Fontinator - Font style detection in Python

Github Repo

Contents:

How to Setup:

Install anaconda Run commands:

1. conda create -n IDA python=3.6
2. activate IDA
3. conda install numpy pandas pillow matplotlib scipy scikit-learn theano h5py graphviz
4. pip install pydot-ng
5. conda install keras -c conda-forge
6. conda install opencv -c conda-forge

if you get an error message containing tensorflow you are using the wrong backend, check out to switch the backend

Tensorflow is currently not supported on 3.6 Easy way:

1. conda create --name tensorflow python=3.5
2. activate tensorflow
3. pip install tensorflow

- or pip install tensorflow-gpu

1. conda install numpy pandas pillow matplotlib scipy scikit-learn theano h5py
2. conda install keras -c conda-forge
3. conda install opencv -c conda-forge

It also supports theano

To switch backends for keras:

1. Edit the jsonfile on your home directory (.keras) folder
2. The Json looks something like this: { "floatx": "float32", "epsilon": 1e-07, "backend": "theano", "image_data_format": "channels_last" }

change the backend to your prefered backend You can also follow this tutorial.

Data Generator

Der folgendes Text bezieht sich auf das Modul unter dem Pfad "Fontinator/DataGenerator". Das Programm-Modul ermöglicht die Erzeugung von Trainings- und Testbildern für den Fontinator. Die Bilder bestehen aus einem einzeiligen Text mit unterschiedlichen Schriftstilen (Fonts). Der erzeugte Text wird zufällig erzeugt. Für eine nähere Beschreibung siehe Präsentationfolien (S. 5-8)

Abhängigkeiten

Das Modul besitzt die folgenden externen Abhängigkeiten:

- PIL

Konfiguration:

Die Konfiguration erfolgt in der Datei 'config.py'. Es kann beispielsweise die Zahl der zu erzeugenden Bilder oder die Fontgröße festgelegt werden. Die zu verwendenden Fonts werden standardmäßig aus dem Ordner 'fonts' gelesen. Die Fonts müssen im .ttf Format vorliegen. Die erzeuten Bilderdaten werden standardmäßig im Ordner 'images' abgelegt. Im Ordner "text_res" befinden sich einige Textdateien, welche die Wörter für die Zufallssätze enthalten.

Ausführung

Zum Starten muss das Skript "createImages.py" ausgeführt werden. Es werden alle Fonts eingelesen und die Trainings- und Testbilder erzeugt.

Hilfsklassen

In dem Ordner "libs" befinden sich alle vom Data Generator verwendeten Hilfsklassen.

WordDict

Die Klasse "WordDict" ermöglicht die Erzeugung von Zufallssätzen. Die Zufallswörter können aus einer Textdatei eingelesen werden.

Fontinator - Feature Engineering

Das Unterprojekt Fontinator Feature Engineering identifiziert, wie das Gesamtprojekt Fontinator, die in einem Bild verwendete Schriftart. Dabei setzt das Unterprojekt auf klassische Ansätze, wie die

Extraktion von bestimmten Features, welche für die anschließende Klassifikation der Schriftart benötigt wird.

Nachfolgend werden die Abhängigkeiten sowie die Verwendung des Systems aufgezeigt. Anschließend erfolgt eine kurze Beschreibung des Systemkonzepts samt Funktionsweise.

Abhängigkeiten

Im Folgenden werden die externen Abhängigkeiten anhand der Entwicklungsumgebung kurz aufgezeigt:

- Python 3.6
- OpenCV 3.2
- Numpy 1.12.1
- PIL 4.1.1
- scikit-learn 0.18.1

Verwendung

Die folgenden Skripte bilden die Hauptkomponenten des Programms:

- trainer.py - Erstellen der Trainings- und Labeldaten samt Training des Klassifikators
- test.py - Test des Programms mit vorgegebenen Bilddaten
- Fontinator.py - Schriftartenklassifikation eines Bildes

trainer.py

Zunächst erfolgt das Training des Klassifikators. Dazu wird das Skript trainer.py ausgeführt. Hierzu werden alle Schriftarten, welche als .ttf - Dateien, in dem Ordner `Fontinator/DataGenerator/fonts` vorliegen, herangezogen. Das verwendete Glyphenset kann im Skript trainer.py mithilfe der Variable `TRAIN_CHARS` modifiziert werden. Die ezeugten Parameter des Klassifiaktors werden in der Datei `classie.pickle` gepseichert. Die Datei `labels.pickle` enthält die zugehörigen Labels und erlaubt somit die spätere Zuordnung eines Klassifikationsergebnisses zu dem entsprechenden Namen der Schriftart.

