HAW HAMBURG

PROJEKTTHEMA

Einfluss von Partial Face Occlusion auf die Erkennungsraten von Deep Neural Networks für Facial Expression Recognition

PROJEKTDOKUMENTATION

Stand: 11. Oktober 2023

Leander Wernst (2555900)

GitHub:
https://github.com/LeanderWernst/facial-expression-recognition

Betreuende Personen:
Prof. Dr. Larissa Putzar
Thorben Ortmann

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Department Medientechnik
Finkenau 35
22081 Hamburg

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Allgemeines

Dieses Projekt dient als Einstieg in die Grundlagen von Deep Neural Networks, da hierzu bislang kein Vorwissen vorhanden war. Es beschäftigt sich mit dem ***Einfluss von Partial Face Occlusion auf die Erkennungsraten von Facial Expression Recognition Modellen*** und soll als Grundlage dienen, um einen Teil eines ggf. nachfolgenden Projektes mit dem Thema *Effekte generativer Bildverfahren zur Verbesserung der Auflösung (Super Resolution) auf Facial Expression Recognition in teilverdeckten Gesichtern* zu beantworten.
Da sich *Deep Neuronal Networks* (DNN) in Studien vielfältiger Felder zur Facial Expression Recognition (FER) bewährt haben und als aktueller Stand der Technik die Erkennungsraten vorheriger Ergebnisse mit bis dahin traditionellen Methoden deutlich übersteigen (Li & Deng, 2022), wird sich dieses Projekt ebenfalls nur mit Methoden des *Deep Learning* beschäftigen.

1.1 Projektziel

1.1.1 Fragestellungen

Die Untersuchung von Effekten durch *Super Resolution* auf FER beinhaltet z.B. die folgenden Teilfragen:

(i) Wie beeinflusst *Partial Face Occlusion* (partielle Gesichtsverdeckung), die bspw. durch das Tragen von VR-Brillen entsteht, die Erkennungsraten von FER-Modellen?

(ii) Hat die Auflösung der Bilddaten einen Einfluss auf die Erkennungsraten von FER allgemein bzw. auf die Performanz bei teilverdeckten Gesichtern?

(iii) Kann das von Super-Resolution-Netzwerken erlernte Wissen zum *Upscaling* von Gesichtern auch auf FER-Tasks transferiert werden und Erkennungsraten positiv beeinflussen?

(iv) Können eventuell positiv auftretende Effekte durch Verwendung von *Face Super Resolution*-Methoden in FER-Tasks auch zur Steigerung der Performance von FER in teilverdeckten Gesichtern beitragen?

1.1.2 Projektumfang

Wie eingangs beschrieben, wird zunächst der Fokus im Rahmen des Projekts auf Frage (i) gelegt. Damit einhergehend ergibt sich der folgende Projektumfang mit den nachstehend grob aufgelisteten Teilaufgaben:

- Auswahl und Organisation eines geeigneten Datensets
- Einarbeitung in Python zum Umgang mit Bilddaten und Deep Neural Networks
- Entwickeln einer Pipeline zur Generierung von *Face Occlusion* und Anpassen des originalen Datensets
- Recherche und Finetuning einer geeigneten Modell-Architektur (*Deep Neural Network*) sowie ggf. Neutraining eines selbst gebauten DNN für den Lerneffekt
- Vergleich der FER-Erkennungsraten zwischen unverdeckten und verdeckten Gesichtern
- Dokumentation des Vorgehens und der Ergebnisse

Je nach Zeitaufwand und Schnelligkeit im Projektfortschritt lässt sich der Projektumfang entsprechend der weiterführenden Fragestellungen erweitern.
Perspektivisch sollen die oben genannten Fragen im Rahmen einer Bachelor-Abschlussarbeit weiter untersucht und beantwortet werden können oder sich aus dem im Projekt angestellten Überlegungen und Erkenntnissen für eine Abschlussarbeit geeignete Fragestellungen entwickeln.

1.2 Anforderungen

Wie in Mollahosseini et al., 2019 beschrieben, gibt es diverse Modelle, um affektive Gesichtszüge (Emotionen) zu quantifizieren, darunter bspw. das kategorische, welches die Emotion aus einer Liste mit sechs Basisemotionen (definiert von Ekman und Friesen, 1971) auszuwählen versucht, das dimensionale, womit ein Wert über eine kontinuierliche emotionale Skala wie Wertigkeit (*Valence*) und Erregung (*Arousal*) gewählt wird und das *Facial Action Coding System* (FACS), das mithilfe von s.g. *Action Units* (AUs) alle möglichen Gesichtsbewegungen beschreibt. Das kategorische stellt hierbei vermutlich das einfachste Modell dar und findet sich auch in vielen Datensets, wie bspw. den frei verfügbaren FER-2013-Daten einer Kaggle-Challenge[1], dem recht aktuellen *AffectNet*-Datensatz oder der *Real-world Affective Faces Database* (RAF-DB) wieder. Aufgrund der Einfachheit dieser Kategorien und ihrer

[1]Kaggle: Online Artificial Intelligence & Machine Learning Community mit Wettbewerben, Modellen, Datensets, Code. Die *FER-2013 Challenge* war ein Wettbewerb zum Entwickeln eines FER-Modells, siehe https://www.kaggle.com/datasets/msambare/fer2013

Verwendung in diversen Datensätzen, wird sich das Projekt auf dieses Modell beschränken. Damit geht einher, dass sich die Emotionserkennung auf einen diskreten Zeitpunkt, d.h. auf ein statisches Bild, bezieht, somit nicht mit Videosequenzen arbeitet und damit keine Veränderungen von Gesichtszügen über einen Zeitraum berücksichtigt.

Die Verdeckung der Gesichter wird durch einfache schwarze Balken realisiert, die ungefähr den Dimensionen einer handelsüblichen VR-Brille entsprechen sollen. Die Ausrichtung (Winkel) der generierten Balken soll sich dabei an der Achse, die durch beide Augen verläuft, orientieren.

Die zu entwickelnden Modelle werden mit *TensorFlow 2*[2] und der zugehörigen *Keras API*[3] umgesetzt. Dabei wird zu Übungszwecken ein eigenes DNN entwickelt und trainiert, für die Evaluierung der Fragestellungen wird eine vorhandene, vortrainierte Referenzarchitektur verwendet und durch Finetuning für den FER-Task angepasst. Die Erkennungsraten sollen dabei in die Nähe (ca. 10 % Abweichung) von State-of-the-Art-Modellen (SOTA-Modellen) kommen. Hierfür werden die Benchmarks von *Papers with Code*[4] zum Vergleich herangezogen. Das beste Modell *POSTER++*[5] hat zum aktuellen Zeitpunkt eine Accuracy von 63,77 % auf dem AffectNet-Datensatz (8 Klassen).
Eine Toleranz von 10 % gegenüber den SOTA-Modellen wird festgelegt, da die Autoren sich i.d.R. auf deutlich mehr Wissen im Bereich des Machine und Deep Learnings stützen können und damit mehr Möglichkeiten kennen ein Modell zu optimieren, sowie oftmals Zugang zu mehr Rechenpower besitzen und damit nicht nur durch schnelleres Training mehr ausprobieren, sondern auch deutlich größere Modelle trainieren können.

Die Dokumentation wird mit LaTeX und der Vorlage zur LaTeX-Formatierung von Abschlussarbeiten der Hochschule für Angewandte Wissenschaften Hamburg geschrieben, um sich mit beidem mit Hinblick auf die Bachelor-Arbeit vertraut zu machen.

1.3 Vorgehen

Die Arbeit am Projekt findet in enger Absprache mit *Thorben Ortmann* statt, dessen Forschungsgebiet im *Affective Computing* liegt. Als Einstiegshilfe in das Gebiet des Deep Learning dient das Buch *Deep Learning with Python, Second Edition* (Chollet, 2021).

[2] TensorFlow ist ein von Google entwickeltes Framework, das Entwicklung sowie Training von Machine Learning Modellen vereinfacht. Siehe https://www.tensorflow.org/

[3] Keras ist eine von François Chollet geschriebene Schnittstelle, die die Verwendung von Tensorflow durch Bereitstellen vieler nützlicher Funktionen nutzerfreundlich gestaltet. Siehe https://keras.io/

[4] *Papers with Code* ist eine kostenlose, offene Website, die mithilfe der Gemeinschaft wissenschaftliche Arbeiten (teils) mit zugehörigem Code, Datensets und Methoden veröffentlicht. Siehe https://paperswithcode.com/task/facial-expression-recognition

[5] siehe https://paperswithcode.com/paper/poster-v2-a-simpler-and-stronger-facial

2 Vorbereitung

2.1 Entwicklungsumgebung

2.1.1 Software & Programmiersprache

Das Projekt wurde mit der Programmiersprache *Python* innerhalb von *Jupyter Notebooks* umgesetzt. Zunächst wurde auf einem Windows 10 Pro (64 Bit) System mit Conda als Paketmanager gearbeitet, wobei sich die Installation von TensorFlow 2.0 in dieser Umgebung als sehr herausfordernd dargestellt hat. Da als Einstiegshilfe in das Thema Deep Learning das Buch *Deep Learning with Python, Second Edition* Chollet verwendet wurde, sollte auch eine möglichst aktuelle, mit dem Buch übereinstimmende Version von TensorFlow installiert werden. Zum Zeitpunkt der Installation war das Version 2.12, wobei Version 2.10 laut offiziellen Installationshinweisen[1] das letzte TensorFlow Release war, das die Verwendung von GPUs auf nativen Windows-Systemen unterstützte.
Um TensorFlow mit GPU-Unterstützung verwenden zu können, wurde zunächst ein offizielles Docker-Image[2] eingesetzt, das zusätzlich durch Installation weiterer Python-Module wie OpenCV, Pandas und Natsort angepasst wurde. Die Virtualisierung durch Docker beanspruchte allerdings wertvolle Ressourcen wie bspw. Arbeitsspeicher, der dringend zum Verarbeiten von Daten kleinerer Übungsprojekte benötigt wurde.
Um möglichst keine Kompromisse eingehen zu müssen, wurde schließlich ein neues Linux-System mit OpenSUSE Leap 15.5 aufgesetzt, auf dem TensorFlow 2.12 schließlich installiert werden konnte.

Die Jupyter Notebooks wurden später so angepasst, dass diese auch in Google Colab (Free Tier) in Verbindung mit Google Drive (wo die Datensätze lagen) verwendet werden konnten und somit auch von unterwegs gearbeitet werden konnte. Ein Problem war zunächst der Flaschenhals, der durch das Laden der Daten aus dem Google Drive in die Umgebung der Virtuellen Maschine von Google Colab, entstand. Dies machte die Arbeit in diesem System so gut wie unbrauchbar, da zwar die Verarbeitung auf der von Google bereitgestellten Grafikkarte recht zügig ging, das Übertragen der Daten des Datensatzes aber sehr viel Zeit in Anspruch

[1] siehe https://www.tensorflow.org/install/pip#windows-native
[2] siehe https://hub.docker.com/r/tensorflow/tensorflow/

nahm, sodass die Schnelligkeit der GPUs nicht genutzt werden konnte. Die Lösung des Problems war es, den Datsensatz zunächst als ZIP-Archiv gepackt auf dem Google Drive zu lagern, dieses dann in die Umgebung der Colab-VM zu kopieren, um die Daten schließlich dort erst zu entpacken. Dieser Prozess ist somit nur einmal initial zum Start einer Colab Session nötig und die benötigten Daten dann vollständig in der Umgebung verfügbar.

2.1.2 Hardware

Auf dem stationären Heimrechner stand folgende Hardware zur Verfügung:

- CPU:
  AMD Ryzen 5 5600X 6-Core CPU 3.70 GHz
- GPU:
  NVIDIA GeForce RTX 3060 Ti (8 GB) [bis 07.08.2023]
  NVIDIA GeForce RTX 4070 (12 GB) [ab 21.08.2023]
- RAM:
  32 GB DDR4-3200

2.2 Datensatz

Als Basis für das Projekt wurde der *AffectNet*-Datensatz verwendet, da dieser nach eigenen Angaben die größte Datenbank kategorischer sowie dimensionaler Emotionsmodelle von *Affect in the Wild* darstellt (Mollahosseini et al., 2019), also solchen Bildern von Gesichtern, die nicht unter Laborbedingungen aufgenommen wurden, sondern aus unterschiedlichen Quellen extrahiert wurden. Hierzu muss erwähnt werden, dass es sich bei den verwendeten Daten um die *small version*, also der kleinen Variante von AffectNet handelt, die aus 291 651 nur manuell durch zwei Personen beschrifteten Bildern besteht und acht Klassen enthält. Ein weiterer Vorteil des Datensatzes ist die vergleichsweise hohe Auflösung der Bilder mit 224×224 Pixeln, was bei der ggf. später verfolgten Fragestellung zum Einfluss dieser auf Erkennungsraten (ii) hilfreich sein kann, da so das Bildmaterial leicht runterskaliert werden kann. Das Trainingsset (bereits vordefiniert im herunterladbaren Datensatz) enthält eine ungleichmäßige Klassenverteilung, siehe Tabelle 2.1. Von diesem Trainingsset wurde ein Validierungsset abgesplittet, das im Datensatz bereits vordefinierte „val_set“ diente als Testdatensatz und hat eine gleichmäßige Klassenverteilung.[3]

[3] je Klasse 500 Bilder, abgesehen von Klasse 7 mit 499 Bildern

Klasse	Definition	Anzahl	Anteil
0	Neutral	74 874	26 %
1	Happiness	134 415	46,7 %
2	Sadness	25 459	8,9 %
3	Surprise	14 090	4,9 %
4	Fear	6378	2,2 %
5	Disgust	3803	1,3 %
6	Anger	24 882	8,7 %
7	Contempt	3750	1,3 %
Gesamt		287 562	100 %

Tabelle 2.1: Klassenverteilung im AffectNet-Datensatz (Trainingsset)

Zusätzlich enthält jedes der beschrifteten Bilder auch 68 *Facial Landmark Points*, die sich dafür eignen die schwarzen Balken der Verdeckung zu berechnen.
Die Bilder liegen im JPG-Format vor und haben jeweils eine Auflösung von 224×224 Pixeln in RGB.

