Lecture 3: Gradientenabstiegsverfahren

BA-INF 153: Einführung in Deep Learning für Visual Computing

Prof. Dr. Reinhard Klein

Nils Wandel

Informatik II, Universität Bonn

24.04.2024

Recap

3 Teile

- Machine Learning Algorithmen
- Recap Wahrscheinlichkeitstheorie
- Statistische Schätzer

Machine Learning Algorithmen

- Lernalgorithmen sind Computerprogramme, welche für eine Aufgabe T und Performanz-Mass P ihre Performanz verbessern, nachdem sie Erfahrung E gewonnen haben.
- Erfahrung wird durch Trainingsdaten gewonnen, während die Performanz auf ungesehenen Testdaten evaluiert wird. Die Fähigkeit zu generalisieren unterscheidet Lernalgorithmen von Optimierungsalgorithmen.
- Fehler können von under- oder overfitting kommen, abhängig von der Modellkapazität
- Hyperparameter sind Einstellungen, die das allgemeine Verhalten des Algorithmus kontrollieren und im vorraus oder durch die Validation Performance gesetzt werden

Recap

Recap Wahrscheinlichkeitstheorie

- Zufallsvariablen bezeichnen unsichere Werte; diese folgen einer Wahrscheinlichkeitsverteilungen anstatt einzelner deterministischer Werte
- Multivariate Wahrscheinlichkeiten sind Verteilungen über mehrere Zufallsvariablen, wobei die einzelnen Verteilungen durch Marginalisierung erhalten werden können
- Bedingte Wahrscheinlichkeiten sind Verteilungen relativ zu einem deterministischen Wert
- Der Satz von Bayes erlaubt uns, bedingte Wahrscheinlichkeiten zu invertieren
- (Kreuz-)Entropie, KL-Divergenz

Statistische Schätzer

- Der Mean Squared Error eines Schätzers setzt sich aus Bias und Varianz zusammen; Die Modellkapazität sollte einen Trade-off zwischen den beiden Fehlerquellen bilden
- Zwei wichtige Schätzer sind Maximum Likelihood (ML) und Maximum a Posteriori (MAP)

Today's Lecture:

Themen Heute

1. Fehler Funktionen
2. Gradient Descent
3. Stochastic Gradient Descent
4. Adaptive Learning Rates

Hinweis - Gastvortrag von Paul Debevec

Dr. Paul Debevec, Chief Research Officer in den Netflix Eyeline Studios und außerordentlicher Forschungsprofessor an der University of Southern California wird diesen *Freitag, 26.4.2024, von 12-14 Uhr in HS2 im HSZ* einen Gastvortrag zu *"From Virtual Cinematography to Virtual Production"* halten.
Innovative Technik von Dr. Paul Debevec kam u.a. bei "The Matrix", "Spider-Man" oder "Avatar" zum Einsatz.
Weitere Infos: `https://www.informatik.uni-bonn.de/de/aktuelles/kolloquium/gastvortrag-debevec-2024`

Part 1

Fehler-Funktionen

Chapter 5.2-4 of **bishop2006pattern** (Link zum herunterladen)

Fehler-Funktionen

Maximum Likelihood-Schätzer

Der Maximum-Likelihood Schätzer ist konsistent und effizient. Anstatt die Likelihood zu maximieren, können wir auch die Negative Log-Likelihood (NLL) minimieren.

$$\hat{\theta}_{ML} = \underset{\theta}{\operatorname{argmax}}\ p(X|\theta) = \underset{\theta}{\operatorname{argmin}}\ -\log(p(X|\theta))$$

Wenn die Daten $X = \{x_1, ..., x_n\}$ iid (d.h. unabhängig und aus der selben Verteilung) gezogen wurden, dann gilt:

$$\hat{\theta}_{ML} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} p(x_i|\theta) = \underset{\theta}{\operatorname{argmin}} \underbrace{-\sum_{i=1}^{n} \log(p(x_i|\theta))}_{\text{"Loss" oder "Fehler"}} = \underset{\theta}{\operatorname{argmin}} \underbrace{f(\theta)}_{\text{oder...}=L(\theta)=E(\theta)}$$

Man kann den ML-Schätzer also als das Minimum einer Fehlerfunktion betrachten, welche durch die Summe von Fehlertermen über die einzelnen Datenpunkte x_i gegeben ist.

Fehler-Funktionen

Training in Machine Learning $\approx$ Maximum Likelihood-Schätzer

In Machine-Learning möchten wir während des Trainings ebenfalls eine Funktion $f(\theta)$, minimieren oder maximieren. Diese wird üblicher weise als "Objective Function" bezeichnet; im Kontext der Minimierung oft auch als *Kostenfunktion (cost function), Fehlerfunktion (error function)* E *oder loss function* L. Die Variable θ, bezüglich der wir optimieren möchten entspricht üblicherweise den Modellparametern (z.B. den Gewichten bei neuronalen Netzen).
Oft kommt für diese Fehlerfunktion die NLL zum Einsatz (siehe z.B. Cross-Entropy, MSE, MAE aus der letzten Vorlesung) und somit entspricht der Machine-Learning Algorithmus genau dem Maximum-Likelihood Punktschätzer.
$\Rightarrow$ wie können wir das Minimum dieser Fehlerfunktion finden?

Optimierung

Optimierung umspannt ein breites Gebiet welches insbesondere Probleme der Minimierung und Maximierung einer Funktion $f(\theta)$ bezüglich der Wertes θ enthält[1]. Oft sind wir dabei nicht am Minimum einer Funktion $f(\theta)$ selbst interessiert, sondern an dem Ort θ, an dem das Minimum vorliegt.

[1] Hier betrachten wir nur Minimierungsprobleme. Die Maximierung einer Funktion $f(\theta)$ kann als Minimierung von $(-f(\theta)$ aufgefasst werden.

Error Surface

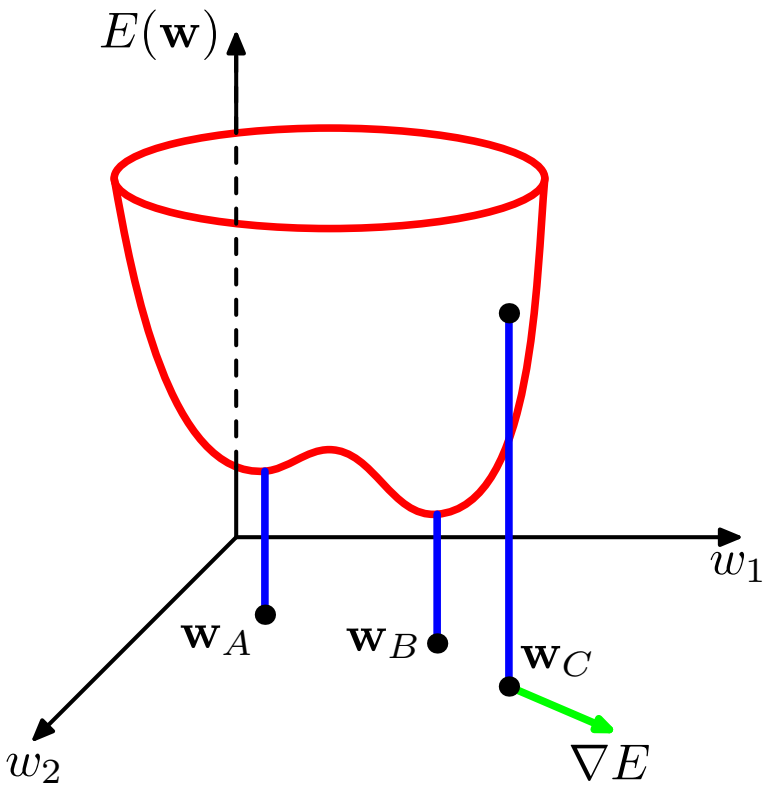


Figure: von Bishop. Geometrische Anschauung der Fehlerfunktion $E(\boldsymbol{w})$ als eine Oberfläche im Gewichtsraum von $\boldsymbol{w}$. $\boldsymbol{w}_A$ und $\boldsymbol{w}_B$ sind ein lokales bzw das globale Minimum von $E(\boldsymbol{w})$. An einem Punkt $\boldsymbol{w}_C$ ist der lokale Gradient der Fehlerfunktion durch den Vektor ∇E gegeben.

