Städtisches Jakob-Fugger-Gymnasium
Qualifikationsphase 2020/2022

Seminararbeit

Thema: Deepfakes im Bereich Fotografie

Verfasser der Seminararbeit: Elias Kohout
Titel des Seminars: Bildgebende Verfahren in der Physik
Seminarleiter: StD Jörg Haas
Abgabetermin: 9.11.2021

Abgegeben am .....
Abschlusspräsentation abgelegt am .....

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 1	
				Summe:	
				Gesamtleistung nach § 29 (7) GSO = Summe:2 (gerundet)	

Unterschrift des Seminarleiters: .....

Inhaltsverzeichnis

1 Einleitung

Technologie bestimmt unser Leben, insbesondere die fortschreitende Digitalisierung dringt in immer mehr Bereiche unseres Lebens vor und damit geht einher, dass immer mehr Daten über uns gesammelt werden. Riesige Unternehmen wie Google oder Facebook verdienen Unmengen an Geld mit dem Sammeln und Verarbeiten von enormen Datenmengen. Dies geht offensichtlich über das Erstellen von Diagrammen hinaus. Ein großer Bereich der Informatik, welcher von diesen Entwicklungen profitiert, ist der des maschinellen Lernens.

Dies ist insbesondere der Fall, wenn es um künstliche *Neuronale Netze* geht, die vor allem mit großen Datenmengen gefüttert werden, um in diesen abstrakte Muster zu erkennen. Hierbei wird oft die Methode des *Deep Learning* verwendet. *Neuronale Netze*, die diesem Schema entsprechen, können für zahlreiche Anwendungsaufgaben verwendet werden, bei denen man mit konventioneller Programmierung nicht weiter kommt. Solche Netze werden zum Beispiel für das Klassifizieren oder Segmentieren von Bildern, verbesserte Suchergebnisse bei Suchmaschinen, Vorschläge in Sozialen Medien, dem Erkennen von Gesichtern und vielen weiteren Anwendungen verwendet. Die Nutzung solcher Netze für das Erstellen von *Deepfakes* ist das Thema dieser Seminararbeit.

Der Begriff *Deepfake* wird seit 2017 [Ngu+21] verwendet und hat seit dem für recht viel Aufmerksamkeit gesorgt. Diese Art der Neuronalen Netzen ermöglicht es das Gesicht einer bestimmten Person auf das einer anderen Person in einem Video zu projizieren und damit theoretisch die projizierte Person alles sagen oder machen zu lassen. Diese Art des Fälschens von Videos ist generell nichts Neues, das Revolutionäre ist die Einfachheit, mit der solche Videos erstellt werden können. Da Programme mit den richtigen künstlichen Neuronalen Netzen frei und kostenlos im Internet zu haben sind, ist nicht mehr nötig als ein Computer mit einer handelsüblichen Grafikkarte, Video- oder Bildmaterial der Person und etwas Zeit, um ein *Deepfake*-Video zu erstellen.

Dass dies Probleme verursachen kann, ist offensichtlich. Heute kann man in den meisten Fällen noch erkennen, dass es sich um ein gefälschtes Video handelt, die Technologie wird jedoch unaufhaltsam besser werden und es wird der Tag kommen, an dem man mit dem Auge die gefälschten Videos nicht mehr von den Echten unterscheiden kann. Die politischen Gefahren sollten jedem klar sein, aber es geht noch weiter. Identitätsdiebstahl mit der Hilfe von *Deepfakes* ist bereits heute ein Problem. Ein sehr großer Teil der *Deepfakes* ist pornografischer Natur und man stelle sich vor wie leicht man das Leben einer Person durch ein virales Video zur Hölle machen kann. Und was ist mit Videomaterial von Überwachungskameras, das vor Gericht standhalten muss und sowieso häufig eine niedrige Auflösung hat, um nur ein paar Beispiele zu nennen.

Da kommt die Frage auf, wie man dem entgegenwirken kann? Eine Möglichkeit, und es gibt bereits Ansätze dafür, wäre *Deepfakes* mit künstlicher Intelligenz zu erkennen. Aber bevor man daran arbeiten kann, muss man *Deepfakes* verstehen und eine generelle Vorsicht und Skepsis vor dem was man im Internet an Bildern und Videos sieht, würde sicherlich auch helfen. Die folgende Arbeit soll genau das tun, also ein Verständnis von der Funktionsweise von Neuronalen Netzen im Allgemeinen, sowie im Speziellen der Funktionsweise und dem Aufbau eines künstlichen *Neuronalen Netzes* zur Erstellung von *Deepfakes* auch anhand von praktischen Beispielen vermitteln.

Begonnen wird mit der Erklärung der grundlegenden Begriffe und Konzepte, die für das Verständnis des weiteren Verlaufs nötig sind. Hierzu zählen unter anderem die Funktion und Implementierung eines künstlichen *Neuron*, eines *Neuronalen Netz*, sowie die Definition und Erklärung von *Convolutional Neural Networks* und *Autoencodern*. Es folgt die detaillierte Beschreibung des Prozesses der Erstellung eines künstlichen *Neuronalen Netzes* zur Erstellung von *Deepfakes* beziehungsweise die tatsächliche Erstellung eines solchen *Deepfake* anhand einer selbst konstruierten Implementierung.

2 Künstliche Neuronale Netze

2.1 Grundlagen

Künstliche *Neuronale Netze (KNN)* sind aus der Bionik heraus entstanden. Die *Bionik* verbindet Biologie und Technik und ahmt so die Natur und das Leben nach. Die thematisierte Technologie ist Nervenzellen und dem Gehirn nachempfunden. Im Folgenden werden nun die Grundlagen von neuronalen Netzen verdeutlicht, dabei werden teilweise vereinfachte Darstellungen verwendet. Für mein Modell verwende ich eine Bibliothek, also Programmcode, der von jemand anderem zur freien Verwendung zugänglich gemacht wurde, namens TensorFlow, die den eigentlichen Bau des *KNN* vereinfacht.

2.2 Grenzen der herkömmlichen Programmierung

Zunächst sollte klargestellt werden, warum diese außergewöhnliche Technologie notwendig ist. Wenn man sich in der Informatik mit dem Erstellen von Gesichtern als Bilder befasst, gerät man mit herkömmlicher Programmierung schnell an seine Grenzen. Denn wie bringt man einem Computer bei, was genau ein Gesicht ist? Für uns Menschen ist das Erkennen von Gesichtern intuitiv, da vergisst man leicht, was für eine enorme Leistung das eigentlich ist. Denn kein Gesicht sieht gleich aus und

dennoch erkennen wir sie. Es geht noch weiter, auch bei einem schrägen oder umgedrehten Gesicht, bei unterschiedlichen Lichtverhältnissen oder wenn wir uns einen Smiley anschauen erkennen wird dennoch ein Gesicht. Computer lernen jedoch nicht wie wir Menschen, man muss ihm ganz konkret definieren, was ein Gesicht ist. Computer speichern alle Daten als Zahlenwerte und führen nur streng definierte Befehle aus. Wie sagen wir also einem Computer, was das recht abstrakte Konzept eines Gesichts ist?

2.3 Lösungen durch künstliche Neuronale Netze

Bei der Lösung orientiert man sich an der Natur und modelliert die Strukturen im Gehirn. Es wird aus Modellen von Neuronen ein künstliches neuronales Netz konstruiert. Dieses ermöglicht bei richtiger Konfiguration zum Beispiel das Erstellen von Gesichtern.

Wenn wir versuchen würden einem Computer ein Gesicht zu beschreiben, könnten wir damit anfangen zu sagen, es gibt zwei Augen und einen Mund. Der Computer weiß aber nicht was das ist, wir müssen also auch ein Auge definieren. Hier könnte man anbringen das es sich hierbei, in einem Bild, um Kreise, Ellipsen und andere Geometrische Formen handelt. Diese wiederum werden durch Kanten zwischen verschiedenen Farben definiert.

Betrachtet man das Innere, also die Funktionsweise, des *KNN* lässt sich hier ein ähnlicher Abstraktionsprozess erkennen, sofern das *KNN* funktioniert also richtig konfiguriert wurde. Da ein solches Netz viele Millionen Parameter haben kann, wird die Konfiguration nicht von Hand gemacht, sondern durch einen Algorithmus. Dieser Prozess wird dann als Lernen bezeichnet.

2.4 Das künstliche Neuron

Wie also ist der kleinste Teil des neuronalen Netzes, das künstliche Neuron, aufgebaut? Es handelt sich hierbei nur um eine mathematische Formel, deren Struktur allerdings einem tatsächlichen biologischen Neuron ähnelt. Man sollte also zunächst ein biologisches Neuron verstehen. Im Zentrum steht der Zellkörper, an dem sich zum Einen die Dendriten und zum Anderen das Axon mit den Synapsen befindet. Die Dendriten sind quasi die Eingabe des Neuron, hier nimmt das Selbe elektrische Signale von anderen Neuronen auf. Überschreitet dabei die Summe der Eingangssignale einen gewissen Wert, gibt die Zelle selbst ein Signal am Axon ab.

Dieses Prinzip wird nun ähnlich in der Sprache der Mathematik abgebildet und in der Informatik implementiert. Als Eingabe wird hierbei eine Reihe von Zahlen verwendet. Diese lassen sich als mehrdimensionaler Vektor darstellen, er wird hier als $\vec{x}$ bezeichnet. Da die Eingabewerte von unterschiedlicher Wichtigkeit sein können,

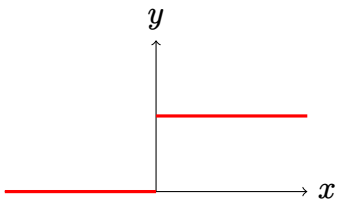


Abbildung 1: Stufenfunktion

werden sie ihrer Bedeutung nach gewichtet. Dazu wird ein weiterer Vektor an Zahlen, mit der gleichen dimensionalen Größe wie $\vec{x}$, verwendet, der hier $\vec{w}$ genannt. Die Werte von den beiden Vektoren werden nun einzeln miteinander multipliziert und anschließend werden die Ergebnisse zu einer Zahl addiert.

$$\vec{x} \circ \vec{g} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} \circ \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \end{pmatrix} = (x_1 \cdot w_1) + (x_2 \cdot w_2) + (x_3 \cdot w_3) + \dots$$

Dies wird auch als Skalarprodukt bezeichnet. Dabei haben Zahlen mit einem großen Gewichtungsfaktor einen größeren Einfluss auf das Ergebnis als Zahlen mit einem kleineren Gewichtungsfaktor.

