Freie Universität Berlin

Fachbereich Mathematik und Informatik

Institut für Mathematik

Bachelorarbeit

Gradienten Verfahren zur Optimierung Neuronaler Netze

Name: Linus Henning

Matrikelnummer: 5114931

Betreuer: Prof. Dr. Volker John

Zweitgutachter: PD Dr. Alfonso Caiazzo

Berlin, 12. Oktober 2021

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Zielsetzung

Gerade in den letzten Jahren steigt das Interesse an Machine Learning Modellen stark an. Einer der Gründe hierfür ist die kontinuierliche Weiterentwicklung von Hardware sowie die zunehmende Digitalisierung. Zusammen sorgt dies für einen stetig wachsenden Anwendungsbereich. Heute werden Machine Learning Modelle in ganz unterschiedlichen Situationen genutzt, von autonom fahrenden Autos bis zur Erkennung von mentalen Krankheiten anhand von Audiodateien.

Im Jahr 1999 definierte der Mathematiker Tom Mitchel Maschine Learning wie folgt:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [1]

Diese Definition findet in zahlreichen Lehrveranstaltungen Anwendung und ihre vage Formulierung lässt bereits die Größe und Vielfältigkeit des Themenbereichs erahnen. In der Praxis kommen Modelle wie Supported Vektor Machiens, Decision Trees und logistische Regression zum Einsatz. Sie unterscheiden sich sowohl im Aufbau als auch in der Aufgabe, die sie zu lösen versuchen.

Es ist gängig, die verschiedenen Algorithmen nach ihrer Aufgabenstellung zu kategorisieren. Man unterscheidet in diesem Falle zwischen überwachtem und unüberwachtem Lernen. Ersteres benötigt Trainingsdaten, die aus mehreren Features und einem Label bestehen. Ziel eines Algorithmus aus dieser Kategorie ist es, für gegebene Features möglichst exakte Vorhersagen für das Label zu treffen. Im Vergleich dazu bestehen die Trainingsdaten beim unüberwachten Lernen nur aus Features und das Ziel ist es, Muster innerhalb der Trainingsdaten auszumachen.

Neuronale Netze sind ein Beispiel für einen Algorithmus aus dem Bereich des überwachten Lernens. In der Praxis finden sie gerade wegen ihrer Zuverlässigkeit sehr häufig Anwendung. Ziel dieser Arbeit ist es, eine Einführung in den Aufbau Neuronaler Netze und ihre zugrunde liegenden Verfahren zu geben. Hierbei sollen stets begleitende Beispiele präsentiert werden.

Zusätzlich soll der Fokus auf der Vorstellung des Gradienten Verfahrens liegen. Als Methode für die Lösung von Optimierungsproblemen ist es eine Art Basisverfahren. Variationen davon wie das Stochastische Gradienten Verfahren oder das Mini Batch Gradienten Verfahren werden ebenfalls betrachtet und hinsichtlich Konvergenzgeschwindigkeit und die Schrittkosten untersucht.

1.2 Historie Neuronaler Netze

Die erste Idee eines Neuronale Netz entstand 1943. Der Neurowissenschaftler Warren McCulloch und der Mathematiker Walter Pitts veröffentlichten ein Paper mit dem Titel „*A logical calculus of the ideas immanent in nervous activity*“. Hier entwickelten beide ein Modell, dass mittels Stromkreise die Funktionsweise eines echten Neurons simuliert. [8]

1956 veranstaltete das Dartmouth College den ersten Workshop zu dem Thema Künstliche Intelligenz. Unter dem Namen „*Dartmouth Summer Research Project on Artificial Intelligence*“ kamen Wissenschaftler zusammen, um sowohl über theoretische Aspekte als auch mögliche praktische Umsetzungen zu sprechen.

Ein weiterer Meilenstein in der Entwicklung Neuronaler Netze gelang Frank Rosenblatt im Jahre 1958. Der Psychologe konstruierte basierend auf den Ideen von McCulloch und Pitts den ersten Perceptron, welcher im einführenden Beispiel näher beschrieben wird. Neu hierbei ist vor allem die Idee, die Gewichte des Perceptron zu algorithmisch zu berechnen. [9]

2 Einführung Neuronaler Netze

2.1 Beispiel Perceptron

In diesem Abschnitt werden wir anhand von fiktiven Daten ein Neuronales Netz erstellen. Angenommen wir wollen Hunde und Katzen nur anhand ihrer Schultergröße und ihres Gewichts kategorisieren. Unsere Aufgabe ist es, ein Modell zu finden, dass für eine gegebene Schulterhöhe und ein gegebenes Gewicht vorhersagt, ob es sich um eine Katze oder einen Hund handelt.

Schulterhöhe in cm	Gewicht in kg	Tierart
23	3	Katze
56	20	Hund
43	12	Hund
33	4,5	Katze
25	9	Hund
41	6	Katze
24	2,5	Katze
70	70	Hund
17	2	Katze
25	8	Hund

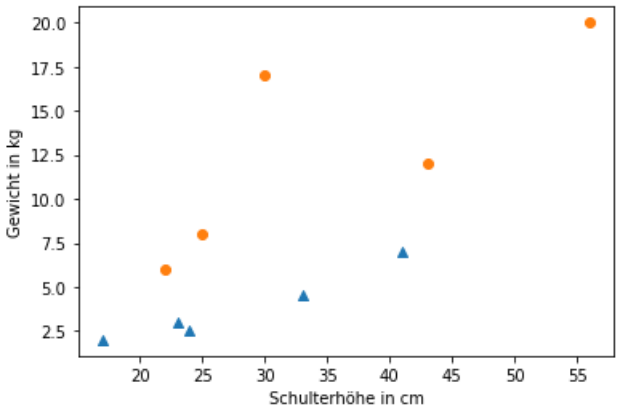

Abb.1: Plot Hunde und Katze. Kreise stehen für Hunde, Dreiecke für Katzen

Eine Möglichkeit hierfür sind die Neuronalen Netze. Bevor wir aber damit beginnen können, müssen wir eine Aktivierungsfunktion einführen. Sie ist das Kernelement eines jeden Neuronalen Netzes und simuliert das Verhalten eines echten Neurons. Was genau eine Aktivierungsfunktion ist, werden wir in einem späteren Abschnitt definieren. In diesem Fall beginnen wir mit einem Beispiel: der Sigmoidfunktion:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Für $x \leq 0$ bildet die Sigmoidfunktion annähernd auf 0 ab und für $x > 0$ annähernd auf 1. Sie ist also als eine glatte Form einer Treppenfunktion zu verstehen. Ein Gewichten oder Schiften des Inputwerts sorgt dafür, dass der Wechsel der Sigmoidfunktion von 0 zu 1 steiler oder verschoben wird. Wie in Abb. 1 und Abb. 2 zu erkennen.

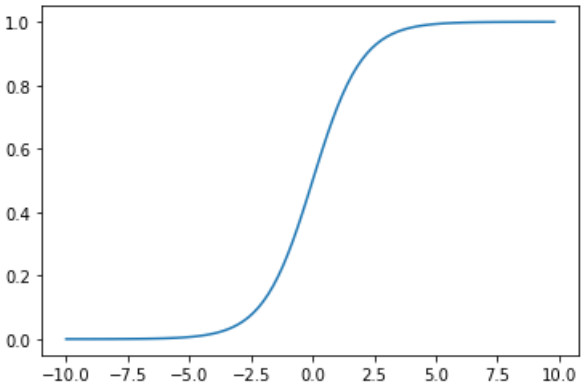

Abb. 2: $\sigma(x)$

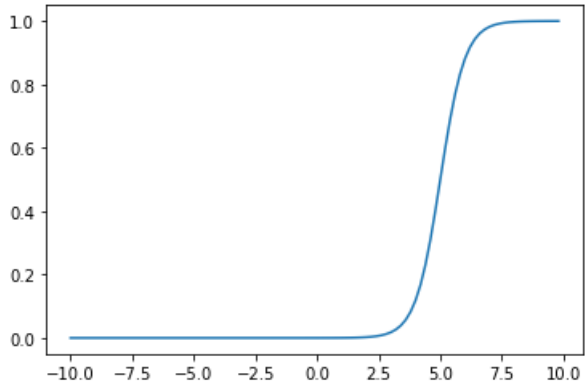

Abb. 3: $\sigma(2(x+5))$

In der Biologie funktioniert ein Neuron auf eine ähnlich Art. Erst bei einem ausreichend großen Input wird es aktiviert und sendet ein Signal.

