des Batches, B_{max} die Anzahl der Batches über eine Epoche, v_{max} das Maximum der Funktion und A bestimmt die Steigung. Der Verlauf dieser Funktion ist für verschiedene Steigungen A in Abbildung 4.19 dargestellt. Durch diese Funktion wird das recht stabile Verhalten der Längen approximiert.

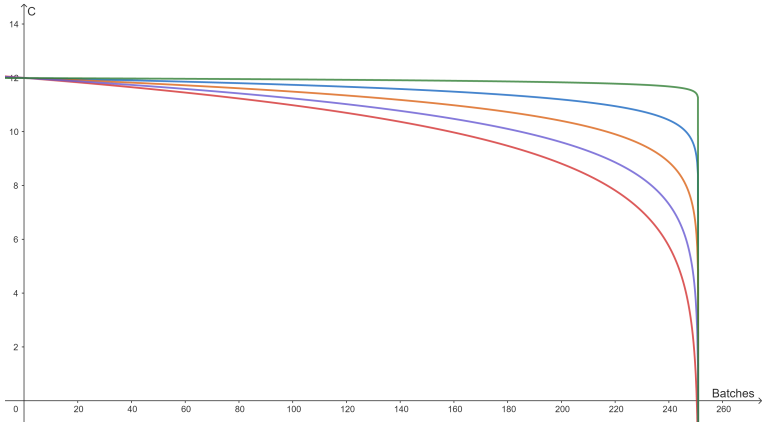

Abbildung 4.19: Verlauf der beispielhaften Log-Funktion zum dynamischen Anpassen der Clipping-Bound C mit verschiedenen Steigungen A.

In Experimenten mit topk-Sortierung und der durch die in Gleichung (4.1) gegebene Funktion dynamisch angepassten Clipping-Bound zeigt sich, dass auch hier ähnliche Ergebnisse wie mit DPSGD erreicht werden konnten. Tabelle 4.10 zeigt, dass DP-Variante der dynamischen Clipping-Bound etwa gleich gute Ergebnisse erzielt, wie die Nicht-DP Varianten Min und Median.

4.6 Diskussion

Werden die Gradientenlängen der einzelnen durch topk-Sortierung gewählten Batches betrachtet, so zeigt sich anhand der Boxplots in Abbildung 4.16, dass diese nahezu die gleiche Länge haben. Dabei hängt jedoch von der Accuracy der Sortierung die Streuung der Gradientenlängen innerhalb eines Batches ab. Abbildung 4.18 zeigt, dass die Gradientenlängen der durch Quantilesort gewählten Batches nach den ersten Epochen auch in etwa

Tabelle 4.10: Ergebnisse von SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Min gegen Median gegen Log mit Maximum für die Learningrate idealem C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Variante	Loss	Acc
SortedDPSGD0.01_min10	1.85	34.27
SortedDPSGD0.01_median10	1.85	34.27
SortedDPSGD0.01_log10	1.85	33.2
SortedDPSGD0.1_min9.5	1.45	52.58
SortedDPSGD0.1_median9.5	1.47	51.84
SortedDPSGD0.1_log9.5	1.42	54.3
SortedDPSGD1.0_min1	1.79	54.79
SortedDPSGD1.0_median1	1.83	54.45
SortedDPSGD1.0_log1	1.66	54.3

die gleichen Längen haben, dies ist aber weniger ausgeprägt als bei der topk-Sortierung. Die Annahme 1 (vgl. 4.1) konnte also bestätigt werden.
Da für die Batches immer die Datenpunkte mit den längsten Gradientenlängen gewählt wurden, stieg, wie in Abbildung 4.3 dargestellt, der Loss zu Beginn des Trainings sehr stark an. Aus Abbildung 4.10 lässt sich entnehmen, dass in den ersten Epochen einzelne Klassen die Batches fast vollständig füllen und dadurch durchgeführte Updates anderer Klassen teilweise ausgleichen. Die Batches mit vorwiegend einer Klasse häufen sich dabei. Besonders in Auge fällt, dass in allen Durchläufen der Loss in den ersten Epochen stark ansteigt. Da jeder Batch so gewählt wird, dass die Datenpunkte möglichst lange Gradienten besitzen, wird durch den Batch die Position des Netzes stark verändert. Ein Batch ist also so zusammengestellt, dass einige wenige Datenpunkte ihren Loss senken, während sich der Loss für viele andere Datenpunkte erhöht. Gerade zu Beginn des Trainings ist das Netz zufällig initialisiert. Daher sind fast alle Datenpunkte zu diesem Zeitpunkt Ausreißer. Die aufeinander folgenden Batches widersprechen sich mehrfach und erhöhen den Loss erneut. Nach einigen Epochen sinkt der Loss wieder und das Netz hat gutes Training. Das kann damit erklärt werden, dass sich nach den ersten Epochen die echten Ausreißer herauskristallisieren und somit immer zu Beginn einer Epoche einen Batch füllen. Durch sie wird das Netz stark verändert, alle darauf folgenden Datenpunkte sind jedoch ähnlicher und ändern die Position des Netzes so, dass der Loss für die Mehrheit wieder sinkt. Die Datenpunkte mit der höchsten Gradientenlänge sind also vorwiegend Ausreißer und beeinflussen das Netz negativ. Der Vergleich der topk-Sortierung und Quantilesort 4.7 zeigt, dass eine schwächere Sortierung gute Werte erreicht. Des Weiteren kann beim Vergleich von Abbildung 4.15 und 4.13 gesehen werden, dass die negativen Auswirkungen auf den Loss geringer sind. Nach Abbildung 4.18 resultiert Quantilesort zwar in einem

ungleichmäßigeren Verlauf der Gradientenlängen in Batches einer Epoche, dennoch zeigt sich, dass eine ungenauere Sortierung den Verlauf nicht negativ, sondern positiv beeinflusst. Dadurch kann die Annahme 2 (vgl. 4.1) also abgelehnt werden.

Durch die Ergebnisse wie in Abbildung 4.3 konnte gezeigt werden, dass SortedDPSGD ein besseres Resultat als DPSGD liefern kann. Bei dem Vergleich der Tabellen 4.5 und 4.7 kann gesehen werden, dass die dynamische Anpassung der Clipping-Bound den Vorteil von SortedDPSGD zusätzlich steigert. Dabei war die Clipping-Bound meistens von den Gradientenlängen des Batches abhängig, konnte allerdings auch (wie in Tabelle 4.10 dargestellt) durch eine statische Funktion approximiert werden, ohne das Ergebnis allzu sehr zu verschlechtern. Damit ist Annahme 3 (vgl. 4.1) bestätigt.

Die Ergebnisse zeigen aber auch, dass SortedDPSGD zwar bessere Ergebnisse als DPSGD erzielen kann, DPSGD aber bei 30 Epochen dennoch einigermaßen ähnlich performt. Tabelle 4.5 zeigt, dass sich die Auswirkungen von SortedDPSGD im Vergleich zu DPSGD auch oft in einem geringeren Loss bei gleichzeitig schwächerer Accuracy äußern. Da das Training der Neuronalen Netze auf dem Loss basiert, ist der Loss als Metrik relevanter als die Accuracy. Damit ist der Vorteil von SortedDPSGD etwas höher als bei einem Vergleich der Accuracy.

Dabei setzt sich SortedDPSGD aus der Sortierung zur Auswahl eines Batches und der dynamischen Anpassung der Clipping-Bound zusammen. Die erhöhten Privacy-Kosten dieser beiden Teile sind das Hauptproblem gegenüber DPSGD.

Für die Auswahl der Batches wird ein Sortierverfahren verwendet. Dabei ist die Güte des Sortierverfahrens ein relevanter Faktor, um Batches gleichlanger Gradienten zu finden. Das als Alternative zu topk vorgeschlagene Sortierverfahren Quantilesort liefert weniger genaue Ergebnisse in der Sortierung, kann allerdings im Training ähnlich gute Resultate liefern wie topk (vergleiche Tabelle 4.7). Selbst wenn die Sortierung durch andere Auswahlverfahren realisiert werden kann, entstehen hohe Privacy-Kosten. Dabei wird die getroffene Auswahl des Batches noch stärker gestört als durch Quantilesort. Ein derart schwächere Auswahl lässt vermutlich auch das Ergebnis geringer werden, wodurch der Vorteil von SortedDPSGD verfallen könnte.

Bei der dynamischen Anpassung der Clipping-Bound muss darauf geachtet werden, dass diese keine zu hohen Werte annehmen kann, da dies sonst zu immer längeren Gradienten führt. Durch zu lange Gradienten in Kombination mit Sortierung wurde gezeigt, dass das Modell schlechter wird oder das Programm abstürzen kann (wie in Abbildung 4.9). Daher sollte die dynamische Clipping-Bound eine Begrenzung nach oben haben. Wird die Clipping-Bound nun geringer als eine feste Clipping-Bound, so reduziert sich auch der Anteil des Noises. Da das nur geschieht, wenn die Längen auch entsprechend kurz sind, sind die Updates effektiver und enthalten weniger Noise. Die Anpassung der Clipping-

Bound ist im Gegensatz zur Sortierung deutlich weniger problematisch. Es konnte in Tabelle 4.10 gezeigt werden, dass trotz Einhaltung von DP diese zu ähnlich guten Ergebnissen führen kann wie datenabhängige Verfahren.
Da die Fortschritte von SortedDPSGD im Training erst verzögert nach einigen Epochen eintreten, benötigt man eine höhere Epochenzahl, um wie in Abbildung 4.14 einen klaren Vorteil gegenüber DPSGD zu zeigen. Höhere Epochen sorgen aber für mehr Updateschritte und erhöhen damit das benötigte Privacy-Budget noch mehr.