Um die Hemmschwelle, die in einem Neuron überschritten werden muss, damit dieses auch ein Signal abgibt, zu implementieren, wird eine mathematische Funktion verwendet. Diese wird Aktivierungsfunktion genannt. Eine sehr simple Variante ist die Stufenfunktion, die im Falle des Überschreitens eines Wertes *eins* ergibt, andernfalls *null* [SS20a].

Man ist hier jedoch nicht nur auf diese Funktion beschränkt. Die beiden Berechnungen lassen sich nun kombinieren und als mathematische Formel ausdrücken. Hierbei ist f_a die Aktivierungsfunktion und y das Ergebnis, welches als Eingabe für das nächste Neuron verwendet werden kann.

$$f_a(\vec{x} \circ \vec{w}) = y$$

2.5 Aufbau eines künstliches Neuronales Netz

Um ein funktionierendes *KNN* zu erhalten, müssen nun mehrere Neuronen verbunden werden. Dies geschieht in Schichten oder *Layers*. Die erste Schicht nimmt die Eingabe auf und gibt sie an die nächste Schicht weiter.

In Tensorflow gibt es ein Modul, das sich um die Schichten kümmert, es heißt *keras.layers*. Hier besteht die erste Schicht an Neuronen aus drei künstlichen Neu-

ronen und die Eingabe besteht aus drei Zahlenwerten. Die letzte Schicht entspricht auch gleich der Größe der Ausgabe, in diesem Fall zwei Zahlenwerte.

```
modell = tensorflow.keras.Sequential(name='Testmodell')
modell.add(tensorflow.keras.layers.Dense( 3, input_shape=(3,) ))
modell.add(tensorflow.keras.layers.Dense( 2 ))
```

Dense steht hierbei für eine recht häufig verwendete Art von Schicht, bei der jedes Ergebnis der vorherigen Schicht als Eingabe für jedes einzelne Neuron dieser Schicht verwendet wird. Die Neuronen sind vollständig verbunden. Das *KNN* könnte man also graphisch so darstellen:

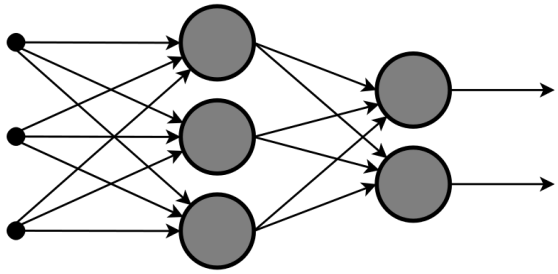


Abbildung 2: *Beispielhafte Struktur eines KNN*

3 Deepfake

3.1 Sammeln und Aufbereiten der Daten

Damit das *KNN* die Ergebnisse liefert, die man erwartet, muss man es mit den richtigen Daten füttern. Wie genau dieser Prozess, der auch als Training bezeichnet wird, funktioniert wird noch erläutert. Wichtig ist, es werden zahlreiche, und das heißt mehrere Tausende oder sogar Zehntausende, Bilder von Gesichter der beiden Personen, deren Gesichter schlussendlich getauscht werden sollen, benötigt. Man beschränkt sich hierbei nur auf den Ausschnitt eines Bildes, das Gesicht, um die Komplexität der Aufgabe gering zu halten. Der wohl einfachste Weg viele Bilder zu finden ist, sie aus einem Video zu extrahieren.

3.1.1 Gesichtserkennung

Wenn man sich dann ein Video zum Beispiel von YouTube heruntergeladen hat, geht es daran die Gesichter in diesem Video zu erfassen. Hierbei kann auch eine Art des maschinellen Lernens verwendet werden, welche mit Kaskaden die Gesichter in den Bilder lokalisiert [VJ01]. Da es dennoch oft schwer ist Videos mit ausschließlich einer Person zu finden, müssen die Gesichter, welche man haben möchte, noch bestätigt werden. Dies wird mit einer Bibliothek namens *face_recognition* umgesetzt. Dabei

wird das gefundene Bild mit einem Bild der gesuchten Person verglichen. Ähneln sie sich ausreichend kann davon ausgegangen werden, dass es sich um Gesichter der selben Person handelt.

3.1.2 Gesichtsextrahierung

Führ man nun dieses Programm aus, wird ein Gesicht in dem Video gefunden und validiert, anschließend ausgeschnitten und in einer Datei auf der Festplatte gespeichert. Dies wird für ein ausreichend langes Video einmal durchgeführt damit später darauf zugegriffen werden kann.

3.2 Aufbau des Neuronalen Netz

Damit eine sinnvolle Struktur für das *KNN* erstellt werden kann, muss zunächst das Ziel festgelegt werden. Hier ist die Aufgabe das aus dem Gesicht von Person A das Gesicht von Person B mit dem selben Gesichtsausdruck kreiert wird. Um dies umzusetzen wird zunächst das Gesicht von Person A auf verhältnismäßig wenige Zahlenwerte reduziert. Diese Werte sollten dann den Gesichtsausdruck widerspiegeln. Daraus wird dann ein Gesicht von Person B konstruiert.

3.2.1 Encoder und Decoder

Mit dem *Encoder* werden die Bilder in Zahlenwerte umgewandelt, mit dem *Decoder* wird daraus wieder ein Bild konstruiert. Die Zahlenwerte sind von der enthalten Datenmenge deutlich kleiner als die Bilder selbst, um das *KNN* zu zwingen, Muster in den Bildern zu erkennen. Es werden für jede Person ein eigener *Decoder* erstellt und Trainiert, der *Encoder* ist jedoch der Gleiche. Dies ist der Fall, um sicherzustellen, dass für jedes Bild dieselben Muster erkannt und damit die Informationen der Zahlenwerte für jeden der beiden Decoder verständlich sind.

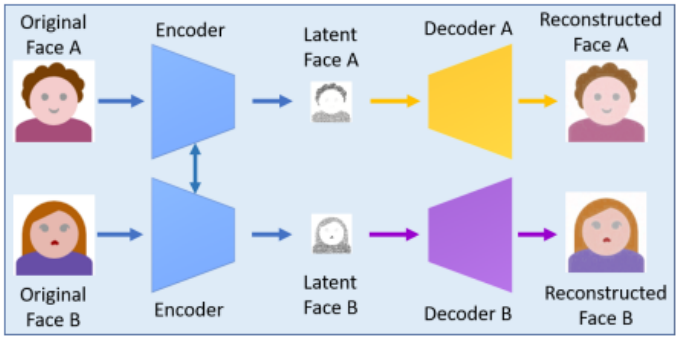


Abbildung 3: *Encoder und Decoder*

3.2.2 Autoencoder

Zum Trainieren werden der *Encoder* und *Decoder* zu einem *Autoencoder* zusammengefasst. Dieser komprimiert dann ein Bild und versucht es möglichst genau wieder aufzubauen. Dies wird für beide Decoder gemacht. Wenn dann der Zeitpunkt kommt zu dem man das Gesicht fälscht, wird dieses durch den *Encoder* geschickt und anschließend mit dem jeweils anderen *Decoder* wieder rekonstruiert [**DLeaningForDeepF˙s2f**].

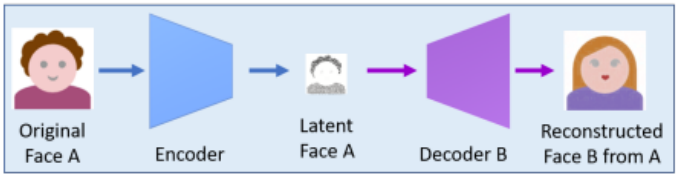


Abbildung 4: *Fälschen von Gesichtern mit Encoder und Decoder*

3.2.3 Convolutional Neural Networks

Bei einem wie zuvor beschrieben simplen *KNN* muss das Bild zur Eingabe in eine Liste von Zahlen umgewandelt werden. Hierzu wird jede Reihe des Bildes als eine Liste von Zahlen übergeben. Dabei geht jedoch weitestgehend ein Verständnis über die räumliche Struktur des Bildes verloren. Um dabei Abhilfe zu schaffen kann man *Convolutional Neural Networks* verwenden. Diese Art der *KNN* integriert die Bildverarbeitung mit Filtermatrizen (eng. *convolution matrix*).

Filtermatrizen werden in der Bildverarbeitung dazu verwendet, zahlreiche Effekte auf Bilder anzuwenden. Es handelt sich hierbei um simple Filter wie zum Beispiel *Blur* oder Kantenerkennung. Der Funktion dieser liegt dabei eine Zahlenmatrix zugrunde, welche beliebig groß sein kann, wobei sie jedoch kleiner als das Bild sein sollte.

+1	0	-1
+2	0	-2
+1	0	-1

Abbildung 5: *Beispielhafte Filtermatrix*

Diese Matrix – nehmen wir in diesem Beispiel an, sie hat ein Größe von 3x3 – wird dann auf ein zu verarbeitendes Bild gelegt, sodass neun Pixel bedeckt werden. Nun wird jeder Wert der Matrix mit dem darunterliegenden Wert des Pixel multipliziert. Alle diese Produkte werden addiert und das Ergebnis ist der Pixelwert des Ergebnisbildes an dieser Stelle. Anschließend ist es noch sinnvoll das Bild zu normalisieren, um negative oder zu große Zahlenwerte für die Pixel zu verhindern. Diese Schritte werden für das gesamte Bild durchgeführt bis ein fast gleichgroßes

gefiltertes Bild entsteht. Je nachdem wie man die Filtermatrix konfiguriert lassen, sich andere Effekte erzielen. Die in Abbildung 5 dargestellt Matrix stellt vertikale Kanten heraus. Dies sieht dann, wenn es auf ein Bild angewendet wird, wie in Abbildung 6 aus.