test.py

Nach dem Training kann das System mithilfe des Skriptes test.py überprüft werden. Hierbei werden in der aktuellen Konfiguration die Bilder des Ordners `Fontinator/TestSets/images/Dataset_1` verwendet. Der Pfad zum Datenset kann in dem Skript test.py mithilfe der Variable `images_path` angepasst werden. Dabei muss die im folgenden Dargestellte Ordnerstruktur eingehalten werden.

```
images_path
├── FORTE
│   ├── forte_0.png
│   ├── forte_1.png
│   ├── ...
```

```
├── arial
│   ├── arial_0.png
│   ├── arial_1.png
│   ├── ...
├── ...
```

Wichtig ist dabei, dass die Ordner, welche die Bilder enthalten entsprechend der zugehörigen Schriftart benannt sind.

Dabei wird für jedes Bild die Schriftart, welche erkannt wurde, ausgegeben. Zusätzlich wird jede erkannte Glyphe des Bildes individuell einer Schriftart zugeordnet. Wurden alle Bilder untersucht, so wird die Genauigkeit der Ergebnisse für die einzelnen Glyphen selbst, sowie für das Gesamtergebniss der Bilder ausgegeben.

Fontinator.py

Ist das Training abgeschlossen kann das Skript -Fontinator.py- für die Klassifikation der Schriftart eines Bildes verwendet werden. Dazu wird das Skript mit Python aufgerufen und der Pfad zum Bild übergeben.

```
python Fontinator.py /Pfad/zum/Bild
```

Das Skript liefert eine absteigende Liste mit den Wahrscheinlichkeiten der erkannten Schriftarten.

Systemkonzept

Das Verarbeitungskonzept kann in mehrere Schritte unterteilt werden, welche für jede Verarbeitung eines Bildes durchgeführt werden:

- Extraktion einzelner Glyphen aus dem Text
- Extraktion von Features aus den einzelnen Glyphen
- Klassifikation anhand der extrahierten Features

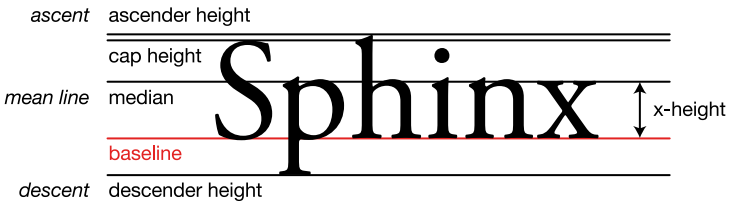
Da die Bilddaten erzeugt werden und damit einen idealen Datensatz darstellen, können auf größere Vorverarbeitungsschritte zur Bildverbesserung verzichtet werden.

Zusätzlich ist das Training des Klassifikators ein weiterer Schritt, welcher einmalig für die Verwendung des Programms durchgeführt werden muss.

Glyphenextraktion

Zunächst wird der im Bild vorhandene Text in einzelne Glyphen zerlegt. Eine Glyphe stellt dabei einen Buchstaben, eine Ziffer, ein Satzzeichen oder ähnliche Textartefakte dar. Hierzu werden zunächst die typografischen Eigenschaften nach dem Vierliniensystem berechnet. Die Linien beschreiben die Ober- und Unterkante von Kleinbuchstaben ohne Oberlänge (z.B. m, o, a). Eine weitere Linie beschreibt die Oberkante von Großbuchstaben, sowie von Kleinbuchstaben mit Oberlänge (l, ö, i). Die vierte Linen beschreibt die Unterkannte von Buchstaben mit Unterlänge (z.B.

g, p, q). - Wikipedia Liniensystem



-Bildquelle: [Wikipedia](https://en.wikipedia.org/wiki/Baseline_(typography)-

Anschließend werden die gefundenen Linien für die eigentliche Glyphenextraktion verwendet. Hierbei werden zunächst zusammenhängende Artefakte mithilfe der OpenCv Methode findContours identifiziert. Die gefundenen Artefakte stellen dabei entweder eine gesamte Glyphe oder einen Teil einer Glyphe dar. Die gefundenen typografischen Linien helfen dabei, einzelne Artefakte wie z.B. i-Punkte dem zugehörigen Artefakt zuzuordnen und somit die Glyphe zu vervollständigen. Die vervollständigten Glyphen werden anschließend an den nächsten Verarbeitungsschritt, die Featureextraktion, weitergereicht.

