Hochschule für Technik und Wirtschaft

Optimierer in Convolutional Neural Networks

von

Martina Brüning

s0540636

Betreuer: Patrick Baumann, M. Sc.

15.09.2021

Zusammenfassung

Optimierer in Convolutional Neural Networks

Im vorliegenden Projekt soll der Validierungsverlust sowie die Validierungsgenauigkeit dreier Optimierer durch Veränderung der Stapelgröße im Bereich Convolutional Neural untersucht werden. Die hierbei verwendeten Optimierer sind Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam) und Layer-wise Adaptive Moments optimizer for Batch training (LAMB). Für die Klassifizierung der Daten wurde die Kreuzentropieverlustfunktion verwendet.

Inhaltsverzeichnug

1. Einführung

1.1. Motivation

Neuronale Netzwerke und die damit verbundene künstliche Intelligenz sind heutzutage allgegenwärtig. Sie sind in fast allen Lebensbereichen präsent. Sie erleichtern die Simulation komplexer Zusammenhänge wie Klimawandel, Forschung und Diagnose in der Medizin, finden sich aber auch in Consumer-Software zum Beispiel zur Objekt-, Sprach- und Bilderkennung. Durch das mit diesen Anwendungen einhergehende massive Datenaufkommen und dessen Verarbeitung und Analyse, ist der effiziente Einsatz von Optimierungsalgorithmen unerlässlich.

1.2. Projektziel

Ziel des Projekts ist es, die Optimieralgorithmen SGD, Adam und LAMB näher auf ihre Leistungsunterschied bei veränderter Stapelgröße und gleichbleibenden Trainingsdurchläufen zu untersuchen. Eine Bilderkennung mithilfe der drei trainierten Modelle ist nicht Bestandteil des Projekts.

1.3. Aufbau und Ablauf der Arbeit

Die Dokumentation beginnt mit einem Grundlagenteil, in welchem ein Grundverständnis zum Thema geschaffen werden soll. Die folgende Konzeption erklärt den Aufbau der Versuchsreihe. Im Abschnitt Implementation ist die Umsetzung des Konzepts erläutert und anhand von Queltextbeispielen verdeutlicht. Die Ergebnisse der Versuchsreihe werden im Kapitel Evaluation beschrieben. Abschließend werden die gewonnenen Erkenntnisse zusammengefasst und ein Ausblick auf kommende Forschung gegeben.

2. Grundlagen

2.1. Convolutional Networks

Convolutional Neural Networks erweitern die konventionellen feed forward neural networks. Der große Unterschied besteht in der verbesserten Mustererkennung in Farbbildern. Um dies zu erreichen, wird die Convolutional Operation angewendet.

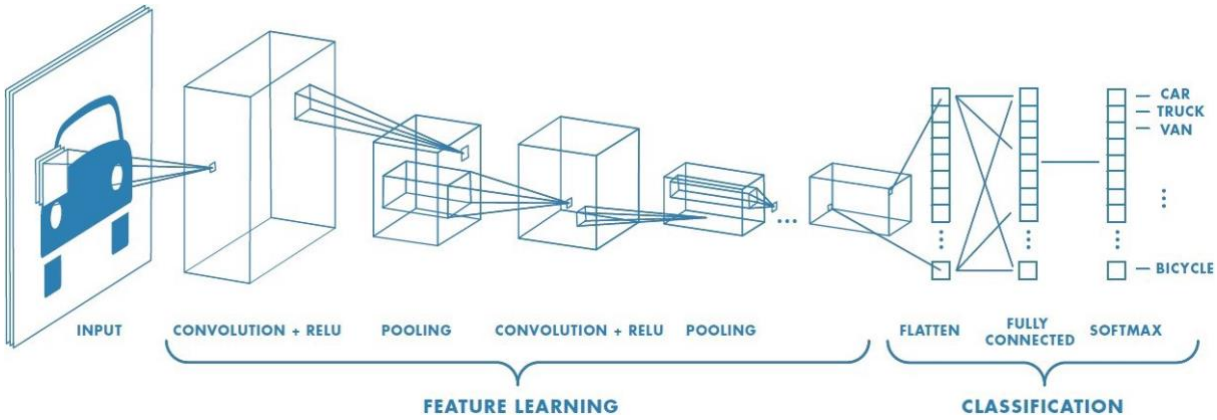

Abbildung 1: Grundlegende CNN Architektur (Saha 2018)

Die Convolutional Operation bezeichnet eine mathematische Faltungsoperation.

Bei dieser iteriert ein Filter f mit einer Schrittlänge s über Eingabedaten n und erzeugt daraus eine Merkmalskarte. Um einen Informationsverlust an den Kanten der Eingabedaten durch den Filter zu verhindern, kann ein Padding p um das Bild hinzugefügt werden. Die folgende pooling Operation verwirft überflüssige Daten der Merkmalskarte und reduziert dadurch die Dimension der Ausgabedaten.

$$\dim\big(pooling(image)\big) = \left(\left[\frac{n_H+2p-f}{s}+1\right], \left[\frac{n_W+2p-f}{s}+1\right], n_c\right) s > 0$$

(Mebsout n.d.)

Die neue Dimension des Bildes errechnet sich durch die Subraktion von Bildlänge n_H und Filtergröße f addiert mit dem Produkt aus 2 mal der Paddingröße p. Das Ergebnis wird dann durch die gewählte Schrittzahl s dividiert, sie muss daher größer als 0 sein. Gleiches gilt für die Bildbreite n_W. Die Farbkanalanzahl bleibt unverändert.

2.2. Optimierer

Optimierer (Contributors, PyTorch 2019) werden angewandt um die Verluste und Gewichtungen im backpropagation Prozess anzupassen.

