Adaptive Systeme - Hausaufgabe 3

Henry Fock

2023-01-30

Ein einfaches Feed Forward Netz

Klassische Feed Forward Netze (FFN) bestehen aus einer Aneinanderreihung von Matrix-Vektor Multiplikationen, gefolgt von einer nicht-linearen Aktivierungsfunktion. Im Training erhält das Netzwerk einen Input x und einen erwarteten Output y. Das Netzwerk hat einen Aufbau von l Schichten, bestehend aus einer Gewichtematrix $W^{(k)}$ mit $k \in \{1, \dots, l\}$, einem Input $h^{(k-1)}$, wobei $h^{(0)} = x$, einem Bias $b^{(k)}$ und einer Aktivierungsfunktion $f : \mathbb{R} \to \mathbb{R}$ besteht. Das Ergebnis pro Schicht wird berechnet aus:

$$a^{(k)} = W^{(k)} h^{(k-1)} + b^{(k)} \tag{1}$$

$$h^{(k)} = f\left(a^{(k)}\right) \tag{2}$$

$$\hat{y} = h^{(l)}$$

Nach Beendigung der letzten Schicht wird der Fehler des Netzwerkes zum erwarteten Ergebnis berechnet. Die dazu verwendete Fehlerfunktion wir mit $\mathcal{L}\left(\hat{y}, y\right)$ bezeichnet.

Um das Netz zu trainieren, müssen die Gradienten der Funktionen mit Bezug auf den Fehler berechnet werden. Formel 1 benötigt eine mit Bezug auf die Gewichte, um diese zu ändern und eine mit Bezug auf den Eingang, um den Gradienten in die darüberliegenden Schichten weiterzuleiten. Die Gradienten lassen sich über die Kettenregel berechnen:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W^{(l)}} &= \frac{\partial a^{(l)}}{\partial W^{(l)}} \cdot \frac{\partial h^{(l)}}{\partial a^{(l)}} \cdot \frac{\partial \mathcal{L}}{\partial h^{(l)}} \\
\delta^{l} &= \frac{\partial h^{(l)}}{\partial a^{(l)}} \cdot \frac{\partial \mathcal{L}}{\partial h^{(l)}} \\
\frac{\partial \mathcal{L}}{\partial W^{(l-1)}} &= \frac{\partial a^{(l-1)}}{\partial W^{(l-1)}} \cdot \frac{\partial h^{(l-1)}}{\partial a^{(l-1)}} \cdot \frac{\partial a^{(l)}}{\partial h^{(l-1)}} \cdot \delta^{l} \\
\delta^{l-1} &= \frac{\partial h^{(l-1)}}{\partial a^{(l-1)}} \cdot \frac{\partial a^{(l)}}{\partial h^{(l-1)}} \cdot \delta^{l}
\end{aligned} \tag{3}$$

$\delta^{(k)}$ bezeichnet hier die Gradienten der vorherigen Schichten, wenn das Netz rückwärts durchlaufen wird. Für den Bias ist die Berechnung des Gradienten analog zu den anderen Gewichten. Als "Input" wird hier 1 verwendet.

Beim so genannten "mini batch gradient descent" werden bspw. 16 Datensätze durch das Netz geschickt, zu jedem der Fehler und die dazugehörigen Gradienten berechnet, aber erst nachdem der letzte Gradient berechnet wurde, werden die Gewichte geupdatet. Um ein gemeinsames update zu machen, werden die Gradienten für die Gewichtupdates aufsummiert und je nach implementierung normiert. Folgend ist ein Ausschnitt einer Implementierung für den Rückwärtsschritt in einer voll vernetzten Schicht:

```
# ...
def backwards(self, prev_grads: np.ndarray) -> np.ndarray:
    self.weight_gradients = np.add(
```

