Vergleich von verschiedenen KI-Klassifizierungsmethoden am Beispiel des Fashion MNIST-Datensatzes in RStudio und Python (Teil 1)

Lars Mehnen

2022-05-07

In dieser Reihe werden wir verschiedene maschinelle und Deep-Learning-Methoden vergleichen, um Kleidungsklassen anhand von Bildern des Fashion MNIST-Datensatzes zu erstellen bzw. mit dem erstellten Modellen zu klassifizieren. In diesem ersten Beitrag werden wir Daten untersuchen und für die Analyse aufbereiten. Um Ihnen zu zeigen, wie Sie eine der Funktionen von RStudios verwenden, um Python von RStudio aus auszuführen, erstelle ich ein neuronales Netzwerk in Python innerhalb der R-Umgebung. Die Ideen und codes wurden teilweise von dieser Seite entnommen

Zu Beginn setzen wir zunächst einen fixen Random-Seed, um sicherzustellen, dass die Ergebnisse reproduzierbar sind.

```
set.seed(815)
```

Importieren und Exploration der Daten

Das `keras`-Paket enthält die Fashion MNIST-Daten, sodass wir die Daten aus diesem Paket direkt nach der Installation problemlos in RStudio importieren können.

```
#library(devtools)
#devtools::install_github("rstudio/keras")
library(keras)
#install_keras()
fashion_mnist <- keras::dataset_fashion_mnist()
```

Das resultierende Objekt `fashion_mnist` ist eine Liste, die aus den Unter-Listen `train` und `test`, wobei diese wiederum aus den `x` und `y` Arrays besteht. Um die Größen/Dimensionen der Bilder zu erfahren, wenden wir die Funktion `dim()` rekursiv auf die `fashion_mnist` Liste an.

```
rapply(fashion_mnist, dim)
```

```
## train.x1 train.x2 train.x3  train.y  test.x1  test.x2  test.x3
test.y
##    60000       28       28    60000    10000       28       28
10000
```

Hier sehen wir, dass das `x`-Array im Trainingsdatensatz 28 Matrizen mit jeweils 60000 Zeilen und 28 Spalten enthält, oder mit anderen Worten 60000 Bilder mit jeweils 28 mal 28 Bildpunkten. Das `y`-Array im Trainingsdatensatz enthält 60000 Beschriftungen (auch

Label genannt) für jedes der Bilder im x-Array der Trainingsdaten. Die Testdaten haben eine ähnliche Struktur, enthalten aber nur 10000 Bilder statt 60000. Der Einfachheit halber benennen wir diese Listenelemente in etwas intuitiveres um (wobei x jetzt Bilder und y Beschriftungen bzw. Label darstellen):

```
c(train.images, train.labels) %<-% fashion_mnist$train
c(test.images, test.labels) %<-% fashion_mnist$test
```

Jedes Bild wird in einer 28x28-Matrix erfasst, wobei der [i, j] Eintrag die helligkeit des Pixels auf einer Skala von 0 (weiß) bis 255 (schwarz) darstellt. Die Labels bestehen aus ganzen Zahlen zwischen null und neun, die jeweils eine ein-eindeutige Kleidungs-klasse darstellen. Da die Kategorienamen nicht in den Daten selbst enthalten sind, müssen wir sie manuell hinzufügen. Beachten Sie, dass die Kategorien in den Daten gleich-verteilt angenommen werden.

```
cloth_cats = data.frame(category = c('Top', 'Trouser', 'Pullover',
'Dress', 'Coat',
                                     'Sandal', 'Shirt', 'Sneaker',
'Bag', 'Boot'),
                        label = seq(0, 9))
```

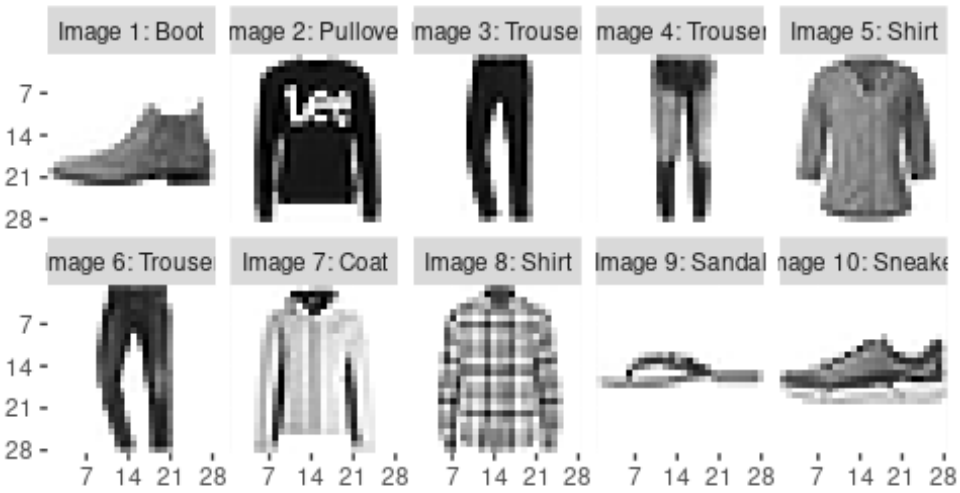
Um eine Vorstellung davon zu bekommen, was die Daten so alles beinhalten und wie sie aussehen, stellen wir die ersten zehn Bilder der Testdaten als Beispiel dar. Dazu müssen wir die Daten zunächst leicht umformen, damit sie zu `ggplot2` kompatibel werden. Wir wählen die ersten zehn Testbilder aus, konvertieren sie in data.frame's um, benennen die Spalten in die Ziffern 1 bis 28 um, erstellen eine Variable namens y mit den Ziffern 1 bis 28 und führen dann das ganze wieder nach der Variable y zusammen. Wir brauchen das Paket `reshape2`, um auf die Funktion `melt()` zugreifen zu können. Dies ergibt eine Datenmatrix (data.frame) von 28 mal 28 gleich 784 mal 3 (y Pixel (= y), x Pixel (= variable) und die Helligkeit (= value)). Wir führen das alles zeilenweise mit der Funktion `rbind.fill()` aus dem `plyr`-Paket zusammen und fügen eine Variable `Image` hinzu, bei der es sich um eine eindeutige Zeichenfolge handelt, die 784 Mal für jedes der neun Bilder wiederholt wird, die die Bildnummer und das entsprechende Testset-Label enthält.

```
library(reshape2)
library(plyr)
subarray <- apply(test.images[1:10, , ], 1, as.data.frame)
subarray <- lapply(subarray, function(df){
  colnames(df) <- seq_len(ncol(df))
  df['y'] <- seq_len(nrow(df))
  df <- melt(df, id = 'y')
  return(df)
})
plotdf <- rbind.fill(subarray)
first_ten_labels <- cloth_cats$category[match(test.labels[1:10],
cloth_cats$label)]
first_ten_categories <- paste0('Image ', 1:10, ': ', first_ten_labels)
plotdf['Image'] <- factor(rep(first_ten_categories,
```

```
unlist(lapply(subarray, nrow))),
                            levels = unique(first_ten_categories))
```

Dann stellen wir diese ersten zehn Testbilder mit dem Paket `ggplot2dar`. Beachten Sie, dass wir die y-Achse umkehren, da der ursprüngliche Datensatz, die Bilder auf dem Kopf stehend enthält. Wir entfernen desweiteren die Legenden- und Achsenbeschriftungen und ändern etwas die Beschriftungen.

```
library(ggplot2)

ggplot() +
  geom_raster(data = plotdf, aes(x = variable, y = y, fill = value)) +

  facet_wrap(~ Image, nrow = 2, ncol = 5) +
  scale_fill_gradient(low = "white", high = "black", na.value = NA) +
  theme(aspect.ratio = 1, legend.position = "none") +
  labs(x = NULL, y = NULL) +
  scale_x_discrete(breaks = seq(0, 28, 7), expand = c(0, 0)) +
  scale_y_reverse(breaks = seq(0, 28, 7), expand = c(0, 0))
```