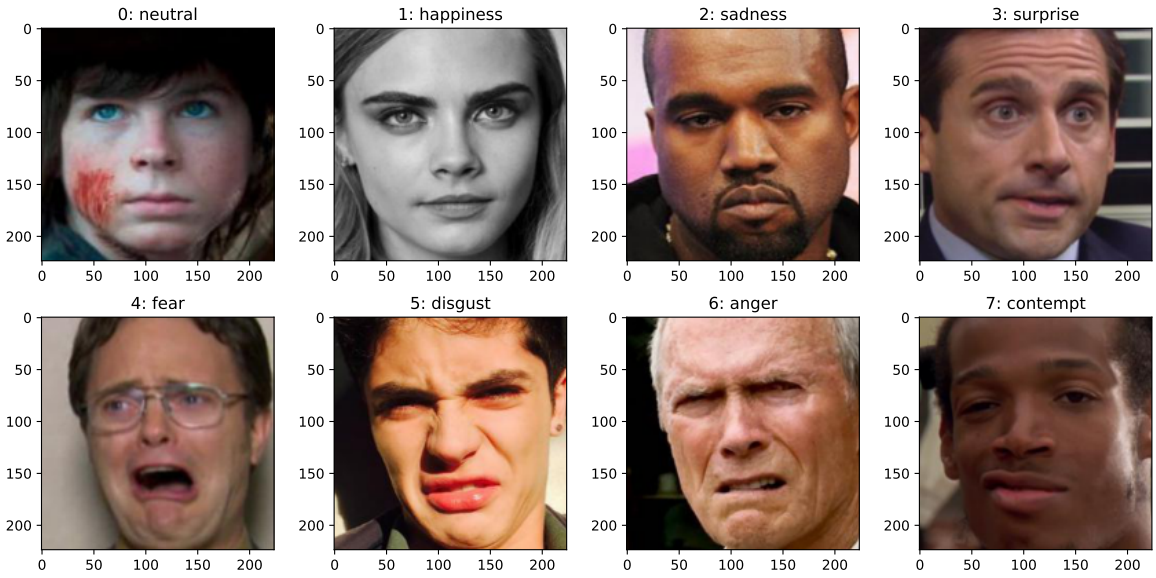


Abbildung 2.1: Beispiele der Bild-Samples des AffectNet-Datensatzes mit Klassen

Zur Einschätzung des Schwierigkeitsgrades zur Klassifikation des Datensatzes schreiben die Autoren: ”[...] AffectNet is a very challenging database and even human annotations have more RMSE than automated methods on existing databases.” (Mollahosseini et al., 2019, S. 26) und erklären, dass das Ergebnis der Bildbeschriftung zeige, dass beide beschriftenden

Personen in 60,7 % miteinander übereinstimmten. (Mollahosseini et al., 2019)

Da sämtliche Bilder des AffectNet-Datensets in einem Ordner lagen und die zugehörigen Klassen in separaten Dateien gespeichert waren, wurden nach den ersten Fehlversuchen ein Modell zu trainieren die Dateien umstrukturiert und Bilder derselben Klasse in einen Ordner verschoben, dessen Bezeichnung die Klasse mithilfe der zugehörigen Ganzzahl angibt.
So konnte die TensorFlow-Funktion `tf.keras.utils.image_dataset_from_directory` zusammen mit dem Parameter `labels='inferred'` verwendet werden, um das Datenset für das Modelltraining ressourcenschonend vorzubereiten. Die Option `inferred` generiert die Klassen ausgehend von der Ordnerstruktur und benötigt so keine separaten Listen bzw. Dateien, um die Labels zu übernehmen.

3 Umsetzung

3.1 Grafik-Pipeline

Um Occlusion, wie sie von VR-Headsets verursacht wird, zu simulieren, muss zunächst eine Grafik-Pipeline entwickelt werden, die den gesamten Bilddatensatz verändert, sodass dieser am Ende sowohl mit als auch ohne Verdeckung vorliegt. Nur so lassen sich valide Aussagen über den Einfluss von teilverdeckten Gesichtern auf die Erkennungsraten von FER treffen. Die Pipeline soll hierbei möglichst nur den Teil des Gesichtes verdecken, der auch unter realen Umständen verdeckt werden würde.

3.1.1 Occlusion-Referenz

Als Referenz für die Dimensionen der Teilverdeckung dient die VR-Brille *Meta Quest 2*, da Meta mit ihr laut Verkaufszahlen von 2016 bis 2021 den größten Marktanteil bei VR-Headsets besitzt (Bocksch, 2023). Nach offiziellen Angaben auf der Produkt-Website, hat diese eine Breite von 224 mm und eine Tiefe von 450 mm (inkl. Kopfband) (Meta Platforms Technologies Ireland Limited, 2023). Angaben zur Höhe lassen sich von offiziellen Quellen nicht finden, auf dem Webserver der Justus-Liebig-Universität Gießen liegt ein Spezifikationsdokument zur Meta Quest 2 („Meta Quest 2“, n. d.), dessen Echtheit sich allerdings nicht überprüfen lässt. Hier werden die Maße der Brille mit 224 mm Breite und 105 mm Höhe angegeben, was sich zumindest mit den offiziellen Breitenangaben deckt. Die Teilverdeckung wird mithilfe eines schwarzen Rechtecks simuliert, die den Proportionen der Meta Quest 2 entspricht.

3.1.2 Berechnung der Verdeckung

Da die einzelnen Bilder des Datensets keine Referenzgrößen beinhalten, um die Größe der Gesichter zu definieren, wird ein ähnlicher Ansatz wie von Houshmand und Khan in (2020) verfolgt: Zunächst wird der Abstand in Pixeln zwischen den Schläfenknochen $\vec{l_0}$ und $\vec{l_{16}}$ (Facial Landmark 0 und Facial Landmark 16) ermittelt, der die Breite w der zu simulierenden VR-Brille darstellt. Anhand der bekannten Dimensionen der Meta Quest 2 wird dann die Höhe h berechnet:

$$w = \left\| \vec{l_0} - \vec{l_{16}} \right\| \qquad \text{und} \qquad h = \frac{224}{105} w$$

Um das Rechteck in seiner Position und Orientierung auszurichten, wird außerdem der Mittelpunkt $\vec{m_{ocd}}$ auf der Strecke $[L_{36}L_{45}]$ von $\vec{l_{36}}$ und $\vec{l_{45}}$ bestimmt. Der Mittelpunkt des Occlusion-Rechtecks soll später mit $\vec{m_{ocd}}$ übereinstimmen. Die Distanz zwischen diesen beiden Vektoren, die jeweils die äußeren Kanten der Augen beschreiben, wird auch *Outer Canthal Distance* (OCD) genannt (Hall et al., 2009).

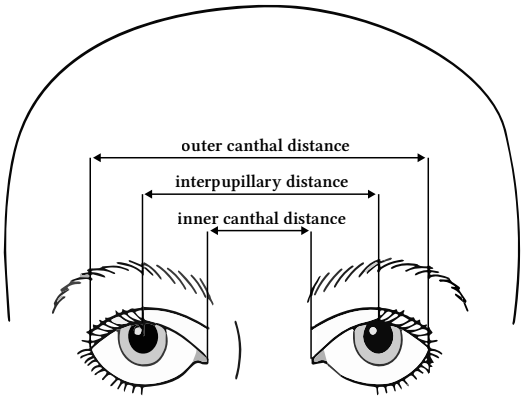


Abbildung 3.1: Schematische Darstellung der wichtigsten periorbitalen[1]Merkmale (In Anlehnung an Hall et al., 2009, S. 30)

$$\vec{m_{ocd}} = \frac{\vec{l_{36}} + \left(\vec{l_{45}} - \vec{l_{36}}\right)}{2}$$

Um die Rotation des Gesichtes auf die Simulation der Brille zu übertragen, wird der Winkel α zwischen der X-Achse und der Strecke $[L_{36}L_{45}]$ bestimmt. Hierfür werden die Längen der Ankathete a und der Gegenkathete b berechnet, um mithilfe des Arcustangens (arctan2) den Winkel zu erhalten:

$$a = l_{36_1} - l_{45_1} \qquad b = l_{36_2} - l_{45_2} \qquad \alpha = \arctan2(b, a)$$

Ausgehend vom Mittelpunkt $\vec{m_{ocd}}$ wird schließlich der erste Punkt $\vec{r_a}$ des Rechtecks berechnet:

$$\vec{r_a} = \vec{m_{ocd}} - \frac{w}{2} \cdot \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} - \frac{h}{2} \cdot \begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix} \tag{3.1}$$

Eine halbe Breite mit dem Vektor $\begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix}$ zu multiplizieren und vom Mittelpunkt $\vec{m_{ocd}}$ zu subtrahieren bewirkt, dass eine Bewegung Richtung Y-Achse entlang einer Geraden auf der

[1] „*Periorbital bedeutet 'in Umgebung der Augenhöhle (Orbita)'*“ (Hircin et al., 2021)

Strecke $[L_{36}L_{45}]$ im gewünschten Winkel α stattfindet. Da $\vec{m_{ocd}}$ im Mittelpunkt des überdeckenden Rechtecks liegen soll, wird die Hälfte der kalkulierten Breite abgezogen. Die Multiplikation einer halben Höhe mit dem Vektor $\begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix}$ um diesen ebenfalls vom Mittelpunkt zu subtrahieren, bewirkt eine ähnliche Bewegung, nun um 90° gegen den Uhrzeigersinn gedreht, um diese auf der lokalen Achse des Rechtecks durchzuführen, welche die Höhe bestimmt.
Abbildung 3.2 veranschaulicht die Einheitsvektoren, die mit Sinus und Kosinus gebildet werden. Durch die Multiplikation werden sie entsprechend skaliert, die Subtraktion führt zum Umkehren ihrer Richtung.

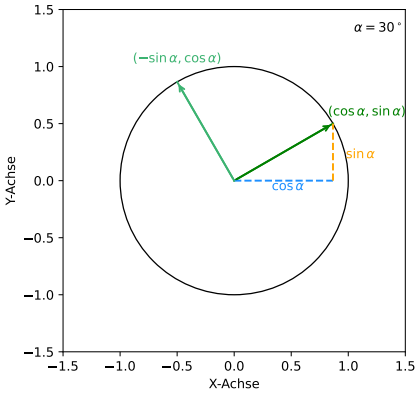


Abbildung 3.2: Ortsvektoren mit Kosinus und Sinus im Einheitskreis [eigene Grafik]

Im Anschluss wird das Rechteck ausgehend von diesem ersten Punkt unter Angabe der kalkulierten Breite und Höhe sowie dem Rotationswinkel gezeichnet.

