Backpropagation Seminar Zusammenfassung

Lucas Sas Brunschier

SS20

GitHub Repository	github.com/sirbubbls/backpropagation-seminar
Presentation	sirbubbls.github.io/backpropagation-seminar
Video	sirbubbls.github.io/backpropagation-seminar/video.mp4

1 Einführung

1.1 Computational Graphs

In der machine learning community ist es üblicherweise gängig, mathematische Operationen nicht auf mathematischer Art und Weise darzustellen, sondern in der Form eines Graphen.

Bei einem computational Graph werden Rechenoperationen und Variablen durch Nodes repräsentiert. Diese Nodes werden durch Pfeile miteinander verbunden um den Datenfluss zu visualisieren.

1.2 Neurale Netze / Deep Learning

Künstliche neuronale Netze bezeichnet eine Teilemenge des Forschungsbereichs des maschinellen Lernen, als Vorbild dienen neuronale Netze der Biologie, jedoch muss dazu erwähnt werden, dass sich die beiden Forschungsbereiche doch weniger miteinander zu tun haben, wie man eventuell zu glauben vermag.

Ein künstliches neuronales Netz ist essentiell eine Verkettung an Funktionen, in dem jeder Layer eine Funktion f^i mit i als Layer Index repräsentiert. Somit lässt sich ein Netz mit 3 Layern mit $f^2(f^1(f^0(X)))$ mit X als Input Daten ausdrücken. Diese Verkettung an Funktionen ist ein sehr wichtiges Konzept, wodurch der Backpropagation Algorithmus überhaupt erst anwendbar wird.

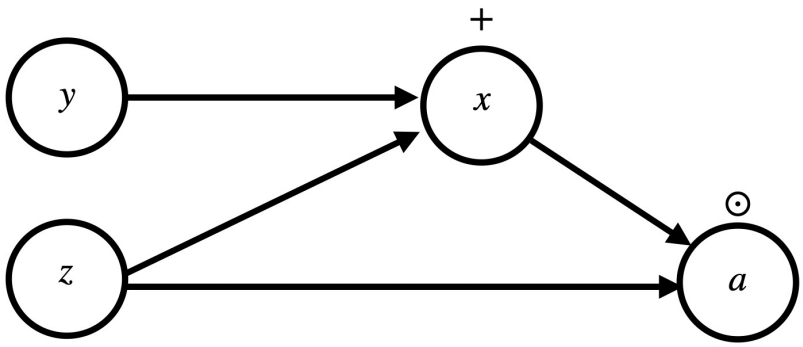

Figure 1: In diesem Beispiel ist gut ersichtlich, wie zwei Funktionen ($x = yz$ und $a = x * z$) durch einen Graphen visualisiert werden können.

2 Backpropagation

2.1 Wozu benötigen wir den Backpropagation Algorithmus?

Der Backpropagation Algorithmus beschäftigt sich mit der Berechnung von Ableitungen in computational Graphs. Meist wird dieser auch fälschlicher Weise als Optimierungsalgorithmus für neurale Netze bezeichnet. Dies trifft jedoch nicht zu, da typischerweise in neuralen Netzwerken der Backpropagation Algorithmus ausschließlich für die Berechnung der Gradients (also Ableitungen) der Weights & Biases der verschiedenen Layer zuständig ist. Diese Gradients werden dann von dem schon bekannten Gradient Descent Algorithmus dazu genutzt um die Cost Function zu minimieren.

2.2 Kettenregel

Die Kettenregel ist ein fundamentaler Bestandteil des Backpropagation Algorithmus, da diese uns ermöglicht Ableitungen von verketteten Funktionen zu bestimmen.

Wie bereits oben schon erwähnt ist ein neuronales Netz eine verkettete Funktion, somit ist die Kettenregel optimal um gewünschte Gradients zu berechnen.

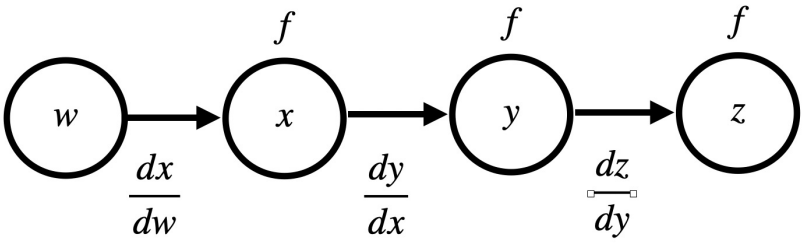

Figure 2: In diesem Beispiel wird eine verkettete Funktion der Form $x = f(w),\ y = f(x),\ z = f(y)$ als Graph dargestellt. Jede Operation leiten wir für sich selber ab und multiplizieren diese, somit erhalten wir die Ableitung $\frac{dz}{dw}$ mit $\frac{dz}{dy}\frac{dy}{dx}\frac{dx}{dw} = f'(f(f(w)))f'(f(w))f'(w)$.

2.3 Forward Propagation

Der Begriff Forward Propagation bezeichnet den Algorithmus, der für die Auswertung eines Datenpunkts durch ein neuronales Netzwerk verwendet wird.

Notation

- σ Aktivierungsfunktion
- L Layer Index
- W Weight Matrix
- a Activations
- b Bias Vektor

Wir wollen die Aktivierungen jeder Neuronen Schicht berechnen, dazu können wir die Formel $a_L = \sigma(W_L a_{L-1} + b_L)$ verwenden. Das Produkt $W_L a_{L-1}$ gibt an wie stark jede Activation des vorherigen Layers durch ein Neuron gewichtet wird. Auf diesen Vektor wird dann ein Bias Wert addiert und die Aktivierungsfunktion auf jedes Element des Vektors ausgeführt.

