Woche 10 Programmierung: Neuronale Netzwerke

Skript

Erarbeitet von
Marc Feger

Lernziele 1
Inhalt 2
Einstieg 2
Der MNIST Datensatz 2
Aufbau von Graustufenbildern 3
Datennormalisierung 3
Aufbau eines neuronalen Netzes 3
Ausblick 5
Weiterführendes Material 6
Disclaimer 6

Lernziele

- Den MNIST-Datensatz beschreiben und laden
- Normalisierung von Bildern durchführen
- Erklären, wie mit dem Befehl models.Sequential Layers zum neuronalen Netz hinzugefügt werden
- Den Unterschied zwischen ReLU und softmax erklären
- Anzahl der Modellparameter berechnen

Projekt „KI für alle"
HeiCAD – Heine Center for Artificial Intelligence and Data Science

Inhalt

Einstieg

Hallo und herzlich willkommen. In diesem Tutorial ist zu sehen, wie ein neuronales Netz programmiert wird.

Der MNIST Datensatz

Als Grundlage für unser neuronales Netz verwenden wir den sogenannten MNIST-Datensatz. MNIST ist ein Datensatz aus 70.000 Bildern. Diese Bilder zeigen auf sogenannten Graustufenbildern handgeschriebene Zahlen von 0 bis 9. Die Größe der jeweiligen Bilder beträgt 28 x 28 Pixel. Um die jeweiligen Trainings- und Testdaten zu laden, kannst du den folgenden Befehl verwenden.

```
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
```

Du kannst sehen, dass hier die jeweiligen Trainings- und Testdaten geladen worden sind. X steht in diesem Fall dafür, dass die Bilder geladen worden sind. Y steht für die entsprechenden Labels. Um nun den Aufbau der Daten genauer zu betrachten, kannst du den sogenannten *shape*-Befehl verwenden. Auch hier kannst du den folgenden Befehl verwenden.

```
print("Trainingsdaten: ", x_train.shape)
```

Du siehst, dass die Trainingsdaten 60.000 Einträge umfassen. Jeder der Einträge steht für ein Bild der Größe 28 x 28. Weiterhin kannst du dir auch den Aufbau der Label anschauen. Dafür kannst du diesen Befehl einfach kopieren und *x_train* durch *y_train* ersetzen.

```
print("Trainingsdaten: ", y_train.shape)
```

Auch hier siehst du, dass für jeden Eintrag in *y_train* ein Bild vorhanden ist. Das kannst du daran erkennen, dass jeweils 60.000 Bilder und entsprechend 60.000 Label vorhanden sind. Das gleiche kannst du genauso gut für die Testdaten wiederholen.

```
print("Testdaten: ", x_test.shape)
print("Testdaten: ", y_test.shape)
```

Auch für die Testdaten kannst du sehen, dass hier nicht 60.000, sondern 10.000 Bilder vorhanden sind. Gleich bleibt aber der Aufbau von 28 x 28 Pixeln pro Bild.

Aufbau von Graustufenbildern

Da du nun den generellen Aufbau der Daten kennst, lohnt es sich auch, ein Beispiel genauer zu betrachten. Dafür kannst du den folgenden Befehl verwenden.

```
example = show_example(x_train, y_train, 0)
```

Wie du siehst, wird hier die Zahl 5 dargestellt. Besonders helle Werte dieses Bildes werden nahe der Zahl 255 dargestellt, dunkle Werte nahe der Zahl 0.

Datennormalisierung

Eine gängige Methode, um die Daten verständlicher für das neuronale Netz zu gestalten, ist die sogenannte Datennormalisierung. Datennormalisierung meint, dass Werte in den Zahlenbereichen 0 bis 1 abgebildet werden. Um dies zu erreichen, reicht es in unserem Beispiel, dass wir durch den maximalen Farbwert, in unserem Fall 255, teilen.

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

x_train, x_test = x_train / 255.0, x_test / 255.0

Um nachzuvollziehen, wie die Daten nach dem Normalisieren aufgebaut sind, kannst du dir wieder unser Beispiel anschauen.

```
example = show_example(x_train, y_train, 0)
```

Wie du sehen kannst, sind nun nicht mehr die Zahlen 0 bis 255 abgebildet, sondern die Zahlen 0 bis 1.