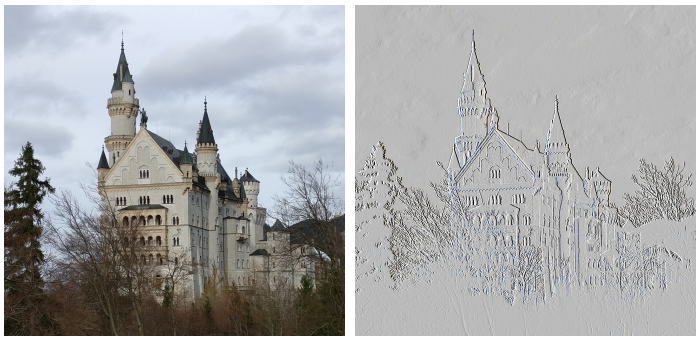
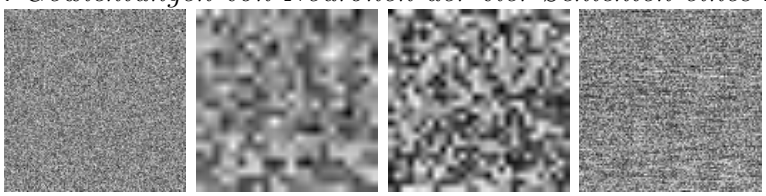


Abbildung 6: *Effekt der Filtermatrix*

Dieses Verfahren eignet sich äußerst gut, um Strukturen in Bilder zu erkennen, weshalb es auch für *Deepfakes* in den *Convolutional Neural Networks* Verwendung findet. Beim *Decoder* wird der gesamte Prozess mehr oder weniger umgedreht um Strukturen aufzubauen. Wenn man ein solches Netz trainiert lässt man den Computer darüber entscheiden welche Werte für die Filtermatrizen ausgewählt werden. Betrachtet man ein einfach verknüpftes *KNN* von innen, sieht dies sehr chaotisch aus. Abbildung 7 stellt genau das dar, dunkle Pixel sind negative Werte, graue Pixel entsprechen Werten nahe Null und Weiße Pixel sind große positive Werte.

Abbildung 7: *Gewichtungen von Neuronen der vier Schichten eines Autoencoder*



Obwohl es theoretisch Muster in den Bildern erkennen sollte, geschieht dies auf eine so komplexe Art und Weise, dass es für uns Menschen völlig unverständlich ist. Es funktioniert, ist aber sicherlich nicht sehr effizient.

Betrachtet man, was mit Bildern geschieht, wenn sie von einem *Convolutional Neural Network* verarbeitet werden, erscheint dies deutlich strukturierter.

3.3 Trainieren des Netzes

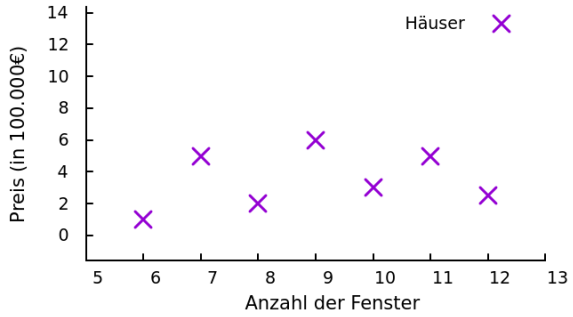
Zum Verbessern der Leistung des Netzes wird es mit Bildern gefüttert. Zu jedem Bild wird zur Überprüfung auch das erwartete Bild mitgeliefert, quasi die Lösung. Im diesem Fall, da es sich um einen *Autoencoder* handelt, sind beide Bilder das Selbe. Allgemeinen sei noch erwähnt, dass das gesamte künstliche Neuronale Netz

mathematisch als Funktion beschriebenen werden kann, die für jedes Gewicht im Netz ein Parameter hat.

3.3.1 Bewertung des Netzes

Damit das künstliche Neuronale Netz trainiert, also in seiner Funktion verbessert werden kann, muss es zunächst bewertet werden. Dies wird mit einem errechneten Wert namens *Fehler* oder *Loss* umgesetzt. Der *Loss* wird mit einer Funktion berechnet, die variiert werden kann. In dem hier beschriebenen Fall wird eine Funktion namens *Mean Squared Error* [Goo21a] verwendet. Die eben genannte Funktion vergleicht das Ergebnis des *KNN* mit den erwarteten Werten und beurteilt wie sehr diese beiden sich ähneln. Für ein besseres Verständnis hier ein Beispiel. Stellen wir uns, vor wir haben ein *KNN* mit einem Neuron und zwei Parametern, einem Eingabewert und einem Ausgabewert und es wird versucht Hauspreise auf Basis der Anzahl der Fenster zu bestimmen. Wir haben also als Eingabe die Anzahl der Fenster und als Ausgabe den Hauspreis in 100.000€. Die Daten zu den Häusern können in einem Diagramm wie in Abbildung 8 dargestellt werden.

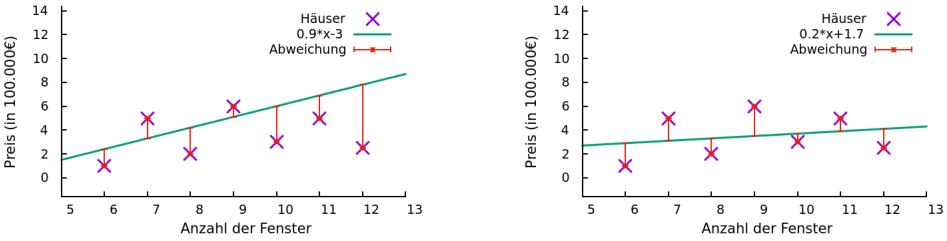
Abbildung 8: *Beispiel Hauspreisanalyse (erfundene Daten)*



Das Neuron kann mit seinen zwei Parametern, wovon einer als *Bias* nur addiert wird, als lineare Funktion in dem Format $f(x) = w_1 \cdot x + w_2$ dargestellt werden. Es ist nun das Ziel des Lernalgorithmus die Gewichte w_1 und w_2 so anzupassen, dass das Neuron eine möglichst genaue Aussage über die Hauspreise machen kann. Zu Beginn wird das Neuron mit zufälligen Werten initialisiert, nehmen wir beispielsweise die Werte $w_1 = 0,9$ und $w_2 = -3$ wie es auf der linken Seite von Abbildung 9 dargestellt wird. Der *Fehler* wird nun aus der Differenz des tatsächlichen Hauspreises und dem mit dem Neuron errechneten Wert gebildet. Um bei der Subtraktion der beiden Werte keine Negative Ergebnisse zu erhalten und große Abweichungen stärker zu gewichten wird der Wert quadriert.

Da mit mehreren Werten gerechnet wird, wird anschließend der Durchschnitt gebildet. Der *Fehler*-Wert der linke Funktion in Abbildung 9 lässt sich also so berechnen:

Abbildung 9: *Beispiel Hauspreisanalyse mit Funktionen*



$$\frac{(f(6)-1)^2+\ldots+(f(12)-2,5)^2}{7} \approx 7,3$$

Die Funktion $g(x) = 0,2x + 1,7$ auf der rechten Seite von Abbildung 9 hat einen *Fehler*-Wert von nur ungefähr $2,8$ und ist damit deutlich akkurater als die zuvor dargestellte Funktion.

Was hier für jedes Haus gemacht wurde, wird bei dem *Deepfake KNN* mit jedem Bild gemacht.

Das Neuronale Netz ist also um so besser, je niedriger der *Fehler*-Wert ist.

3.3.2 Der Lernprozess

Der in Kapitel 3.3.1 beschriebene Prozess des Findens von dem *Fehler*-Wert kann auch als Funktion beschrieben werden. Und da das *KNN* für jede Kombination von Werten für die Parameter einen *Fehler*-Wert hat, kann man diese *Fehlerfunktion* auch graphisch darstellen. Dies wird möglich indem man den *Fehler* für alle Kombinationen berechnet. Für das obige Beispiel mit zwei Parametern ist dies noch möglich, man erhält einen dreidimensionalen Graphen. Es ist nun das Ziel des Lernens, ein Minimum in der *Fehlerfunktion* zu finden. Es wäre jedoch ineffizient eine Großzahl von *Fehler*-Werten wahllos zu ermitteln und sich das Beste auszusuchen. Deshalb wird für einen *Fehler*-Wert die Steigung der *Fehlerfunktion* in diesem Punkt berechnet. Mit der lokalen Ableitung kann die Richtung und Stärke bestimmt werden, mit der der Parameter, bzw. die Gewichtung w verändert werden muss, um einem Minimum näher zu kommen. Dabei wird für jeden Parameter eine partielle Ableitung bestimmt. Der Parameter wird dann um den Wert der Steigung g reduziert. Um hier nun noch ein bisschen mehr Kontrolle über die Geschwindigkeit der Änderungen zu haben wird der Wert der Steigung noch mit der zuvor festgelegten Lernrate L multipliziert [SS20b]. Dieser Prozess wird für jeden Lernschritt wiederholt.

$$w_{neu} = w_{alt} - L \cdot g$$

Dieser Prozess des *Ändern der Gewichte* ist die Aufgabe des *Optimizers*, welcher

auch frei wählbar ist. Der wohl Simpelste, der außerdem nach der oben beschriebenen Formel funktioniert, heißt *Stochastic Gradient Descent (SGD)* [Goo21c].