Stationäre Punkte

Stationäre (oder auch kritische) Punkte einer stetig differenzierbaren Funktion $f(\theta)$ sind Punkte, an denen die Ableitung $\partial_\theta f(\theta) = 0$ ist. Dies können z.B. lokale Minima, Maxima oder Sattelpunkte sein:

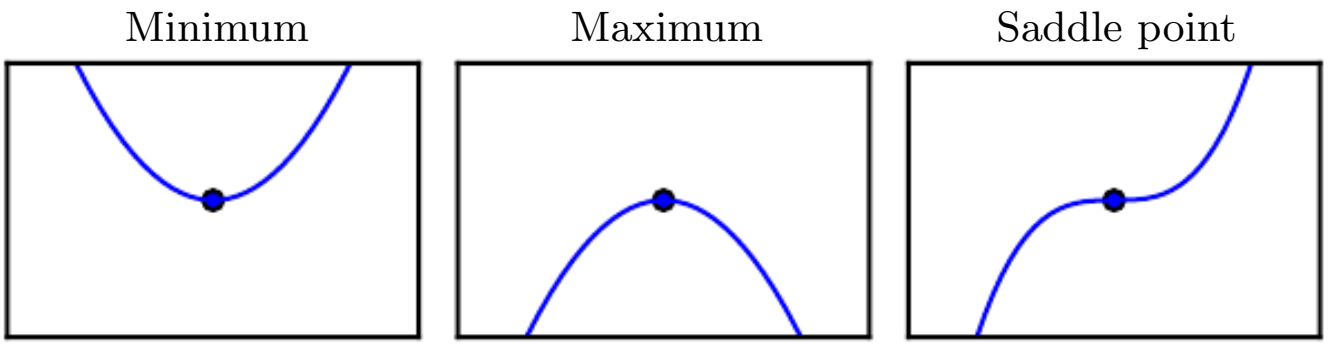


Figure: 4.2 von Goodfellow

.

Im Allgemeinen kann eine Funktion $f(\theta)$ mehrere kritische Punkte besitzen.

Lokale und Globale Minima

Sei $D \subseteq \mathbb{R}^n$ und $f : D \to \mathbb{R}$ ein Funktion.

Globales Minimum

An einer Stelle $\hat{\theta}$ liegt ein *globales Minimum* vor, wenn $f(\hat{\theta}) \leq f(\theta) \forall \theta \in D$.

Lokales Minimum

An einer Stelle $\hat{\theta}$ liegt ein *lokales Minimum* vor, wenn eine Umgebung U um $\hat{\theta}$ existiert, sodass $f(\hat{\theta}) \leq f(\theta) \, \forall \theta \in U$.

Es ist üblicherweise nicht möglich ein globales Minimum zu finden, weil es viele lokale Minima gibt (z.B. aufgrund von nichtlinearen Abhängigkeiten / Symmetrien in den Modellparametern). Oftmals zeigen lokale Minima aber bereits eine gute Performanz.

Lokale und Globale Minima

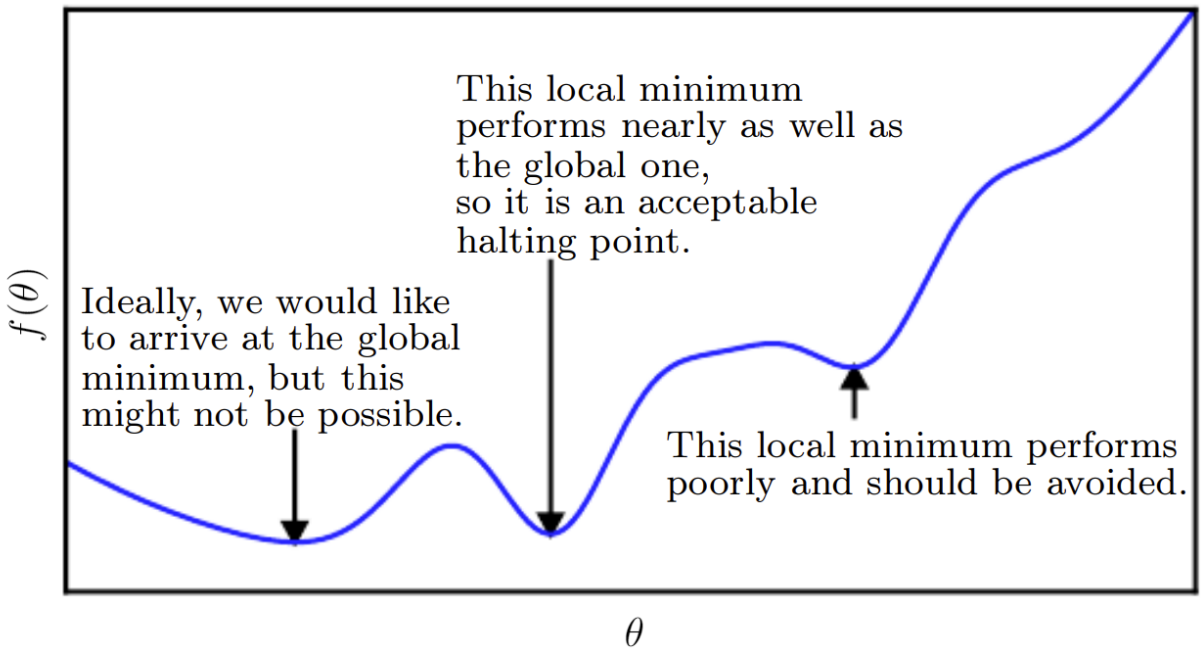


Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function.

Figure: 4.3 von Goodfellow: Lokale Minima können ebenfalls eine akzeptable Lösung sein.

Konvexität

Sei $D \subseteq \mathbb{R}^n$ und $f : D \to \mathbb{R}$ ein Funktion. Dann ist f konvex, wenn für alle $x, y \in D$ und $t \in [0, 1]$ gilt, dass:

$$f(x\,t + y\,(1-t)) \leq t f(x) + (1-t) f(y)$$

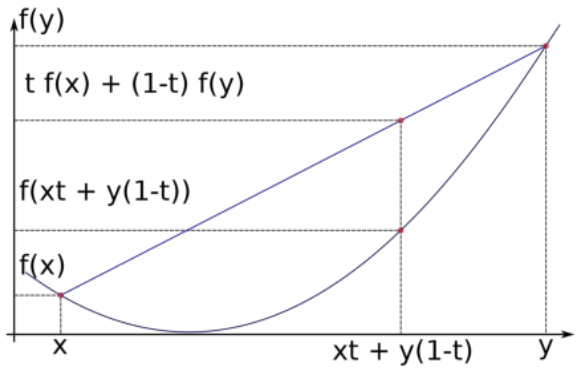


Figure: Veranschaulichung von Konvexität (Quelle: [Wikipedia])

Wenn f konvex ist und ein lokales Minimum bei $\hat{\theta}$ besitzt, so ist $\hat{\theta}$ auch ein globales Minimum.
$\Rightarrow$ Leider ist in Machine Learning die Fehlerfunktion üblicherweise nicht konvex bezüglich der Eingabe (welche üblicherweise den Netzwerkparametern entspricht).

Wie können wir Minima finden?

Es gibt zahlreiche Optimierungsstrategien, um Minima einer Fehlerfunktion zu finden. Zum Beispiel:

Least Squares Optimierung eines linearen Modells

Um ein lineares Modell von der Form $y(x) = \theta_1 a_1(x) + \theta_2 a_2(x) + ... + \theta_m a_m(x)$ an Daten (x_i, y_i) mit least squares zu fitten, kann man die Normalengleichung lösen:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \, ||A\theta - y||^2 = (A^* A)^{-1} A^* y, \text{ wobei: } A_{ij} = a_j(x_i)$$

Problem: Diese Methode funktioniert nur für Least Squares. Ausserdem sind ML-Modelle üblicherweise nicht linear!

Gridsearch

Problem: Der Rechenaufwand skaliert exponentiell mit der Anzahl an Parametern! Schon bei wenigen Parametern stösst dieses Verfahren an seine Grenzen (Bei lediglich 6 Parametern und einer geringen Gitterauflösung von 10 Punkten müssten der Fehler bereits 10^6 mal ausgewertet werden).

Wie können wir Minima finden?

Evolutionäre Algorithmen

Evolutionäre Algorithmen (zum Beispiel CMA-ES) funktionieren gut bis ca. 100-1000 Parameter. Darüber hinaus werden sie allerdings ineffizient und Neuronale Netze haben üblicherweise deutlich mehr als 1000 Parameter. Zur Bestimmung von Hyperparametern können sie dennoch hilfreich sein (für weitere Informationen siehe z.B. auch [Nevergrad]).

Gradientenbasierte Optimierung

Wenn das Modell und die Fehlerfunktion differenzierbar sind, dann können wir Gradienteninformationen für die Optimierung ausnutzen. Dies erlaubt auch noch bei sehr grossen Modellen eine effiziente Optimierung.

Part 2

Gradient Descent

Chapter 4.3 of **Goodfellow-et-al-2016-Book** und Kapitel 35.4 in "Mathematik" von Tilo Arens et al

Gradient Descent - Motivation

Gesucht: $\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ f(\boldsymbol{\theta})$

Da wir dieses globale Minimum nicht analytisch finden können, müssen wir ein numerisches Verfahren verwenden, um $\hat{\boldsymbol{\theta}}$ iterativ zu approximieren. Dazu wählen wir einen Anfangswert $\boldsymbol{\theta}_0$ und berechnen dann iterativ Update-Schritte:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \Delta\boldsymbol{\theta}_i$$

Hierbei verwenden verschiedene Algorithmen unterschiedliche Update-Schritte $\Delta\boldsymbol{\theta}_i$. Beachte, dass solche numerischen Verfahren üblicherweise nur lokale Minima finden können (insbesondere, wenn $f(\theta)$ nicht konvex ist).

Gradient Descent - Motivation

Können wir Informationen aus der Ableitung von $f(\theta)$ benutzen, um den Update-Schritt $\Delta\boldsymbol{\theta}_i$ zu berechnen?

