Generalizable Deepfake Detection

Bernhard Birnbaum

31. März 2025

1 Aufgabe & Stand der Technik

Im Rahmen des Praktikums „Implementierung in Forensik und Mediensicherheit“ soll ein Modell zur Klassifikation von DeepFake-Bildsequenzen konzeptioniert, implementiert und evaluiert werden. Mit geeigneten Metriken soll überprüft werden, ob über die Detektion hinaus eine Unterscheidung von verschiedenen DeepFake-Techniken (explizit *face-swap* (`FS`) und *face-reenactment* (`FR`)) etabliert werden kann.

DeepLearning-Architekturen für sequentielle Daten:
Um sequentielle Daten (z.B. Frames eines Videos) zu klassifizieren, werden rekurrente Architekturen genutzt (*recurrent neural networks*, RNNs). Eine weiterentwickelte Form von RNNs stellen die *long short-term memory networks* (LSTMs) dar, welche durch ihre speziellen Speicherzellen mit Gate-Mechanismen langfristige Abhängigkeiten effizienter lernen können und dabei dazu beitragen, das *vanishing gradients problem* zu reduzieren. Sogenannte *bidirectional LSTMs* ermöglichen darüber hinaus die Verarbeitung von Bildsequenzen in Vor- und Rückrichtung, was eine verbesserte Erfassung von zeitlichen Kontextinformationen ermöglicht.

Merkmalsextraktor als Baseline-Modell:
Das konzeptionierte Modell basiert auf einem Merkmalsextraktor, der einem Fine-Tuning unterzogen wird. Die während dieser Untersuchung erzeugten Modelle nutzen *convolutional neural networks* (CNNs) als Baseline, um die Komplexität der Eingabedaten zu reduzieren: entweder ResNet50 oder EfficientNetB0. Beide Architekturen sind besonders gut für die Klassifikation von Bildern geeignet und erzielen dabei State-of-the-Art-Ergebnisse, was sie zu gängigen Merkmalsextraktoren macht. ResNet50 [1] nutzt sogenannte *residual connections* bzw. *skip connections*, wodurch die Eingabedaten eine oder mehrere Schichten im Netzwerk überspringen können (wirkt ebenfalls *vanishing gradients problem* entgegen). EfficientNetB0 [2] hingegen versucht, die Skalierung von Netzen in Tiefe, Breite und Auflösung zu optimieren (*compound model scaling*), was zu einer deutlichen Verringerung der Anzahl von Parametern führt und dadurch eine bessere Performance bei gleicher Eingabegröße `(224, 224, 3)` ermöglicht.

Geeigneter Datensatz:
Um eine ausreichende Generalisierung zu erreichen sowie repräsentative Auswertungen durchführen zu können, wird ein breit gestreuter Trainings- bzw. Test-Datensatz benötigt. Der DF40-Datensatz [3] umfasst DeepFake-Erzeugnisse von 40 verschiedenen Tools inklusive der Originaldaten. Im Rahmen dieser Arbeit soll in einem ersten Proof-of-Concept zunächst zwischen den Klassen *original* (`OR`), *face-swap* (`FS`, 10 Tools) und *face-reenactment* (`FR`, 13 Tools) unterschieden werden.

2 Konzept

2.1 DF40-Datensatz

Da der DF40-Datensatz bereits als vorverarbeitete Version angeboten wird, müssen keine größeren Schritte zur Vorverarbeitung der Daten (wie z.B. *face detection* oder *cropping*) durchgeführt werden. Dementsprechend haben alle Frames dieselben Abmessungen (`256x256x3`), allerdings variiert die Länge der einzelnen Sequenzen zwischen 8 und 32.

Sequenzlängen-Filterung:
Aus diesem Grund muss bereits im Vorfeld festgelegt werden, mit welcher fixen Sequenzlänge das Modell arbeiten soll. Deshalb werden alle Sequenzen entweder zugeschnitten (sollten sie zu lang sein) oder aus dem Datenset entfernt (sollten sie zu kurz sein). Alternativ könnten zu kurze Sequenzen auch mit Padding aufgefüllt werden; da der Datensatz allerdings groß genug ist und nur

wenige Sequenzen betroffen sind, wurde darauf verzichtet.