Daten Vorverarbeitung

Jetzt wird es Zeit für die eher technischen Arbeit, d.h. die Bildklassifizierung (die Label aus den Bilddaten zu schätzen). Zuerst müssen wir unsere Daten wiedermalumformen, um sie von einem mehrdimensionalen Array in eine zweidimensionale Matrix umzuwandeln. Dazu vektorisieren wir jede 28 x 28 Matrix in eine Spalte der Länge 784, und dann verbinden wir die Spalten für alle Bilder übereinander und nehmen schließlich die Transponierte der

resultierenden Matrix. Auf diese Weise können wir ein 28 x 28 x 60000-Array in eine 60000 x 784-Matrix umwandeln. Wir normalisieren die Daten auchnoch, indem wir durch die minimale Helligkeit (schwarz) von 255 dividieren.

```
train.images <- data.frame(t(apply(train.images, 1, c))) /
max(fashion_mnist$train$x)
test.images <- data.frame(t(apply(test.images, 1, c))) /
max(fashion_mnist$train$x)
```

Wir erstellen nun noch zwei Datensätze, die alle Trainings- und Testdaten (Bilder und Beschriftungen) enthalten.

```
pixs <- 1:ncol(fashion_mnist$train$x)^2
names(train.images) <- names(test.images) <- paste0('pixel', pixs)
train.labels <- data.frame(label = factor(train.labels))
test.labels <- data.frame(label = factor(test.labels))
train.data <- cbind(train.labels, train.images)
test.data <- cbind(test.labels, test.images)
```

Künstliche Neurale Netze

Beginnen wir nun mit dem Aufbau eines einfachen neuronalen Netzes, um unsere Kleidungsstücke zu klassifizieren. Neuronale Netze enthalten Knoten, die untereinander Werte über Kanten austauschen. Normalerweise ist der Input an jedem Knoten eine Zahl, die gemäß einer nichtlinearen Funktion, der Summe der Inputs mit den dazugehörigen Gewichten der Kanten, transformiert wird, wobei dies die Parameter sind, die während des Trainierens optimiert werden können.

In diesem Beitrag wir gezeigt, wie künstliche neuronale Netze mit einer unterschiedlichen Anzahl von Hidden-Layer abschneiden, und ich vergleiche diese Netze auch mit einem Convolutional-Neuronal-Net (CNN), das bei visuellen Bildern oft besser geeignet ist. Wir werden sehen wie einige grundlegende Modelle codiert werden, aber es wird hierbei auf weitere optimierungen des Netzes weitgehend verzichtet. Im Wesentlichen kommt es darauf an, dass diese Netzwerk-Parameter / -Architektur weitgehend von der Datenstruktur, Größe und Komplexität der Input-daten abhängen. Je mehr verborgene Schichten hinzugefügt werden, desto komplexere nichtlineare Beziehungen können modelliert werden.

Obwohl neuronale Netze mit TensorFlow und Keras problemlos in RStudio erstellt werden können, werden wir hier Funktionen von RStudio verwenden, mit der Sie Python in RStudio ausführen können. Dies kann auf zwei Arten erfolgen: Entweder wir wählen „Terminal“ in der Ausgabekonsole in RStudio und führen Python über das Terminal aus, oder wir verwenden die Basisfunktion `system2()`, um Python in RStudio auszuführen.

Um den Befehl `system2()` zu verwenden, ist es wichtig, zuerst zu prüfen, welche Version von Python verwendet werden soll. Sie können überprüfen, welche Versionen von Python auf Ihrem Computer installiert sind, indem Sie `python --version` in Terminal ausführen. Beachten Sie, dass Sie mit RStudio 1.1 (1.1.383 oder höher) Terminal direkt von RStudio aus auf der Registerkarte „Terminal“ ausführen können. Sie können auch `python3 --`

`version` ausführen, um zu überprüfen, ob Sie Python Version 3 installiert haben. Auf meinem Computer gibt `python3 --version` Python 3.8.10 zurück. Sie können auch `which python` (oder `which python3`, wenn Sie Python Version 3 installiert haben) in Terminal ausführen, wobei hier der Pfad zurückgegeben wird, in dem Ihre Version Python installiert ist. Da wir in diesen Beispielen Python Version 3 verwenden werden, geben wir dies als Pfad für Python in der Funktion `use_python()` aus dem `reticulate` Paket an. Ob die gewünschte Python-Version verwendet wird, können wir mit dem `sys`-Paket von Python überprüfen. Stellen Sie bitte sicher, dass Sie im folgenden Code den Pfad auf die Version von Python ändern, die Sie verwenden möchten, und wo diese Version letztendlich installiert ist.

```
Sys.setenv(RETICULATE_PYTHON = "/usr/bin/python3")
library(reticulate)
use_python("/usr/bin/python3")

sys <- import("sys")
sys$version

## [1] "3.8.10 (default, Mar 15 2022, 12:22:08) \n[GCC 9.4.0]"
```

Nachdem wir nun die zu verwendende Version von Python angegeben haben, können wir unser Python-Skript in RStudio mit der Funktion `system2()` ausführen.

```
python_file <- "simple_neural_network_fashion_mnist.py"
system2("python3", args = c(python_file), stdout = NULL, stderr = "")
```

Ich werde Sie nun Schritt für Schritt durch das im obigen Befehl aufgerufene Skript führen. Zuerst laden wir die erforderlichen Pakete in Python und setzen den Random Seed für die Wiederholbarkeit.

```
import numpy as np
np.random.seed(815)
import tensorflow
tensorflow.random.set_seed(9876)
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Conv2D,
MaxPooling2D
from keras.layers import Dropout, SpatialDropout2D

from keras.applications.vgg19 import VGG19
from keras.preprocessing import image
from keras.applications.vgg19 import preprocess_input

from keras.models import Model
from keras.datasets import fashion_mnist
```

```
from tensorflow.keras.utils import to_categorical

from keras import models
from keras import layers
from keras import optimizers
```

Wir laden dann die Mode-MNIST-Daten aus „Keras“ und normalisieren die Daten, indem wir sie durch die minimale Helligkeit von 255 dividieren.

```
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
train_images = train_images / train_images.max()
test_images = test_images / test_images.max()
```

Wir beginnen nun mit der Erstellung eines einfachen neuronalen Netzwerks, das eine Hidden-Layer Schicht enthält. Beachten Sie, dass wir zuerst die Eingabe von 28 x 28 Pixeln glätten müssen, da wir hier die nicht transformierten, aber normalisierten Daten verwenden. Wir fügen eine Hidden-Layer Schicht hinzu, die Ausgabe `output = relu(dot(input, kernel) + bias)`, wobei sich die (`relu`) Aktivierungs-Funktion als günstig erwiesen hat. Wir setzen die Anzahl der Knoten auf 128, da dies in unserem Fall ebenfalls günstig ist. Die Anzahl der Knoten könnte im Wesentlichen eine der Zahlen 32, 64, 128, 256 und 512 sein. Der `softmax`-Layer ordnet dann jeder der zehn Bekleidungsklassen, je na Input, Wahrscheinlichkeiten der Zugehörigkeit zu, weshalb es in diesem Layer auch zehn Knoten gibt.

```
three_layer_model = Sequential()
three_layer_model.add(Flatten(input_shape = (28, 28)))
three_layer_model.add(Dense(128, activation = 'relu'))
three_layer_model.add(Dense(10, activation = 'softmax'))
```