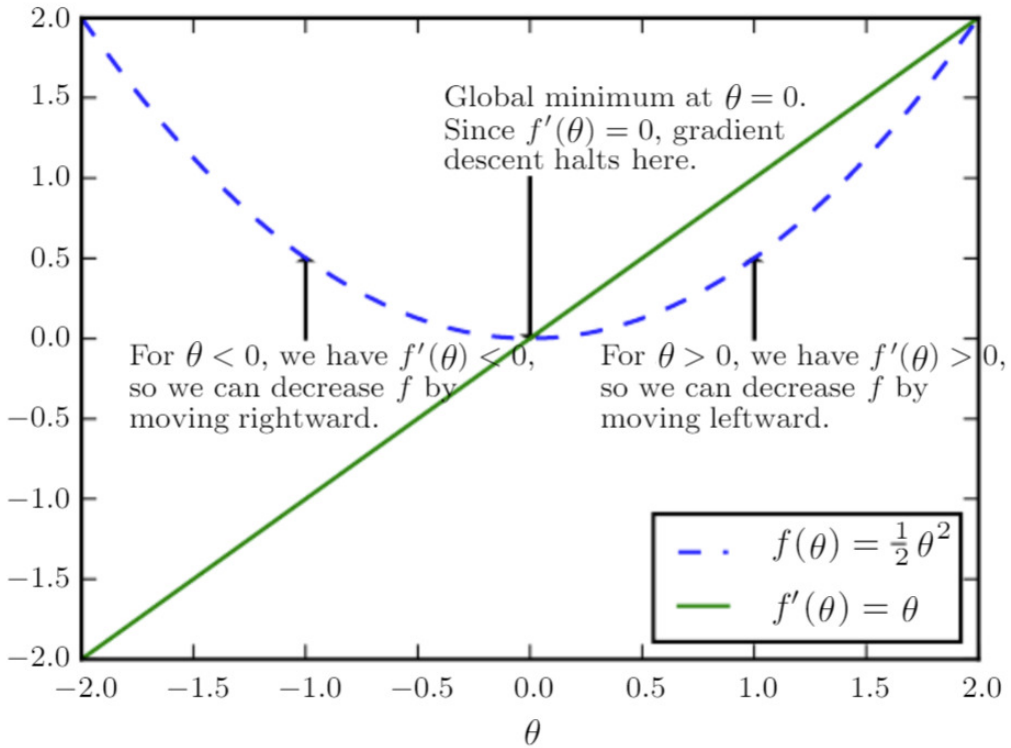


Figure 4.1: An illustration of how the gradient descent algorithm uses the derivatives of a function can be used to follow the function downhill to a minimum.

Figure: 4.1 von Goodfellow: die Ableitung kann Informationen über die Richtung und Distanz zum Minimum enthalten.

Gradient Descent Optimization

Eines der grundlegendsten Updates für $\Delta\boldsymbol{\theta}_i$ ist, einen kleinen Schritt entlang der Richtung des negativen Gradientens von $f(\theta)$ zu gehen. D.h.:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \underbrace{\epsilon}_{\substack{\text{learning} \\ \text{rate}}} f'(\boldsymbol{\theta}_i).$$

wobei ϵ die *learning rate* ist – ein positiver Skalar-Wert, welcher die Schrittweite des Verfahrens entlang des negativen Gradienten vorgibt.

... dies ist das standard Gradientenabstiegsverfahren.

Gradienten - Ableitung in mehreren Dimensionen

Die Fehlerfunktion eines Neuronalen Netzes ist üblicherweise von vielen Parametern (insbesondere den Gewichten des Neuronalen Netzes) abhängig. Deshalb müssen wir mit den partiellen Ableitungen dieser Parameter arbeiten. Dazu führen wir das Konzept des *Gradienten* ein.

Gradient

Sei $f(\theta) : \mathbb{R}^n \to \mathbb{R}$ eine skalare Funktion, dann gibt der Gradient ein Vektorfeld zurück, welches die partiellen Ableitungen von $f(\theta)$ enthält.

$$(\nabla f(\boldsymbol{\theta}))_i = \frac{\partial}{\partial \theta_i} f(\boldsymbol{\theta}).$$

Der Gradient generalisiert das Konzept der Ableitung auf mehrere Dimensionen und wird mit dem Nabla-Operator ∇ abgekürzt. $\nabla f(\boldsymbol{\theta})$ gibt einen Gradientenvektor zurück, dessen i-te Einträge jeweils die i-te partielle Ableitung $\frac{\partial}{\partial \theta_i} f(\boldsymbol{\theta})$ enthalten. Die partielle Ableitung $\frac{\partial}{\partial \theta_i} f(\boldsymbol{\theta})$ misst, wie sich f ändert, wenn wir lediglich θ_i von $\boldsymbol{\theta}$ ändern. Ein stationärer Punkt ist dann gegeben, wenn jeder Eintrag des Gradienten null ist.

Richtungsableitung

Eine Richtungsableitung $\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta})$ von $f(\boldsymbol{\theta})$ in der Richtung eines Einheitsvektors $\boldsymbol{u}$ ist definiert als die Änderungsrate von f in der Richtung von $\boldsymbol{u}$ an der Stelle $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta}) = \lim_{\alpha \to 0} \frac{f(\boldsymbol{\theta} + \alpha \boldsymbol{u}) - f(\boldsymbol{\theta})}{\alpha} = \boldsymbol{u}^{\intercal} \cdot \nabla f(\boldsymbol{\theta}).$$

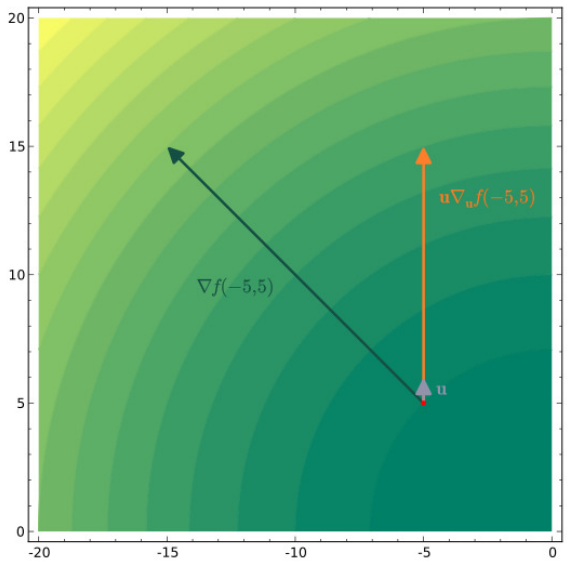


Figure: Plot einer Richtungsableitung [Link]. Konturplot von $f(\theta_x, \theta_y) = \theta_x^2 + \theta_y^2$, mit Gradientenvektor $\nabla f(\boldsymbol{\theta})$ in blau und Richtungsableitung $\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta})$ in orange.

Richtungsableitung

Eine Richtungsableitung $\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta})$ von $f(\boldsymbol{\theta})$ in der Richtung eines Einheitsvektors $\boldsymbol{u}$ ist definiert als die Änderungsrate von f in der Richtung von $\boldsymbol{u}$ an der Stelle $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta}) = \lim_{\alpha \to 0} \frac{f(\boldsymbol{\theta} + \alpha \boldsymbol{u}) - f(\boldsymbol{\theta})}{\alpha} = \boldsymbol{u}^\intercal \cdot \nabla f(\boldsymbol{\theta}).$$

Um f zu minimieren, müssen wir die Richtung $\boldsymbol{u}$ finden, entlang der f am schnellsten abnimmt, d.h.:

$$\min_{\boldsymbol{u}, \boldsymbol{u}^\intercal \boldsymbol{u} = 1} \boldsymbol{u}^\intercal \nabla f(\boldsymbol{\theta}) = \min_{\boldsymbol{u}, \boldsymbol{u}^\intercal \boldsymbol{u} = 1} \|\boldsymbol{u}\|_2 \cdot \|\nabla f(\boldsymbol{\theta})\|_2 \cdot \cos(\phi),$$

wobei ϕ der Winkel zwischen $\nabla f(\boldsymbol{\theta})$ und $\boldsymbol{u}$ ist. Da $\boldsymbol{u}$ ein Einheitsvektor ist, gilt $\|\boldsymbol{u}\|_2 = 1$ wodurch die obige Minimierung zu $\min_{\boldsymbol{u}} \cos(\phi)$ vereinfacht werden kann, da $\nabla f(\boldsymbol{\theta})$ nicht von $\boldsymbol{u}$ abhängt.

Q: Wann wird $\min_{\boldsymbol{u}} \cos(\phi)$ minimiert?