Featureextraktion

Bei der Featureextraktion werden markante Merkmale (Features) aus den einzelnen Glyphen ermittelt. Entscheidend ist es hierbei Features zu wählen, welche die individuellen Charakteristiken der Schriftarten beschreiben und somit eine Differenziation zwischen diesen erlauben. Als Startpunkt wurden hierbei einige Features nach Andrew Bray implementiert und durch weitere eigens definierte Features ergänzt. Hierbei werden die im folgenden vorgestellten Features verwendet:

- Anzahl dunkler Pixel
- Länge des Umfangs aller Komponenten einer Glyphe
- Anzahl der Pixel der skelletierten Glyphe
- Mittlere horizontale Position alle dunklen Pixel
- Mittlere vertikale Position aller dunklen Pixel
- Anzahl der horizontalen Kanten im Bild
- Anzahl zusammenhängender Komponenten
- Anzahl von Löcher (z.B. innerer Kreis im Buchstaben O)
- Horizontale Varianz der Position aller dunklen Pixel
- Vertikale Varianz der Position aller dunklen Pixel

Die ermittelten Features werden anschließend mithilfe von Merkmalen, welche die Größe der Glyphen beschreiben, normiert und sind somit weitestgehend unabhängig von der Schriftgröße. Diese Features bilden anschließend den Featurevektor, welcher für die Klassifikation verwendet wird.

Klassifikation

Mithilfe des gefundenen Featurevektors erfolgt anschließend die Klassifikation der Schriftart. Um

den Einfluss aller Features gleichzuhalten, wird der Featurevektor auf den Bereich Null bis Eins normiert.

Für die Klassifikation kommt ein One-Nearest-Neighbor Klassifikator von scikit-learn zum Einsatz.

Hierbei wird jede Glyphe einzeln klassifiziert und anschließend mitihlfe aller Glyphen eines Textes eine Mehrheitsabstimmung für die Schriftart durchgeführt.

Training

Das Training des Klassifikators erfolgt mithilfe von generierten Bilder der einzelnen Buchstaben einer jeden Schriftart. Hierbei wird somit für ein einfacheres Labeling der Trainingsdaten auf die Glyphenextraktion verzichtet. Anschließend erfolgt die Featureextraktion, wie bereits oben beschrieben. Mithilfe der erzeugten Bild- und Labeldaten erfolgt das Training des Klassifikators, sowie die Berechnung der für die Normalisierung notwendigen Parameter. Anschließend werden alle Parameter für die spätere Verwendung abgespeichert.

Auswertung

Abschließend kann das Programm auf seine Funktionsweise getestet werden. Dieses geschieht entweder über das Skript `Fontinator.py` oder über das Test-Skript `test.py` (wie bereits oben beschrieben). Aus dem jeweiligen Bild werden folglich die Glyphen extrahiert, sowie daraus die Features und der Featurevektor ermittelt. Der Klassifikator vergleicht die gefundenen Features mit den bereits antrainierten Features und weist jedem Bild eine Schriftart zu.

Neural Networks for font-style recongnition

Der folgendes Text bezieht sich auf das Modul unter dem Pfad "Fontinator/NeuralNet/Oli". Dieses ermöglicht die klassifizierung der Bilder mithilfe von Neuronalen Netzen. Dafür werden zwei Ansätze untersucht:

- Fully Connected Neural Networks (Siehe Präsentationsfolien S. 21-24)
- Convolutional Neural Networks

Abhängigkeiten

Das Modul besitzt die folgenden externen Abhängigkeiten:

- numpy
- pandas
- pillow
- matplotlib
- scipy

- scikit-learn
- theano
- keras
- h5py
- graphviz
- pydot-ng
- keras
- opencv

In der Datei "Fontinator/Documents/HOW_TO_SETUP.md" wird die Installation der benötigten Python Umgebung erklärt.

Hierfür wird Anaconda verwendet.

Ausführbare Skripte

Dieser Abschnitt beschreibt alle ausführbaren Skripte im Modul.\ Alle ausfürbaren Skripte enthalten am Anfang einen kurzen Bereich zur Konfiguration.

