Labor 28.10.2019

Zur Erinnerung: Die Server stehen nur während der Laborzeit zur Verfügung. Bitte vergewissern Sie sich nach dem Beenden des Labors Spyder geschlossen zu haben.

Stichwörter: Functional API, Verzweigte Architekturen, MNIST, Cifar100, CNN, Deep-Learning

Im vorherigen Labor haben Sie die „sequentielle“ API kennen gelernt, heute werden Sie die „Functional API“ kennen lernen. Diese ermöglicht es Verzweigungen in Ihr Netz einzubauen.

Die Dokumentation zur Functional API können Sie hier finden: https://keras.io/getting-started/functional-api-guide/

Aus dem Beispiel…

```
# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)
```

…lässt sich die Vorgehensweise erkennen. Dabei werden Signale an dem jeweiligen Layer weitergegeben.

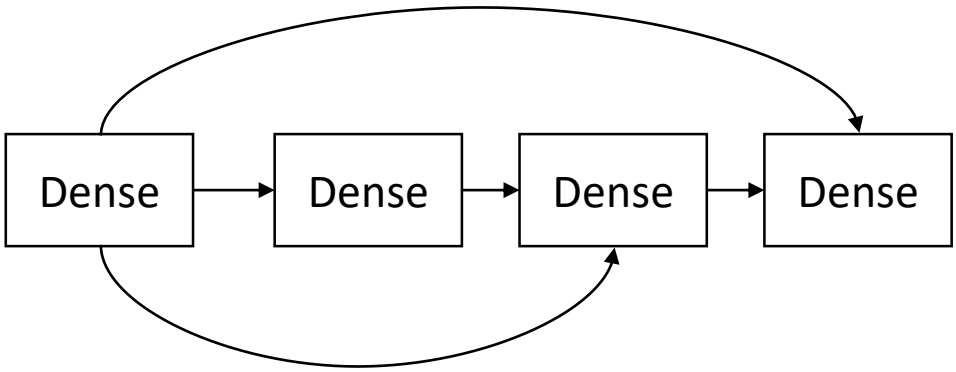
1) Schreiben Sie Ihr Netz aus dem vorherigen Labor mit Hilfe der Functional API, verwenden Sie dabei den Optimizer der am besten funktioniert hat. (Falls Sie das vorherige Labor noch nicht fertig gestellt haben, verwenden Sie den „adam“ Optimizer und erstellen Sie ihr eigenes Netz jetzt). Vergewissern Sie sich, dass es keine Unterschiede zwischen den beiden Modellen gibt mit Hilfe des „model.summary()“ Befehls. Verwenden Sie den „geflatteten“ MNIST-Datensatz aus dem vorherigen Labor zum Training. Vergleichen Sie die jeweiligen Lernkurven.
2) Eventuell haben Sie nun unterschiedliche Ergebnisse erhalten, obwohl die Architektur gleichgeblieben ist. Woran könnte dies liegen, wie kann man dagegen vornehmen?
3) Ein Pseudocode um Verzweigungen einzubauen kann wie folgt aussehen:

```
import keras
import numpy as np


# Input Layer
input1 = keras.layers.Input(shape=(16,))

# Hidden Layer
x = keras.layers.Dense(8, activation='relu')(input1)

# Concat Layer
concated = keras.layers.Concatenate()([x, input1])

# Output Layer
out = keras.layers.Dense(1,activation="linear")(concated)

model = keras.models.Model(inputs=input1, outputs=out)
model.compile("adam","mean_squared_error")
X = np.random.randint(0,100,size=(100,16))
Y = np.random.randint(0,100,size=(100,1))
model.fit(X,Y)
```

Um die einzelnen Signale zu verbinden benötigt man „Merge-Layer“: https://keras.io/layers/merge/

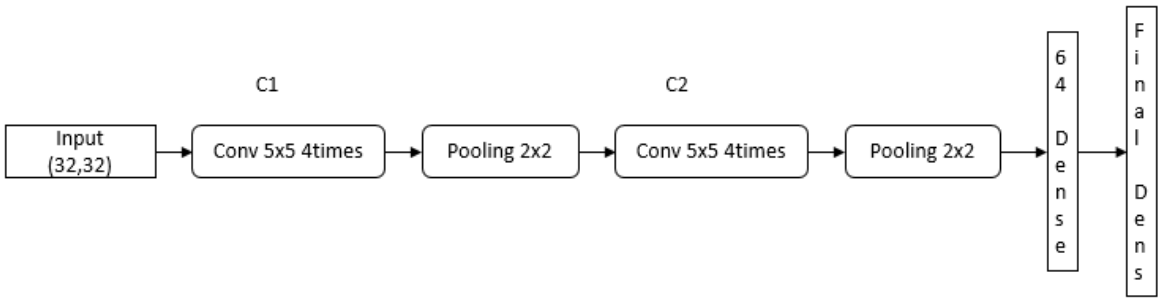
Machen Sie sich mit den einzelnen Merge-Layern vertraut und skizzieren Sie die Architektur wie in Aufgabe 4) dargestellt.