Basierend auf der Sigmoidfunktion werden wir ein simples Neuronales Netz konstruieren. Da wir nur einen Knoten beziehungsweise ein Neuron verwenden, spricht man von einem Perceptron.
Ein Perceptron bekommt einen Inputvektor $a \in \mathbb{R}^p$. In unserem Beispiel haben wir nur die Schulterhöhe in cm und das Gewicht in kg gegeben haben, also ist $p = 2$. Jeder Eintrag von a wird mit einem Gewicht (Weight) w_i , $i \leq 2$ multipliziert. Anschließend summieren wir:

$$w_1 a_1 + w_2 a_2 + b$$

Wobei b ein vom Input unabhängiger Parameter (Bias) ist. Im letzten Schritt wenden wir die Sigmoidfunktion an. Der Output des Perceptron ist damit definiert als:

$$\sigma(w_1 a_1 + w_2 a_2 + b) = \sigma\left(\sum_{k=1}^{2} w_k a_k + b\right)$$

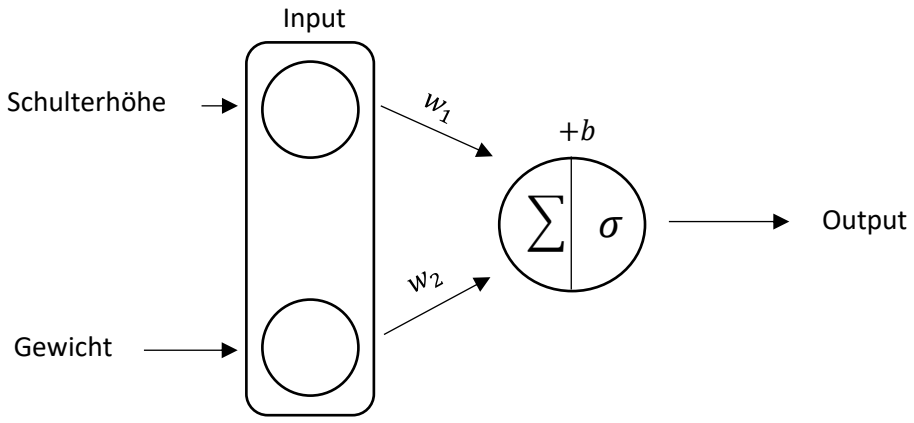

Abb 4: Aufbau Perceptron

Ein Perceptron ist in der Lage, diese simple Aufgabe zu lösen. Allerdings werden komplexere Aufgabenstellungen dazu führen, dass ein Perceptron allein nicht mehr ausreichend genau vorhersagt. Die Idee der Neuronalen Netzen ist, eine Vielzahl an Perceptron zu kombinieren, um auch diese Probleme lösen zu können.

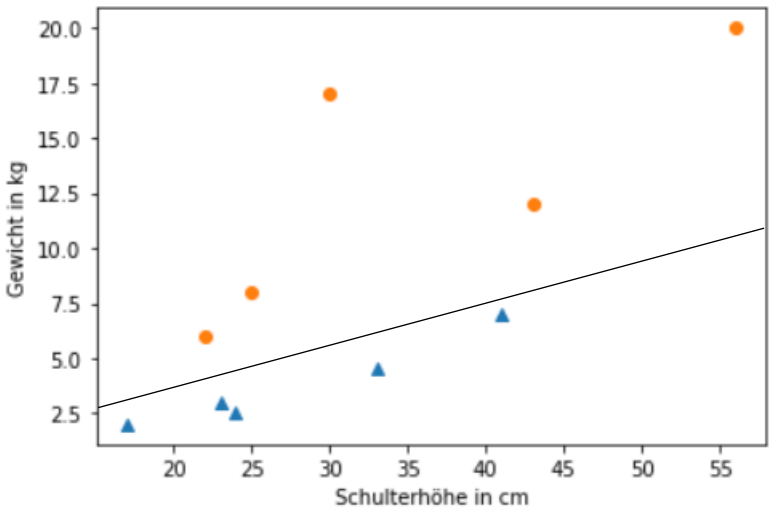

Abb. 5: Klassifizierung des Perceptron mit $w_1 = 1{,}2;\ w_2 = -0{,}2$ und $b = -1{,}4$

2.2 Aufbau Neuronaler Netze

2.2.1 Notation

Neuronale Netze bestehen aus mehreren Ebenen, sogenannten Layern. Input- und Outputlayer haben wir schon bei dem Perceptron kennen gelernt. Allerdings können bei Neuronalen Netzen beliebig viele weitere Layer zwischen dem Input- und Outputlayer existieren. Man spricht hier von den Hiddenlayern.

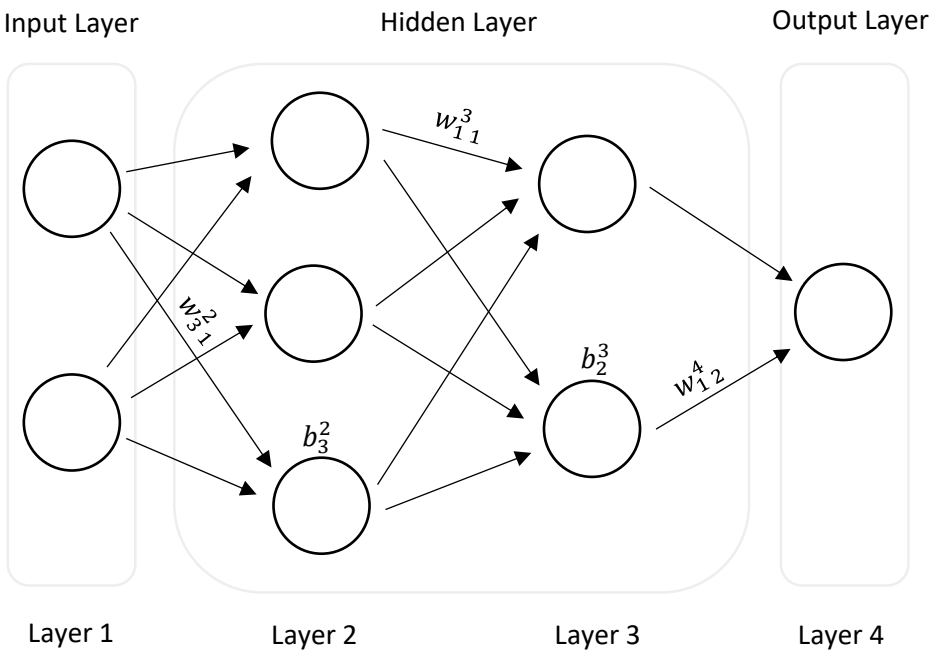

Abb. 6: Beispiel eines Neuronalen Netzes mit mehreren Layern

Jedes einzelne Layer verfügt dabei über beliebig viele Knoten, die vom Aufbau her einem Perceptron gleichen. Das bedeutet, jeder Knoten bekommt als Input den Output des vorherigen Layers und berechnet dazu einen eigenen Output. Wie auch bei einem Perceptron besitzt jeder Knoten Weights- und Biaswerte. Diese werden benutzt, um den Output zu berechnen. Für ein Neuronales Netz N führen wir folgenden Notation ein:

Sei L die Anzahl an Layern in N. Die Anzahl der Knoten im l-ten Layer nennen wir n_l, wobei $0 < l \leq L$.

Sei a_j^i der Output der Aktivierungsfunktion des j-ten Neurons in dem i-ten Layer. Der Vektor a^1 sei hierbei der Input.

$$a_j^i = \sigma\left(\sum_k^{n_{i-1}} (w_{jk}^i a_k^{i-1}) + b_j^i\right)$$

w_{jk}^i sei das Gewicht des k-ten Neurons im $(i-1)$-ten Layer für den j-ten Neuron des i-ten Layers.

b_j^i sei das Bias-Wert des j-ten Neurons im i-ten Layer.

Zur besseren Übersicht definieren wir außerdem:

$$z_j^i = \sum_k^{n_{i-1}} (w_{jk}^i a_k^{i-1}) + b_j^i$$

Wir werde später zum leichteren Verständnis in die Vektorschreibweise wechseln. Passend zur vorherigen Definition soll daher gelten:

$$a^i = \sigma(w^i \times a^{i-1} + b^i)$$

Die Notation orientiert sich an [5].

