Institut für Analysis und Numerik
Universität Magdeburg
Thomas Richter
Gozel Judakova

03.07.2020

thomas.richter@ovgu.de
gozel.judakova@ovgu.de

Übung Nr. 12 zur Vorlesung Algorithmische Mathematik II
Sommersemester 2020

Abgabe bis zum 10. Juli 2020

Deep neural networks Wir haben bisher neuronale Netze der folgenden Art betrachtet:

- Wir stellen eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ dar.
- D.h., wir haben n Eingaben und m Ausgaben.
- Das neuronale Netz besteht also aus einer Schicht mit m künstlichen Neuronen, jedes hat n Eingaben, einen Bias und jeweils eine Ausgabe.

Der Übergang zu *deep neural networks*, also *Tiefen neuronalen Netzen* folgt einer einfachen Idee

- Zur Darstellung einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ wählen wir ein Netzwerk mit L Schichten $N_1, \ldots, N_L$
- Jede schicht stellt eine Funktion $f_i : \mathbb{R}^{i_l} \to \mathbb{R}^{o_l}$ dar.
- Dabei gilt $i_1 = n$, da f_1 die Eingabeschicht ist und $o_L = m$ da f_L die Ausgabeschicht ist. Ansonsten gilt $i_l = o_{l-1}$, d.h. die Ausgabe von Schicht $l-1$ ist die Eingabe von Schicht l
- Das resultierende Netzwerk mit L Schichten hat dann

  $$N_{kn} = \sum_{l=1}^{L} o_l$$

  künstliche Neuronen sowie

  $$N_{Wb} = \sum_{l=1}^{L} o_l \cdot (i_l + 1)$$

  Gewichte und Biase.

Hiermit ist das Wesentliche zusammengefasst und im Folgenden wird es vor allem darum gehen, wie wir solche komplexeren neuronalen Netzwerke mathematisch und auch in Python effizient darstellen.
Man lese nach *Satz 8.24* (*Gradient des einschichtigen neuronalen Netzes*) bis zu Algorithmus *8.9 Auswerten eines feedforward Netzes*. Berechnung des Netzes sowie der Gradienten sind sehr technisch und es ist nicht notwendig, die praktische Umsetzung genau nachzuvollziehen. Diese abschießende Woche ist es sinnvoller mit dem (ab Sonntag) bereit gestellten Programm zu experimentieren. Man gebe kurze Antworten:

1. Wir stellen mit neuronalen Netzen eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ dar. Dennoch betrachten wir die eingabe nicht als Vektor $x \in \mathbb{R}^n$ sondern als Matrix $X \in \mathbb{R}^{N \times n}$ und geben uns auch in Python mühe, Modell f und Zielfunktional J, sowie den Gradienten direkt für die ganze Matrix X auszuwerten. Was ist diese Matrix und warum formulieren wir alles für die komplette Matrix auf einmal?

2. Was ist die Frobeniusnorm?

3. Was ist das dyadische Produkt zweier Vektoren $x, y \in \mathbb{R}^n$?

4. Wieviele freie Parameter hat das in Abbildung 8.9 dargestellte Netzwerk?

Backpropagation Die Methode zur Berechnung des Gradienten dieser Art von Netzen wird *Backpropagation* genannt. Man überfliege den Abschnitt *Berechnung des Gradienten* und mache sich klar, dass dies nur eine mehrfache Anwendung der Kettenregel ist. Es ist nicht wichtig, die exakte Umsetzung zu verstehen, da diese sehr technisch ist und eigentlich nur ein richtiges Umgehen mit den vielen Indizes verlangt.

Trainieren und Testen Man lese das abschließende Beispiel *Numerisches Beispiel - Trainieren des Netzes* und gebe kurze Antworten

1. Wie kann es sein, dass das erste neuronale Netzwerk 99% der Daten korrekt einordnet, aber dennoch keinen Kreis darstellt?