Nach der Erstellung des neuronalen Netzes kompilieren wir es. Wir spezifizieren `sparse_categorical_crossentropy` als loss function, die gut für kategoriale Mehrklassen geeignet ist. Der Optimierer regelt die Lernrate; `adam` (adaptive moment estimation) ähnelt dem klassischen stochastischen Gradientenabstieg und ist normalerweise eine sichere Wahl für den Optimierer. Wir legen unsere Metrik auf die Genauigkeit (oder den Prozentsatz) korrekt klassifizierter Bilder fest. Im Folgenden passen wir das Modell unter Verwendung von zehn Iterationen durch die Trainingsdaten („Epochen“) an unseren Trainingsdatensatz an. Hier werden 70 % der Daten für das Training und 30 % für die Validierung verwendet.

```
three_layer_model.compile(loss = 'sparse_categorical_crossentropy',
                          optimizer = 'adam', metrics = ['acc'])
three_layer_model.fit(train_images, train_labels, epochs = 10,
                      validation_split = 0.3, verbose = 2)

## Epoch 1/10
## 1313/1313 - 3s - loss: 0.5271 - acc: 0.8154 - val_loss: 0.4220 -
val_acc: 0.8537 - 3s/epoch - 2ms/step
## Epoch 2/10
## 1313/1313 - 2s - loss: 0.3953 - acc: 0.8573 - val_loss: 0.3821 -
```

```
val_acc: 0.8653 - 2s/epoch - 1ms/step
## Epoch 3/10
## 1313/1313 - 2s - loss: 0.3531 - acc: 0.8722 - val_loss: 0.3693 -
val_acc: 0.8648 - 2s/epoch - 1ms/step
## Epoch 4/10
## 1313/1313 - 2s - loss: 0.3260 - acc: 0.8803 - val_loss: 0.3650 -
val_acc: 0.8724 - 2s/epoch - 1ms/step
## Epoch 5/10
## 1313/1313 - 2s - loss: 0.3046 - acc: 0.8879 - val_loss: 0.3394 -
val_acc: 0.8791 - 2s/epoch - 1ms/step
## Epoch 6/10
## 1313/1313 - 2s - loss: 0.2908 - acc: 0.8926 - val_loss: 0.3260 -
val_acc: 0.8807 - 2s/epoch - 1ms/step
## Epoch 7/10
## 1313/1313 - 2s - loss: 0.2745 - acc: 0.8996 - val_loss: 0.3182 -
val_acc: 0.8840 - 2s/epoch - 1ms/step
## Epoch 8/10
## 1313/1313 - 2s - loss: 0.2662 - acc: 0.9005 - val_loss: 0.3362 -
val_acc: 0.8772 - 2s/epoch - 1ms/step
## Epoch 9/10
## 1313/1313 - 2s - loss: 0.2536 - acc: 0.9062 - val_loss: 0.3396 -
val_acc: 0.8819 - 2s/epoch - 1ms/step
## Epoch 10/10
## 1313/1313 - 2s - loss: 0.2452 - acc: 0.9098 - val_loss: 0.3222 -
val_acc: 0.8875 - 2s/epoch - 1ms/step
## <keras.callbacks.History object at 0x7f68c274f700>
```

Als nächstes geben wir die Ergebnisse des Modells bzgl. Trainings- und Test-Verluste bzw. -Genauigkeit aus.

```
test_loss, test_acc = three_layer_model.evaluate(test_images,
test_labels)

##   1/313 [..............................] - ETA: 5s - loss: 0.2631 -
acc: 0.8750 52/313 [===>..........................] - ETA: 0s - loss:
0.3299 - acc: 0.8804107/313 [=========>....................] - ETA: 0s
- loss: 0.3475 - acc: 0.8788160/313 [==============>..............] -
ETA: 0s - loss: 0.3583 - acc: 0.8744212/313
[===================>..........] - ETA: 0s - loss: 0.3625 - acc:
0.8749264/313 [========================>.....] - ETA: 0s - loss:
0.3528 - acc: 0.8783313/313 [==============================] - 0s
961us/step - loss: 0.3518 - acc: 0.8781

print("Model with three layers and ten epochs -- Test loss:",
test_loss * 100)

## Model with three layers and ten epochs -- Test loss:
35.183143615722656

print("Model with three layers and ten epochs -- Test accuracy:",
test_acc * 100)
```

```
## Model with three layers and ten epochs -- Test accuracy:
87.80999779701233
```

Wir können erkennen, dass das neuronale Netz mit einer Hidden-Layer Schicht mit einer Testgenauigkeit von 87.43 % bereits relativ gut abschneidet. Es scheint jedoch, dass wir leicht überangepasst sind (dh das Modell ist zu gut an einen bestimmten Datensatz angepasst und lässt sich daher nicht gut auf andere Datensätze übertragen), da die Genauigkeit des Trainingssatzes (88.92 %) etwas höher ist als die des Testdatensatzes. Es gibt mehrere Möglichkeiten, eine Überanpassung (overfitting) in neuronalen Netzwerken zu vermeiden, z. B. die Vereinfachung unseres Modells durch Reduzierung der Anzahl der verborgenen Schichten und Neuronen, das Hinzufügen von Dropout-Schichten, die zufällig einige der Verbindungen zwischen den Schichten entfernen, und das frühzeitige Stoppen, wenn der Validierungsverlust anzusteingen beginnt.

```
five_layer_model = Sequential()
five_layer_model.add(Flatten(input_shape = (28, 28)))
five_layer_model.add(Dense(128, activation = 'relu'))
five_layer_model.add(Dense(128, activation = 'relu'))
five_layer_model.add(Dense(128, activation = 'relu'))
five_layer_model.add(Dense(10, activation = 'softmax'))

five_layer_model.compile(loss = 'sparse_categorical_crossentropy',
                         optimizer = 'adam', metrics = ['acc'])
five_layer_model.fit(train_images, train_labels, epochs = 10,
                     validation_split = 0.3, verbose = 2)


## Epoch 1/10
## 1313/1313 - 2s - loss: 0.5209 - acc: 0.8123 - val_loss: 0.4085 -
val_acc: 0.8510 - 2s/epoch - 2ms/step
## Epoch 2/10
## 1313/1313 - 2s - loss: 0.3802 - acc: 0.8608 - val_loss: 0.3716 -
val_acc: 0.8678 - 2s/epoch - 2ms/step
## Epoch 3/10
## 1313/1313 - 2s - loss: 0.3422 - acc: 0.8739 - val_loss: 0.3696 -
val_acc: 0.8639 - 2s/epoch - 1ms/step
## Epoch 4/10
## 1313/1313 - 2s - loss: 0.3178 - acc: 0.8827 - val_loss: 0.3397 -
val_acc: 0.8794 - 2s/epoch - 1ms/step
## Epoch 5/10
## 1313/1313 - 2s - loss: 0.2999 - acc: 0.8887 - val_loss: 0.3333 -
val_acc: 0.8818 - 2s/epoch - 1ms/step
## Epoch 6/10
## 1313/1313 - 2s - loss: 0.2838 - acc: 0.8938 - val_loss: 0.3232 -
val_acc: 0.8813 - 2s/epoch - 1ms/step
## Epoch 7/10
## 1313/1313 - 2s - loss: 0.2686 - acc: 0.8984 - val_loss: 0.3192 -
val_acc: 0.8861 - 2s/epoch - 1ms/step
## Epoch 8/10
## 1313/1313 - 2s - loss: 0.2610 - acc: 0.9011 - val_loss: 0.3440 -
```

```
val_acc: 0.8737 - 2s/epoch - 1ms/step
## Epoch 9/10
## 1313/1313 - 2s - loss: 0.2493 - acc: 0.9054 - val_loss: 0.3382 -
val_acc: 0.8828 - 2s/epoch - 1ms/step
## Epoch 10/10
## 1313/1313 - 2s - loss: 0.2399 - acc: 0.9102 - val_loss: 0.3288 -
val_acc: 0.8874 - 2s/epoch - 1ms/step
## <keras.callbacks.History object at 0x7f68bc2cc8b0>

test_loss, test_acc = five_layer_model.evaluate(test_images,
test_labels)

##   1/313 [..............................] - ETA: 5s - loss: 0.1977 -
acc: 0.9062 46/313 [===>..........................] - ETA: 0s - loss:
0.3494 - acc: 0.8770 93/313 [=======>......................] - ETA: 0s
- loss: 0.3503 - acc: 0.8767142/313 [============>.................] -
ETA: 0s - loss: 0.3622 - acc: 0.8715189/313
[=================>............] - ETA: 0s - loss: 0.3786 - acc:
0.8690238/313 [====================>........] - ETA: 0s - loss:
0.3702 - acc: 0.8725288/313 [=========================>...] - ETA: 0s
- loss: 0.3643 - acc: 0.8739313/313 [==============================] -
0s 1ms/step - loss: 0.3612 - acc: 0.8752

print("Model with five layers and ten epochs -- Test loss:", test_loss
* 100)

## Model with five layers and ten epochs -- Test loss:
36.11840307712555

print("Model with five layers and ten epochs -- Test accuracy:",
test_acc * 100)

## Model with five layers and ten epochs -- Test accuracy:
87.51999735832214
```