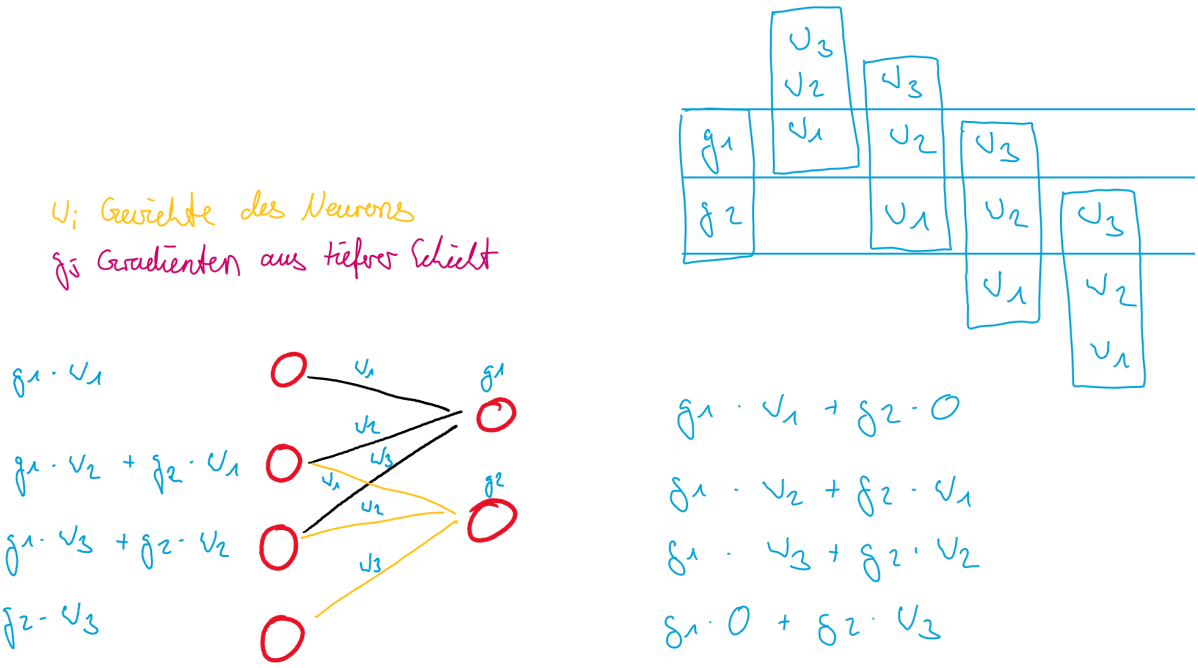
```
        self.weight_gradients, np.dot(prev_grads, self.train_input.T))
    self.num_samples += 1
    return np.dot(self.weights[:,1:].T, prev_grads)

def train(self, lr: float = 0.01) -> None:
    if self.num_samples > 0:
        self.weights -= lr * (self.weight_gradients / self.num_samples)
```

Convolutional layer

Eine Convolutional (deutsch: Faltungs-) Schicht, besitzt die Eigenschaft nicht voll vernetzt zu sein. Es werden immer nur ein paar Inputs zu einem neuen Wert zusammengefasst, im Gegensatz zur Vollvernetzten Schicht, bei der alle Inputs zu einem Ergebnis geformt werden. Bei einer Faltung in einem Bild wird bspw. ein Fenster von 3×3 über das Gesammte Bild gezogen und die Faltung berechnet. Das Ergebnis ist ein neues Bild, bei dem jedes Pixel das Ergebnis aus der Faltung im durch das Fenster ist. Rechnerisch ist es so, als würde man ein Neuron mit 9 Eingängen und einem Ausgang über das Bild schieben. Die einzelnen Fenster können als individuelle Datensätze aufgefasst werden, wodurch dies das Selbe Prinzip wie beim "mini batch" ist. Um Also die Gradienten für die Gewichte zu bilden, müssen lediglich die Gradienten der Gewichte mit jedem Fenster als Input gebildet und aufsummiert werden.

Schwieriger ist das Weiterleiten der Gradienten in die nächst höhere Schicht.



