Projekt 1:
Klassifikation von Datteln mit einem neuronalem Netz in Keras

Philipp Petermeier
Git-Archiv

2022.06.05

Inhaltsverzeichnis

1 Überblick

Die Aufgabenstellung von Projekt 1 besteht darin, diesen Datensatz [Kok+21] in dem Datteln anhand von numerischen Parametern kategorisiert werden zu nutzen, um mit Keras[Cho+15] ein neuronales Netz zu bauen, mit dem die Klassen verlässlich vorhersagt werden können. Es sollen dabei geeignete Metriken und Tensorboard zur Darstellung genutzt werden, um das Modell zu optimieren. Folgende Schritten sollen im folgenden übersichtsartig dokumentiert werden:

- Exploration des Datensatzes mit Visualisierung
- Preprocessing
- Aufbau eines ersten Models
- Darstellung des Models mit Tensorboard
- Optimizierung mit KerasTuner

Das Git-Archiv ist frei verfügbar und organisiert um für das ganze Semester sinnvoll nutzbar zu sein: Jedes Projekt hat einen eigenen Ordner, mit Daten für die Aufgabenstellung, Code und Outputs. Von der main.py aus werden die einzelnen Projekte importiert und einfach aufgerufen, um einzelnes testen zu ermöglichen und die einzelnen hier besprochenen Schritte sind jeweils in einzelnen Methoden in project1.py abgespeichert. Der Ordner "Presentation-Code"enthält die codebeispiele aus der Vorlesung, auf die zum Lernen teilweise zugegriffen wurde. Die Outputs umfassen mit python erstellte Grafiken, logs und modelle die zur Demonstration erstellt wurden, sowie die LaTeX-Dateien der Dokumentation selbst.

2 Datensatz

Als einen ersten Schritt gilt es den Datensatz nachzuvollziehen, zu diesem Zweck wurden die Funktionen von Pandas[Jef+20] genutzt, da die Datei damit auch eingelesen und folgend auch bearbeitet wird.

```
def get_overview_and_plot(data): #Getting an overview of the data, plot class distribution
    expressionslist = data.Class.value_counts()
    print(data.shape)
    print(expressionslist)
    expressions = data.Class.value_counts(normalize=True).mul(100)  # Get % of Classes
    # ------------------- Plotting
    plt.style.use('_mpl-gallery-nogrid')
    colors = plt.get_cmap('Blues')(np.linspace(0.2, 0.7, len(expressions)))
    px = 1 / plt.rcParams['figure.dpi']
    fig, ax = plt.subplots(figsize=(600 * px, 600 * px))
    ax.pie(expressions, colors=colors, radius=12, center=(16, 16),
           wedgeprops={"linewidth": 1, "edgecolor": "white"}, frame=True, labels=expressions.index.values.tolist()
           , autopct="%.2f")

    ax.set(xlim=(0, 32), xticks=np.arange(1, 32),
           ylim=(0, 32), yticks=np.arange(1, 32))

    plt.show()
```

Figure 1: Pandas-Anweisungen für erste Übersicht

```
C:\Repositories\SoSe2022_Machine_Learning
(898, 35)
DOKOL     204
SAFAVI    199
ROTANA    166
DEGLET     98
SOGAY      94
IRAQI      72
BERHI      65
Name: Class, dtype: int64

Process finished with exit code 0
```

Figure 2: Output des entsprechenden Codes

Wie zu sehen ist enthält der Datensatz 898 Einträge mit 35 Merkmalen, eines der Merkmale ist dabei die Zielklasse als String, welche es zu encoden gilt. Die anderen 34 sind floats, die lediglich normalisiert werden müssen. Ebenfalls zu sehen ist die Verteilung der einzelnen Klassen, deren ungleiche Verteilung bereits in der Aufgabenstellung angedeutet worden war. Es ist für ein sinnvolles Training also nötig, die Daten stratifiziert zu splitten um selection bias beim Training zu vermeiden.(siehe nächste Seite)