Aufbau eines neuronalen Netzes

Kommen wir nun zur tatsächlichen Implementierung eines neuronalen Netzes. Grundsätzlich gilt, dass ein neuronales Netz aus drei Teilen besteht. Am Anfang steht die sogenannte Eingabeschicht. Diese ist dafür verantwortlich, Daten entgegenzunehmen und in eine Repräsentation zu überführen, sodass die darauffolgenden versteckten Schichten mathematische Operationen darauf ausführen können. Am Ende eines neuronalen Netzes steht die sogenannte Ausgabeschicht. Diese Ausgabeschicht trifft eine Aussage darüber, was in den Daten enthalten ist. Übertragen auf unseren Datensatz, den MNIST-Datensatz, heißt das, dass das neuronale Netz eine Aussage darüber trifft, welche Zahl in einem Eingabebild enthalten ist. Kommen wir nun zur tatsächlichen Implementierung.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
```


CC BY

```
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

Hier kannst du sehen, wie das Modell aufgebaut ist. Direkt fällt auf, dass das Modell sequentiell arbeitet. Das kannst du dir so vorstellen, als würdest du ein Lego-Set aufbauen und die Steine in eine Reihenfolge bringen. Das heißt, sequentiell bedeutet, dass die einzelnen Schichten des neuronalen Netzes in der Reihenfolge, wie sie angegeben sind, zusammenstecken. Schauen wir uns einmal das neuronale Netz im Detail an.

Zu Beginn sehen wir eine sogenannte Flatten-Schicht. Diese Flatten-Schicht ist die Eingabeschicht unseres Netzes. Die Flatten-Schicht ist dafür zuständig, unsere Bilddaten, 28 x 28 Pixel groß, in eine flachere Repräsentation zu überführen. Was das genau heißt, betrachten wir später in der Zusammenfassung des Modells. Die zweite Schicht ist eine Dense-Schicht, bestehend aus 128 Neuronen, welche die Aktivierungsfunktion *relu* verwenden. Diese Schicht kann mitunter auch als versteckte Schicht bezeichnet werden. Am Ende des neuronalen Netzes steht wieder eine Dense-Schicht mit zehn Neuronen. Wir verwenden hier zehn Neuronen, da in unseren Daten die Zahlen von 0 bis 9, also zehn mögliche Antworten enthalten sind. Weiterhin wird die Aktivierungsfunktion *softmax* verwendet, damit die Aussagen des neuronalen Netzes in eine Wahrscheinlichkeit umgewandelt werden können. Das hat den Vorteil, dass am Ende die Aussage innerhalb von Wahrscheinlichkeiten interpretiert werden kann und so nicht die rohe Ausgabe des neuronalen Netzes für sich alleine steht.

Schauen wir uns einmal die Zusammenfassung dieses Modells an. Dafür kannst du die Methode *summary* verwenden.

```
model.summary()
```

Insgesamt kannst du in dieser Darstellung sehen, dass wir hier ein sequenzielles Modell aus drei Schichten haben. Weiterhin kannst du die einzelnen Schichten im Detail betrachten: Flatten, Dense und Dense. Ebenso kannst du sehen, wie die Ausgaben der jeweiligen Schichten ausfallen. Du weißt ja, als Eingabe für das neuronale Netz werden Bilder der Größe 28 x 28 verwendet. Flatten ist dafür verantwortlich, Bilder in eine flachere Repräsentation zu überführen. D. h., aus den 28 x 28 Pixel großen Bildern wird nun eine 1 x 784 große Darstellung gemacht. Das kommt daher, dass du dir das vorstellen kannst wie einen Aktenvernichter, in den wir unser Bild rein schieben. Dieser Aktenvernichter zerstört unser Bild in einzelne Streifen, welche jeweils ein Pixel in der Breite und 28 Pixel in der Länge abbilden. Flatten macht genau das, nur baut es am Ende diese zerstörten Bilder wieder zusammen, eben in einer Form 1 x 784. 784 steht hier für 28 * 28. D. h. wir zerlegen das Bild in 28 gleich große Streifen der Größe 28 Pixel. Weiter kannst du sehen, dass Flatten keine Parameter hat. Das liegt daran, dass Flatten eben nur dafür zuständig ist, unser Bild in eine flachere Repräsentation zu überführen.

CC BY

Kommen wir nun zur versteckten Schicht. Wie du sehen kannst, besteht diese aus 128 Neuronen. Diese Neuronen sind dafür zuständig, die Eingabe der Flatten-Schicht entgegenzunehmen und an die darauffolgende Dense-Schicht weiterzugeben. Auch kannst du sehen, dass hier 100.480 Parameter zustande kommen. Diese Zahl kommt zustande, weil wir hier 128 Neuronen haben, welche jeweils 784 Pixel der Flatten-Schicht entgegennehmen. D. h. hier entstehen 128 * 784 Eingaben. Hinzuzählen müssen wir aber auch die Neuronen selbst, daher haben wir hier n Neuronen * m Eingaben + n Neuronen.