2.2.1. Kreuzentropie

Die Verlustfunktion ist ein wichtiger Bestandteil neuronaler Netze. Sie berechnet die Genauigkeit des verwendeten Modells, indem sie zwei Wahrscheinlichkeiten vergleicht und somit den Gradientenabstieg beeinflusst.

$\frac{\partial}{\partial\theta} J\theta = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \log h_\theta\left(x^{(i)}\right) + (1 - y^{(i)})\log\left(1 - h_\theta\left(x^{(i)}\right)\right)]$
(Baumann 2021)

Da in der binären Klassifikation nur $[0\,,1]$ Werte zulässig sind, ist je nach Wert von $y^{(i)}$ ein Term vernachlässigbar. Die Formel vereinfacht sich somit auf einen Term, bei dem der berechnete Eingangswert $x^{(i)}$ mit dem wahren Wert $y^{(i)}$ logarithmiert wird. Die Summe wird dann durch die Anzahl der Trainingsbeispiele dividiert. Der vorangestellte Minusoperator minimiert den Verlust mit zunehmender Annäherung der beiden Wahrscheinlichkeitswerte $y^{(i)}, x^{(i)}$.

2.2.2. Stochastic Gradient Descent

Die Grundfunktionalität eines Gradientenabstiegsverfahrens besteht darin, den tiefsten Punkt einer mathematischen Funktion durch schrittweise Iteration zu finden. Um den tiefsten Punkt zu finden, wird ein zufälliger Startpunkt gewählt.

$w_{k+1} = w_k - \alpha_k * \nabla f_k\,(w_k)$ (Bottou, Curtis and Nocedal 2018)

Ausgehend von der Startposition wird für jede neue Position w_{k+1} das Produkt aus Lernrate α_{k+1} und Ergebnis des errechneten Gradienten aus der Verlustfunktion $\nabla f_k\,(w_k)$ von der aktuellen Position w_t subtrahiert. Das Multiplizieren eines konstanten Momentums β (Quian n.d.) mit der aktuellen Position w_t soll ein ein Steckenbleiben an Sattelpunkten verhindern und schnelleres Konvergieren des Gradienten ermöglichen.

$w_{k+1} = \beta * w_k - \alpha_k * \nabla f_k\,(w_k)$ (Quian n.d.)

Der Zusatz stochastisch bedeutet, dass bei jedem Durchgang nur die Daten der gewählten Stapelgröße verarbeitet werden und eine Aktualisierung nach jedem Durchgang vorgenommen wird. (Amir, Koren and Livni 2021)

2.2.3. Adaptive Moment Estimation

Adam verwendet eine adaptive Lernrate, basierend auf den vorherigen Ergebnissen. Er kombiert AdaGrad (Duchi, Hazan and Singer 2011) und RMSProp (Hinton, Srivastava and Swersky n.d.) zu einem leistungsstarken und effizienten Optimierer (Kingma and Lei Ba 2015). Der Algorithmus berechnet mit den beiden Konstanten $0 < \beta_1,\ \beta_2\ < 1$ die durchschnittlichen Momentenschätzungen und beschleunigt das Konvergieren des Gradienten. Der Gradient g_t bezieht sich auf das Ergebnis aus der Verlustfunktion. Hierbei bezeichnet m_t den geschätzten Durchschnitt der Schrittweite und v_t die Varianz des Gradienten des Durchlaufs t.

$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$ (Kingma and Lei Ba 2015)

$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2$ (Kingma and Lei Ba 2015)

Die Werte m_t und v_t müssen wegen der initialen Startwerte von 0 aktualisiert werden, da sonst der Gradient schneller gegen 0 konvergieren würde und zu ungenauen Ergebnissen führen würde.

$\widehat{m_t} = \frac{m_t}{(1-\beta_1^t)}$ (Kingma and Lei Ba 2015)

$\widehat{v_t} = \frac{v_t}{(1-\beta_2^t)}$ (Kingma and Lei Ba 2015)

Die Aktualisierungsfunktion berechnet die neue Position θ_{t+1} indem von der aktuellen Position θ_t, das Produkt aus Schrittweite α_t und dem Quotienten der aktualisierten Schrittweite $\widehat{m_t}$ und dem Normalisierungsparameter $\widehat{v_t}$, subtrahiert wird. Die ε Komponente ist ein minimaler Wert um eine eventuelle Divisionen durch 0 im Nenner zu verhindern.

$\theta_{t+1} = \theta_t - \alpha_t \frac{\widehat{m_t}}{(\sqrt{\widehat{v_t}} + \varepsilon)}$ (Kingma and Lei Ba 2015)

2.2.4. Layer-wise Adaptive Moments optimizer for Batch training

Der LAMB (You, et al. 2020) Algorithmus baut grundlegend auf Adam auf. Er verwendet daher ebenfalls adaptive Lernraten, trainiert aber durch Anwendung schichtweiser Lernratenoptimierung effizienter mit sehr großen Stapelgrößen. (You, et al. 2020). Darüberhinaus wird die Adaptivität schichtweise berechnet. Ebenfalls wie bei Adam, werden zwei Momenta errechnet: m_t^l bezieht sich auf die durchschnittliche Schrittweite und v_t^l bezeichnet die Varianz des Gradienten der Schicht l, der Index t bezieht sich auf den aktuellen Durchlauf und der Gradient g_t^l auf die aktuelle Position. Die konstanten Momenti $0 < \beta_1, \beta_2 < 1$ sind Komponenten in der Schätzwertebererchnung und dienen zur Verbesserung der Trainingswerte.

$m_t^l = \beta_1 * m_{t-1}^l + (1 - \beta_1) g_t^l$ (You, et al. 2020), (Science, et al. 2020)

$v_t^l = \beta_2 * v_{t-1}^l + (1 - \beta_2) g_t^l * g_t^l$ (You, et al. 2020), (Science, et al. 2020)

Hieraus ergibt sich, wie auch bei Adam, eine notwendige Korrektur von m_t^l und v_t^l.