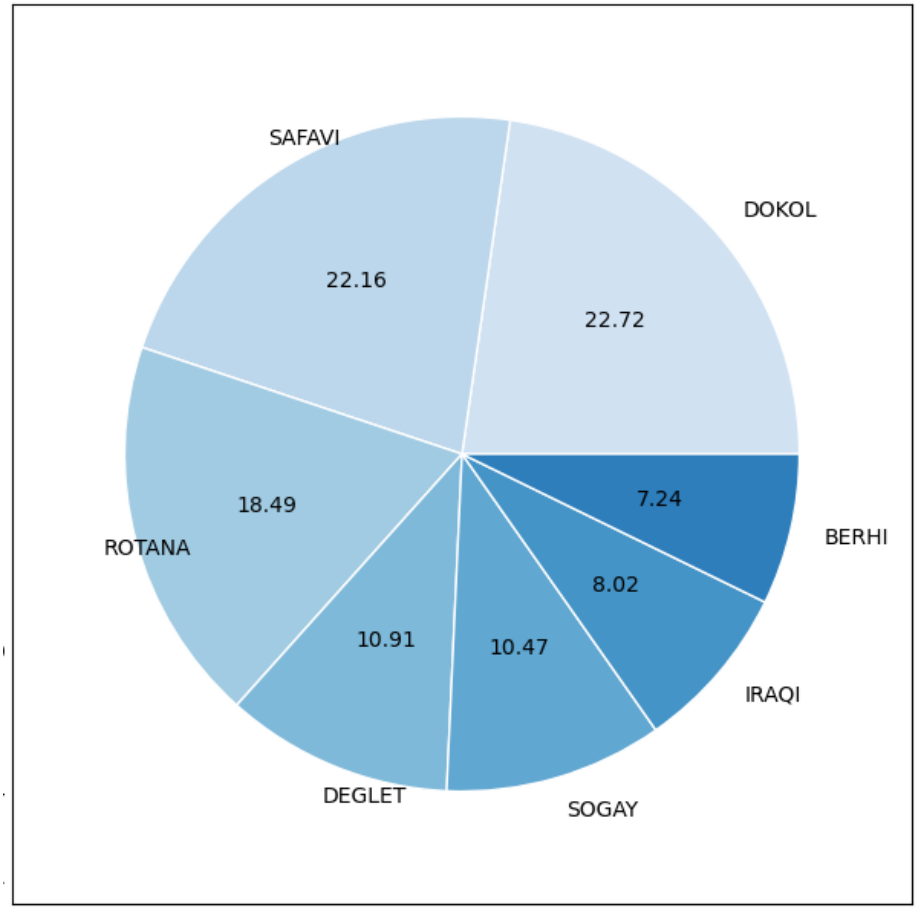

Figure 3: Prozentuale Verteilung der Klassen auf den Datensatz visualisiert in einem Kuchendiagramm, erstellt mit matplotlib [Hun07]

3 Preprocessing

Das Preprocessing wurde nicht in einem Layer sondern einfach im Code erledigt. Die Zielklassen wurden einfach in einem Dictionary gespeichert, dann die Klassen-Spalte als ein neue Objekt definiert und mit to categorical von keras.util auf das dictionary gemapt. Der Datensatz wurde ebenfalls als neues Objekt gespeichert und dann mit dem standard-Scaler von Scikit [Ped+11] normalisiert. Ebenfalls mit Scikit wurde der train-test-split getätigt. Es wurde ein 80-20 Split gewählt, ein random split festgelegt um die Ergebnisse reproduzierbar zu halten und vor allem die Stratifizierung der Daten auf True gesetzt, wie bereits im vorherigen Kapitel erwähnt.

```
def preprocessing(data): #Create Dict of target classes, map onto those, scale inputs, create test-train-split
    class_labels = {
        'DOKOL': 0,
        'SAFAVI': 1,
        'ROTANA': 2,
        'DEGLET': 3,
        'SOGAY': 4,
        'IRAQI': 5,
        'BERHI': 6
    }
    y = data["Class"]
    y = to_categorical(y.map(class_labels))
    x = data.drop(["Class"], axis=1)
    x = StandardScaler().fit_transform(x)

    # Splitting the Data, it is important to stratify according to classes to avoid sampling bias as there is u
    random_state = 42
    test_size = 0.2
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size, random_state=random_state,
                                                        stratify=y)
    return x_train, x_test, y_train, y_test
```

Figure 4: Preprocessing-Code

```python
def run_first_model(x_train, x_test, y_train, y_test): # First model for documentation purposes
    logdir_first_model = os.path.join(ROOT_DIR, 'Project_1', "logs", "first_model")

    first_model = tf.keras.Sequential()
    first_model.add(layers.Dense(512, activation='relu'))
    first_model.add(layers.GaussianDropout(0.3))
    first_model.add(layers.Dense(7, 'softmax'))
    first_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=["categorical_accuracy"])

    first_model.fit(x_train, y_train, epochs=25,
                    validation_data=(x_test, y_test), verbose=1,
                    callbacks=[tf.keras.callbacks.TensorBoard(logdir_first_model)])
    first_model.summary()
    first_model.save(os.path.join(ROOT_DIR, 'Project_1', "model", "first_model"))
```