Richtungsableitung

Eine Richtungsableitung $\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta})$ von $f(\boldsymbol{\theta})$ in der Richtung eines Einheitsvektors $\boldsymbol{u}$ ist definiert als die Änderungsrate von f in der Richtung von $\boldsymbol{u}$ an der Stelle $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{u}} f(\boldsymbol{\theta}) = \lim_{\alpha \to 0} \frac{f(\boldsymbol{\theta} + \alpha \boldsymbol{u}) - f(\boldsymbol{\theta})}{\alpha} = \boldsymbol{u}^\intercal \cdot \nabla f(\boldsymbol{\theta}).$$

Um f zu minimieren, müssen wir die Richtung $\boldsymbol{u}$ finden, entlang der f am schnellsten abnimmt, d.h.:

$$\min_{\boldsymbol{u}, \boldsymbol{u}^\intercal \boldsymbol{u} = 1} \boldsymbol{u}^\intercal \nabla f(\boldsymbol{\theta}) = \min_{\boldsymbol{u}, \boldsymbol{u}^\intercal \boldsymbol{u} = 1} \|\boldsymbol{u}\|_2 \cdot \|\nabla f(\boldsymbol{\theta})\|_2 \cdot \cos(\phi),$$

wobei ϕ der Winkel zwischen $\nabla f(\boldsymbol{\theta})$ und $\boldsymbol{u}$ ist. Da $\boldsymbol{u}$ ein Einheitsvektor ist, gilt $\|\boldsymbol{u}\|_2 = 1$ wodurch die obige Minimierung zu $\min_{\boldsymbol{u}} \cos(\phi)$ vereinfacht werden kann, da $\nabla f(\boldsymbol{\theta})$ nicht von $\boldsymbol{u}$ abhängt.

Q: Wann wird $\min_{\boldsymbol{u}} \cos(\phi)$ minimiert?
... wenn $\cos(\phi) = -1$, d.h. wenn $\phi = \pi$ und $\boldsymbol{u}$ also in die entgegengesetzte Richtung von $\nabla f(\boldsymbol{\theta})$ zeigt.

Gradient Descent

Wir haben gesehen, dass die Richtungsableitung von $f(\boldsymbol{\theta})$ entlang $\boldsymbol{u}$ genau dann minimiert wird, wenn $\boldsymbol{u}$ in die gegensätzliche Richtung von $(\boldsymbol{\theta})$ zeigt. D.h. wir können f am schnellsten verringern, wenn wir uns entlang der Richtung des negativen Gradienten bewegen. Diese Methode nennt man auch "steepest descent" (steilster Abstieg).

Wir bewegen uns von $\boldsymbol{\theta}_i$ zu einem neuen Punkt $\boldsymbol{\theta}_{i+1}$,

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \epsilon \nabla f(\boldsymbol{\theta}_i),$$

wobei ϵ die *learning rate* ist – ein positiver Skalar-Wert, welcher die Schrittweite des Verfahrens entlang des negativen Gradienten vorgibt.

Learning Rate ϵ

Die Auswahl eines guten ϵ kann kompliziert sein. Wenn es zu klein ist, wird die Konvergenz zum Minimum langsam. Wenn ϵ zu gross ist, dann kann es sein, dass das Verfahren gar nicht zum Minimum konvergiert. Für jetzt wählen wir ein einfaches Vorgehen, indem wir ϵ auf eine kleine Konstante setzen.

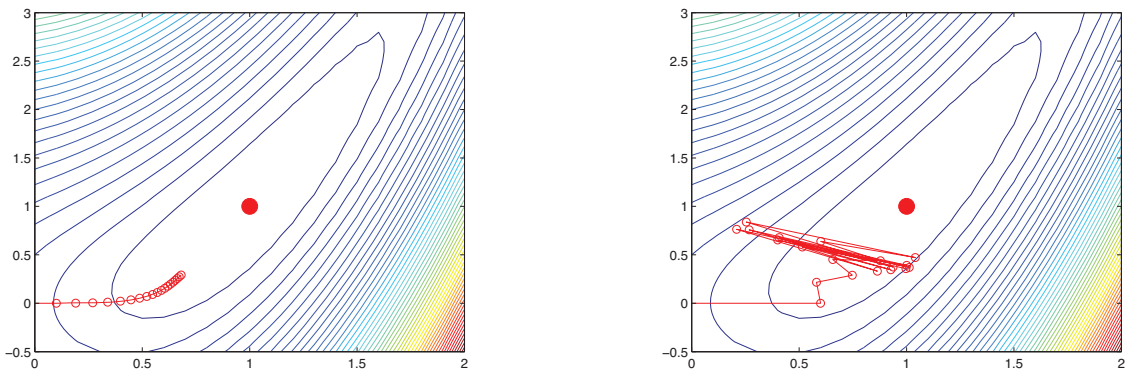


Figure 8.2 Gradient descent on a simple function, starting from $(0, 0)$, for 20 steps, using a fixed learning rate (step size) η. The global minimum is at $(1, 1)$. (a) $\eta = 0.1$. (b) $\eta = 0.6$. Figure generated by `steepestDescentDemo`.

Figure: 8.2 von Murphy.

Jacobi-Matrix

Partielle Ableitungen

Für eine Funktion f mit Vektoren als Inputs und Outputs (d.h. $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n$) können wir eine *Jacobi-Matrix* $\boldsymbol{J_f}(\boldsymbol{\theta}) \in \mathbb{R}^{n \times m}$ einführen, welche die partiellen Ableitungen von $\boldsymbol{f}$ an der Stelle x enthält:

$$(\boldsymbol{J_f}(\boldsymbol{\theta}))_{i,j} = \frac{\partial}{\partial \theta_j} \boldsymbol{f}(\boldsymbol{\theta})_i$$

Ableitungen zweiter Ordnung

Ableitungen zweiter Ordnung sind "Ableitungen von Ableitungen". Die zweite Ableitung ist eine Grösse für die Krümmung einer Funktion und ist interessant für Gradientenbasierte Optimierungsverfahren, weil sie andeuten kann, wie viel Verbesserung mit einem Gradienten-Schritt zu erwarten ist.

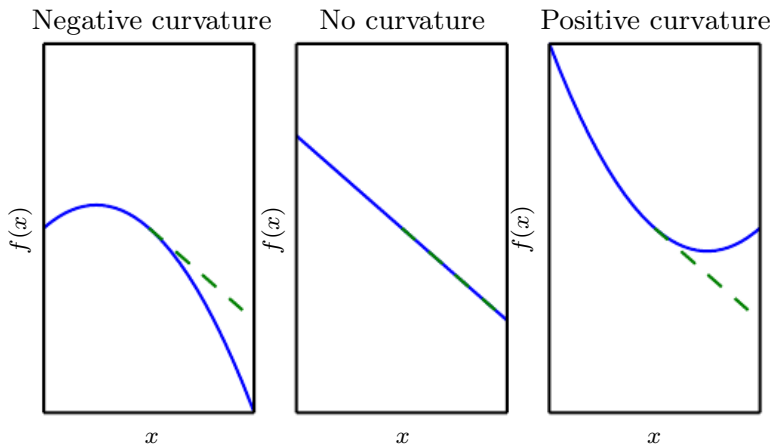


Figure 4.4: The second derivative determines the curvature of a function. Here we show quadratic functions with various curvature. The dashed line indicates the value of the cost function we would expect based on the gradient information alone as we make a gradient step downhill. In the case of negative curvature, the cost function actually decreases faster than the gradient predicts. In the case of no curvature, the gradient predicts the decrease correctly. In the case of positive curvature, the function decreases slower than expected and eventually begins to increase, so steps that are too large can actually increase the function inadvertently.

Figure: 4.4 von Goodfellow.

Hesse-Matrix

Die *Hesse-Matrix* $\boldsymbol{H} \in \mathbb{R}^{n \times n}$ enthält die Ableitungen zweiter Ordnung einer Funktion $f : \mathbb{R}^n \to \mathbb{R}$ und ist das Äquivalent zur Jacobi-Matrix $\boldsymbol{J}_{\nabla \boldsymbol{f}}(\theta)$ des Gradienten $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$. Ihre Matrixeinträge für i, j sind definiert als:

$$(\boldsymbol{H}_f(\boldsymbol{\theta}))_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} f(\boldsymbol{\theta}) \ = (\boldsymbol{J}_{\nabla \boldsymbol{f}}(\boldsymbol{\theta}))_{i,j}.$$

Die Richtungsableitung zweiter Ordnung entlang eines Richtungsvektors $\boldsymbol{u}$ ist gegeben durch $\boldsymbol{u}^\intercal \boldsymbol{H} \boldsymbol{u}$. Wenn $\boldsymbol{u}$ ein Eigenvektor von $\boldsymbol{H}$ ist, dann entspricht die zweite Ableitung dem entsprechendem Eigenwert. Für andere Richtungen entspricht die zweite Ableitung einem gewichteten Mittel von Eigenwerten, wobei solche Richtungen höher gewichtet werden, die besser mit den jeweiligen Eigenvektoren übereinstimmen. Dies impliziert, dass die zweite Richtungsableitung zwischen dem kleinsten und grössten Eigenwert von $\boldsymbol{H}$ liegen muss.

Approximate Second-Order Methods

Bisher haben wir lediglich Methoden erster Ordnung behandelt. Methoden zweiter Ordnung nutzen auch zweite Ableitungen um die Optimierung zu beschleunigen.

