Einführung in die Neuroinformatik SoSe 2019

Institut für Neuroinformatik
PD Dr. F. Schwenker

8. Aufgabenblatt (Abgabe bis 2. Juli 2019 zur Vorlesung)

Aufgabe 1 (3 Punkte): Momentum-Optimierer [Pen and Paper]

Wir haben bisher auf das klassische Gradientenabstiegsverfahren gesetzt, um ein neuronales Netzwerk zu trainieren. In mehreren Iterationen haben wir dabei die Gewichte anhand der Gradienten in der Fehlerlandschaft angepasst. Auch wenn das klassische Lernverfahren durchaus zum Ziel führt, so kann es unter Umständen einige Zeit dauern, bis alle Gewichte genügend adaptiert wurden. Aus diesem Grund sind einige Ergänzungen zum klassischen Gradientenverfahren über die Jahre entwickelt worden. Ihr gemeinsames Ziel ist es, schneller ein gutes Minimum zu erreichen. In dieser Aufgabe wollen wir uns der Hinzunahme eines Momentumterms widmen.

Die generelle Idee des Momentum-Optimierers ist es, auch den bisherigen Pfad in der Fehlerlandschaft mit in den Updateprozess einzubeziehen. Mit Hilfe von diesen Informationen wird dann versucht, bessere Updateschritte anzuwenden, die schneller zum Ziel führen. Konkret wird die Schrittweite beschleunigt, sofern die Richtung des Gradienten über mehrere Iterationen hinweg gleich bleibt, andernfalls findet wieder eine Stabilisierung statt.

Hier wollen wir uns zuerst mit einem eindimensionalen Beispiel beschäftigen. In diesem Fall sind die Updateregeln für eine Fehlerfunktion $E(w)$ definiert als

$$\begin{aligned} m(0) &= 0 \\ m(t) &= \alpha \cdot m(t-1) + \frac{\partial E}{\partial w(t-1)} \\ w(t) &= w(t-1) - \eta \cdot m(t). \end{aligned} \tag{1}$$

Dabei bezeichnet m den Momentumterm, $\alpha \in [0;1]$ den Momentumfaktor und $\eta > 0$ wie gewohnt die Lernrate.

1. Wir wollen Gleichung 1 zuerst anhand der einfachen Fehlerfunktion

   $$E(w) = 2 \cdot w^2$$

   ausprobieren. Berechne unter Annahme von $\alpha = 0.9, \eta = 0.1$ und $w(0) = 20$ zwei Iterationen des Lernverfahrens, d. h. $w(1)$ und $w(2)$.

2. Der Momentumterm baut auf den zwei grundlegenden Fällen „Beschleunigung“ und „Stabilisierung“ auf. In der folgenden Tabelle sind die Gradienten der ersten beiden Iterationen gegenübergestellt. Entscheide für jeden Fall, ob es zu einer Beschleunigung (+) oder einer Stabilisierung (–) des Momentumterms kommt (Reibungsverluste können ignoriert, d. h. es kann $\alpha = 1$ angenommen werden).

	$\frac{\partial E}{\partial w(1)} < 0$	$\frac{\partial E}{\partial w(1)} > 0$
$\frac{\partial E}{\partial w(0)} < 0$		
$\frac{\partial E}{\partial w(0)} > 0$		

3. Abschließend wollen wir uns noch verschiedene Pfaden des Momentum-Optimierers ansehen. Rufe dazu eine Animation[1] auf, in der in einer zweidimensionalen Fehlerlandschaft verschiedene Parameterkonstellationen getestet werden können. Hinweis zur Abgabe: bitte bei eigenen Anpassungen den Link mit den Einstellungen beilegen.

   a) Erkläre, wieso der Pfad zu weit nach unten und so zuerst etwas über das Ziel hinaus geht. Hinweis: es kann hilfreich sein, den Verlauf des Momentumterms $\boldsymbol{m} = (m_1, m_2)$ einzublenden.

   b) Wieso steigt die m_2-Kurve zuerst an, obwohl die Updateschritte sich nach unten bewegen (das Gewicht w_2 fällt ab)?

   c) Der Nachteil des Momentum-Optimierers ist, dass es zu unerwünschten Oszillationen kommen kann. Finde dazu eine Einstellung, welche dieses Problem verdeutlicht (gerne auch mit einer anderen Fehlerfunktion).

   d) Versuche in maximal 20 Iterationen einen möglichst kleinen Fehlerwert zu erreichen. Finde dazu eine geeignete Parameterkonstellation $(\eta, \alpha, t_{\max})$. Bitte angeben, welcher Fehlerwert erreicht wurde.

Aufgabe 2 (7 Punkte): *Adaptive Learning Rates*

Eine beliebte Erweiterung für Optimizer ist die Berücksichtigung von individuellen Lernraten für jedes Gewicht im Netzwerk. Darin liegt die Vermutung zu Grunde, dass verschiedene Gewichte im Lernprozess auch unterschiedlich stark angepasst werden müssen. Eine globale Lernrate ist daher nicht ausreichend. Es gibt mehrere Umsetzungen für diese Idee und in dieser Aufgabe wollen wir uns dem Konzept der *adaptive learning rates* annähern[2].