Gewichtung der Klassen:
Da die Anzahl von Instanzen pro Klasse im Datensatz nicht gut balanciert ist, werden Klassengewichte (*class weights*) verwendet, um ein ausgeglichenes Training zu ermöglichen. Desweiteren hängen die tatsächlichen Elementzahlen von der verwendeten Sequenzlänge ab, wie in Tabelle 1 dargestellt. Der Split in Trainings- und Testdaten ist bereits vom Datensatz vorgegeben und wird entsprechend übernommen.

Tabelle 1: Anzahl Instanzen pro Klasse im Trainings- und Testdatensatz mit Klassengewichten

Sequenzlänge	**Trainingsdaten**			**Klassengewichte (≈)**			**Testdaten**		
	`OR`	`FS`	`FR`	`OR`	`FS`	`FR`	`OR`	`FS`	`FR`
8 Frames	999	6510	8467	5.3307	0.8180	0.6290	999	7075	9423
12 Frames	999	6509	8460	5.3280	0.8177	0.6292	999	7070	9405
16 Frames	999	6495	8445	5.3183	0.8180	0.6291	999	7058	9369

Einlesen des Datensatzes:
In Listing 1 im Anhang ist die Ordnerstruktur des Datensatzes im `io/`-Verzeichnis dargestellt.

1. **Durchsuchen der Ordnerstruktur**: Zunächst wird die Ordnerstruktur für jede Klasse nach Elementen durchsucht. Dabei werden die einzelnen `*.png`-Dateien einer Sequenz anhand ihres Dateinamens sortiert, um die Reihenfolge sicherzustellen. Beim Laden der Elemente wird außerdem die zuvor beschriebene Sequenzlängen-Filterung angewandt.
2. **Mischen**: Alle erfassten Items werden anschließend initial auf Pfad-Ebene sortiert, damit eine ausgeglichene Reihenfolge zur Bildung von Batches vorliegt. Außerdem wird der Trainingsdatensatz vor jeder Epoche erneut durchmischt.
3. **Preprocessing-Funktion**: Die einzelnen Bilder der Sequenzen (`256x256x3`) werden decodiert und in Tensoren der Form `(224, 224, 3)` umgewandelt (*resizing*). Bei ResNet50 muss zudem eine weitere Preprocessing-Funktion[1] verwendet werden, damit das Eingabeformat korrekt ist. Danach werden die Frames entlang der zeitlichen Achse gestapelt (*tensor stacking*). Auf die Labels wird zudem eine One-Hot-Kodierung angewandt.

2.2 Aufbau des Modells

Im folgenden wird der grundsätzliche Aufbau des entwickelten Modells zur Klassifikation beschrieben sowie die einzelnen Schichten charakterisiert.

1. **Input-Layer**: In der Eingabeschicht wird die Form des Eingabetensors definiert. Diese ergibt sich zum einen aus den Abmessungen eines einzelnen Frames `(224, 224, 3)`, zum anderen aus der Länge der Sequenz, mit der das Modell arbeiten soll.
   Beispiel für valide Form einer Eingabe (für Sequenzlänge 12): `(12, 224, 224, 3)`
2. **TimeDistributed-Wrapper mit Merkmalsextraktion**: Diese Schicht implementiert die Merkmalsextraktion des Baseline-Modells der zu klassifizierenden Bildsequenz, wahlweise durch ResNet50 oder EfficientNetB0. Dabei ist das Ziel, die Eingangsgröße eines jeden Einzelbildes durch Reduktion mit Faltungsschichten und Pooling-Operationen schrittweise zu verkleinern. Für jeden Frame `(224, 224, 3)` entsteht dadurch eine Feature-Map der Form `(7, 7, 2048)` bei ResNet50 bzw. `(7, 7, 1280)` bei EfficientNetB0. Die Merkmale aus der letzten Faltung des Netzwerks werden abschließend mit einem *2d global average pooling* zusammengefasst, wodurch jeder Frame der Sequenz auf einen Tensor der Form `(2048)` für ResNet50 bzw. `(1280)` für EfficientNetB0 reduziert wird. Da die Merkmalsextraktion auf jeden Frame der zu klassifizierenden Bildsequenz parallel angewandt werden muss, wird die Schicht in einem sogenannten TimeDistributed-Wrapper implementiert.
   Beispiel für valide Form einer Ausgabe (für Sequenzlänge 12, ResNet50): `(12, 2048)`
   Beispiel für valide Form einer Ausgabe (für Sequenzlänge 12, EfficientNetB0): `(12, 1280)`