2.2.2 Aktivierungsfunktion

Aktivierungsfunktionen geben den Output der Knoten an. In den meisten Fällen handelt es sich hierbei um nicht-lineare Funktionen, wie die Sigmoidfunktion. Lineare Funktionen, auch wenn theoretisch möglich, kommen in der Praxis kaum zur Anwendung. Da hintereinander ausgeführte lineare Funktionen wieder linear sind, eignen sie sich nicht für die Lösung komplexer/nicht-linearer Aufgaben. Darüber hinaus gibt es keine eindeutige Definition einer Aktivierungsfunktion. Da wir jedoch in einem späteren Abschnitt den Algorithmus der Backpropagation vorstellen werden, müssen wir zusätzlich fordern, dass die Aktivierungsfunktion differenzierbar ist.

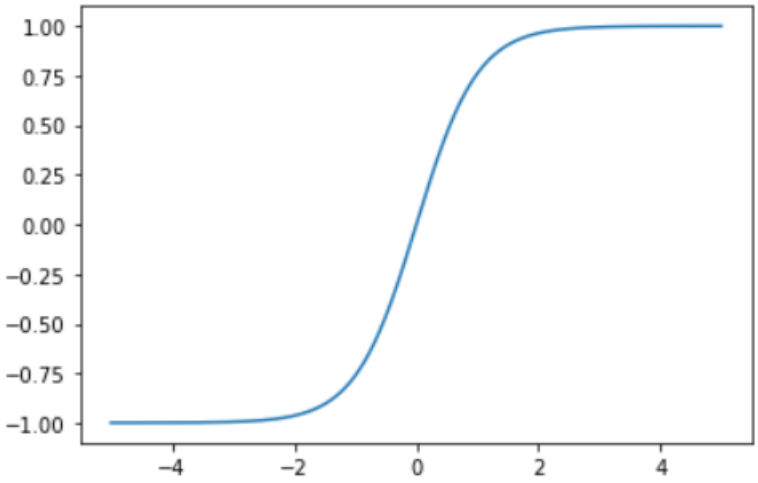

Abb. 7: Tanh(x) eine weitere nicht-lineare Aktivierungsfunktion

Auch wenn für die kommenden Abschnitte die Wahl der Aktivierungsfunktion unerheblich ist, spielt sie doch in der Praxis eine entscheidende Rolle [2].

Bemerkung: Die festgelegte Notation lässt vermuten, dass lediglich eine Aktivierungsfunktion für das gesamte Neuronale Netz zu wählen ist. Tatsächlich kann jedoch für jedes Layer eine eigene Aktivierungsfunktion gewählt werden. Dies kann zu einer höheren Genauigkeit des Modells führen und ist in der Praxis weit verbreitet. Um die Beweise jedoch übersichtlicher zu halten, wurde hierauf verzichtet.

2.2.3 Kostenfunktion

Um bewerten zu können, wie gut ein Neuronales Netz gegebene Daten vorhersagt, müssen wir eine Kostenfunktion (Lossfunction) definieren. Angenommen wir haben n viele Datenpunkte. Jeder Datenpunkt besteht aus einem Inputvektor x und einem Outputvektor y. Ein Neuronales Netz berechnet für jeden Vektor x einen Outputvektor $\hat{y}$. Die Kostenfunktion misst dann den durchschnittlichen Unterschied zwischen y und $\hat{y}$ für alle n Datenpunkte. Ein mögliches Beispiel ist die Quadratische Kostenfunktion $\frac{1}{2n}\sum_{i=0}^{n-1}\left(y^i - \hat{y}^i\right)^2$. Im Allgemeinen definieren wir eine Kostenfunktion wie folgt:

Definition (Kostenfunktion): Für ein Neuronales Netz mit Weights W und Bias Werten B und einer Menge Datenpunkten $D = \{(x_i, y_i) | 0 \leq i < n\}$, sei:

$$C(W, B, D) = \frac{1}{n}\sum_{i=0}^{n-1} C_i(W, B, (x_i, y_i))$$

die *Kostenfunktion*.

Zusätzlich sollen die einzelnen C_i zu einer gegebenen Menge D lediglich von dem Output a^L des Neuronalen Netzes abhängen. Beide Anforderungen an die Kostenfunktion sind nötig, um die Methode der Backpropagation einzuführen, welche auf partiellen Ableitungen beruht.

3 Optimierung

Haben wir ein Modell zu bestimmten Inputs und Outputs gegeben, sind wir daran interessiert die einzelnen Weights und Bias Werte so zu verändern, dass die Kostenfunktion minimiert wird. Die Optimierungsaufgabe kann also formuliert werden als: Für eine gegebene Funktion f, finde

$$f(w^*) \coloneqq \min_w f(w)$$

Für diese Aufgabe gibt es mehrere geeignete Algorithmen. Im Folgenden werden wir drei gängige Algorithmen kennenlernen: das Gradienten Verfahren, das Stochastische Gradienten Verfahren und das Mini Batch Verfahren.

3.1 Gradienten Verfahren

Das Gradienten Verfahren (GV) ist ein iteratives Verfahren mit dem Ziel ein Minimum einer Funktion zu finden. Hierbei wird in jedem Schritt ein Stück in die Richtung des Gradienten gegangen. Da wir an dem Minimum der Funktion interessiert sind, bedeutet das für den Algorithmus, dass wir in die negative Richtung des Gradienten gehen müssen. Wir erhalten folgende allgemeine Iterationsvorschrift:

1. Starte mit Punkt $w_0 \in R^p$.
2. Berechne $w_{t+1} = w_t - \eta_t \nabla f(w_t)$.
3. Beende Algorithmus, wenn Abbruchbedingung erfüllt ist.

w_0 ist hierbei die Startposition und $\eta_t > 0$ die Schrittgröße. Als Abbruchbedingung sind mehrere Bedingungen möglich. Gefordert werden kann zum Beispiel, dass der $\|\nabla f(w_t)\|$ kleiner als ε ist. In der Praxis bricht man in der Regel nach einer bestimmten Anzahl an Iterationen ab.

3.1.1 Konvergenzanalyse

Um die Konvergenz des GV zu beweisen, müssen wir spezielle Bedingungen an die Funktion f stellen. Hierzu führen wir folgende Definitionen ein.

Definition (Lipschitz stetiger Gradient): Wir sagen, eine differenzierbare Funktion $f: X \to \mathbb{R}$ hat einen *Lipschitz stetigen Gradienten*, wenn eine Konstante $L > 0$ existiert, sodass für alle $x, y \in X$ gilt:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

Lemma 1 (Lipschitz stetiger Gradient): *Sei $f: \mathbb{R}^m \to \mathbb{R}$ eine differenzierbare Funktion mit Lipschitz stetigem Gradienten mit Lipschitz-Konstante L. Dann gilt für alle $x, y \in X$ folgende Ungleichung:*

$$f(x) \leq f(y) + \langle x - y, \nabla f(y) \rangle + \frac{L}{2}\|x - y\|^2$$

Beweis: Der Beweis orientiert sich an [7].

Wir nutzen dir Taylorentwicklung um den Punkt $f(x)$ an der Stelle y bis zur 0. Ordnung:

$$f(x) = f(y) + \int_0^1 \langle \nabla f\big(y + \tau(x - y)\big), x - y \rangle d\tau$$
$$= f(y) + \langle \nabla f(y), x - y \rangle + \int_0^1 \langle \nabla f\big(y + \tau(x - y)\big) - \nabla f(y), x - y \rangle d\tau$$

Wir können mittels Cauchy-Schwarz Ungleichung weiter abschätzen:

$$\leq f(y) + \langle \nabla f(y), x - y \rangle + \int_0^1 \big\|\nabla f\big(y + \tau(x - y)\big) - \nabla f(y)\big\| \|x - y\| d\tau$$

Um das Intergral weiter abzuschätzen, benutzen wir die Voraussetzung das f einen Lipschitz stetigen Gradienten hat.

$$\big\|\nabla f\big(y + \tau(x - y)\big) - \nabla f(y)\big\| \leq L\big\|\big(y + \tau(x - y)\big) - \mathrm{y}\big\| = L\tau\|\mathrm{x} - \mathrm{y}\|$$

Daraus folgt für $f(x)$:

$$f(x) \leq f(y) + \langle \nabla f(y), x - y \rangle + L \int_0^1 \tau \|x - y\|^2 d\tau$$
$$= f(y) + \langle x - y, \nabla f(y) \rangle + \frac{L}{2}\|x - y\|^2$$

■

Fordern wir von f lediglich einen Lipschitz stetigen Gradienten, können wir für das GV bereits die Konvergenz des Gradienten zeigen.

