Aufgabenblatt 6

Praktikum Computer Vision
SoSe 2018

Christian Wilms

17. Mai 2018

Aufgabe 1 — Ein Neuron selber bauen

Im Rahmen dieser Aufgabe sollt ihr ein einzelnes Neuron bauen, trainieren und zur Klassifikation von Bildern an Hand der Merkmale Mittelwert und Standardabweichung nutzen. Das Neuron soll dabei eine lineare Entscheidungsgrenze haben, Aktivierungsfunktionen sind daher nicht nötig. Das Neuron liefert daher als Ausgabe $y = \vec{w}^T\vec{x} + b$, basierend auf den Eingaben x_1 und x_2 für Mittelwert und Standardabweichung eines Bildes sowie den Gewichten w_1, w_2 und b.

1. Um das Training und die Klassifikation gut und sinnvoll durchführen zu können, werden viele Beispielbilder benötigt. Um nicht große Datenmengen laden zu müssen, nutzt das Skript `bilderGenerator.py` aus dem CommSy. Dieses Skript beinhaltet die Funktion `zieheBilder(anzBilder)`, welche ein gewünschte Anzahl an Bildern zweier Klassen erzeugt und gleich deren Merkmale (Mittelwert und Standardabweichung) sowie das Label zurückliefert. Lasst euch mit zwei Aufrufen von `zieheBilder(anzBilder)` so 500 Trainingsbilder und 50 Validierungsbilder generieren.
2. Nach der Generierung der Daten sollt ihr nun die Trainingsdaten erst einmal darstellen. Dazu reicht eine Ebene auf deren x-Achse die Mittelwerte und auf deren y-Achse die Standardabweichungen aufgetragen werden (oder umgekehrt). Erzeugt dafür zunächst einen einzelnen, leeren Plot:

```
plt.close('all')
fig, ax = plt.subplots(1, 1)
```

   Nun könnt ihr mit der Methode `plot(xCoords, yCoords, marking)` am Objekt `ax` einzelne Punkte oder ganze Listen von Punkten plotten. `xCoords` bzw. `yCoords` können dabei einzelne Werte für die x- bzw. y-Koordinate sein oder Listen von x- bzw. y-Koordinaten zum Plotten mehrerer Punkte gleichzeitig. `marking` ist ein String aus zwei Komponenten und legt die Art der Darstellung der zu plottenden Punkte fest. Bspw. können mit `'rx'` rote (`r`) Kreuze (`x`) geplottet werden.
   Achtet beim Plotten der Trainingsdaten darauf, dass diese gemischt sind. Filtert also erst die Koordinaten der *+1-Klasse* heraus und danach die der *-1-Klasse*, bspw. über (np.where).

3. Iteriert nun über die erzeugten Trainingsdaten und wendet auf jedes Datum (Merkmale eines Bildes) die lineare Klassifikation

$$y = f(\vec{x}) = \vec{w}^T \vec{x} + b = w_1 \cdot x_1 + w_2 \cdot x_2 + b \tag{1}$$

mit $w_1 = 0.0001$, $w_2 = -0.0002$ und $b = 0.001$ an.
Macht das Gleiche nun mit den Validierungsdaten, merkt euch die Vorzeichen als Vorhersage und vergleicht die Vorzeichen der Vorhersagen mit den tatsächlichen Labels der Validierungsdaten. Referenzergebnisse siehe `spoiler.txt` im CommSy.

4. Nun soll die Berechnung der partiellen Ableitungen erfolgen. Dazu muss bei jedem Durchlauf eines Trainingsdatums ermittelt werden, ob die Klassifikation korrekt war, also das Vorzeichen von y mit dem jeweiligen Trainingslabel t übereinstimmte ($\text{sgn}(t) \neq \text{sgn}(y)$). Ist die Klassifikation nicht korrekt, so müssen wir die alten w_1, w_2 und b anpassen. Berechnet dazu zunächst die partiellen Ableitungen (s. Folie 19) für $\frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_2}$ und $\frac{\partial L}{\partial b}$. Aktualisiert nun w_1, w_2 und b mit Hilfe der partiellen Ableitungen und einer Learning Rate α von 0.0000005 nach jedem Durchlauf eines Datums. Trainiert so euer Neuron einmal über den gesamten Trainingsdaten. Verbessert sich das Ergebnis auf den Validierungsdaten, wenn ihr zur Klassifikation dieser nun eure trainierten Werte für w_1, w_2 und b nutzt? Referenzergebnisse siehe `spoiler.txt` im CommSy.