Kommen wir nun zur Ausgabeschicht. Diese Ausgabeschicht besteht aus zehn Neuronen. Jedes Neuron dieser Schicht wiederum ist mit 128 Neuronen der vorherigen Schicht verbunden, d. h. jedes einzelne Neuron der Ausgabeschicht hat 128 Eingaben. Betrachten wir wieder die Anzahl der Parameter, können wir hier wieder unsere Rechenvorschrift anwenden. Wir sehen also, 10 * 128 + die 10 einzelnen Neuronen ergibt die gesamte Anzahl an zu trainierenden Parametern.

Ausblick

Ich habe dir gezeigt, wie der MNIST-Datensatz aufgebaut ist. Wir haben uns auch ein Beispiel für ein Bild einer handgeschriebenen Zahl angesehen. Wenn du dich erinnerst, das Beispiel zeigte die Zahl 5. Ich habe dir auch gezeigt, wie man ein neuronales Netz programmieren kann, d. h. wie es sequentiell aufgebaut ist. Was ich dir nicht gezeigt habe ist, wie das neuronale Netz trainiert wird. D. h. das neuronale Netz, was wir programmiert haben, weiß noch gar nicht, wofür es überhaupt eingesetzt wird. Es hat noch nichts gelernt. Um dir zu verdeutlichen, dass unser neuronales Netz eigentlich noch gar nichts kann, zeige ich dir folgenden Code als Ausblick. Um unser Beispiel in eine Ausgabe des neuronalen Netzes zu überführen, können wir dieses an das neuronale Netz übergeben.

Wir übergeben das Beispiel *example* an unser Modell. Dieses ist in der Lage, wie angegeben, das Beispiel zu zerlegen und in eine Ausgabe zu überführen. Numpy gibt in diesem Fall an, dass unsere Ausgabe einfach als numpy-Array zurückgegeben wird. Schauen wir uns einmal an, was in den *predictions* tatsächlich drinsteht.

```
print("Vorhersagen: ", predictions)
```

Wie du siehst, sind hier zehn Ausgabewerte. Diese Ausgabewerte stehen jeweils für ein mögliches Label. An erster Stelle steht die Wahrscheinlichkeit, dass das neuronale Netz meint, dass in unserem Beispiel die Zahl 0 abgebildet ist. Der letzte Eintrag steht für die 9. Da *softmax* als Aktivierungsfunktion in der letzten Schicht verwendet wird, sind diese Zahlen als Wahrscheinlichkeiten interpretierbar. Um also herauszufinden, was unser neuronales Netz denkt, dass die richtige Antwort für unser Beispiel sei, müssen wir uns nur auf die Suche nach der höchsten Wahrscheinlichkeit begeben. Mit einem Blick sehen wir, dass die vierte Zahl gemeint sein muss, d. h. hier an Indexposition 5 finden wir die höchste Wahrscheinlichkeit. Dass es sich hierbei um Wahrscheinlichkeiten handelt, können wir

CC BY

ebenfalls herausfinden, indem wir alle möglichen Aussagen des neuronalen Netzes aufsummieren.

```
print("Summe Vorhersagen: ", predictions.sum())
```

Wir sehen, dass diese Summe sehr nahe an der 1 ist. Wenn du dich richtig erinnerst, wurde in unserem Beispiel die Zahl 5 gezeigt.

```
print("Beispiel echtes Label: ", y_train[0])
```

Um jetzt eine konkrete Antwort von unserem neuronalen Netz zu erzeugen, können wir die sogenannte *argmax*-Funktion verwenden. Da die Ausgabe bereits innerhalb Wahrscheinlichkeiten interpretierbar ist, können wir diese anwenden, um die größte Zahl aus unseren Vorhersagen herauszulesen.

```
print("NN geschätzte Label: ", np.argmax(predictions))
```

Argmax sucht nach dem größten Wert in der übergebenen Liste und gibt den jeweiligen Index zurück. Da unsere *predictions* von 0 bis 9 reichen, wird hier die 4 zurückgegeben, also der fünfte Eintrag, den wir auch als größtes Element identifiziert haben. Was wir hier sehen ist, dass das neuronale Netz wirklich keine Ahnung davon hat, was es eigentlich tun soll. Es macht nur wilde Vorhersagen. Wie das neuronale Netz dann tatsächlich trainiert und optimiert werden kann, siehst du im kommenden Tutorial. Bis dahin.

Weiterführendes Material

https://www.tensorflow.org/tutorials/keras/classification

Disclaimer

Transkript zu dem Video „Woche 10 Programmierung: Neuronale Netzwerke“, Marc Feger. Dieses Transkript wurde im Rahmen des Projekts ai4all des Heine Center for Artificial Intelligence and Data Science (HeiCAD) an der Heinrich-Heine-Universität Düsseldorf unter der Creative Commons Lizenz CC-BY 4.0 veröffentlicht. Ausgenommen von der Lizenz sind die verwendeten Logos, alle in den Quellen ausgewiesenen Fremdmaterialien sowie alle als Quellen gekennzeichneten Elemente.