Es scheint, dass das Modell mit zwei zusätzlichen Schichten nicht besser abschneidet als das vorherige Modell mit nur einem Hidden-Layer, da sowohl die Trainingsgenauigkeit (87,66 %) als auch die Testsatzgenauigkeit (86,43 %) geringer und der Verlust (38,38) höher ist. Lassen Sie uns nun versuchen, ob das Hinzufügen weiterer fünf Hidden-Layer die Modell-Güte verbessert, oder ob wir bereits erkennen können, dass eine zunehmende Modellkomplexität die Güte nicht mehr wesentlich verbessern kann.

```
ten_layer_model = Sequential()
ten_layer_model.add(Flatten(input_shape = (28, 28)))
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(128, activation = 'relu'))
```

```
ten_layer_model.add(Dense(128, activation = 'relu'))
ten_layer_model.add(Dense(10, activation = 'softmax'))

ten_layer_model.compile(loss = 'sparse_categorical_crossentropy',
                        optimizer = 'adam', metrics = ['acc'])
ten_layer_model.fit(train_images, train_labels, epochs = 10,
                    validation_split = 0.3, verbose = 2)

## Epoch 1/10
## 1313/1313 - 3s - loss: 0.6049 - acc: 0.7771 - val_loss: 0.4680 -
val_acc: 0.8297 - 3s/epoch - 2ms/step
## Epoch 2/10
## 1313/1313 - 3s - loss: 0.4254 - acc: 0.8468 - val_loss: 0.3916 -
val_acc: 0.8647 - 3s/epoch - 2ms/step
## Epoch 3/10
## 1313/1313 - 3s - loss: 0.3829 - acc: 0.8624 - val_loss: 0.3972 -
val_acc: 0.8608 - 3s/epoch - 2ms/step
## Epoch 4/10
## 1313/1313 - 3s - loss: 0.3609 - acc: 0.8711 - val_loss: 0.3849 -
val_acc: 0.8629 - 3s/epoch - 2ms/step
## Epoch 5/10
## 1313/1313 - 3s - loss: 0.3396 - acc: 0.8768 - val_loss: 0.3856 -
val_acc: 0.8641 - 3s/epoch - 2ms/step
## Epoch 6/10
## 1313/1313 - 3s - loss: 0.3270 - acc: 0.8831 - val_loss: 0.3521 -
val_acc: 0.8793 - 3s/epoch - 2ms/step
## Epoch 7/10
## 1313/1313 - 3s - loss: 0.3123 - acc: 0.8883 - val_loss: 0.3552 -
val_acc: 0.8796 - 3s/epoch - 2ms/step
## Epoch 8/10
## 1313/1313 - 3s - loss: 0.3044 - acc: 0.8891 - val_loss: 0.3671 -
val_acc: 0.8682 - 3s/epoch - 2ms/step
## Epoch 9/10
## 1313/1313 - 3s - loss: 0.2874 - acc: 0.8972 - val_loss: 0.3594 -
val_acc: 0.8773 - 3s/epoch - 2ms/step
## Epoch 10/10
## 1313/1313 - 3s - loss: 0.2855 - acc: 0.8966 - val_loss: 0.3825 -
val_acc: 0.8682 - 3s/epoch - 2ms/step
## <keras.callbacks.History object at 0x7f68bc0efa00>

test_loss, test_acc = ten_layer_model.evaluate(test_images,
test_labels)

##   1/313 [..............................] - ETA: 4s - loss: 0.1581 -
acc: 0.9375 43/313 [===>..........................] - ETA: 0s - loss:
0.3827 - acc: 0.8634 87/313 [=======>......................] - ETA: 0s
- loss: 0.3854 - acc: 0.8606132/313 [===========>..................] -
ETA: 0s - loss: 0.4182 - acc: 0.8554177/313
[===============>..............] - ETA: 0s - loss: 0.4236 - acc:
0.8563222/313 [====================>.........] - ETA: 0s - loss:
0.4138 - acc: 0.8582269/313 [========================>.....] - ETA: 0s
```

```
- loss: 0.4050 - acc: 0.8607313/313 [==============================] -
0s 1ms/step - loss: 0.4053 - acc: 0.8598

print("Model with ten layers and ten epochs -- Test loss:", test_loss
* 100)

## Model with ten layers and ten epochs -- Test loss:
40.52708446979523

print("Model with ten layers and ten epochs -- Test accuracy:",
test_acc * 100)

## Model with ten layers and ten epochs -- Test accuracy:
85.97999811172485
```

Das Modell mit acht Hidden-Layer schneidet in Bezug auf Trainings- (87.81 %) und Testgenauigkeit (86.89 %) sowie Verlusten (38.36) am besten ab. Dennoch ist der Leistungsunterschied zwischen dem ersten Modell mit nur einem Hidden-Layer und dem aktuellen Modell mit acht Hidden-Layern nur recht gering. Obwohl es scheint, dass wir mit so vielen Hidden-Layer zusätzliche Komplexität modellieren können, welche die Genauigkeit des Modells verbessert, müssen wir uns fragen, ob eine zunehmende Modellkomplexität auf Kosten der Interpretierbarkeit und des rechnerischen Aufwandes diese geringfügige Verbesserung der Genauigkeit und des Verlusts wert ist. Zudem werden wir noch sehen, das diese Genauikeit ebenfalls mit Vorsicht zu genießen ist.