5. w_1, w_2 und b sollen jetzt anders initialisiert werden. b's werden im Allgemeinen mit 0 initialisiert, fügt dies entsprechend in euren Code ein. Für w_1 und w_2 werden Zufallszahlen entsprechend einer Normalverteilung gezogen. Nutzt dazu die Funktion `np.random.normal()` und setzt den Mittelwert der Verteilung auf 0 und die Standardabweichung auf 0.001. Lasst die gesamte Klassifikation nun mehrfach komplett durchlaufen (mit neuer Initialisierung von w_1, w_2 und b), was beobachtet ihr?

6. Neuronale Netze werden meist in mehreren Epochen trainiert, d.h. die gesamten Trainingsdaten werden mehrfach durch das Netz geschickt und die Parameter zwischendurch nicht neu initialisiert. Implementiert dies für euer Neuron und lasst es 100 Epochen lang trainieren. Wie verändern sich die Ergebnisse nun? Lasst ggf. beide Varianten mehrfach durchlaufen.

7. Visualisiert nun eure Entscheidungsgrenze, indem ihr in kleinen Schritten (Schrittweite 0.1) über den gesamten Wertebereich $0 \ldots 255 \times 0 \ldots 128$ iteriert und die jeweiligen Werte in euren gelernte lineare Klassifikation einsetzt. Ist das Ergebnis nun nahe 0 ($y < 0.00001$), plottet ihr für diese Koordinate einen Punkt. Es sollte so eine Gerade entstehen, die beide Klassen einigermaßen gut voneinander trennt.

8. **Zusatzaufgabe**: Die Daten sollten vor Beginn der Klassifikation immer normiert werden, d.h. sie sollten im Bereich $-0.5 \ldots 0.5$ liegen. Dies hat den Vorteil, dass sich Parameter leichter zwischen Modellen und Anwendungen transferieren lassen. Implementiert dies, indem ihr die Trainings- und Validierungsdaten durch das jeweilige Maximum des Merkmals als Float teilt und danach den Mittelwert der Trainingsdaten je Merkmal als abzieht. Die Daten sollten dann im Bereich $-0.5 \ldots 0.5$. Nun müsst ihr die Learning Rate anpassen. Wie und warum?

9. **Zusatzaufgabe**: Da wir nur drei Parameter haben, können wir auch mit einer umfassenden Suche (Grid Search) nach der optimalen Parameterkombination für w_1, w_2 und b suchen. Probiert dazu im nicht normierten Fall (ohne die vorherige Zusatzaufgabe) für die drei Parameter alle möglichen Kombinationen der Werte $-2, -1.9, \ldots, 1.9, 2$ aus. Ist dies besser als das Training? Wie ist die Laufzeitkomplexität dieser Suche $\mathcal{O}(...)$.

Aufgabe 2 — Neuronales Netz als Klassifikator

Im Rahmen dieser Aufgabe sollen erste Erfahrungen mit Keras gesammelt werden. Dazu bedienen wir uns wieder der Daten aus Aufgabenblatt 3.2 (Hirsche, Schiffe, Autos in Farbbildern). Allerdings wollen wir jetzt den Klassifikator von einem Nächster-Nachbar-Klassifikator auf ein Neuronales Netz umstellen.