Adaptive learning rates haben als weiteres Ziel, die Magnitude der Gradienten etwas zu normalisieren. Dabei wird versucht, sie so zu skalieren, dass sie weder zu klein noch zu groß werden können. Damit sollen die bereits bekannten Probleme *vanishing gradients* und *exploding gradients* gemildert werden.

[1] https://milania.de/blog/Introduction_to_neural_network_optimizers_%5Bpart_1%5D_-_momentum_optimization#fig:MomentumOptimizer_Trajectory

[2] Der dazugehörige Algorithmus ist als „RMSProp“ [2] bekannt, welcher wiederum eine Verbesserung des „AdaGrad“-Algorithmus [1] darstellt.

Generell ist bei Verwendung von individuellen Lernraten wichtig, eine Möglichkeit zu finden, diese automatisch anzupassen. Bei der Menge an lernenden Parametern in einem Netzwerk wäre dies ansonsten nicht praktikabel. *Adaptive learning rates* passen die Lernrate für jedes Gewicht anhand von Skalierungsfaktoren $s(t)$ (mit $s(0) = 0$) an, die separat für jedes Gewicht auf Basis der vorherigen Gradienten gebildet werden (eindimensionaler Fall):

$$\begin{aligned} s(t) &= \beta \cdot s(t-1) + (1-\beta) \cdot \nabla E\left(w(t-1)\right)^2 \\ w(t) &= w(t-1) - \eta \cdot \frac{\nabla E\left(w(t-1)\right)}{\sqrt{s(t) + \varepsilon}} \end{aligned} \tag{2}$$

Dabei ist $\beta \in [0; 1]$ ein neuer Hyperparameter, mit dem wir uns auch noch etwas beschäftigen werden und $\varepsilon \in \mathbb{R}_{>0}$ (z. B. $\varepsilon = 10^{-8}$) ein Glättungsparameter, der verhindert, dass wir durch 0 dividieren. Wie üblich bezeichnet $w(t)$ das Gewicht über die Iterationszeit t, η ist die globale Lernrate und $E(w)$ die Fehlerfunktion.

1. [Python] Berechne die ersten 10 Gewichtsupdates (bis einschließlich $t = 10$) für die Fehlerfunktion

   $$E(w) = -2 \cdot (w+5)^2 + (w-6)^3 \tag{3}$$

   und stelle das Ergebnis grafisch dar (siehe Abbildung 1). Setze die weiteren Parameter auf $\eta = 0.5, \beta = 0.9, \varepsilon = 10^{-8}$ und beginne bei $w(0) = 4$. Implementiere dazu die folgenden Funktionen:

   - `error(w)`: berechnet $E(w)$ für die Werte im Vektor `w`.
   - `error_grad(w)`: berechnet $\nabla E(w)$ für die Werte im Vektor `w`.
   - `adaptive(eta, beta, iterations)` berechnet die Gewichtssequenz $w(t)$ nach Gleichung 2. Der zurückgegebene Vektor sollte `iterations + 1` Werte enthalten.

2. [Pen and Paper] $s(t)$ akkumuliert die Information über die Magnitude aller vorherigen Gradienten und wird in der Gewichtsanpassung dazu verwendet, den Updateschritt entsprechend zu normalisieren. Für die Milderung der genannten Probleme ist dabei vor allem relevant, ob der Faktor $\sqrt{s(t) + \varepsilon}$ größer oder kleiner als 1 ist:

Problem	$\sqrt{s(t) + \varepsilon} < 1$	$\sqrt{s(t) + \varepsilon} > 1$
vanishing gradients		
exploding gradients		

   a) Bestimme für jeden der vier Fälle, ob das Problem jeweils eher gemildert oder eher verstärkt wird.

b) Argumentiere, warum die Fälle, in denen die Probleme sogar noch verstärkt werden, wahrscheinlich weniger häufig auftreten. Hinweis: es empfiehlt sich, die Situation über mehrere Iterationen hinweg zu betrachten.

3. [Pen and Paper] Sehen wir uns nun den Fall $\beta = 0$ an (zudem gelte $\varepsilon = 0$).

   a) Vereinfache Gleichung 2 für diesen Fall und gib an, wie $w(t)$ berechnet wird. Verwende dabei die Signumfunktion

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

   b) Zeige die Grafik aus dem ersten Teil erneut für diesen Fall an und interpretiere das Ergebnis.

   c) Nehmen wir nun an, wir würden die Updateregeln im Zweidimensionalen anwenden. Was sind die möglichen (Himmels-)Richtungen, in die sich die Updates bewegen können?

   d) Nenne ein weiteres Optimierungsverfahren aus der Vorlesung, das diesem Fall ähnlich ist.