Abbildung 3.3: Datensatz-Sample vor und nach der Verdeckung

Codeblock 3.1: Python-Funktion der Grafikpipeline zum Berechnen der Verdeckung

```
import os
import gc
import pdb
from tqdm import tqdm
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
from natsort import natsorted

def occlude_face_images(input_path_img, input_path_annotations,
                        output_path, batch_size=0, mem_clean_after=100):
        img_list = os.listdir(input_path_img) # list of all images in folder
        img_list = natsorted(img_list)

        batch_counter = 0
        # Create a tqdm progress bar
        total_images = min(len(img_list),
                           batch_size if batch_size != 0 else batch_size + 1)
        progress_bar = tqdm(total=total_images,
                            desc='Processing␣Images', unit='images')

        # Create checkpoint file
        chkpt = os.path.join(output_path, "checkpoint.txt")
        if os.path.exists(chkpt):
            with open(chkpt, 'r') as f:
                last_processed = f.readline().strip()
        else:
            last_processed = ''

        # Get index of last processed file
        last_processed_index = -1 if last_processed == '' \
                                  else img_list.index(last_processed)

        for i in range(last_processed_index + 1, len(img_list)):
            if batch_counter == batch_size and batch_counter > 0:
                break

        # split filename from extension ['filename', '.ext']
        filename = os.path.splitext(img_list[i])[0]
```

```
        ext = os.path.splitext(img_list[i])[1]

        # Load facial landmarks from the npy file
        l = np.load(os.path.join(input_path_annotations, filename + '_lnd.npy'))

        # Convert landmarks to a 2D array with shape (68, 2)
        l = l.reshape((68, 2))

        # Load corresponding face image
        img = mpimg.imread(os.path.join(input_path_img, img_list[i]))

        # Get original image dimensions
        original_height, original_width, _ = img.shape

        # Create a figure with the same dimensions as the original image
        fig = plt.figure(figsize=(original_width / 100, original_height / 100),
                                  dpi=100)
        ax = fig.add_axes([0, 0, 1, 1])

        ax.imshow(img)
        ax.set_frame_on(False)
        ax.axis('off')

        # Calculate the distance between facial landmark 0 and facial landmark 16
        # norm = vector length, euclidean distance = distance
        w = np.linalg.norm(l[0] - l[16])

        # meta quest 2 aspect ratio (width/height)
        ar = 224/105

        # Calculate the height of the rectangle
        h = w / (ar)

        # Calculate the center point between
        # facial landmark 36 and facial landmark 45 (eyes)
        m_ocd = l[36] + (l[45] - l[36]) / 2

        # Calculate the angle between facial landmark 36 and facial landmark 45
        a = l[45, 0] - l[36, 0] # Ankathete, x-axis
        b = l[45, 1] - l[36, 1] # Gegenkathete, y-axis
        alpha = np.arctan2(b, a)

```

```
        # Calculate the corner point of the rectangle
        R_a = m_ocd - w / 2 * np.array([np.cos(alpha), np.sin(alpha)]) \
              - h / 2 * np.array([-np.sin(alpha), np.cos(alpha)])

        # Create and add the rectangle patch to the plot
        rect = patches.Rectangle((R_a[0], R_a[1]), w, h, angle=np.degrees(alpha),
                                 facecolor='black')
        ax.add_patch(rect)

        # Save File
        plt.savefig(output_path + filename + '_occ.jpg', format='jpg',
                    bbox_inches='tight', pad_inches=0)

        # Remove Image from Memory
        plt.close(fig)

        # Update the checkpoint after processing each file
        with open(chkpt, 'w') as f:
        f.write(filename + ext)
        batch_counter += 1
        progress_bar.update(1)

        # Close progress bar when done
        progress_bar.close()
```

3.2 Deep Neural Network

Da keine Vorerfahrung mit DNNs vorlag, wurde sich an den Inhalten und der Struktur von *Deep Learning with Python, Second Edition* (Chollet, 2021) orientiert. So wurde sich zunächst mit einzelnen *Layern* und zugehörigen Aktivierungsfunktionen in Keras vertraut gemacht, um ausgehend von einfach miteinander verknüpften *Dense Layern* in *Convolutional Neural Networks* (Convnets) einzusteigen, um schließlich eine vortrainierte Referenzarchitektur durch *Finetuning* auf den FER-Task anzupassen.
Dabei wurde insgesamt viel experimentiert, unterschiedliche Layer verschiedener Größe miteinander kombiniert, Lernraten und Optimizer angepasst und längeres Training mit *Early Stopping* (mit *Validation Loss* als ausschlaggebende Metrik) limitiert. Nachfolgend werden nur die ersten oder besten Versuche näher beschrieben.

3.2.1 Basic Convolutional Neural Network

Da es sich bei Facial Expression Recognition auf Grundlage von Bilddatensätzen um einen *Computer Vision Task* handelt, werden für das Projekt Convolutional Neural Networks verwendet, die gemäß Chollet aufgrund ihrer bemerkenswert guten Ergebnisse bei Bildklassifizierungswettbewerben nun fast durchgängig in Computer-Vision-Anwendungen eingesetzt würden. (2021) In seiner *Introduction to convnets* (Chollet, 2021, S. 202) stellt Chollet eine einfache Architektur eines Convnets vor, die auch als Basis für den ersten Klassifizierungstest für den FER-Task dienen sollte und sich im Input unterscheidet, da die Bilder des AffectNet-Datensatzes andere Auflösungen und Farbkanäle (224 × 224 Pixel, RGB) als die des im Buch aufgeführten Datensatzes (28×28 Pixel, Grayscale) des *Modified National Institute of Standards and Technology* (MNIST) haben sowie in der Anzahl der *units* (Knoten bzw. Neuronen) im Dense Layer aufgrund der unterschiedlichen Klassenanzahl.

Codeblock 3.2: Python-Code Definition Convolutional Network mit Keras

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(224, 224, 3)) # width, height, channels
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu") (inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten() (x)
outputs = layers.Dense(8, activation="softmax")(x) # 8 classes in AffectNet
model = keras.Model(inputs=inputs, outputs=outputs)
```

Das Convnet besteht aus einem ***Input Layer***, der einen *rank-3-Tensor*[2] ausgibt. Nach Chollet können Tensoren einfach als „data stored in multidimensional NumPy arrays“ (Chollet, 2021, S. 31) verstanden werden. Sie sind Container für (numerische) Daten und eine Generalisierung von Matrizen (rank-2-Tensoren) mit einer beliebigen Anzahl an Dimensionen. Der *rank* bezeichnet dabei die Anzahl der Dimensionen (engl. *dimensions*) bzw. Achsen (engl. *axis*). Die Form (engl. *shape*) eines Tensors bezieht sich auf die Anzahl der Dimensionen entlang einer Achse. (Chollet, 2021) Entsprechend ist die *Output Shape* des Input-Layers der

[2]Wie der Tabelle entnehmbar, eigentlich ein rank-4-Tensor, wobei *None* für die Anzahl der *Samples* bzw. die *Batch Size* steht. Gemäß der Tensor-Flow-Dokumentation ist diese durch den None-Parameter variabel: *„[It is] a reminder that the model expects the input to have a batch size as the outermost dimension, which in this case can be flexible due to the [None] value.“* („TensorFlow Guide (JS)“, 2022)

Layer	(Type)	Output Shape	Param #
input_1	(InputLayer)	[(None, 224, 224, 3)]	0
conv2d	(Conv2D)	(None, 222, 222, 32)	896
max_pooling2d	(MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1	(Conv2D)	(None, 109, 109, 64)	18 496
max_pooling2d_1	(MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2	(Conv2D)	(None, 52, 52, 128)	73 856
flatten	(Flatten)	(None, 346 112)	0
dense	(Dense)	(None, 8)	2 768 904
Total params:			2 862 152
Trainable params:			2 862 152
Non-trainable params:			0

Tabelle 3.1: Einfache Convnet-Architektur nach Chollet (angepasst)

Convnet-Architektur `(224, 224, 3)`, was (Höhe, Breite, Farbtiefe) entspricht.[3]
Nach dem Input-Layer folgt ein ***Convolutional Layer***. Dieser lernt entgegen einem Dense Layer keine globalen, sondern lokale Muster in seinem durch den Input bestimmten *Feature Space*. Dies bringt gleich zwei Vorteile: Einerseits sind die erlernten Muster unabhängig von ihrer Position im Bild („*translation-invariant*“, Chollet, 2021, S. 205), wodurch weniger Training Samples benötigt werden. Andererseits können dadurch räumliche Hierarchien („*spatial hierachies*“, Chollet, 2021, S. 205) der Muster erlernt werden, da der erste Convolutional Layer kleinere Muster lernt, ein folgender Convolutional Layer aus diesen Features wieder größere Muster bilden kann. Convolutions arbeiten mit rank-3-Tensoren, die *Feature Maps* genannt werden, und geben auch wieder welche zurück, die s.g. *Output Feature Maps*. Die Channel-Achse, dessen Tiefe bei einem Input-Bild noch bei drei (RGB) bzw. 1 (Grayscale) liegt, kann in diesen Tensoren beliebig groß sein, da sie nach einer Convolution nicht mehr für die Farbtiefe, sondern für die Anzahl der *Filter* (auch *Kernel* genannt, Raschka et al., 2022) steht. Sie kodieren unterschiedliche Aspekte der Inputdaten. (Chollet, 2021) Die Anzahl der Filter eines Convolutional-Layers kann, wie in 3.2 ersichtlich wird, durch den Parameter `filters` definiert werden. Die *Kernel-Size* (angegeben mit dem Parameter `kernel_size`) beschreibt dabei die Größe eines durch die Convolution aus dem Input extrahierten Teilstücks, das üblicherweise bei 3×3 (oder auch 5×5) liegt. Diese Teilstücke werden durch ein *Sliding Window* entsprechender Größe aus jeder möglichen Position innerhalb des Input-Tensors extrahiert. (Chollet, 2021) Die Convolutional Layers verwenden eine *Rectified-Linear-Unit*-Funktion (siehe Infobox 3.1).

[3] Chollet benennt hierzu die s.g. *channels-last convention*, was der Standard in TensorFlow ist, sowie die *channels-first convention*, bei der die Farbkanäle vor den Angaben der Höhe und Breite genannt werden. (2021)

Auf den Convolutional Layer folgt ein ***Max Pooling 2D Layer***. Die Funktion einer Max-Pooling-Operation ist es, die von den Convolutions erstellten *Feature Maps* durch s.g. *Downsampling* in ihrer Komplexität zu reduzieren. (Chollet, 2021) Laut Raschka et al. hat dieses auch *Subsampling* genannte Vorgehen zwei Vorteile:

- Max-Pooling führt eine *lokale Invarianz* ein, da kleine Veränderungen benachbarter Werte nicht das Ergebnis der Max-Pooling-Operation verändern, womit die Features robuster ggü. Datenrauschen in den Inputdaten sind. (Raschka et al., 2022)
- Pooling reduziert die Größe der Features, wodurch die Effizenz der Berechnungen gesteigert wird und durch die Reduzierung der Feature-Anzahl auch Overfitting entgegenwirken kann. (Raschka et al., 2022)

Beim Max-Pooling wird ausgehend vom *Stride* (z.B. $P_{2\times2}$), der den Bereich für eine Pooling-Operation festlegt, jeweils der höchste Wert innerhalb dieses Bereichs behalten. Beim *Mean-* bzw. *Average-Pooling* wird der Mittelwert des Bereichs für den neuen Wert gebildet. (Raschka et al., 2022)
Nach dem letzten Convolutional Layer wird ein ***Flatten Layer*** eingesetzt. Dieser ist nötig, da der am Ende folgende ***Dense Layer***, der zur Klassifizierung benötigt wird, mit eindimensionalen Vektoren arbeitet. Aufgabe des Flatten Layers ist es also, den dreidimensionalen Output der Convolution auf eine Dimension zu reduzieren. (Chollet, 2021)
Der Dense-Layer, der eine *Softmax*-Aktivierungsfunktion verwendet (siehe Infobox 3.1), hat eine Output Shape von (8), da acht Klassen im Datenset vorhanden sind.

Infobox 3.1: Aktivierungsfunktionen im Deep Learning

Tensor-Operationen, aus denen Neuronale Netze bestehen, sind einfache geometrische Transformationen der Eingangsdaten im multidimensionalen Raum.[a] (Chollet, 2021) Werden mehrere *Affine Transformationen*[b] miteinander kombiniert, erhält man wieder eine Affine Transformation (Chollet, 2021) der Form $\mathbf{y} = \mathbf{WX} + \mathbf{b}$. Ohne eine Aktivierungsfunktion würde bspw. ein Dense Layer eines Neuronalen Netzes auch nur aus zwei linearen Operationen, dem *Skalarprodukt* (engl. *dot product*) und der Addition, bestehen – die Verkettung mehrerer Layer wie in einem DNN würde somit keinen Vorteil bringen. (Chollet, 2021) Die Verwendung einer nicht-linearen Aktivierungsfunktion wie der **Re**ctified **L**inear **U**nit (ReLU) sorgt somit für einen erweiterten *hypothesis space* („*space of possibilities*“, Chollet, 2021, S. 87). Neben weiteren verwendbaren Funktionen ist die ReLU dabei die beliebteste im Deep Learning, (Chollet, 2021) da sie auch das Problem des verschwindenden Gradienten behebt. (Raschka et al., 2022)

Im Klassifikations-Layer wird (in diesem Fall) eine Softmax-Aktivierungsfunktion verwendet. Sie hängt nicht von der Ausgabe eines einzelnen Neurons ab, sondern von allen des gesamten Layers und hat den Vorteil, dass die Summe der erzeugten Ausgabewerte aus beliebigen Eingabewerten immer Eins ergibt (Ernst et al., 2020) und dies somit als Wahrscheinlichkeitsverteilung zu den vorhandenen (acht) Klassen interpretiert werden kann. Eine Softmax-Aktivierungsfunktion ist damit für nicht-binäre Klassifizierungsprobleme geeignet. (Raschka et al., 2022)

In einem binären Klassifikationsproblem hingegen sollte eine *Sigmoid*-Funktion verwendet werden, die nur ein Skalar zwischen Null und Eins zurückgibt und damit die Wahrscheinlichkeit zu einer Klassenzugehörigkeit angibt. (Chollet, 2021)

[a] Tensoren können als Koordinaten von Punkten im geometrischen Raum betrachtet werden. (Chollet, 2021)

[b] „*An affine transform [...] is the combination of a linear transform (achieved via a dot product with some matrix) and a translation (achieved via a vector addition).*“ (Chollet, 2021, S. 46)

Das Modell wurde mit den folgenden Einstellungen kompiliert:

Codeblock 3.3: Python-Code Modell-Kompilierung