Alle Skripte informieren den Benutzer in der Console über die aktuellen Arbeitsschritte.

Die Skripte können direkt in der Konsole aufgerufen werden => "python ".

trainNN

Das Skript "trainNN.py" startet das Trainieren eines Fully Connected Neural Networks.

evaluateNN

Das Skript "evaluateNN.py" erlaubt die Evaluation eines mit "trainNN.py" erzeugten Models.

trainCNN

Das Skript "trainCNN.py" startet das Trainieren eines Convolutional Neural Networks.

evaluateNN

Das Skript "evaluateCNN.py" erlaubt die Evaluation eines mit "trainCNN.py" erzeugten Models.

Abgespeicherte Modelle

Download

Da die vortrainierten Keras-Modelle eine Größe von 200-500mb besitzen, sind diese nicht im Repository enthalten. Die vortrainierten Modelle können unter folgendem Link gefunden werden. Die Keras-Modelle sollten in dem Ordner "Fontinator/NeuralNet/SavedModels" abgespeichert werden.

Aufbau

Für jedes Model gibt es einen eigenen Unterordner (z.B. LT2, CNN_RAND_80). Für jedes Model wurden die Struktur und die Gewichtungen des Netzes abgespeichert. Zusätzlich sind noch weitere Metadaten verfügbar:

- Ein Diagram, das anzeigt wie sich die Accuracy beim Trainieren verhalten hat.
- SVG- und PNG-Grafiken, die den Aufbau des Models zeigen.
- JSON-Datei, die das Label Mapping gespeichert hat
- CSV-Datei, die jegliche Informationen über das Training des Models enthält (epoch,val_loss,val_acc,loss,acc,tdiff)

Hilfsklassen

Im Ordner "libs" befinden sich alle von diesem Modul verwendeten Hilfsklassen.

ModelSerializer

Die Klasse übernimmt das Abspeichern und Laden eines Keras-Models von der Festplatte.

ImageLoader

Die Klasse unterstützt beim Laden aller Bilddaten.

TrainingLogger

Die Klasse protokolliert alle wichtigen Informationen, die beim Training eines Keras-Models anfallen. Die protokollierten Informatioen werden in einem Pandas-Dataframe gespeichert. Dieses kann auch in einer CSV-Datei abgespeichert werden. Außerdem können auch direkt einige aussagekräftige Diagramme erzeugt werden.

ProcessingPipeline

Die Klasse managt den kompletten Lifecycle eines Keras-Models.

Dazu gehören:

- Laden der Bilder und Labels
- Preprocessing
- Laden eines Models aus einer Datei
- Trainieren eines Models
- Abspeichern von Struktur und der Gewichtungen eines Models
- Prediction
- Evaluierung von Bildern

Preprocessor

Enthält mehrere Preprocessor-Klassen, welche die Vorverarbeitung der Bilder für das Neuronale Netz übernehmen.

Der "SimplePreprocessor" übernimmt die Vorverarbeitung eines Bildes für ein Fully Connected NN.

Dazu gehören die Schritte:

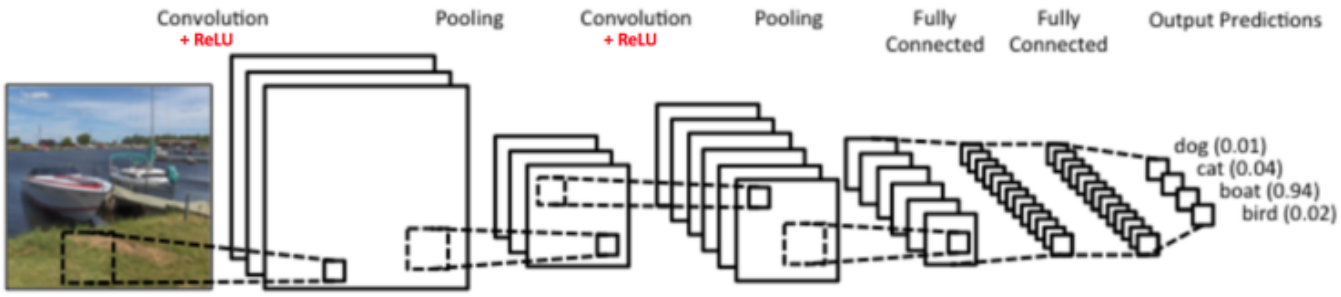
- Binarisierung
- Flattening

Der "ConvPreprocessor" übernimmt die Vorverarbeitung eines Bildes für ein Convolutional NN. Dabei wird das Bild nur binarisiert.