Newton Verfahren

Das Newton Verfahren wird üblicherweise verwendet um Nullstellen zu finden. Wenn wir es auf den Gradienten der Fehlerfunktion $\nabla f(\boldsymbol{\theta})$ anwenden, können damit Nullstellen des Gradienten und somit stationäre Punkte der Fehlerfunktion $f(\boldsymbol{\theta})$ gefunden werden. Dazu machen wir eine Taylorentwicklung um den Gradienten $\nabla f(\boldsymbol{\theta}_i)$ an der Stelle $\boldsymbol{\theta}_i$ und setzen gleich 0:

$$\nabla f(\boldsymbol{\theta}^*) \approx \nabla f(\boldsymbol{\theta}_i) + \boldsymbol{H}_f(\boldsymbol{\theta}_i)(\boldsymbol{\theta}^* - \boldsymbol{\theta}_i) \stackrel{!}{=} 0$$

Wobei $\boldsymbol{H}_f(\boldsymbol{\theta}_i)$ die Hessian von f bezüglich $\boldsymbol{\theta}$ an der Stelle $\boldsymbol{\theta}_i$ ist. Wenn wir dies nun nach dem kritischen Punkt $\boldsymbol{\theta}^*$ auflösen, erhalten wir folgende Update-Regel:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - (\boldsymbol{H}_f(\boldsymbol{\theta}_i))^{-1} \nabla f(\boldsymbol{\theta}_i)$$

Wenn man das Verfahren mit x_0 nahe genug an einem Minimum initialisiert wird und die Hesse-Matrix positiv definit ist, konvergiert das Newtonverfahren gegen dieses Minimum. Dies ähnelt dem Gradienten-Abstiegsverfahren erster Ordnung, allerdings ist das update effizienter, da die Hessian auch die Krümmung von f berücksichtigt.

Approximate Second Order Methods

Wenn wir das Newton update betrachten:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - (\boldsymbol{H}_f(\boldsymbol{\theta}_i))^{-1} \nabla f(\boldsymbol{\theta}_i)$$

... was sind die Vorteile / Nachteile der Newton Methode?

Weitere Methoden

Andere Methoden zweiter Ordnung wie zum Beispiel *conjugate gradients* und quasi-Newton Methoden wie *BFGS* und *L-BFGS* versuchen die Berechnung der Inversen der Hesse-Matrix zu umgehen (siehe auch Kapitel 35.4. in "Mathematik" von Tilo Arens et al). Allerdings ist ihre naive Anwendung auf grossen Trainingssets oft unpraktisch und noch ein offenes Forschungsgebiet.

Zu grosse Schrittweite ϵ

Mit Hilfe der Hesse Matrix können wir nochmals genauer den Effekt von zu grossen Schrittweiten ϵ analysieren:

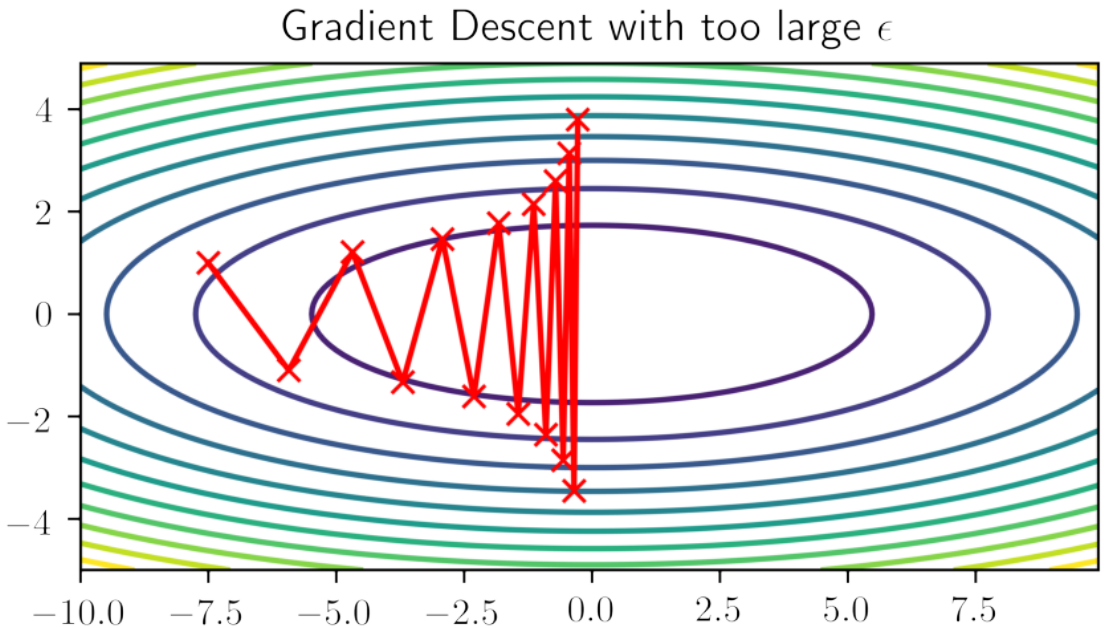


Figure: Gradient Descent mit zu grossem ϵ.

Zu grosse Schrittweite ϵ

Mit Hilfe der Hesse Matrix können wir nochmals genauer den Effekt von zu grossen Schrittweiten ϵ analysieren:
Nehmen wir an, wir nutzen eine learning rate ϵ und einen Gradienten $\boldsymbol{g} = \nabla f(\boldsymbol{\theta}_i)$ an der Stelle $\boldsymbol{\theta}_i$, um den nächsten Update-Schritt $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \epsilon \boldsymbol{g}$ zu berechnen. Dann können wir $f(\boldsymbol{\theta}_{i+1})$ durch eine Taylor-Entwicklung zweiter Ordnung annähern:

$$f(\boldsymbol{\theta}_{i+1}) \approx f(\boldsymbol{\theta}_i) + (\boldsymbol{\theta}_{i+1} - \boldsymbol{\theta}_i)^\intercal \boldsymbol{g} + \frac{1}{2}(\boldsymbol{\theta}_{i+1} - \boldsymbol{\theta}_i)^\intercal \boldsymbol{H}(\boldsymbol{\theta}_{i+1} - \boldsymbol{\theta}_i)$$

$$= \underbrace{f(\boldsymbol{\theta}_i)}_{\text{original cost}} - \underbrace{\epsilon \boldsymbol{g}^\intercal \boldsymbol{g}}_{\substack{\text{expected improv.} \\ \text{from slope}}} + \underbrace{\frac{1}{2}\epsilon^2 \boldsymbol{g}^\intercal \boldsymbol{H} \boldsymbol{g}}_{\substack{\text{correction to account} \\ \text{for curvature of function}}}$$

... wenn nun $\frac{1}{2}\epsilon^2 \boldsymbol{g}^\intercal \boldsymbol{H} \boldsymbol{g} > \epsilon \boldsymbol{g}^\intercal \boldsymbol{g}$ ist, dann ist $f(\boldsymbol{\theta}_{i+1}) > f(\boldsymbol{\theta}_i)$ und das Verfahren konvergiert nicht.

Ill Conditioning

Eine grosse Herausforderung bei der Optimierung mit Gradient Descent ist, wenn die Hesse-Matrix schlecht konditioniert ist. Dies führt dazu, dass die Error-surface einem "schmalen Tal" entspricht, sodass wir kleine learning rates ϵ wählen müssen und die Gradienten oft nicht zum Minimum zeigen, was zu "Zick-Zack" Pfaden führen kann:

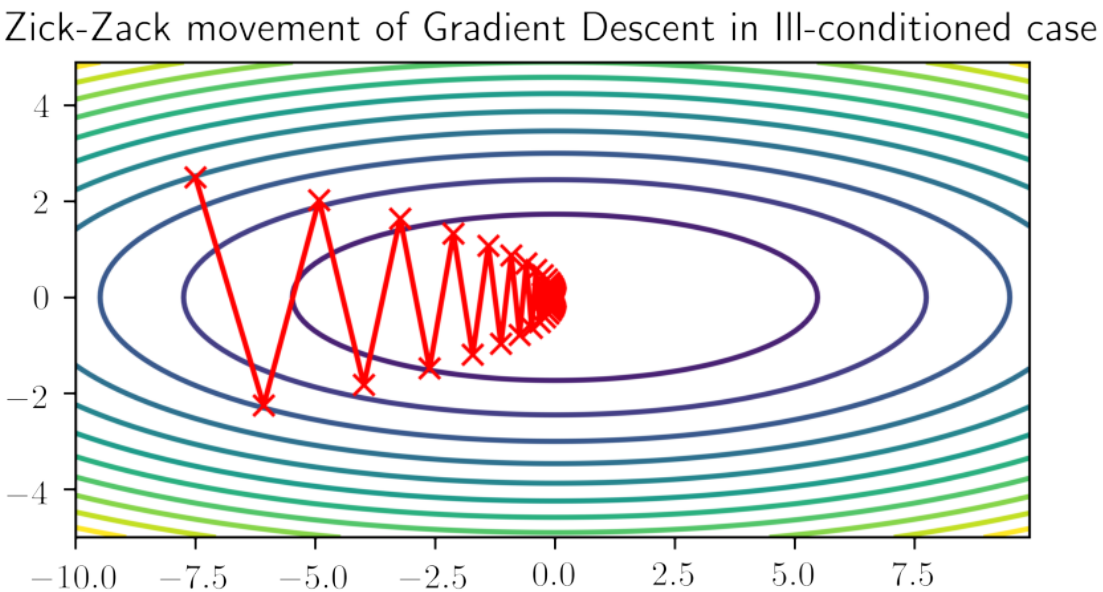


Figure: Gradienten zeigen oft nicht zum Minimum, wenn die Hesse Matrix schlecht konditioniert ist