```
model.compile(optimizer="rmsprop",
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])
```

Der *Optimizer* definiert, wie das Neuronale Netz die Gewichte anpasst ausgehende von der

Loss-Funktion. (Chollet, 2021) Die *Root Mean Square Propagation* (RMSProp) ist eines von mehreren Optimierungsverfahren, das anstelle des einfachen Gradientenabstiegs zum Einsatz kommt und neben einer adaptiven Lernrate auch Momente benutzt, um das Training zu verbessern. Hierzu wird nicht nur der Gradient der aktuellen Iteration, sondern auch der der vorherigen berücksichtigt. Dies hat den Vorteil, dass die Wahl der richtigen Lernrate (in der Keras-Implementation standardmäßig bei 0.001[4]) weniger kritisch ist. (Ernst et al., 2020) RMSProp ist außerdem das in den verwenderten Beispielen von Chollet verwendete Verfahren, weshalb es hier übernommen wurde.
Die Loss-Funktion gibt an, wie das Modell seine Performance und Abweichung der Vorhersagen zu den Trainingsdaten misst. (Chollet, 2021) Die *Sparse-Categorical-Crossentropy* ist dabei die Funktion, die für nicht-binäre Klassifizierungsprobleme wird. Für binäre Probleme (nur zwei Klassen) eignet sich die *Binary Crossentropy*, wobei teilweise auch erstere eingesetzt werden kann und wird. (Raschka et al., 2022)
Die *Sparse*-Version muss verwendet werden, da die Klassen als Integer angegeben sind. Bei einer *One-Hot-Kodierung*[5] wird die „normale“ *Categorical Crossentropy* verwendet. (Chollet, 2021) Die *Metrics* definieren, welche Metriken während des Trainings und des Tests ausgegeben werden sollen. (Chollet, 2021) Die *Accuracy* gibt die korrekt klassifizierten Samples im Trainings- und Testdatenset an. (Ernst et al., 2020)

Um dieses Modell mit dem AffectNet-Datensatz zu trainieren, mussten die Bilder zunächst von Pixel-Bildern zu NumPy-Arrays umgewandelt, die zugehörigen Klassen aus den entsprechenden NumPy-Dateien eingelesen und in Trainings- sowie Testdatenset aufgeteilt werden. Aufgrund der Menge konnten die Bilddaten aber nicht komplett erfolgreich in den Arbeitsspeicher geladen werden, weshalb ein Limit als Parameter in eine entsprechende Funktion zum Laden der Daten eingebaut wurde, um das Training zunächst mit einem kleinen Teil des Datensatzes zu testen. Auch wurde mit dem Erstellen von Generator-Objekten experimentiert, was aber verworfen wurde, um einen ersten Test nicht zu komplex zu machen.

Codeblock 3.4: Python-Code Preprocessing AffectNet-Datensatz

```
1 import os
2 import numpy as np
3 from natsort import natsorted
4 from tensorflow.keras.utils import load_img
5 from tensorflow.keras.utils import img_to_array
6
7 train_img_dir = r'P:\_FER\AffectNet\train_set\images'
8 train_label_dir = r'P:\_FER\AffectNet\train_set\annotations'
```

[4] siehe https://keras.io/api/optimizers/rmsprop/
[5] Kodierung von Klassen mit 0 und 1

```
test_img_dir = r'P:\_FER\AffectNet\val_set\images'
test_label_dir = r'P:\_FER\AffectNet\val_set\annotations'

def load_data(train_img_dir, train_label_dir, test_img_dir,
              test_label_dir, limit_train=None, limit_test=None):
        x_train, y_train, x_test, y_test = [], [], [], []

        x_train_files = os.listdir(train_img_dir)
        x_train_files = natsorted(x_train_files)

        y_train_files = os.listdir(train_label_dir)
        y_train_files = natsorted(y_train_files)

        x_test_files = os.listdir(test_img_dir)
        x_test_files = natsorted(x_test_files)

        y_test_files = os.listdir(test_label_dir)
        y_test_files = natsorted(y_test_files)

        for i, file in enumerate(x_train_files):
                if i == limit_train:
                        break
                file_path = os.path.join(train_img_dir, file)
                img = load_img(file_path)
                data = img_to_array(img)
                x_train.append(data)

        for i, file in enumerate(y_train_files):
                if i == limit_train * 4: # four files per image
                        break
                if "_exp.npy" in file:
                    file_path = os.path.join(train_label_dir, file)
                    data = np.load(file_path)
                    y_train.append(data)

        for i, file in enumerate(x_test_files):
                if i == limit_test:
                        break
                file_path = os.path.join(test_img_dir, file)
                img = load_img(file_path)
                data = img_to_array(img)
                x_test.append(data)
```

```
51
52          for i, file in enumerate(y_test_files):
53                  if i == limit_test:
54                          break
55                  if "_exp.npy" in file:
56                      file_path = os.path.join(test_label_dir, file)
57                      data = np.load(file_path)
58                      y_test.append(data)
59
60          x_train, y_train, x_test, y_test = np.array(x_train), \
61                                             np.array(y_train).astype(int), \
62                                             np.array(x_test), \
63                                             np.array(y_test).astype(int)
64
65          return (x_train, y_train), (x_test, y_test)
```

Die Shape der geladenen Daten musste durch *Tensor Reshaping* angepasst und schließlich normalisiert (in einen Wertebereich zwischen Null und Eins umgeformt) werden. Beim Reshaping werden die Reihen und Spalten der Tensoren verändert, sodass sie der *Target Shape* entsprechen. (Chollet, 2021) Durch das Normalisieren bringen wir alle Features in denselben Wertebereich, wodurch das Modell die Daten besser verarbeiten kann. (Raschka et al., 2022)

Codeblock 3.5: Python-Code Reshaping und Normalisierung

```
1 train_images = train_images.reshape((10000, 224, 224, 3))
2 train_images = train_images.astype("float32") / 255
3 test_images = test_images.reshape((3999, 224, 224, 3))
4 test_images = test_images.astype("float32") / 255
```

Das Modell wurde schließlich mit 10 000 Samples in 5 Epochen mit einer Batch-Size von 64 trainiert. Während des Trainings stieg die Accuracy von ca. 0.4492 auf 0.4609 und auch der Loss verringerte sich nur minimal.
Mit Evalulierung des Modells anhand der Testdaten wurde dann klar, dass etwas nicht korrekt funktionierte: Die Test Accuracy lag bei 0.125, also $\frac{1}{8}$, was genau den acht möglichen Klassen entsprach. Gleichzeitig fand das Training zuerst noch auf der CPU statt, was ca. 3 Minuten pro Epoche dauerte. Beim Training der Daten mithilfe der GPU, was zu diesem Zeitpunkt noch mit dem in 2.1.1 beschriebenen Docker-Image stattfand, war mit dieser Methode der mit 8 GB zu geringe VRAM der NVIDIA RTX 3060Ti Grafikkarte das Problem, sodass das Training mit dieser Anzahl an Samples gar nicht ausgeführt werden konnte. Da die schnellere

Berechnung durch die GPU ggü. der CPU aber nichts an dem Problem des nicht funktionierenden (ratenden) Modells ändern würde, wurde dieses Problem hinten angestellt.
Die Vermutung bzgl. der schlechten Accuracy war, dass etwas beim Laden der Daten nicht so funktionierte, wie es sollte und dass bspw. die Klassen-Labels den Bilddaten nicht korrekt zugeordnet wurden.
Nach kurzer Recherche wurde die TensorFlow-Funktion `image_dataset_from_directory`[6] bekannt, die ein Tensorflow-Dataset aus den Dateien eines Dateipfades erstellt. Vor dem Einsatz dieser Funktion wurde, wie ebenfalls unter 2.1.1 beschrieben, auf ein Linux-System mit OpenSUSE Leap 15.5 und TensorFlow 2.12 gewechselt.

Der Code wurde schließlich angepasst, um die neue Funktion einzubinden:

Codeblock 3.6: Python-Code load_data Funktion

```
def load_data(train_img_dir, test_img_dir):
        train_dataset = tf.keras.utils.image_dataset_from_directory(
                train_img_dir,
                labels='inferred',
                label_mode='int',
                color_mode='rgb',
                batch_size=32,
                image_size=(224, 224),
                shuffle=False,
                validation_split=0.2,
                subset='both'
        )
        test_dataset = tf.keras.utils.image_dataset_from_directory(
                test_img_dir,
                labels='inferred',
                label_mode='int',
                color_mode='rgb',
                batch_size=32,
                image_size=(224, 224),
                shuffle=False,
                validation_split=None
        )
        return train_dataset, test_dataset
```

Die TensorFlow-Funktion bietet die Möglichkeit die Labels automatisch (*inferred*) durch die Ordner-Sturktur zu erkennen („TensorFlow API v2.13.0“, 2023), was voraussetzt, dass alle Bilder einer zugehörigen Klasse innerhalb eines Ordners liegen und dieser entsprechend

[6] siehe https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory

der Klasse mit einem Integer (siehe Nummerierung unter 2.1) gekennzeichnet ist. Da die Ordnerstruktur des AffectNet-Datensets die Daten nur in die unterschiedlichen Sets einteilt (`train_set` und `val_set`), wurden alle Dateien entsprechend ihrer Klassen umstrukturiert:

Codeblock 3.7: Python-Code Sortieren der AffectNet Bilddaten nach Klassen

```
import os
import shutil
import numpy as np
def move_images_to_class_folders(image_dir, annotation_dir):
        images = [file for file in os.listdir(image_dir) if file.endswith('.jpg')]

        for image in images:
            image_name, image_ext = os.path.splitext(image)
            image_num, _ = image_name.split('_')
            npy_file = image_num + '_exp.npy'
            npy_path = os.path.join(annotation_dir, npy_file)

            if not os.path.isfile(npy_path):
                continue

            class_val = np.load(npy_path)

            subfolder_dir = os.path.join(image_dir, str(class_val))
            if not os.path.exists(subfolder_dir):
                os.makedirs(subfolder_dir)

            src = os.path.join(image_dir, image)
            dest = os.path.join(subfolder_dir, image)
            shutil.move(src, dest)

move_images_to_class_folders('/home/lndr/dev/FER/AffectNet/train_set/images',
                             '/home/lndr/dev/FER/AffectNet/train_set/annotations')
move_images_to_class_folders('/home/lndr/dev/FER/AffectNet/val_set/images',
                             '/home/lndr/dev/FER/AffectNet/val_set/annotations')
```

Das Laden der Datensets mit der neuen Funktion löste gleich auch das Problem, dass die Speicher (VRAM bzw. RAM) aufgrund der Datenmenge zuvor nicht ausreichten, da die Funktion die Daten in Batches lädt. Somit konnte das gesamte Datenset für das Training verwendet werden.

Codeblock 3.8: Python-Code Laden der Daten mit load_data

```python
train_img_dir = '/home/lndr/Development/FER/AffectNet_sorted/train_set'
test_img_dir = '/home/lndr/Development/FER/AffectNet_sorted/val_set'

(train_dataset, val_dataset), test_dataset = load_data(train_img_dir, test_img_dir)

# >> Found 287651 files belonging to 8 classes.
# >> Using 230121 files for training.
# >> Using 57530 files for validation.
# >> Found 3999 files belonging to 8 classes.
```

Das Training wurde auf diese Weise wiederholt, trotzdem bewegten sich die Metriken nicht und die Accuracy blieb bei 0.125. Es folgten diverse Versuche, das Modell umzubauen, es zu erweitern, mit *Data Augmentation* zu arbeiten, mehr Layer einzubauen, Regularisierung einzuführen und vieles mehr – das Modell kam nicht über eine Accuracy von $\frac{1}{8}$ hinaus.

Das Problem konnte schließlich identifiziert werden: Die Daten wurden der Reihe nach geladen und durch den Split eines Validation Sets von 20 % aus den Trainingsdaten waren dem Trainings Set gar nicht alle Klassen bekannt. Die Lösung lag also darin, die Daten zu mischen, um so in jedem Set auch eine entsprechende Verteilung der Klassen zu gewährleisten. Die oben genannte TensorFlow-Funktion in 3.6 bringt diese Möglichkeit mithilfe zweier Parameter mit, sodass diese nur ergänzt werden mussten:

Codeblock 3.9: Python-Code load_data Funktion mit Shuffle

```python
def load_data(train_img_dir, test_img_dir):
        train_dataset = tf.keras.utils.image_dataset_from_directory(
                train_img_dir,
                labels='inferred',
                label_mode='int',
                color_mode='rgb',
                batch_size=32,
                image_size=(224, 224),
                shuffle=True,                # << activate shuffling
                seed=random.randint(0,1000), # << use random shuffle seed
                validation_split=0.2,
                subset='both'
        )
        test_dataset = tf.keras.utils.image_dataset_from_directory(
                test_img_dir,
                labels='inferred',
```

