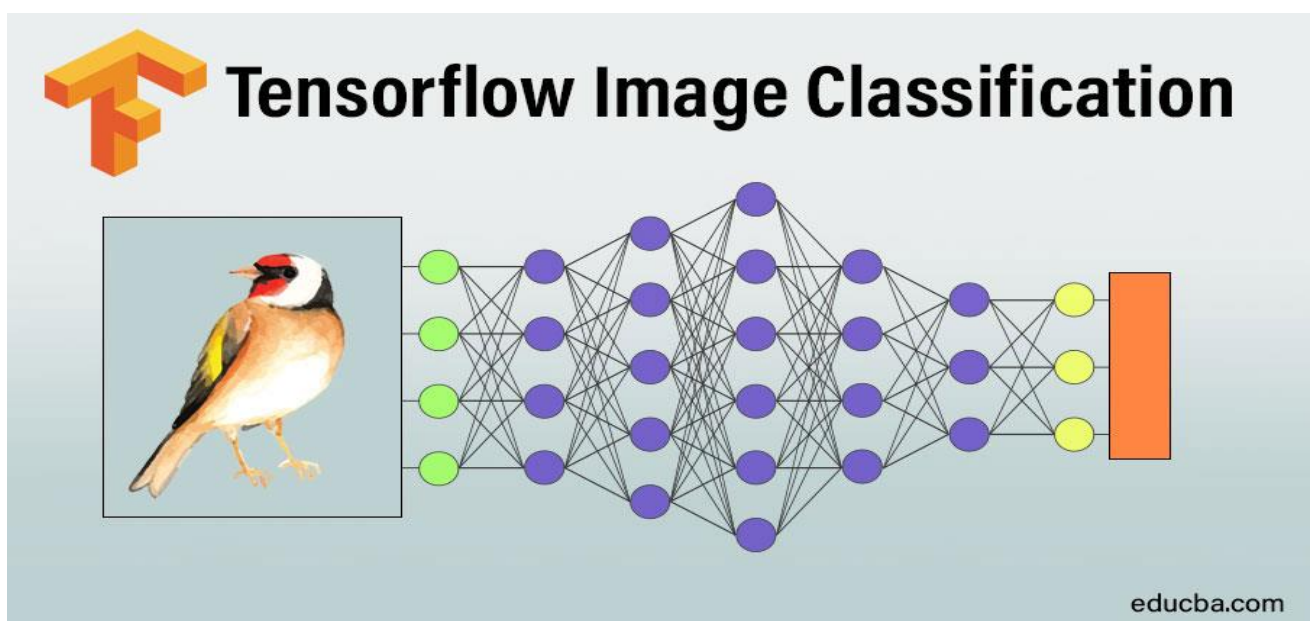


D5.1.2: Advances in Intelligent Systems (ISS)

Von: Justin Bergmann

1. Themenwahl
2. Was macht Image Classification
3. Code
 - 3.1 = Import Libraries
 - 3.2 = Loading CIFAR-100
 - 3.3 = Adding augmentation
 - 3.4 = Building Model Architecture
 - 3.5 = Compiling the model
 - 3.6 = Displaying model summary
 - 3.7 = Training the model
 - 3.8 = Visualizing learning curve of model
 - 3.9 = Printing Test Accuracy
 - 3.10 = Saving the model
 - 3.11 = Making Predictions
 - 3.12 = Predicted and actual classes
4. Probleme / Abschlussworte
5. Quellen



1. Themenwahl

Gesichts-/Handerkennung:

Ursprünglich hatte ich mich für das Thema der Gesichts-/Handerkennung entschieden, da mich die Anwendung von Bildverarbeitungsalgorithmen zur Identifizierung und Analyse menschlicher Merkmale interessiert.

Jedoch stellte ich fest, dass ich bei der Erforschung dieses Themas nicht so recht vorankam. Trotz meines Interesses und meiner Begeisterung stieß ich auf einige Herausforderungen, die mich daran hinderten, meine Ziele zu erreichen.