Nachdem wir nun gesehen haben, wie sich die Anzahl der Hidden-Layer auf die Modellleistung auswirkt, wollen wir herausfinden, ob sich eine Erhöhung der Anzahl der Epochen von zehn auf fünfzig (d. h. der Anzahl der Iterationen des Modells durch die Trainingsdaten) positiv auf unser ersten Neuronalen Netzes mit einem Hidden-Layer auswirkt.

```
three_layer_model_50_epochs = three_layer_model.fit(train_images,
train_labels,
                                                    epochs = 50,
validation_split = 0.3,
                                                    verbose = 2)


## Epoch 1/50
## 1313/1313 - 2s - loss: 0.2383 - acc: 0.9100 - val_loss: 0.3274 -
val_acc: 0.8844 - 2s/epoch - 1ms/step
## Epoch 2/50
## 1313/1313 - 2s - loss: 0.2279 - acc: 0.9141 - val_loss: 0.3190 -
val_acc: 0.8911 - 2s/epoch - 1ms/step
## Epoch 3/50
## 1313/1313 - 2s - loss: 0.2217 - acc: 0.9168 - val_loss: 0.3416 -
val_acc: 0.8821 - 2s/epoch - 1ms/step
## Epoch 4/50
## 1313/1313 - 2s - loss: 0.2133 - acc: 0.9202 - val_loss: 0.3261 -
val_acc: 0.8918 - 2s/epoch - 1ms/step
## Epoch 5/50
```

```
## 1313/1313 - 2s - loss: 0.2064 - acc: 0.9220 - val_loss: 0.3341 -
val_acc: 0.8904 - 2s/epoch - 1ms/step
## Epoch 6/50
## 1313/1313 - 2s - loss: 0.2008 - acc: 0.9245 - val_loss: 0.3225 -
val_acc: 0.8912 - 2s/epoch - 1ms/step
## Epoch 7/50
## 1313/1313 - 2s - loss: 0.1906 - acc: 0.9286 - val_loss: 0.3282 -
val_acc: 0.8882 - 2s/epoch - 1ms/step
## Epoch 8/50
## 1313/1313 - 2s - loss: 0.1887 - acc: 0.9282 - val_loss: 0.3420 -
val_acc: 0.8855 - 2s/epoch - 1ms/step
## Epoch 9/50
## 1313/1313 - 2s - loss: 0.1837 - acc: 0.9313 - val_loss: 0.3433 -
val_acc: 0.8891 - 2s/epoch - 1ms/step
## Epoch 10/50
## 1313/1313 - 2s - loss: 0.1788 - acc: 0.9338 - val_loss: 0.3349 -
val_acc: 0.8918 - 2s/epoch - 1ms/step
## Epoch 11/50
## 1313/1313 - 2s - loss: 0.1721 - acc: 0.9360 - val_loss: 0.3530 -
val_acc: 0.8900 - 2s/epoch - 1ms/step
## Epoch 12/50
## 1313/1313 - 2s - loss: 0.1705 - acc: 0.9358 - val_loss: 0.3443 -
val_acc: 0.8913 - 2s/epoch - 1ms/step
## Epoch 13/50
## 1313/1313 - 2s - loss: 0.1644 - acc: 0.9384 - val_loss: 0.3465 -
val_acc: 0.8931 - 2s/epoch - 1ms/step
## Epoch 14/50
## 1313/1313 - 2s - loss: 0.1594 - acc: 0.9408 - val_loss: 0.3682 -
val_acc: 0.8881 - 2s/epoch - 1ms/step
## Epoch 15/50
## 1313/1313 - 2s - loss: 0.1561 - acc: 0.9410 - val_loss: 0.3686 -
val_acc: 0.8897 - 2s/epoch - 1ms/step
## Epoch 16/50
## 1313/1313 - 2s - loss: 0.1545 - acc: 0.9415 - val_loss: 0.3736 -
val_acc: 0.8908 - 2s/epoch - 1ms/step
## Epoch 17/50
## 1313/1313 - 2s - loss: 0.1491 - acc: 0.9442 - val_loss: 0.3987 -
val_acc: 0.8897 - 2s/epoch - 1ms/step
## Epoch 18/50
## 1313/1313 - 2s - loss: 0.1449 - acc: 0.9456 - val_loss: 0.3713 -
val_acc: 0.8903 - 2s/epoch - 1ms/step
## Epoch 19/50
## 1313/1313 - 2s - loss: 0.1418 - acc: 0.9466 - val_loss: 0.3804 -
val_acc: 0.8879 - 2s/epoch - 1ms/step
## Epoch 20/50
## 1313/1313 - 2s - loss: 0.1382 - acc: 0.9476 - val_loss: 0.3746 -
val_acc: 0.8941 - 2s/epoch - 1ms/step
## Epoch 21/50
## 1313/1313 - 2s - loss: 0.1343 - acc: 0.9498 - val_loss: 0.3888 -
val_acc: 0.8946 - 2s/epoch - 1ms/step
```

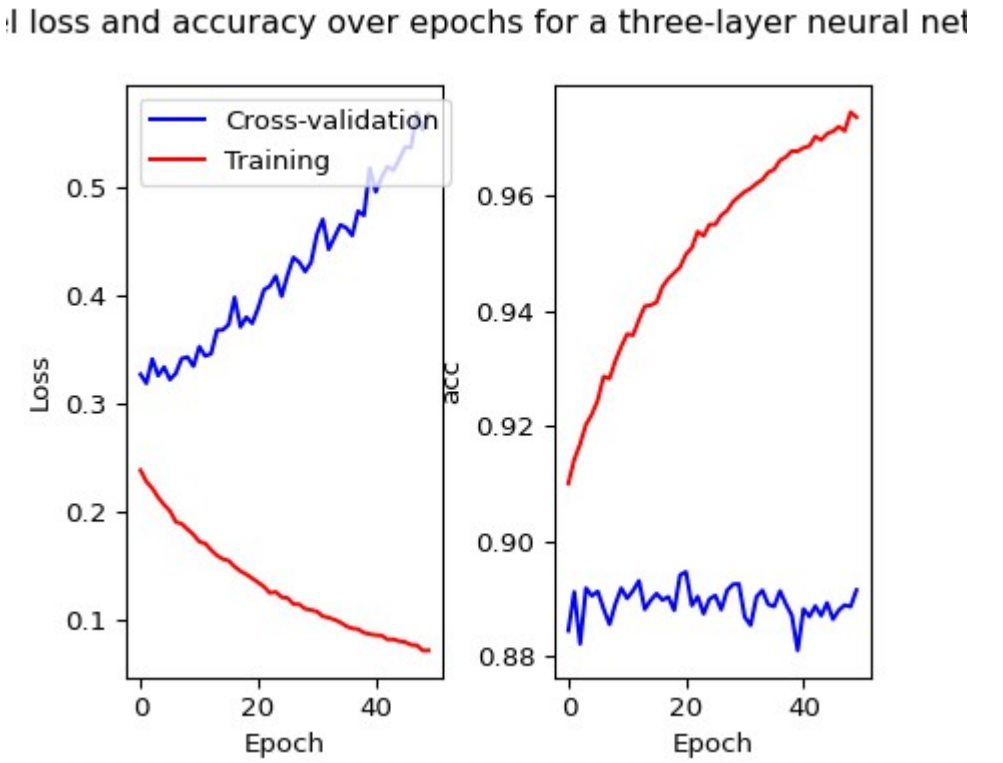
```
## Epoch 22/50
## 1313/1313 - 2s - loss: 0.1304 - acc: 0.9510 - val_loss: 0.4057 -
val_acc: 0.8888 - 2s/epoch - 1ms/step
## Epoch 23/50
## 1313/1313 - 2s - loss: 0.1247 - acc: 0.9538 - val_loss: 0.4092 -
val_acc: 0.8903 - 2s/epoch - 1ms/step
## Epoch 24/50
## 1313/1313 - 2s - loss: 0.1255 - acc: 0.9530 - val_loss: 0.4185 -
val_acc: 0.8873 - 2s/epoch - 1ms/step
## Epoch 25/50
## 1313/1313 - 2s - loss: 0.1207 - acc: 0.9550 - val_loss: 0.3997 -
val_acc: 0.8899 - 2s/epoch - 1ms/step
## Epoch 26/50
## 1313/1313 - 2s - loss: 0.1199 - acc: 0.9550 - val_loss: 0.4193 -
val_acc: 0.8905 - 2s/epoch - 1ms/step
## Epoch 27/50
## 1313/1313 - 2s - loss: 0.1144 - acc: 0.9566 - val_loss: 0.4357 -
val_acc: 0.8881 - 2s/epoch - 1ms/step
## Epoch 28/50
## 1313/1313 - 2s - loss: 0.1140 - acc: 0.9574 - val_loss: 0.4313 -
val_acc: 0.8914 - 2s/epoch - 1ms/step
## Epoch 29/50
## 1313/1313 - 2s - loss: 0.1099 - acc: 0.9590 - val_loss: 0.4227 -
val_acc: 0.8924 - 2s/epoch - 1ms/step
## Epoch 30/50
## 1313/1313 - 2s - loss: 0.1087 - acc: 0.9599 - val_loss: 0.4311 -
val_acc: 0.8925 - 2s/epoch - 1ms/step
## Epoch 31/50
## 1313/1313 - 2s - loss: 0.1074 - acc: 0.9607 - val_loss: 0.4573 -
val_acc: 0.8868 - 2s/epoch - 1ms/step
## Epoch 32/50
## 1313/1313 - 2s - loss: 0.1030 - acc: 0.9613 - val_loss: 0.4710 -
val_acc: 0.8853 - 2s/epoch - 1ms/step
## Epoch 33/50
## 1313/1313 - 2s - loss: 0.1016 - acc: 0.9621 - val_loss: 0.4432 -
val_acc: 0.8903 - 2s/epoch - 1ms/step
## Epoch 34/50
## 1313/1313 - 2s - loss: 0.0998 - acc: 0.9628 - val_loss: 0.4546 -
val_acc: 0.8914 - 2s/epoch - 1ms/step
## Epoch 35/50
## 1313/1313 - 2s - loss: 0.0973 - acc: 0.9641 - val_loss: 0.4659 -
val_acc: 0.8889 - 2s/epoch - 1ms/step
## Epoch 36/50
## 1313/1313 - 2s - loss: 0.0938 - acc: 0.9646 - val_loss: 0.4632 -
val_acc: 0.8886 - 2s/epoch - 1ms/step
## Epoch 37/50
## 1313/1313 - 2s - loss: 0.0917 - acc: 0.9662 - val_loss: 0.4561 -
val_acc: 0.8913 - 2s/epoch - 1ms/step
## Epoch 38/50
## 1313/1313 - 2s - loss: 0.0909 - acc: 0.9667 - val_loss: 0.4787 -
```

```
val_acc: 0.8890 - 2s/epoch - 1ms/step
## Epoch 39/50
## 1313/1313 - 2s - loss: 0.0877 - acc: 0.9678 - val_loss: 0.4748 -
val_acc: 0.8871 - 2s/epoch - 1ms/step
## Epoch 40/50
## 1313/1313 - 2s - loss: 0.0864 - acc: 0.9677 - val_loss: 0.5186 -
val_acc: 0.8809 - 2s/epoch - 1ms/step
## Epoch 41/50
## 1313/1313 - 2s - loss: 0.0855 - acc: 0.9684 - val_loss: 0.4967 -
val_acc: 0.8882 - 2s/epoch - 1ms/step
## Epoch 42/50
## 1313/1313 - 2s - loss: 0.0851 - acc: 0.9686 - val_loss: 0.5098 -
val_acc: 0.8868 - 2s/epoch - 1ms/step
## Epoch 43/50
## 1313/1313 - 2s - loss: 0.0816 - acc: 0.9703 - val_loss: 0.5199 -
val_acc: 0.8887 - 2s/epoch - 1ms/step
## Epoch 44/50
## 1313/1313 - 2s - loss: 0.0816 - acc: 0.9697 - val_loss: 0.5165 -
val_acc: 0.8870 - 2s/epoch - 1ms/step
## Epoch 45/50
## 1313/1313 - 2s - loss: 0.0799 - acc: 0.9708 - val_loss: 0.5266 -
val_acc: 0.8892 - 2s/epoch - 1ms/step
## Epoch 46/50
## 1313/1313 - 2s - loss: 0.0791 - acc: 0.9712 - val_loss: 0.5381 -
val_acc: 0.8864 - 2s/epoch - 1ms/step
## Epoch 47/50
## 1313/1313 - 2s - loss: 0.0765 - acc: 0.9720 - val_loss: 0.5375 -
val_acc: 0.8879 - 2s/epoch - 1ms/step
## Epoch 48/50
## 1313/1313 - 2s - loss: 0.0760 - acc: 0.9713 - val_loss: 0.5704 -
val_acc: 0.8888 - 2s/epoch - 1ms/step
## Epoch 49/50
## 1313/1313 - 2s - loss: 0.0712 - acc: 0.9745 - val_loss: 0.5542 -
val_acc: 0.8886 - 2s/epoch - 1ms/step
## Epoch 50/50
## 1313/1313 - 2s - loss: 0.0713 - acc: 0.9737 - val_loss: 0.5685 -
val_acc: 0.8914 - 2s/epoch - 1ms/step

test_loss, test_acc = three_layer_model.evaluate(test_images,
test_labels)

##   1/313 [..............................] - ETA: 5s - loss: 1.3608 -
acc: 0.8125 51/313 [===>..........................] - ETA: 0s - loss:
0.6231 - acc: 0.8768103/313 [========>.....................] - ETA: 0s
- loss: 0.6324 - acc: 0.8753158/313 [==============>..............] -
ETA: 0s - loss: 0.6541 - acc: 0.8774210/313
[====================>.........] - ETA: 0s - loss: 0.6663 - acc:
0.8784262/313 [========================>.....] - ETA: 0s - loss:
0.6318 - acc: 0.8824313/313 [==============================] - 0s
963us/step - loss: 0.6225 - acc: 0.8845
```

```
print("Model with three layers and fifty epochs -- Test loss:",
test_loss * 100)

## Model with three layers and fifty epochs -- Test loss:
62.25403547286987

print("Model with three layers and fifty epochs -- Test accuracy:",
test_acc * 100)

## Model with three layers and fifty epochs -- Test accuracy:
88.45000267028809
```