```
17                 label_mode='int',
18                 color_mode='rgb',
19                 batch_size=32,
20                 image_size=(224, 224),
21                 shuffle=True,                # << activate shuffling
22                 seed=random.randint(0,1000), # << use random shuffle seed
23                 validation_split=None
24         )
25         return train_dataset, test_dataset
```

Mit diesen Einstellungen konnte das Modell schließlich so trainiert werden, dass die Performance über der eines *Random Classifiers*[7] lag. Dennoch lagen die Ergebnisse (Accuracy) immer noch im sehr niedrigen Bereich, angefangen bei ca. 0.277, was durch das Erweitern des DNNs (weitere Convolutional Layer, Einsatz von mehr Filtern, Data Augmentation) in vielen Experimenten[8] bis maximal 0.38 (38 % mit dem Optimizer *Adam* bei einer initialen Lernrate von 1e−6) gesteigert werden konnte.
Die Architektur dieses Modells sah wie folgt aus:

Tabelle 3.2: Eigene Convnet-Architektur für Facial Expression Recognition

Layer	**(Type)**	**Output Shape**	**Param #**
input_1	(InputLayer)	[(None, 224, 224, 3)]	0
rescaling	(Rescaling)	(None, 224, 224, 3)	0
conv2d	(Conv2D)	(None, 222, 222, 64)	1792
max_pooling2d	(MaxPooling2D)	(None, 111, 111, 64)	0
conv2d_1	(Conv2D)	(None, 109, 109, 128)	73 856
max_pooling2d_1	(MaxPooling2D)	(None, 54, 54, 128)	0
conv2d_2	(Conv2D)	(None, 52, 52, 256)	295 168
conv2d_3	(Conv2D)	(None, 50, 50, 256)	590 080
conv2d_4	(Conv2D)	(None, 48, 48, 256)	590 080
max_pooling2d_2	(MaxPooling2D)	(None, 24, 24, 256)	0
conv2d_5	(Conv2D)	(None, 22, 22, 512)	1 180 160

Fortsetzung auf nächster Seite

[7] Man beachte, dass die Klassenverteilung des Testdatensets gleichmäßig ist, sodass die Klassifizierung aller Test-Samples allein mit der Klasse „Happiness“ nicht zu einer Accuracy von 46,7 % (siehe Tabelle 2.1 zur Klassenverteilung des Trainingssets) führen würde. Es wird daher ein Random Classifier, der bei acht Klassen eine Accuracy von 12,5 % erreichen würde, als minimales Performance-Ziel angesetzt.

[8] Leider sind zwischenzeitlich beim Sicherungsversuch in GitHub mit einem unachtsamen Shell-Befehl sämtliche bis dahin auf dem Linux-System erstellten Notebooks unwiederbringlich gelöscht worden. Auch der Versuch einer Datenwiederherstellung brachte keinen Erfolg. Glücklicherweise waren einzelne Notebooks noch auf dem zuvor verwendeten Windows-System bzw. in Google Colab hochgeladen.

Tabelle 3.2: Fortsetzung von vorheriger Seite

Layer	**(Type)**	**Output Shape**	**Param #**
conv2d_6	(Conv2D)	(None, 20, 20, 512)	2 359 808
conv2d_7	(Conv2D)	(None, 18, 18, 512)	2 359 808
flatten	(Flatten)	(None, 165888)	0
dense	(Dense)	(None, 8)	1 327 112
Total params:			8 777 864
Trainable params:			8 777 864
Non-trainable params:			0

Aufgrund der eingeschränkten VRAM-Kapazität der verfügbaren Grafikkarte (NVIDIA RTX 3060Ti 8 GB) brach das Training mit größeren Modellen bspw. mit einem zusätzlichen Max Pooling 2D und einem weiteren Convolutional Layer mit 1024 Filtern teilweise ab, sodass sich im nächsten Schritt mit *Transfer Learning* beschäftigt wurde, um die Performance zu vergleichen und abschätzen zu können, welche Methode erfolgversprechender ist. Der kurzzeitige Versuch die Berechnungszeit durch den Einsatz von *Mixed Precision*[9] zu verkürzen und die Speicherbelastung zu verringern, brachte keinen Erfolg, da bei Verwendung der Methode unterschiedliche Fehlermeldungen auftraten.

3.2.2 Feature Extraction

Chollet beschreibt Feature Extraction als Methode, die von einem vorher trainierten Modell erlernte Repräsentationen verwendet, um diese auf neue Daten anzuwenden und daraus interessante Features zu extrahieren. Hierfür wird die s.g. *Convolutional Base* verwendet und *eingefroren* (unveränderlich gemacht), sodass die trainierten Gewichte dieser Basis bestehen bleiben. Der in dem vortrainierten Modell vorhandenen Classification Layer wird verworfen und mit einem neuen ersetzt. Nur dieser wird neu trainiert. (2021) Chollet verwendet in seinem Beispiel ein *VGG16*-Modell (siehe 3.4), das mit *ImageNet* trainiert wurde – einem Datenset dessen Klassen aus Tieren und Objekten des Alltags bestehen. Er begründet die Wahl des Modells damit, dass es zwar deutlich älter ist und nicht mehr dem State of the Art entspricht, dafür aber ähnlich aufgebaut ist, wie das zuvor eingeführte, einfache Convolutional Neural Network. (Chollet, 2021)
Das VGG16 mit ImageNet-Gewichten wurde zum Test auch für den FER-Task verwendet, was aber, wie vorher bereits aufgrund der sehr unterschiedlichen Klassen (Tiere und Objekte im Gegensatz zu Gesichtern) angenommen, keine guten Ergebnisse brachte.

[9] siehe https://www.tensorflow.org/guide/mixed_precision

3.2.3 Finetuned Neural Network

Beim Finetuning wird nicht nur der letzte Classification Layer angepasst und neu trainiert, sondern auch einige der zuvor geschalteten Layer. (Chollet, 2021) Um für den FER-Task besser passende Modelle zu verwenden, wurde auf solche zurückgegriffen, die mit *VGGFace2* trainiert wurden, einem Datenset, das zur Gesichtserkennung (Identifikation) von der University of Oxford erstellt wurde. (Cao et al., 2018) Es enthält 3,31 Millionen Bilder von 9131 Personen, die von *Google Images* heruntergeladen wurden, mit durchschnittlich 362,6 Bildern pro Person. Außerdem zeigen die Bilder große Variation bzgl. Pose, Alter, Belichtung, ethnischer Zugehörigkeit und Beruf. (Cao et al., 2018)
Da die damit trainierten Modelle ursprünglich mithilfe des *Caffe Deep Learning Frameworks* implementiert wurden und somit nicht für die Verwendung mit TensorFlow geeignet waren, wurden diese (VGG16, ResNet50, SeNet50) von Refik Can Malli konvertiert und via GitHub zur Verfügung gestellt.[10]
Da sich (durch eigene Experimente mit dem AffectNet-Datensatz) reine Feature Extraction als weniger erfolgversprechend ggü. dem Finetuning herausgestellt hat, wird im Folgenden auch nur darauf eingegangen.

VGG16

Als erstes wurde wieder das VGG16-Modell verwendet, da es am übersichtlichsten (bezogen auf die Anzahl der Layer) ist. Die obersten (letzten *fully connected*) Layer wurden entfernt und durch eigene Layer ersetzt. Außerdem wurden nur die letzten vier Layer der Convolutional Base (drei Convolutional und ein Max Pooling 2D Layer) trainierbar gemacht, sodass hierfür neue Gewichte gelernt werden konnten.

Codeblock 3.10: Python-Code Pre-trained VGG16-Modell laden und anpassen

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras_vggface.vggface import VGGFace

# Load Base Model
vgg_model = VGGFace(model="vgg16", include_top=False, input_shape=(224, 224, 3))
vgg_model.trainable = True
for layer in vgg_model.layers[:-4]:
    layer.trainable = False

```

[10] siehe https://github.com/rcmalli/keras-vggface

```
# Add Head Model (Finetune)
inputs = keras.Input(shape=(224, 224, 3))
x = keras.applications.vgg16.preprocess_input(inputs)
x = vgg_model(x)
x = layers.Flatten(name='flatten')(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(8, activation='softmax', dtype="float32")(x)
model = keras.Model(inputs, outputs)
```

Zwischen den Dense Layern wurden ein *Dropout Layer* gesetzt, um Overfitting herauszuzögern. Laut Chollet ist Dropout eine der effektivsten und meist genutzten Methoden zur Regularisierung von Neuronalen Netzen und funktioniert, indem abhängig von einer s.g. *Dropout Rate* zufällig Werte von Output Features fallengelassen, also auf Null gesetzt, werden. (2021) Mit einer Learning Rate von 1e−5 und RMSProp als Optimizer lagen in Versuchen die besten getesteten Modelle bei einer Accuracy von 41,2 %.

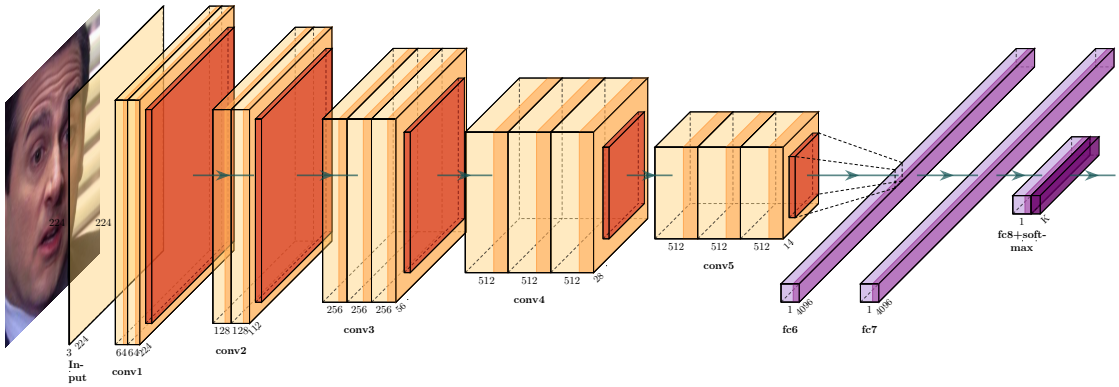


Abbildung 3.4: Schematische Darstellung der Layer des VGG16-Modells [adaptiert von (Iqbal, 2020, /examples/VGG16)]

ResNet50

Als nächstes wurde eine ResNet50-Architektur getestet, die sich „Residual Connections“ (Chollet, 2021, S. 252) zunutze macht und von He et al., 2015 in 2015 vorgestellt wurde. Je tiefer ein DNN, desto schwieriger wird es, diese zu trainieren. (He et al., 2015) Wie Chollet erklärt, entsteht durch die Verkettung vieler Layer ein Informationsrauschen, da Fehlerinformationen zum Anpassen einer Funktion durch andere Funktionen durchlaufen müssen. Ist

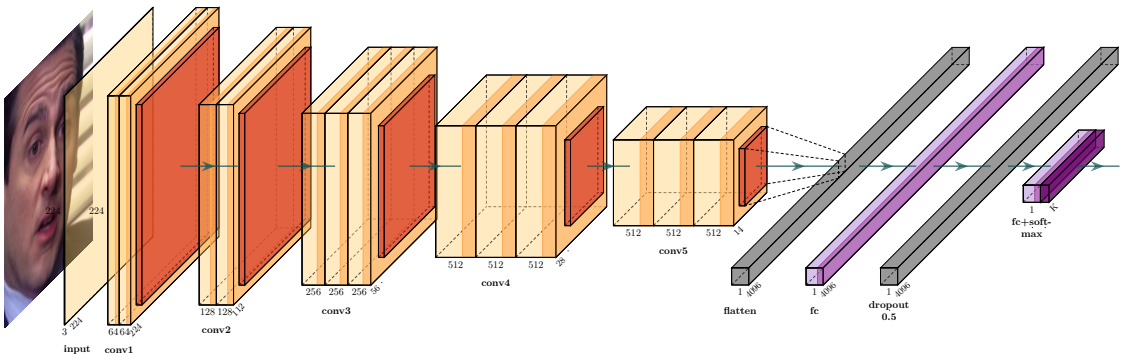


Abbildung 3.5: Schematische Darstellung der Layer des finetuned VGG16-Modells [adaptiert von (Iqbal, 2020, /examples/VGG16)]

diese Verkettung zu lang, überwiegt irgendwann das Rauschen die Information, die für das Gradient Descent Verfahren benötigt wird und die Backpropagation funktioniert nicht mehr. (2021) Man nennt es auch das *„vanishing gradients problem“*. (Chollet, 2021, S. 252) Bei einer Residual Connection wird der rauschlose Input eines Layers aufgehoben, bevor er verarbeitet wird, um ihn später auf den Output zu addieren. Dadurch bleibt die ursprüngliche Information erhalten. Zu beachten ist dabei, dass zum Addieren beide Tensoren dieselbe Shape benötigen, was nicht der Fall ist, wenn bspw. ein Convolutional Layer mit einer höheren Anzahl an Filtern oder ein Max Pooling Layer verwendet wird. Zum Beheben des Problems, wird die Residual Information (der gespeicherte Input) einfach mithilfe eines 1×1 Convolutional Layers auf die richtige Shape projiziert. (Chollet, 2021)

Für das Training wurde wie schon beim VGG16-Modell nur das Basis-Modell übernommen, d.h. nur die Convolutional Layer Blöcke ohne den letzten Dense Layer. Es wurden außerdem alle Gewichte bis auf die der letzten 33 Layer (des Basis-Modells) eingefroren, was bedeutet, dass nach den Conv4-Blöcken (siehe hierzu Grafik 3.7) alles neu trainiert wurde.
Die ersten Ergebnisse des trainierten Modells sahen mit einer Accuracy von 46,8 % schon deutlich vielversprechender aus, als noch mit VGG16, weshalb versucht wurde, das ResNet50-Modell noch weiter zu optimieren und mit mit verschiedenen Einstellungen zu experimentieren.