4.7 Fazit

Wie eingangs erläutert, ist ein großes Problem bei DPSGD, dass viele Informationen der Gradienten durch das Clippen auf C verloren gehen. Dabei kann das dafür verantwortliche C jedoch nicht simpel auf einen hohen Wert gesetzt werden, da der Noises mit diesem Parameter C skaliert wird. Dadurch ist Differential Privacy erfüllt, jedoch kann es zu schlechteren Ergebnissen führen, wenn das C zu groß für die Gradienten innerhalb des Batches gewählt wurde.
Um dies auszugleichen, wurde eine Methode entworfen, um Batches mit Datenpunkten ähnlicher Gradientenlängen zu füllen. Aufgrund der gleichen Länge innerhalb des Batches kann die Clipping-Bound auf einen entsprechenden Wert geändert werden, um den Einfluss des Noises gering zu halten und dennoch viele Informationen aus den einzelnen Gradienten zu erhalten.
Das neue Verfahren hat dabei ohne Einhaltung von DP einen geringen Vorteil gegenüber gewöhnlichem DPSGD. Dabei sind die Privacy-Kosten des neuen Verfahrens jedoch aufgrund der Sortierung sehr viel höher als bei DPSGD. Zum jetzigen Zeitpunkt hat die neue Methode aufgrund dieser hohen Privacy-Kosten allerdings wenig praktischen Nutzen. Es konnte immerhin gezeigt werden, dass eine weniger effektive Sortierung als topk ähnlich gute Ergebnisse liefert. Dies gibt Anlass zu der Hoffnung, dass ein Auswahlverfahren im Zusammenhang mit der DP die erzielten Vorteile nicht vollständig zunichtemacht.
Die Idee, einen Batch aus in etwa gleich langen Gradienten zusammenzustellen, lohnt ebenfalls einer weiterführenden Betrachtung. Die Annahme einer Verbesserung des Trainings bei gleich langen Gradienten innerhalb eines Batches ist sehr intuitiv. Um dies weiter zu untersuchen, sind folgende Ansätze möglich:

a) Im Rahmen dieser Arbeit konnten die Auswirkungen von Sortierung bei sehr statischen Hyperparametern der Experimente eingehend untersucht werden. Gegebenenfalls könnte daran anschließend die Auswirkung bei einer Änderung der Hyperparameter untersucht werden.

b) Da hier der optimale Fall untersucht wurde, konnte aufgezeigt werden, dass immerhin ein geringer Nutzen vorhanden ist. Dabei wurden jedoch keine Untersuchungen im DP-Kontext unternommen, um zu prüfen, welche Auswirkungen eine Reduzierung beispielsweise der Korrektheit der Sortierung auf das Resultat hat. Eine derartige Untersuchung könnte sich lohnen.

c) Untersuchungen bezüglich des Trainingsverhaltens könnten ebenfalls interessant sein. Unter anderem, ob eine andere Learningrate, die am Anfang geringer ist, in der Mitte ansteigt und dann sinkt, besser funktioniert. Dadurch könnte verhindert werden, dass in den ersten Epochen der Loss zu stark steigt.

d) Eine weitere Möglichkeit besteht darin, die Batches nicht mit den längsten Gradienten, sondern nur mit möglichst ähnlichen Gradienten auszuwählen. Die Variante des Rejectionsorts kann ein solches Ergebnis liefern, wenn dessen Threshold dicht am Durchschnitt bleibt. Da gerade am Anfang einer Epoche die Ausreißer das Netz stören, kann untersucht werden, ob eine solche Wahl den Beginn einer Epoche glätten kann und inwiefern sich die Ausreißer dadurch verändern.

e) Des Weiteren wäre es interessant zu untersuchen, inwiefern die Ausreißer zu Beginn einer Epoche auch in späteren Epochen Ausreißer sind. Es hat sich in den Ergebnissen gezeigt, dass Klassen, die zu Beginn der vorherigen Epoche vorwiegend waren, ebenfalls in der darauffolgenden Epoche am Anfang standen. Bleiben die Datenpunkte an ähnlichen Positionen der Sortierung, kann die Häufigkeit der Sortierung reduziert werden, was niedrigere Privacy-Kosten ermöglicht.

f) SortedDPSGD erreicht in den ersten Epochen wenig, da das Netz erst in eine Position gebracht werden muss, bei der die Ausreißer bekannt werden. Deshalb könnte untersucht werden, inwiefern SortedDPSGD bei Finetuning von Netzwerken performt. Da Netzwerke wie Resnet [HZRS] jedoch recht groß sind, liegt eine Hauptschwierigkeit darin, die Berechnung aller Gradienten effizient durchzuführen. Dafür bietet sich beispielsweise eine lazy-Computation an. Das bedeutet, dass statt einer Berechnung über alle Datenpunkte von Anfang an, beispielsweise im Rahmen des bereits erwähnten Rejectionsamplings, erst nach der Auswahl verschiedener Möglichkeiten eine Berechnung Sample für Sample erfolgen könnte.

5 ProjectedDPSGD

Um zu prüfen, inwiefern die Updateschritte von DPSGD zu verbessern sind, wird als weiterer Ansatz H2 untersucht, also welche Auswirkungen eine zu DPSGD verschiedene Berechnung des Updatevektors hat.

5.1 Idee

Ein Neuronales Netz wird durch iteratives Training verbessert. Die Optimierung erfolgt mittels einer Loss-Funktion, die die Abweichung eines zu einem Input generierten Output und einem gegebenen Output darstellt. An der aktuellen Position des Netzwerkes wird die Steigung der durch die Loss-Funktion berechneten Landschaft analysiert. Ein Gradient zeigt in die Richtung der größten negativen Steigung, seine Länge ist von der Stärke der Steigung abhängig. Ein Gradient zeigt somit die Optimierung des Netzes bezüglich der Loss-Funktion an. Das Ziel ist dabei, ein Minimum zu finden. Betrachtet man die Trainingslandschaft der Loss-Funktion, kann der Fall auftreten, dass ein lokales Maximum, im Folgenden als "Hügel" bezeichnet, den Weg zu einem lokalen Minimum, im Folgenden als "Tal" bezeichnet, versperrt. Ein Gradient, der nur von der Steigung der aktuellen Position des Netzes abhängt, kann somit nicht auf ein Tal hinter einem Hügel zeigen. Stattdessen muss der Hügel erst umlaufen werden, wodurch viele Trainingsschritte notwendig werden. Diese vielen Schritte sind meist nicht optimal gewählt, da sie von den lokalen Bedingungen und den dortigen Gradienten abhängen. Dieses Szenario kann sowohl bei SGD als auch DPSGD auftreten. SGD betrachtet die Gradienten so wie sie berechnet wurden, wodurch diese eine beliebige Länge haben können. Bei DPSGD reduziert die Clipping-Bound die Länge der Gradienten und damit auch den Updateschritt. Alle Gradienten werden auf eine maximale Länge geclippt, wodurch ein Gradient, der stark an dem Hügel vorbeiziehen würde, in geclippter Variante den Updateschritt weniger beeinflussen kann. Dadurch erhöht sich die Zahl der benötigten Schritte noch mehr als im Fall von SGD.

Die hier untersuchte Idee stellt einen Algorithmus vor, der frühzeitig auf Änderungen reagiert und damit einen direkteren Weg als DPSGD gehen kann. Die Schritte sind damit vorausschauend gewählt. Dabei soll geprüft werden, ob die Schritte von DPSGD trotz dieses Vorteils ebenfalls effektiv sind und ob sich die Entwicklung eines DP-Algorithmus lohnt,

der Schritte vorschlägt, die frühzeitiger als DPSGD reagieren. Werden weniger Schritte benötigt, sinkt das notwendige Privacy-Budget. Ein Updateschritt muss dabei weiterhin innerhalb der Clipping-Bound C bleiben und zu gleichen Teilen aus den Gradienten der verschiedenen Datenpunkte bestehen.

Die Idee des Algorithmus ist es, einen virtuellen Schritt zu tätigen und von dort neue Gradienten zu generieren. Die Gradienten aus dem Zwischenschritt werden zu einem endgültigen Schritt zusammengefasst. Dieses Verfahren soll dafür sorgen, dass mehrere gegensätzliche Schritte vermieden werden können.

Dieser Algorithmus heißt ProjectedDPSGD, da dieser durch Projektion die Effizienz von DPSGD erhöhen kann. Das Vorgehen ist ähnlich zu einem Algorithmus namens projected SGD. Der generelle Verlauf des Projected SGD-Algorithmus ist in Algorithmus 3 zu sehen. Dieser berechnet die Schritte des SGD-Algorithmus und projiziert sie in einen Unterraum. In Varianten des Projected SGD-Algorithmus kommen auch mehrere virtuelle Schritte vor [XSYC].

Algorithm 3: General Projected SGD (Outline)[PSE]

Input: Examples $\{x_1, ..., x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, group size L, Projektion $\Pi_\Theta : \Theta \to \Theta$.
Initialize $\theta_0 \in \Theta$ randomly
for $t \in [T]$ **do**
    Compute gradient
    For each $i \in L_t$, compute $\boldsymbol{g}_t(x_i) \leftarrow \nabla\theta_t\mathcal{L}(\theta_t, x_i)$
    Descent
    $\theta_{t+1} \leftarrow \Pi_\Theta\left(\theta_t - \eta_t \boldsymbol{g}_t\right)$
Output θ_T

Die Anwendung für ProjectedDPSGD ist inspiriert von Mądry et al. [MMS+] und kommt aus dem Einsatz für Adversarily Examples. Das sind Daten, die absichtlich so manipuliert wurden, dass das Neuronale Netz diese falsch klassifiziert, obwohl die Änderung nur sehr gering ist. Bei dem vorgestellten Ansatz mit Projected SGD im Folgenden PGD werden sehr gute Ergebnisse erreicht. Dabei wird PGD mit mehreren Schritten verwendet, um ein möglichst gutes Update innerhalb einer Sphäre zu finden, ohne dass es zu großen Änderungen kommt. Dies weist Ähnlichkeit zu der von C aufgespannten Hypersphäre in DPSGD auf. Das Clippen um C entspricht dabei einer Projektion auf die Oberfläche einer Hypersphäre mit Radius C, wenn der Gradient eine Länge $> C$ besitzt.

5.1.1 Algorithmus

Um den Updateschritt ähnlich zu DPSGD zu berechnen, ist es notwendig, dass die von der Clipping-Bound C aufgespannte Sphäre nicht verlassen wird. Es wird also innerhalb von C nach einem Optimum gesucht. Dazu eignet sich eine Berechnung wie sie die Gleichung (5.1) zeigt.

$$v = \frac{1}{\rho} v_{old} + \left(1 - \frac{1}{\rho}\right) v_{new} \tag{5.1}$$

Der neue Updateschritt v setzt sich hier aus zwei Gradienten zusammen. Der Algorithmus berechnet dafür einen Vektor und clippt diesen auf C um v_{old} zu bilden. Dieser wird daraufhin mit dem Parameter $\frac{1}{\rho}$ skaliert. Es wird ein virtueller Zwischenschritt entlang des skalierten v_{old} gemacht. Von dieser neuen Position aus wird nach Art von DPSGD ein weiterer Vektor erzeugt und geclippt, dies ist v_{new}. Dieser Vektor wird durch den Parameter $1-\frac{1}{\rho}$ skaliert. Der endgültige Updateschritt ist nun die Summe der beiden Zwischenschritte.

Der resultierende Vektor kann nie größer sein als C, da jeder der Vektoren maximal die Länge C besitzt, und die Summe der Parameter $\frac{1}{\rho} + \left(1 - \frac{1}{\rho}\right) = 1$ ist. Der Algorithmus erfüllt jedoch nicht DP, da v_{new} auf den Gradienten nach dem Update des Batches basiert. Ein einzelner Datenpunkt geht dabei nicht nur durch seinen Gradienten in das Update ein, sondern ist bereits durch das vorherige Update in den Parametern enthalten. Es ist offensichtlich, dass dadurch der Term nicht mehr linear ist, wodurch die Privacy-Eigenschaft nicht für ProjectedDPSGD bewiesen werden kann.

Ein Ablauf des Algorithmus ist in Abbildung 5.1 dargestellt. Dabei ist die Darstellung an der Abbildung 2.2 von DPSGD orientiert.