Satz 1: *Sei $f: X \to \mathbb{R}$ eine Funktion mit Lipschitz stetigem Gradienten mit Lipschitz-Konstante L. Dann gilt für das Gradienten Verfahren mit fester Schrittweite $\eta_t = \eta = \frac{1}{L}$:*

$$\|\nabla f(w_t)\| \leq \varepsilon \text{ in höchstens } O\left(\frac{1}{\varepsilon^2}\right) \text{ Iterationen}$$

Beweis: Der Beweis orientiert sich an [4].

Der Gradient von f ist Lipschitz stetig, nach Lemma 1 wissen wir daher:

$$f(w_{t+1}) \leq f(w_t) - \eta_t \|\nabla f(w_t)\|^2 + \frac{L\eta_t^2}{2}\|\nabla f(w_t)\|^2$$
$$= f(w_t) - \frac{1}{L}\|\nabla f(w_t)\|^2 + \frac{1}{2L}\|\nabla f(w_t)\|^2$$
$$= f(w_t) - \frac{1}{2L}\|\nabla f(w_t)\|^2$$

Umgestellt bekommen wir daher folgende Ungleichung:

$$\frac{1}{2L}\|\nabla f(w_t)\|^2 \leq f(w_t) - f(w_{t+1})$$

Summieren wir die Ungleichungen bis zu einer beliebigen Iteration T auf, erkennen wir, dass die Terme auf der rechten Seite sich gegenseitig aufheben:

$$\sum_{t=0}^{T} \frac{1}{2L}\|\nabla f(w_t)\|^2 \leq \big(f(w_0) - f(w_1)\big) + \big(f(w_1) - f(w_2)\big) + \cdots + \big(f(w_T) - f(w_{T+1})\big)$$
$$= f(w_0) - f(w_{T+1}) \leq f(w_0) - f(w^*)$$

Wobei w^*eine optimale Lösung ist. Die linke Seite der Ungleichung ist eine Summe nicht-negativer Summanden, die durch $f(w_0) - f(w^*)$ nach oben beschränkt ist. $\|\nabla f(w_t)\|$ konvergiert also gegen null.

Wir wollen die linke Seite weiterabschätzen und betrachten dafür:

$$\|\nabla f(w')\| := \min_{i \leq T} \|\nabla f(w_i)\|$$

Damit können wir schreiben:

$$\sum_{t=0}^{T} \frac{1}{2L}\|\nabla f(w_t)\|^2 \geq \frac{T+1}{2L}\|\nabla f(w')\|^2$$

Nach Umstellen erhalten wir:

$$\|\nabla f(w')\| \leq \sqrt{\frac{2L\big(f(w_0) - f(w^*)\big)}{T+1}}$$

Wir setzen nun die rechte Seite der Ungleichung gleich $\varepsilon > 0$ und stellen nach T um:

$$\sqrt{\frac{2L\big(f(w_0) - f(w^*)\big)}{T+1}} = \varepsilon$$
$$\frac{\sqrt{2L\big(f(w_0) - f(w^*)\big)}}{\sqrt{T+1}} = \varepsilon$$
$$\frac{\sqrt{2L\big(f(w_0) - f(w^*)\big)}}{\varepsilon} = \sqrt{T+1}$$
$$T = 2L\big(f(w_0) - f(w^*)\big)\frac{1}{\varepsilon^2} - 1$$

ε können wir beliebig klein wählen, da wir bereits gezeigt haben, dass $\sum_{t=0}^{T} \frac{1}{2L} \|\nabla f(w_t)\|^2$ eine Nullfolge ist.

Da $2L\big(f(w_0) - f(w^*)\big)$ nicht von T abhängt folgt $T \in O\left(\frac{1}{\varepsilon^2}\right)$.

■

Dies bedeutet, dass das Gradienten Verfahren schnell zu einer ungenauen Approximation konvergiert, danach allerdings wegen $T \in O\left(\frac{1}{\varepsilon^2}\right)$ kaum noch genauer wird. Eine Konvergenz der Funktionswerte können wir zeigen, indem wir zusätzlich zum Lipschitz stetigen Gradienten fordern, dass f stark konvex seien soll. Dazu führen wir folgende Definitionen ein.

Definition (Konvexe Funktion): Eine Funktion $f: X \to \mathbb{R}$ nennen wir *konvex*, wenn für alle $x, y \in X$ und jedes $0 < \lambda < 1$ mit $\lambda x + (1-\lambda)y \in X$ gilt:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$

Definition (Stark konvexe Funktion): Eine differenzierbare konvexe Funktion $f: X \to \mathbb{R}$ nennen wir *σ-stark konvex* genau dann, wenn eine Konstante $\sigma > 0$ existiert, sodass für alle $x, y \in X$ gilt:

$$f(x) \geq f(y) + \langle \nabla f(y), x - y \rangle + \frac{\sigma}{2} \|x - y\|^2$$

Bemerkung: Für eine Funktion, die stark konvex ist und einen Lipschitz stetigen Gradienten besitzt, gilt $\sigma \leq L$. Diese Ungleichung folgt direkt aus dem Einsetzten der Definitonen:

$$f(y) + \langle x - y, \nabla f(y) \rangle + \frac{\sigma}{2} \|x - y\|^2 \leq f(x) \leq f(y) + \langle \nabla f(y), x - y \rangle + \frac{L}{2} \|x - y\|^2$$

Lemma 2 (Polyak-Lojasiewicz Ungleichung): *Sei $f: \mathbb{R}^m \to \mathbb{R}$ differenzierbar und σ-stark konvex. Außerdem sei x^* ein Vektor für den $f(x^*)$ minimal ist. Dann gilt für alle $y \in \mathbb{R}^m$:*

$$\|\nabla f(y)\|^2 \geq 2\sigma\big(f(y) - f(x^*)\big)$$

Beweis: Da f σ-stark konvex ist gilt:

$$f(x) \geq f(y) + \langle \nabla f(y), x - y \rangle + \frac{\sigma}{2} \|x - y\|^2$$

Wir minimieren nun beide Seiten der Ungleichung (was die Ungleichung erhält). Für die linke Seite gilt dann:

$$\min_x (f(x)) = f(x^*)$$

Die rechte Seite leiten wir nach y ab und setzen die Ableitung gleich 0:

$$\nabla f(y) + \sigma(x - y) = 0$$
$$=> x_1 = y - \frac{1}{\sigma}\nabla f(y)$$

Für die Ungleichung erhalten wir damit:

$$f(x^*) \geq f(y) + \langle \nabla f(y), -\frac{1}{\sigma}\nabla f(y)\rangle + \frac{\sigma}{2}\left\|-\frac{1}{\sigma}\nabla f(y)\right\|^2$$
$$= f(y) - \frac{1}{\sigma}\langle \nabla f(y), \nabla f(y)\rangle + \frac{1}{2\sigma}\|\nabla f(y)\|^2$$
$$= f(y) - \frac{1}{\sigma}\|\nabla f(y)\|^2 + \frac{1}{2\sigma}\|\nabla f(y)\|^2 = f(y) - \frac{1}{2\sigma}\|\nabla f(y)\|^2$$

Umgestellt folgt daraus:

$$\|\nabla f(y)\|^2 \geq 2\sigma\big(f(y) - f(x^*)\big)$$

■

Satz 2: *Sei $f: X \to \mathbb{R}$ eine Funktion mit Lipschitz stetigem Gradienten mit Lipschitz-Konstante L und σ-stark konvex. Dann gilt für das Gradienten Verfahren mit Schrittweite $\eta_t = \eta = \frac{1}{L}$:*

$$f(w_t) - f(w^*) \leq \left(1 - \frac{\sigma}{L}\right)^t (f(w_0) - f(w^*))$$

Beweis: Der Beweis orientiert sich an [3].