Einen großen Durchbruch bzgl. der Performance gab es schließlich, als nach einiger Recherche bzgl. dem Umgang mit einem *Imbalanced Dataset*[11], wie es AffectNet mit der unterschied-

[11] Ein vorheriger Test, bei dem das Datenset insgesamt so reduziert wurde, dass alle Klassen dieselbe Anzahl an Samples enthielt, brachte (vermutlich wegen dieser starken Datenreduzierung) keine guten Ergebnisse.

lichen Anzahl Bilder pro Klasse ist, die Möglichkeit entdeckt wurde, beim Training eine Gewichtung der Klassen anzugeben. Klassen für die weniger Samples zur Verfügung stehen werden so stärker gewichtet und Klassen mit mehr Daten abgeschwächt. Die Gewichte wurden wie folgt berechnet:

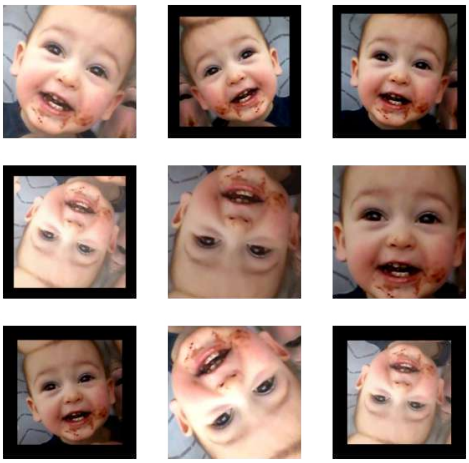
Codeblock 3.11: Python-Code Gewichtung für Imbalanced Dataset berechnen

```python
import numpy as np
# calculate weights on class distribution from 0-7
total_images = 287_652
images_per_class = [74_874, 134_415, 25_459, 14_090, 6_378, 3_803, 24_882, 3_750]
class_distribution = [d/total_images for d in images_per_class]
# inverse class distribution for weights (lower dist = higher weight)
class_weights = [1.0 / dist for dist in class_distribution]
# normalize
class_weights_normalized = class_weights / np.sum(class_weights)
# create dict for use with TensorFlow
class_weights_dict = {i: weight for i, weight in enumerate(class_weights_normalized)}
```

Nach vielen Experimenten wurden Layer und Einstellungen des Modells wie folgt für das Training eingesetzt und vorgenommen:

Aufzählung 3.1: Layer und Parameter des ResNet50-Modells zum Training

- Gewichte der letzten 33 Layer trainierbar (ab Conv5-Blöcken)
- Data Augmentation (siehe Abbildung 3.6)
- *Flatten* und *Dense Layer* (1024 Filter) vor dem Klassifikations-Layer
- *Optimizer* Adam mit einer initialen Lernrate von 1e−3
- Automatische Reduzierung der Lernrate bis 1e−8 mit einem Faktor von 0.2 und einer *Patience* von 5 Epochen bei Erreichen eines Validation Loss Plateaus
- *Early Stopping* mit Validation Loss als Metrik und einer *Patience* von 20 Epochen



Parameter	Wert
Random Flip	horizontal & vertikal
Random Rotation	0.1
Random Brightness	0.2
Random Contrast	0.2
Random Zoom	0.3; Fill: „constant“

Abbildung 3.6: Beispiel & Parameter der Data Augmentation [eigene Grafik]

Codeblock 3.12: Python-Code Laden und Trainieren des ResNet50-Modells

```
from tensorflow.keras import layers
from keras_vggface.vggface import VGGFace
# Define Data Augmentation Layer
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal_and_vertical"),
        layers.RandomRotation(.1),
        layers.RandomBrightness(.2),
        layers.RandomContrast(.2),
        layers.RandomZoom(.3, fill_mode="constant"),
    ]
)
# Load Base Model
vgg_model = VGGFace(model="resnet50", include_top=False, input_shape=(224, 224, 3))
vgg_model.trainable = True
for layer in vgg_model.layers[:-33]: # starting with res5
layer.trainable = False

# Add Head Model
inputs = keras.Input(shape=(224, 224, 3))
x = data_augmentation(inputs)
x = keras.applications.resnet.preprocess_input(x)
x = vgg_model(x)
x = layers.Flatten(name='flatten')(x)
x = layers.Dense(1024, activation="relu")(x)
outputs = layers.Dense(8, activation='softmax')(x)
```

```
model = keras.Model(inputs, outputs)

# Compile Model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
loss = "sparse_categorical_crossentropy",
metrics = ["accuracy"])

# Load dataset
(train_dataset, val_dataset), test_dataset = load_data(train_img_dir, test_img_dir)

from datetime import datetime
now = datetime.today()
current_time = now.strftime("%Y%m%d_%H-%M")
model_name = f"{current_time}_fer_vggface-resnet50_finetuned_"-

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath=model_name + 'E{epoch:02d}-VLoss{val_loss:.2f}.keras',
        save_best_only=False,
        save_freq='epoch',
        monitor="val_loss",
        verbose=1),
        keras.callbacks.EarlyStopping(
        monitor="val_loss",
        patience=20),
        keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
        factor=0.2,
        patience=5,
        verbose=1,
        min_lr=1e-8
    )
]
# fit model
print(f'Saving␣model␣as:␣{model_name}')
history = model.fit(x=train_dataset, epochs=50, validation_data=val_dataset,
                    class_weight=class_weights_dict, callbacks=callbacks)
```

Dieses Modell lieferte die bisher beste Accuracy des Projekts mit 58,9 % (nach dem gesamten Trainingsdurchlauf und Early Stopping nach 25 Epochen). Durch den Vergleich von zwischengespeicherten Modellen des Durchgangs erreichte der Trainingsstand nach 16 Epochen eine Test Accuracy von 59,3 % und ist damit nur noch ca. 4,5 % vom aktuell besten SOTA-Modell Poster++ entfernt.

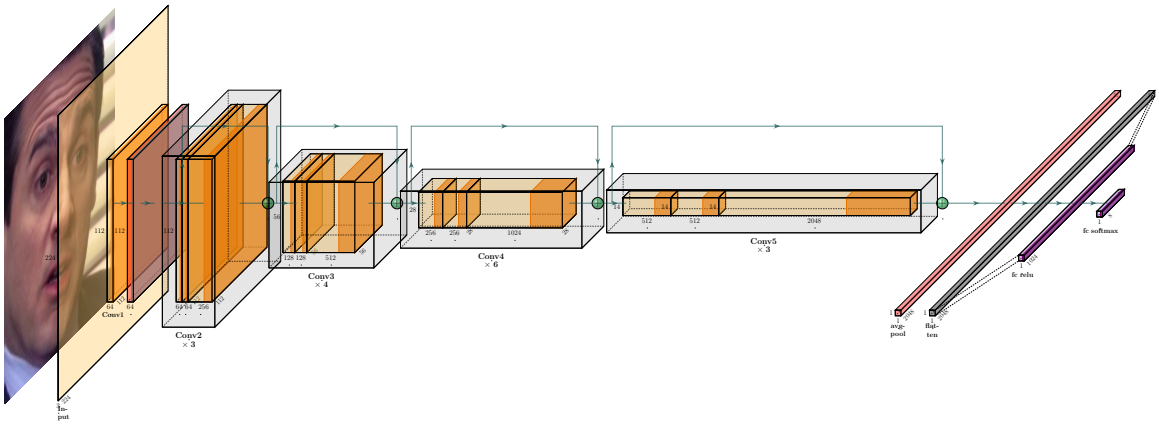


Abbildung 3.7: Schematische Darstellung der Layer des finetuned ResNet50-Modells [adaptiert von (Pan, 2022, ResNet101)]

Die in 3.7 sichtbare Modell-Architektur sowie die in 3.1 aufgeführten Einstellungen dienen als Grundlage für die Evaluierung und werden für Trainings mit angepasstem Datensatz übernommen, um die Ergebnisse miteinander zu vergleichen.

SeNet50

Zum Vergleich mit dem ResNet50-Modell wurde auch das ebenfalls von Rafik Can Malli für TensorFlow konvertierte und mit VGGFace vortrainierte SeNet50-Modell getestet. SeNet steht für *„Squeeze-and-Excitation Network“*. (Hu et al., 2017) Das Modell baut auf ResNet50 auf und versucht durch die zusätzliche Verwendung s.g. Squeeze-and-Excitation-Blocks (SE-Blocks) die Performance zu steigern. Diese SE-Blocks werden eingeführt *„[...] with the goal of improving the quality of representations produced by a network by explicitly modelling the interdependencies between the channels of its convolutional features.“* (Hu et al., 2017, S. 1).
Das SeNet50 wurde mit denselben Einstellungen und zusätzlichen Layern zum Fine-Tuning wie zuvor beim ResNet50 trainiert. Aufgrund der zusätzlichen Layer durch die SE-Blocks wurden jedoch die letzten 54 statt nur 33 Layer des Basismodells neu trainiert, was in beiden Fällen aber dem Beginn des fünften Residual-Blocks (siehe Abbildung 3.7 *Conv5*) entspricht. Da das SeNet50 bei gleichen Parametern aber nur eine Test-Accuracy von 58,1 % erreichen konnte, wurde sich nicht tiefer gehend mit dieser Art Modell befasst.

3.2.4 Finales Training

Das ResNet50 wurde schließlich mit drei unterschiedlichen Datensätzen trainiert:

(i) Unveränderter AffectNet-Datensatz

(ii) AffectNet-Datensatz mit verdeckten Gesichtern

(iii) Aus (i) und (ii) gemischter Datensatz (beide Datensätze komplett zusammengelegt)[12]

Hierbei wurden jeweils die Einstellungen wie in 3.1 beschrieben beibehalten und jeweils das trainierte Modell mit dem geringsten Validation Loss sowie das zuletzt (nach dem Durchlauf aller Epochen oder dem Anhalten aufgrund von Early Stopping) gespeicherte zur Evaluierung auf dem Testdatensatz verwendet.

[12]Das Testdatenset beinhaltet weiterhin 500 Bilder je Klasse (499 bei Klasse 7) wobei jeweils 50 % aus den occluded und normalen Samples besteht. Es wurde darauf geachtet, dass kein Sample in beiden Varianten vorkommt – die Gesichter sind also, egal ob normal oder verdeckt, einzigartig im Datensatz.

4 Ergebnisse

Die zugrunde liegenden Ergebnisse resultieren aus dem Training des 3.2.3 beschriebenen und mit VGG-Face vortrainierten ResNet50-Modells, da es wie erläutert die beste Performance der getesteten Architekturen zeigte.
Aus einem Trainingsdurchlauf wurden jeweils zwei Modelle für den Vergleich herangezogen: Das Modell mit dem geringsten Validation Loss und das letzte Modell des Trainings. Die verwendete Architektur und die zugehörigen Trainingseinstellungen können 3.1 entnommen werden.

4.1 AffectNet

Die hier unter 4.1 aufgeführten Werte und Ergebnisse beziehen sich alle auf das Training mit dem AffectNet-Datensatz.

4.1.1 Training mit unverändertem Datensatz

Test-Datensatz	Epoche	Val. Loss	Val. Accuracy	Test Accuracy	Veränderung
Normal (i)	E5	0,892	68,24 %	52,79 %	0 % (Basis)
Occluded (ii)	E5	0,892	68,24 %	30,21 %	-42,77 %
Normal (i)	E25*	0,9672	61,86 %	58,89 %	0 % (Basis)
Occluded (ii)	E25*	0,9672	61,86 %	38,08 %	-35,34 %

**letzte Epoche*

Tabelle 4.1: Modell-Performance mit Training des unveränderten Datensatzes

In Epoche 5 erreicht das Training mit dem unveränderten Datensatz den geringsten Validation Loss mit 0,892 bei einer Validation Accuracy von 68,24 %, was in einer Test Accuracy von 52,79 % resultiert. Diese reduziert sich um 42,77 % auf eine Test Accuracy von 30,21 %, wenn das trainierte Modell auf dem angepassten Datensatz mit den verdeckten Gesichtern (Occluded Datensatz) getestet wird.

Das Training wird aufgrund der eingestellten Early-Stopping-Parameter nach 25 Epochen bei einem Validation Loss von 0,9672 und einer Validation Accuracy von 61,86 % beendet. Das Modell dieser Epoche resultiert in einer Test Accuracy von 58,89 %, welche sich durch Testen auf dem Occluded Datensatz um 35,34 % reduziert und damit bei einer Test Accuracy von 38,08 % liegt.
In Tabelle 4.1 nicht aufgeführt, aber aufgrund der höheren Accuracy erwähnenswert, ist, dass nach Epoche 16 Epochen bei einem Loss von 0,9916 eine Test Accuracy von 59,3 % erreicht wird und diese nur noch ca. 4,5 % von den aktuellen SOTA-Modellen entfernt liegt.

4.1.2 Training mit Occluded Datensatz

Test-Datensatz	Epoche	Val. Loss	Val. Accuracy	Test Accuracy	Veränderung
Occluded (ii)	E14	1,123	56,88 %	46,39 %	0 % (Basis)
Normal (i)	E14	1,123	56,88 %	44,69 %	-3.66 %
Occluded (ii)	E34*	1,1557	55,9 %	49,64 %	0 % (Basis)
Normal (i)	E34*	1,1557	55,9 %	47,21 %	-4,9 %

**letzte Epoche*

Tabelle 4.2: Modell-Performance mit Training des Occluded Datensatzes

Beim Training mit dem Occluded Datensatz erreicht das Modell nach Epoche 14 den geringsten Validation Loss mit 1,123 bei einer Validation Accuracy von 56,88 %. Getestet auf dem Occluded Datensatz wird eine Test Accuracy von 46,39 % erzielt, auf dem unveränderten (normalen) Datensatz eine Accuracy von 44,69 %, was einer Reduktion um 3,66 % entspricht.
Epoche 34 ist die letzte Epoche des Trainingsdurchlaufs, bei dem das Modell einen Validation Loss von 1,1557 bei einer Validation Accuracy von 55,9 % vorweist und eine Test Accuracy von 49,64 % (Occluded Datensatz) zeigt. Um 4,9 % reduziert ist die Test Accuracy desselben Modells, wenn der unveränderte Datensatz getestet wird, und erzielt 47,21 %.