1. Ladet zunächst euren Code von Aufgabe 3.2.2. Falls ihr diesen nicht mehr finden könnt, nehmt die Musterlösung aus dem CommSy (Woche 3). Ändert den Code nun so, dass zunächst erstmal die Merkmale Mittelwert und Standardabweichung für alle Trainings- bzw. alle Validierungsbilder berechnet werden und in zwei Arrays der Shape `(60,6)` bzw. `(30,6)` geschrieben werden. Ändert außerdem den Datentyp der Arrays auf `np.float32`.
2. Um mit Keras zu arbeiten, importiert die folgenden Module:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
from keras.optimizers import SGD
```

   Setzt außerdem noch vor diesen Import-Anweisungen den Zufallsgenerator auf den Wert 123, um für eine Reproduzierbarkeit der Ergebnisse zu sorgen.

```
import numpy as np
np.random.seed(123)  # um die Gewichte immer gleich
    zufaellig zu initialisieren
from tensorflow import set_random_seed
set_random_seed(123) # um die Gewichte immer gleich
    zufaellig zu initialisieren
```

3. Nun müsst ihr die Labels, die aktuell $1, 4$ und 8 sind, auf $0, 1, 2$ umkodieren. Danach könnt ihr `Y_train` bzw. `Y_test` (für die Validierunsgdaten) wie in den Folien beschrieben erzeugen.
4. Baut nun ein Modell wie in den Folien auf, das aus zwei Dense-Layern mit je 8 Neuronen und einer ReLU-Aktivierungsfunktion besteht gefolgt von einem Dense-Layer mit 3 Neuronen (Anzahl der Klassen) und einer Softmax-Aktivierungsfunktion. Die Softmax-Aktivierungsfunktion sorgt hier dafür, dass die Ausgabe der drei letzten Neuronen als Wahrscheinlichkeitsverteilung interpretiert werden können (Summe ist gleich 1). Denkt daran, dass ihr die `input_shape` im ersten Layer setzen müsst.

5. Kompiliert das Modell nun mit den auf den Folien gezeigt den Parametern.
6. Berechnet die Ergebnisse mit Hilfe der `fit()` Methode des Models und einer Batch-Size von 1 sowie 500 Epochen.
7. Evaluiert das Modell auf den Validierungsdaten. Sind die Ergebnisse besser als mit dem Nächster-Nachbar-Klassifikator?
8. **Zusatzaufgabe**: Führt in einer Schleife das Experiment mehrfach durch. Was passiert und warum passiert dies?

Aufgabe 3 — Zusatzaufgabe: CIFAR-10 Datensatz

1. Klassifiziert die Bilder aus dem gesamten CIFAR-10 Datensatz mit Hilfe des Mittelwerts und der Standardabweichung und einem Nächster-Nachbar-Klassifikator. Die Bilder und Labels könnt ihr direkt mit Keras herunterladen.

```
from keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

   Als Nächster-Nachbar-Klassifikator bietet sich die Implementation aus der Bibliothek scikit-learn an. Referenzergebnisse findet ihr in der `spoiler.txt` im CommSy.

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=1)
XTr = np.array(trDeskriptors).reshape((60,-1)) #bei 60
    Trainingsbildern
yTr = trLabels
XVal = np.array(vaDeskriptors).reshape((30,-1)) #bei 30
    Validierungsbildern
clf.fit(XTr, yTr)
predictions = clf.predict(XVal)
```

2. Führt die Klassifikation des gesamten CIFAR-10 Datensatzes mit Mittelwert und Standardabweichung durch, allerdings mit einem Neuronalen Netz als Klassifikator. Das Neuronale Netz sollte 3 Layer mit jeweils 128 Neuronen haben. Als Batch Size solltet ihr 32 verwenden, als Anzahl Epochen 10 und als Learning Rate 0.01. Die allgemeine Struktur ist gleich zu dem Netz aus Aufgabe 2. Normiert die Mittelwerte und Standardabweichungen unbedingt vor der eigentlichen Klassifikation, indem ihr diese durch 255.0 bzw. 128.0 teilt.
3. Nutzt als Deskriptor nun das abgerollte Bilder selbst. Ihr könnt dazu direkt die Bilder als Eingabe in einen Flatten-Layer als ersten Layer übergeben. Die restliche Architektur des Netzes sowie der weiteren Parameter bleiben gleich. Normiert die Bilder ebenso wieder, indem ihr sie durch 255.0 teilt.