Die Strukturen der Gesichts-/Handerkennung waren leicht darzustellen allerdings lagen meine Probleme darin, das Model zu Skalieren sowie zu trainieren.

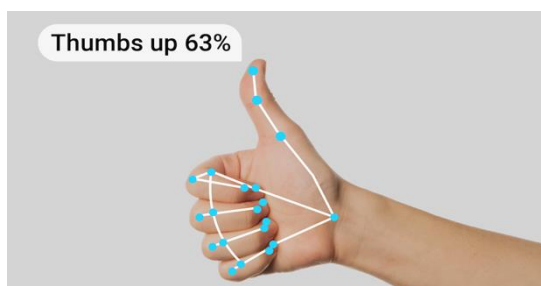


Image Classification:

Schließlich lenkte ich meine Forschungsinteressen auf das Gebiet der Bildklassifizierung. Dieser Schritt wurde vor allem durch die vielfältigen Anwendungsmöglichkeiten dieses Themas motiviert, die sich über verschiedene Bereiche erstrecken. Als Beispiel die Erkennung der verschiedenen Muster in Bildern.

Zudem begeisterte mich die Fülle an verfügbaren Ressourcen und Werkzeugen, die es ermöglichen, innovative Algorithmen zu entwickeln. Besonders faszinierte mich die Idee, durch automatisierte Bildanalyse und Klassifizierung Muster und Merkmale in den Bildern zu identifizieren, um zu sehen, ob mein Modell die richtigen Entscheidungen fällt.



2. Was macht Image Classification:

Bildklassifizierung ist ein Bereich der künstlichen Intelligenz und maschinellen Bildverarbeitung, der sich mit der automatischen Zuordnung von Bildern zu vordefinierten Kategorien oder Klassen befasst. Das Ziel ist es, ein Modell zu erstellen, das in der Lage ist, Bilder zu analysieren und richtig zu klassifizieren, ohne dass menschliche Intervention erforderlich ist. Dies wird oft durch den Einsatz von Deep-Learning-Techniken erreicht, insbesondere durch Convolutional Neural Networks (CNNs).

1.Punkt:

Vorbereitung der Daten:

Wir sammeln viele Bilder von verschiedenen Dingen, die wir klassifizieren möchten, z.B. wie in meinem Projekt mit CIFAR-100.

2.Punkt:

Bei der Entwicklung eines CNN-Modells werden verschiedene Schichten miteinander verbunden, um eine effektive Bildklassifizierung zu ermöglichen. Diese Schichten haben jeweils spezifische Aufgaben:

Faltungsschichten:

Diese Schichten extrahieren Merkmale aus den Eingangsbildern, indem sie kleine Filter über das Bild schieben und bestimmte Muster identifizieren, wie z.B. Kanten, Ecken oder Texturen.

Pooling-Schichten:

Nach den Faltungsschichten folgen oft Pooling-Schichten, die dazu dienen, die Dimensionalität der Daten zu reduzieren. Sie aggregieren Informationen aus benachbarten Regionen und erstellen so abstraktere Darstellungen des Bildes.

Vollständig verbundene Schichten:

Diese Schichten sind am Ende des Netzwerks platziert und dienen dazu, die extrahierten Merkmale in eine endgültige Klassifizierung umzuwandeln. Sie nehmen die abstrakten Merkmale aus den vorherigen Schichten auf und kombinieren sie, um eine Vorhersage darüber zu treffen, zu welcher Klasse das Bild gehört.

3.Punkt:

Training des Modells:

Das Modell wird mit den Trainingsdaten trainiert, um die Gewichte der verschiedenen Schichten so anzupassen, dass es in der Lage ist, die Bilder korrekt zu klassifizieren. Während des Trainings durchläuft jedes Bild das Netzwerk, wo es Schicht für Schicht verarbeitet wird, und die Vorhersagen des Modells werden mit den tatsächlichen Labels verglichen. Basierend auf den Fehlern werden die Gewichte des Netzwerks angepasst, um die Leistung zu verbessern.