4.1.3 Training mit Mischdatensatz

Wird das Modell mit dem gemischten (die kompletten normalen Daten zusammengelegt mit den allen verdeckten) Datensatz traininert, wird der geringste Validation Loss mit 1,0281 bei einer Validation Accuracy von 60,7 % in Epoche 18 erreicht. Dies führt zu einer Test Accuracy von 48,14 % auf dem Mischdatensatz. Getestet auf dem unveränderten Datensatz kann diese um 20,57 % auf 58,04 % gesteigert werden. Eine Steigerung von 10,18 % auf 48,11 % ggü. dem Mixed-Testdatensatz erfolgt, wenn der Occluded Datensatz getestet wird.

Test-Datensatz	Epoche	Val. Loss	Val. Accuracy	Test Accuracy	Veränderung
Mixed (iii)	E18	1,0281	60,7 %	48,14 %	0 % (Basis)
Normal (i)	E18	1,0281	60,7 %	58,04 %	+20,57 %
Occluded (ii)	E18	1,0281	60,7 %	53,04 %	+10,18 %
Mixed (iii)	E38*	1,1067	59,62 %	49,06 %	0 % (Basis)
Normal (i)	E38*	1,1067	59,62 %	58,94 %	+20,14 %
Occluded (ii)	E38*	1,1067	59,62 %	54,06 %	+10,19 %

**letzte Epoche*

Tabelle 4.3: Modell-Performance mit Training des gemischten Datensatzes
(Dopplung der Gesichter durch die zwei Varianten „*normal*“ und „*verdeckt*“)

Das Training läuft bis Epoche 38, bei dem ein Validation Loss von 1,1067 und eine Validation Accuracy von 59,62 % erreicht wird. Getestet auf dem Mischdatensatz wird eine Test Accuracy von 49,06 % erreicht, auf dem normalen Datensatz eine von 58,94 %, was einer Steigerung um 20,14 % entspricht. Der Test auf dem Occluded Datensatz erzielt eine Accuracy von 54,06 % und damit eine Steigerung um 10,19 % ggü. dem Mischdatensatz.

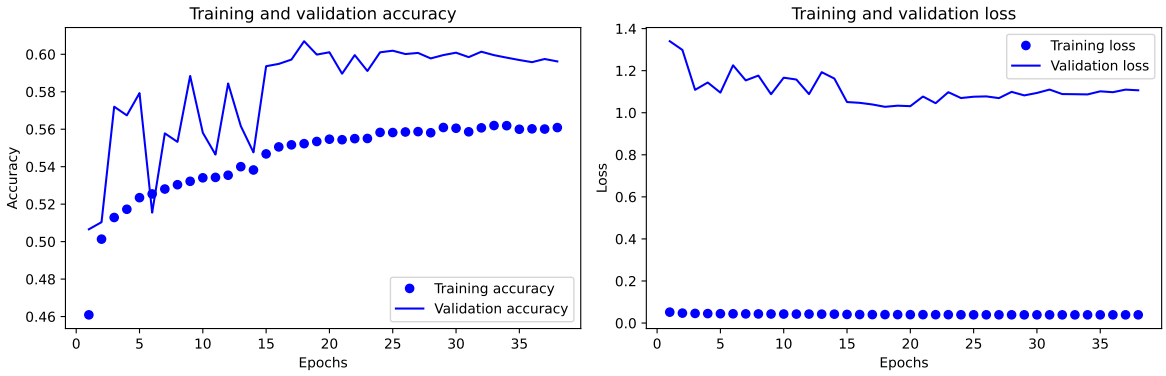


Abbildung 4.1: Exemplarischer Trainingsverlauf zu Tabelle 4.3 mit Mischdatensatz

Außerdem wurde ein weiterer Trainingsdurchlauf mit einem leicht angepassten Trainingsdatensatz durchgeführt. Dieser beinhaltete nicht wie zuvor die durch das Zusammenlegen beider Sample-Varianten (normal & verdeckt) entstehende doppelte Anzahl an Bildern (575 124), sondern weiterhin 287 567 Samples. Hierbei wurde sichergestellt, dass die Gesichter nur einmalig in vorkommen. Ein Gesicht, das verdeckt in diesen Datensatz integriert wurde, war also kein zweites Mal unverdeckt vorhanden.

In diesem Fall lief das Training bis Epoche 50 mit einem Validation Loss von 1,0635 und einer

Test-Datensatz	Epoche	Val. Loss	Val. Accuracy	Test Accuracy	Veränderung
Mixed (iii)	E33	1,0554	59,38 %	53,34 %	0 % (Basis)
Normal (i)	E33	1,0554	59,38 %	57,16 %	+7,16 %
Occluded (ii)	E33	1,0554	59,38 %	48,11 %	-9,81 %
Mixed (iii)	E50*	1,0635	59,1 %	53,21 %	0 % (Basis)
Normal (i)	E50*	1,0635	59,1 %	57,34 %	+7,76 %
Occluded (ii)	E50*	1,0635	59,1 %	48,04 %	-9,72 %

**letzte Epoche*

Tabelle 4.4: Modell-Performance mit Training des gemischten Datensatzes (Gesichter einzigartig im Trainingsdatensatz)

Validation Accuracy von 59,1 %, was auf dem Mischdatensatz getestet in einer Test Accuracy von 53,21 %, auf dem normalen Testdatensatz in 57,34 % (Steigerung von 7,76 %) und auf dem verdeckten in 48,04 % resultierte (Verringerung um 9,72 %).
Der geringste Validation Loss des Durchlaufs wurde mit 1,0554 bei einer Validation Accuracy von 59,38 % in Epoche 33 erreicht. Hierbei resultierte die Test Accuracy auf dem Mischdatensatz in 53,34 %, auf dem normalen wurden 57,16 % erreicht (Steigerung von 7,16 %) und auf dem Occluded Datensatz 48,11 % (Verringerung um 9,72 %).

5 Diskussion

Es war zu erwarten, dass die Einbrüche in der Performance am größten sind, wenn das Modell mit dem unveränderten Datensatz trainiert wird und dann Samples des verdeckten Datensatzes klassifizieren soll, was auf die im Testset fehlenden Informationen (obere Gesichtspartie) zurückzuführen ist.
Die Performance auf dem Occluded Datensatz ist deutlich höher, wenn es ausschließlich mit diesen Daten trainiert wurde, da das Modell im Training wesentliche Features nur ohne die sichtlich wichtige obere Hälfte des Gesichts erlernen muss. Dennoch bleibt die Performance auf dem normalen Datenset nicht gleich, obwohl dieselben Informationen zur Verfügung stehen, diese aber durch weitere (nicht gänzlich unbekannte[1]) ergänzt werden. Diese Performance-Einbrüche halten sich mit maximal 4,9 % in Grenzen.
Im Hinblick auf eine Generalisierung zeigt sich das Training mit dem Mischdatenset am performantesten. Es zeigt kaum Einbußen im Vergleich zu den speziell mit einem Datensatz trainierten Modellen. Das ist nicht überraschend, da das Mischdatenset alle Daten der jeweiligen Varianten enthält. Soll das Modell also möglichst verdeckte sowie unverdeckte Gesichter erkennen können, ist auch ein Training mit beiden Varianten unerlässlich. Soll ein Modell entweder verdeckte oder normale, unverdeckte Gesichter erkennen, so scheint es sich positiv auszuwirken, dass es im Training beide Varianten desselben Gesichts kennenlernt. Betrachtet man Tabelle 4.3, sieht man, dass obwohl die Testdaten komplett fremd sind, das Modell in beiden Fällen sogar die Performance der speziell trainierten Modelle schlägt. Wird das Modell allerdings vor die Aufgabe gestellt, sowohl verdeckte als auch unverdeckte Gesichter zu erkennen, scheint es beim Vergleich von Tabelle 4.3 mit 4.4 besser zu performen, wenn die im Training verwendeten Gesichter einzigartig bleiben und nicht beide Varianten desselben Gesichts trainiert werden.
Für eine weitere Erhöhung der Generalisierung und bessere Performance wäre es ggf. auch denkbar, dass sich die Verdeckung unterschiedlicher Teile des Gesichts positiv auswirken könnte. So könnte das Modell ggf. robuster ggü. fehlenden Features wie der oberen Gesichtspartie werden.
Auffallend an den in 4 dargestellten Ergebnissen ist, dass der geringste Validation Loss nicht automatisch zur besten Test Accuracy führt. Und auch eine höhere Validation Accuracy ist kein Indikator dafür, dass die Test Accuracy ebenfalls hoch ausfällt. Das in 4.1.1 im Text

[1] Man berücksichtige das Vortraining des ResNet50 durch den VGG-Face-Datensatz.

beschriebene Beispiel, bei dem nach Epoche 16 das beste Ergebnis (Test Accuracy) des gesamten Trainingsdurchlaufs erzielt wird, verdeutlicht dies. Keine der im Training verwendeten Metriken deuten darauf hin, dass dieses Modell die beste Performance erreichen könnte. Erst der Test der einzelnen Epochen-Modelle, hat dies gezeigt.[2] Die Ergebnisse scheinen (mit sehr geringer Abweichung in Tabelle 4.4) zu zeigen, dass längeres Training (trotz höherem Validation Loss oder geringerer Validation Accuracy) eine höhere Accuracy aufweist. Das könnte auch erklären, warum das Training mit dem großen Mischdatensatz die speziell trainierten Modelle leicht übertrifft: Es wird zwar mit denselben Abbruchbedingungen trainiert, allerdings mit der doppelten Menge an Daten pro Epoche. Ob längeres Training tatsächlich hilft, müsste weiter untersucht werden. Zumindest scheint ein Overfitting bei Betrachtung von Abbildung 4.1 noch nicht zu limitieren.
Um abschließend auf die Fragestellung (i) in 1.1.1, wie Partial Face Occlusion die Erkenntnisraten von FER-Modellen beinflusst, zurückzukommen, lässt sich festhalten, dass die Einbußen bei über einem Drittel liegen, sofern das FER-Modell nicht speziell für diesen Occlusion-Task trainiert wird. Dieser Einbruch ist bei den bereits recht schwachen Erkennungsraten von um die 60 % bei unverdeckten Gesichtern und einem Datensatz wie AffectNet sehr hoch. Trotzdem sieht man, dass mit speziellem Training dieser Unterschied deutlich gesenkt werden kann (-8,27 % Test Accuracy Normal vs. Occluded; siehe Tabelle 4.3).
Grundsätzlich stellt sich aber auch die Frage, ob das kategorische Modell nach Ekman und Friesen anhand von statischen Bildern zur Klassifizierung von Emotionen überhaupt geeignet ist, da es bereits beim Einordnen durch den Menschen, wie auch in Mollahosseini et al., 2019 beschrieben, unterschiedliche Wahrnehmungen gibt und nicht jede Emotion auch von jeder Person gleich gelabelt wird.[3] Dennoch wird jedes Einordnungsmodell seine Vor- und Nachteile haben und vielleicht erst die multimodale Facial Expression Recognition für deutlich genauere Erkennungsraten „*in the wild*“ sorgen können.

[2] Da die Modelle der einzelnen Epochen hierzu anhand des Testdatensatzes evaluiert wurden, darf hiernach allerdings nicht optimiert werden, da sonst ein *Information Leak* stattfindet. Die Tests wurden aus reiner Neugier angestellt.

[3] In Mollahosseini et al., 2019 wird unter *3.3 Annotation Agreement* beschrieben, dass zwei Kommentatoren, die 36 000 Bilder gelabelt haben, bei 60,7 % Übereinstimmung stattfand.

Literatur

Bocksch, R. (2023). *Meta führend bei VR-Headsets: Virtual Reality*. Verfügbar 22. Mai 2023 unter https://de.statista.com/infografik/29172/virtual-reality-headsets-im-vergleich/

Cao, Q., Shen, L., Xie, W., Parkhi, O. M., Zisserman, A., Group, V. G., of Engineering Science, D., & of Oxford, U. (2018). VGGFace2: A dataset for recognising faces across pose and age.

Chollet, F. (2021). *Deep Learning with Python, Second Edition*. Manning Publications Co. LLC.

Ekman, P., & Friesen, W. V. (1971). Constants across cultures in the face and emotion. *Journal of Personality and Social Psychology*, *17*(2), 124–129. https://doi.org/10.1037/h0030377

Ernst, H., Schmidt, J., & Beneken, G. (2020). *Grundkurs Informatik Grundlagen und Konzepte für die erfolgreiche IT-Praxis – Eine umfassende, praxisorientierte Einführung: Grundlagen und Konzepte für die erfolgreiche IT-Praxis – Eine umfassende, praxisorientierte Einführung*. Springer Vieweg.