$\widehat{m_t^l} = \frac{m_t}{(1-\beta_1^t)}$ (You, et al. 2020), (Science, et al. 2020)

$\widehat{v_t^l} = \frac{v_t}{(1-\beta_2^t)}$ (You, et al. 2020), (Science, et al. 2020)

Für jede Schicht l wird nun das Verhältnis von Schichtgewichten w_t^l und Gradientaktualisierrung g_t^l errechnet. Hierbei werden zu große oder zu kleine Unterschiede der Werte korrigiert um Instablilität oder Stagnation zu vermeiden. Daraus ergibt sich die schichtweise skalierte Lernrate r_t^l. Der Zähler r_1 enthält die aktuelle Position des Gradienten. (You, et al. 2020)

$r_1 = \phi(||w_{t-1}^l||)$ (You, et al. 2020), (Science, et al. 2020)

Um die Gewichtungen bei den Aktualisierung möglichst klein zu halten wird eine weitere Komponente r_2 hinzugefügt.

$$r_2 = \left\| \frac{\widehat{m_t^l}}{\widehat{v_t^l + \varepsilon}} + \lambda * w_{t-1}^l \right\|$$ (You, et al. 2020), (Science, et al. 2020)

Hierbei addiert sich zur normalisierten neuen Position $\frac{\widehat{m_t^l}}{\widehat{v_t^l + \varepsilon}}$ das Produkt aus Gewichtsverlustkonstante λ und aktueller Position w_{t-1}^l. Die Gewichtsverlustskonstante λ soll eine verbesserte Generalisierung bieten und Neugewichtungen möglichst klein halten um potentielle Steigungen des Gradienten zu vermeiden. So ergibt sich für den Vertrauensverhältniswert r_t^l folgende Formel:

$$r_t^l = \frac{r_1}{r_2}$$

Daraus ergibt sich folgende Aktualisierungsfunktion für den LAMB Algorithmus.

$$w_t^l = w_{t-1}^l - \eta * r_t^l * \left(\frac{\widehat{m_t^l}}{\widehat{v_t^l + \varepsilon}} + \lambda * w_{t-1}^l \right)$$

(You, et al. 2020), (Science, et al. 2020), (Quian n.d.)

Der Term $\eta * r_t^l$ passt die Lernrate adaptiv für die Schichten an.

3. Konzeption

Die Versuchsreihen wurde auf einem Laptop Computer der Marke Apple MacBook Air mit 16 GB Ram und einem 1,6 GHz Dual-Core Intel Core i5 Prozessor und einer eingebauten Grafikkarte Intel UHD Graphics 617 mit 1536 MB Speicher. Diese bietet laut Hersteller keine Nvidia CUDA Unterstützung an.

Die verwendete Bilddatenbank ist der CIFAR-10 Datensatz (Krizhevsky, Nair and Hinton 2009). Die Hyperparameter wurden für die erste

Versuchsreihe auf 64 Bilder pro Stapel und 20 Epochen pro Trainigsdurchlauf festgelegt. Bei der zweiten Versuchsreihe wurde die Stapelgröße auf 512 bei gleichbleibender Anzahl von Epochen gesetzt. Für jeden Optimierer wurden jeweils drei komplette Durchgänge gespeichert um daraus den Durchschnitt zu errechnen. Die Lernrate wurde bei beiden Versuchsreihen beim Standardwert von 0.001 belassen. Als Modell diente ein siebenschichtiges CNN.

4. Implementation

4.1. Hauptmenü

Das Hauptmenü bietet eine numerische Menüauswahl an. Hier werden die Optimierer (Contributors, PyTorch 2019), die Gesamtübersicht der Benchmarks, oder das Verlassen des Programms ausgewählt. Alle Optimiereroptionen verwenden die gleichen oben genannten Hyperparameter zum Modelltraining. Die beiden Benchmarkoptionen stellen die Verlaufskurven aller trainierten Optimierer und deren Validierungserluste sowie deren Validierungsgenauigkeiten als Verlaufskurven in getrennten Schaubildern dar. (Hunter, et al. 2021)

4.2. Service

Bei der Initializierung des Serviceobjekts lädt PyTorch (Contributors, PyTorch 2019) einmalig die Daten der CIFAR-10 Bilddatenbank (Krizhevsky, Nair and Hinton 2009) lokal auf den Rechner. Die Bilder werden normalisiert, zufällig nach der angegebenen Stapelgröße gemischt und lokal im Programmordner gespeichert.

```
def __init__(self):
    ...
    self.training_data = torch.utils.data.DataLoader(
        datasets.CIFAR10(root='./data', train=True, download=True,
          transform=self.transform), self.batch_size, shuffle=True)
    ...
```

In der Trainingsfunktion wird das Modell mit den Daten der Bilderdatenbank trainiert. Die Anzahl der Epochen bestimmt die Durchläufe durch die Datenbank. In jedem Durchgang wird aus dem festgelegten Stapel ein zufällig ausgewähltes Bild ausgewählt, dem Modell übergeben und der retunierte Tensor der forward() Funktion in der Kreuzentropiefunktion klassifiziert. Vor dem Backpropagation Prozess werden alle Gradienten auf Null gesetzt. Im Backpropagation Prozess übernimmt der gewählte Optimierer die Neugewichtung der Kanten und die Aktualisierung der Parameter. Zuletzt wird der Trainingsverlust des Durchgangs von einem PyTorch Tensor (Contributors, PyTorch 2019) zu einem Python integer (Foundation, Python 2021) konvertiert und in jedem weiteren Durchgang aufaddiert. (siehe Anhang A)