4.Punkt:

Model sichern:

Das Speichern des trainierten Klassifizierungsmodells ermöglicht es, die trainierten Gewichte und die Modellarchitektur zu sichern. Dadurch können man das Modell später wiederverwenden, um Vorhersagen auf neuen Daten zu treffen, ohne den Trainingsprozess erneut durchführen zu müssen.

5.Punkt:

Beim Testen der Bildklassifizierung wird das trainierte Modell auf neuen Bildern aus einem

separaten Datensatz evaluiert, um seine Leistung zu bewerten, indem die Vorhersagen des Modells mit den tatsächlichen Labels verglichen werden.

3. Code:

3.1 Importing Libraries

Ich habe Jupyter Notebook/ Google Colab für mein Projekt benutzt. Da Jupyter Notebooks eine interaktive Umgebung bieten, die es ermöglicht, Code in einzelnen Zellen auszuführen und die Ergebnisse direkt darunter anzuzeigen. Dies erleichterte das Experimentieren und die Analyse von Daten, insbesondere wenn man verschiedene Modelle trainiert und ihre Leistung vergleicht.

Importing Libraries

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar100
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt
import numpy as np
import random
```

Am Anfang des Codes importiert man zuerst alle notwendigen Bibliotheken.

TensorFlow und Keras Importe:

Importiert die TensorFlow-Bibliothek, die für das maschinelle Lernen und Deep Learning verwendet wird.

from tensorflow.keras import layers, models:

Importiert die Keras-API aus TensorFlow, die zum Aufbau und zur Definition von neuronalen Netzwerken verwendet wird.

from tensorflow.keras.datasets import cifar100:

Importiert den CIFAR-100-Datensatz, der aus 100 verschiedenen Klassen von Bildern besteht.

from tensorflow.keras.utils import to_categorical:

Importiert eine Funktion zum One-Hot-Encoding der Zielvariablen.

from tensorflow.keras.preprocessing.image import ImageDataGenerator:

Importiert den Bildgenerator von Keras, der zur Datenaugmentierung verwendet werden kann, um die Anzahl der Trainingsdaten zu erhöhen und die Generalisierungsfähigkeit des Modells zu verbessern.

import matplotlib.pyplot as plt:

Importiert die Matplotlib-Bibliothek, um Diagramme und Bilder darzustellen.

import numpy as np:

Importiert die NumPy-Bibliothek, die für numerische Berechnungen und die Arbeit mit Arrays verwendet wird.

import random:

Importiert das Modul random, das für die Erzeugung von Zufallszahlen und -auswählen verwendet wird.

Meine nächsten Schritte:

3.2 Loading CIFAR-100 dataset

In diesem Codeausschnitt wird zunächst der CIFAR-100-Datensatz geladen, der aus Trainings- und Testdaten besteht. Die Trainingsdaten enthalten Bilder von 100 verschiedenen Klassen, und jedes Bild ist mit einer entsprechenden Klasse (Label) versehen. Die Liste `class_names` speichert die Namen dieser Klassen entsprechend ihren Indizes.

Anschließend werden 100 ausgewählte Trainingsbilder visualisiert. Dies geschieht in einer 2x5-Rasteranordnung, wobei jedes Bild mit seinem entsprechenden Klassennamen beschriftet ist.

Dies ermöglicht es, einen schnellen visuellen Überblick über die Bilder und ihre zugehörigen Klassen zu erhalten. Die Bilder werden mit Hilfe der `imshow`-Funktion aus Matplotlib angezeigt, während die Klassennamen mit der `title`-Funktion gesetzt werden.

Schließlich wird `plt.axis('off')` verwendet, um die Achsenbeschriftungen für die Bilder zu deaktivieren und eine bessere visuelle Darstellung zu erzielen.



Danach in dem Codeabschnitt werden die Pixelwerte der Trainings- und Testbilder normalisiert, um sicherzustellen, dass sie Werte zwischen 0 und 1 haben.

3.3 Adding augmentation

Dieser Codeausschnitt implementiert zwei wichtige Schritte:

Datenaugmentierung:

Ein Bildgenerator namens `datagen` wird erstellt, um Daten durch Rotation, Verschiebung

und horizontale Spiegelung zu erzeugen. Dieser wird an die Trainingsdaten `x_train` angepasst.

Vorbereitung der Zielvariablen:

Die Zielvariablen `y_train` und `y_test` werden in binäre Klassenmatrizen umgewandelt, um sie für die Klassifikation vorzubereiten.

3.4 Building Model Architecture

Mein Code implementiert eine komplexe neuronale Netzwerkarchitektur, die für die Bildklassifizierung verwendet wird.

resnet_block:

Diese Funktion definiert einen ResNet-Block, der mehrere gestapelte Convolutional-Schichten enthält. Ein ResNet-Block ermöglicht es dem Netzwerk, tiefere Strukturen zu erlernen und das Problem des Verschwindens oder Explodierens der Gradienten zu mildern. Der Parameter `conv_shortcut` steuert die Verwendung einer optionalen Shortcut-Verbindung, die dazu beiträgt, Dimensionalitätsunterschiede zwischen den Eingangs- und Ausgangsschichten auszugleichen.

transformer_block:

Diese Funktion implementiert einen Transformer-Block, der aus einem Multi-Head-Attention-Mechanismus und einem Feedforward-Netzwerk besteht.

Der Multi-Head-Attention-Mechanismus ermöglicht es dem Modell, komplexe Beziehungen zwischen verschiedenen Teilen der Eingabedaten zu modellieren, während das Feedforward-Netzwerk zusätzliche nicht-lineare Transformationen durchführt.

modified_complex_net_with_transformer_and_resnet:

Diese Funktion definiert die Gesamtarchitektur des neuronalen Netzwerks.

Es beginnt mit einer Reihe von Convolutional-Schichten, gefolgt von einem Transformer-Block und mehreren ResNet-Blöcken. Die Anzahl der ResNet-Blöcke und ihre Struktur werden durch die Parameter `num_resnet_blocks` und `num_blocks_list` gesteuert.

Am Ende des Netzwerks befinden sich Global Average Pooling- und Dense-Schichten für die Klassifizierung.

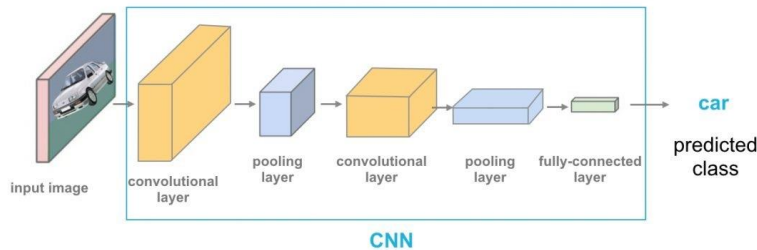
input_shape und num_classes:

Diese Variablen definieren die Form der Eingabedaten und die Anzahl der Klassen für die Klassifizierungsaufgabe.

model:

Am Ende wird das Modell initialisiert und zurückgegeben, das dann für das Training verwendet werden kann.

Insgesamt erstellt mein Code eine neuronale Netzwerkarchitektur, die sowohl ResNet- als auch Transformer-Blöcke integriert, um komplexe Bildklassifizierungsaufgaben zu bewältigen.



<https://viso.ai/wp-content/uploads/2021/03/cnn-convolutional-neural-networks-1060x362.jpg>

3.5 Compiling the model

Der Codeabschnitt konfiguriert das Modell für das Training:

Es wird der Adam-Optimizer für die Optimierung des Modells festgelegt.

Dieser Optimierungsalgorithmus passt die Lernrate für jeden Parameter des Modells adaptiv an und kombiniert Momentum und Lernratenanpassung, um das Training zu beschleunigen.

Die Kategorielle Kreuzentropie wird als Verlustfunktion festgelegt.

Diese Funktion wird häufig für mehrklassige Klassifikationsprobleme verwendet (Weshalb ich diese genommen habe), da sie die Unterschiede zwischen den Modellvorhersagen und den tatsächlichen Labels effektiv misst.

Die Genauigkeit wird als Metrik festgelegt, um die Leistung des Modells während des Trainings und der Auswertung zu überwachen. Die Genauigkeitsmetrik misst, wie oft das Modell die richtigen Vorhersagen trifft.

3.6 Displaying model summary

Die Methode `summary()` gibt eine prägnante textbasierte Zusammenfassung des Modells aus. Diese Zusammenfassung enthält Informationen über die Architektur des Modells, die Anzahl der Parameter und die Form der Ein- und Ausgangstensoren.

Dies ermöglicht einen schnellen Überblick über die Struktur und Komplexität des Modells.