3.3.3 Versuchsreihe zur Lernrate

Bei dem Erstellen eines KNN kann an zahlreiche Stellschrauben gedreht werden. Sehr wichtig für erfolgreiches Trainieren des Netzes ist die Lernrate. Um den Einfluss dieser zu demonstrieren und schlussendlich einen Wert in der richtigen Größenordnung zu wählen, wurde eine Versuchsreihe mit verschieden Lernraten und *Optimizern* durchgeführt. Hierzu wurde folgendes Modell verwendet.

Schichttyp	**Eingabeform**	**Ausgabeform**	**Parameter**
Reshape	120, 120, 3	43200	0
Dense	43200	500	21600500
Dense	500	100	50500
Dense	100	500	50500
Dense	500	43200	21643200
Reshape	43200	120, 120, 3	0

Tabelle 1: *Struktur des Modells zu der Versuchsreihe zur Lernrate*

In der ersten Spalte von Tabelle 1 wird angegeben, um was für eine Art von Schicht es sich handelt. Die zweite Spalte gibt an in welchem Format und zu welcher Anzahl die verschiedene Datenpunkte an die jeweilige Schicht übergeben werden. Steht nur eine Zahl in der Tabelle wird der Schicht eine simple eindimensionale Liste an Zahlen übergeben. Da ein Bild jedoch mehrdimensional ist, in diesem Fall 120 Pixelwerte in x-Richtung, 120 Pixelwerte in y-Richtung und zu jedem Pixel 3 Farbwerte, finden sich als Ein- und Ausgabe des gesamten Netzes jeweils drei Werte. Dies ist auch der Grund, warum zum Beginn und Ende eine *Reshape*-Schicht also Umformungsschicht steht, die aus den dreidimensionalen Werten des Bildes eine einfache Liste an Werten macht beziehungsweise am Ende andersherum. Das Modell besteht hier aus zwei Teilen, dem *Encoder*, der die ersten drei Schichten ausmacht, und dem *Decoder* der aus den letzten drei Schichten besteht.

Das Modell wurde nun mit dem *SGD-Optimizer* und verschiedenen Lernraten für jeweils sechs Stunden trainiert. Da effektiv zwei Decoder trainiert werden sind die angegeben Daten über einen Zeitraum von ungefähr drei Stunden. Folgende Lernraten wurden ausprobiert.

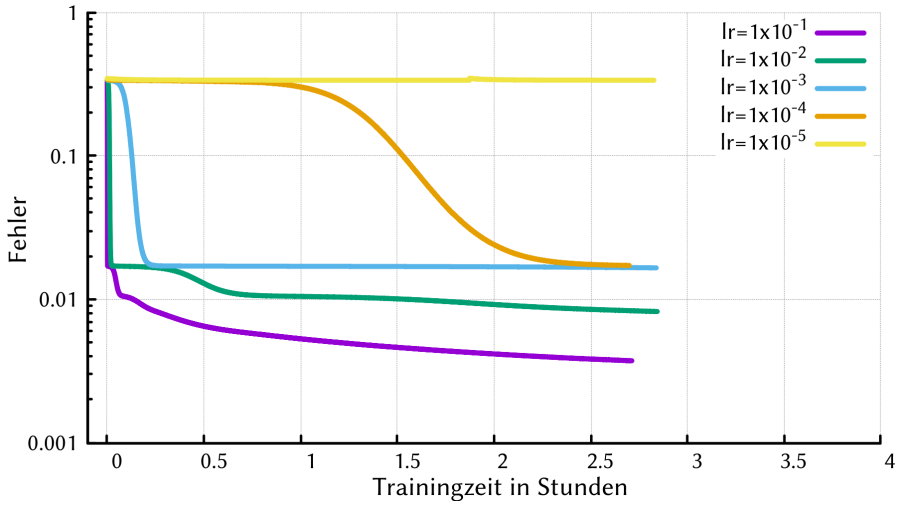
Die in Tabelle 2 angegebene Epoche ist quasi eine Lerneinheit, diese unterscheiden sich leicht, da die Modelle für eine bestimmte Zeit trainiert wurden. Es ist bereits zu beobachten, dass je größer die Lernrate ist desto geringer ist auch der Fehler. Dies wird besonders deutlicher, wenn man den Wert des Fehler über die Trainingszeit als Diagramm (Abbildung 10) darstellt.

Man kann außerdem erkennen, dass sich der Fehler bei der Lernrate 10^{-5} überhaupt

Lernrate	Epochen	min. Fehler	max. Fehler	Ø Fehler
$1 \cdot 10^{-1}$	6520	0.0037	0.3261	0.0055
$1 \cdot 10^{-2}$	6150	0.0082	0.3361	0.0120
$1 \cdot 10^{-3}$	5930	0.0166	0.3419	0.0288
$1 \cdot 10^{-4}$	5019	0.0172	0.3418	0.1589
$1 \cdot 10^{-5}$	6033	0.3356	0.3458	0.3367

Tabelle 2: *Fehlerwerte bei verschiedenen Lernraten, SGD (gerundete Werte)*

Abbildung 10: *Fehler für verschiedene Lernraten, SGD*

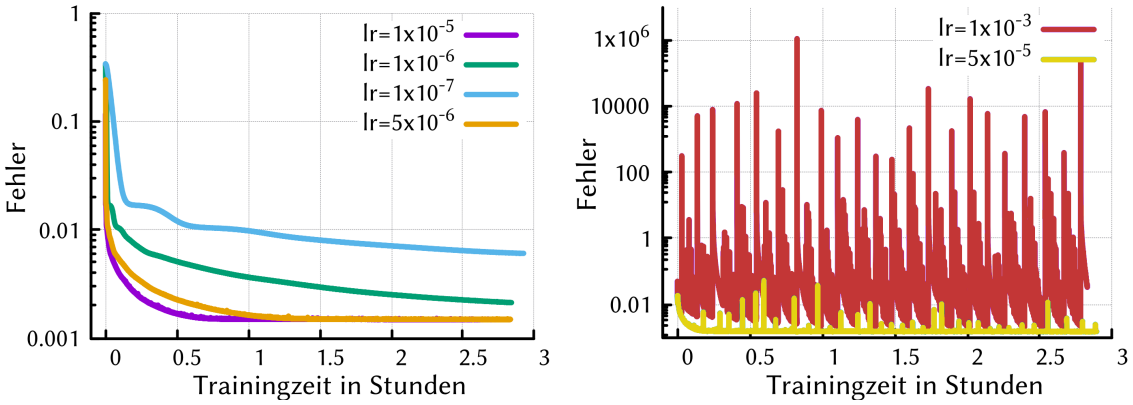


nicht mehr verbessert. Wenn man die Lernrate also senkt scheint es einen Wert zu geben, ab dem der Lernalgorithmus unbrauchbar wird und sich das Modell nicht verbessert. Erhöht man die Lernrate stetig, findet sich auch eine Grenze ab der das Lernen nicht mehr effektiv ist. Wird versucht das Modell mit einer Lernrate von 0,5 zu trainieren, steigt der Fehler äußerst schnell in absurde Höhen an. Dieser hier extreme Effekt wird als *Overfitting* [**NNPython:Overfitting**] bezeichnet und entsteht indem man mit zu großen Schritten über das Minimum der Fehler-Funktion hinausschießt. Dies wird noch deutlicher, wenn man sich einen Alternativen *Optimizer*, namens *Adam* [Goo21b; SS20c], anschaut. Dieser ist einer der am häufigsten verwendeten Optimizer. Er wurde auf die gleiche Modellstruktur aus Tabelle 1 mit verschiedenen Lernraten angewandt.

Lernrate	Epochen	min. Fehler	max. Fehler	Ø Fehler
$1 \cdot 10^{-3}$	7450	0.0019	1094034.7500	195.1310
$1 \cdot 10^{-4}$	7499	0.0015	0.1098	0.0017
$5 \cdot 10^{-5}$	7341	0.0015	0.0526	0.0016
$1 \cdot 10^{-5}$	6649	0.0015	0.1159	0.0017
$5 \cdot 10^{-6}$	6212	0.0015	0.2405	0.0019
$1 \cdot 10^{-6}$	4213	0.0021	0.3315	0.0044
$1 \cdot 10^{-7}$	3805	0.0061	0.3392	0.0138

Tabelle 3: *Fehlerwerte bei verschiedenen Lernraten, Adam (gerundete Werte)*

Abbildung 11: *Fehler für verschiedene Lernraten, Adam*



Auch hier wird deutlich, das Modell lernt schneller, wenn die Lernrate größer ist. Wird die Lernrate jedoch größer als ungefähr 10^{-5} fängt die Kurve (Abbildung 11) an unregelmäßig zu werden, bis sie anfängt extrem zu fluktuieren und damit nicht mehr effektiv am lernen ist. Wählt man die Lernrate jedoch zu klein, mit dem Gedanken *Overfitting* zu vermeiden, lernt das Modell deutlich langsamer. Die richtige Wahl der Lernrate ist also essenziell für das erfolgreiche Trainieren eines *KNN*.

3.4 Erstellen eines Deepfake Videos

Nun da das *KNN* trainiert wurde, bleib nur noch das eigentliche *Deepfake* Video zu erstellen. Das ist jetzt recht einfach, aus dem zu fälschenden Video werden die Gesichter extrahiert, vom *Encoder* eingelesen und von dem jeweils Anderen *Decoder* zu einem neuen Gesicht aufgebaut. Dieses Gesicht wird an der Gleichen Stelle wieder ins Video eingefügt und fertig ist das *Deepfake* Video.

3.5 Ergebnisse

Was jetzt recht simple klang, ist in der tatsächlichen Umsetzung deutlich schwerer. Die Schwierigkeit liegt dabei in der Festlegung der richtigen Struktur des *KNN*. Hierbei funktioniert die Umsetzung des *Autoencoder* sehr gut, das Problem liegt in dem Umwandeln der Bilder. Schlussendlich habe ich das beste Ergebnis mit einem *Convolutional Neural Network*, dessen Ergebnisse man in Abbildung 12 und 13 sieht, erhalten.