2. Warum ist es wesentlich, ein neuronales Netzwerk mit Daten zu testen, die nicht zum Trainieren verwendet wurden?

Programmieraufgabe 12.1:

Auf der Homepage stehen 2 Programme zum Testen, `ein—schicht—netz.py` zur Darstellung von neuronalen Netzen mit einer Schicht sowie `deep—neural—network.py` zur Darstellung von neuronalen Netzen mit mehreren Ebenen. Versuchen Sie (freiwillig) sich mit den Programmen vertraut zu machen.
Einige Anhaltspunkte zum Experimentieren mit `ein—schicht—netz.py`

- Versuchen Sie mit dem Programm die folgende Aufgabe umzusetzen:
  1. Als Eingabe wählen wir zufällige 2×2 Matrizen, diese stellen wir aber einfach als Vektoren der Länge 4 da, also `n = 4` sowie `X = 0.25*np.random.randn(n,N)` im Programm.
  2. Das Modell soll eintscheiden, ob die Matrix "fast" singulär ist, was wir anhand von $|\det(A)| < 0.01$ entscheiden wollen, in Python und da die Matrix als Vektor gespeichert ist `if np.abs(X[0,j]*X[3,j]—X[1,j]*X[2,j])<0.01`.
  3. Versuchen Sie, ob Sie ein entsprechendes Netzwerk trainieren können.
  4. Testen Sie nun das Netzwerk. D.h. sie Erstellen einfach weitere Eingabedaten, etwa

     ```
     # Testdaten
     NT= 1000

     XT = 0.25*np.random.randn(n,NT)  # zufaellig
     YT = np.zeros( (m,NT) )
     # Y mit richtigem Ergebnis fuellen
     ```

     und werten das trainierte Netz dann auf diesen Daten aus. Erreichen Sie Generalisierbarkeit?

Einige Anhaltspunkte zum Experimentieren mit `deep—neural—network.py`

- Das Trainieren dauert oft sehr lange. Manchmal wird bei den eingestellten Parametern auch kein Ergebnis erreicht. D.h., das trainierte Netzwerk verhält sich manchmal nicht besser als das zufällige untrainierte Netzwerk
- Probieren Sie andere Netzwerk-Architekturen, z.B. wie in Abbildung 8.14 im Skript dargestellt oder sogar nocht einfachere, z.B. $2 \to 3 \to 1$.
- Es scheint klar zu sein, dass das Netzwerk nur dann gut generalisierbar ist (d.h. auch auf weitere Daten anwendbar, was man z.B. an der Darstellung des Netzes als Funktion ablesen kann), wenn genug Testdaten vorhanden sind und diese den Kreis auch gut abdecken. Versuchen Sie die Anzahl der Daten N sowie den Ursprung des Kreises zu variieren um bessere Ergebnisse zu erhalten.
- Die Funktion `feedforward_gradient(...)` berechnet auch die Ausgabe jedes einzelnen Neurons. Um die Funktionsweise des Netwerks besser zu verstehen können

wir auch jedes Neuron für sich als Funktion plotten. Fügen Sie den folgenden Code ganz am Ende des Programms ein um alle Neuronen einzeln darzustellen.

```
### Wir geben nun die einzelnen Neuronen aus
[FF,Fx,Fy] = feedforward_gradient(W,b,XX)

for l in range(L):
    #Fx[l]=(0.5+Fx[l]).astype(int)   # evtl um auf 0/1 zu runden
    for j in range(Fx[l].shape[0]):
        print('Neuron {} von {} in Ebene {}'.format(j,Fx[l].shape
            ↪ [0],l))
        ax.add_patch(plt.Circle((KX, KY), KR, color='r', alpha=0.1))
        plt.title('Neuron {} von {} in Ebene {}'.format(j,Fx[l].
            ↪ shape[0],l))
        plt.scatter(XX[0,:],XX[1,:],c=Fx[l][j,:])
        plt.show()
```

- Sie können versuchen, das Problem der "fast singulären Matrizen" mit einem komplexeren Netzwerk umzusetzen.