

Kategorisch Train

January 14, 2021

1 Trainieren des neuronalen Netzes Kategorisch

Zu Beginn müssen wieder alle Bibliotheken eingebunden werden.

```
[6]: #import all necessary packages
import os
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
import IPython
import kerastuner as kt
```

Für den Fall, dass die bereits mit Skript 1 vorbereiteten Datensätze nicht im selben Verzeichnis wie die Python-Datei liegt muss das Verzeichnis des Datensatzes angegeben werden. Dies kann mittels folgendem Befehl durchgeführt werden:

```
[7]: #function to set the start working directory manually
#not necessary, if the .py file is executed in the directory of the .csv files
#os.chdir('D:\\OneDrive - bwedu\\Uni\\09 ABC 1\\Neuronale_
↳Netzwerke\\Python\\Wein')
```

Nun muss das Arbeitsverzeichnis geändert werden. Dabei wird das Arbeitsverzeichnis so geändert, dass auf die zuvor vorbereiteten Daten zugegriffen werden kann.

Codeabsatz 1 speichert das aktuelle Arbeitsverzeichnis.

Codeabsatz 2 ändert das Arbeitsverzeichnis.

Codeabsatz 3 importiert die mittels Datenvorbereitungsskript zuvor erstellten Daten.

Codeabsatz 4 convertiert die Daten in ein numpy-Array.

Dabei ist zu beachten, dass die erste Spalte des Datensatzes nicht von Relevanz ist und daher nicht beachtet wird.

Die Daten werden von Kategorie 3-9 in Kategorie 0-6 konvertiert.

Im letzten Codeabsatz wird nach dem Import der Daten wieder in das Ausgangsverzeichnis gewechselt.

```
[8]: #save the current working directory
current_wd = os.getcwd()

#change the working directory to the 'prep_dataset' directory, where the
→prepared data are saved as .csv files
os.chdir('prep_dataset')

#import the prepared data as panda dataframes
W_train_samples = pd.read_csv('wine_train_samples.csv')
W_train_labels = pd.read_csv('wine_train_labels_cont.csv')

#convert panda dataframe to a numpy array
#leave out the first column, because this are the row numbers
W_train_samples = pd.DataFrame.to_numpy(W_train_samples)[: , 1:13]
W_train_labels = pd.DataFrame.to_numpy(W_train_labels)[: , 1]-3

#change the working directory back to the original working directory
os.chdir(current_wd)
```

Folgende Codezeilen erzeugen ein zufälliges neuronales Netz (model_builder), welches später trainiert werden kann.

Zu beachten ist, dass dem model_builder die Parameter (Anzahl layer, learning Rate und Anzahl Knoten) systematisch auswählt.

Für dieses Beispiel kann der model_builder die Parameter in folgenden Bereichen wählen:

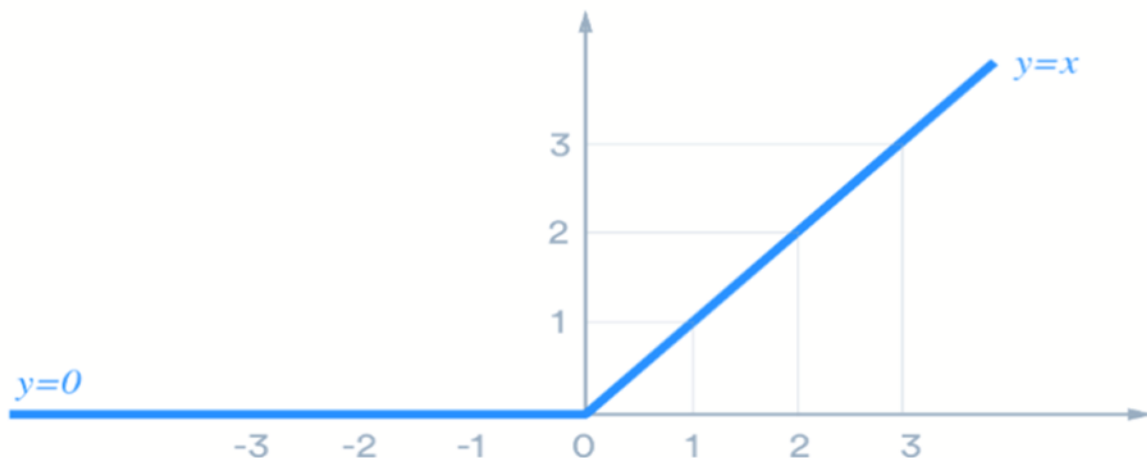
Anzahl hidden layer: 1-5

Learning Rate: 1e-5 bis 1e-4

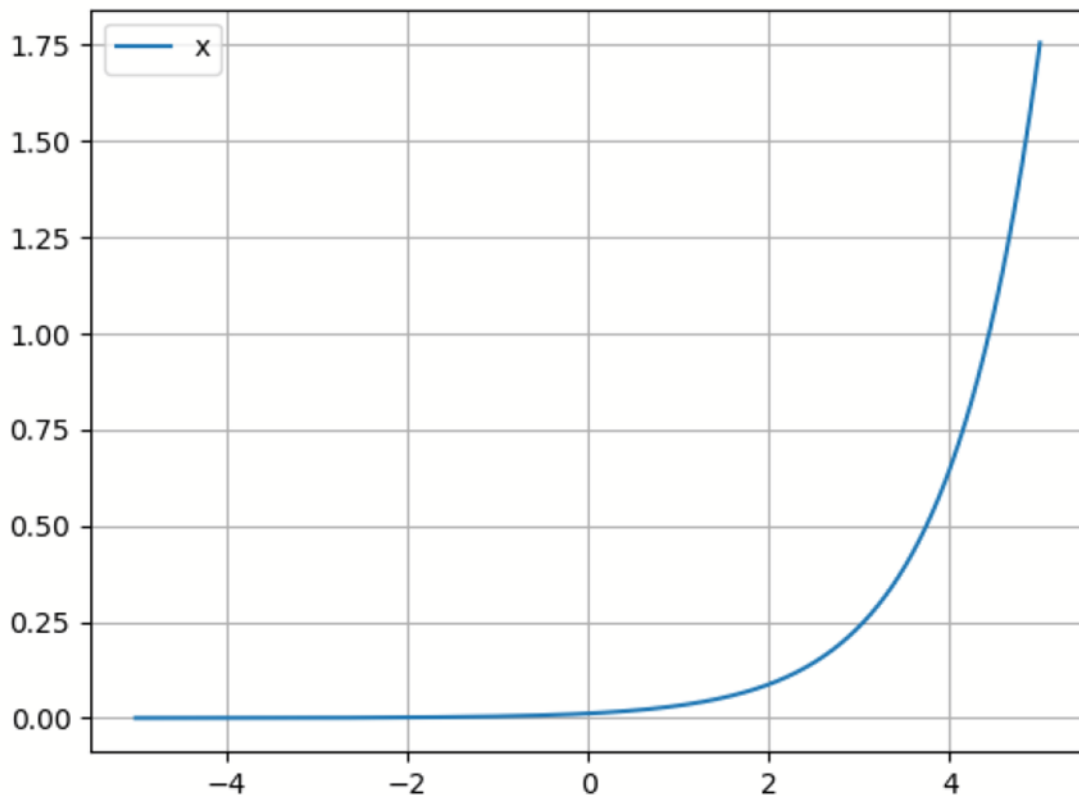
Knoten der hidden Layer: 16-160 (Differenz der Anzahl an Knoten zwischen den einzelnen Layer: 16 Knoten)

Aktivierungsfunktion: relu bzw. softmax

relu:



softmax:



Beim kompilieren des Modells ist zu beachten, dass hier als Fehler (loss) die sparse categorical crossentropy berechnet wird.