Da f einen Lipschitz stetigen Gradienten besitzt, gilt:

$$f(w_{t+1}) \leq f(w_t) + \langle w_{t+1} - w_t, \nabla f(w_t)\rangle + \frac{L}{2}\|w_{t+1} - w_t\|^2$$

Wenden wir darauf die Iterationsvorschrift des GV mit $\eta = \frac{1}{L}$ an, erhalten wir:

$$f(w_{t+1}) \leq f(w_t) - \frac{1}{L}\langle \nabla f(w_t), \nabla f(w_t)\rangle + \frac{1}{2L}\|\nabla f(w_t)\|^2$$

$$f(w_{t+1}) - f(w_t) \leq -\frac{1}{2L}\|\nabla f(w_t)\|^2$$

Mit der Polyak-Lojasiewicz Ungleichung erhalten wir dann:

$$f(w_{t+1}) - f(w_t) \leq -\frac{\sigma}{L}(f(w_t) - f(w^*))$$

Indem wir $f(w_t) - f(w^*)$ zu beiden Seiten addieren erhalten wir:

$$f(w_{t+1}) - f(w^*) \leq \left(1 - \frac{\sigma}{L}\right)(f(w_t) - f(w^*))$$

Die Ungleichung rekursiv angewendet ergibt die Aussage des Satzes. Da $\sigma \leq L$ konvergiert das Gradienten Verfahren.

■

3.1.2 Iterationskosten

Die Schrittkosten für einen einzelnen Iterationsschritt entstehen hauptsächlich, durch die Berechnung des Gradienten. Wir erinnern uns, dass die Kostenfunktion definiert war als: $C(W, B, D) = \frac{1}{n}\sum_{i=0}^{n-1} C_i(W, B, (x_i, y_i))$. Der für die Iteration zu berechnende Gradient ∇C ist also der Durchschnitt aller individueller Gradienten ∇C_i und hängt daher von der Anzahl der Datenpunkte n ab. Für die Schrittkosten des GV gilt daher: Sei $w \in \mathbb{R}^d$ und $f(w) = \frac{1}{n}\sum_{i=0}^{n-1} f_i(w)$, dann liegen die Kosten für einen Iterationsschritt in $\mathcal{O}(nd)$.

3.2 Stochastisches Gradienten Verfahren

Eine Anforderung an die Kostenfunktion ist, dass sie sich als Mittelwert der Kostenfunktionen aller einzelnen Datenpunkte darstellen lässt. Diese Voraussetzung überträgt sich auch auf den Gradienten, der sich als Mittelwert aller individueller Gradienten darstellen lässt. Die Idee beim Stochastischen Gradienten Verfahren (SGV) baut genau hierauf auf. Statt in jedem Iterationsschritt alle individuellen Gradienten zu berechnen, wird hier in jedem Schritt nur ein einzelner Gradient berechnet.

Für einen beliebigen Datenpunkt (x_i, y_i) sei $\nabla f_i(w)$ der individuelle Gradient. Gleichzeitig soll gelten, dass $\mathbb{E}[\nabla f_i(w)\,] = \nabla f(w)$.

Für das SGV stellen wir folgenden Algorithmus auf:

1. Starte mit Punkt $w_0 \in R^p$.
2. Wähle einen zufälligen Datenpunkt (x_i, y_i) und berechne $\nabla f_{i_t}(w_t)$.
3. Berechne $w_{t+1} = w_t - \eta_t \nabla f_{i_t}(w_t)$.
4. Beende Algorithmus, wenn Abbruchbedingung erfüllt ist.

3.2.1 Konvergenzanalyse

Satz 3: *Sei $f: X \to \mathbb{R}$ eine Funktion mit Lipschitz stetigem Gradienten mit modulus L und σ-stark konvex. Sei außerdem $\mathbb{E}\left[\left\|\nabla_{i_t} f(w_t)\right\|^2\right] \leq C^2$ für alle w_t mit beliebigem C. Dann gilt für das Stochastische Gradienten Verfahren mit Schrittweite $\eta_t = \eta < \frac{1}{2\sigma}$:*

$$\mathbb{E}[f(w_t) - f(w^*)] \leq (1 - 2\sigma\,\eta)^t [\, f(w_0) - f(w^*)] + \frac{LC^2\eta}{4\sigma}$$

Beweis: Der Beweis orientiert sich an [3].

Da f einen Lipschitz stetigen Gradienten besitzt, gilt:

$$f(w_{t+1}) \leq f(w_t) + \langle w_{t+1} - w_t, \nabla f(w_t)\rangle + \frac{L}{2}\|w_{t+1} - w_t\|^2$$

Wenden wir darauf die Iterationsvorschrift des SGV an, erhalten wir:

$$f(w_{t+1}) \leq f(w_t) - \eta\langle \nabla f_{i_t}(w_t), \nabla f(w_t)\rangle + \frac{L\eta^2}{2}\left\|\nabla f_{i_t}(w_t)\right\|^2$$

Wir betrachten von beiden Seiten den Erwartungswert in Bezug auf i_k.

$$\mathbb{E}[f(w_{t+1})] \leq f(w_t) - \eta\langle \mathbb{E}\left[\nabla f_{i_t}(w_t)\right], \nabla f(w_t)\rangle + \frac{L\eta^2}{2}\mathbb{E}\left[\left\|\nabla f_{i_t}(w_t)\right\|^2\right]$$

Da $\mathbb{E}\left[\nabla f_{i_t}(w_t)\right] = \nabla f(w_t)$ und $\mathbb{E}\left[\left\|\nabla_{i_t} f(w_t)\right\|^2\right] \leq C^2$, können wir weiter abschätzen:

$$\mathbb{E}[f(w_{t+1})] \leq f(w_t) - \eta\|\nabla f(w_t)\|^2 + \frac{L\eta^2}{2}C^2$$

Mit der Polyak-Lojasiewicz Ungleichung erhalten wir dann:

$$\mathbb{E}[f(w_{t+1})] \leq f(w_t) - \eta 2\sigma(f(w_t) - f(w^*)) + \frac{L\eta^2C^2}{2}$$

Wir ziehen von beiden Seiten $f(x^*)$ ab:

$$\mathbb{E}[f(w_{t+1}) - f(w^*)] \leq (1 - \eta 2\sigma)(f(w_t) - f(w^*)) + \frac{L\eta^2C^2}{2}$$

Diese Ungleichung rekursiv angewendet ergibt:

$$\mathbb{E}[f(w_{t+1}) - f(w^*)] \leq (1 - \eta 2\sigma)^{t+1}\left(f(w_0) - f(w^*)\right) + \frac{L\eta^2C^2}{2}\sum_{j=0}^{t}(1 - \eta 2\sigma)^j$$

$$\leq (1 - \eta 2\sigma)^{t+1}\left(f(w_0) - f(w^*)\right) + \frac{L\eta^2C^2}{2}\sum_{j=0}^{\infty}(1 - \eta 2\sigma)^j$$

$$= (1 - \eta 2\sigma)^{t+1}\left(f(w_0) - f(w^*)\right) + \frac{L\eta^2C^2}{2}\frac{1}{\eta 2\sigma}$$

$$= (1 - \eta 2\sigma)^{t+1}\left(f(w_0) - f(w^*)\right) + \frac{L\eta C^2}{4\sigma}$$

Da $\eta < \frac{1}{2\sigma}$ können wir im letzten Schritt den Limes der geometrischen Reihe benutzen.

■

3.2.2 Iterationskosten

Wie beim GV entstehen die Schrittkosten hauptsächlich durch die Berechnung des Gradienten. Da beim SGV jedoch nur ein einzelner individueller Gradient berechnet wird, liegen die Kosten für einen Iterationsschritt nur bei in $\mathcal{O}(d)$.