Die Ergebnisse sind verbesserungswürdig, und ohne gravierende strukturelle Änderungen im *KNN* scheinen es, dass auch keine besseren Ergebnisse möglich sind. Auch langes Trainieren ist da nicht die Lösung. Nach ein paar Stunden wird ein Minimum erreicht und das *KNN* verbessert sich nicht mehr. Das konnte man sehr gut in Kapitel 3.3.3 beobachten, betrachtet man das linke Diagramm in Abbildung 11 kann man erkennen, dass der Fehler-Wert nie unter 0,0015 fällt. Dies wird sich auch nicht ändern,

Abbildung 12: *Links: Eingabebild, Mitte: Mit dem Autoencoder erstellt, Rechts: Als Biden rekontruiert*

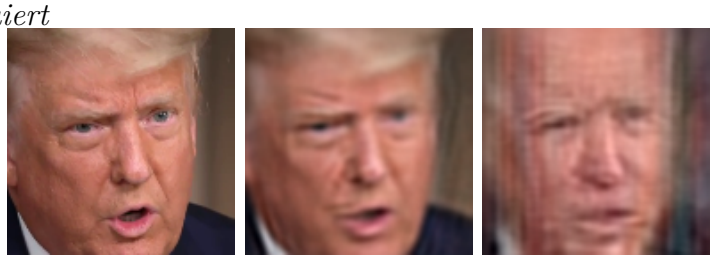


Abbildung 13: *Links: Eingabebild, Mitte: Mit dem Autoencoder erstellt, Rechts: Als Trump rekonstruiert*

wenn das *KNN* noch länger trainiert wird. Dies war zumindest bei den Netzen, die im Zuge dieser Arbeit konstruiert wurden der Fall.

Größere Netze sind auch nur begrenzt möglich. Die benötigte Menge an Arbeitsspeicher, sowohl vom Prozessor als auch von der Grafikkarte, überschreitet schnell die handelsüblichen wenigen Gigabyte. Außerdem dauert das Trainieren pro Epoche deutlich länger. Und wie man den Abbildungen 14 und 15 sieht, hat dies auch keinen maßgeblichen Vorteil. Und das obwohl das Netz, mit dem die unteren Bilder erstellt wurden, mit rund 12 Millionen mehr als doppelt so viele Parameter hat, wie das Netz von den Abbildungen 12 und 13.

Abbildung 14: *Links: Eingabebild, Mitte: Mit dem Autoencoder erstellt, Rechts: Als Biden rekonstruiert*

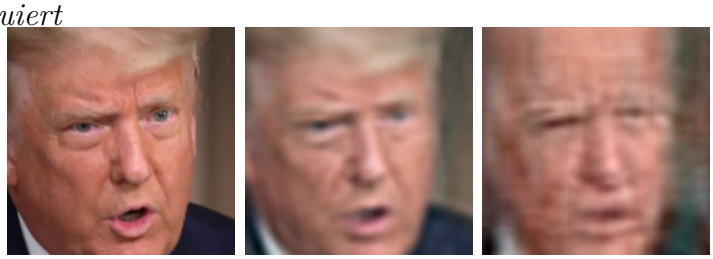
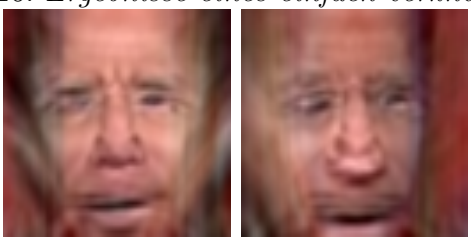


Abbildung 15: *Links: Eingabebild, Mitte: Mit dem Autoencoder erstellt, Rechts: Als Trump rekonstruiert*

Im Vergleich zu einem einfach verknüpften *KNN* kann man jedoch Verbesserungen erkennen. In Abbildung 16 sieht man Gesichter, die mit einem größeren einfach verknüpften *KNN* erstellt wurden. Es wurde hierbei versucht aus dem Gesicht von Biden, das von Chuck Norris zu machen.

Abbildung 16: *Ergebnisse eines einfach verknüpften KNN*



3.6 Fazit und Ausblick

Auch wenn mein *KNN* zur Erstellung von *Deepfakes* niemanden täuschen wird, gibt es deutlich fortgeschrittenere Werkzeuge mit denen sich täuschend echte gefälschte Videos erstellen lassen. Doch man sollte die Gefahr, die mit dieser Technologie einhergeht nicht unterschätzen. Potente Programme zur Erstellung von *Deepfakes* sind leicht zugänglich und durch soziale Medien können sich Falschinformation äußerst schnell verbreiten. In der Zukunft werden *Deepfakes* immer besser werden, bis zu dem Punkt wo wir sie als Menschen nicht mehr von echten Videos unterscheiden können. Wir müssen uns also überlegen wie man damit umgehen sollte. Es gibt zwar den Ansatz mit Maschinellem Lernen *Deepfakes* zu erkennen, allerdings sind wir noch weit davon entfernt, dies auf die großen Mengen an Video- und Bildmaterial, das wir konsumieren, anzuwenden. Es bleibt also die Frage, ob wir den Quellen, aus denen wir unsere Informationen beziehen, trauen können? Ich denke stellen uns diese Frage viel zu selten, obwohl die Antwort häufig *Nein* heißt. Nie von Falschinformation getäuscht zu werden, ist unmöglich, aber durch kritisches und hinterfragendes Denken kann der Großteil entlarvt werden. Also fangen wir doch an, uns häufiger zu fragen, kann das wirklich wahr sein?

Literaturverzeichnis

[Goo21a] Google. *tf.keras.losses.MeanSquaredError*. 2021. URL: **https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanSquaredError**.

[Goo21b] Google. *tf.keras.optimizers.Adam*. 2021. URL: **https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam**.

[Goo21c] Google. *tf.keras.optimizers.SGD*. 2021. URL: **https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD**.

[Ngu+21] Thanh Thi Nguyen u. a. *Deep Learning for Deepfakes Creation andDetection: A Survey, Seite 1*. 26. Apr. 2021. URL: **https://arxiv.org/pdf/1909.11573.pdf** (besucht am 18. 06. 2021).

[SS20a] Joachim Steinwendner und Roland Schwaiger. *Neuronale Netze programmieren mit Python*. 2. Aufl. Bonn: Reihnwerk Computing, 2020, S. 31–36.

[SS20b] Joachim Steinwendner und Roland Schwaiger. *Neuronale Netze programmieren mit Python*. 2. Aufl. Bonn: Reihnwerk Computing, 2020, S. 126–128.

[SS20c] Joachim Steinwendner und Roland Schwaiger. *Neuronale Netze programmieren mit Python*. 2. Aufl. Bonn: Reihnwerk Computing, 2020, S. 214–216.

[VJ01] Paul Viola und Michael Jones. *Rapid Object Detection using a Boosted Cascade of Simple Features*. 2001. URL: **https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf** (besucht am 18. 06. 2021).

Abbildungsverzeichnis

Quellcode-Dateien zur Implementierung

1 Neuronales Netz zum erstellen eines Deepfakes

Dieses Dokument ermöglicht Dir, ein Deepfake-Video zu erstellen, wären Du über die Funktion und den Aufbau des dazu verwendeten künstlichen Neuronalen Netzes lernst.

Das folgende Dokument setzt sich aus folgenden Schritten zusammen:

1. Einlesen und Aufbereiten von den Daten zum Trainieren
    - Extrahieren von Gesichter aus einem Video
    - Einlesen dieser Daten
2. Initialisieren des künstlichen Neuronalen Netzes
    - Definieren der Struktur des Encoder und Decoder
    - Kombinieren des Encoder und Decoder zu den Autoencodern
    - Kompilieren der Modelle mit einem Optimizer
3. Trainieren
    - Festlegen von Checkpoints für das Trainieren
    - Eigentliches Trainieren des Netzes
4. Fälschen eines neuen Videos

1.1 Einlesen und Aufbereiten von den Daten zum Trainieren

Um dem künstlichen Neuronalen Netz beizubringen Gesichter zu erkennen und zu erstellen, werden zahlreiche Bilder von Gesichtern benötigt. Im Fall von Deepfakes ist Bildmaterial von zwei Personen nötig. Einfachheitshalber wird als Input Videomaterial verwendet.

1.1.1 Extrahieren von Gesichter aus einem Video

In den Variablen *PFAD_A* und *PFAD_B* werden die Pfade zu den Videos gespeichert, die dazu verwendet werden Bilder von zwei Personen zu erxtrahieren. Es werden mit der Hilfe der Klasse `Gesichterextrahierer`, die in einer externen Datei definiert wurde, die Gesichter aus den Videos extrahiert. Die Bilder werden in einem Verzeichnis gespeichert.

[ ]:
```
import Gesichterextrahierer as GE

PFAD_A = './daten/videomaterial/Joe_Biden/nur_joe_biden_gemischt.mp4'
PFAD_B = './daten/videomaterial/Chuck_Norris/nur_Chuck_Norris_gemischt.mp4'
PFAD_KASKADE = './daten/cascades/haarcascade_frontalface_default.xml'

g = GE.Gesichterextrahierer(PFAD_KASKADE)
g.lade(PFAD_A)
g.extrahiereGesichter(
```

```
    max_anzahl_bilder=3000,
    ordner_ausgabe='./daten/lernen/Gesichter/A'
)
g.lade(PFAD_B)
g.extrahiereGesichter(
    max_anzahl_bilder=3000,
    ordner_ausgabe='./daten/lernen/Gesichter/B'
)

del PFAD_A, PFAD_B, PFAD_KASKADE, g, GE
```

1.1.2 Einlesen der Bilddateien

Dass die Bilder als Datei gespeichert wurden, erspart uns beim nächsten Mal den vorherigen Schritt. Die Dateien müssen nun allerdings wieder ins Programm geladen werden.