```
Total params: 23024868 (87.83 MB)
Trainable params: 23000036 (87.74 MB)
Non-trainable params: 24832 (97.00 KB)
```

3.7 Training the model

In diesem Codeabschnitt wird das Modell trainiert:

```
epochs = 30
batch_size = 64
history = model.fit(datagen.flow(x_train, y_train, batch_size=batch_size), epochs=epochs)
```

epochs = 30: Definiert die Anzahl der Epochen, also wie oft das Modell über den gesamten Datensatz trainiert wird.

batch_size = 64: Legt die Batch-Größe fest, also wie viele Beispiele gleichzeitig durch das Modell verarbeitet werden. Hier sind es 64 Bilder pro Batch.

history = model.fit(datagen.flow(x_train, y_train, batch_size=batch_size), epochs=epochs): Die fit () -Methode wird verwendet, um das Modell mit den Trainingsdaten zu trainieren. datagen.flow(x_train, y_train, batch_size=batch_size) erzeugt Batches von augmentierten Trainingsdaten mithilfe des zuvor definierten Bildgenerators datagen. Die Anzahl der Epochen wird durch den Parameter epochs festgelegt. Das Trainingsergebnis wird in der Variablen history gespeichert, um später auf die Trainingshistorie zugreifen zu können.

Training for 20 more epochs