Figure 5: Konfiguration des ersten Models im Quellcode

4 Modelbau

Mit der Sequential-Klasse wird das model modular aufgebaut, es wurde lediglich ein breiter relu-aktivierter layer erstellt und ein gaussianDropout layer dahinter geschaltet um overfitting zu vermeiden. Im späteren Verlauf wurde der Dropout- durch einen Noise-Layer ersetzt. Die Aktivierungsfunktion für den Kategorie-Layer ist dann eine softmax, um die eingegangenen Information in Wahrscheinlichkeiten einer Klassenzugehörigkeit umzuwandeln. Die Verlustfunktion und Metriken wurden dem Klassifizierungsproblem mit mehr als 2 Klassen entsprechend gewählt. Das compilierte model wird danach in 25 Epochen trainiert, und das tracking auf einfachen Fortschrittsbar gesetzt, da der Trainingsverlauf mit der callbacks-Funktion an einem definiertem Ort gespeichert wird. Diese Daten werden folgend mit Tensorboard dargestellt, das model selbst wird kurz zusammengefasst und dann in einem eigenem Ordner gespeichert, sodass zu einem späterem Zeitpunkt erneut darauf zurück gegriffen werden kann.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               17920
_________________________________________________________________
gaussian_dropout (GaussianDr (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 3591
=================================================================
Total params: 21,511
Trainable params: 21,511
Non-trainable params: 0
```

Figure 6: Zusammenfassung des erstellten models

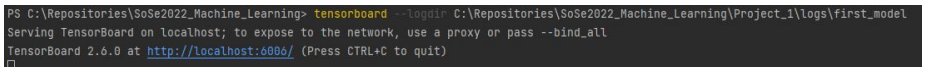

Figure 7: Start von Tensorboard im Terminal nach erfolgtem und geloggtem model-Training

5 Tensorboard

Die Logs des Trainings des ersten models können nun mit Tensorboard ausgelesen werden. Tensorboard wird über das Terminal aufgerufen und benötigt den Ordnerpfad der Logs, die ausgewertet werden sollen. Diese Logs werden dann automatisiert aufbereitet, grafisch dargestellt und die Ergebnisse über einen Browser auf einem für Tensorboard freigegebenem lokalem Port zugänglich gemacht.

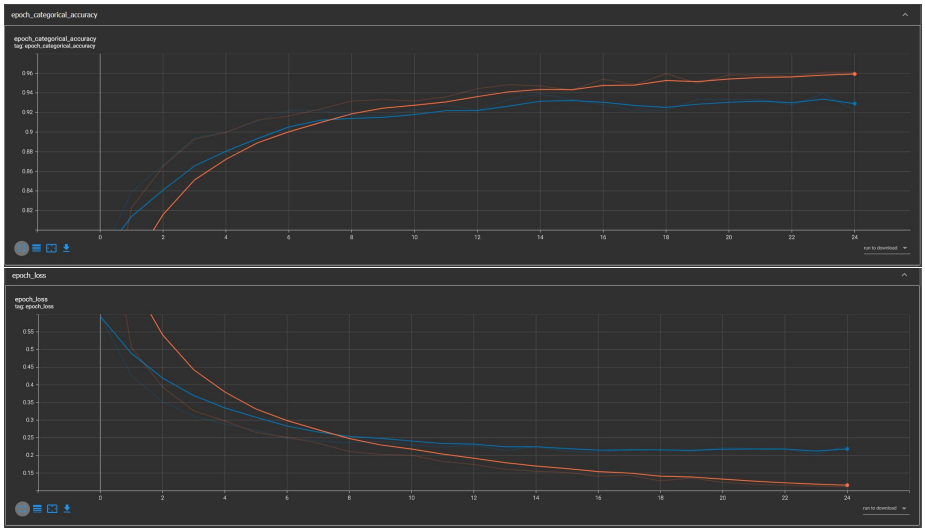

Figure 8: Von Tensorboard erstellte Plots der Trainingswerte: Kategoriale Genauigkeit und Epochenverlust

6 KerasTuner

Mit dem KerasTuner[OMa+19] kann ein Keras-Model nach optimalen Hyperparametern untersucht werden. Die vom Tuner benötigten Inputs bestehen in einem Funktionsaufruf zum Aufbau eines Keras-Models mit flexiblen Parametern, welche den Suchraum definieren innerhalb dessen die Optimierung stattfindet. Zunächst einmal muss in der buildModel-Methode entschieden werden, welche Hyperparameter überhaupt betrachtet werden sollen und welche fixiert sind. Gleichzeitig ist zumindest in diesem Fall auch zu beachten, dass die Anzahl der Layer, also die Architektur des Neuronalen Netzes fix gesetzt wird.

