Keras CNN Alex

Das Projekt NN zur Schriftartenerkennung setzt auf CNN. Grundlage bilden dabei Bilddaten, die ähnlich wie bei anderen NN eingelesen werden. Durch eine Ordnerstruktur werden verschiedene Bilder mit Schriften einem Label zugeordnet. Eine Beispielhafte Struktur sieht wie folgt aus:

Arial

* Bild 1
* Bild 2

Calibri

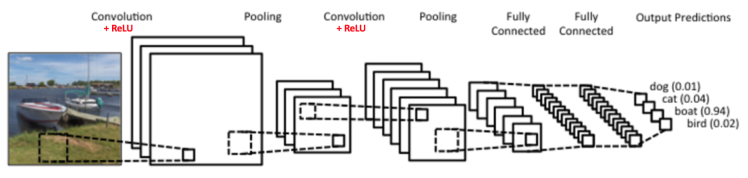
* Bild 1

…

…

Die einzelnen Bilder bestehen dabei aus Texten in der jeweiligen Schriftart. Der Text wird variabel über ein Skript erzeugt. Der genaue Aufbau wird in Kapitel [] erklärt.

Der Aufbau eine CNN unterscheidet sich durch die Möglichkeit mehrdimensionale Daten zu verarbeiten. Dazu werden Filter verwendet, die Ausschnitte der Daten zeigen. Diese Ausschnitte werden wiederum verglichen und als Features zur Merkmalerkennung herangezogen. Folgendes Bild zeigt einen Überblick über ein CNN.



<https://ujwlkarn.files.wordpress.com/2016/08/screen-shot-2016-08-07-at-4-59-29-pm.png?w=748>

Als Eingabe erwartet das Netz ein Bild, welches im ersten Schritt durch drei Filter läuft. Jeder Filter erzeugt dabei ein gleichgroßes Ausgabebild in dem jeder Pixel durch die Summe des Filters ersetzt wird. Mit einem Poolinglayer werden anschließend die einzelnen Pixelfeatures auf Bildausschnitte reduziert, was die Feature Matrizen verkleinert (im Bild durch eine Reduzierung von Breite und Höhe sichtbar). Anschließend wiederholen sich die Schritte des Netzes zweimal, bis die Daten auf einen eindimensionalen Vektor reduziert werden. Das gezeigte Netz wird durch zwei Fully Connected Layer ergänzt welche jedes komprimierte Feature mit einem Ausgabeneuron verknüpft. Durch die Verknüpfung aller Ein- und Ausgabeneuronen entstehen Wahrscheinlichkeiten für verschiedene Classifier, die am Ende den vier Classifiers dog, cat, boat und bird zugeordnet sind.

Für die Applikation zur Schriftartenerkennung wurde Python mit Keras und Tensorflow ausgewählt. Alle benötigten Frameworks sowie die Installationsanleitungen können in [] nachgelesen werden.

Der Aufbau der Software besteht aus mehreren Modulen. Anhand der Ordnerstruktur sollen alle wesentlichen Funktionalitäten erklärt werden.

Ordner:

dataloader -> enthält alle wichtigen Skripte um Daten aus Ordnerstrukturen zum Laden und Unterteilen in Test und Trainingsdaten. Dabei werden auch Möglichkeiten zum One-Hot Encoding sowie dem Import als flachen Vektor geboten.

modelpipe -> dieser Ordner enthält die Vereinfachung der Hintergrund Prozesse. So werden die wichtigen Funktionalitäten: Daten laden, Netz trainieren, Netz prüfen, Daten zuordnen sowie das Model laden vereinfacht. So können auch mehrere Netze geladen und in einem Batchprozess trainiert werden.

Model -> in diesem Ordner sind die definierten Modelle sowie deren Ergebnisse zu finden.

Skripte:

main.py -> dieses Skript startet den Trainingsprozess, indem die Trainingsdaten geladen und mehrere Modelle trainiert werden.

state.py -> dieses Skript erzeugt eine Statistik aus vorhandenen CSV Daten eines Models. Zusätzlich wird zu jedem Modell ein passendes Abbild erstellt, das die Input- und Output Layer zeigt.

evaluate.py -> dieses Skript führt eine Evaluierung der Modelle gegen die übergebenen Datensetzte aus.