```
model.fit(datagen.flow(x_train, y_train, batch_size=batch_size), epochs=20)
```

Dieser Abschnitt des Codes trainiert das Modell, indem er den Daten-Generator datagen verwendet, um augmentierte Trainingsdaten in Batches zu erzeugen.

Diese Batches werden verwendet, um das Modell über 20 Epochen hinweg zu trainieren. Während des Trainings passen sich die Gewichte des Modells kontinuierlich an, um die Verlustfunktion zu minimieren und die Leistung des Modells zu verbessern.

3.8 Visualizing learning curve of model

Der vorliegende Code generiert zwei Diagramme, um die Trainingsleistung des Modells über mehrere Epochen hinweg zu verfolgen:

Diagramm für die Trainingsverluste:

Die Trainingsverluste für jede Epoche werden in einem Linienplot dargestellt, wobei die x-Achse die Epochen und die y-Achse die Verlustwerte darstellen.

Dieses Diagramm zeigt, wie sich der Verlust des Modells im Laufe des Trainings entwickelt und ob das Modell die Fähigkeit verbessert hat, die Daten zu modellieren und Vorhersagen mit geringerem Fehler zu machen.

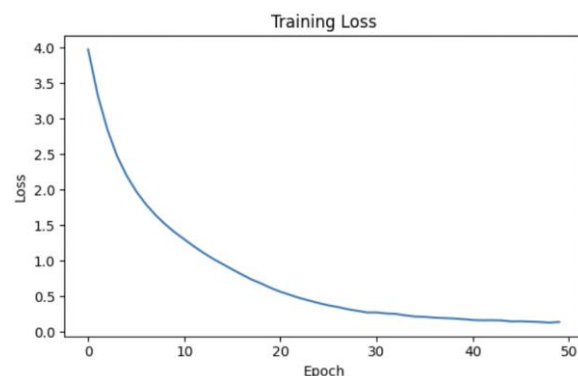
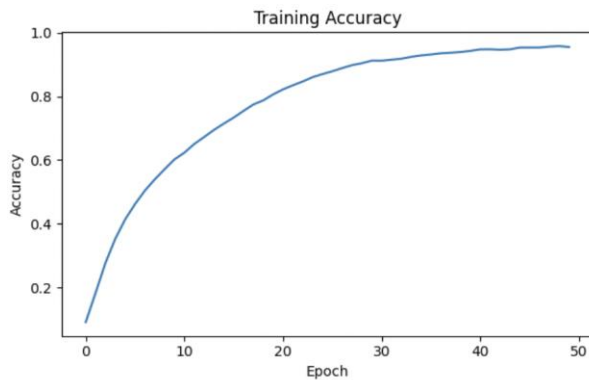


Diagramm für die Trainingsgenauigkeit:

Die Trainingsgenauigkeit für jede Epoche wird in einem separaten Linienplot dargestellt, wobei die x-Achse die Epochen und die y-Achse die Genauigkeitswerte darstellen.

Dieses Diagramm veranschaulicht, wie sich die Genauigkeit des Modells im Laufe des Trainings verbessert hat und wie gut es die Trainingsdaten klassifiziert hat.

Die Kombination dieser beiden Diagramme ermöglicht es, die Leistung des Modells während des Trainings ganzheitlich zu bewerten und zu verstehen.