`erstelleDatensatz(pfad: str) -> list[list]`

Lädt alle Bilder in dem übergebenen Verzeichnis in zwei Datensätze und gibt diese als Liste zurück. Jeder Pixelwert wird durch 255 geteilt, um die Werte auf den Bereich zwischen 0 und 1 zu projektieren. Dies stellt sicher, dass die Werte des künstlischen Neuronale Netzes (KNN), wenn das Bild übergeben wird, nicht zu groß werden. Die Bilder werden in zwei Datensätze umgewandelt, zu 75% zum Trainieren des KNN und zu 25% zum Prüfen und Bewerten der Leistung des Netzes.

`teileListe(liste: list, verteilung: float) -> list[list]`

Teilt die übergebene Liste in zwei Listen und gibt diese zurück. Die erste zurückgegebene Liste hat ein Länge von n-% der übergebenen Liste, wobei n als Kommazahl zwischen 1 und 0 mit `verteilung` übergeben wird.

`verzerren(bild: list, staerke: int) -> 'Bild'`

Gibt eine verzerrte Version des übergebene Bild, als würde man das Bild von weiter rechts betrachten. Die `stärke` (eine ganze Z)

[ ]:
```
import numpy as np
import cv2
import os

def erstelleDatensatz(pfad: str, anzahl: int) -> list:
    bilder = []
    for wurzel, ordner, dateien in os.walk(pfad):
        dateien = [e for e in dateien if e.split(".")[-1].lower() in ['png',␣
 ↪'jpg', 'jpeg']]
        dateien = dateien[:int(anzahl/4)]
        for datei in dateien:
            bild = cv2.imread(os.path.join(wurzel, datei))
            bild = bild.astype('float32')
            bild /= 255.0
```

```
            for e in [bild, np.fliplr(bild), verzerren(bild, 10), np.
 ↪fliplr(verzerren(bild, 10))]:
                bilder.append(e)
                if len(bilder) >= anzahl: break

    np.random.shuffle(bilder)
    bilder_train, bilder_test = teileListe(bilder, 0.75)
    bilder_train, bilder_test = np.array(bilder_train), np.array(bilder_test)
    print('%d Bilder aus %s geladen.' % (len(bilder), pfad))
    return [bilder_train, bilder_test]

def teileListe(liste: list, verteilung: float) -> list:
    x = int(len(liste)*verteilung)
    return [liste[:x], liste[x:]]

def verzerren(bild: list, staerke: int) -> list:
    hoehe, breite = bild.shape[0:2]
    punkte_von = np.float32([[0, 0], [0, hoehe], [breite, 0], [breite, hoehe]])
    punkte_nach = np.float32([[0, staerke], [0, hoehe-staerke], [breite, 0],␣
 ↪[breite, hoehe]])
    matrix = cv2.getPerspectiveTransform(punkte_von, punkte_nach)
    bild_verzerrt = cv2.warpPerspective(bild, matrix, (breite, hoehe))
    bild_verzerrt = bild_verzerrt[staerke:hoehe-staerke, staerke:breite-staerke]
    return cv2.resize(bild_verzerrt, (breite, hoehe))

datensatz_gesichter_A_train, datensatz_gesichter_A_test =␣
 ↪erstelleDatensatz('daten/lernen/Gesichter/A', 5000)
NAME_AUTOENCODER_A = 'Biden'

datensatz_gesichter_B_train, datensatz_gesichter_B_test =␣
 ↪erstelleDatensatz('daten/lernen/Gesichter/B', 5000)
NAME_AUTOENCODER_B = 'Trump'
```

1.1.3 Um zu prüfen, ob die Bilder korrekt geladen wurden

[ ]:
```
from matplotlib.pyplot import imshow
%matplotlib inline

imshow(cv2.cvtColor(datensatz_gesichter_A_test[4], cv2.COLOR_BGR2RGB))

del imshow
```

1.2 Initialisieren des Neuronalen Netzes

Nun wird das künstlichen Neuronale Netz initialisiert. Dazu wird die Struktur des Netzes definiert und das Modell anschließend kompiliert.

1.2.1 Definieren der Struktur des Encoder und Decoder

Der `NAME` wird als Idenetifikation und zum Abspeichern verwendet.

`logSummary(string: str)`

Die Funktion, die später dazu verwendet wird die Zusammenfassung des Netzes abzuspeichern.

`gibEncoder()`

Definiere hier deinen Encoder. Das Modell wird von der Funktion zurückgegeben.

`gibDecoder()`

Definiere hier deinen Decoder. Das Modell wird von der Funktion zurückgegeben.

[ ]:
```
import tensorflow as tf

IMG_SHAPE = (128, 128, 3)
NAME = "CNN_medium"

def logSummary(string: str):
    with open(f"./daten/modelle/{NAME}/modell.info", "a") as datei:
        datei.write(string + "\n")

def gibEncoder():
    encoder = tf.keras.Sequential(name='encoder')
    encoder.add(tf.keras.layers.Conv2D(32, kernel_size=3, strides=1,␣
 ↪padding='same', input_shape=( IMG_SHAPE ) ))
    encoder.add(tf.keras.layers.MaxPooling2D((2,2)))
    encoder.add(tf.keras.layers.Conv2D(32, kernel_size=3, strides=1,␣
 ↪padding='same'))
    encoder.add(tf.keras.layers.MaxPooling2D((2,2)))
    encoder.add(tf.keras.layers.Conv2D(64, kernel_size=3, strides=1,␣
 ↪padding='same'))
    encoder.add(tf.keras.layers.MaxPooling2D((2,2)))
    encoder.add(tf.keras.layers.Conv2D(256, kernel_size=3, strides=1,␣
 ↪padding='same'))
    encoder.add(tf.keras.layers.MaxPooling2D((2,2)))
    encoder.add(tf.keras.layers.Flatten())
    encoder.add(tf.keras.layers.Dense( 256 ))
    encoder.add(tf.keras.layers.Dense( (8*8*256)))
    encoder.add(tf.keras.layers.Reshape( (8, 8, 256) ))
    encoder.add(tf.keras.layers.Conv2DTranspose(256, kernel_size=5, strides=2,␣
 ↪padding='same'))
```

```
    encoder.summary(print_fn=logSummary)
    print(encoder.summary())
    return encoder

def gibDecoder():
    decoder = tf.keras.Sequential(name='decoder')
    decoder.add(tf.keras.layers.Conv2DTranspose(64, kernel_size=3, strides=2,␣
 ↪padding='same', input_shape=(16,16,256)))
    decoder.add(tf.keras.layers.Conv2DTranspose(32, kernel_size=2, strides=2,␣
 ↪padding='same'))
    decoder.add(tf.keras.layers.Conv2DTranspose(32, kernel_size=3, strides=1,␣
 ↪padding='same'))
    decoder.add(tf.keras.layers.Conv2DTranspose(3, kernel_size=3, strides=2,␣
 ↪padding='same'))

    decoder.summary(print_fn=logSummary)
    print(decoder.summary())
    return decoder
```

1.2.2 Kombinieren des Encoder und Decoder zu den Autoencodern

[ ]:
```
def gibAutoencoder(name):
    x = tf.keras.layers.Input( shape=IMG_SHAPE, name='input_layer' )
    encoder, decoder = gibEncoder(), gibDecoder()
    autoencoder = tf.keras.Model(x, decoder(encoder(x)), name=name)

    print(autoencoder.summary())
    return autoencoder
```

1.2.3 Kompilieren der Modelle mit einem Optimizer

`gibKompiliertenAutoencoder(name)`

Kombiniert den Encoder und Decoder zu einem Autoencoder und gibt diesen als kompiliertes Modell zurück.

[ ]:
```
OPTIMIZER_FUNKTION = tf.keras.optimizers.Adam(learning_rate=1e-5)
LOSS_FUNKTION = tf.keras.losses.MeanSquaredError()

def gibKompiliertenAutoencoder(name):
    autoencoder = gibAutoencoder(name)
    autoencoder.compile(optimizer=OPTIMIZER_FUNKTION, loss=LOSS_FUNKTION)

    return autoencoder
```

Falls bereits ein Modell mit dem zuvor definierten Namen exestiert, wird dieses geladen. Ist das nicht der Fall, wird ein neues Modell erstellt. Anschließend wird zur Übersicht der Fehler-Wert der Modelle ermittelt.

```
[ ]: try:
        autoencoder_A = tf.keras.models.load_model(f"./daten/modelle/{NAME}/
     ↪{NAME_AUTOENCODER_A}/")
        autoencoder_B = tf.keras.models.load_model(f"./daten/modelle/{NAME}/
     ↪{NAME_AUTOENCODER_B}/")
        print("Modelle von der Festplatte geladen.\n")
    except Exception as e:
        print(e)
        try:
            os.mkdir(f"./daten/modelle/{NAME}/")
            os.mkdir(f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_A}/")
            os.mkdir(f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_B}/")
            os.mkdir(f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_A}/Bilder/")
            os.mkdir(f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_B}/Bilder/")
        except FileExistsError:
            pass
        autoencoder_A = gibKompiliertenAutoencoder(name="autoencoder_A")
        autoencoder_B = gibKompiliertenAutoencoder(name="autoencoder_B")

    loss = autoencoder_A.evaluate(datensatz_gesichter_A_test[:32],␣
     ↪datensatz_gesichter_A_test[:32])
    print(f"Aktueller Loss von A ({NAME_AUTOENCODER_A}): {loss}")

    loss = autoencoder_B.evaluate(datensatz_gesichter_B_test[:32],␣
     ↪datensatz_gesichter_B_test[:32])
    print(f"Aktueller Loss von B ({NAME_AUTOENCODER_B}): {loss}")
```