Convolutional Neural Network font-style recongnition

CNN

The structure of a CNN differs by the possibility of processing multi-dimensional data. For this purpose, filters are used which show excerpts of the data. These sections are again compared and used as feature recognition features. The following figure shows an overview of a CNN.

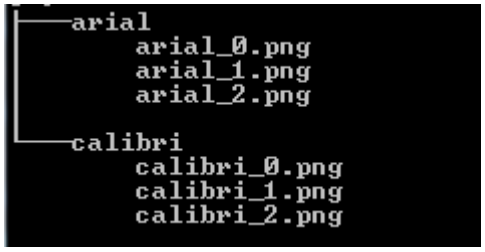


As input, the network expects an image, which runs through three filters in the first step. Each filter produces an equal-sized output image in which each pixel is replaced by the sum of the filter. With a poolinglayer the individual pixel features are then reduced to image excerpts, which reduces the feature matrices (visible in the picture by a reduction in width and height). Subsequently, the steps of the network are repeated twice until the data is reduced to a one-dimensional vector. The network shown is supplemented by two Fully Connected Layers, which link each compressed feature to an output euron. By linking all input and output eurons, probabilities are created for different classifiers, which are assigned to the four classifiers dog, cat, boat and bird.

Basics

The project NN for fonts recognition is based on CNN. It uses image data, which is read in the same way as for other NNs. A folder structure maps different images with fonts to a label. An example

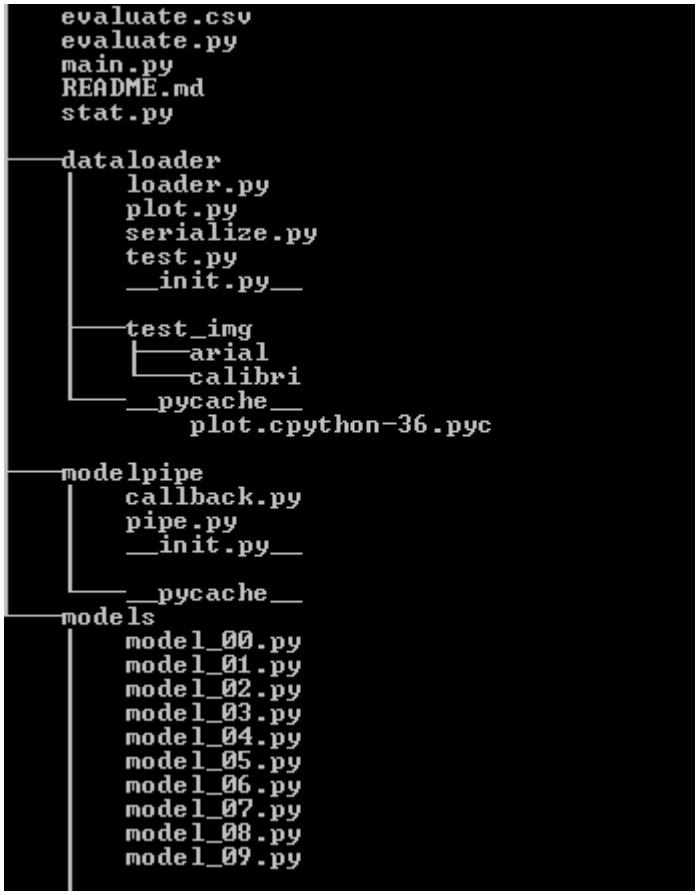
structure is as follows:



The individual images consist of texts in the respective font. The text is generated variably via a script. The exact structure is explained in chapter [Data Generator].

Python was selected with Keras and Tensorflow for the font recognition application. All necessary frameworks as well as the installation instructions can be read in [How to Setup]. The structure of the software consists of several modules. All essential functionalities are to be explained using the folder structure.

Structure



Folders

- dataloader: contains all important scripts for data from folder structures for loading and dividing in test and training data. It also offers possibilities for one-hot encoding and import as a flat vector.
- modelpipe: this folder contains the simplification of the background processes. Thus, the important functionalities: load data, train network, check network, assign data as well as simplify

loading the model. Thus, several networks can be loaded and trained in a batch process.
- model: in this folder are the defined models and their results.