Folgende Programmierbeispiele zeigen die Anwendung der Pipe:

1) Trainieren eines Models:

from modelpipe import pipe

datapipe = pipe.Pipe(data\_path = "../../images/Dataset\_1", train\_size=0.6)

datapipe.load\_data(flatten=0, print\_out=1)

result = datapipe.run(model\_name = "models/model\_01", epochsize = 300, batch\_size = 100)

Im Beispiel wird ein Pipe Objekt mit dem Datenpfad und dem Prozentualen Anteil an Trainingsdaten erstellt. Anschließend werden die Daten über den Befehl load\_data als mehrdimensionaler Array geladen. Anschließend wird das Model "model/model\_01.py" mit 300 Epochen und einer Batchsize von 300 trainiert. Das trainierte Model wird im Ordner "model/model\_01/" mit Metainformationen gespeichert.

2) Evaluieren eines Models:

from modelpipe import pipe

datapipe = pipe.Pipe(data\_path = "../../images/Dataset\_1", train\_size=0.6)

datapipe.load\_data(flatten=0, print\_out=1)

result = datapipe.eval(model\_name = "models/model\_0"+str(i), batch\_size = 100)

Ähnlich wir im ersten Beispiel wird hier ein Pipe Objekt erstellt und Daten geladen. Anschließend werden die Daten über die eval Funktion evaluiert. Dabei wird der Pfad zum Model benötigt das im ersten Beispiel erstellt wurde. Es werden automatisch die model.h5 und model.json im jeweiligen Ordner geladen. Das result Objekt ist ein Array der an der ersten Stelle den Fehlerwert und an der zweiten Stelle die Genauigkeit in Prozent enthält.

3) Erstellen von Bildern zur Analyse des Models

from dataloader import plot

plot.plot\_csv\_multipath("models/model\_01/plot.csv", figure="Model\_1").savefig("model1.png")

Mit diesem Codesnippet wird die plot.csv aus dem Model Ordner geladen und geplottet. Anschließend wird der Plot als model1.png gespeichert.

from modelpipe import pipe

from keras.utils import plot\_model

model = pipe.Pipe().get\_model(model\_name="models/model\_01")

plot\_model(model, to\_file='model.png', show\_layer\_names=True, show\_shapes=True)

Das zweite Code Snippet zeigt das Erstellen einer Model Übersicht, dabei werden die Layer des Models geplottet. Hier wird zusätzliche Software benötigt.

4) Predictions

from modelpipe import pipe

imgArray = "Your Images go here"

predictions = pipe.Pipe().predict(model\_name="models/model\_01", imgs=imgArray)

Dieses Beispiel zeigt wie mehrere Bilder über die predict Methode zugeordnet werden. Dazu muss der Pfad zum Model das im ersten Beispiel erstellt wurde angegeben werden. Die Bilder werden als imgArray mitgegeben. Im Beispiel wurden keine Bilder geladen.

5) Erstellen eines Models

model.add(Conv2D(8, kernel\_size=(3, 3), activation='relu', input\_shape=(40,1200,3)))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(16, kernel\_size=(3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(24, kernel\_size=(3, 3), activation='sigmoid'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256, activation='relu'))

model.add(Dense(len(classes), activation='softmax'))

model.compile(loss=keras.losses.mean\_squared\_error, optimizer="rmsprop", metrics=['accuracy'])

In diesem letzten Beispiel wird ein Model in "models/model\_01.py" dargestellt. Das Model verfügt automatisch über ein model Objekt, ein classes Objekt, sowie verschiedene Keras importe. Das Skript wird zur Laufzeit von der run Methode geladen und kann so die vor instanziierten Objekte und Referenzen nutzen. So ist eine optimale Trennung von Abhängigkeiten möglich was es einfach macht Modelle auszutauschen.