1.3 Trainieren

1.3.1 Festlegen von Checkpoints für das Trainieren

Hier werden Callbacks definiert, die später Trainingsfunktion übergeben werden. Die Callbacks enthalten Anweisungen, die während dem Training ausgeführt werden. Hierzu zählt zum Beispiel das regelmäßige Speichern des Fortschritts.

```
[ ]: from tensorflow.keras.callbacks import ModelCheckpoint
    import LoggingCallback as lc


    autoencoder_A_logging_callback = lc.LoggingCallback(
        pfad_modell=f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_A}/",
        bild_A=datensatz_gesichter_A_test[1],
        bild_B=datensatz_gesichter_B_test[1]
    )
```

```
autoencoder_B_logging_callback = lc.LoggingCallback(
    pfad_modell=f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_B}/",
    bild_A=datensatz_gesichter_A_test[1],
    bild_B=datensatz_gesichter_B_test[1]
)

autoencoder_A_checkpoint_callback = ModelCheckpoint(
    f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_A}/",
    monitor='val_loss',
    save_best_only=True
)

autoencoder_B_checkpoint_callback = ModelCheckpoint(
    f"./daten/modelle/{NAME}/{NAME_AUTOENCODER_B}/",
    monitor='val_loss',
    save_best_only=True
)
```

1.3.2 Eigentliches Trainieren des Netzes

Nun wird das Modell trainiert. In der dritten Zeile kann die gewünschte Dauer des Trainings definiert werden. Die beiden Autoencoder werden abwechselnd für jeweils ein Epoche trainiert, dann wird Encoder zwischen den Modellen getauscht. Dies soll sicherstellen, dass die beiden Autoencoder die geichen Muster in den Bilder erkennen und somit schlussendlich Bilder fälschen. Die batch_size bestimmt wie viel Bilder gleichzeitig trainiert werden. Ist dieser Wert zu hoch kommt es schnell zu einem OOM (Out of Memory, also dem Volllaufen des Arbeitsspeichers) Error.

[ ]:
```
import time, gc

ZEITPUNKT_ENDE = time.time() + int(8*60*60)

while time.time() < ZEITPUNKT_ENDE:
    print("!- Noch für ~{:.1f}h beschäftigt.".format(( ZEITPUNKT_ENDE-time.
 ↪time() )/3600) )

    autoencoder_A.fit(
                      datensatz_gesichter_A_train,
                      datensatz_gesichter_A_train,
                      epochs=1,
                      batch_size=16,
                      shuffle=True,
                      validation_data=(datensatz_gesichter_A_test,␣
 ↪datensatz_gesichter_A_test),
                      callbacks=[autoencoder_A_checkpoint_callback,␣
 ↪autoencoder_A_logging_callback]
                     )
```

```
    autoencoder_B.layers[1] = autoencoder_A.get_layer('encoder')
    gc.collect()

    autoencoder_B.fit(
                      datensatz_gesichter_B_train,
                      datensatz_gesichter_B_train,
                      epochs=1,
                      batch_size=16,
                      shuffle=True,
                      validation_data=(datensatz_gesichter_B_test,␣
↪datensatz_gesichter_B_test),
                      callbacks=[autoencoder_B_checkpoint_callback,␣
↪autoencoder_B_logging_callback]
                     )
    autoencoder_A.layers[1] = autoencoder_B.get_layer('encoder')
    gc.collect()
```

1.3.3 Fälschen eines neuen Videos

Nun geht es daran das Gefälschte Video zu erstellen.

Zunächst hat man hier die Möglichkeit eine Vorschau der Leistungsfähigkeit des Modells zu erhalten.

Wechsle zwischen `autoencoder_A` und `autoencoder_B`, um das Modell zur jeweils anderen Person zu ändern.

Wechsle zwischen `datensatz_gesichter_A_test` und `datensatz_gesichter_B_test`, um die Person, deren Gesichter dem Modell übergeben werden zu ändern.

Ändere den Index nach `datensatz_gesichter_*_test`, um ein anderes Gesicht der Person auszuwählen.

[ ]:
```
from matplotlib.pyplot import imshow
%matplotlib inline

img = cv2.cvtColor(autoencoder_A.predict(datensatz_gesichter_A_test[1].
 ↪reshape(1, 128, 128, 3))[0], cv2.COLOR_BGR2RGB)
img = cv2.normalize(img, None, 0, 1, cv2.NORM_MINMAX)
imshow(img)
```

Und abschließend kann hier ein Video gefälscht werden. Ändere auch hier `autoencoder_*`, um das Modell zu wechseln.

[ ]:
```
import Gesichterextrahierer as GE

PFAD_KASKADE = './daten/cascades/haarcascade_frontalface_default.xml'
```

```
def fake(bild):
    bild = bild.astype('float32')
    bild /= 255.0
    erg = autoencoder_A.predict(bild.reshape(1, 128, 128, 3))[0]
    erg = cv2.normalize(erg,  None, 0, 255, cv2.NORM_MINMAX)
    return erg

g = GE.Gesichterextrahierer(PFAD_KASKADE)
g.lade('./daten/Biden.mp4')
g.fuerGesichterMache(fake, 10000, True)
```