1. ResNet and ResNetV2, `https://keras.io/api/applications/resnet/#resnet50-function`

3. **Bidirectional-Wrapper mit LSTM**: Die Sequenz von Feature-Maps wird in der nächsten Schicht mit einem LSTM verarbeitet. Da das LSTM aus 256 Neuronen besteht und mit Hilfe des Bidirectional-Wrappers in beide Richtungen arbeitet, können insgesamt 512 Merkmalsdimensionenen erfasst werden. Desweiteren werden mehrere Maßnahmen gegen Overfitting in das Modell etabliert: Zum einen wird auf die Eingaben und rekurrenten Verbindungen der LSTM-Zellen ein Dropout von 0.5 eingeführt, zum anderen wird auf die Gewichte der Eingangsverbindungen und rekurrenten Verbindungen eine L2-Regularisierung von 0.00003 angewandt. Die Ausgabe des BiLSTM enthält somit eine verdichtete Darstellung der gesamten Bildsequenz, einschließlich relevanter zeitlicher Abhängigkeiten.
   Beispiel für valide Form einer Ausgabe: `(512)`
4. **Dropout-Layer**: Nach der Sequenzverarbeitung wird eine weitere Dropout-Schicht der Stärke 0.25 auf die gesamte Ausgabe des BiLSTMs angewandt.
5. **Dense-Layer**: In einer abschließenden vollständig verbundenen Schicht werden die vom BiLSTM extrahierten Merkmale in die Wahrscheinlichkeiten für die einzelnen Klassen umgewandelt, wobei es für jede Klasse ein Neuron in der Ausgabeschicht gibt. Dabei werden besonders große Gewichte bestraft, indem eine weitere L2-Regularisierung von 0.003 zum Einsatz kommt. Als Aktivierungsfunktion wird *softmax* genutzt, welche die Rohwerte in Wahrscheinlichkeiten für die Klassen umwandelt (Summe über alle Ausgabeneuronen ist 1).
   Beispiel für valide Form einer Ausgabe: `(3)`

In Abbildung 1 ist der Aufbau des Modells schematisch dargestellt.

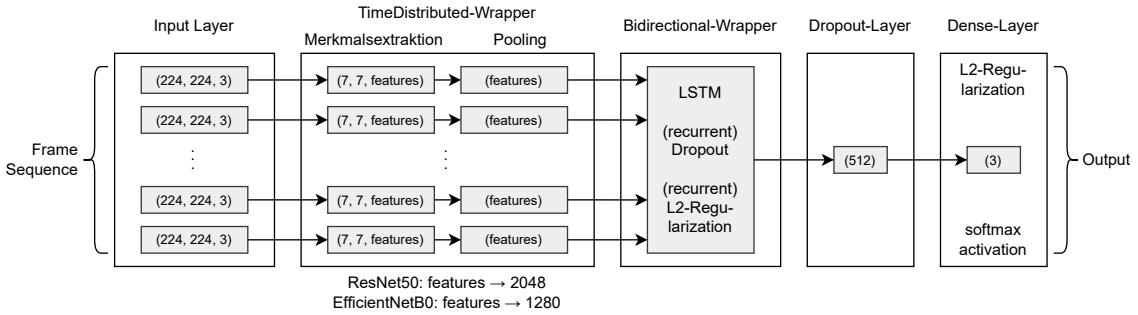

Abbildung 1: Schematische Darstellung der Schichten des Modells

2.3 Training

Im folgenden werden der Ablauf sowie die Parameter des Trainings erläutert.