3.9 Printing Test Accuracy

Hier wird die Evaluierung des Modells auf einem separaten Testdatensatz durchgeführt und die Testgenauigkeit ausgegeben.

`model.evaluate(x_test, y_test, verbose=2):`

Hier wird das Modell mithilfe der `evaluate`-Methode bewertet. Die Methode erhält als Eingabe den Testdatensatz `x_test` und die entsprechenden Labels `y_test`. Das Modell wertet seine Leistung auf diesen Daten aus, indem es Vorhersagen für `x_test` macht und diese mit den tatsächlichen Labels `y_test` vergleicht. Der Parameter `verbose=2` gibt an, dass der Fortschritt während der Evaluierung angezeigt werden soll.

`test_loss, test_accuracy = ...:`

Die `evaluate`-Methode gibt zwei Werte zurück: den Testverlust (`test_loss`) und die Testgenauigkeit (`test_accuracy`). Diese Werte werden in den Variablen `test_loss` und `test_accuracy` gespeichert.

`print(f'Test Accuracy: {test_accuracy*100:.2f} %')`

Hier wird die Testgenauigkeit formatiert ausgegeben. Die %-Formatierung zeigt die Genauigkeit in Prozent an. Die `.2f`-Formatierung stellt sicher, dass die Genauigkeit auf zwei Dezimalstellen gerundet wird, um eine angemessene Anzeige zu gewährleisten. Die Ausgabe zeigt die Testgenauigkeit des Modells auf dem Testdatensatz.

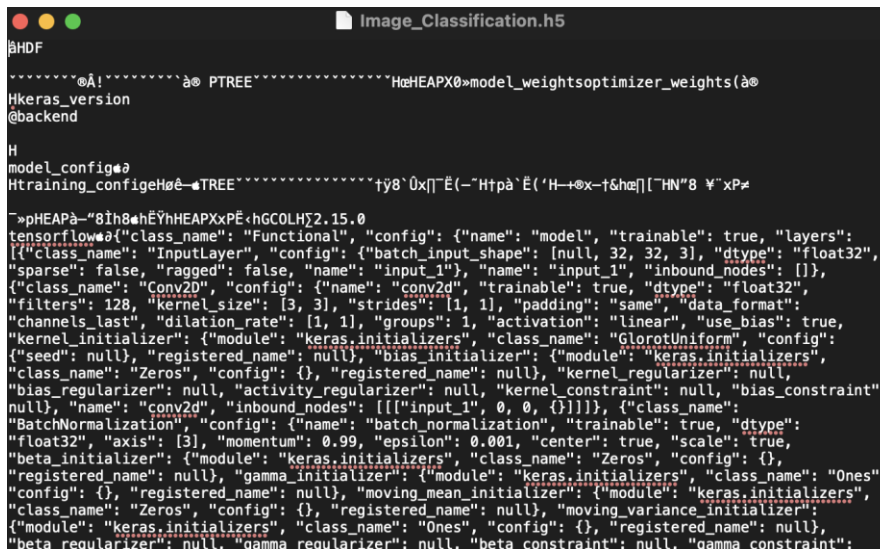
3.10 Saving the Model

In diesem überarbeiteten Code werden mehrere Schritte zusammengefasst:

Das Mouneten von Google Drive erfolgt in einem Schritt.

Das Speichern des gesamten Modells, der Modellarchitektur als JSON und der Modellgewichte erfolgt in separaten Schritten.

Das Anzeigen des Inhalts des Google Drive-Verzeichnisses erfolgt mit dem `!ls`-Befehl.