Hall, B. D., Graham, J. M., Cassidy, S. B., & Opitz, J. M. (2009). Elements of morphology: Standard terminology for the periorbital region. *American Journal of Medical Genetics Part A*, *149A*(1), 29–39. https://doi.org/10.1002/ajmg.a.32597

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. https://doi.org/10.48550/ARXIV.1512.03385

Hircin, E., Freyer, T., Nolte, J., & No. (2021, 28. Dezember). *DocCheck Flexikon: Periorbital*. Verfügbar 30. August 2023 unter https://flexikon.doccheck.com/de/Periorbital

Houshmand, B., & Khan, N. M. (2020). Facial Expression Recognition Under Partial Occlusion from Virtual Reality Headsets based on Transfer Learning. *2020 IEEE Sixth International Conference on Multimedia Big Data (BigMM)*, 70–75. https://doi.org/10.1109/bigmm50055.2020.00020

Hu, J., Shen, L., Albanie, S., Sun, G., & Wu, E. (2017). Squeeze-and-Excitation Networks. https://doi.org/10.48550/ARXIV.1709.01507

Iqbal, H. (2020, 17. Januar). *PlotNerualNet: /examples/VGG16*. Verfügbar 30. August 2023 unter https://github.com/HarisIqbal88/PlotNeuralNet

Li, S., & Deng, W. (2022). Deep Facial Expression Recognition: A Survey. *IEEE Transactions on Affective Computing*, *13*(3), 1195–1215. https://doi.org/10.1109/taffc.2020.2981446

Meta Platforms Technologies Ireland Limited. (2023). *Meta Quest 2: Immersives all-in-one VR-Headset*. Verfügbar 22. Mai 2023 unter https://www.meta.com/de/quest/products/quest-2/

Meta Quest 2: Specifications. (n. d.). Verfügbar 22. Mai 2023 unter https://www.uni-giessen.de/de/studium/lehre/projekte/nidit/goals/specifications_quest-2.pdf

Mollahosseini, A., Hasani, B., & Mahoor, M. H. (2019). AffectNet: A Database for Facial Expression, Valence, and Arousal Computing in the Wild. *IEEE Transactions on Affective Computing*, *10*(1), 18–31. https://doi.org/10.1109/taffc.2017.2740923

Pan, B. A. (2022, 15. April). *ResNet50 Tex*. Verfügbar 30. August 2023 unter https://github.com/HarisIqbal88/PlotNeuralNet/issues/24#issuecomment-1099801603

Raschka, S., Liu, Y. (, Mirjalili, V., & Dzhulgakov, D. (2022). *Machine Learning with Pytorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Packt Publishing, Limited.

TensorFlow API v2.13.0: tf.keras.utils.image_dataset_from_directory. (2023, 20. August). Verfügbar 26. August 2023 unter https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory

TensorFlow Guide (JS): Models and layers. (2022, 1. November). Verfügbar 15. August 2023 unter https://www.tensorflow.org/js/guide/models_and_layers#model_summary

Anhang

Zeiterfassung

Erfasst wurde die produktive Arbeit am Projekt. Modelltrainingszeiten wurden nur dann erfasst, wenn es sich um kleinere Trainings zum Testen kleiner Parameteränderungen handelte und der Prozess genau beobachtet wurde oder wenn währenddessen Recherche zum Optimieren stattfand. Trainingszeiten der großen Modelle, die mit dem kompletten Datensatz teilweise über Nacht trainiert wurden, sind nicht aufgeführt.

Tabelle I: Zeiterfassung der Projektarbeit

Nr.	**Datum**	**Start**	**Ende**	**Pause**	**Bereich***	**Dauer**	
	dd.mm.yyyy	hh:mm	hh:mm	hh:mm	siehe auch Tab. II	hh:mm	Std.
1	24.04.2023	10:30	11:00	-	Sonstige	00:30	0,50
2	03.05.2023	11:00	18:00	01:00	Recherche	06:00	6,00
3	04.05.2023	09:30	16:00	01:00	Recherche	05:30	5,50
4	08.05.2023	10:30	11:00	-	Projektmanagement	00:30	0,50
5	08.05.2023	13:00	15:30	-	Projektmanagement	02:30	2,50
6	11.05.2023	15:00	17:30	-	Vorbereitung	02:30	2,50
7	16.05.2023	11:00	15:00	-	Projektmanagement	04:00	4,00
8	16.05.2023	17:15	19:30	-	Vorbereitung	02:15	2,25
9	17.05.2023	10:00	21:15	03:00	Vorbereitung	08:15	8,25
10	18.05.2023	10:00	14:00	-	Vorbereitung	04:00	4,00
11	19.05.2023	17:00	20:00	-	Umsetzung	03:00	3,00
12	20.05.2023	12:45	16:15	-	Umsetzung	03:30	3,50
13	22.05.2023	13:30	16:00	-	Umsetzung	02:30	2,50
14	23.05.2023	10:00	15:00	-	Dokumentation	05:00	5,00
15	24.05.2023	11:00	16:00	-	Umsetzung	05:00	5,00
16	25.05.2023	13:00	17:00	-	Umsetzung	04:00	4,00
17	30.05.2023	10:45	14:30	-	Projektmanagement	03:45	3,75
18	01.06.2023	11:30	15:30	-	Vorbereitung	04:00	4,00

Fortsetzung auf nächster Seite

Tabelle I: Fortsetzung von vorheriger Seite

Nr.	Datum	Start	Ende	Pause	Bereich*	Dauer	
	dd.mm.yyyy	hh:mm	hh:mm	hh:mm		hh:mm	Std.
19	02.06.2023	16:00	18:00	-	Recherche	02:00	2,00
20	04.06.2023	15:30	17:30	-	Recherche	02:00	2,00
21	05.06.2023	14:00	16:00	-	Recherche	02:00	2,00
22	06.06.2023	12:00	13:00	-	Recherche	01:00	1,00
23	07.06.2023	14:00	18:00	-	Umsetzung	04:00	4,00
24	08.06.2023	11:00	17:30	01:00	Recherche	05:30	5,50
25	09.06.2023	09:00	13:00	-	Recherche	04:00	4,00
26	12.06.2023	13:00	18:30	-	Umsetzung	05:30	5,50
27	13.06.2023	13:00	15:00	-	Umsetzung	02:00	2,00
28	14.06.2023	14:00	19:00	-	Umsetzung	05:00	5,00
29	15.06.2023	12:00	20:00	01:00	Vorbereitung	07:00	7,00
30	19.06.2023	14:00	16:00	-	Umsetzung	02:00	2,00
31	20.06.2023	16:00	20:00	-	Umsetzung	04:00	4,00
32	21.06.2023	13:00	16:00	-	Umsetzung	03:00	3,00
33	22.06.2023	10:00	16:00	00:45	Umsetzung	05:15	5,25
34	23.06.2023	10:30	18:00	01:00	Umsetzung	06:30	6,50
35	24.06.2023	11:00	13:30	-	Umsetzung	02:30	2,50
36	25.06.2023	13:00	21:00	01:00	Umsetzung	07:00	7,00
37	26.06.2023	10:30	16:15	01:00	Umsetzung	04:45	4,75
38	27.06.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
39	28.06.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
40	04.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
41	05.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
42	06.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
43	07.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
44	10.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
45	11.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
46	12.07.2023	10:00	20:00	02:00	Umsetzung	08:00	8,00
47	13.07.2023	07:00	20:00	02:00	Umsetzung	11:00	11,00
48	14.07.2023	06:00	16:00	03:00	Umsetzung	07:00	7,00
49	15.07.2023	05:30	06:15	-	Projektmanagement	00:45	0,75
50	22.07.2023	11:00	16:30	-	Dokumentation	05:30	5,50
51	28.07.2023	15:00	19:00	-	Umsetzung	04:00	4,00
52	15.08.2023	08:30	12:30	-	Dokumentation	04:00	4,00
53	16.08.2023	09:00	12:00	-	Dokumentation	03:00	3,00

Fortsetzung auf nächster Seite

Tabelle I: Fortsetzung von vorheriger Seite

Nr.	**Datum**	**Start**	**Ende**	**Pause**	**Bereich***	**Dauer**	
	dd.mm.yyyy	hh:mm	hh:mm	hh:mm		hh:mm	Std.
54	24.08.2023	11:00	16:30	01:30	Dokumentation	04:00	4,00
55	25.08.2023	10:00	14:30	-	Dokumentation	04:30	4,50
56	26.08.2023	12:00	15:45	00:30	Dokumentation	03:15	3,25
57	29.08.2023	12:00	17:45	02:00	Dokumentation	03:45	3,75
58	30.08.2023	10:00	14:00	-	Dokumentation	04:00	4,00
59	11.09.2023	15:00	16:00	-	Dokumentation	01:00	1,00
60	19.09.2023	10:00	14:00	-	Umsetzung	04:00	4,00
61	20.09.2023	14:00	17:00	-	Umsetzung	03:00	3,00
62	26.09.2023	11:00	16:15	-	Dokumentation	05:15	5,25
63	27.09.2023	11:30	17:00	-	Dokumentation	05:30	5,50
64	28.09.2023	10:30	11:00	-	Dokumentation	00:30	0,50
65	28.09.2023	15:15	16:00	-	Sonstige	00:45	0,75
66	10.10.2023	22:30	23:00	-	Umsetzung	00:30	0,50
67	11.10.2023	11:30	17:30	-	Dokumentation	06:00	6,00
Gesamt:						**295,00 Std.**	

**Kommentare zur groben Beschreibung siehe nachfolgende Tabelle II*

Tabelle II: Kommentare zur Zeiterfassung

Nr.	Kommentar
1	Absprache Larissa & Thorben Projektmöglichkeiten
2	Papers und Chollet lesen
3	Papers und Chollet lesen
4	Absprache Larissa & Thorben Projektziel
5	LaTeX Vorlage aufsetzen und Projektziel beschreiben
6	AffectNet herunterladen und sichten, Conda & PyPackages aktualisiseren
7	Dokumentation in LaTeX strukturieren, Literatur organisieren, Dokumentation anfangen
8	TensorFlow Installationsprobleme
9	TensorFlow Installation; Docker-Image herunterladen & customizen (Pandas, OpenCV), Stichpunkte Doku, AffectNet Exploration mit Python
10	Auslesen & plotten Facial Landmarks, Recherche 68 point system
11	Tests mit Facial Landmarks
12	Rectangle Occlusion Patch Tests
13	Rectangle Occlusion Patch Tests
14	Dokumentation LaTeX Rectangle Calculation
15	Pipeline Tests & Pipeline Anwendung
16	Optimierung Pipeline-Code: Batch-Processing mit Checkpoint, Pipeline Fortsetzung
17	Kurze Absprache mit Thorben zum Stand und weiteren Vorgehen, Train-Set Occlusion Processing
18	Neues Python Virtual Env aufsetzen und Tensorflow whl installieren; Tensorflow Install-Test, Chollet lesen
19	Chollet
20	Neural Network Beispiele testen (Chollet)
21	Neural Network Beispiele testen (Chollet)
22	Chollet lesen
23	Chollet, Beispiele ausprobieren, Recherche
24	Convnet Tests Chollet
25	Convnet Klassifizierungstests
26	ConvNet bauen und erste Klassifizierung von AffectNet-Set
27	ConvNet Experimente Colab
28	Klassifikationstests (CPU), Generatoren für Batch-Verarbeitung verwenden, Mem-Probleme versuchen zu lösen
29	Modell-Tests, OpenSUSE Leap-Installation, Tensorflow Linux Installation, Datensets kopieren, sortieren
30	TF-Dataset verwenden und Convnet-Modell neu trainieren

Fortsetzung auf nächster Seite

Tabelle II: Fortsetzung von vorheriger Seite

Nr.	Kommentar
31	Convnet-Tests, um schlechte Validation Accuracy zu reduzieren, Chollet & mehr lesen
32	Convnet-Tests, um schlechte Validation Accuracy zu reduzieren, Chollet & mehr lesen
33	Convnet-Tests, um schlechte Validation Accuracy zu reduzieren, Chollet & mehr lesen
34	Convnet-Tests, um schlechte Validation Accuracy zu reduzieren, Chollet & mehr lesen
35	Convnet-Tests, um schlechte Validation Accuracy zu reduzieren, Chollet & mehr lesen
36	Chollet lesen, Recherche, Fehlersuche ConvNet, Korrektur Datasets, Parameter- und Modelltests, Doku
37	Zeiterfassung ergänzen, Modelltests & Optimierungen
38	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
39	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
40	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
41	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
42	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
43	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
44	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
45	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
46	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
47	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
48	Training, Evaluierung, Optimierungen; Recherche zu weiteren Maßnahmen
49	Zeiterfassung, Modelltraining prüfen und evaluieren, in Colab übertragen
50	Dokumentation: Allgemeines, Vorbereitung
51	Fix fehlendes Bild in Occluded Dataset, Occ Dataset in Klassenordner sortieren, Upload Drive und Vorbereitung Colab
52	Doku DNNs
53	Doku Code, DNNs
54	Doku Code, DNNs
55	Doku Code, DNNs
56	Doku Code, DNNs
57	Doku Code, DNNs
58	Doku Code, DNNs
59	Doku Code, DNNs
60	Tests, Retraining, Training Occluded
61	Mix Occluded mit Normal Dataset: 50/50, Train 50/50, Evaluierung
62	Dokumentation, Umsetzung, Ergebnisse
63	Dokumentation Ergebnisse, Diskussion
64	Korrekturen & kleine Ergänzungen Doku

Fortsetzung auf nächster Seite

Tabelle II: Fortsetzung von vorheriger Seite

Nr.	Kommentar
65	Gespräch mit Thorben Ortmann
66	Mixed Training mit Complete Datasets vorbereiten, Trainingsprozess starten
67	Auswertung & Doku Mixed Training, Korrekturen Dokumentation