(a) Gleichungen für die manuelle bildung der Ableitungens

(b) Erzeugen der Gradienten über Faltungsopperation

Figure 1: Bilden der Gradienten mit Hilfe der Faltung über die Gewichte

Abbildung 1a zeigt, wie die Gradienten für die nächste Schicht gebildet werden, nämlich der Vorherige Gradient mal dem Gewicht, mit dem der Input das Ergebnis des Outputs beeinflusst hat. Dabei ist zu beachten, dass die schwarzen und gelben Verbindungen die selben Gewichte besitzen (In blau beschriftet)! Diese Berechnung lässt sich mit Hilfe einer Faltung widerspiegeln. Dafür werden die Gewichte einmal um

180° im Filter gedreht(!) und eine "Volle-Faltung" auf die Gradienten angewendet. An Stellen, wo der Filter keinen Gradienten zum Matchen hat, wird dieser mit 0 Multipliziert. Der Ablauf ist in Abbildung 1b dargestellt. Wie man sehen kann, ist die Berechnung exakt die selbe, wie in Abbildung 1a.

Max- Average Pooling

Pooling-Operationen fassen ein Eingabefenster zu einer Ausgabe zusammen, ohne Verwendung von Gewichten. Der Gradient beschreibt, welchen Einfluss ein Parameter auf das Ergebnis hatte. Bei einem Max-Pooling, wird lediglich der höchste Wert im Fenster weitergegeben, weshalb auch nur dieser einen Einfluss auf das Ergebnis hat. Damit gilt, wenn x_i der höchste Wert ist:

$$\nabla \max(x_1, x_2, \ldots, x_n) = \delta_{i,k}, \text{mit} i, k \in \{1, \ldots, n\} \tag{4}$$

Beim Max-Pooling ist der Gradient 1 für den höchsten Wert im Fenster, 0 für alle anderen. In Formel 4 wird dies durch das Kronecker-Delta beschrieben. Beim Average-Pooling gilt:

$$\nabla \frac{1}{n} \cdot (x_1 + x_2 + \ldots + x_n) = \frac{1}{n} \tag{5}$$

Beim Average-Pooling werden die Gradienten wieder mit Hilfe der Faltung berechnet. Das Vorgehen ist das Selbe, wie in der Faltungsschicht, nur werden als Gewichte $\frac{1}{n}$ verwendet, wobei n die Anzahl der Felder im Filter sind.

Aktivierungs-/Transferfunktionen

Transferfunktionen sind nicht lineare Funktionen $f : \mathbb{R} \to \mathbb{R}$ (Verwendung in Formel 2). f wird auf jedes Element im Input angewendet. Um die Gradienten zu bestimmen muss also die Ableitung von f bestimmt werden und auf alle Inputs angewendet werden. Die meist verwendeten Transferfunktionen sind die Rectified Linear Unit (ReLU) und die Sigmoid-/Fermi-Funktion:

$$\text{ReLU}(x) = \begin{cases} x & if x > 0 \\ 0 & else \end{cases} \tag{6}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

$$\tag{8}$$

Die Ableitungen sind wie folgt:

$$\frac{d\text{ReLU}(x)}{dx} = \begin{cases} 1 & if x > 0 \\ 0 & else \end{cases} \tag{9}$$

$$\frac{d\sigma(x)}{dx} = \sigma(x) \cdot (1 - \sigma(x)) \tag{10}$$

Berechnung der Gradienten in LeNet

Das LeNet hat eine simple Architektur, bestehend aus zwei 2D-Faltungsschichten, denen jeweils eine Sigmoid-Aktivierung und ein Average-Pooling folgt. Das Ergebnis wird in einen eindimensionalen Vektor transformiert und durch 3 voll vernetzte Schichten geleitet, jede gefolgt von einer Sigmoid-Aktivierung.

Abbildung 2 zeigt den Aufbau der Architektur.

Folgend sollen die Berechnungsschritte für den Gradient-Descent Algorithmus aufgeführt werden. Gegeben sei der Gradient des berechneten Outputs mit Bezug auf den Fehler $\frac{\partial \mathcal{L}}{\partial \bar{y}}$. Zuerst müssen die Gradienten durch