III Conditioning

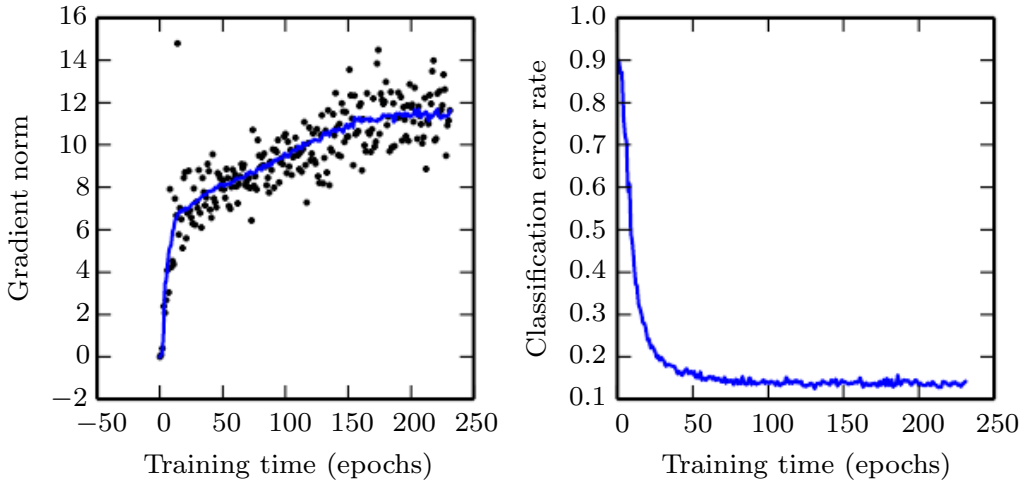


Figure 8.1: Gradient descent often does not arrive at a critical point of any kind. In this example, the gradient norm increases throughout training of a convolutional network used for object detection. *(Left)*A scatterplot showing how the norms of individual gradient evaluations are distributed over time. To improve legibility, only one gradient norm is plotted per epoch. The running average of all gradient norms is plotted as a solid curve. The gradient norm clearly increases over time, rather than decreasing as we would expect if the training process converged to a critical point. *(Right)*Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.

Figure: 8.1 von Goodfellow. Manchmal steigen die Gradienten während des Trainings (links). Dennoch sinkt der Fehler auf ein akzeptables level (rechts). Man kann die learning rate verringern um die grösseren Krümmungen in der Fehlerfunktion zu kompensieren.

Part 3

Stochastic Gradient Descent

Chapter 4.3 of **Goodfellow-et-al-2016-Book**

Stochastic Gradient Descent

In der Praxis (insbesondere im Kontext von Deep Learning) ist die Berechnung des Gradienten sehr rechenintensiv, wenn er auf sämtlichen Daten des Datensets durchgeführt wird. Des weiteren möchte man üblicherweise innerhalb eines Updateschritts die benötigten Daten im Arbeits- / Grafikspeicher behalten, was jedoch unmöglich wird, wenn wir das gesamte Datenset berücksichtigen.
Deshalb werden üblicherweise stochastische Gradientenabstiegsverfahren verwendet, welche in jedem Schritt nur eine kleine Untermenge (Mini-Batch) des Datensets berücksichtigen:
Nehmen wir an, dass jedes Trainingssample i.i.d. ist, dann haben wir gesehen, dass sich die Negative Log-Likelihood L als eine Summe von Einzeltermen, welche jeweils ein Trainingssample $(x_i, y_i) \in X$ aus dem Datenset X enthalten schreiben lässt:

$$L_\theta(X) = \frac{1}{n}\sum_{i=1}^{n} L(f_\theta(x_i), y_i) \Rightarrow \nabla_\theta L_\theta(X) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L(f_\theta(x_i), y_i)$$

$f_\theta(x)$ bezeichnet hier eine durch θ parametrisierte Funktion (z.B. ein NN).

Mini-Batch-Gradient als erwartungstreuer Schätzer

Die Approximation des Gradienten mit Hilfe eines Mini-Batches kann als ein erwartungstreuer Schätzer des Gradienten $\nabla_\theta L_\theta(X)$ aufgefasst werden. Sei X ein Datenset mit n Trainingsbeispielen und $\mathcal{X}_k = \{X' \subset X : |X'| = k\}$ die Menge aller Mini-Batches aus X der Größe (batchsize) k. Wenn wir nun $\hat{X}$ uniform zufällig aus $\mathcal{X}_k$ ziehen, dann gilt für den Erwartungswert des errechneten Gradienten:

$$\mathbb{E}\left[\nabla_\theta L_\theta(\hat{X})\right] = \mathbb{E}\left[\frac{1}{k}\sum_{x\in\hat{X}} \nabla_\theta L_\theta(x)\right] = \frac{1}{|\mathcal{X}_k|}\sum_{\hat{X}\in\mathcal{X}_k}\frac{1}{k}\sum_{x\in\hat{X}}\nabla_\theta L_\theta(x)$$

$$= \frac{1}{k}\sum_{x\in X}\frac{\binom{n-1}{k-1}}{\binom{n}{k}}\nabla_\theta L_\theta(x) = \frac{1}{k}\sum_{x\in X}\frac{k}{n}\nabla_\theta L_\theta(x) = \frac{1}{n}\sum_{x\in X}\nabla_\theta L_\theta(x) = \nabla_\theta L_\theta(X)$$

$|\mathcal{X}_k| = \binom{n}{k}$ ist die Anzahl aller Minibatches. $\binom{n-1}{k-1}$ gibt an, in wievielen Minibatches aus $\mathcal{X}_k$ ein Datensample $x \in X$ vorkommt.

(Mini)-Batch Algorithmen

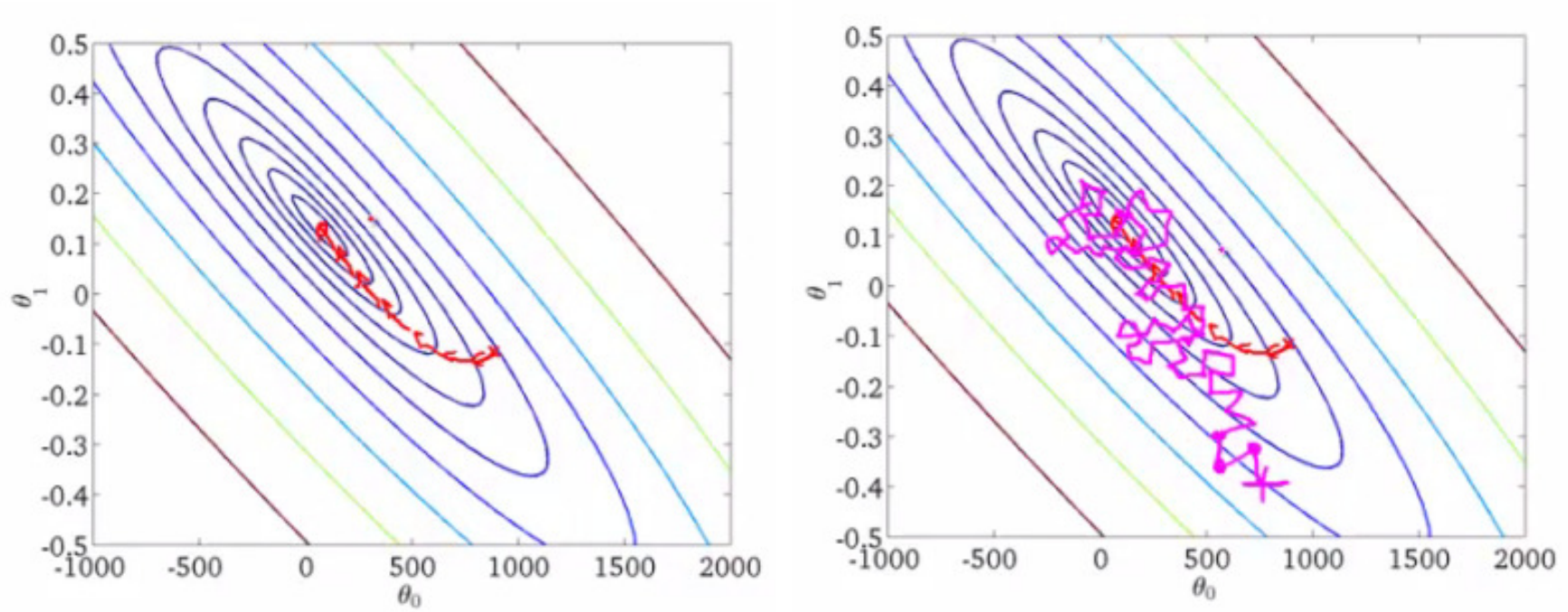


Figure: Gradient descent mit exakten Gradienten (links) und stochastischen Gradienten (rechts). Abbildung von [hier]

Batchgröße

Die Wahl der Batchgröße hängt von verschiedenen Faktoren ab

- Die Varianz des geschätzten Gradienten eines Batches sinkt mit der Batch-Größe. Angenommen I ist die Indexmenge eines Batches und die einzelnen Gradienten $\boldsymbol{g}_i = \nabla_\theta L(f_\theta(x_i), y_i)$ sind i.i.d. mit Varianz σ^2, dann ergibt sich[2] für die Varianz der Gradienten:

$$Var\left(\frac{1}{|I|}\sum_{i\in I}\boldsymbol{g}_i\right) = \frac{1}{|I|^2}\sum_{i\in I} Var(\boldsymbol{g}_i) = \frac{\sigma^2}{|I|}.$$

- Größere Batches liefern eine bessere Schätzung des Gradienten, benötigen jedoch mehr Rechenarbeit.
- Kleine Batchgrößen benötigen durch die größere Varianz in den Gradienten meist eine niedrigere Lernrate damit das Training stabil bleibt. Dieser Trade-off kann das Training verlangsamen.