Zum Schluss wird das fertige Modell zurückgegeben.

```
[9]: #the definition of the model building function
#here it is possible to specify the model, the most parameters
#with hp functions it is possible to specify a range of values, in which an
    ↪ optimizer can optimize the model
def model_builder(hp):
    #the number of layers in a range from 1 to 5
    hp_layers = hp.Int('layers', min_value = 1, max_value = 5)
    #test different learning rates
    hp_learning_rate = hp.Choice('learning_rate', values = [1e-4, 3*1e-5,
    ↪ 1e-5])
```

```

#the input layer with 12 nodes
inputs = keras.Input(shape=(12,))

x = inputs
for i in range(10):
    #for each layer a different number of nodes (16 to 160)
    x = keras.layers.Dense(units = hp.Int('units_' + str(i),
        min_value = 16, max_value = 160, step = 16), activation = "relu")(x)

#the output layer with 7 nodes
outputs = keras.layers.Dense(7, activation = "softmax")(x)

#set the model together
model = keras.Model(inputs, outputs)

#compile the model
model.compile(optimizer = keras.optimizers.Adam(learning_rate
    = hp_learning_rate),
    loss = keras.losses.
    ↳SparseCategoricalCrossentropy(from_logits = True),
    metrics = ['accuracy'])

return model #return the finished model

```

Da die Daten im Weindatensatz (Qualität 3-9) nicht ausgeglichen sind, werden diese hier mit Gewichtungen versehen (class_weight), man beachte, dass Python hier die Zählung mit 0 beginnt (0-6 entspricht Güte 3-9).

Im 2. Absatz werden die einzelnen Einstellungen für den tuner definiert.

Wichtig: Sollte die Optimierung mehrmals wiederholt werden, so muss vor jeder Wiederholung zuerst das Verzeichnis des vorherigen tunings manuell gelöscht werden.

```

[10]: # the weights to consider the circumstance, that the training data aren't
    ↳distributed equally
class_weight = {0: 9.4, 1: 1.3, 2: 0.13, 3: 0.1, 4: 0.26, 5: 1.47, 6: 56.7}

#set settings for the tuner
tuner = kt.Hyperband(model_builder,
    objective = 'val_accuracy',
    max_epochs = 400,

    factor = 3, #the higher, the faster the optimizing, but
    ↳the smaller the probability to find the best model

    #how often should search the optimizer for optimal
    ↳model

```

```

hyperband_iterations = 1,

    ####delete directory before optimizing again!!!
    directory = 'tuner_wine_cat_full', #where should the
    optimizer save the log files for the optimizing
    project_name = 'wine_quality_advanced')

```

In unserem Code verwenden wir sogenannte Callback-Funktionen.

Sobald über 10 Epochen (patience) kein Fortschritt (Verbesserung der Genauigkeit) mehr erreicht wird, wird das Training abgebrochen.

```

[11]: #define the necessary callbacks
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
    restore_best_weights=True)

```

Nun werden die erzeugten neuronalen Netze optimiert. Folgende Parameter werden dafür verwendet:

epochs: 400 -> Insgesamt maximal 400 Wiederholungen
 batch_size: 12 -> Paralleles Training des neuronalen Netzes durch Daten
 validation_split: 0.1 -> 10% der Daten werden für eine Validation verwendet
 callbacks (Abruf wie oben definiert)

```

[12]: #start the optimizer
tuner.search(W_train_samples, W_train_labels, epochs = 400, batch_size = 12,
    validation_split = 0.1, callbacks = [early_stopping_cb] ,class_weight =
    class_weight)

```

```

Trial 725 Complete [00h 02m 40s]
val_accuracy: 0.3603448271751404

```

```

Best val_accuracy So Far: 0.5189655423164368
Total elapsed time: 01h 04m 08s
INFO:tensorflow:Oracle triggered exit

```

Durch folgenden Befehl wird aus allen getesteten neuronalen Netzen das beste ausgewählt und gespeichert (alle anderen neuronalen Netze werden nicht beachtet).

```

[13]: # Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials = 1)[0]

```

Durch folgenden print-Befehl werden die vorher definierten Parameter des besten neuronalen Netzes am Bildschirm ausgegeben.

```

[14]: #print the results of the optimizer
print(f"")

```

```

        The hyperparameter search is complete.
        Learning rate: {best_hps.get('learning_rate')}
        Number of Layers: {best_hps.get('layers')}
        """
for i in range(best_hps.get('layers')):
    print("Number of nodes in layer %i: %f" % (i, best_hps.get('units_' +
→str(i))))

```

```

The hyperparameter search is complete.
Learning rate: 0.0001
Number of Layers: 2

```

```

Number of nodes in layer 0: 160.000000
Number of nodes in layer 1: 112.000000

```

Durch folgenden Befehl wird die log file directory für die graphische Darstellung mittels tensorboard definiert.

```

[15]: #define the log-directory for the logfiles for tensorboard
root_logdir = os.path.join(os.getcwd(), "my_logs")
def get_run_logdir():
    import time
    run_id = 'wine_quality_cat_full_' + time.
→strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)
run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_06_07-15_15_22'

```

Die callbacks werden wie folgt implementiert:

Codezeile 1: Speichern des besten Resultats als “wine_quality_cat_full4.h5”

Codezeile 2: Das neuronale Netz speichert immer nur das Modell mit der höchsten Genauigkeit als Variable in Python.