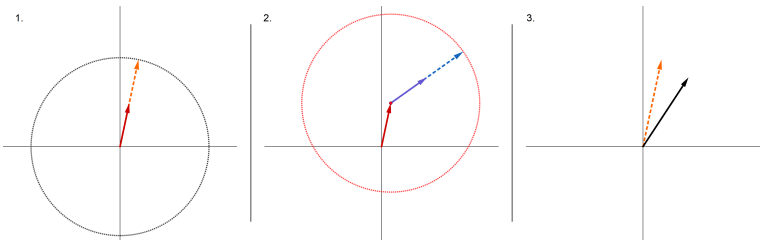

Abbildung 5.1: Darstellung des ProjectedDPSGD-Algorithmus im zweidimensionalen Raum für $rho = 2$.
1. Der erste Schritt ist wie von DPSGD berechnet und skaliert mit Faktor $\frac{1}{\rho}$, 2. Von der neuen Position wird erneut eine DPSGD-Berechnung mit Skalierung um $1 - \frac{1}{\rho}$ duchgeführt, 3. Die in 1. und 2. berechneten Vektoren bilden zusammen den resultierenden Vektor (schwarz) im Vergleich zu dem ursprünglichen DPSGD-Vektor (orange)

Der vorgestellte Algorithmus 4 ist dabei eine modifizierte Variante des DPSGD-Algorithmus von Abadi et al. [ACG+].

Algorithm 4: ProjectedDPSGD (Outline)

Input: Examples $\{x_1, ..., x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C, proportion ρ.
Initialize θ_0 randomly
for $t \in [T]$ **do**
  Take a random sample L_t with sampling probability L/N
  $\bar{\theta} \leftarrow \theta$
  Make the first Substep
  Compute initial gradient
  For each $i \in L_t$, compute $\boldsymbol{g}_t(x_i) \leftarrow \nabla\theta_t\mathcal{L}(\theta_t, x_i)$
  Clip gradient
  $\check{\boldsymbol{g}}_t(x_i) \leftarrow \boldsymbol{g}_t(x_i)/\max\left(1, \frac{\|g_t(x_i)\|_2}{C}\right)$
  Partially Descent with parameter ρ
  $\bar{\theta}_{t+1} \leftarrow \theta_t - \eta_t\frac{1}{\rho}\check{\boldsymbol{g}}_t$
  Second Substep
  Compute second gradient for new position
  For each $i \in L_t$, compute $\boldsymbol{g}_t(x_i) \leftarrow \nabla\bar{\theta}_{t+1}\mathcal{L}(\bar{\theta}_{t+1}, x_i)$
  Clip gradient
  $\bar{\boldsymbol{g}}_t(x_i) \leftarrow \boldsymbol{g}_t(x_i)/\max\left(1, \frac{\|\boldsymbol{g}_t(x_i)\|_2}{C}\right)$
  Compute full gradient step with both steps
  $\check{\boldsymbol{g}}_t(x_i) = \frac{1}{\rho}\check{\boldsymbol{g}}_t(x_i) + \left(1 - \frac{1}{\rho}\right)\bar{\boldsymbol{g}}_t(x_i)$
  Add noise
  $\tilde{\boldsymbol{g}}_t \leftarrow \frac{1}{L}(\sum_i \check{\boldsymbol{g}}_t(x_i) + \mathcal{N}(0, \sigma^2C^2\mathbf{I}))$
  Descent
  $\theta_{t+1} \leftarrow \theta_t - \eta_t\tilde{\boldsymbol{g}}_t$
Output θ_T and compute the overall privacy cost (ϵ, δ) using a privacy accounting method.

Der Beginn des Algorithmus ist wie aus DPSGD bekannt. Als Batch wird eine zufällige Untermenge aus den Datenpunkten gewählt. Diese sind innerhalb einer Epoche disjunkt. Für diesen Batch werden nach dem Verfahren von DPSGD die einzelnen Gradienten berechnet und geclippt. Es wird der gemittelte Vektor $v_{old} = \check{\boldsymbol{g}}_t(x_i)$ mit Länge l gewählt, jedoch noch kein Noise hinzugefügt, sondern ein Updateschritt einer Kopie des Netzes entlang des skalierten Vektors $\frac{1}{\rho}v_{old}$ getätigt. Dieses Update hat somit Länge $\frac{l}{\rho}$ und Learningrate η_t. Von diesem Punkt aus werden die Gradienten für die neue Position des Netzwerkes auf dem Batch bestimmt. Der Durchschnittsvektor $v_{new} = \bar{\boldsymbol{g}}_t(x_i)$ ist aus den geclippten Gradienten berechnet. Die Vektoren v_{old} und v_{new} werden nun durch den Faktor

$\frac{1}{\rho}$, beziehungsweise $1 - \frac{1}{\rho}$ miteinander verrechnet. Daraus bildet sich der endgültige Vektor als Summe der beiden skalierten Zwischenschritte. Durch die Division mittels ρ kann der Gesamtvektor nie länger als C werden. Am Ende wird auf diesen errechneten Updatevektor mit C skalierter Noise addiert, wie schon aus DPSGD bekannt. Das Netzwerk wird nun von der ursprünglichen Position mittels des neuen Updatevektors verschoben.
Da die Umgebung während des Updateschrittes betrachtet wird, kann besonders frühzeitig auf lokale Änderungen in der Landschaft reagiert werden. Dies soll dafür sorgen, dass ein direkterer Weg genommen wird. Dabei wurde sich stark an DPSGD orientiert, um diese besser miteinander vergleichen zu können.
Der Algorithmus hat durch den zusätzlichen Rechenschritt nur einen geringen Overhead und ist somit recht effizient.

5.2 Motivation

Um zu zeigen, dass der neue Algorithmus in entsprechender Landschaft der Loss-Funktion ein besseres Ergebnis als normales DPSGD liefern kann, wurde eine künstliche Loss-Funktion konstruiert. In Abbildung 5.2 ist ein Maximum zwischen dem Startpunkt und dem Minimum gegeben. Es sind die Wege der Algorithmen um das Maximum herum hin zum Minimum dargestellt. Die Ebene ist dabei geneigt und die Umgebung läuft dem Minimum zu.
Beide Algorithmen haben beim Experiment identische Hyperparameter wie Learningrate und Clipping-Bound. ProjectedDPSGD ist in rot gekennzeichnet, DPSGD in Gelb.
Beide Algorithmen können keinen direkten Schritt durch den Hügel zum Minimum hin tun. Daher arbeiten sich diese um den Hügel herum. ProjectedDPSGD bleibt dabei jedoch dichter am Fuß des Hügels und nimmt dadurch einen etwas direkteren Weg. Dies resultiert in weniger Schritten, die zusätzlich meistens kürzer sind. ProjectedDPSGD benötigt in diesem Beispiel 23 Schritte, DPSGD 27.
Da es in diesem konstruierten Szenario effizienter ist, den alternativen Algorithmus zu verwenden, werden weitere Experimente durchgeführt. Dabei ist das Ziel, zu evaluieren, ob sich der vorgeschlagene Algorithmus auch in komplexeren Situationen und im echten Training behaupten kann. Insbesondere da nicht feststeht, ob ein solch konstruiertes Problem auch in der Realität vorkommt. Finetuning auf vortrainierten Netzen sollte eine simplere Loss-Landschaft bieten, in der ein solches Szenario eher vorkommen kann.

5.3 Versuchsdurchführung

Es wurden identische Netze A und B erzeugt. Diese werden unabhängig voneinander trainiert. Bei Netz A kommt der von Abadi et al. [ACG+] bekannte Algorithmus DPSGD

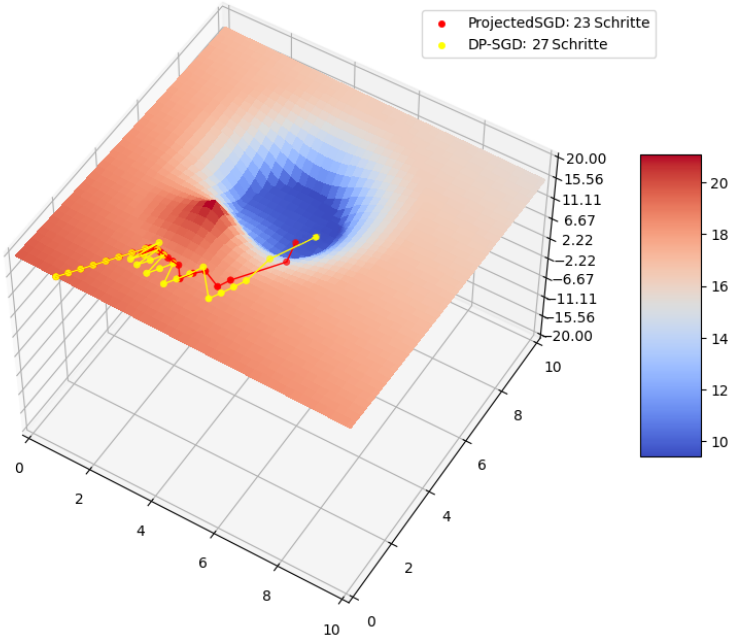

Abbildung 5.2: Training mittels DPSGD und ProjectedDPSGD auf einer konstruierten Loss-Funktion im 3-Dimensionalen Raum.

zum Einsatz. Netz B wird mit dem neuen ProjectedDPSGD-Algorithmus trainiert.
Um den Noise, der sich mehr oder weniger negativ auf den Trainingsverlauf auswirkt, nicht zu berücksichtigen, wurden mehrere Durchläufe durchgeführt und die Ergebnisse gemittelt.

Die Experimente wurden auf dem CIFAR10-Datensatz [KHo] durchgeführt. Dabei wurden die Modelle *Conv_CIFAR10_Net* (siehe im Anhang Listing A.2) und *Resnet18_CIFAR10_Net* (siehe im Anhang Listing A.3) verwendet.

Das ResNet18 [HZRS] ist ein Residual Network mit 18 Layern. Diese Art der Netzwerkarchitektur eignet sich besonders für tiefe Netzwerke und ist meist auf dem Datensatz ImageNet [RDS+] vortrainiert. Durch das vorherige Training auf Bildern sind die Parameter der unterschiedlichen Layer bereits geeignet, um relevante Zusammenhänge aus den Daten zu extrahieren. Ein solches Netz wird dann auf einem Datensatz wie hier CIFAR10 fein abgestimmt. Diese Art des Trainings ist besonders geeignet für DP, da der Großteil des Trainings auf öffentlichen und unbedenklichen Daten durchgeführt wurde [DBH+]. Das Optimieren auf einem kleinen privaten Datensatz ermöglicht bessere Resultate trotz geringer Privacy-Kosten. In dem hier genutzten vortrainierten Resnet18 wurde der letzte Linear-Layer, welcher in die Klassen zuordnet, ausgetauscht. Der neue Layer bildet nun auf die für CIFAR10 notwendige Anzahl von 10 Klassen ab.

In ersten Experimenten wurden als Hyperparameter eine Batchgröße von 200 bei 30 Epochen gewählt. Als Learningrate wurde hier ebenfalls die bereits vorgestellte adaptive Variante 2.3 mittels Cosinus verwendet. Dabei wurde zwischen den initialen Learningrates $0.01, 0.1$ und 1.0 variiert.
Die DPSGD betreffenden Hyperparameter sind der Noise-Multiplier $\sigma = 1.1$ sowie verschiedene Clipping-Bounds C aus den Werten $1.0, 2.5$ und 5.0.