Im Validierungsprozess wird das komplette CIFAR-10 Validierungsset mit dem trainierten Modell durchlaufen. Hierfür wird die Gradientenberechnung ausgesetzt und jedes Bild desSets an die forward() Funktion des CNN‘s übergeben. Aus dem Rückgabewert klassifiziert die Kreuzentropiefunktion das Bild und errechnet den Validierungsverlust. Dieser Verlust wird aufaddiert, um am Validierungsende aus Verlust und Setgröße, den Quotienten als Listenelement für die spätere Zusammenfassung zu speichern.
Die korrekt erkannten Bilder werden von einem PyTorch Tensor Datentyp (Contributors, PyTorch 2019) zu einem Python integer (Foundation, Python 2021) konvertiert. Sind alle Epochen durchlaufen, wird aus der Summe der korrekt erkannten Bilder und allen Bildern im Validierungsset der Prozentsatz als Listenelement für die Erstellung des Genauigkeitsgraphen gespeichert. (siehe Anhang B)

4.3. Modell

In der Modellklasse wird die Struktur des CNN’s festgelegt. Es umfasst vier sequentiell angeordnete CNN Schichten und drei sequentielle lineare Schichten. Jede convolutional Schicht erwartet eine feste Eingabegröße, sowie eine feste Ausgabegröße; diese umfasst die Merkmalskarte welche die erkannten Merkmale des Filters enthält. Die Filtergröße ist als 3x3 Pixelmatrix angelegt, welche sich jeweils um einen Pixel verschiebt. Das padding wurde auf einen Pixel mit dem Wert 0 gesetzt um die Pixelmatrix bei der Eingabegröße des 32x32 Bildes um den Filter auch an den Rändern

anwenden zu können. Jede CNN-Schicht verarbeitet die ausgegebene Merkmalskarte des Filters in einer Batch Normalisierung (Contributors, PyTorch 2019) und eine ReLU Aktivierungsfunktion (Contributors, PyTorch 2019). Auf jeder zweiten convolutional Schicht wird eine MaxPool2d (Contributors, PyTorch 2019) und Dropout (Contributors, PyTorch 2019) Operation angewendet.

Die Batch Normalisierung gleicht große und kleine Schwankungen der Merkmalskarten aus, um den Lernprozess zu beschleunigen und zu verbessern. Die ReLU Aktivierungsfunktion gibt für jede negativen Wert eine 0, ansonsten eine 1 aus . Sie ist effizient, konvergiert schnell und verhindert eine Sättigung des Gradienten. Die Pooling-Funktion ist auf eine 2x2 Matrix gesetzt welche nur den Maximalwert weitergibt und somit die Dimension der Ausgabe gegenüber der Eingabe verringert und das Training beschleunigt. Die Dropout Funktion setzt mit der angegebenen Wahrscheinlichkeit einen zufällig gewählten Eingabewert auf 0, dies verbessert die Genauigkeit des Trainings. (siehe Anahng C)

4.4. Speicherung

Die Speicherung der einzelnen Trainingsergebnisse und Validierungswerte erfolgt in einem separaten Ordner im Programmpfad. Sie werden in zwei seperaten Dateien im standardisierten csv Format (Foundation, Python 2021) gespeichert.

Das Lesen der csv Dateien erfolgt zeilenweise. Da sich die Verlustwerte aus Einzelwerten der durchlaufenen Epochen ergeben, enspricht jede Zeile einem Durchlauf, dessen Elemente in einer Liste gespeichert werden. Um die Werte nachher als Kurve darzustellen werden die Listen als Listenelement einer weiteren Liste abgelegt. (siehe Anhang D)

Die gemessenen Genauigkeitswerte sind Endergebnisse der Durchläufe und können daher als Elemente in einer einfachen Liste abgespeichert werden.

4.5. Berechnungen

Um den Durchschnitt jeder Trainingsepoche zu errechnen, werden die Trainingsdatenlisten spaltenweise eingelesen und die Summe aller Elemente durch die Anzahl der vorhandenen Zeilen dividiert. Anschließend werden alle Werte in einer neuen Liste gespeichert. (siehe Anhang E)

Für die Berechnungen des Verlusts und Prozensatzes aus den Validierungen, werden alle Elemente aus der erhaltenen Liste addiert, ebenfalls durch die Anzahl aller Zeilen beziehungsweise Durchläufe geteilt und in einer neuen Liste gespeichert.

4.6. Plotter

Die Plotterklasse erhält die Liste mit den berechneten Durchschnittswerten aus der Calculator Klasse und zeichnet daraus die Kurven. Die Verlustwerte aus dem Training und der Validierung werden zur besseren Veranschaulichung in einem Schaubild zusammengefasst. Hierbei zeigt jeder Punkt den Durchschnittswert eines Durchgangs an. Die Genauigkeitswerte aller Optimierer werden in einem separaten Schaubild angezeigt. Nach der Anzeige werden die Graphen in einem separaten Ordner im Programmordner als getrennte Bilddateien abgespeichert.