Codezeile 3: Das neuronale Netz stoppt das Training, wenn nach 20 Wiederholungen keine weitere Erhöhung der Genauigkeit stattfindet.

```

[16]: #define the necessary callbacks
checkpoint_cb = keras.callbacks.ModelCheckpoint("wine_quality_cat_full4.h5",
→save_best_only=True)
early_stopping_cb2 = keras.callbacks.EarlyStopping(patience=20,
→restore_best_weights=True)
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

```

In folgendem Schritt wird das optimale Modell der Variable model zugewiesen, die nun das optimale untrainierte neuronale Netz repräsentiert, das anschließend noch trainiert werden muss.

```

[17]: # Build the model with the optimal hyperparameters
model = tuner.hypermodel.build(best_hps)

```

Mit folgendem Befehl erfolgt das finale Training des neuronalen Netzes. Mit der batch_size 6, maximal 1000 Epochen, 10% der Daten als Validierung und der callback-Funktionen.

```
[18]: #train the model
history = model.fit(W_train_samples, W_train_labels, shuffle = True, batch_size=
↳ 1, epochs = 1000, validation_split = 0.1, callbacks = [checkpoint_cb,
↳ early_stopping_cb2, tensorboard_cb], class_weight = class_weight)
```

Epoch 1/1000

1/5220 [...] - ETA: 0s - loss: 0.2517 - accuracy:

0.0000e+00WARNING:tensorflow:From C:\Users\Tobia\anaconda3\lib\site-
packages\tensorflow\python\ops\summary_ops_v2.py:1277: stop (from
tensorflow.python.eager.profiler) is deprecated and will be removed after
2020-07-01.

Instructions for updating:

use `tf.profiler.experimental.stop` instead.

WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to the
batch time (batch time: 0.0010s vs `on_train_batch_end` time: 0.0326s). Check
your callbacks.

5220/5220 [=====] - 4s 766us/step - loss: 0.5946 -
accuracy: 0.3500 - val_loss: 1.8723 - val_accuracy: 0.3862

Epoch 2/1000

5220/5220 [=====] - 4s 711us/step - loss: 0.5882 -
accuracy: 0.2626 - val_loss: 1.8525 - val_accuracy: 0.3431

Epoch 3/1000

5220/5220 [=====] - 4s 715us/step - loss: 0.5809 -
accuracy: 0.2841 - val_loss: 1.8112 - val_accuracy: 0.3655

Epoch 4/1000

5220/5220 [=====] - 4s 698us/step - loss: 0.5757 -
accuracy: 0.2937 - val_loss: 1.8343 - val_accuracy: 0.2776

Epoch 5/1000

5220/5220 [=====] - 4s 693us/step - loss: 0.5726 -
accuracy: 0.2807 - val_loss: 1.7801 - val_accuracy: 0.3690

Epoch 6/1000

5220/5220 [=====] - 4s 707us/step - loss: 0.5698 -
accuracy: 0.3092 - val_loss: 1.8339 - val_accuracy: 0.2879

Epoch 7/1000

5220/5220 [=====] - 4s 702us/step - loss: 0.5684 -
accuracy: 0.3153 - val_loss: 1.7907 - val_accuracy: 0.3379

Epoch 8/1000

5220/5220 [=====] - 4s 690us/step - loss: 0.5670 -
accuracy: 0.3224 - val_loss: 1.7822 - val_accuracy: 0.3534

Epoch 9/1000

5220/5220 [=====] - 4s 711us/step - loss: 0.5659 -
accuracy: 0.3218 - val_loss: 1.7720 - val_accuracy: 0.3586