Das für DPSGD relevante ϵ ergibt sich aus der Batchgröße, Datensatzgröße, Noise-Multiplier σ, Anzahl der Epochen und dem gewählten $\delta = 10^{-5}$ und hat den Wert $\epsilon = 1.729$. Dieser Wert liegt im unteren Teil des als schwach beschriebenen DP-Bereich $1 \leq \epsilon \leq 10$.
Zum Training kann ein beliebiges $\rho \geq 1$ verwendet werden. In den Experimenten wird dieser Parameter für ProjectedDPSGD auf $\rho = 2$ festgesetzt, da somit beide Vektoren gleichviel in das Update eingehen. Dadurch sollen die Änderungen deutlicher werden. Wählt man für den ProjectedDPSGD-Algorithmus als Parameter $\rho = 1$ oder $\rho \to \infty$, so entspricht dies gewöhnlichem DPSGD-Training.
Um das Training der Netzwerke nachzuvollziehen, wurden Loss und Accuracy mitgeschrieben. Zusätzlich wurde auch die Cosinus-Ähnlichkeit zwischen dem resultierenden Updatevektor und dem ersten Zwischenschritt mitbetrachtet, um den Unterschied zwischen dem Standard DPSGD-Schritt und unserem Algorithmus aufzuzeigen.
Dabei ist die Cosinus-Ähnlichkeit der Cosinus des Winkels zweier Vektoren. Dieses Maß ist gut geeignet, um die Ähnlichkeit der Richtungen zweier Vektoren zu untersuchen. Die Werte sind dabei zwischen -1 und 1. Die 0 zeigt dabei die Orthogonalität der Vektoren an, wobei ein Wert, der gegen 1 geht, zeigt, dass die Vektoren in die gleiche Richtung zeigen. Ein negativer Wert stellt die Vektoren als entgegengesetzt dar. [SB]

5.4 Ergebnisse

Bei den Tests auf *Conv_CIFAR_Net* zeigt sich, dass der neue Algorithmus auf untrainierten Netzen keinen Vorteil bringt.
Nach Abbildung 5.3 ist der Verlauf der Genauigkeit und des Losses beider Algorithmen bei kleineren Learningrates nahezu identisch.
Bei Learningrate 1.0 zeigt sich, dass die Steigung der Genauigkeit von ProjectedDPSGD in den ersten Epochen etwas geringer ist als bei DPSGD. Der dort entstandene Abstand bleibt das gesamte Training über erhalten. Im Loss hingegen ist ein deutlicherer Unterschied zu sehen. Es zeigt sich, dass bei beiden Algorithmen der Loss erst fällt, um dann nach 3 Epochen wieder anzusteigen. Dies ist bei beiden Algorithmen zu beobachten und

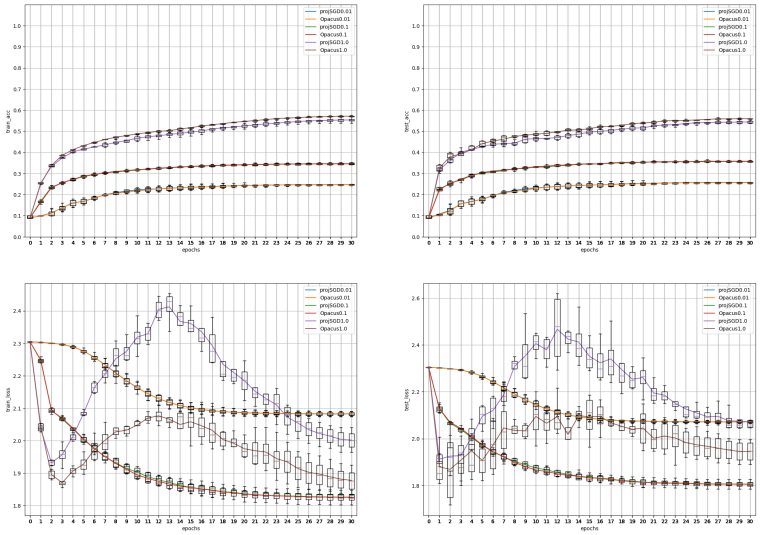

Abbildung 5.3: Verläufe von DPSGD mit $C = 1$ gegen ProjectedDPSGD mit $\rho = 2$ über 30 Epochen auf dem CIFAR10-Datensatz mittels *Conv_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

liegt an der großen Learningrate. Durch diese sind die Updateschritte zu groß und der Loss sinkt nicht. Ist der Schritt zu groß, stimmt zwar die Richtung, jedoch werden Minima übersprungen. ProjectedDPSGD hat dabei deutlich stärkere Schwankungen des Losses und erreicht zeitweise Werte, die noch über dem Startwert liegen. Gegen Ende des Trainings nähern sich beide Algorithmen wieder an, DPSGD jedoch mit offensichtlich niedrigeren Loss. ProjectedDPSGD sollte sich der Intuition nach besser eignen, um bei solch zu großen Schritten die Richtung zu ändern, jedoch steigt der Loss hier noch höher. Dies kann damit erklärt werden, dass die Richtung zwar angepasst wird, dadurch jedoch der zu große Schritt noch weiter von dem Minimum entfernt ist als bei DPSGD. Wie in Tabelle 5.3 zu sehen, ist DPSGD in allen Durchläufen besser oder gleich gut.

Tabelle 5.1: Ergebnisse von DPSGD mit $C = 1$ gegen ProjectedDPSGD mit $\rho = 2$ über 30 Epochen auf dem CIFAR10-Datensatz mittels *Conv_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Variante	Loss	Acc
DPSGD0.01	2.07	25.69
DPSGD0.1	1.81	35.8
DPSGD1.0	1.95	55.93
ProjectedDPSGD0.01	2.07	25.73
ProjectedDPSGD0.1	1.81	35.67
ProjectedDPSGD1.0	2.07	54.38

Auch bei einer größeren Clipping-Bound liegen die Ergebnisse der Algorithmen sehr dicht beieinander. Bei größerer Learningrate ist klar zu sehen, dass beide Netze einen ungleichmäßigen Loss haben. ProjectedDPSGD hat in den meisten Fällen einen höheren und ungleichmäßigeren Loss als DPSGD.
Wird die Clipping-Bound erhöht, so kann ProjectedDPSGD in seinem Updateschritt auch bei kleinen Learningrates stärker variieren als zuvor. In Abbildung 5.4 ist zu sehen, dass sich das negativ auf den Verlauf auswirkt. Bei Learningrate 1.0 beginnen die Netze nach

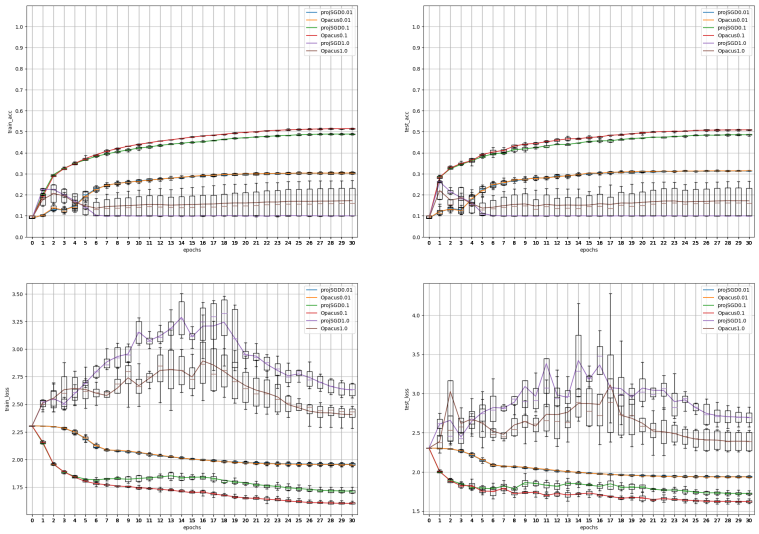

Abbildung 5.4: Verläufe von DPSGD mit $C = 5$ gegen ProjectedDPSGD mit $\rho = 2$ über 30 Epochen auf dem CIFAR10-Datensatz mittels *Conv_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

wenigen Epochen wieder zu verlernen. DPSGD behält dabei jedoch einen höheren Wert als ProjectedDPSGD. In den anderen Fällen ist ProjectedDPSGD ebenfalls schwächer als das gewöhnliche DPSGD.
Den Werten der Tabelle 5.2 ist zu entnehmen, dass ProjectedDPSGD auch bei sehr hohen Noise kaum geschicktere Updates wählt, um dies auszugleichen. Die Differenz bei Learningrate 1.0 ist zu klein, als dass ein Vorteil erkennbar ist.
Um das Verhalten des ProjectedDPSGD zu erklären, kann die Cosinus-Ähnlichkeit betrachtet werden. In Abbildung 5.5 ist zusehen, dass die Cosinus-Ähnlichkeit zwischen dem von Opacus und dem von ProjectedDPSGD berechneten Update sehr von der Learningrate abhängt.
Ist die Learningrate klein, so sind der ursprüngliche Updatevektor und der durch ProjectedDPSGD berechnete nahezu identisch. Bei einer größeren Learningrate variiert die Cosinus-Ähnlichkeit immer mehr.
In Abbildung 5.6 ist zu sehen, dass die Clipping-Bound ebenfalls starke Auswirkungen

Tabelle 5.2: Ergebnisse von DPSGD mit $C = 1$ gegen ProjectedDPSGD mit $\rho = 2$ über 30 Epochen auf dem CIFAR10-Datensatz mittels *Conv_CIFAR10_Net*, Noise-Multiplier $\sigma = 2.6$, $\epsilon = 0.538$

Variante	Loss	Acc
Variante	Loss	Acc
DPSGD0.01	2.07	25.6
DPSGD0.1	1.77	37.08
DPSGD1.0	2.12	30.26
ProjectedDPSGD0.01	2.07	25.53
ProjectedDPSGD0.1	1.77	37.29
ProjectedDPSGD1.0	2.1	30.99

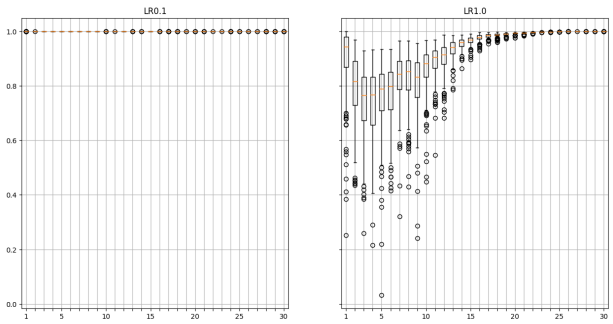

Abbildung 5.5: Cosinus-Ähnlichkeit der von ProjectedDPSGD mit $\rho = 2$ und DPSGD mit $C = 1$ berechneten Updatevektoren zu verschiedenen Lerningrates über 30 Epochen auf dem CIFAR10-Datensatz mittels *Conv_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

auf den Updateschritt hat.