[2] Das ist nur eine Motivation und kein Beweis. Für eine gründlichere Analyse, siehe z. B. Qian et al., 2020: The Impact of the Mini-batch Size on the Variance of Gradients in Stochastic Gradient Descent

Batchgröße

Die Wahl der Batchgröße hängt von verschiedenen Faktoren ab

- Wenn alle Beispiele eines Batches parallel verarbeitet werden (wie z. B. bei GPUs), dann skaliert der Speicherbedarf mit der Batchgröße. Meist ist die Speichergröße der GPU der limitierende Faktor.
- Multiprozessor-Architekturen können nicht vollständig ausgelastet werden, wenn die Batchgrößen zu klein sind, da ab einer minimalen Größe die fixen Berechnungskosten überwiegen.
- Manche Hardware ist auf bestimmte Arraygrößen optimiert, z. B. sind Zweierpotenzen für GPUs meist von Vorteil
- Kleine Batches haben einen Regularisierungseffekt durch Rauschen in der Approximation

Stochastic Gradient Descent

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

Figure: Algorithm 8.1 von Goodfellow.

Learning Rate ϵ

Ein entscheidender Parameter in SGD ist die learning rate ϵ.

In der Praxis wird die learning rate häufig über die Zeit (bzw Trainingsiterationen) reduziert. Dies ist machbar, da SGD ein Rauschen in den Gradienten verursacht, sodass der Gradient nie verschwinden wird - selbst bei einem Minimum. Eine Bedingung für die Konvergenz von SGD ist, dass:

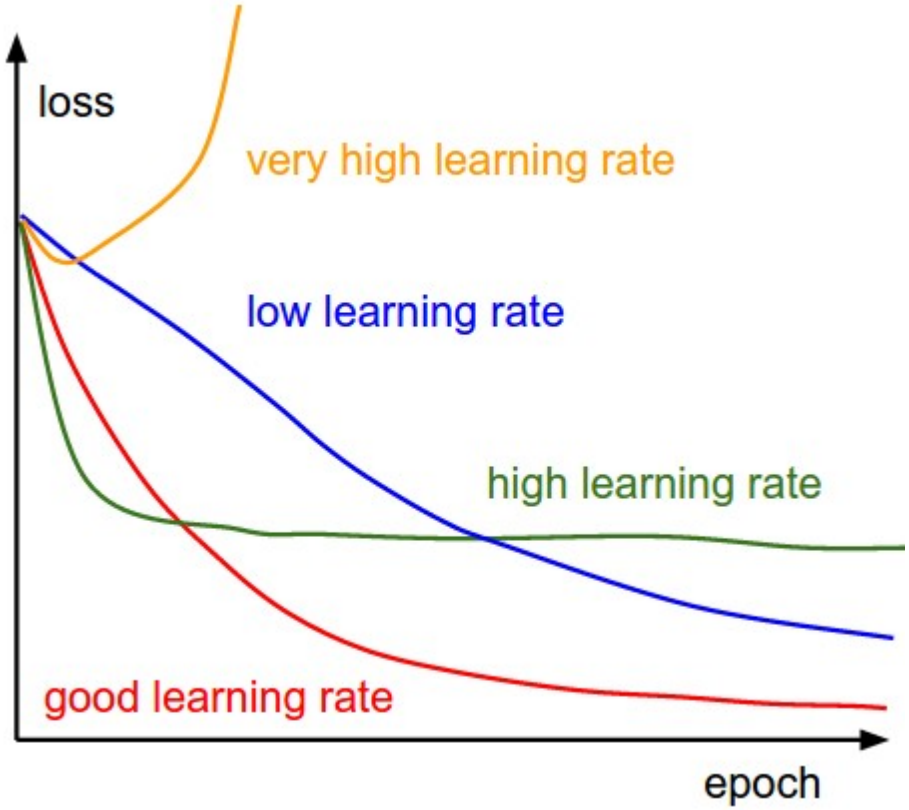
$$\sum_{k=1}^{\infty} \epsilon_k = \infty.$$

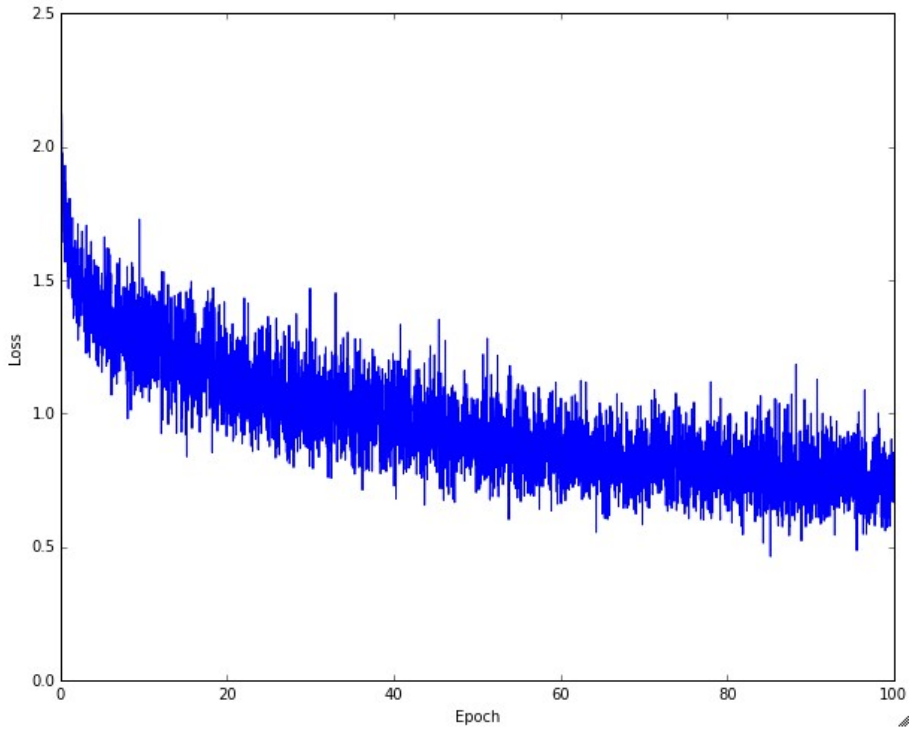
Es ist üblich, die learning rate nach einem festgelegten Plan zu reduzieren (z.B. linearer Rückgang bis Iteration τ, danach konstant):

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau, \qquad \text{wobei } \alpha = \min(\frac{k}{\tau}, 1)$$

Das Optimieren der Hyperparameter $\epsilon_0, \epsilon_\tau$ und τ benötigt üblicherweise etwas "trial and error" und ist oft eher Kunst statt Wissenschaft...

Learning Rate





Abbildungen von [hier].

Learning Rate

Alternativ kann man den Validation Fehler während des Trainings beobachten und die Lernrate verringern, wenn der Fehler eine Ebene erreicht.