Epoch 10/1000

5220/5220 [=====] - 4s 703us/step - loss: 0.5649 -
accuracy: 0.3500 - val_loss: 1.8057 - val_accuracy: 0.3155

Epoch 11/1000
5220/5220 [=====] - 4s 714us/step - loss: 0.5639 - accuracy: 0.3297 - val_loss: 1.7310 - val_accuracy: 0.4397

Epoch 12/1000
5220/5220 [=====] - 4s 712us/step - loss: 0.5625 - accuracy: 0.3402 - val_loss: 1.7523 - val_accuracy: 0.4103

Epoch 13/1000
5220/5220 [=====] - 4s 715us/step - loss: 0.5612 - accuracy: 0.3603 - val_loss: 1.8115 - val_accuracy: 0.3241

Epoch 14/1000
5220/5220 [=====] - 4s 701us/step - loss: 0.5608 - accuracy: 0.3487 - val_loss: 1.8059 - val_accuracy: 0.3293

Epoch 15/1000
5220/5220 [=====] - 4s 686us/step - loss: 0.5584 - accuracy: 0.3433 - val_loss: 1.7538 - val_accuracy: 0.4069

Epoch 16/1000
5220/5220 [=====] - 4s 691us/step - loss: 0.5589 - accuracy: 0.3623 - val_loss: 1.8233 - val_accuracy: 0.3190

Epoch 17/1000
5220/5220 [=====] - 4s 696us/step - loss: 0.5574 - accuracy: 0.3563 - val_loss: 1.7582 - val_accuracy: 0.4034

Epoch 18/1000
5220/5220 [=====] - 4s 705us/step - loss: 0.5567 - accuracy: 0.3596 - val_loss: 1.7762 - val_accuracy: 0.3655

Epoch 19/1000
5220/5220 [=====] - 4s 723us/step - loss: 0.5561 - accuracy: 0.3536 - val_loss: 1.7439 - val_accuracy: 0.4103

Epoch 20/1000
5220/5220 [=====] - 4s 719us/step - loss: 0.5557 - accuracy: 0.3458 - val_loss: 1.7359 - val_accuracy: 0.4276

Epoch 21/1000
5220/5220 [=====] - 4s 700us/step - loss: 0.5544 - accuracy: 0.3527 - val_loss: 1.7355 - val_accuracy: 0.4431

Epoch 22/1000
5220/5220 [=====] - 4s 699us/step - loss: 0.5543 - accuracy: 0.3603 - val_loss: 1.7638 - val_accuracy: 0.3948

Epoch 23/1000
5220/5220 [=====] - 4s 698us/step - loss: 0.5533 - accuracy: 0.3690 - val_loss: 1.7701 - val_accuracy: 0.3931

Epoch 24/1000
5220/5220 [=====] - 4s 702us/step - loss: 0.5527 - accuracy: 0.3690 - val_loss: 1.7359 - val_accuracy: 0.4259

Epoch 25/1000
5220/5220 [=====] - 4s 708us/step - loss: 0.5518 - accuracy: 0.3621 - val_loss: 1.7816 - val_accuracy: 0.3810

Epoch 26/1000
5220/5220 [=====] - 4s 714us/step - loss: 0.5488 - accuracy: 0.3640 - val_loss: 1.7391 - val_accuracy: 0.4241

Epoch 27/1000
5220/5220 [=====] - 4s 714us/step - loss: 0.5473 -
accuracy: 0.3594 - val_loss: 1.7573 - val_accuracy: 0.4121
Epoch 28/1000
5220/5220 [=====] - 4s 701us/step - loss: 0.5437 -
accuracy: 0.3573 - val_loss: 1.7530 - val_accuracy: 0.4017
Epoch 29/1000
5220/5220 [=====] - 4s 713us/step - loss: 0.5374 -
accuracy: 0.3642 - val_loss: 1.7642 - val_accuracy: 0.3879
Epoch 30/1000
5220/5220 [=====] - 4s 718us/step - loss: 0.5408 -
accuracy: 0.3575 - val_loss: 1.7635 - val_accuracy: 0.3966
Epoch 31/1000
5220/5220 [=====] - 4s 719us/step - loss: 0.5371 -
accuracy: 0.3529 - val_loss: 1.7613 - val_accuracy: 0.4017