3.3 Mini Batch Verfahren

Das Mini Batch Verfahren (MBV) kann als Verallgemeinerung des Stochastischen Gradienten Verfahrens gesehen werden. Während beim SGV in jeder Iteration ein zufälliger Datenpunkt für die Berechnung des individuellen Gradienten verwendet wird, benutzt das MBV für die Berechnung eine beliebig große Teilmenge an zufälligen Datenpunkten (Batch). Sei $n > 0$ die Anzahl aller Datenpunkte, dann definieren wir den Teilgradienten für eine beliebig große Teilmenge an Datenpunkten $\mathcal{B} \subset \{(x_i, y_i) | 0 < i \leq n\}$ als:

$$\nabla f_{\mathcal{B}}(w) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(w) \quad , wobei\ 0 < |\mathcal{B}| \leq n$$

Wie auch bei dem Stochastischen Gradienten Verfahren muss $\nabla f_{\mathcal{B}}(w)$ ein erwartungstreuer Schätzer von $\nabla f(w)$ sein, also $\mathbb{E}[\nabla f_{\mathcal{B}}(w)] = \nabla f(w)$. In dem Fall $|\mathcal{B}| = n$ gilt offensichtlich $\nabla f_{\mathcal{B}}(w) = \nabla f(w)$.

Für das MBV stellen wir folgenden Algorithmus auf:

1. Starte mit Punkt $w_0 \in R^p$.
2. Wähle Teilmenge $\mathcal{B}_t$ und berechne $\nabla f_{\mathcal{B}_t}(w)$.
3. Berechne $w_{t+1} = w_t - \eta_t \nabla f_{\mathcal{B}_t}(x) = w_t - \frac{\eta_t}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \nabla f_i(w_t)$.
4. Beende Algorithmus, wenn Abbruchbedingung erfüllt ist.

Je nach Wahl der Batch Größen liegt das Mini Batch Verfahren bezüglich Konvergenzrate und Iterationskosten näher am GV oder näher am SGV.

3.4 Schrittweite

Die Schrittweite, auch Learning Rate genannt, spielt eine entscheidende Rolle für die Konvergenz und insbesondere für die Konvergenzgeschwindigkeit der Verfahren. In den Beweisen konnten wir unter speziellen Bedingungen maximale Werte festlegen, die die Konvergenz des Verfahrens garantieren. In der Praxis ist jedoch meist wenig über die Struktur der Daten (konvex, Lipschitz stetig) bekannt. Häufig wird die Learning Rate daher mittels systematischen Testens ermittelt.

3.5 Backpropagation

Alle drei Verfahren basieren auf der Berechnung individueller Gradienten. Eine Möglichkeit bei Neuronalen Netzen diese Gradienten zu berechnen, ist die Methode der Backpropagation. Im Folgenden wird die Funktionsweise bewiesen.

Definition: Wir definieren den Fehler des j-ten Neuronen in Layer l als:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Und δ^l als den Fehler des gesamten Layers.

Satz 4: *Es gelten folgende vier Aussagen:*

B1: $\delta^L = \nabla_a C \odot \sigma'(z^L)$

B2: $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$ für $2 \leq l \leq L-1$

B3: $\frac{\partial C}{\partial b_j^l} = \delta_j^l$ für $2 \leq l \leq L$

B4: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ für $2 \leq l \leq L$

Beweis: Der Beweis orientiert sich an [5].

B1: Nach Definition von δ_j^L gilt:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

Mit der Kettenregel können wir die rechte Seite umschreiben und erhalten:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}$$

a_k^L hängt jedoch nur von z_j^L, wenn $k = j$. Die Summe kann daher vereinfacht werden:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

Wir haben a_j^L definiert als $a_j^L = \sigma(z_j^L)$. Das bedeutet:

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$$

und wir erhalten:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

Was B1 in Komponentenschreibweise ist.

B2: Wir starten wieder mit der Definition von δ_j^l:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Indem wir erneut die Kettenregel benutzen, können wir die rechte Seite umschreiben:

$$\delta_j^l = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}$$
$$= \sum_{k=1}^{n_{l+1}} \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Im letzten Schritt sehen wir uns $\frac{\partial z_k^{l+1}}{\partial z_j^l}$ an. Dazu betrachten wir die Definition von z_k^{l+1}:

$$z_k^{l+1} = \sum_j (w_{kj}^{l+1}\, a_j^l) + b_k^{l+1} = \sum_j (w_{kj}^{l+1}\, \sigma(z_j^l)) + b_k^{l+1}$$

Für die Ableitung gilt daher:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l)$$

Damit gilt für δ_j^l:

$$\delta_j^l = \sum_{k=1}^{n_{l+1}} w_{kj}^{l+1}\delta_k^{l+1}\,\sigma'(z_j^l)$$

Was B2 in Komponentenschreibweise ist.

B3: Um B3 zu beweisen, beginnen wir mit der Definition von z_k^l:

$$z_k^l = \sum_j (w_{kj}^l\, \sigma(z_j^{l-1})) + b_k^l$$

Das abgeleitet nach b_k^l ist offensichtlich 1. Betrachten wir nun $\frac{\partial C}{\partial b_j^l}$ und benutzen erneut die Kettenregel:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l$$

B4: Nach der Definition von z_j^l gilt:

$$z_j^l = \sum_k (w_{jk}^l \, a_k^{l-1}) + b_j^l$$

Und für die Ableitung:

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}$$

Wobei die Ableitung nun nicht mehr *j* abhängt. Gleichzeitig ist auch offensichtlich, dass:

$\frac{\partial z_s^l}{\partial w_{jk}^l} = 0$ für $s \neq j$

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^l}\frac{\partial z_s^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a_k^{l-1} = \delta_j^l a_k^{l-1}$$

■

3.6 Optimierung in der Praxis

Im Zuge des Trainingsprozesses optimieren wir die Parameter des Neuronalen Netzes, sodass es möglichst gute Vorhersagen für die Trainingsdaten trifft. In der Praxis ist dies jedoch nicht die tatsächliche Aufgabe des Modells. Vielmehr sind wir daran interessiert, möglichst gute Vorhersagen für noch unbekannte Daten zu treffen.

Das Neuronale Netz soll also gleichzeitig möglichst genau die Trainingsdaten vorhersagen, jedoch ohne dabei zu spezifisch zu werden. Man spricht von dem Bias-Variance Tradeoff.

Um diesen Tradeoff zu überwachen, werden üblicherweise im Vorfeld alle bekannten Daten in Trainingsdaten und Testdaten unterteilt. Mit den Trainingsdaten optimieren wir die Parameter des Modells. Die Testdaten nutzen wir, um zu verifizieren, wie gut das Modell unbekannte Daten vorhersagt.

Ein Neuronales Netz welches sehr genau die Trainingsdaten vorhersagt, jedoch die Testdaten signifikant schlechter vorhersagt, nennen wir overfitted.

Wie gut ein Modell die Trainingsdaten vorhersagt, hängt vor allem von der Größe des Modells, der Anzahl der Iterationen zur Optimierung und der Anzahl an Trainingsdaten ab. Zu viele Knoten und Layer erlauben dem Neuronalen Netz, die Trainingsdaten zu genau vorherzusagen und zu overfitten. Es gibt jedoch Methoden dem entgegenzuwirken.

1. Early Stopping

Beim Early Stopping wird der Trainingsprozess unterbrochen, wenn die Genauigkeit der Vorhersagen der Testdaten deutlich schlechter wird. Im Zuge des Trainingsprozesses sind kleine Schwankungen in der Genauigkeit üblich. Beim Early Stopping muss daher darauf geachtet werden, dass das Training nicht zu früh unterbrochen wird. Eine Möglichkeit ist es im Vorfeld eine Anzahl an Iterationen festzulegen. In diesem Fall wird das Training erst unterbrochen, wenn auch nach dem Abwarten dieser Anzahl an Iterationen keine Verbesserung der Genauigkeit der Trainingsdaten vorlag.

2. Dropout

Dropout beschreibt ein Vorgehen während des Trainingsprozesses, bei dem während jeder Iteration einige Knoten aus dem Modell nicht betrachtet werden. Dies betrifft auch alle dazugehörigen Gewichte. Da so in jedem Schritt eine kleinere Version des Neuronalen Netzes betrachtet wird, tendieren die vorhanden Knoten dazu, allgemeiner vorherzusagen. [6]

4 Implementierung in Python

Wir werden ein Neuronales Netz mit echten Daten erstellen. Als Programmiersprache werden wir dazu Python verwenden. Python ist die im Bereich Data Science am häufigsten genutze Sprache. Bibliotheken wie Tensorflow und Pytorch bieten die Möglichkeit, mit wenig Aufwand Neuronale Netze zu kreieren. Zusätzlich werden auch mehrere Datasets zur Verfügung gestellt. In diesem Beispiel arbeiten wir mit dem MNIST Dataset von Tensorflow.