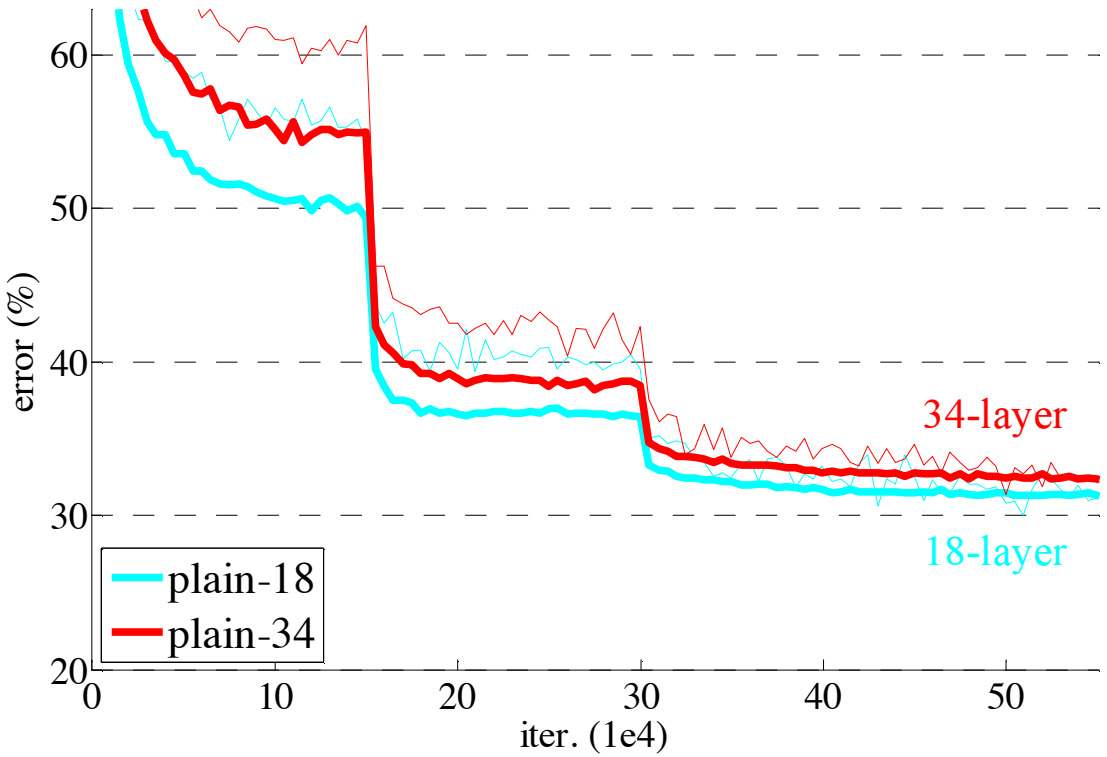


Figure: Abbildung von [Hier]

Impuls (Momentum)

SGD kann langsam sein und *momentum* kann das Verfahren beschleunigen, wenn es starke Krümmungen im Loss, kleine aber konsistente Gradienten oder verrauschte Gradienten gibt. Momentum aggregiert die Gradienten von vergangenen Iterationen mit einer exponentiellem Abnahme um weiter in deren Richtung zu laufen. Dadurch verringert sich die Varianz der Update-Schritte. Die update Regel ist gegeben durch:

$$\boldsymbol{v} \longleftarrow \alpha \boldsymbol{v} - \underbrace{\epsilon \nabla_{\boldsymbol{\theta}} \Big(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{\theta}^{(i)}; \boldsymbol{\theta}), y^{(i)}) \Big)}_{\text{current gradient elements}}$$

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

wobei $\alpha \in [0, 1)$ festlegt, wie schnell die Beiträge der vorherigen Gradienten abnehmen. Dies kann man auch als Low-pass filter für die Gradienten interpretieren.
Wenn wir immer Gradienten $\boldsymbol{g}$ beobachten, dann wird $\boldsymbol{v}$ immer weiter mit $-\epsilon \boldsymbol{g}$ beschleunigt und die Endgeschwindigkeit $\frac{\epsilon \|\boldsymbol{g}\|}{1-\alpha}$ erreichen. Deshalb ist es nützlich, den momentum als einen Hyperparameter $\frac{1}{1-\alpha}$ zu betrachten. Übliche werte für α sind 0.5, 0.9, 0.99.

Momentum

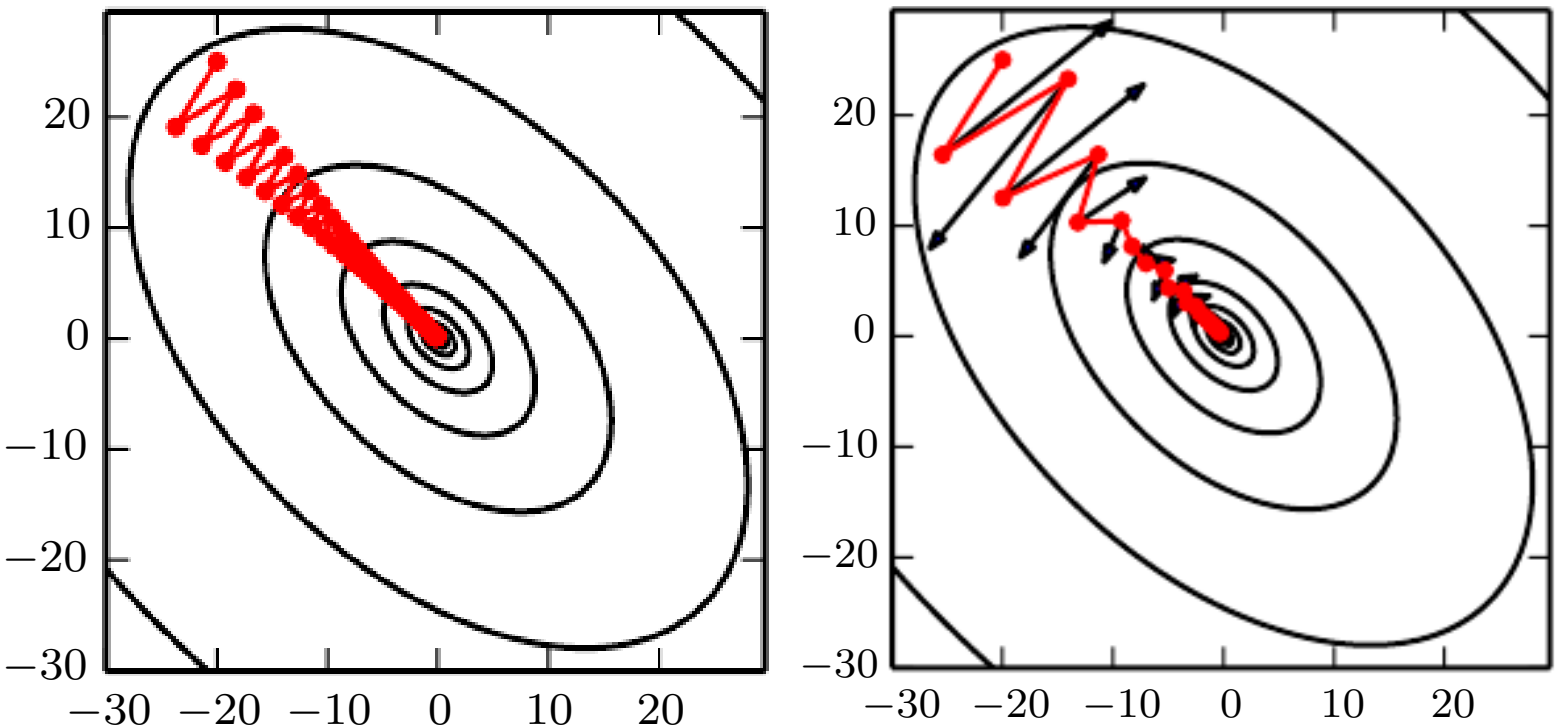


Figure: 4.6 und 8.5 von Goodfellow. Der Einsatz von Momentum versucht Probleme mit (1) schlecht konditionierten Hesse-Matrizen und (2) der Varianz des stochastic gradient zu lösen. Beide Bilder zeigen eine quatrische Loss-Funktion mit einer schlecht konditionierten Hesse-Matrix. Die Konturen stellen dabei ein Tal dar. Links sieht man wie der originale SGD Zeit damit verschwendet in Richtung des größten Gefälles hin- und herzuspringen. Rechts hingegen werden die Schritte durch den zusätzlichen Momentum-Term korrigiert - sie bewegen sich eher längst entlang des Tals.

SGD with Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Figure: Algorithmus 8.2 von Goodfellow.

Part 4

Adaptive Learning Rates

Chapter 8.5, 8.6 of **Goodfellow-et-al-2016-Book**.

Adaptive Learning Rates

Die Wahl des Hyperparameters für die learning rate ϵ ist schwierig und hat einen grossen Einfluss auf die Modell-Performanz.

Die Topologie der objective function kann stark variieren, mit hoher Sensitivität in einigen Richtungen des Parameter-Raums und niedriger Sensitivität in anderen Richtungen. Somit ist es ein logischer Schritt, die learning rate für jeden Parameter separat zu betrachten und automatisch anzupassen. Adaptive learning rate algorithms sind unter anderem:

- **AdaGrad** skaliert eine feste learning rate invers-proportional zur Wurzel aus der Summe aller vorhergegangenen quadrierten Gradienten-Werte.
- **RMSProp** ändert die quadrierte Gradienten-Summe von AdaGrad in einen exponentiell gewichteten gleitenden Mittelwert (moving average)
- **Adam** fügt momentum zu RMSProp hinzu.

Leider gibt es keinen Konsens bezüglich der Wahl des adaptiven Algorithmus. Da sich noch kein einzelner "bester" Algorithmus hervorgetan hat, liegt die Wahl oft bei den Präferenzen des Anwenders und der Problemstellung.

Adaptive Learning Rates: AdaGrad

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Figure: Algorithmus 8.4 von Goodfellow. (Das $\odot$-Symbol steht für elementweise Multiplikation)

Adaptive Learning Rates: RMSProp

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. $(\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

Figure: Algorithmus 8.5 von Goodfellow.

Adaptive Learning Rates: RMSProp with Momentum

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.

Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

Figure: Algorithmus 8.6 von Goodfellow.

Adaptive Learning Rates: Adam

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Figure: Algorithmus 8.7 von Goodfellow.