4) Beschreiben Sie unten gezeigte Architektur mit Hilfe der Functional API. Die Anzahl der Neuronen ist Ihnen überlassen.



5) Trainieren Sie diese Architektur ebenfalls mit dem „geflatteten“ MNIST-Datensatz
6) Welche Vorteile können Verzweigungen in den Architekturen bringen?
7) Um Overfitting entgegenzuwirken gibt es sogenannte „Dropout“-Layer, finden Sie die Beschreibung der Dropout-Layer in der Keras API und implementieren Sie jeweils ein Dropout-Layer zwischen zwei Dense-Layern
8) Vergleichen Sie die Ergebnisse, was fällt Ihnen auf wenn Sie Dropout verwenden?

9) Für die weiteren Aufgaben laden Sie bitte den MNIST Datensatz aus den Numpy-Dateien und flatten diesen nicht, sondern verwenden die originalen Bilder. Es ist Ihnen überlassen ob Sie die sequenzielle oder die Functional API für die folgenden Aufgaben verwenden.
10) Machen Sie sich mit Conv-Layern vertraut, die API hierzu finden Sie unter: https://keras.io/layers/convolutional/ . Vom besonderen Interesse sind zunächst die „Conv2D“ Layer, die vor allem zur Bildbearbeitung verwendet werden.
11) Machen Sie sich mit den Pooling-Layern vertraut, die API hierzu finden Sie unter: https://keras.io/layers/pooling/ . Vom besonderen Interesse sind zunächst die „MaxPooling2D“ – Layer, die vor allem zur Bildbearbeitung verwendet werden.
12) Unten sehen Sie die bekannte „LeNet“ – Architektur. Berechnen Sie die Größe der produzierten Feature-Map nach C1. Nehmen Sie dabei an, dass eine Stride von 1 und Zero-Padding der Größe 1 gewählt wurde.



Quelle: LeNet-05, Proc. Of the IEEE, November 1998

13) Eine beispielhafte Implementierung eines CNNs könnte wie folgt aussehen:

```
from keras import datasets
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
from keras.utils import to_categorical

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

x_train_cat = to_categorical(x_train)

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]),
                 activation="relu")

model.add(Conv2D(32, (3, 3)),activation="relu")
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512),activation="relu")
model.add(Dropout(0.5))
model.add(Dense(10),activation="softmax")
```

Halten Sie sich an dieses Beispiel, um das LeNet-Netz zu implementieren. Sie finden diese Datei unter dem Namen „ExampleCNN.py“. Den MNIST-Datensatz können Sie wieder über die Numpy-Dateien herunterladen.
Fügen Sie Dropout-Layer an den Stellen ein, an denen Sie es für sinnvoll halten.

14) Trainieren Sie die von Ihnen erstelle Architektur, wählen Sie dabei eine Epochenzahl von mindestens 25.

15) Laden sie den Cifar-100-Datensatz (https://keras.io/datasets/#cifar100-small-image-classification) Dieser wird ihnen wieder über Numpy-Files (wie der MNIST-Datensatz) zur Verfügung gestellt. Laden Sie dabei ebenfalls wieder Trainings und Testdaten. Der Cifar100 Datensatz besteht dabei aus 32x32 großen RGB Bildern, und besitzt 100 unterschiedliche Klassen. Der CIFAR-100-Datensatz ist dabei in 20 Superklassen und 100 Klassen eingeteilt. Diese sind hier aufgelistet:

Superclass	**Classes**
Aquatic mammals	Beaver, dolphin, otter, seal, whale
Fish	Aquarium fish, flatfish, ray, shark, trout
Flowers	Orchids, poppies, roses, sunflowers, tulips
Food containers	Bottles, bowls, cans, cups, plates
Fruit and vegetables	Apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear,leopard,lion,tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm

people	baby, boy, girl, man, woman
Reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Bauen Sie ihre Architektur so um, dass Sie auf dem Cifar-100 Datensatz trainiert werden kann

16) Trainieren Sie ihre Architektur auf dem Cifar-100 Datensatz

Vorschau/Vorbereitung/Bonus:

1) Keras stellt sogenannte „Applications“ zur Verfügung: https://keras.io/applications/ . Machen Sie sich mit diesen vertraut und probieren Sie eine Application auf den Cifar-100-Datensatz anzuwenden

2) Um zu sehen welche Klassen miteinander verwechselt werden ist es sinnvoll sich neben der Genauigkeit noch andere Metriken mit zu verwenden. Eine beliebte Art ist die sogenannte „Confusion-Matrix“: https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py . Wenden Sie diese Matrix auf ihr trainiertes Le-net an, welche Klassen werden häufig miteinander verwechselt?