4.1 MNIST Dataset

Das MNIST Dataset besteht aus Bildern handgeschriebener Ziffern (null bis neun) in einem 28x28 Pixel Format. Jedes Bild hat eine Zahl zugeordnet, die die handgeschriebenen Ziffern beschreibt. Aufgeteilt werden die Daten in ein 60.000 Einträge großes Trainingsset und ein 10.000 Einträge großes Testset.

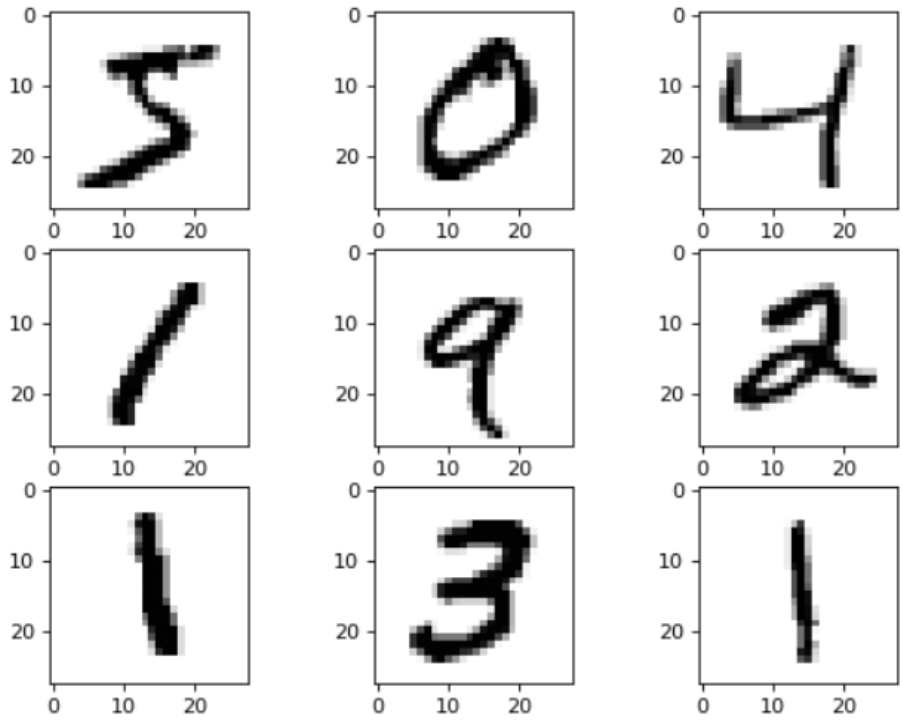

Abb. 8: Auszug aus dem MNIST Dataset

4.2 Aufbau des Modells

Mit der in Tensorflow enthaltenen API Keras werden wir ein Neuronales Netz erstellen. Als Input bekommt es die handgeschriebenen Ziffern. Dabei werden die einzelnen Pixel des Bildes betrachtet. Jeder Pixel hat einen Wert zwischen 0 (Weiß) und 255 (Schwarz). Jedes Bild kann also als 28x28 Matrix verstanden werden. Um von einem Neuronale Netz als Input verstanden

werden zu können, wird diese Matrix in einen eindimensionalen Vektor mit derselben Anzahl an Einträgen umgeschrieben. Output des Neuronalen Netzes soll ein 10 Einträge großer Vektor sein. Genau ein Eintrag des Vektors ist Eins, während der Rest aus Nullen besteht. Die Position der Eins repräsentiert dann die vom Modell erkannte Zahl. Im Hidden Layer erstellen wir zwei Layer mit je 100 Knoten. Bei der Wahl der Aktivierungsfunktion beschränken wir uns diesmal nicht auf eine einzige Funktion. Das Wählen mehrerer Aktivierungsfunktion in den verschiedenen Layern ist gängige Praxis. In diesem Fall benutzen wir in den beiden Hidden Layern die Sigmoidfunktion und im Outputlayer die Softmax-Funktion. Die Softmax-Funktion sorgt dafür, dass unser Outputvektor genau eine Eins und neun Nullen enthält:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad j = 1, \dots, K$$

4.3 Training des Modells

Für das Trainieren des Modells benutzen wir nur das Trainingsset. Um den Fehler des Modells zu messen, definieren wir folgende Kostenfunktion:

$$C = \frac{1}{n}\sum_{i=0}^{n-1} y_i \log(\hat{y}_i) = \frac{1}{60000}\sum_{i=0}^{59999} y_i \log(\hat{y}_i)$$

Die Funktion ist bekannt als Kreuzentropie mehrerer Klassen. In unserem Fall existieren zehn Klassen, für jede Ziffer eine.

Zum Optimieren der Parameter betrachten wir drei verschiedene Algorithmen: Gradienten Verfahren, Stochastisches Gradienten Verfahren und Mini Batch Verfahren mit Batch gleich 1.000. Dabei beginnen alle drei mit den gleichen Startparametern. Alle drei Verfahren sollen laufen, bis sie die Kostenfunktion auf einen Wert von 0,2 minimiert wurde. Die Learning Rate wurde durch systematisches Testen ermittelt. Beim Gradienten Verfahren verwenden wir eine Learning Rate von $\eta = 0{,}0002$, bei dem Stochastischen Gradienten Verfahren $\eta = 0{,}6$ und bei dem Mini Batch Verfahren benutzen wir $\eta = 0{,}01$.

4.4 Auswertung

Auswertung Konvergenz:

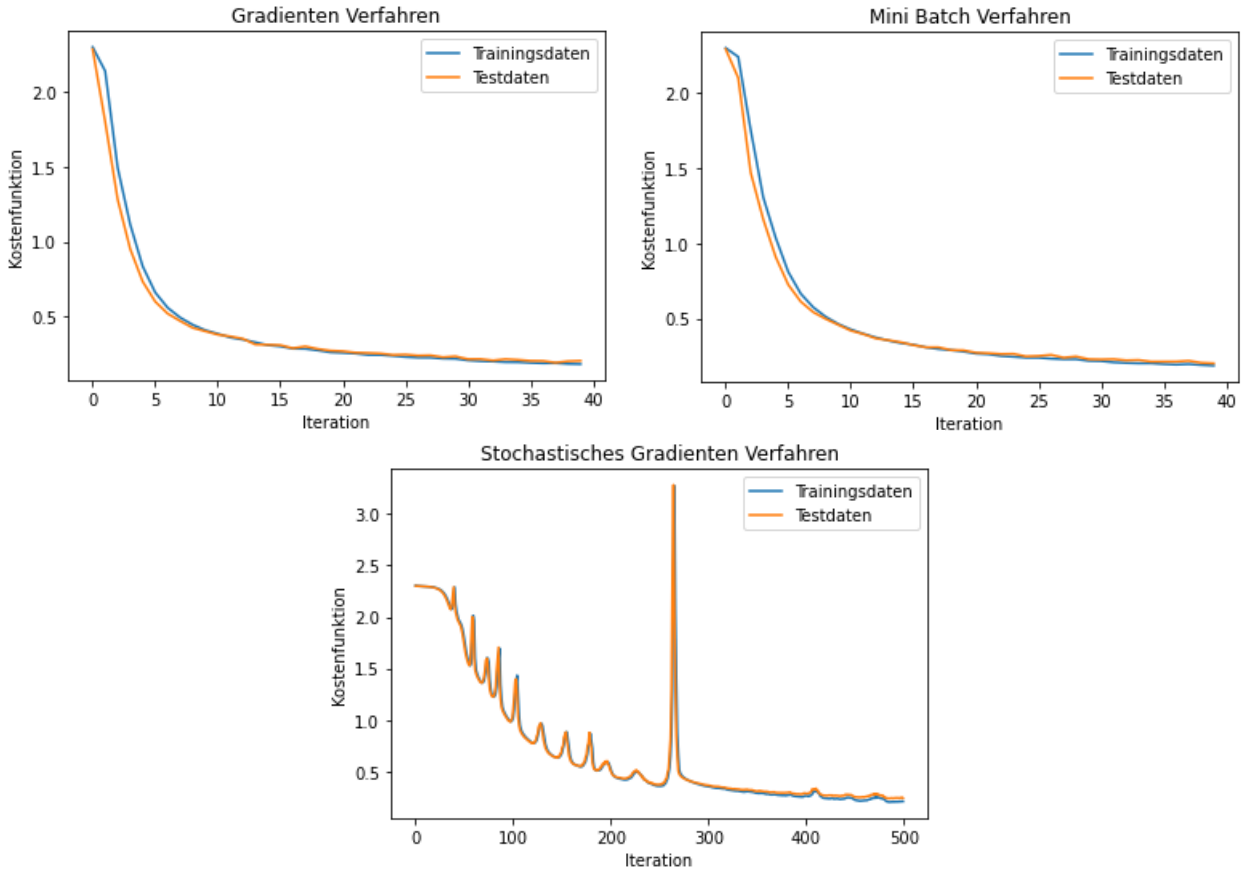

Abb. 9: Kostenfunktion während einzelner Iterationsschritte

Das Gradienten Verfahren und das Mini Batch Verfahren konvergieren mit nahezu identischer Geschwindigkeit. Während der ersten 5 Iterationsschritten sinkt die Kostenfunktion stark. Danach flacht die Kurve ab, sinkt jedoch weiter. Nach 40 Iterationen hat die Kostenfunktion bei beiden Verfahren den Zielwert von 0,2 unterschritten.