5. Evaluation

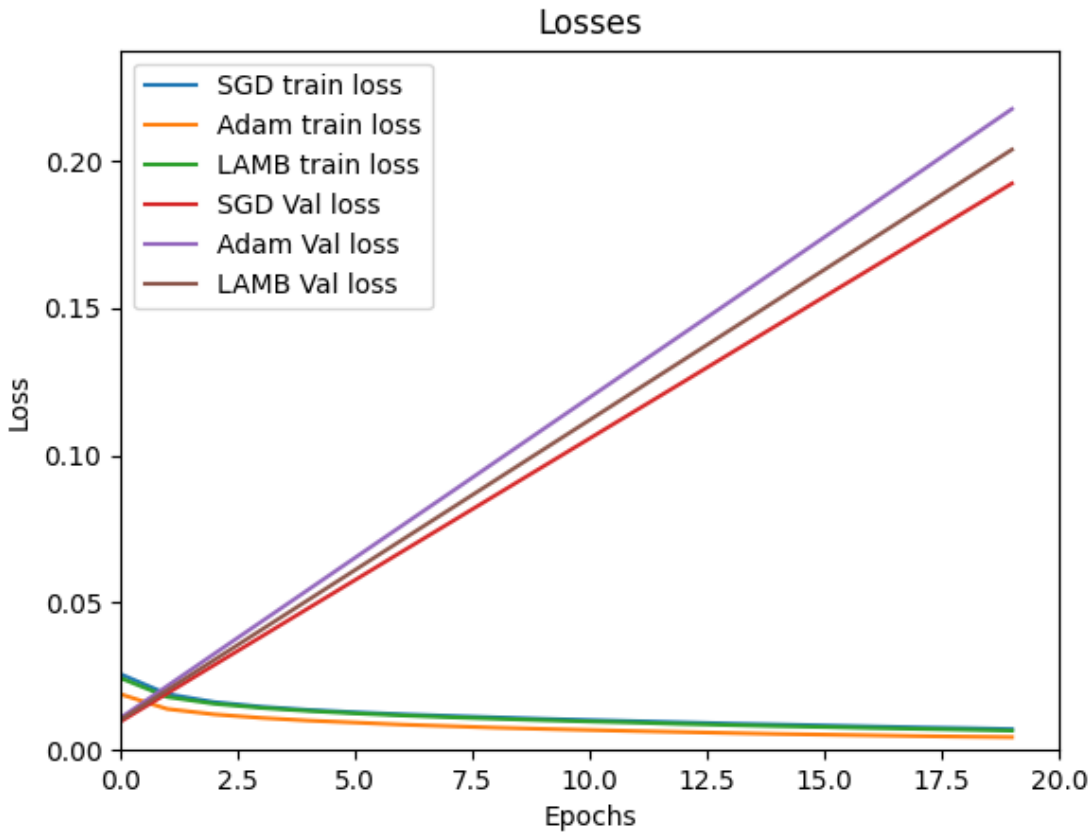

Abbildung 2: Durchschnittliche Verlustwerte aller Optimierer bei 64 Bildern pro Stapel und drei kompletten Trainingsdurchläufen.

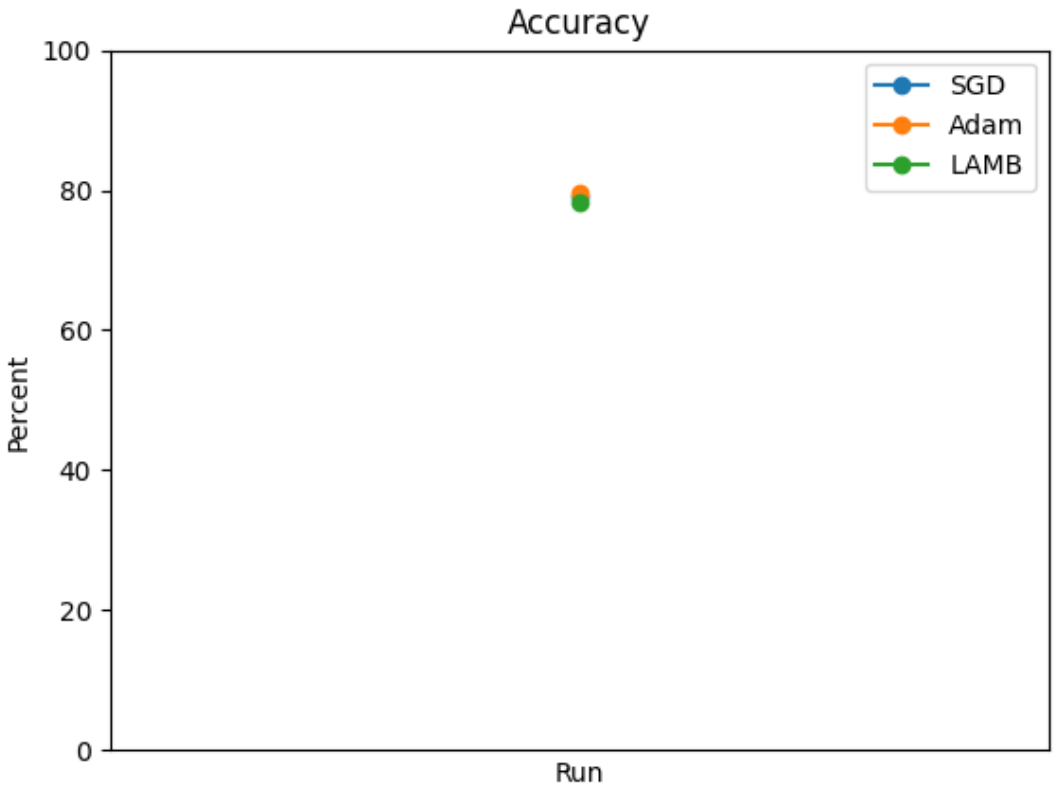

Abbildung 3: Durchschnittliche Validierunggenauigkeiten der Optimierer bei 64 Bilderrn pro Stapel und drei kompletten Durchläufen.

	Trainingsverlust	Validierungsverlust	Genauigkeit
SGD	0,007091	0,192426	79,19%
Adam	0,004356	0,217715	79,58%
LAMB	0,006589	0,203922	78,36%

Tabelle 1: Übersicht aller Durchschnittswerte bei einer Stapelgröße von 64 CIFAR-10 Bildern

Vergleicht man die angewendeten Optimierer bei der relativ kleinen Stapelgröße von 64 Bildern, fällt auf, dass SGD + Momentum und Adam nahezu identische Ergebnisse liefern. Nur LAMB hat ein geringfügig schlechteres Ergebnis. Die Verhältnisse des Trainingsverlusts zum Validierungsverlust bewegen sich für SGD bei 1:27, für Adam bei 1:49 für LAMB bei 1:30.