Scripts:

- main.py: this script starts the training process by loading the training data and training several models.
- state.py: this script generates a statistics from existing CSV data of a model. In addition, a suitable image is created for each model that shows the input and output layers.
- evaluate.py: this script performs an evaluation of the models against the given data sets.

Pipeline

This Code is designed for easy use. All scripts are tightened togehter in a pipeline. The following Code shows any option provided by this pipe:

```
# include data pipe
from modelpipe import pipe
import numpy as np
#create pipe
#data_path => path to the training and testdata
#   data must be provided in a folderstructure
#   data
#     - label 1
#       - image1
#       - image2
#     - label 2
#train_size => the size of test (1-x) and trainings data (x)
#
#this call creates an pipe object and sets up variables
datapipe = pipe.Pipe(data_path = "E:\images\pirates", train_size=0.6)
#load data
#flatten => provides a flattend datastructure, where all images are provided in a
single array instead of a [h][w][c] #array (width, height, channel)
#print => enabels or disables logging
#test_x and test_y contain the testdata (x) and the testlabels (y)
#all other data gets saved in the pipe for later use
test_x, test_y = datapipe.load_data(flatten=0, print_out=1)
#train model
#model_name => path to a keras model descriped in [] without .py
(models/model_01.py gets loaded)
#the created model gets saved in a folder with the model_name path.
(models/model_01 folder contains the created data)
#epochsize => amount of epochs run by the pipe
#batchsize => amount of images in one batch with same weigth
#result contains the loss and metricies from the model run with test data.
result = datapipe.run(model_name = "models/model_01", epochsize = 3000, batchsize =
100)
print(result)
#evaluate model (to use with existing models)
#takes optional parameters
#model_name => model_name defaults to the data in the run methode
#test_x => test_x data defaults to the data loaded in the load_data methode
```

```
#test_y => test_y data defaults to the data loaded in the load_data methode
#batch_size => defaults to batchsize defined in the run methode or undefined
result = datapipe.eval()
print(result)
#predict
#model_name => model_name defaults to the data in the run methode
#imgs => images that should get predicted as np.array
#one_hot => set labels to one_hot encode, default is percentages
result = datapipe.predict(np.array([test_x[0]]), one_hot=1)
print(result)
```

Visualization

```
# include plot options
from dataloader import plot
#read data from csv and plot it. The csv gets created in the model_name folder and
contains the process per epoch data.
#plots loss, percentage and time per epoch charts
plot.plot_csv_multipath("models/model_01/plot.csv")
#plot all data in one image
plot.plot_csv("models/model_01/plot.csv")
```

Minimal Code Samples

1) Training

```
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)
```

The example creates a pipe object with the data path and the percentage of training data. The data is then loaded as a multi-dimensional array using the load_data command. Then the model "model / model_01.py" is trained with 300 epochs and a batch size of 100. The trained model is stored in the "model / model_01 /" folder with meta information.

2) Evaluation

```
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_01", batch_size = 100)
```

Similarly, in the first example, a pipe object is created and data is loaded. The data are then evaluated via the eval function. This requires the path to the model created in the first example. The model.h5 and model.json are loaded automatically in the respective folder. The resulting object is an array containing the error value in the first place and the accuracy in percent in the second

place.

3) Creating images to analyze the model

```
from dataloader import plot
plot.plot_csv_multipath("models/model_01/plot.csv",
figure="Model_1").savefig("model1.png")
```

With this snippet code, the plot.csv is loaded from the model folder and plotted. The plot is then saved as model1.png.

```
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)
```

The second code snippet shows the creation of a model overview, the layers of the model are plotted. Additional software is required.

4) Predictions

```
from modelpipe import pipe
imgArray = "Your Images go here"
predictions = pipe.Pipe().predict(model_name="models/model_01", imgs=imgArray)
```

This example shows how multiple images are assigned using the predict method. To do this, the path to the model created in the first example must be specified. The images are given as imgArray. In the example, no images were loaded.

5) Creating a model

```
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 this last example, a model is displayed in "models / model_01.py". The model automatically has a model object, a class object, as well as various keras importe. The script is loaded at runtime by the

run method and can use the objects and references instanced before. Thus, an optimal separation of dependencies is possible which makes it easy to exchange models.