Übung 06
Kennenlernen von **Keras** in **Python** mit **MNIST**-Fashion
INFI-IS
5xHWII

January 30, 2023

Abgabetermin: lt. mündlicher Vereinbarung
Übungsleiter: Albert Greinöcker

Ziel der Übung: In der Aufgabenstellung sind weitere Beschreibungen zum Paket `Keras` gegeben. Darauf basierend soll versucht werden, die Vorgangsweise des gemeinsamen Skriptes zu MNIST zu verstehen und das Modell zu verbessern.

1 Laden, Aufbereitung und Veranschaulichung der Datens

Analog zum gemeinsamen Beispiel mit dem MNIST-Datensatz mit den hangeschriebenen Zahlen gibt es einen Datensatz mit Modeartikeln (Erstellt von Zalando). Dieser kann folgendermaßen geladen werden:

```
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

Eine grundsätzliche Beschreibung des Datensatzes befindet sich unter: `https://github.com/zalandoresearch/fashion-mnist`
Die Kategorien sind in folgende Zahlen kodiert:

0. T-shirt/Top
1. Hose
2. Pullover
3. Kleid
4. Mantel
5. Sandalen
6. Hemd
7. Sneaker
8. Tasche
9. Halbschuhe

1.1 Überblick über die Daten verschaffen

1.1.1 Die Beschaffenheit (Dimensionen und Häufigkeiten) der Daten abfragen)

1.1.2 Wie viele Kleidungsstücke sind pro Kategorie verfügbar?

1.1.3 Wie bekommt man die Pixel des zehnten Bildes, wie die Info, welches Kleidungsstück hier dargestellt werden soll?

1.2 Visualisierung der Daten

1.2.1 Bitte die ersten 100 Bilder aus den Daten generieren, damit man einen Eindruck bekommt wie sie gespeichert sind

Dieses Beispiel sollte helfen:

```
for i in range(0,100):
  im = Image.fromarray(train_images[i])
  real = train_labels[i]
  im.save("/home/albert/tmp/mnist/%d_%d.jpeg" % (i, real))
```

1.2.2 Bilder der gleichen Kategorie exportieren

Um eine bessere Vergleichbarkeit zu erhalten wäre ein Export sinnvoll, wo Bilder der gleichen Kategorie in einem Verzeichnis liegen. Hinweis: Das Paket `os` kann hier beim Erzeugen von Verzeichnissen helfen. Eine Beschreibung gibt es hier: `https://www.geeksforgeeks.org/create-a-directory-in-python/`

2 Aufbauen des Models

2.1 Erzeugen der Modellstruktur

In unserem Beispiel wurde nur ein hidden layer verwendet, was nicht optimal ist. In diesem Beispiel soll ein zweiter Layer verwendet werden. Zu Beginn soll der erste Layer 128 Units haben, der zweite 24.
Hier ein Beispiel eines Modells:

```
model = keras.Sequential(
  [
    keras.Input(shape=(784,)),
    layers.Dense(16, activation="relu"),
    layers.Dense(num_classes, activation="softmax"),
  ]
)
```

1. `layer_dense`: Eine Ebene wo alle Knoten mit allen Vorgängerknoten verbunden sind
2. `units`: Wie viele Neuronen hat die Ebene im Neuronalen Netz
3. `activation`: Die Aktivierungsfunktion, also wie die summierten und gewichteten Eingangswerte an den Ausgang weitergegeben werden. folgende wichtige Funktionen gibt es:
   (a) **relu**: `f(x) = max(0,x)`: <0 wird zu 0, >0 wird einfach weitergegeben
   (b) **sigmoid**: "Quetscht" beliebige Werte in ein Intervall von 0-1
   (c) **softmax**: Gibt Wahrscheinlichkeiten bei den entsprechenden Ausgängen aus

(d) Eine Liste weiterer Aktivierungsfunktionen gibt es unter: `https://keras.io/api/layers/activations/`

Einen Überblick über das erzeugte Model bekommt man mit `model.summary()`

2.2 Compilieren des Modells

Beim Compilieren des Modells werden Metriken Eigenschaften über das Lernen festgelegt:

```
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

1. **loss**: Verlustfunktion, die die Unterschiede zwischen Vorhersagen und tatsächlichem Wert bestimmt und einen Verlustscore vergibt. Mögliche Werte:
   (a) **categorical_crossentropy**: Bei Kategorien am Ausgang (wie in unserem Fall)
   (b) **binary_crossentropy**: Bei 2 Kategorien
2. **optimizer**: Welcher Backpropagation-Algorithmus wird angewandt. Nimmt basierend auf dem Verlustscore eine Aktualisierung der Gewichte vor. Mögliche Einstellungen:
   (a) **adam**: Stochastischer Gradientenabstieg. Sucht schrittweise das Minumum
   (b) **rmsprop**: Bewegt sich schneller auf das Minimum zu mit der Gefahr es zu überschreiten
3. **metrics**: Der Wert, der die Klassifikations-Qualität wiedergibt: Am Besten `'accuracy'` verwenden, das ist die Korrektklassifikationsrate.

3 Beobachtung des Lernprozesses

Das Lernen wird mit folgendem Code durchgeführt:

```
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

1. **history**: Beinhaltet die Ergebnisse des Lernens
2. **epochs**: Wie oft soll der komplette Datensatz durchlaufen werden zum Lernen
3. **batch_size**: Nach wie vielen Durchläufen sollen die Gewichte upgedated werden
4. **validation_split**: Welcher Anteil wird für die Validierung verwendet

Wie verhält sich der Lernprozess über die Epochen? Wann könnte/sollte man eigentlich abbrechen?

3.1 Evaluation der Ergebnisse

Natürlich soll das Modell anhand der Test- und nicht anhand der Trainingsdaten evaluiert werden:

```
score = model.evaluate(x_test, y_test, verbose=2)
```

3.2 Analyse der Ergebnisse

Diese Anweisung führt die Klassifikation durch:

```
pred = model.predict(x_test)
print(pred[1]) #Welche Wahrscheinlichkeiten werden fuer die einzelnen Labels berechnet
print(y_labels[1]) #Das Label dazu
```

3.3 Optimierung

Es kann an vielen Parametern optimiert werden, wir beschränken uns vorerst auf diese:

- Anzahl der Neuronen pro Ebene
- Anzahl der Ebenen

Der Wettbewerb wer das beste Ergebnis erreicht ist eröffnet!
Das fertige Modell soll gespeichert werden. Hier die Befehle zum Laden und Speichern eines Modells:

4 Weiterführende Analysen

Im 2. Beispiel auf github sind ein paar Ansätze, wie man sich die Ergebnisse der Vorhersage (predict) veranschaulichen kann. Bitte diese anwenden und ggf. erweitern.