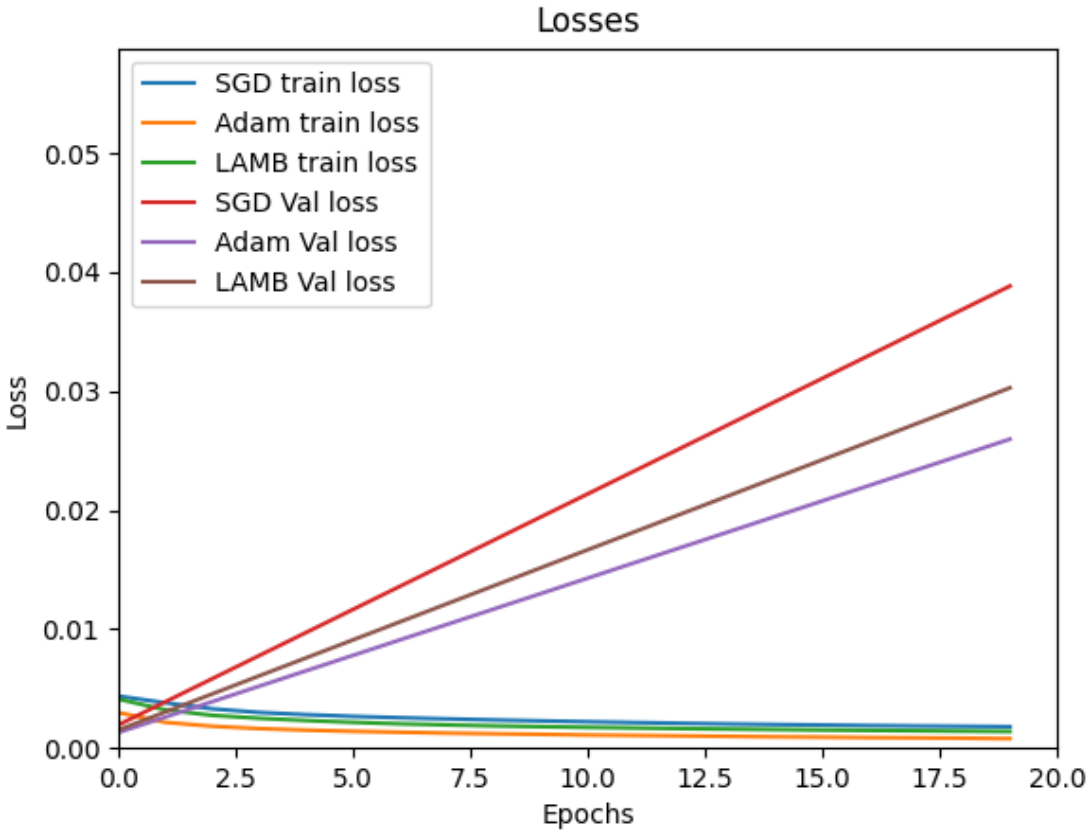

Abbildung 4: Durchschnittliche Verlustwerte aller Optimierer bei 512 Bildern pro Stapel und drei kompletten Trainingsdurchläufen.

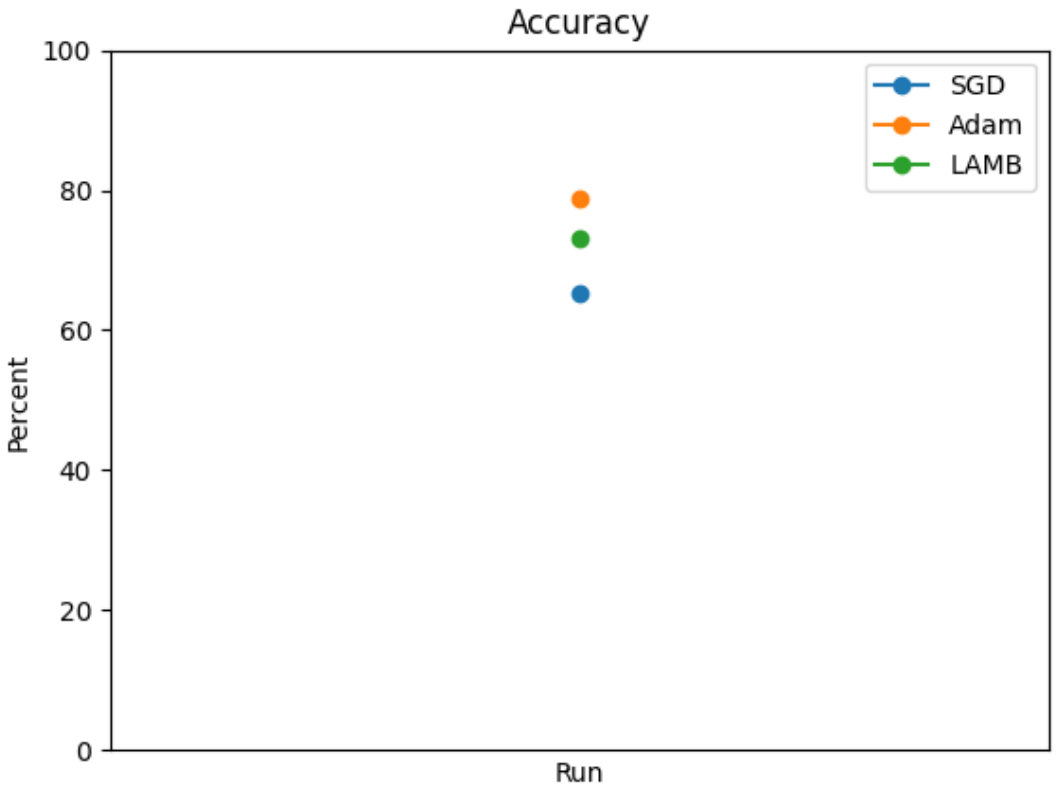

Abbildung 5: Durchschnittliche Validierunggenauigkeiten der Optimierer bei 512 Bilderrn pro Stapel und drei kompletten Durchläufen.