2.4 Algorithmus

Als Voraussetzung für den Backpropagation Algorithmus müssen wir zuerst den Forward Propagation Algorithmus ausführen und einige Werte jedes Layers speichern:

- den Wert z, bevor die Aktivierungsfunktion ausgeführt wird
- die Activations a eines Layers

Die erste Ableitung die wir berechnen müssen ist die Ableitung der Cost Function $J = \frac{1}{2}(X - y)^2$ also $J' = y - X$ mit X als der vorausgesagte Wert und y als der erwartete Wert. Diesen Wert weisen wir der Hilfsvariable g zu.

Nun führen wir folgende Schritte für jeden Layer k in unserem neuralen Netzwerk aus.

1. Die Cost Function in Abhängigkeit des Wertes vor der Aktivierungsfunktion.

   $\nabla_z J = g * \sigma'(z) = g$ auch diesen Gradient weisen wir der Variable g zu.

2. Berechnung der Weights in Abhängigkeit der Cost Function J.

   $\nabla_w J = a_{L-1} * g$ da $a_{L-1}w_L + b_L$ nach w_L abgeleitet a_{L-1} ergibt.

3. Berechnung des Biases in Abhängigkeit der Cost Function J.

   $\nabla_b J = g$ da $a_{L-1}w_L + b_L$ nach b_L abgeleitet 1 ergibt.

4. Wir können nun die neuen Weights und Gradients mit Hilfe der berechneten Gradients $\nabla_z J$ und $\nabla_w J$ anpassen. Die neuen Weights und Biases werden in einer separaten Datenstruktur gespeichert und erst am Ende des Backpropagation Algorithmus dem eigentlichen Netzwerk zugewiesen.

   > **Anmerkung**
   > Dieser Teil gehört technisch nicht mehr zum Backpropagation Algorithmus, sondern typischerweise zum Gradient Descent Algorithmus.

   $w_k^* = w_k - \lambda \nabla_{w_k} J$
   $b_k^* = b_k - \lambda \nabla_{b_k} J$

5. Nun müssen wir das neue g bestimmen, dass an den Layer $L - 1$ übergeben wird.

   $g = g * w_L$ da wir $a_{L-1}w_L + b_L$ nach a_{L-1} ableiten.

6. Sobald wir den letzten hidden Layer des Netzwerks erreicht haben müssen wir statt a_{L-1} die Input Matrix X nutzen um die Gradients zu errechnen.

7. Nach der Iteration aller Layer müssen nur noch unser Weights und Biases des neuronales Netz aktualisiert werden.

   $w_k = w_k^*$ Zuweisung der während der Iteration gespeicherten Weights w_k^*
   $w_k = b_k^*$ Zuweisung der während der Iteration gespeicherten Weights b_k^*

2.5 General Backpropagation

Der Backpropagation Algorithmus lässt sich nicht ausschließlich nur mit neuralen Netzwerken nutzen, sondern kann auch bei computational Graphen einer beliebigen Form Ableitungen bestimmen.

Formell wird für die generelle Definition des Backpropagation Algorithmus folgende Parameter benötigt:

- die Menge aller Variablen T, deren Gradienten wir berechnen müssen
- den Graphen G
- die Variable z, die wir differenzieren wollen

Zur Vorbereitung bilden wir den Graphen G' aus G dieser enthält alle Variablen, die Vorfahren von z und Nachfahren von T sind. Zudem initialisieren wir die Datenstruktur `grad_table`, in der wir Variablen Gradients zuweisen können. Zu beginn füllen wir diese Datenstruktur mit `grad_table[z] = 1`, da $\frac{\partial z}{\partial z} = 1$.

Wir können den Algorithmus in zwei Teile aufteilen, im äußeren Teil iterieren wir durch alle Elemente in T, und rufen in jedem Loop die rekursive Funktion `build_grad` auf.

```
for v in T:
    build_grad(v, G, G_1, grad_table)
return [grad_table[v] for v in T]
```

Der Base Case der rekursiven Funktion `build_grad` ist erreicht, sobald sich die zu berechnende Variable `v` sich bereits in `grad_table` befindet,

also zu Beginn bei $v = z$ mit `build_grad[v] = 1`. Wir iterieren durch alle direkten Nachfahren von `c` und speichern sowohl die Operation durch die, die Variable erzeugt wurde als auch den Gradient der Variable durch `d = build_grad`. Wir führen die Backpropagation Operation `bprop()` der gespeicherten Operation aus und diese berechnet den Gradient der aktuellen Variable durch den Gradient des Nachfahren `d`, die Variable `v` die wir differenzieren wollen und den tatsächlichen Inputs `get_inputs()`.
Da wir bei mehreren Konsumenten auch mehrere Gradients erhalten müssen wir diese schließlich auch summieren oder den Durchschnitt als Gradient für die zu berechnende Variable v benutzen.

```
def build_grad(v, G, G_1, grad_table):
    if v in grad_table: return grad_table[v]
    i = 1
    for c in get_consumers(v, G_1):
        op = get_operation(c)
        d = build_grad(c, G, G_1, grad_table)
        g[i] = op.bprop(get_inputs(c, G_1), v, d)
        i += 1
    g = sum(g)
    grad_table[v] = g
    return g
```

3 Quellen

- Deep Learning (Ian Goodfellow, Yoshua Bengio & Aaron Courville)
- https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complic
- Wikipedia: https://en.wikipedia.org/wiki/Backpropagation
- Wikipedia: https://en.wikipedia.org/wiki/Delta_rule