Das dreischichtige Modell, das mit fünfzig Epochen trainiert wurde, hat die höchsten Trainings- (89.19 %) und Testgenauigkeiten (88.04 %), die wir bisher gesehen haben. Der Verlust (60.83 %) ist jedoch auch etwa ein Drittel größer als wir zuvor gesehen haben. Darüber hinaus ist das Modell auch weniger zeiteffizient, da die Erhöhung der Genauigkeit unwesentlich ist, aber das Anpassen des Modells aber erheblich länger dauert. Um den Kompromiss zwischen Minimierung des Verlusts und Maximierung der Genauigkeit besser zu verstehen, zeichnen wir den Modellverlust und die Genauigkeit über die Anzahl der Epochen für die Trainings- und Kreuz-Validierungs-Daten auf.

```
# Plot loss as function of epochs
plt.subplot(1, 2, 1)
plt.plot(three_layer_model_50_epochs.history['val_loss'], 'blue')
plt.plot(three_layer_model_50_epochs.history['loss'], 'red')
plt.legend(['Cross-validation', 'Training'], loc = 'upper left')
plt.ylabel('Loss')
plt.xlabel('Epoch')

# Plot accuracy as function of epochs
plt.subplot(1, 2, 2)
plt.plot(three_layer_model_50_epochs.history['val_acc'], 'blue')
plt.plot(three_layer_model_50_epochs.history['acc'], 'red')
plt.ylabel('acc')
plt.xlabel('Epoch')
plt.subplots_adjust(wspace = .35)

# Include plot title and show the plot
plt.suptitle('Model loss and accuracy over epochs for a three-layer
neural network')
plt.show()
```



Wir können hier recht gut beobachten, dass für die Trainingsdaten der Verlust auf null sinkt, während die Genauigkeit durch Overfitting auf 1 ansteigt. Aus diesem Grund überprüfen wir auch die Güte des Modells bzgl. Kreuzvalidierungsdaten, bei denen wir beobachten können, dass der Verlust mit der Anzahl der Epochen zunimmt, während die Genauigkeit relativ stabil bleibt. Unter Verwendung dieser Zahl können wir eine möglichst „optimale“ Anzahl von Epochen auswählen, sodass die Genauigkeit maximiert und der Verlust minimiert wird. Wenn wir uns die Genauigkeit der Kreuzvalidierungsdaten ansehen, sehen wir, dass die Genauigkeitsspitze bei etwa 20 Epochen liegt, für die der Verlust etwa 0,4 beträgt. Allerdings werden ähnliche Genauigkeiten, aber viel geringere Verluste und Modellierungszeiten mit etwa 6 und 12 Epochen erreicht, und somit könnten wir uns eher dafür entscheiden, unser Modell mit etwa 6 oder 20 Epochen zu trainieren.

In Bezug auf die Modellausgabe sind die zurückgegebenen klassifizierungs Wahrscheinlichkeiten pro Klasse bzw. Kleidungskategorie. Wir können das Mehrheitsvotum berechnen, indem wir die Klasse nehmen, die das Maximum der vorhergesagten Wahrscheinlichkeiten aller Klassen hat. Wir können die ersten zehn Elemente des `majority_vote` dictionary ausdrucken, welches wir wie folgt erhalten:

```
predictions = three_layer_model.predict(test_images)
for i in range(10):
  print("Prediction " + str(i) + ": " +
str(np.argmax(np.round(predictions[i]))))
  print("Actual " + str(i) + ": " + str(test_labels[i]))

## Prediction 0: 9
## Actual 0: 9
```

```
## Prediction 1: 2
## Actual 1: 2
## Prediction 2: 1
## Actual 2: 1
## Prediction 3: 1
## Actual 3: 1
## Prediction 4: 0
## Actual 4: 6
## Prediction 5: 1
## Actual 5: 1
## Prediction 6: 4
## Actual 6: 4
## Prediction 7: 6
## Actual 7: 6
## Prediction 8: 5
## Actual 8: 5
## Prediction 9: 7
## Actual 9: 7
```

Alle außer der fünften (Nummer 4) Klassifizierung (prediction) sind richtig. In der Fünften wird ein Hemd (Kategorie 6) fälschlicherweise als Oberteil (Kategorie 0) klassifiziert.