	Trainingsverlust	Validierungsverlust	Genauigkeit
SGD	0,001790	0,038822	65,38%
Adam	0,000787	0,025956	78,91%
LAMB	0,001390	0,030269	73,23%

Tabelle 2: Übersicht aller Durchschnittswerte bei einer Stapelgröße von 512 CIFAR-10 Bildern

Nach Erhöhung der Stapelgröße verliert SGD signifikant an Genauigkeit und die Generalisierungslücke verringert sich auf 1:21. Adam liefert eine gleichbleibende Genauigkeit, verbessert aber die Generalisierungslücke signifikant auf 1:32. Der LAMB Optimierer hat nun eine geringere Genauigkeit, verringert aber die Generalisierungslücke ebenfalls signifikant auf 1:21.

6. Fazit

6.1. Zusammenfassung

Betrachtet man die Ergebnisse der beiden Trainingsdurchläufe, ist ersichtlich, dass die Trainings- und Validierungsergebnisse signifikant auseinanderliegen. Dies deutet auf ein übertrainiertes Netzwerk hin. Betrachtet man die erste Versuchsreihe, so lässt sich erkennen, dass alle Optimierer fast identische Genauigkeiten liefern, aber große Generalisierungslücken aufweisen, was sich am Verhältnis von Trainingsverlust zu Validierungsverlust ablesen lässt (Abb.: 2). Die schlechten Werte von Adam und LAMB gegenüber SGD lassen sich darauf zurückführen, dass adaptive Verfahren zwar schneller im Training konvergieren, aber bei der Validierung ebenfalls schneller ein Plateau erreichen, wohingegen SGD bei gleichbleibender Schrittzahl langsamer komvergiert, aber bessere Validierungen liefert. (Zhou, et al. n.d.)

Betrachtet man nun die zweite Versuchsreihe, bleibt der SGD Algorithmus zwar in der Generalisierung stabil, liefert aber jetzt eine viel schlechtere Genauigkeit, wohingegen die Optimierer Adam und LAMB etwas geringere Genauigkeiten aufweisen, aber die Generalisierungslücke minimierern. Daraus wird ersichtlich, dass die Algorithmen Adam und LAMB, bei größer werdender Stapelgröße durch Verwendung adaptiver Lernraten eine verbesserte Leistung gegenüber SGD liefern. Es zeigt aber auch, dass LAMB keinen großen Leistungssprung gegenüber Adam hervorbringt.

6.2. Ausblick

Hinsichtlich der stetigen Verbesserung der Rechenleistung, Vergrößerung der Rechenzentren, Forschungen im Bereich Neuronaler Netze, aber auch der rasant anwachsenden Datenmengen, werden eine Weiterentwichklung und Verbesserung der aktuellen Optimierer unabdingbar machen. Darüberhinaus muss auch die ethische Komponente in Betracht gezogen werden, da KI-Systeme nicht nur positiven Einfluss auf die Gesellschaft haben können, sondern auch missbraucht werden kann. Somit bleibt die Verantwortung bei den Entwicklern und Firmen mit ihrer Forschung die Wirkung und Auswirkung auf die Gesellschaft positiv zu gestalten.

7. Abbildungsverzeichnis

8. Tabellenverzeichnis

9. Quellenverzeichnis

-

10. Abkürzungsverzeichnis

A

Adam ..... Adaptive Moment Estimation

C

CNN ..... *Convolutional Neural Network*, *Convolutional Neural Network*
csv ..... comma-separated values

K

KI ..... *Künstliche Intelligenz*

L

LAMB ..... Layer-wise Adaptive Moments optimizer for Batch training

R

ReLU ..... *Rectified Linear Unit*

S

SGD ..... *Stochastic Gradient Descent*

11. Onlinequellen

Saha, Sumit. 2018. *towards data science.* 15. 12. Zugriff am 08. 07 2021. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Kingma, Diederick P., und Jimmy Lei Ba. 2015. *arXiv.org.* Zugriff am 05. 07 2021. https://arxiv.org/pdf/1412.6980.pdf.

Foundation, The Python Software. 2021. *Python.* 08. 07. Zugriff am 08. 07 2021. https://docs.python.org/3/library/csv.html.

Krizhevsky, Alex, Vinod Nair, und Geoffrey Hinton . 2009. *The CIFAR-10 dataset.* Zugriff am 07. 07 2021. https://www.cs.toronto.edu/~kriz/cifar.html.

Hunter, John, Darren Dale, Eric Firing, Michael Droettboom, und Matplotlib Development Team. 2021. *matplotlib.org.* 08. 05. Zugriff am 07. 07 2021. https://matplotlib.org.

O'Shea, Keiron, und Ryan Nash. 2015. *arVix.org.* 02. 12. Zugriff am 05. 07 2021. https://arxiv.org/pdf/1511.08458.pdf.

Novik, Nikolai. 2020. *pytorch-optimizer.* Zugriff am 12. 07 2021. https://pytorch-optimizer.readthedocs.io/en/latest/api.html#lamb.

Contributors, Torch. 2019. *PyTorch.* Zugriff am 07. 07 2021. https://pytorch.org/docs/stable/optim.html.

—. 2019. *PyTorch.* Zugriff am 12. 07 2021. https://pytorch.org/docs/stable/data.html.