```

#####@Ä!#####à@ PTREE#####HøHEAPX0>model_weightsoptimizer_weights(à@
Hkeras_version
@backend
H
model_config#3
Htraining_configHøè-@TREE#####ty8`Ùx[]É(-~Htpà`É('H→@x-†&hø[][-HN"8 ¥~xP#
~>pHEAPà-"8Ih8#hËYhHEAPXxPè<hGCOLH]2.15.0
tensorflow{"class_name": "Functional", "config": {"name": "model", "trainable": true, "layers":
[{"class_name": "InputLayer", "config": {"batch_input_shape": [null, 32, 32, 3], "dtype": "float32",
"sparse": false, "ragged": false, "name": "input_1", "name": "input_1", "inbound_nodes": []},
{"class_name": "Conv2D", "config": {"name": "conv2d", "trainable": true, "dtype": "float32",
"filters": 128, "kernel_size": [3, 3], "strides": [1, 1], "padding": "same", "data_format":
"channels_last", "dilation_rate": [1, 1], "groups": 1, "activation": "linear", "use_bias": true,
"kernel_initializer": {"module": "keras.initializers", "class_name": "GlorotUniform", "config":
{"seed": null}, "registered_name": null}, "bias_initializer": {"module": "keras.initializers",
"class_name": "Zeros", "config": {}, "registered_name": null}, "kernel_regularizer": null,
"bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint":
null, "name": "conv2d", "inbound_nodes": [[["input_1", 0, 0, {}]]], {"class_name":
"BatchNormalization", "config": {"name": "batch_normalization", "trainable": true, "dtype":
"float32", "axis": [3], "momentum": 0.99, "epsilon": 0.001, "center": true, "scale": true,
"beta_initializer": {"module": "keras.initializers", "class_name": "Zeros", "config": {},
"registered_name": null}, "gamma_initializer": {"module": "keras.initializers", "class_name": "Ones",
"config": {}, "registered_name": null}, "moving_mean_initializer": {"module": "keras.initializers",
"class_name": "Zeros", "config": null}, "moving_variance_initializer": {"module": "keras.initializers",
"class_name": "Ones", "config": {}, "registered_name": null}, "beta_regularizer": null, "gamma_regularizer": null, "beta_constraint": null, "gamma_constraint":

```

3.11 Making Predictions

Das `model.predict()` -Statement wird verwendet, um Vorhersagen für neue Daten zu treffen, die nicht im Trainingsprozess verwendet wurden.

`x_test` ist der Testdatensatz, für den Vorhersagen getroffen werden sollen.

`model.predict()` wird aufgerufen, um Vorhersagen für den Testdatensatz `x_test` zu machen.

`predictions` ist das Ergebnis der Vorhersagen, das die vorhergesagten Ausgabewerte für jeden Eingabedatensatz enthält.

= 313/313 [=====] - 10s 31ms/step

3.12 Predicted and actual classes

Dieser Codeabschnitt wählt zufällig 10 Indizes aus dem Testdatensatz aus, um Vorhersagen zu machen und diese mit den tatsächlichen Klassen zu vergleichen.

Hier ist, was dieser Code macht:

1. `random_indices = random.sample(range(len(x_test)), 10):`

Hier werden 10 zufällige Indizes aus dem Testdatensatz `x_test` ausgewählt.

2. `selected_x_test = x_test[random_indices]` und `selected_y_test = y_test[random_indices]:`

Hier werden die ausgewählten Testdaten und ihre entsprechenden Labels extrahiert, basierend auf den zufällig ausgewählten Indizes.

3. `selected_predictions = model.predict(selected_x_test):`

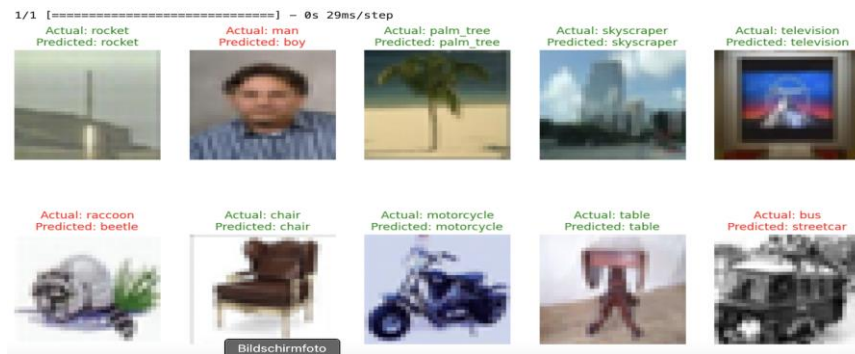
Hier werden Vorhersagen für die ausgewählten Testdaten gemacht, indem `model.predict()` aufgerufen wird.

4. Die Schleife dient dazu, die ausgewählten Testbilder zusammen mit ihren vorhergesagten und tatsächlichen Klassen anzuzeigen.

In jeder Iteration der Schleife wird ein Testbild angezeigt, zusammen mit dem tatsächlichen

und vorhergesagten Klassenlabel. Die Farbe des Texts wird grün dargestellt, wenn die Vorhersage korrekt ist, und rot, wenn sie falsch ist.

5. plt.show(): Dieser Befehl zeigt die gezeichneten Bilder an.



4.Probleme / Abschlussworte:

Probleme:

5. Quellen

Buch:

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow : von Aurélien Géron.

Links:

Github:

<https://github.com/BergmannJustin/Image-Classification-Pr-fung>

Tensorflow:

https://www.tensorflow.org/api_docs/python/tf/keras/Model

<https://www.tensorflow.org/tutorials>

<https://www.tensorflow.org/datasets/catalog/cifar100>

<https://de.wikipedia.org/wiki/TensorFlow>

Youtube:

https://www.youtube.com/watch?v=o_3mboe1jYI

https://www.youtube.com/watch?v=nc7FzLiB_AY

<https://www.youtube.com/watch?v=jztpslzEGc&t=3188s>

<https://www.youtube.com/watch?v=t0EzVCvQjGE>

Allgemein:

<https://www.kaggle.com/>

<https://pyimagesearch.com/category/machine-learning/>

<https://paperswithcode.com/dataset/cifar-100>