Beim Stochastischen Gradienten Verfahren sind immer wieder Ausreißer der Kostenfunktion zu erkennen. Dennoch ist eine klare Konvergenz erkennbar. Verglichen mit dem Gradienten Verfahren und dem Mini Batch Verfahren konvergiert das Stochastische Gradienten Verfahren weniger stark. Die Kurve sinkt nicht so stark ab und hat erst nach 484 Iterationen den Wert 0,2 erreicht.

Bei allen drei Verfahren liegen die Werte der Kostenfunktion für die Trainingsdaten und für die Testdaten stets nah beieinander. Dies spricht dafür, dass die Größe des Neuronalen Netzes passend gewählt war. Das Modell ist nicht overfitted.

Auswertung Iterationskosten:

	GV	MBV	SGV
∅ Zeit pro Iteration	33 s	0,87 s	0,18 s
Gesamtzeit	1320 s	34,8 s	90 s

Abb. 10: Vergleich der Iterationszeiten

Für einen Iterationsschritt benötigt das Gradienten Verfahren durchschnittlich 33 Sekunden, das Mini Batch Verfahren 0,87 Sekunden und das Stochastische Gradienten Verfahren 0,18 Sekunden. In der praktischen Umsetzung sieht man, dass die Rechenzeit nicht mit der Anzahl der zu berechnenden individuellen Gradienten skaliert. Ein Grund dafür ist, dass manche Prozesse vom Computer parallel ausgeführt wurden. Dies würde vor allem die Zeit vom GV reduzieren, da hier am meisten Gradienten parallel berechnet werden könnten.

5 Fazit

Das MBV konvergiert am schnellsten und benötigt dafür am wenigsten Zeit. Die Learning Rate scheint gut gewählt zu seien. Kleinere Batchgrößen können möglichweise die Gesamtzeit noch weiter reduzieren. Eine schnellere Konvergenz durch größere Batchgrößen zu erreichen ist nicht wahrscheinlich, da bereits dieselbe Konvergenzgeschwindigkeit wie die des GV erreicht wurde.

Das GV konvergierte zwar mit dem MBV am schnellsten, benötigte aufgrund der vielen Datenpunkte jedoch deutlich länger als die anderen beiden Verfahren. Möglicherweise kann die Wahl der Learning Rate noch optimiert werden, sodass eine noch schnellere Konvergenz möglich ist. Aufgrund der deutlich längeren Iterationszeit ist das GV als Optimierungsverfahren für diese Aufgabe ungeeignet.

Das SGV konvergiert langsam, benötigt aber für alle Iterationen weniger Zeit als das GV. Die großen Ausreißer könnten in der Praxis jedoch problematisch sein.

Quellen

[1] Mitchell T. (1997). *Machine Learning*. New York, NY: McGraw Hill

[2] Olgac, A & Karlik, Bekir. (2011). *Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks*. International Journal of Artificial Intelligence And Expert Systems. 1. 111-122.

[3] Karimi H., Nutini J., Schmidt M. (2016). *Linear Convergence of Gradient and Proximal-Gradient Methods Under the Polyak-Łojasiewicz Condition*. In: Frasconi P., Landwehr N., Manco G., Vreeken J. (eds) Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2016. Lecture Notes in Computer Science, vol 9851. Springer, Cham.

[4] Alexander Johannes Smola, S.V.N. Vishwanathan. (2008). *Introduction to Machine Learning*. Cambridge University Press.

[5] Michael A. Nielsen. (2015). *Neural Networks and Deep Learning*, Determination Press.

[6] Srivastava, Nitish & Hinton, Geoffrey & Krizhevsky, Alex & Sutskever, Ilya & Salakhutdinov, Ruslan. (2014*). Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research. 15. 1929-1958.

[7] Gower, R. (2019). *Convergence Theorems for Gradient Descent*.

[8] McCulloch, W.S., Pitts, W. (1943). *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics 5, 115–133.

[9] Rosenblatt, F. (1958). *The perceptron: a probabilistic model for information storage and organization in the brain*. Psychological review, 65 6, 386-408.

Anhang

Python Code

```
# MNist Daten

# Einlesen der Daten

import tensorflow as tf
import matplotlib.pyplot as plt

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Graphische Darstellung der ersten 9 Einträge

fig = plt.figure(figsize=(8, 6), dpi=80)
for i in range(9):
    plt.subplot(331 + i)
    plt.imshow(x_train[i], cmap = plt.cm.binary)
plt.show()

# Erstellen des Neuronalen Netzes

def Erstellen_Modell():
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(100, activation="sigmoid", kernel_initi
alizer=tf.keras.initializers.RandomNormal(mean=0, stddev=0.01, seed=1),
        bias_initializer=tf.keras.initializers.Zeros()))
    model.add(tf.keras.layers.Dense(100, activation="sigmoid", kernel_initi
alizer=tf.keras.initializers.RandomNormal(mean=0, stddev=0.01, seed=2),
        bias_initializer=tf.keras.initializers.Zeros()))
    model.add(tf.keras.layers.Dense(10, activation="softmax", kernel_initia
lizer=tf.keras.initializers.RandomNormal(mean=0, stddev=0.01, seed=3),
        bias_initializer=tf.keras.initializers.Zeros()))
    return model

# GV

# Optimierung

model = Erstellen_Modell()

model.compile(optimizer = tf.compat.v1.train.GradientDescentOptimizer(learn
ing_rate=0.0002),
             loss = "sparse_categorical_crossentropy",
             metrics = ["accuracy"])

history_gv = model.fit(x_train, y_train, validation_data=(x_test, y_test),
epochs=40, batch_size=1, verbose=1)

# Graphische Auswertung

plt.plot(history_gv.history['loss'])
```

```
plt.plot(history_gv.history['val_loss'])
plt.title('Gradienten Verfahren')
plt.ylabel('Kostenfunktion')
plt.xlabel('Iteration')
plt.legend(['Trainingsdaten', 'Testdaten'], loc='upper right')
plt.show()

# MBV

# Optimierung

model = Erstellen_Modell()

model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate=0.01),
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_mbv = model.fit(x_train, y_train, validation_data=(x_test, y_test),
epochs=40, batch_size=60, verbose=1)

# Graphische Auswertung

plt.plot(history_mbv.history['loss'])
plt.plot(history_mbv.history['val_loss'])
plt.title('Mini Batch Verfahren')
plt.ylabel('Kostenfunktion')
plt.xlabel('Iteration')
plt.legend(['Trainingsdaten', 'Testdaten'], loc='upper right')
plt.show()

# SGV

# Optimierung

model = Erstellen_Modell()

model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate=0.6),
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_sgv = model.fit(x_train, y_train, validation_data=(x_test, y_test),
epochs=500, batch_size=60000, verbose=1)

#Graphische Auswertung

plt.plot(history_sgv.history['loss'])
plt.plot(history_sgv.history['val_loss'])
plt.title('Stochastisches Gradienten Verfahren')
plt.ylabel('Kostenfunktion')
plt.xlabel('Iteration')
plt.legend(['Trainingsdaten', 'Testdaten'], loc='upper right')
plt.show()
```

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die an gegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.