Convolutional Neural Networks, CNN's

Im folgenden zeigen wir, wie Sie ein konvolutionelles neuronales Netzwerk aufbauen und seine Güte mit den zuvor vorgestellten neuronalen Netzwerken vergleichen, vor allem, weil sich gezeigt hat, dass CNN's im Allgemeinen bei visuellen Bilddaten besser geeignet sind. Was in einem Convolutional Neural Network im Wesentlichen passiert, ist, dass eine kleinere Matrix (die „Filtermatrix" oder „Kernel") über die Vollbildmatrix gleitet, sich also Pixel für Pixel bewegt, die Filtermatrix wird mit dem entsprechenden Teil der Vollbildmatrix Matrix-multipliziert, welcher durch die Filtermatrix in diesem Moment abgedeckt wird. Folgend, summieren wir diese Werte und wiederholt dies dann, bis die gesamte Bildmatrix abgedeckt ist.

Da wir unsere Daten für ein Convolutional Neural Network etwas anders aufbereiten müssen, laden wir die Daten neu und formen die Bilder um, um sie zu „glätten" (flatten). Die letzte „1" in den Reshape-Dimensionen steht für Graustufen, da wir die Bilder ja in einer Schwarz-Weiß-Skala bereitgestellt bekommen. Wenn wir RGB-Bilder hätten, würden wir die „1" in eine „3" für RGB-Werte ändern.

```
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
train_images = train_images.reshape(60000, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
```

Wir müssen jetzt erst einmal sicher stellen, dass die Werte der Pixel, die von 0 bis 255 reichen, vom Typ-Float sind und normalisieren dann die Werte wie zuvor.

```
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')
```

```
train_images = train_images / train_images.max()
test_images = test_images / test_images.max()
```

Das hier benutzte Convolutional Neural Network kann nicht mit kategorialen Labels umgehen. Daher wandeln wir die Labels in binäre Werte-Vektoren um, wobei alle Vektoren die Länge zehn haben (da es ja zehn Kategorien gibt), eine „1“ an der Index-Stelle der Kategorie und Nullen an den anderen Stellen. Beispielsweise würden die Kategorien 3 und 8 wie folgt codiert [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] und [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]. Diese Transformation wird als „One Hot Encoding“ bezeichnet.

```
train_labels_bin = to_categorical(train_labels)
test_labels_bin = to_categorical(test_labels)
```

Jetzt können wir mit dem Aufbau unseres Convolutional Neural Network beginnen. Die erste Schicht `Conv2D` ist eine Convolution-Schicht, die eine zweidimensionale Matrix von 28 mal 28 Pixeln in Graustufen als Eingabe verwendet. Wir verwenden in dieser Schicht nach wie vor 128 Knoten, da die Menge der Daten nicht wirklich extrem groß ist und wir unser Modell nicht unnötig komplex machen wollen. Die Filtermatrix hat die Größe 3 x 3, was durchaus Standard ist. Wie zuvor verwenden wir die lineare („relu“) Aktivierungsfunktion. Die `MaxPooling2D`-Schicht reduziert die Dimensionalität (und damit die erforderliche Rechenleistung), indem sie das Maximum des Teils des Eingangsbildes ausgibt, der von der Filtermatrix erfasst wird. Die `Flatten`-Schicht glättet einfach das Ergebnis der vorherigen Ebene in einen einzigen langen Vektor. Wie wir zuvor gesehen haben, weist der `softmax`-Layer dann jeder der zehn Kleidungskategorien Zugehörigkeits-Wahrscheinlichkeiten zu. Beachten Sie, dass wir denselben Optimierer und dieselbe Metrik wie zuvor verwenden, aber dass wir jetzt „categorical_crossentropy“ anstelle von „sparse_categorical_crossentropy“ als Verlustfunktion verwenden. Der Grund dafür ist, dass ersteres für One-Hot-codierte Labels funktioniert, während die andere für kategoriale Labels funktioniert.

```
conv_model = Sequential()
conv_model.add(Conv2D(128, (3, 3), input_shape = (28, 28, 1)))
conv_model.add(Activation('relu'))
conv_model.add(MaxPooling2D(pool_size = (2, 2)))
conv_model.add(Conv2D(128, (3, 3)))
conv_model.add(Activation('relu'))
conv_model.add(MaxPooling2D(pool_size = (2, 2)))
conv_model.add(Flatten())
conv_model.add(Dense(128))
conv_model.add(Dense(10))
conv_model.add(Activation('softmax'))
```

Wir optimieren nun unser Modell an die Trainingsdaten an, wobei wir das Argument `batch_size` gleich der Anzahl der Neuronen in den Convolution-Schicht (= 128) setzen.

```
conv_model.compile(loss = "categorical_crossentropy",
                   optimizer = 'adam', metrics = ['acc'])
conv_model.fit(train_images, train_labels_bin, batch_size = 128,
```

```
                epochs = 10, verbose = 2)


## Epoch 1/10
## 469/469 - 4s - loss: 0.4678 - acc: 0.8303 - 4s/epoch - 9ms/step
## Epoch 2/10
## 469/469 - 3s - loss: 0.3095 - acc: 0.8870 - 3s/epoch - 6ms/step
## Epoch 3/10
## 469/469 - 3s - loss: 0.2634 - acc: 0.9035 - 3s/epoch - 6ms/step
## Epoch 4/10
## 469/469 - 3s - loss: 0.2347 - acc: 0.9144 - 3s/epoch - 6ms/step
## Epoch 5/10
## 469/469 - 3s - loss: 0.2109 - acc: 0.9225 - 3s/epoch - 6ms/step
## Epoch 6/10
## 469/469 - 3s - loss: 0.1903 - acc: 0.9305 - 3s/epoch - 6ms/step
## Epoch 7/10
## 469/469 - 3s - loss: 0.1756 - acc: 0.9359 - 3s/epoch - 6ms/step
## Epoch 8/10
## 469/469 - 3s - loss: 0.1593 - acc: 0.9424 - 3s/epoch - 6ms/step
## Epoch 9/10
## 469/469 - 3s - loss: 0.1441 - acc: 0.9483 - 3s/epoch - 6ms/step
## Epoch 10/10
## 469/469 - 3s - loss: 0.1337 - acc: 0.9520 - 3s/epoch - 6ms/step
## <keras.callbacks.History object at 0x7f68c0711fd0>

test_loss, test_acc = conv_model.evaluate(test_images,
test_labels_bin)

##   1/313 [..............................] - ETA: 40s - loss: 0.5772
- acc: 0.9375 31/313 [=>............................] - ETA: 0s -
loss: 0.2782 - acc: 0.9244  65/313 [=====>........................] -
ETA: 0s - loss: 0.2856 - acc: 0.9212 98/313
[========>.....................] - ETA: 0s - loss: 0.2974 - acc:
0.9129131/313 [============>.................] - ETA: 0s - loss:
0.3063 - acc: 0.9098165/313 [==============>..............] - ETA: 0s
- loss: 0.2998 - acc: 0.9102198/313 [=================>............] -
ETA: 0s - loss: 0.3040 - acc: 0.9105231/313
[====================>........] - ETA: 0s - loss: 0.2970 - acc:
0.9104265/313 [========================>.....] - ETA: 0s - loss:
0.2849 - acc: 0.9130299/313 [============================>..] - ETA: 0s
- loss: 0.2876 - acc: 0.9114313/313 [==============================] -
1s 2ms/step - loss: 0.2838 - acc: 0.9121

print("Convolutional model ten epochs -- Test loss:", test_loss * 100)

## Convolutional model ten epochs -- Test loss: 28.379136323928833

print("Convolutional model ten epochs -- Test accuracy:", test_acc *
100)

## Convolutional model ten epochs -- Test accuracy: 91.21000170707703
```