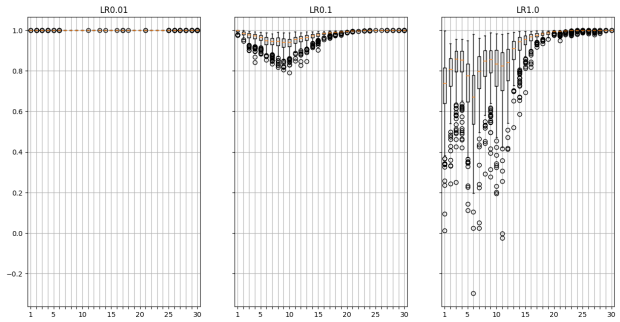

Abbildung 5.6: Cosinus-Ähnlichkeit der von ProjectedDPSGD mit $\rho = 2$ und DPSGD mit $C = 5$ berechneten Updatevektoren zu verschiedenen Lerningrates über 30 Epochen auf dem CIFAR10-Datensatz mittels *Conv_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Die Differenz der Updateschritte ist also größer, je größer die Learningrate oder Clipping-Bound ist. Dies ist auch zu erwarten, da sich bei größerer Clipping-Bound auch die Länge des Updateschrittes erhöht. Die Learningrate nimmt ebenfalls starken Einfluss, da diese auch im imaginären Schritt verwendet wird. Durch eine Änderung bei diesen Parametern ist der imaginäre Schritt länger und die Landschaft verändert sich mehr. Dadurch ändert sich der Unterschied zwischen dem ursprünglichen und dem resultierenden Updateschritt.

Im Verlauf des Trainings werden die Updateschritte immer ähnlicher. Dies kann damit erklärt werden, dass die Learningrate sinkt und damit ähnliche Werte der niedrigeren Learningrate angenommen werden. Zusätzlich ist der Verlauf dabei glatter als dies bei einem reinen Wechsel der Learningrate der Fall wäre. Daher kann vermutet werden, dass das Netz soweit trainiert ist, dass die Loss-Landschaft eindeutiger ist, wie im Fall von Finetuning.

Den Ergebnissen nach war die Performanz von ProjectedDPSGD bei einem untrainierten Netz in allen Fällen geringer als die des entsprechenden DPSGD-Algorithmus.

Im Training mit einem vortrainierten Netzwerk ist die Loss-Landschaft eindeutiger, wodurch sich die Algorithmen stärker ähneln als zuvor. In Abbildung 5.7 ist zu sehen, dass der Verlauf bei kleiner Clipping-Bound nahezu identisch ist. Selbst bei größerer Learningrate ist kaum ein Unterschied festzustellen.

Dabei ist im Vergleich der Abbildungen 5.8 und 5.4 zu sehen, dass die höhere Clipping-Bound beim vortrainierten Netzwerk einen wesentlich geringeren Unterschied macht als bei untrainierten Netzwerken. Wie in Abbildung 5.8 zu sehen, sind die Verläufe mit gerin-

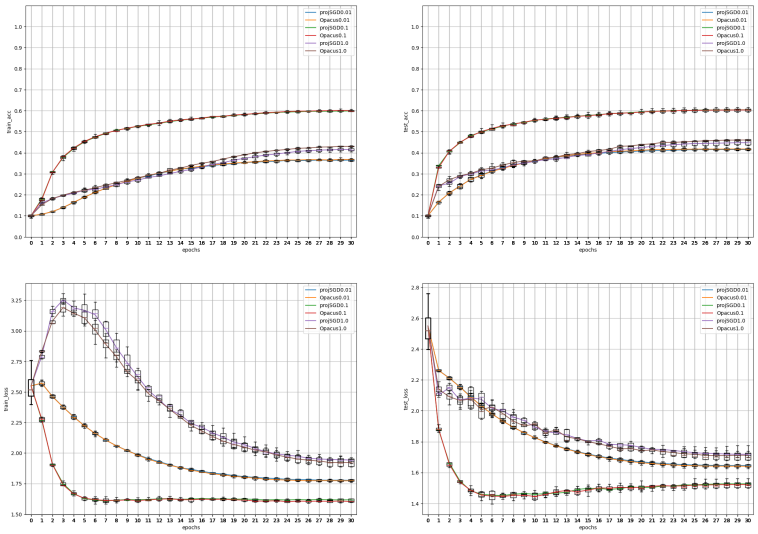

Abbildung 5.7: Verläufe von DPSGD mit $C = 1$ gegen ProjectedDPSGD mit $\rho = 2$ über 30 Epochen auf dem CIFAR10-Datensatz mittels *Resnet18_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

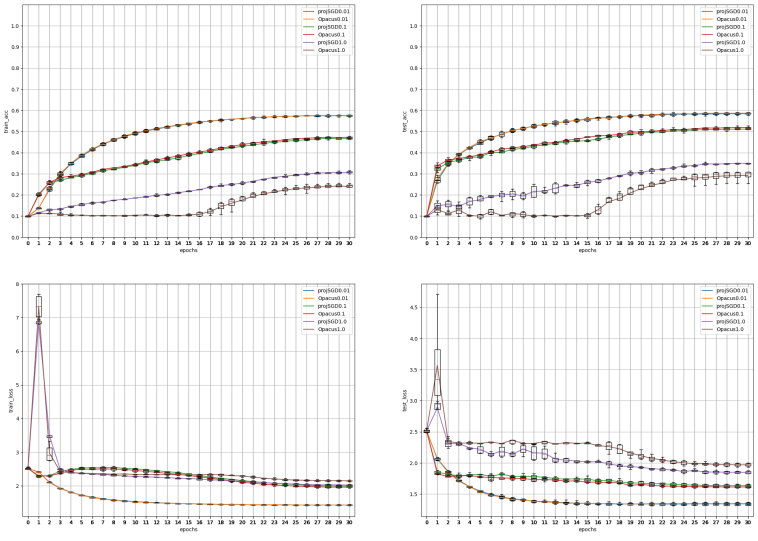

Abbildung 5.8: Verläufe von DPSGD mit $C = 5$ gegen ProjectedDPSGD mit $\rho = 2$ über 30 Epochen auf dem CIFAR10-Datensatz mittels *Resnet18_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

geren Learningrates sehr viel besser als bei Learningrate 1.0. Im Falle einer großen Learningrate und Clipping-Bound performt DPSGD nicht mehr sonderlich gut, da die Updateschritte zu groß sind und der Noise großen Einfluss hat. ProjectedDPSGD kann hier beim Finetuning ein geringfügig besseres Training vorweisen, welches allerdings keinen Vorteil gegenüber einer geschickt gewählten Learningrate hat.
Wird eine große Clipping-Bound gewählt, so sind die Ergebnisse meist schlechter. Insbesondere bei großer Learningrate schneiden beide Algorithmen nicht gut ab, da eine hohe Learningrate bei Finetuning ungeeignet ist. Es fällt auf, dass ProjectedDPSGD etwas bessere Ergebnisse liefert als DPSGD bei hoher Clipping-Bound und Learningrate. ProjectedDPSGD eignet sich dafür, die zu großen Schritte bei der simpleren vortrainierten Loss-Landschaft etwas mehr in die richtige Richtung zu lenken. Dennoch sind die erreichten Werte nicht sonderlich gut.
Der Intuition nach sollte ProjectedDPSGD einen Vorteil im Finetuning haben, da bei diesem eher eine solche Loss-Landschaft vorkommen kann. In den Experimenten wurde jedoch gezeigt, dass es nahezu keinen Unterschied im Training zwischen den beiden Algorithmen gab.
Betrachtet man die Cosinus-Ähnlichkeit der verschiedenen Updatevektoren auf dem vortrainierten Netz, so sind auch hier die Werte dicht an 1, wenn auch weniger als auf einem untrainierten Netz. Wie in Abbildung 5.9 zu erkennen, trägt im Gegensatz zu dem untrainierten Netz die hohe Learningrate bei kleiner Clipping-Bound weniger zur Streuung bei.

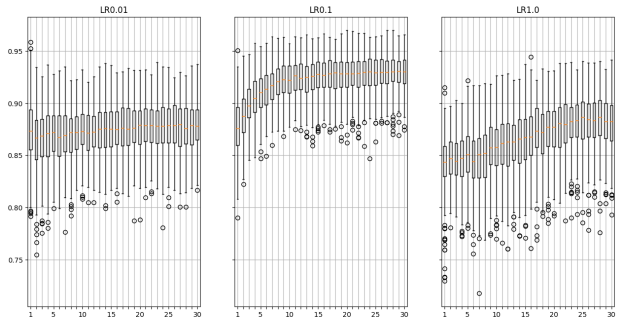

Abbildung 5.9: Cosinus-Ähnlichkeit der von ProjectedDPSGD mit $\rho = 2$ und DPSGD mit $C = 1$ berechneten Updatevektoren zu verschiedenen Lerningrates über 30 Epochen auf dem CIFAR10-Datensatz mittels *Resnet18_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Abbildung 5.10 zeigt, dass auf dem vortrainierten Netz bei großer Clipping-Bound die kleinen Learningrates sehr ähnlich sind, während die Learningrate 1.0 eine größere Streu-

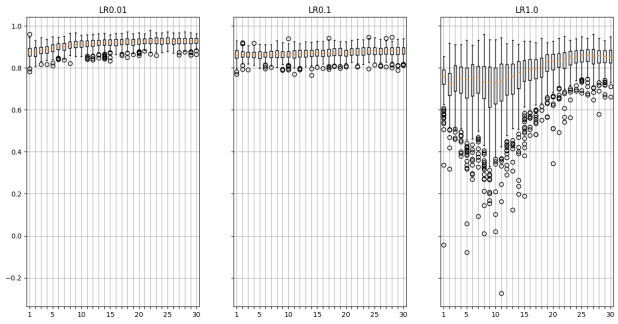

Abbildung 5.10: Cosinus-Ähnlichkeit der von ProjectedDPSGD mit $\rho = 2$ und DPSGD mit $C = 5$ berechneten Updatevektoren zu verschiedenen Lerningrates über 30 Epochen auf dem CIFAR10-Datensatz mittels *Resnet18_CIFAR10_Net*, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

ung hat. Im Gegensatz zu dem untrainierten Netz ist *Resnet18_CIFAR10_Net* von Beginn an im Finetuning. Daher ist hier, anders als in den Abbildungen 5.5 und 5.6, nur bei Learningrate 1.0 ein Verringern der Streuung über die Epochen festzustellen.
Sowohl auf untrainierten als auch bei vortrainierten Netzen ist der Verlauf und das Endresultat nahezu identisch zu DPSGD. Es hat sich also keine eindeutige Verbesserung gezeigt. Sind die Clipping-Bound und Learningrate gering, so ist der imaginäre Schritt mit Faktor $\frac{1}{\rho}$ ebenfalls sehr gering. Der von diesem neuen Punkt aus erzeugte Vektor ist demnach sehr ähnlich zu dem ursprünglichen Vektor von DPSGD. Dies lässt sich auch anhand der Cosinus-Ähnlichkeit dieser Vektoren bestätigen, welche über das gesamte Training oft dicht an 1 liegt.

