

Neuronales Netzwerk – MNIST 1

Grundlage für die Übung bildet das MNIST-Dataset. Hierbei handelt es sich um 70.000 Bilder handgeschriebener Ziffern; wobei 60.000 Beispiele im Trainings- und 10.000 Beispiele im Testdatensatz enthalten sind.


 1.1 Laden Sie das MNIST-Dataset aus der Keras-Installation¹. Details liefert die referenzierte Dokumentation. Vor allem der Bereich „Returns“ ist interessant. Verschaffen Sie sich einen Überblick (Stichwort *shape* des Datasets u. eines Images oder einfache Ausgabe des Inhaltes).

Abbildung 1 zeigt eine beliebige Ziffer Pixel für Pixel. Die *shape* einer Ziffer beträgt (28,28) – ein Bild ist also 28 Pixel breit als auch hoch. Die möglichen Graustufenwerte bewegen sich zw. 0 und 255.

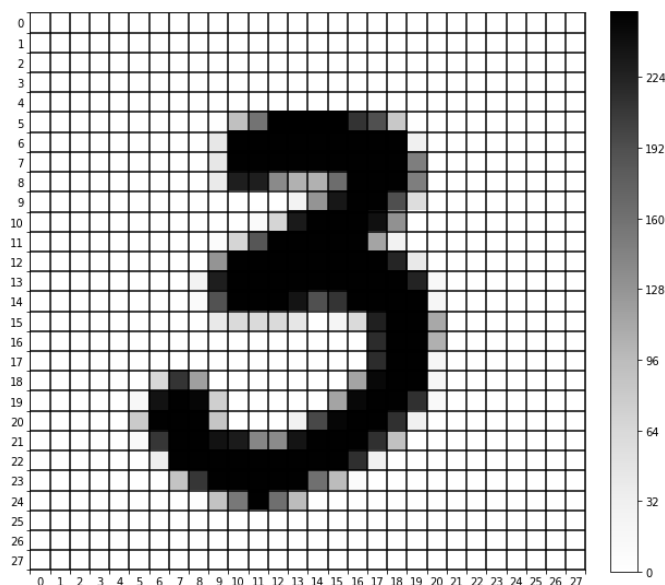



Abbildung 1: MNIST Ziffer Pixel by Pixel

 1.2 Erstellen Sie basierend auf `X_train` (Trainingsbilder) ein `pd.DataFrame`. Beachten Sie hierbei, dass das *DataFrame* Daten in einer bestimmten Form erwartet. Der gesuchte Output gestaltet sich wie folgt:


¹ <https://keras.io/api/datasets/mnist/>

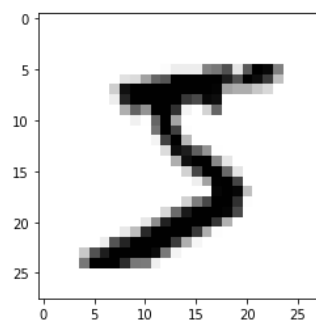
	0	1	2	3	4	5	6	7	8	9	...	774	775	776	777	778	779	780	781	782	783
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 784 columns

Wir halten fest: Jede Zeile entspricht genau einem Bild, das 28x28 (=784 Pixel) umfasst.

Hinweis: Dieser Arbeitsschritt hat für den weiteren Übungsverlauf keine Bedeutung und ist lediglich als verständnisfördernde Maßnahme (von `pd.DataFrame`) zu verstehen.

 1.3 Geben Sie mit `matplotlib.pyplot.imshow2` die Ziffer an Position 0 des Numpy-Arrays `x_train` aus. Gesuchtes Ergebnis (mit `cmap="gray_r"`):



Sehen Sie bei den korrespondierenden Antworten nach, ob das tatsächlich die Ziffer „5“ ist.

Nachdem Klarheit bzgl. der Daten herrscht, gilt es im nächsten Schritt das Modell zu definieren. Es gibt mehrere Möglichkeiten. Im Beispielsfall erstellen wir ein sog. *Sequential Model* mit zwei Ebenen (vgl. Theorieeinheit):

```
1: model = Sequential()
2: model.add(Dense(64, activation="relu", input_shape=(784,)))
3: model.add(Dense(10, activation="softmax"))

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


Mit der *Sequential*-API kann man ohne großen Aufwand einfache Architekturen erstellen. Komplexere Architekturen sind mit *Sequential* nicht möglich. Wie dem Code-Beispiel zu entnehmen ist, kann man Ebene für Ebene das Modell definieren.

² https://matplotlib.org/3.1.1/api/as_gen/matplotlib.pyplot.imshow.html

In Zeile 1 erfolgt die Instanziierung der für das *Sequential Model* erforderlichen Klasse `Sequential`³. Mit deren Hilfe ist es möglich, mehrere Ebenen zu einem linearen Stapel von Ebenen zu gruppieren. Der Hidden-Layer (vgl. Theoriebeispiel) wird in Zeile 2 erstellt und dem Modell hinzugefügt. Für diesen sehen wir 64 Knoten (Neuronen) vor. Nachdem es sich um den ersten Layer handelt, ist festzulegen, in welchem Format die Daten – also der Tensor – zu erwarten sind. Hierzu dient das Argument `input_shape`, das einen *Tupel* als Wert erwartet. Als Aktivierungsfunktion ist beim *Hidden Layer* „relu“ zu wählen.


Wichtig: Beim ersten Layer eines Modells ist immer die Shape der zu erwartenden Daten zu definieren – aber nur beim ersten!

Der Output-Layer wird mit Zeile 3 definiert.

 1.4 Überlegen Sie, warum es genau 10 Knoten (=Ausgänge) braucht.

Als Aktivierungsfunktion kommt *Softmax* bei der Ausgabeschicht zum Einsatz. Mithilfe der Softmax erhalten wir Werte, die die Wahrscheinlichkeit der jeweiligen Kategorie (Ziffern 0 – 9) widerspiegeln.

Abschließend muss das Modell kompiliert werden (siehe Zeile 4). Hierbei ist mit dem Argument `optimizer` anzugeben, nach welchem Verfahren wir die Gewichte trainieren möchten. Die von der Verlustfunktion berechnete Größe gilt es während des Trainings zu **minimieren**. Wie das geschehen soll, legt das Argument `loss` fest. Im Beispielsfall kommt als Kostenfunktion `categorical_crossentropy` zum Einsatz.

 1.5 Implementieren Sie das Modell und kompilieren Sie dieses gemäß Vorgaben.

Nach erfolgreicher Kompilierung kann das Modell trainiert werden. Hierzu stellt das *Sequential-API* die Methode `fit` bereit. Mitzugeben sind die Trainingsbeispiel und deren Antworten (Labels). Ebenso ist die Angabe, wie oft der Trainingsdatensatz durchlaufen werden soll⁴, in Form des Argumentes `epochs` festzulegen. 128 bei `batch_size` bedeutet, dass nach 128 Beispielen die Gewichte aktualisiert werden. Der `batch_size`-Wert sollte eine Zweierpotenz sein.