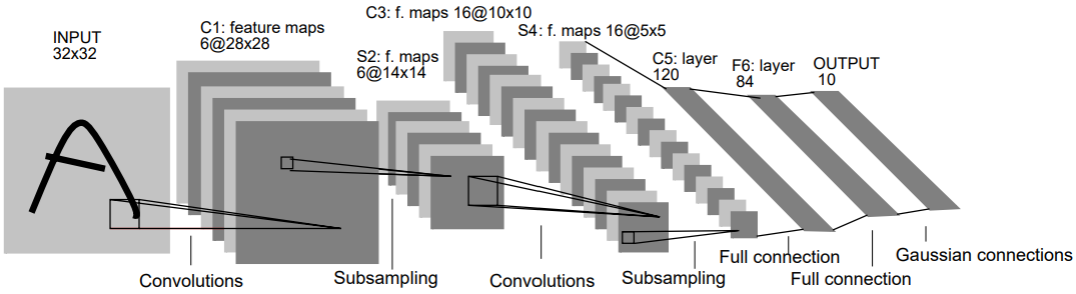


Figure 2: Architeckur des LeNets (Entnommen aus LeNet Paper)

die voll vernetzten Schichten getragen werden. Das Vorgehen dazu ist bereits in den Formeln 3 gegeben. Im Flattening-Schritt (vgl. S4 - S5 in Abbildung 2) werden die berechneten Gradienten der Knoten in die zugehörigen Zellen in der Matrix eingetragen. Als nächstes muss der Gradient durch die Average-Pooling Layer geleitet werden. Im einfachen Fall geschieht dies, wie in Abschnitt beschrieben. Im LeNet läuft das Fenster für das Pooling nicht nur einen Schritt, sondern die Fenster sind so gewählt, dass diese sich nicht überlappen. Daher muss die Matrix der Gradienten (Jaccobi-Matrix) erst vergrößert werden, hier um das doppelte, da das Pooling die Layer zuvor halbiert hat.

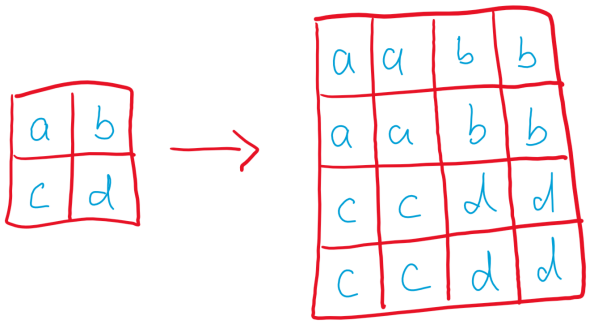
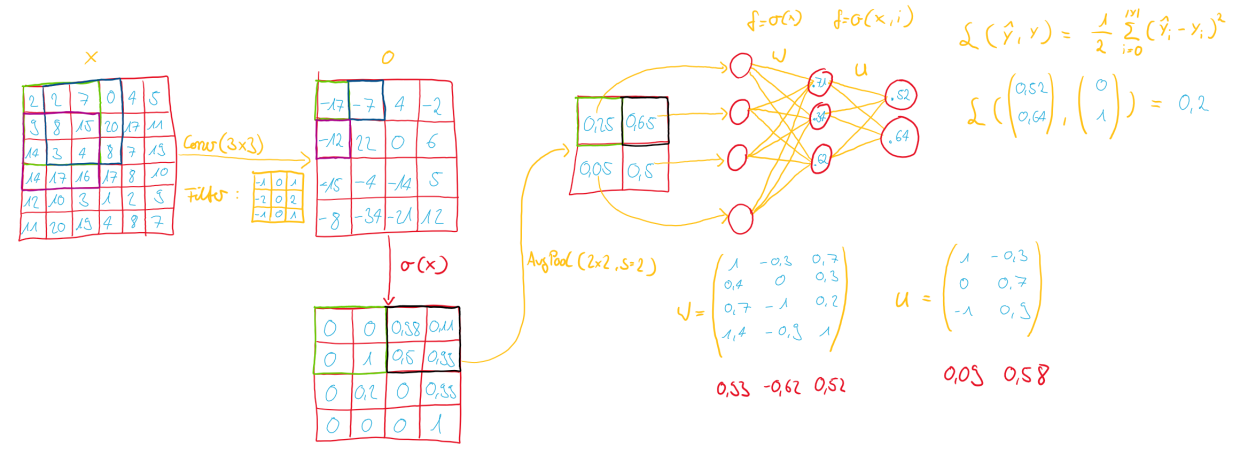