5.5 Diskussion

Wie in der Tabelle 5.1 zu sehen, hat ProjectedDPSGD in den Ergebnissen in den meisten Fällen ähnlich abgeschnitten wie DPSGD. Auch im Fall von hohem Noise waren die Ergebnisse aus Tabelle 5.2 sehr dicht beisammen. Es fällt auf, dass ProjectedDPSGD in nahezu allen Fällen leicht schlechtere Resultate liefert als DPSGD. Die in Abbildung 5.6 dargestellten Cosinus-Ähnlichkeiten zeigen, dass ProjectedDPSGD immer recht ähnliche Schritte zu DPSGD gewählt hat. In Abbildung 5.4 ist zu sehen, dass der Verlauf der Accuracy und insbesondere der des Losses unregelmäßiger und schlechter ist als von DPSGD. Daraus lässt sich schließen, dass DPSGD sehr gute Updatevektoren vorschlägt, während nur ähnliche Vektoren schlechtere Ergebnisse liefern. Durch die Cosinus-Ähnlichkeiten kann man sehen, dass diese ähnlichen Vektoren dabei etwas zu dem von DPSGD vorgeschlagenen Weg verschoben sind, sodass sich diese ähnlich wie Noise negativ auswirken.

Dass nach der Cosinus-Ähnlichkeit wie in Abbildung 5.6 ab etwa der Hälfte der Epochen das Update immer stärker dem von DPSGD gleicht, lässt sich auf die adaptive Learningrate zurückführen. Dies kann zusätzlich durch das Auftreten einer Art Finetuning-Effekt verstärkt sein. Dabei wird die Loss-Landschaft eindeutiger und ändert sich weniger lokal, sodass die vorgeschlagenen Updates von DPSGD und ProjectedDPSGD ähnlicher sind.

Im Finetuning auf dem vortrainierten Netzwerk sind, wie in Abbildung 5.7 zu sehen, die Verläufe von ProjectedDPSGD und DPSGD noch ähnlicher. Dabei sind die Werte in Abbildung 5.9 auch dicht an 1. Beim Vergleich mit dem unsortierten Fall aus Abbildung 5.5 fällt allerdings auf, dass kein so deutlicher Unterschied zwischen den Verläufen mit anderen Learningrates zu sehen ist wie im untrainierten Fall. Bei größerer Clipping-Bound ist die Streuung bei Learningrate größer (vergleiche Abbildung 5.10) und es ist auch bei den späteren Epochen ein Verlauf hin zu weniger Streuung festzustellen. In dem vorliegenden Fall liegt das auch an der sinkenden Learningrate.

Da ProjectedDPSGD nur in dem von Abbildung 5.2 dargestellten Fall besser war, jedoch nicht im Training, liegt die Vermutung nahe, dass das Trainingsgelände selten so eindeutig ist wie in der Motivation verwendet. Unter anderem ist bei echtem Training die Loss-Landschaft von sehr viel mehr Dimensionen abhängig. Da sowohl die Learningrate als auch die Clipping-Bound einen großen Einfluss darauf haben, wie unterschiedlich die Vektoren sind, ist davon auszugehen, dass die Loss-Landschaft lokal wenig variiert und erst bei großen Entfernungen andere Gradienten entstehen.

Im Training auf dem vortrainierten Netzwerk ist in Abbildung 5.8 ein Verlauf zu sehen, bei dem ProjectedDPSGD besser abschnitt als DPSGD. Dabei zeigte sich in dem Durchlauf mit hoher Learningrate, dass erst gegen Ende des Trainings, aufgrund der bis dahin geschrumpften Learningrate, DPSGD keine guten Updates wählen konnte. Gleichzeitig konnte ProjectedDPSGD von Beginn an seinen Loss stetig reduzieren. Erklärt wird das damit, dass es ProjectedDPSGD möglich war, tatsächlich bessere Updates innerhalb der Clipping-Bound zu wählen. Diese konnten den zu großen Updateschritt in eine geeignetere Richtung lenken. Im Gegensatz dazu konnte auf dem untrainierten Netzwerk, wie in Abbildung 5.4 zu sehen, bei hoher Learningrate ProjectedDPSGD kein solches Resultat zeigen. DPSGD gelang in diesem Fall ein, wenn auch nicht sehr gutes, dennoch besseres Training als ProjectedDPSGD. ProjectedDPSGD kann also nur auf der vortrainierten und damit etwas eindeutigeren Loss-Landschaft ein besseres Update finden.

ProjectedDPSGD reagiert frühzeitiger auf Veränderungen als DPSGD. Der Idee nach sollte sich dies positiv auf den entstehenden Updateschritt auswirken, da eine effektivere Richtung eingeschlagen werden kann. Ähnliche Ergebnisse lassen sich allerdings durch eine geringere Learningrate erzielen, da dadurch auch frühzeitiger auf Änderungen reagiert wird. ProjectedDPSGD kombiniert das hier nur in einem einzigen Schritt. Dabei kön-

nen aber auch Fälle auftreten, in denen ProjectedDPSGD, ähnlich wie bei einer zu kleinen Learningrate, zu früh reagiert. Würde beispielsweise bei DPSGD ein Updateschritt über eine kleine Erhebung getätigt, so kann es bei ProjectedDPSGD vorkommen, dass der durch $\frac{1}{\rho}$ skalierte virtuelle Zwischenschritt auf eben diesem geringen Anstieg landet. Da sich dadurch die Richtung der Gradienten ändert, ist der resultierende Vektor ein schwaches Update. Es können also sowohl Fälle eines positiven als auch negativen Einflusses durch ProjectedDPSGD eintreten. Da ProjectedDPSGD nie einen Vorteil gegenüber DPSGD zeigen konnte, ist zu vermuten, dass DPSGD die für die Clipping-Bound optimalen Updates tätigt.

5.6 Fazit

In dieser Arbeit wurde das Verhalten des neuen Algorithmus mit Fokus auf einige wenige Hyperparameter, nämlich Clipping-Bound und Learningrate, untersucht. Des Weiteren wurden die Algorithmen in für Resnet18 optimalen Konfigurationen getestet, die beispielsweise von externen Quellen als besonders geeignet validiert wurden. DPSGD wählt den Updatevektor direkt aus den geclippten Gradienten. Da diese durch das Clippen nicht mehr so eindeutig wie bei SGD sind, sollte untersucht werden, ob DPSGD die Clipping-Bound gut genug ausnutzt und die bestmöglichen Schritte wählt, oder ob es eine bessere Position des Netzes innerhalb von C gibt. In diesem Kapitel wurde daher der Algorithmus ProjectedDPSGD vorgestellt. Mit diesem sollte ein zu DPSGD alternativer Updatevektor gefunden werden, der frühzeitig auf Veränderungen der Umgebung reagiert und somit innerhalb von C bessere Updates finden kann. Der Algorithmus wurde auf dem Datensatz CIFAR10 evaluiert, wobei sich herausgestellt hat, dass es weder im Finetuning noch im gewöhnlichen Training einen Vorteil gegenüber DPSGD gibt. Stattdessen wurde der Verlauf des Losses meist unregelmäßig und die Ergebnisse leicht schlechter.
Die Ergebnisse können damit begründet werden, dass ProjectedDPSGD zwar leicht andere Updateschritte als DPSGD durchführt, diese jedoch keinen oder einen leicht negativen Effekt ähnlich wie Noise haben.
ProjectedDPSGD eignet sich, um die Umgebung der DPSGD-Updates zu untersuchen. Es könnte sich lohnen, diesen Ansatz unter folgenden Aspekten weiterzuverfolgen.

a) Da in den Experimenten ProjectedDPSGD mit stark eingeschränkter Auswahl der Hyperparameter ausgewertet wurde, sollten weitere Hyperparameter darauf untersucht werden, ob diese einen positiven Einfluss haben. Zu diesen zählt unter anderem die Batchsize, die hier nur mit Größe 200 untersucht wurde oder der Parameter ρ.

b) Der ProjectedDPSGD-Algorithmus kann mittels eines Parameters Rounds r erweitert werden. Dieser würde nicht nur einen Zwischenschritt, sondern r Zwischenschritte zulassen. Durch die erhöhte Anzahl der Schritte könnte damit die Updateumgebung deutlicher betrachtet werden.

6 Fazit

6.1 Zusammenfassung

Neuronale Netze übernehmen in der heutigen Zeit immer mehr Aufgaben. Dabei sind die Möglichkeiten von Spracherkennung über Bildverarbeitung sehr umfangreich. In jedem Fall benötigen diese Netze zum Training große Datenmengen. Da gezeigt wurde, dass aus einem trainierten Netz Trainingsdaten extrahiert werden können, wurde das Verfahren von Differential Privacy für beweisbaren Schutz der Privatsphäre entwickelt. Darauf aufbauend ist DPSGD eine Variante des sehr effektiven und weit verbreiteten SGD-Algorithmus. Aufgrund der nötigen Einschränkungen von DPSGD ist die Performanz reduziert. Der DPSGD-Algorithmus verliert dabei aufgrund des Beschränkens von den Gradientenlängen mit einer Clipping-Bound Informationen. Gleichzeitig skaliert der hinzugefügte Noise mit der gewählten Clipping-Bound, weshalb diese nicht zu groß gewählt sein sollte.
In der vorliegenden Arbeit habe ich zwei verschiedene Strategien zur Verbesserung des Trainings im DPSGD Algorithmus mit den zugrundeliegenden Hyptothesen untersucht.
Im ersten Ansatz SortedDPSGD wurden die Auswirkungen von mit Sortierung gewählten Batches untersucht. Insbesondere das Zusammenspiel mit der Veränderung der Clipping-Bound innerhalb einer Trainingsepoche. Es hat sich gezeigt, dass die Variante mit Sortierung einer festen Clipping-Bound meist keinen Vorteil brachte. Wurde zusätzlich die Clipping-Bound dynamisch verändert, so konnte ein geringerer Loss erreicht werden. Die Accuracy des Netzes stieg allerdings erst in den späteren Epochen an. Der Trainingsverlauf war in beiden Varianten mit Sortierung verzögert, wodurch eine höhere Epochenzahl benötigt wurde. Da SortedDPSGD allerdings nicht die DP-Eigenschaft erfüllt und im Optimalfall evaluiert wurde, können sich diese leicht besseren Ergebnisse bei einer DP-Variante verringern. Daher ist SortedDPSGD ein guter Ansatz, benötigt jedoch weitere Untersuchungen, bevor sich ein echter Nutzen ergibt.
Für den zweiten Ansatz ProjectedDPSGD wurde der Algorithmus selbst modifiziert, um besser auf kleine Änderungungen der Loss-Landschaft zu reagieren. Dies wurde mit DPSGD verglichen, um nachzuvollziehen, ob DPSGD innerhalb der Cliping-Bound, also dem möglichen Suchraum, auch möglcihst gute Updates erzeugt. Die Ergebnisse haben gezeigt, dass ProjectedDPSGD sowohl im Training eines untrainierten Netzes, als auch im Finetuning zu DPSGD sehr ähnliche Updateschritte vorschlägt. Diese haben keinen

oder einen negativen Effekt auf das Training. Es entsteht dabei ein fast identischer oder ungleichmäßiger Verlauf. Dadurch kann geschlossen werden, dass leicht vorausschauende Updateschritte keine bessere Strategie als DPSGD sind. DPSGD setzt die Schritte innerhalb der Clipping-Bound so gut, dass nur ähnliche Updateschritte weniger Nutzen bringen.