```
model.fit(x_train, y_train, epochs=5, batch_size=128)
```

Output:

```
Epoch 1/5
60000/60000 [=====] - 5s 86us/step - loss: 5.4959
- accuracy: 0.9023
Epoch 2/5
60000/60000 [=====] - 4s 74us/step - loss: 0.6809
- accuracy: 0.9515
Epoch 3/5
60000/60000 [=====] - 4s 73us/step - loss: 0.4874
- accuracy: 0.9646
Epoch 4/5
```

³ <https://keras.io/api/models/sequential/>

⁴ Es sei denn, `steps_per_epochs` ist auf etwas anderes als „none“ gesetzt.

```
60000/60000 [=====] - 4s 73us/step - loss: 0.4131
- accuracy: 0.9709
Epoch 5/5
60000/60000 [=====] - 4s 73us/step - loss: 0.3327
- accuracy: 0.9760
```

Nochmals zur Erinnerung: Die Trainingsbilder weisen eine Shape von (28, 28) auf. Das Modell benötigt die Daten in (784,0), was sich aber mit *reshape* problemlos anpassen lässt.

Die Aufbereitung der Trainings-Antworten ist etwas komplizierter. Um diese zu verstehen, ist die Betrachtung ausgehend von einem konkreten Klassifizierungsergebnisses überaus hilfreich.

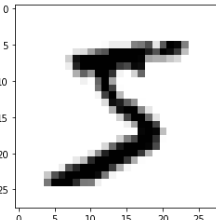
Man bedenke, dass die Klassifizierung von Bildern mit Ziffern von 0 bis 9 das erklärte Ziel ist. Es ist also eine dementsprechende Aussage vom Modell zu erwarten. Jetzt ist es aber nicht so, dass der Output-Layer „Das ist die Ziffer 5“ ausgibt. Was wir erhalten, sind Wahrscheinlichkeiten, und zwar für jede Ziffer einen Wert (siehe unten, Output). In Summe ergeben die Werte 1. Was bedeutet das jetzt für die Aufbereitung der Trainings-Beispiele?

```
y_pred = model.predict(x_test)
y_pred[0]
```

Output:

```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

Angenommen, das aktuelle Trainingsbild enthält eine 5 (links), dann ist diese Information (rechts) dem Modell beim Trainieren bereitzustellen.

<p>X_train[0]</p> 	<p>Y_train[0]</p> <p>5</p>
---	----------------------------

Das geschieht, indem wir die Antworten *one-hot* kodieren. Die Umsetzung gestaltet sich wie folgt: Ausgangsbasis bildet ein 10 Stellen umfassender Vektor – für jede Ziffer eine Stelle.


```
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
```


Um jetzt die, für das oben angeführte Beispiel mit der Ziffer 5, *one-hot* kodierte Antwort zu erhalten, ist an der Stelle 5 des Vektors eine „1“ zu setzen.

```
[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]
```

Allgemeine formuliert: Es ist an jener Stelle die „1“ zu setzen, die dem Wert der Ziffer entspricht.

Die Umsetzung kann mittels `to_categorical`⁵ aus dem Keras-Paket erfolgen.

 1.6 Trainieren Sie das Modell mit den Trainingsdaten (`x_train`, `y_train`). Das setzt voraus, dass die Shape sowohl bei den Beispielen als auch den Antworten den Anforderungen des Modells entsprechen.

 1.7 Die Validierung des Modells kann mittels `evaluate` erfolgen. Hierbei sind dann logischerweise `x_test` und `y_test` zu verwenden. Auch in diesem Fall muss die Shape bei den Testbildern sowie die One-hot-Kodierung bei den Antworten korrekt sein.

```
model.evaluate(x_test, y_test)
```

Output:

```
10000/10000 [=====] - 0s 50us/step  
[0.7909989478800276, 0.9611999988555908]
```

⁵ https://keras.io/api/utils/python_utils/#to_categorical-function