Listing 1: Gesichterextrahierer.py

```
"""
Ein Skript das sowohl als eigenständiges Programm genutzt werden kann, als auch
als Modul importiert werden kann. Ermöglicht das Extrahieren und bearbeiten von
Gesichtern in Videos.

Nutzung als Modul:
1. Mit dem Pfad zur Kaskade initalisieren.
2. Optional die Bildergröße der Ausgabe mit 'setzeBildergroesse' setzen.
3. Video laden mit 'lade'.
4. Video mit den anderen Methoden
    - fürGesichterMache
    - extrahiereGesichter
    - extrahiereUndValidiereGesichter
        bearbeiten.

Nutzung als Skript:
-> Mit Python3 und der '-h' Option starten um die Hilfe angezeigt zu bekommen.
"""
import sys, cv2, time, os
import numpy as np
import face_recognition

class Gesichterextrahierer:
    def __init__(self, pfad_cascade: str):
        self.CASCADE = cv2.CascadeClassifier(pfad_cascade)
        self.bildgroesse_output = 128
        self.counter = 0

    def setzeBildgroesse(self, bildgroesse_output):
        self.bildgroesse_output = bildgroesse_output

    def lade(self, pfad_video):
        self.pfad_video = pfad_video
        self.video = cv2.VideoCapture(pfad_video)
        self.fps = self.video.get(cv2.CAP_PROP_FPS)
        self.frame_height = int(self.video.get(cv2.CAP_PROP_FRAME_HEIGHT))
        self.frame_width = int(self.video.get(cv2.CAP_PROP_FRAME_WIDTH))

    def play(self):
        """
        Spielt das geladene Video in einem neuen Fenster ab. Escape beendet die
        Wiedergabe.
        """
        while True:
            ist_bild, bild = self.video.read()  # Nächsten Frame einlesen
            if cv2.waitKey(1) == 27 or not ist_bild:
```

```
                break
            cv2.imshow('Deepfakeskript - Video', bild)
            time.sleep(1/self.fps-1)
        self.video.set(cv2.CAP_PROP_POS_FRAMES, 1)  # Video zurücksetzen, damit
        # es wieder eingelesen werden kann
        cv2.destroyAllWindows()

    def fuerGesichterMache(self, funktion, max_anzahl_gesichter, speichern=True):
        """
        Fürt die übergebene Funktion für alle Gesichter in dem Video aus. Es
        werden maximal ein Gesicht pro Bild erkannt.

        :funktion: Eine Funktion, der beim Ausführen ein Parameter, der
                Bildausschnitt des Gesichts, als Liste übergeben wird.
                Gibt die Funktion False zurück wird das gerade bearbeitete Bild
                übersprungen.
                Gibt die Funktion True wird der Counter erhöht und das nächste
                Bild geladen, sofern nicht das Maximum erreicht wurde.
                Gibt die Funktion ein Bild als Liste zurück, wird dies anstelle
                des übergebenen Gesichts in das Bild des Videos eingesetzt. Die
                Ein- und Ausgabeliste muss die gleiche Form haben.
                Es kann von der Funktion aus `self.counter` zugegriffen werden,
                um die Anzahl an bearbeiteten Bildern zu erhalten.
        :max_anzahl_gesichter: Definiert die Anzahl an Gesichtern, die maximal
                bearbeitet werden. Die Funktion endet vorzeitig wenn die
                Videodatei zu Ende ist.
        :speichern: Bei True wird eine Videodatei abgespeichert, die die
                mögliche Änderungen beinhaltet.
                Bei False ist dies nicht der Fall.
        :return: Gibt die Anzahl an bearbeiteten Bildern zurück.
        """
        if speichern:  # Vorbereitung für das Abspeichern des neuen Videos
            pfad_Video_ohne_Endung = ".".join(self.pfad_video.split(".")[:-1])
            video_writer = cv2.VideoWriter(
                                    f'{pfad_Video_ohne_Endung}_geändert.mp4',
                                       cv2.VideoWriter_fourcc(*'mp4v'),
                                    self.fps,
                                    (self.frame_width, self.frame_height)
                                )

        ist_bild, bild = self.video.read() # Ersten Frame einlesen

        while ist_bild:  # Solange das Video nicht zu Ende ist
            bild_grau = cv2.cvtColor(bild, cv2.COLOR_BGR2GRAY)
            gesichter = self.CASCADE.detectMultiScale(
                                    bild_grau,
                                    scaleFactor=1.3,
                                    minNeighbors=5,
                                    minSize=(self.bildgroesse_output,)*2
```

```
                            )  # Gesichter erkennen
            if len(gesichter):
                x, y, breite, hoehe = gesichter[0]  # Nur das erste Gesicht
                gesicht = bild[y:y+hoehe, x:x+breite]  # Kopieren des Gesichts
                gesicht_skaliert = cv2.resize(
                                        gesicht,
                                        (self.bildgroesse_output,)*2
                                    )
                # Die übergebene Funktion ausführen
                rueckgabe_f = funktion( gesicht_skaliert )
                if type(rueckgabe_f) == bool and rueckgabe_f == False:
                        break  # Mit dem nächsten Frame fortfahren
                if speichern:
                    if type(rueckgabe_f) != bool:
                        rueckgabe_f = cv2.resize(
                                            rueckgabe_f,
                                            (breite, hoehe)
                                        )
                        # Rückgabe der Funktion in das Bild einsetzen
                        bild[y:y+hoehe, x:x+breite] = rueckgabe_f
                    video_writer.write(bild)  # Frame an das neue Video anhängen
                print("\r"+"Es wurden erfolgreich %d Bilder bearbeitet."
                                                        % self.counter, end='')
                self.counter += 1
            elif speichern:
                video_writer.write(bild)  # Frame an das neue Video anhängen
                                          # auch wenn kein Gesicht gefunden
                                          # wurde

            ist_bild, bild = self.video.read()  # Nächsten Frame einlesen
            if self.counter >= max_anzahl_gesichter:
                break

        if speichern: video_writer.release()
        self.video.set(cv2.CAP_PROP_POS_FRAMES, 1)  # Video zurücksetzen, damit
                                                    # es wieder eingelesen
                                                    # werden kann
        x = self.counter
        self.counter = 0
        return x


    def extrahiereGesichter(self, max_anzahl_gesichter: int, ordner_ausgabe: str):
        """
        Findet alle Gesichter aus dem Video und speichert sie ab.

        :max_anzahl_gesichter: Definiert die Anzahl an Gesichtern, die maximal
                bearbeitet werden. Die Funktion endet vorzeitig wenn die
                Videodatei zu Ende ist.
```

```
145         :ordner_ausgabe: Den Pfad zum Ordner in dem die Bilder abgespeichert
146                 werden, dieser muss vorhanden sein.
147         """
148         def funktion(bild):
149             cv2.imwrite(os.path.join(
150                     ordner_ausgabe, f'Gesicht_{self.counter}.png'), bild)
151             return True
152 
153         print("Exportiere Bilder nach %s." % ordner_ausgabe)
154         anzahl_extrahierter_bilder = self.fuerGesichterMache(
155                                                 funktion,
156                                                 max_anzahl_gesichter,
157                                                 speichern=False
158                                             )
159         print("\r"+"Es wurden erfolgreich %d Bilder nach %s exportiert."
160                                 % (anzahl_extrahierter_bilder, ordner_ausgabe))
161 
162 
163     def extrahiereUndValidiereGesichter(self, referenzbild: list,
164                     max_anzahl_gesichter: int, ordner_ausgabe: str, toleranz=0.6):
165         """
166         Findet alle Gesichter aus dem Video und speichert sie ab, falls sie dem
167         Referenzgesicht ausreichend ähneln.
168 
169         :referenzbild: Ein Bild in Form einer Liste das zum Vergleich bei der
170                 Validierung verwendet wird.
171         :max_anzahl_gesichter: Definiert die Anzahl an Gesichtern, die maximal
172                 bearbeitet werden. Die Funktion endet vorzeitig wenn die
173                 Videodatei zu Ende ist.
174         :ordner_ausgabe: Den Pfad zum Ordner in dem die Bilder abgespeichert
175                 werden, dieser muss vorhanden sein.
176         :toleranz: Die euklidische Distanz, die maximal zwischen den Vekoren,
177                 die die zu vergleichenden Gesichter repräsentieren,
178                 liegen darf, damit diese als von der gleichen Person gelten.
179                 (siehe face_recongnition.compare_faces)
180         """
181         encoding_referenz = face_recognition.face_encodings(referenzbild)
182         def funktion(bild):
183             try:
184                 # Gesicht in ein Vektorrepräsentation umwandeln
185                 encoding_bild = face_recognition.face_encodings(bild)[0]
186                 if face_recognition.compare_faces(encoding_referenz,
187                                         encoding_bild, tolerance=toleranz)[0]:
188                     cv2.imwrite(os.path.join( ordner_ausgabe,
189                                             f'Gesicht_{self.counter}.png'), bild)
190                     return True
191             except Exception: pass
192             return False
193 
```

```
194         print("Validiere und exportiere Bilder nach %s." % ordner_ausgabe)
195         anzahl_extrahierter_bilder = self.fuerGesichterMache(
196                                                 funktion,
197                                                 max_anzahl_gesichter,
198                                                 speichern=False
199                                     )
200         print("\r"+"Es wurden erfolgreich %d Bilder nach %s exportiert."
201                                 % (anzahl_extrahierter_bilder, ordner_ausgabe))
202
203
204 def main(argv):  # Wird ausgeführt, wenn das Skript direkt ausgeführt wird
205     bildgroesse_ausgabe = 128
206     max_anzahl_bilder = 50000
207     pfad_cascade = "./daten/cascades/haarcascade_frontalface_default.xml"
208     ordner_ausgabe = "./daten/Gesichter"
209     pfad_validierungbild = ""
210     toleranz = 0.6
211
212     for index, argument in enumerate(argv):
213         if argument[0] == '-':
214             if 'g' == argument[1]:
215                 bildgroesse_ausgabe = int(argv[index+1])
216             elif 'a' == argument[1]:
217                 max_anzahl_bilder = int(argv[index+1])
218             elif 'c' == argument[1]:
219                 pfad_cascade = argv[index+1]
220             elif 'o' == argument[1]:
221                 ordner_ausgabe = argv[index+1]
222             elif 'v' == argument[1]:
223                 pfad_validierungbild = argv[index+1]
224             elif 't' == argument[1]:
225                 toleranz = float(argv[index+1])
226             elif 'h' == argument[1]:
227                 print("Nutzung: %s [Optionen] [Videodatei]\n" % argv[0])
228                 print("""Optionen:
229     -g : Größe der Ausgabe Bilder in Pixel
230     -a : Maximale Anzahl der zu extrahierenden Bildern
231     -c : Pfad für die Haarcascade
232     -o : Zielordner für die Ausgabe
233     -h : Drucken dieser Hilfenachricht
234     -v : Pfad zu einem Validierungsbild
235     -t : Toleranz für die Gesichtsvalidierung\n""")
236                 return
237
238     g = Gesichterextrahierer(pfad_cascade)
239     g.setzeBildgroesse(bildgroesse_ausgabe)
240     g.lade(argv[-1])
241     if pfad_validierungbild:
242         validierungsbild = cv2.imread(pfad_validierungbild)
```

```
        g.extrahiereUndValidiereGesichter(
                validierungsbild,
                max_anzahl_bilder,
                ordner_ausgabe,
                toleranz=toleranz
        )
    else:
        g.extrahiereGesichter(max_anzahl_bilder, ordner_ausgabe)


if __name__ == "__main__":
    main(sys.argv)
```

Listing 2: LoggingCallback.py

```
"""
Die Klasse LoggingCallback, erbt von tensorflow.keras.callbacks.Callback

Wird als Instanz beim Trianieren mit der fit-Methode als callback übergeben. Die
Methoden, welche mit "on_..." beginnen werden zu dem entsprechenden Zeitpunkt
wärend des Trainingsprozesses aufgerufen und dokumentieren den Trainings-
vortschritt in der Datei 'train.log' im Verzeichnis des Modells. Bei jedem
Trainingsbeginn wird ein Gesicht von jeder der beiden Personen durch das Modell
geschickt und im Verzeichnis 'Bilder' abgespeichert.
"""

import tensorflow.keras.callbacks
import time, os, cv2
import numpy as np

class LoggingCallback(tensorflow.keras.callbacks.Callback):
    def __init__(self, pfad_modell: str, bild_A: list, bild_B: list):
        self.pfad_modell = pfad_modell
        self.bild_A = bild_A.reshape(1, 128, 128, 3)
        self.bild_B = bild_B.reshape(1, 128, 128, 3)
        try: os.mkdir(os.path.join(self.pfad_modell, 'Bilder/'))
        except FileExistsError: pass

    def log(self, text: str):
        with open(os.path.join(self.pfad_modell, 'train.log'), "a") as datei:
            datei.writelines("{};{}\n".format(time.time(), text) )

    def speichereBild(self, bild: list, pfad: str):
        bild = cv2.normalize(bild,
                             None,
```

```
                        alpha = 0,
                        beta = 255,
                        norm_type = cv2.NORM_MINMAX,
                        dtype = cv2.CV_32
                )
        bild = bild.astype(np.uint8)
        cv2.imwrite(pfad, bild)

    def on_train_begin(self, logs=None):
        self.log("Starte Training;")
        predicted_A = self.model.predict(self.bild_A)[0]
        predicted_B = self.model.predict(self.bild_B)[0]
        self.speichereBild(
                predicted_A,
                os.path.join(self.pfad_modell,
                "Bilder/{}_Bild_A.png".format(time.time()) )
        )
        self.speichereBild(
                predicted_B,
                os.path.join(self.pfad_modell,
                "Bilder/{}_Bild_B.png".format(time.time()) )
        )

    def on_epoch_begin(self, epoch, logs=None):
        logString = "on_epoch_begin;epoch;{};".format(epoch)
        for key, val in logs.items():
            logString += "{};{};".format(key, val)
        self.log(logString)

    def on_epoch_end(self, epoch, logs=None):
        logString = "on_epoch_end;epoch;{};".format(epoch)
        for key, val in logs.items():
            logString += "{};{};".format(key, val)
        self.log(logString)

    def on_train_batch_begin(self, batch, logs=None):
        logString = "on_train_batch_begin;batch;{};".format(batch)
        for key, val in logs.items():
            logString += "{};{};".format(key, val)
        self.log(logString)

    def on_train_batch_end(self, batch, logs=None):
        logString = "on_train_batch_end;batch;{};".format(batch)
        for key, val in logs.items():
            logString += "{};{};".format(key, val)
        self.log(logString)
```

Erklärung zur Seminararbeit

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

…….., den ……..
Ort Datum

……
Unterschrift des Verfassers