6.2 Ausblick

Die Resultate zeigen, dass Sortierung zwar beim Training hilft, jedoch zur Zeit noch keinen echten Nutzen hat. Die Forschungsmöglichkeiten aus dem Fazit 4.7 zielen dabei vor allem auf Ansätze zur Verifizierung der Ergebnisse unter Berücksichtigung der Differential Privacy. Es sind aber auch Ansätze aufgeführt, die die Effektivität des Verfahrens weiter steigern können.
Da ProjectedDPSGD im Training keinen Vorteil gegenüber DPSGD gebracht hat, konnte geschlossen werden, dass DPSGD eine sehr gute Position innerhalb der Clipping-Bound findet. Trotz des geringen Effekts von ProjectedDPSGD kann dieser weiter zur Untersuchung von DPSGD verwendet werden. Es sind weitere Forschungsansätze im Fazit 5.6 des Kapitels 5 gegeben. Die dort beschriebenen Forschungsansätze stellen Erweiterungen des Algorithmus dar, um noch besser auf die Umgebung zu reagieren und somit herauszufinden, ab welchem Punkt die Updates besser sind als von DPSGD.

A Anhang

A.1 Modelle

Listing A.1: Beispiel eines 3-layer Fully Connected Neural Networks: *MNIST_Net*

```
Batch size: 250
Input shape: (250, 1, 28, 28)


==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MnistNet                                 [250, 10]                 --
+ Sequential: 1-1                        [250, 10]                 --
|    + Flatten: 2-1                      [250, 784]                --
|    + Linear: 2-2                       [250, 256]                200,960
|    + ReLU: 2-3                         [250, 256]                --
|    + Linear: 2-4                       [250, 32]                 8,224
|    + ReLU: 2-5                         [250, 32]                 --
|    + Linear: 2-6                       [250, 10]                 330
==========================================================================================
Total params: 209,514
Trainable params: 209,514
Non-trainable params: 0
Total mult-adds (M): 52.38
==========================================================================================
Input size (MB): 0.78
Forward/backward pass size (MB): 0.60
Params size (MB): 0.84
Estimated Total Size (MB): 2.22
==========================================================================================
```

Listing A.2: Beispiel eines Convolutional Neural Networks: *Conv_CIFAR_Net*

```
Batch size: 200
Input shape: (200, 3, 32, 32)


==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
```

```
==========================================================================================
ConvCifarNet                                [200, 10]                 --
+ Sequential: 1-1                           [200, 10]                 --
|    + Conv2d: 2-1                          [200, 32, 32, 32]         896
|    + ReLU: 2-2                            [200, 32, 32, 32]         --
|    + AvgPool2d: 2-3                       [200, 32, 16, 16]         --
|    + Conv2d: 2-4                          [200, 64, 16, 16]         18,496
|    + ReLU: 2-5                            [200, 64, 16, 16]         --
|    + AvgPool2d: 2-6                       [200, 64, 8, 8]           --
|    + Conv2d: 2-7                          [200, 64, 8, 8]           36,928
|    + ReLU: 2-8                            [200, 64, 8, 8]           --
|    + AvgPool2d: 2-9                       [200, 64, 4, 4]           --
|    + Conv2d: 2-10                         [200, 128, 4, 4]          73,856
|    + ReLU: 2-11                           [200, 128, 4, 4]          --
|    + AdaptiveAvgPool2d: 2-12              [200, 128, 1, 1]          --
|    + Flatten: 2-13                        [200, 128]                --
|    + Linear: 2-14                         [200, 10]                 1,290
==========================================================================================
Total params: 131,466
Trainable params: 131,466
Non-trainable params: 0
Total mult-adds (G): 1.84
==========================================================================================
Input size (MB): 2.46
Forward/backward pass size (MB): 88.49
Params size (MB): 0.53
Estimated Total Size (MB): 91.47
==========================================================================================
```

Listing A.3: Beispiel eines Deep residual Networks für CIFAR10: *Resnet18_CIFAR10_Net*

```
Batch size: 200
Input shape: (200, 3, 32, 32)


==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
PreTrained                               [200, 10]                 --
+ ResNet: 1-1                            [200, 10]                 --
|    + Conv2d: 2-1                       [200, 64, 16, 16]         9,408
|    + GroupNorm: 2-2                    [200, 64, 16, 16]         128
|    + ReLU: 2-3                         [200, 64, 16, 16]         --
|    + MaxPool2d: 2-4                    [200, 64, 8, 8]           --
|    + Sequential: 2-5                   [200, 64, 8, 8]           --
|    |    + BasicBlock: 3-1              [200, 64, 8, 8]           --
```

```
|    |    |    + Conv2d: 4-1            [200, 64, 8, 8]          36,864
|    |    |    + GroupNorm: 4-2         [200, 64, 8, 8]          128
|    |    |    + ReLU: 4-3              [200, 64, 8, 8]          --
|    |    |    + Conv2d: 4-4            [200, 64, 8, 8]          36,864
|    |    |    + GroupNorm: 4-5         [200, 64, 8, 8]          128
|    |    |    + ReLU: 4-6              [200, 64, 8, 8]          --
|    |    + BasicBlock: 3-2             [200, 64, 8, 8]          --
|    |    |    + Conv2d: 4-7            [200, 64, 8, 8]          36,864
|    |    |    + GroupNorm: 4-8         [200, 64, 8, 8]          128
|    |    |    + ReLU: 4-9              [200, 64, 8, 8]          --
|    |    |    + Conv2d: 4-10           [200, 64, 8, 8]          36,864
|    |    |    + GroupNorm: 4-11        [200, 64, 8, 8]          128
|    |    |    + ReLU: 4-12             [200, 64, 8, 8]          --
|    + Sequential: 2-6                  [200, 128, 4, 4]         --
|    |    + BasicBlock: 3-3             [200, 128, 4, 4]         --
|    |    |    + Conv2d: 4-13           [200, 128, 4, 4]         73,728
|    |    |    + GroupNorm: 4-14        [200, 128, 4, 4]         256
|    |    |    + ReLU: 4-15             [200, 128, 4, 4]         --
|    |    |    + Conv2d: 4-16           [200, 128, 4, 4]         147,456
|    |    |    + GroupNorm: 4-17        [200, 128, 4, 4]         256
|    |    |    + Sequential: 4-18       [200, 128, 4, 4]         --
|    |    |    |    + Conv2d: 5-1       [200, 128, 4, 4]         8,192
|    |    |    |    + GroupNorm: 5-2    [200, 128, 4, 4]         256
|    |    |    + ReLU: 4-19             [200, 128, 4, 4]         --
|    |    + BasicBlock: 3-4             [200, 128, 4, 4]         --
|    |    |    + Conv2d: 4-20           [200, 128, 4, 4]         147,456
|    |    |    + GroupNorm: 4-21        [200, 128, 4, 4]         256
|    |    |    + ReLU: 4-22             [200, 128, 4, 4]         --
|    |    |    + Conv2d: 4-23           [200, 128, 4, 4]         147,456
|    |    |    + GroupNorm: 4-24        [200, 128, 4, 4]         256
|    |    |    + ReLU: 4-25             [200, 128, 4, 4]         --
|    + Sequential: 2-7                  [200, 256, 2, 2]         --
|    |    + BasicBlock: 3-5             [200, 256, 2, 2]         --
|    |    |    + Conv2d: 4-26           [200, 256, 2, 2]         294,912
|    |    |    + GroupNorm: 4-27        [200, 256, 2, 2]         512
|    |    |    + ReLU: 4-28             [200, 256, 2, 2]         --
|    |    |    + Conv2d: 4-29           [200, 256, 2, 2]         589,824
|    |    |    + GroupNorm: 4-30        [200, 256, 2, 2]         512
|    |    |    + Sequential: 4-31       [200, 256, 2, 2]         --
|    |    |    |    + Conv2d: 5-3       [200, 256, 2, 2]         32,768
|    |    |    |    + GroupNorm: 5-4    [200, 256, 2, 2]         512
|    |    |    + ReLU: 4-32             [200, 256, 2, 2]         --
|    |    + BasicBlock: 3-6             [200, 256, 2, 2]         --
|    |    |    + Conv2d: 4-33           [200, 256, 2, 2]         589,824
|    |    |    + GroupNorm: 4-34        [200, 256, 2, 2]         512
|    |    |    + ReLU: 4-35             [200, 256, 2, 2]         --
```

```
|    |    |    + Conv2d: 4-36            [200, 256, 2, 2]          589,824
|    |    |    + GroupNorm: 4-37         [200, 256, 2, 2]          512
|    |    |    + ReLU: 4-38              [200, 256, 2, 2]          --
|    + Sequential: 2-8                   [200, 512, 1, 1]          --
|    |    + BasicBlock: 3-7              [200, 512, 1, 1]          --
|    |    |    + Conv2d: 4-39            [200, 512, 1, 1]          1,179,648
|    |    |    + GroupNorm: 4-40         [200, 512, 1, 1]          1,024
|    |    |    + ReLU: 4-41              [200, 512, 1, 1]          --
|    |    |    + Conv2d: 4-42            [200, 512, 1, 1]          2,359,296
|    |    |    + GroupNorm: 4-43         [200, 512, 1, 1]          1,024
|    |    |    + Sequential: 4-44        [200, 512, 1, 1]          --
|    |    |    |    + Conv2d: 5-5        [200, 512, 1, 1]          131,072
|    |    |    |    + GroupNorm: 5-6     [200, 512, 1, 1]          1,024
|    |    |    + ReLU: 4-45              [200, 512, 1, 1]          --
|    |    + BasicBlock: 3-8              [200, 512, 1, 1]          --
|    |    |    + Conv2d: 4-46            [200, 512, 1, 1]          2,359,296
|    |    |    + GroupNorm: 4-47         [200, 512, 1, 1]          1,024
|    |    |    + ReLU: 4-48              [200, 512, 1, 1]          --
|    |    |    + Conv2d: 4-49            [200, 512, 1, 1]          2,359,296
|    |    |    + GroupNorm: 4-50         [200, 512, 1, 1]          1,024
|    |    |    + ReLU: 4-51              [200, 512, 1, 1]          --
|    + AdaptiveAvgPool2d: 2-9            [200, 512, 1, 1]          --
|    + Sequential: 2-10                  [200, 10]                 --
|    |    + Dropout: 3-9                 [200, 512]                --
|    |    + Linear: 3-10                 [200, 10]                 5,130
==========================================================================================
Total params: 11,181,642
Trainable params: 11,181,642
Non-trainable params: 0
Total mult-adds (G): 7.41
==========================================================================================
Input size (MB): 2.46
Forward/backward pass size (MB): 162.22
Params size (MB): 44.73
Estimated Total Size (MB): 209.40
==========================================================================================
```