Obwohl wir immer noch überangepasst sind, stellen wir fest, dass das Convolutional Neural Network eine bessere Güte als die zuvor gesehenen neuronalen Netze erzielt und eine Trainingssatzgenauigkeit von 91.34 % und eine Testsatzgenauigkeit von 91.34 % sowie einen geringeren Verlust von 27.28 erreicht. Dies war zu erwarten, da sich CNN's bei visuellen Bilddaten als gut erwiesen haben. Mal sehen, ob wir Overfitting reduzieren können, indem wir die Anzahl der Neuronen von 128 auf 64 reduzieren, einen `Dropout`-Layer hinzufügen und ein frühes Stoppen ermöglichen. Beachten Sie, dass die Rate im Dropout-Layer der Prozentsatz der Verbindungen zwischen den Layern ist, die entfernt werden. `SpatialDropout2D` ist eine spezielle Art von `Dropout`-Layer für CNN's, die bestimmte Multiplikationen der Filtermatrix mit Teilen des Originalbildes weg lässt, bevor sie sich über alle Bewegungen der Filtermatrix über das Originalbild hinweg zusammenfasst.

```
conv_model_reduce_overfit = Sequential()
conv_model_reduce_overfit.add(Conv2D(64, (3, 3), input_shape = (28,
28, 1)))
conv_model_reduce_overfit.add(Activation('relu'))
conv_model_reduce_overfit.add(MaxPooling2D(pool_size = (2, 2)))
conv_model_reduce_overfit.add(Dropout(0.5))
conv_model_reduce_overfit.add(Conv2D(64, (3, 3)))
conv_model_reduce_overfit.add(SpatialDropout2D(0.5))
conv_model_reduce_overfit.add(Activation('relu'))
conv_model_reduce_overfit.add(MaxPooling2D(pool_size = (2, 2)))
conv_model_reduce_overfit.add(Flatten())
conv_model_reduce_overfit.add(Dense(64))
conv_model_reduce_overfit.add(Dropout(0.5))
conv_model_reduce_overfit.add(Dense(10))
conv_model_reduce_overfit.add(Activation('softmax'))
```

Bei der Anpassung unseres Modells ermöglichen wir auch ein frühes Stoppen, um eine Überanpassung möglichst zu verhindern. Anstatt alle angegebenen Epochen zu durchlaufen, stoppen wir die Iterationen durch die Epoche automatisch, sobald festgestellt wird, dass der Validierungsverlust zunimmt.

```
conv_model_reduce_overfit.compile(loss = "categorical_crossentropy",
                  optimizer = 'adam', metrics = ['acc'])
conv_callback = keras.callbacks.EarlyStopping(monitor = 'val_loss',
patience = 3)
conv_model_reduce_overfit.fit(train_images, train_labels_bin,
validation_split = 0.3,
               epochs = 10, verbose = 2, callbacks = [conv_callback],
batch_size = 64)

## Epoch 1/10
## 657/657 - 3s - loss: 0.7586 - acc: 0.7287 - val_loss: 0.4479 -
val_acc: 0.8415 - 3s/epoch - 4ms/step
## Epoch 2/10
## 657/657 - 2s - loss: 0.5065 - acc: 0.8207 - val_loss: 0.3831 -
val_acc: 0.8642 - 2s/epoch - 3ms/step
## Epoch 3/10
```

```
## 657/657 - 2s - loss: 0.4484 - acc: 0.8413 - val_loss: 0.3721 -
val_acc: 0.8688 - 2s/epoch - 3ms/step
## Epoch 4/10
## 657/657 - 2s - loss: 0.4247 - acc: 0.8513 - val_loss: 0.3332 -
val_acc: 0.8773 - 2s/epoch - 3ms/step
## Epoch 5/10
## 657/657 - 2s - loss: 0.4055 - acc: 0.8573 - val_loss: 0.3231 -
val_acc: 0.8841 - 2s/epoch - 3ms/step
## Epoch 6/10
## 657/657 - 2s - loss: 0.3911 - acc: 0.8614 - val_loss: 0.3186 -
val_acc: 0.8853 - 2s/epoch - 3ms/step
## Epoch 7/10
## 657/657 - 2s - loss: 0.3817 - acc: 0.8663 - val_loss: 0.3076 -
val_acc: 0.8907 - 2s/epoch - 3ms/step
## Epoch 8/10
## 657/657 - 2s - loss: 0.3718 - acc: 0.8684 - val_loss: 0.2977 -
val_acc: 0.8912 - 2s/epoch - 3ms/step
## Epoch 9/10
## 657/657 - 2s - loss: 0.3658 - acc: 0.8710 - val_loss: 0.2922 -
val_acc: 0.8929 - 2s/epoch - 3ms/step
## Epoch 10/10
## 657/657 - 2s - loss: 0.3584 - acc: 0.8729 - val_loss: 0.3322 -
val_acc: 0.8791 - 2s/epoch - 3ms/step
## <keras.callbacks.History object at 0x7f68c06a9610>

test_loss, test_acc = conv_model_reduce_overfit.evaluate(test_images,
test_labels_bin)

##   1/313 [..............................] - ETA: 11s - loss: 0.5455
- acc: 0.8750 34/313 [==>...........................] - ETA: 0s -
loss: 0.3333 - acc: 0.8906  70/313 [=====>........................] -
ETA: 0s - loss: 0.3401 - acc: 0.8826107/313
[=========>....................] - ETA: 0s - loss: 0.3543 - acc:
0.8744143/313 [============>.................] - ETA: 0s - loss:
0.3531 - acc: 0.8776180/313 [================>.............] - ETA: 0s
- loss: 0.3593 - acc: 0.8757217/313 [====================>.........] -
ETA: 0s - loss: 0.3560 - acc: 0.8757253/313
[========================>......] - ETA: 0s - loss: 0.3498 - acc:
0.8757289/313 [============================>...] - ETA: 0s - loss:
0.3474 - acc: 0.8767313/313 [==============================] - 0s
1ms/step - loss: 0.3452 - acc: 0.8775

print("Convolutional model ten epochs reduced overfit -- Test loss:",
test_loss * 100)

## Convolutional model ten epochs reduced overfit -- Test loss:
34.52414870262146

print("Convolutional model ten epochs reduced overfit -- Test
accuracy:", test_acc * 100)
```

```
## Convolutional model ten epochs reduced overfit -- Test accuracy:
87.74999976158142
```

In den Ergebnissen erkennen wir, dass die Trainings- und Testgenauigkeiten zwar abgenommen haben, aber jetzt viel näher an den vorherigen Ergebnissen sind. Die Testgenauigkeit hat nicht wesentlich abgenommen, aber die Trainingsgenauigkeit, was bedeutet, dass Overfitting viel weniger ein Problem darstellt als zuvor. Als Nächstes können wir die ersten zehn Vorhersagen aus dem Modell und die ersten zehn tatsächlichen Labels anzeigen und sie vergleichen.

```
predictions = conv_model_reduce_overfit.predict(test_images)
for i in range(10):
  print("Prediction " + str(i) + ": " +
str(np.argmax(np.round(predictions[i]))))
  print("Actual " + str(i) + ": " + str(test_labels[i]))

## Prediction 0: 9
## Actual 0: 9
## Prediction 1: 2
## Actual 1: 2
## Prediction 2: 1
## Actual 2: 1
## Prediction 3: 1
## Actual 3: 1
## Prediction 4: 6
## Actual 4: 6
## Prediction 5: 1
## Actual 5: 1
## Prediction 6: 4
## Actual 6: 4
## Prediction 7: 6
## Actual 7: 6
## Prediction 8: 5
## Actual 8: 5
## Prediction 9: 7
## Actual 9: 7
```

Beim Vergleich dieser Klassifizierung mit den ersten zehn Label im Datensatz stellen wir fest, dass diese ersten zehn richtig sind!