—. 2019. *PyTorch.* Zugriff am 17. 08 2021. https://pytorch.org/docs/stable/tensors.html.

Foundation, The Python Software. 2021. *Python .* 16. 08. Zugriff am 17. 08 2021. https://docs.python.org/3/library/stdtypes.html.

Contributors, Torch. 2019. *PyTorch.* Zugriff am 2021. 08 17. https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html?highlight=batchnorm#torch.nn.BatchNorm2d.

—. 2019. *PyTorch.* Zugriff am 17. 08 2021. https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html?highlight=relu#torch.nn.ReLU.

Mebsout, Ismail. kein Datum. *towards data science.* Zugriff am 2021. 08 17. https://towardsdatascience.com/convolutional-neural-networks-mathematics-1beb3e6447c0.

Contributors, Torch. 2019. *PyTorch.* Zugriff am 18. 08 2021. https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html.

—. 2019. *PyTorch.* Zugriff am 18. 08 2021. https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html?highlight=maxpool2d#torch.nn.MaxPool2d.

Duchi, John, Elad Hazan, und Yoram Singer. 2011. *Journal of Machine Learning Research.* 07. Zugriff am 18. 08 2021. https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

Hinton, Geoffrey, Nitish Srivastava, und Kevin Swersky. kein Datum. https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf.

You, Yang, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, und Cho-Jui Hsieh. 2020. *arxiv.* 03. 01. Zugriff am 20. 08 2021. https://arxiv.org/pdf/1904.00962.pdf.

Amir, Idan, Tomer Koren, und Roi Livni. 2021. *arxiv.* 01. 07. Zugriff am 20. 08 2021. https://arxiv.org/pdf/2102.01117v2.pdf.

Baumann, Patrick. 2021. *HTW moodle.* Zugriff am 21. 08 2021. https://moodle.htw-

berlin.de/pluginfile.php/1163173/mod_resource/content/3/03_Logistic_Regression.pdf.

Science, NUS Department of Computer, Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, et al. 2020. *youtube.com.* 22. 09. Zugriff am 24. 08 2021. https://www.youtube.com/watch?v=kwEBP-Wbtdc&t=416s.

Quian, Ning. kein Datum. *http://citeseerx.ist.psu.edu/index.* Zugriff am 27. 08 2021. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.5612&rep=rep1&type=pdf.

Bottou, L´eon, Frank E. Curtis, und Jorge Nocedal. 2018. *arxiv.* 08. 02. Zugriff am 08. 09 2021. https://arxiv.org/pdf/1606.04838.pdf.

Zhou, Pan, Jiashi Feng, Chao Ma, Caimin Xiong, Steven HOI, und Weinan E. kein Datum. *NeurIPS Proceedings.* Zugriff am 08. 09 2021. https://proceedings.neurips.cc/paper/2020/file/f3f27a324736617f20abbf2ffd806f6d-Paper.pdf.

12. Bildquellen

Saha, Sumit. 2018. *towards data science.* 15. 12. Zugriff am 08. 07 2021. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

13. Anhang

Anhang A:

```
def training(self, optimizer, optimizer_name, learning_rate, epochs):
    ...
    for epoch in range(epochs):
        train_loss = 0
        for images, labels in self.training_data:
            images, labels = images.to(self.device),
                             labels.to(self.device)
```

```
            output = self.model(images)
            loss = self.criterion(output, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        self.training_loss.append(train_loss /
                            len(self.training_data.dataset))
        print(f'Epoch {epoch} - Training loss: {train_loss /
                    len(self.training_data.dataset):.10f}')
```

Anhang B:

```
def validate_cifar(self, epochs):
    ...
    for _ in range(epochs):
        with torch.no_grad():
            for images, labels in self.validation_data:
                images, labels = images.to(self.device),
                                        labels.to(self.device)
                outputs = self.model(images)
                valid_loss += self.criterion(outputs, labels).item()
                predicted = outputs.argmax(1, True)
                correct += predicted.eq(labels.view_as
                            (predicted)).sum().item()
            self.validation_loss.append(valid_loss /
                                len(self.validation_data.dataset))
    self.validation_accuracy.append((100.0 * correct /
                    len(self.validation_data.dataset) / epochs))
    ...
```

Anhang C:

```
def __init__(self):
    super().__init__()
```

```
    self.conv_layer1 = Sequential(
        Conv2d(3, 24, 3, 1, 1), BatchNorm2d(24), ReLU(), MaxPool2d(2),
        Conv2d(24, 24, 3, 1, 1), BatchNorm2d(24), ReLU(), MaxPool2d(2),
        Conv2d(24, 32, 3, 1, 1), BatchNorm2d(32), ReLU(),MaxPool2d(2),
        Conv2d(32, 64, 2, 1, 1), BatchNorm2d(64), ReLU(), MaxPool2d(2),
    self.linear_layer = Sequential(Linear(64 * 4 * 4, 128), ReLU(),
        Linear(128, 64), ReLU(),
        Linear(64, 10))
```

Anhang D:

```
def load_loss_csv(self, optimizer_name, kind, batch_size):
    loss_list = []
    try:
        with open(f'{self.save_path}{optimizer_name}_
                  {kind}_{batch_size}.csv', newline='') as csv_loss:
            loss = csv.reader(csv_loss, delimiter=',',
                                    quoting=csv.QUOTE_NONNUMERIC)
            number_of_lists = 0
            data_list = []
            for row in loss:
                data_list.append(row)
                number_of_lists += 1
        loss_list.append(data_list)
    ...
```

Anhang E:

```
def calc_loss_average(self, loss_list, number_of_lists):
        average_list = []
        for j in range(len(loss_list[0][0])):
            element = 0.0
            for i in range(len(loss_list[0])):
                element += float(loss_list[0][i][j])
            average_list.append(element / float(number_of_lists))
        return average_list
```