```python
def build_model(hp): #Builds a model for the KerasTuner to run over together with search-space definition. Architecture & Hyperparameter
    # ------------------- Variables, for ease of change
    layers_sizes = [128, 256, 512, ]
    activations = ['relu', ]
    noise = [0.2, 0.3, 0.4, ]
    losses = ["categorical_crossentropy", ]
    optimizers = ["adam", ]
    classifier = ["softmax", ]
    # ------------------- Architecture of the model itself (Which kind of layer where is fixed here)
    model = tf.keras.Sequential()
    model.add(layers.Dense(units=hp.Choice("units", layers_sizes), activation=hp.Choice("activation", activations),
                           input_shape=(34,)))
    model.add(layers.GaussianNoise(hp.Choice("noise", noise)))
    model.add(layers.Dense(units=hp.Choice("units2", layers_sizes), activation=hp.Choice("activation2", activations)))
    model.add(layers.GaussianNoise(hp.Choice("noise2", noise)))
    model.add(layers.Dense(7, activation=hp.Choice("classification", classifier)))
    model.compile(loss=hp.Choice("loss", losses), optimizer=hp.Choice("optimizers", optimizers), metrics=["mae", "acc"])
    return model
```

Figure 9: BuildModel Methode mit den vom KerasTuner gefundenen optimalen Parametern

Sobald die buildModel-Methode komplett ist, übergibt man dem KerasTuner diese, den Parameter auf den optimiert werden soll, weitere Anweisungen für die Anzahl und den Verlauf der Suche, wohin die Logs für Tensorboard, die den Verlauf der Suche dokumentieren, und wo das beste Modell gespeichert werden sollen, und ruft dann auf. Der KerasTuner läuft dann alle möglichen Konfigurationen durch (sofern diese nicht widersprüchlich sind weil beispielsweise geringere maximale Versuchsanzahl als Anzahl Konfigurationen), loggt und speichert hier im Repository das beste gefundene model.

Die Konfigurationen von Verlustfunktion categorical crossentropy, Optimierer

Figure 10: KerasTuner-Methode mit den genutzten Konfigurationen

adam und Aktivierungsfunktion softmax für zu den Kategorienlayer hat sich definitiv als die beste herausgestellt, sodass hier keine Optionen mehr angegeben werden, um den Suchraum für einen Test klein zu halten. Die Werte für Layergröße und Noise-Konstanten sind in den Test so weit fluktuiert, dass die Änderungen wahrscheinlich im Toleranzbereich zu finden sind, weswegen die drei immer wieder gefundenen Werte jeweils aufgeführt sind. Sobald der Tuner einmal durchgelaufen ist kann man sich selbstverständlich den gesamten Verlauf des Tunings auf dem Tensorboard nachvollziehen um zu verstehen, welche Kombinationen wie sinnvoll sind, auch wenn der Tuner selbst sehr viel erledigt - er kann nur in dem Raum suchen, den er übergeben bekommt und hier ist die Intuition des Entwicklers entscheidend. Das hier beschriebene Vorgehen, zunächst das Problem präzise zu formulieren, die Daten gründlich zu betrachten und dann ein angemessenes erstes model zu entwerfen, um im späteren Verlauf mit dem KerasTuner zu testen, in welche Richtung Hyperparameter optimiert werden können hat sich als sehr sinnvoll dargestellt und wird bei den nächsten Aufgaben sicherlich ähnlich gehandhabt werden.

7 Sicherung

Im Git-Archiv sind neben dem Code folgende Daten zu finden: Eine .yml mit allen benötigten Informationen um die virtuelle Umgebung mit conda problemlos aufsetzen zu können, Tensorboard-Logs des ersten models und des letzen Durchlaufes des KerasTuners unter /logs, sowie das erste und beste model, welche bei der aktuellen Konfiguration der Methoden einfach nur geladen und zusammengefasst geprintet werden. Der Verlauf des Trainings kann mit Tensorboard nachvollzogen werden, und sämtliche andere Funktionen können mit einem einfachen Aufruf von runproject1() in project1.py zu Test- oder Demonstationszwecken aufgerufen werden.

References

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Cho+15] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[OMa+19] Tom O'Malley et al. *KerasTuner*. 2019.

[Jef+20] Jeff Reback et al. *pandas-dev/pandas: Pandas 1.0.3*. 2020. DOI: 10.5281/ZENODO.3715232.

[Kok+21] Murat Koklu et al. "Classification of Date Fruits into Genetic Varieties Using Image Analysis". In: *Mathematical Problems in Engineering* 2021 (2021), pp. 1–13. ISSN: 1024-123X. DOI: 10.1155/2021/4793293.