A.2 Grafiken

A.3 Tabellen

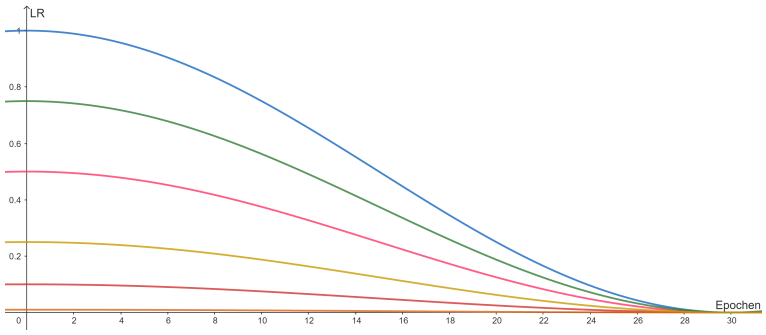

Abbildung A.1: Verlauf der adaptiven Learningrate basierend auf einer Cosinusfunktion zu verschiedenen initialen Learningrates über 30 Epochen

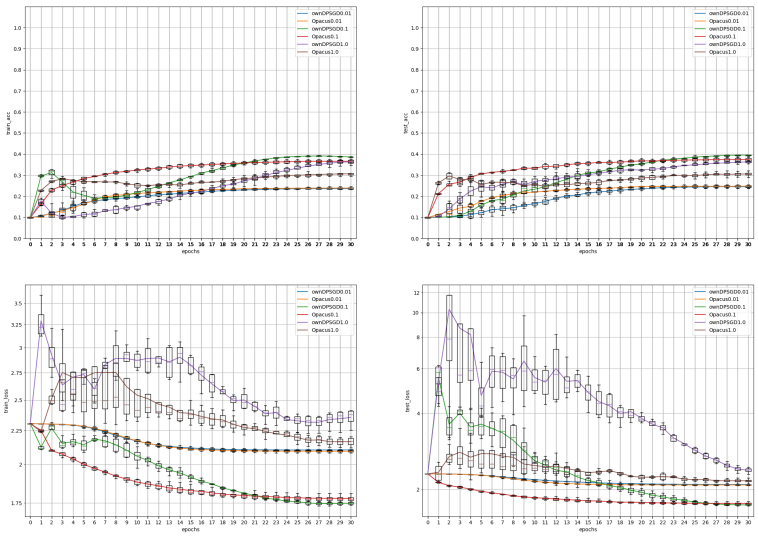

Abbildung A.2: Verläufe von DPSGD mit $C = 1$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 2.6$, $\epsilon = 0.538$

Tabelle A.1: Ergebnisse von DPSGD mit $C = 1$ und SortedDPSGD mit topk-Sortierung und fester Clipping-Bound C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

Variante	Loss	Acc
DPSGD0.001	2.3	12.26
DPSGD0.01	1.95	31.15
DPSGD0.1	1.62	50.95
DPSGD1.0	2.4	17.29
SortedDPSGD0.001	2.3	13.16
SortedDPSGD0.01	1.98	29.98
SortedDPSGD0.1	1.56	52.6
SortedDPSGD1.0	3.28	26.78

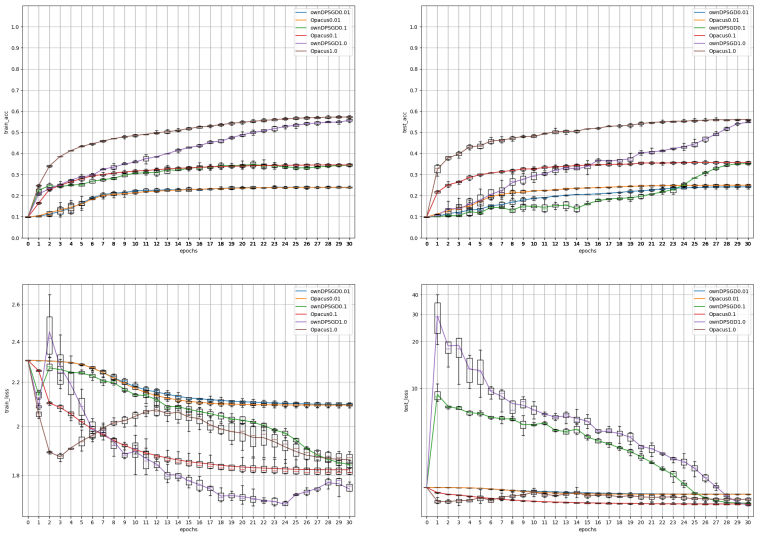

Abbildung A.3: Verläufe von DPSGD mit $C = 1$ und SortedDPSGD mit topk-Sortierung mit künstlicher Diversität und fester Clipping-Bound C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 1.729$

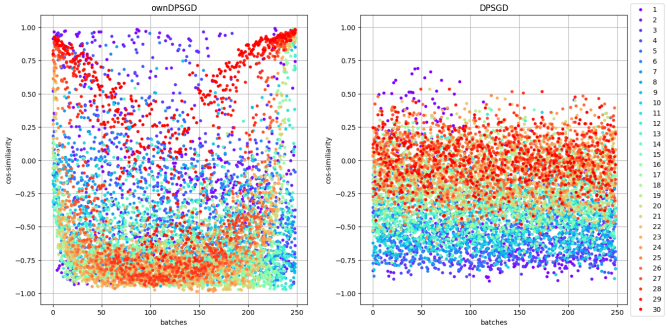

Abbildung A.4: Die Cosinus-Ähnlichkeiten aufeinanderfolgender Updateschritte in SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median mit Maximum $C = 9.5$ und DPSGD Training über 30 Epochen auf dem CIFAR10-Datensatz, Learingrate 0.1

Tabelle A.2: Ergebnisse von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Median mit Maximum C über 50 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1$, $\epsilon = 2.268$

Variante	Loss	Acc
DPSGD0.01	1.86	33.78
DPSGD0.1	1.59	55.52
SortedDPSGD0.01	1.87	33.57
SortedDPSGD0.1	1.44	56.78

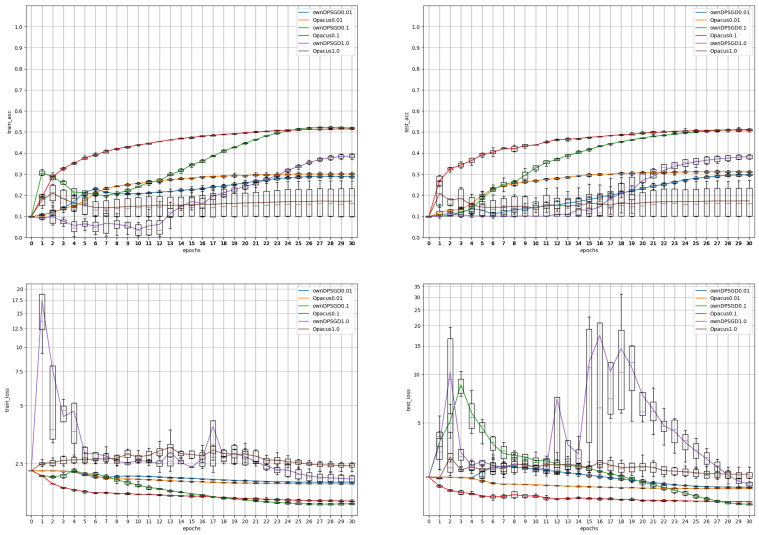

Abbildung A.5: Verläufe von DPSGD mit $C = 5$ und SortedDPSGD mit topk-Sortierung und dynamischer Clipping-Bound durch Min mit Maximum C über 30 Epochen auf dem CIFAR10-Datensatz, Noise-Multiplier $\sigma = 1.1, \epsilon = 1.729$

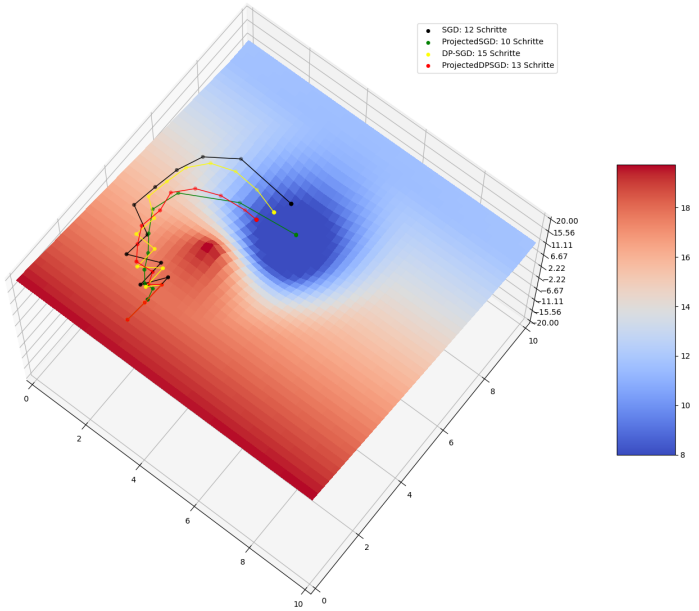

Abbildung A.6: Training von SGD, ProjectedSGD, DPSGD und ProjectedDPSGD auf einer konstruierten Loss-Funktion im 3-Dimensionalen Raum.

Literaturverzeichnis

[ACG+] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 308–318. Association for Computing Machinery. event-place: Vienna, Austria.

[ATMR] Galen Andrew, Om Thakkar, H. Brendan McMahan, and Swaroop Ramaswamy. Differentially private learning with adaptive clipping.

[con] pytorch/opacus. original-date: 2019-12-07T01:58:09Z.

[DBH+] Soham De, Leonard Berrada, Jamie Hayes, Samuel L. Smith, and Borja Balle. Unlocking high-accuracy differentially private image classification through scale.

[Den] Li Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. 29(6):141–142.

[DFM+] Wenxin Du, Canyon Foot, Monica Moniot, Andrew Bray, and Adam Groce. Differentially private confidence intervals.

[DR] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. 9(3):211–407.

[FCL] Jie Fu, Zhili Chen, and XinPeng Ling. SA-DPSGD: Differentially private stochastic gradient descent based on simulated annealing.

[HSW+] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models.

[HZRS] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[KHo] Alex Krizhevsky, Geoffrey Hinton, and others. Learning multiple layers of features from tiny images. Publisher: Toronto, ON, Canada.

[LT] Jingcheng Liu and Kunal Talwar. Private selection from private candidates. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 298–309. Association for Computing Machinery.

[MMS+] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks.

[PGM+] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[PSE] Sejun Park, Umut Simsekli, and Murat A. Erdogdu. Generalization bounds for stochastic gradient descent via localized ε-covers.

[PSY+] Venkatadheeraj Pichapati, Ananda Theertha Suresh, Felix X. Yu, Sashank J. Reddi, and Sanjiv Kumar. AdaCliP: Adaptive clipping for private SGD.

[RDS+] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. 115(3):211–252.

[SB] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. 24(5):513–523.

[SSSS] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models.

[XSYC] Quan Xiao, Han Shen, Wotao Yin, and Tianyi Chen. Alternating implicit projected SGD and its efficient variants for equality-constrained bilevel optimization.

[YSS+] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. Opacus: User-friendly differential privacy library in PyTorch.

[YZC+] Da Yu, Huishuai Zhang, Wei Chen, Jian Yin, and Tie-Yan Liu. Large scale private learning via low-rank reparametrization.