4. [Pen and Paper] $\beta = 0$ ist ein Extremfall, in dem die Magnituden der Gradienten gar keine Rolle mehr spielen und nur noch die Vorzeichen (d. h. die Richtungen) relevant sind. Wenn wir nun auch wieder $\beta > 1$ betrachten, so kommt auch der Magnitude wieder eine Bedeutung zu. Um dies noch etwas weiter zu untersuchen, betrachten wir die Gradienten der ersten beiden Iterationen $g_1 = \nabla E(w(0))$ und $g_2 = \nabla E(w(1))$ als Variablen und untersuchen, wie sich der zweite Gradient g_2 durch die Normalisierung verändert (es sei $\eta = 1$ angenommen)

$$C_a(g_1, g_2) = \left| \frac{g_2}{\sqrt{s(2) + \varepsilon}} \right| - |g_2|,$$

   d. h. inwieweit *adaptive learning rates* das Update beeinflussen[3]. Für $C_a(g_1, g_2) > 0$ wird der Gradient verstärkt und für $C_a(g_1, g_2) < 0$ abgeschwächt.

   a) Berechne $C_a(-2, 2)$ für den Fall $\beta = 0$ ($\varepsilon = 0$). Hinweis: für eine schnelle Berechnung ist die Vereinfachung aus der vorherigen Übungsaufgabe hilfreich.

   b) In dieser Animation[4] ist $C_a(g_1, g_2)$ für unterschiedliche Werte für β dargestellt.

[3] Mehr Details können diesem Abschnitt entnommen werden.

[4] https://milania.de/blog/Introduction_to_neural_network_optimizers_[part_2]_%E2%80%93_adaptive_learning_rates_(RMSProp%2C_AdaGrad)#fig:AdaptiveLearning_DecayingParameter

i. Erkläre, wie aus der Grafik abgelesen werden kann, dass die Bedeutung der Magnitude für steigendes β zunimmt.

ii. Erkläre, wie die Ergebnisse der Grafik mit den Normalisierungeigenschaften (Verstärkung von zu kleinen und Abschwächung von zu großen Gradienten) aus der zweiten Teilaufgabe zusammenpassen.

5. [Pen and Paper] Abschließend wollen wir uns auch für dieses Optimierungsverfahren noch unterschiedliche Pfade ansehen. Rufe dazu die entsprechende Animation[5] auf, in der in einer zweidimensionalen Fehlerlandschaft wieder verschiedene Parameterkonstellationen getestet werden können.

   a) Warum bewegt sich die s_1-Kurve mit steigendem β weiter nach rechts und wird breiter?

   b) Warum liegt die s_2-Kurve größtenteils über der s_1-Kurve?

   c) Erkläre mit Hilfe der vorherigen Antwort, warum der Pfad in diesem Beispiel zielstrebiger auf das Minimum zugeht (insbesondere im Vergleich zum Momentum-Pfad).

   d) Finde eine neue Startposition, in der die s_2-Kurve größtenteils unterhalb der s_1-Kurve verläuft.

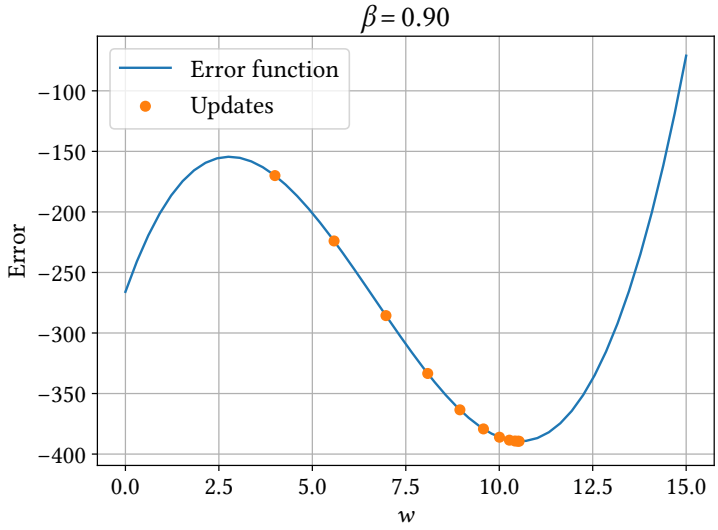

Abbildung 1: Beispiel für die Gewichtsupdates mit *adaptive learning rates* für die Fehlerfunktion aus Gleichung 3.

[5] https://milania.de/blog/Introduction_to_neural_network_optimizers_[part_2]_%E2%80%93_adaptive_learning_rates_(RMSProp%2C_AdaGrad)#fig:AdaptiveLearning_Trajectory

Literatur

[1] John Duchi, Elad Hazan und Yoram Singer. „Adaptive Subgradient Methods for Online Learning and Stochastic Optimization“. In: *Journal of Machine Learning Research* 12 (Juli 2011), S. 2121–2159.

[2] Tijmen Tieleman und Geoffrey Hinton. *Lecture 6a. Overview of mini-batch gradient descent*. 2012. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.