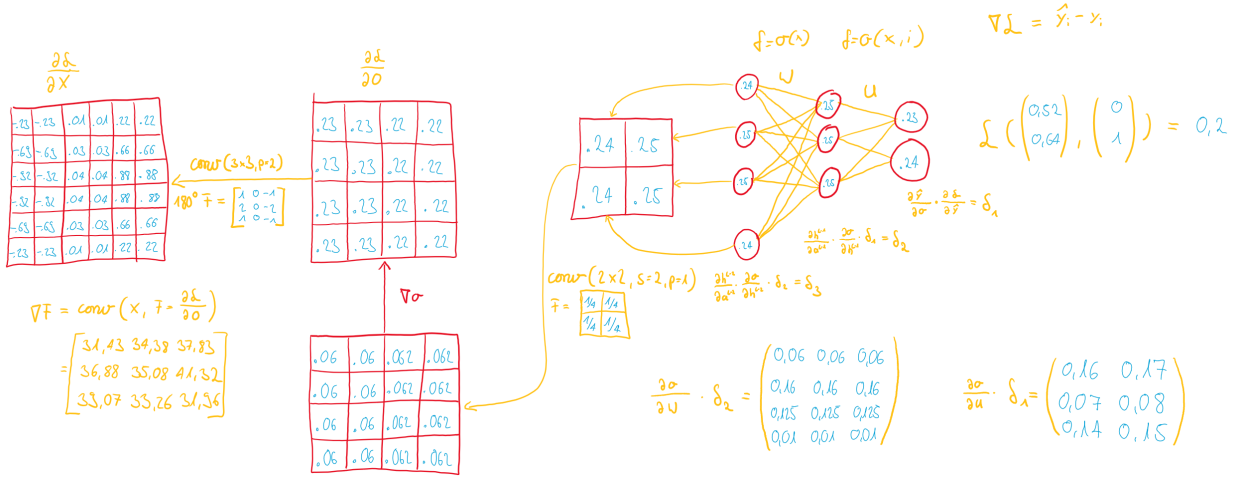
Figure 3: Vergrößern der Matrix

In Abbildung 3 wird das Vergrößern illustriert. Anschließend kann eine Faltung mit den selben Eigenschaften, wie das Pooling und einem Filter entsprechend $\frac{1}{n}$ an jeder Stelle, durchgeführt werden, um die Gradienten durch das Pooling zu erhalten.

Als nächstes geht jeder einzelne Gradient durch die Ableitung der Transferfunktion. Die Bildmatrix bleibt dadurch in allen Dimensionen erhalten. Als nächstes folgen die Gradienten durch die Faltung. Hier muss einmal der Gradient von $\mathcal{L}$ mit Bezug auf den Filter und einmal mit Bezug auf den Input h berechnet werden. Für beides kann die Faltungsoperation verwendet werden. Um den Gradienten des Filters ($\frac{\partial \mathcal{L}}{\partial F}$) zu berechnen, berechnet man die Faltung vom Input, mit der soeben berechneten Jaccobi-Matrix als Filter. Das Ergebnis entspricht exakt den Dimensionen des verwendeten Filters. Dies entspricht genau dem aufsummieren der einzelnen Gradienten für jedes Fenster, wie in Abschnitt beschrieben. Der Gradient für die nächste Schicht wird wie in Abschnitt berechnet. D.h. eine “volle Faltung” (Padding von 2 Feldern um die Matrix) mit einem um 180° gedrehtem Filter. Dieses verfahren wiederholt sich für die die nächste Gruppe von Pooling, Aktivierung und Faltung. In Abbildung 4 wird für einen simplifizierten Aufbau des LeNets der Vorwärts und Rückwärtsschritt berechnet, der die Erklärung exemplarisch unterstützen soll. Die GRadienten wurden schnell ähnlich, da hier stark gerundet wurde und die Änderungen im hinteren Kommabereich erst bemerkbar sind.



(a) Der Vorwärtsschritt in einem simplifizierten LeNet



(b) Durchgerechnetes Backpropegation für ein simplifiziertes LeNet

Figure 4: Berechnung des Vorwärts- und Rückwärtsschrittes eines simplifizierten LeNets