Trainingskonfigurationen:
Insgesamt sollen 6 verschiedene Konfigurationen der Modellarchitektur trainiert werden. Dabei werden zum einen zwei verschiedene Merkmalsextraktoren als Baseline-Modell getestet (ResNet50 und EfficientNetB0), zum anderen wird die zu klassifizierende Sequenzlänge variiert (8 Frames, 12 Frames, 16 Frames). Um den variierenden Speicherbedarf durch die verschiedenen Sequenzlängen auszugleichen, wird die Batch-Size entsprechend angepasst (8 Frames → 12er Batches, 12 Frames → 8er Batches, 16 Frames → 4er Batches). Zur besseren Vergleichbarkeit der Modelle wurde auf *early stopping* verzichtet und alle Konfigurationen werden für exakt 9 Epochen trainiert.

Optimierer:
Zum Trainieren wird der Optimizer AdamW verwendet, eine um Gewichtszerfall (*weight decay*) erweiterte Variante des Adam-Optimizers (*adaptive moment estimation* mit L2-Regularisierung). AdamW verwaltet für jede Gewichtskomponente eine eigene Lernrate, die dynamisch angepasst wird, wodurch auf einen zusätzlichen LR-Scheduler verzichtet werden kann.

Verlustfunktion:
Als zu minimierende Verlustfunktion wird *categorical cross-entropy loss* verwendet:

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \text{ mit Anzahl Klassen } C, \text{ Grundwahrheit } y_i \text{ und Vorhersage } \hat{y}_i$$

Merkmalsextraktoren:
Dem ausgewählten Merkmalsextraktor (ResNet50 bzw. EfficientNetB0) werden initial vortrainierte Gewichte via `imagenet`[2] zugewiesen. Desweiteren wird in den ersten 3 Epochen des Trainings ein Fine-Tuning der Gewichte durchgeführt. Die Anzahl von Schichten, deren Gewichte für das Training freigegeben werden sollen, wird mit folgender Formel berechnet:

$$u_i = \begin{cases} \left\lceil \frac{d}{2^{(2+i)}} \right\rceil & \text{if } 0 \leq i \leq 2 \\ 0 & \text{otherwise} \end{cases} \quad \text{für Epoche } i \text{ und Tiefe des Merkmalsextraktors } d$$

Lernrate:
Die Größe der Lernrate wird basierend auf $\lambda = 0.001$ nach folgender Vorschrift gesetzt:

$$\lambda_i = \begin{cases} \frac{\lambda}{10 \cdot 2^i} & \text{if } 0 \leq i \leq 2 \\ \max(\frac{\lambda}{2^i}, 1 \cdot 10^{-5}) & \text{otherwise} \end{cases} \quad \text{für Epoche } i \text{ und Lernrate } \lambda$$

Dadurch ergeben sich die Parametrisierungen und somit auch die vollständige Trainingskonfiguration für alle 9 Epochen, wie in Tabelle 2 dargestellt.

Tabelle 2: Keras-Schichten für Fine-Tuning der Merkmalsextraktoren und Lernrate pro Epoche

Epoche	**ResNet50** ($d = 176$)	**EfficientNetB0** ($d = 239$)	**Lernrate** ($\lambda = 0.001$)
Epoche 1	$u_0 = 44$ (25%)	$u_0 = 60$ (25%)	$\lambda_0 = 1 \cdot 10^{-4}$
Epoche 2	$u_1 = 22$ (12.5%)	$u_1 = 30$ (12.5%)	$\lambda_1 = 5 \cdot 10^{-5}$
Epoche 3	$u_2 = 11$ (6.25%)	$u_2 = 15$ (6.25%)	$\lambda_2 = 2.5 \cdot 10^{-5}$
Epoche 4	$u_i = 0$ (n.a.)	$u_i = 0$ (n.a.)	$\lambda_3 = 1.25 \cdot 10^{-4}$
Epoche 5			$\lambda_4 = 6.25 \cdot 10^{-5}$
Epoche 6			$\lambda_5 = 3.125 \cdot 10^{-5}$
Epoche 7			$\lambda_6 = 1.5625 \cdot 10^{-5}$
Epoche 8			$\lambda_7 = 1 \cdot 10^{-5}$
Epoche 9			$\lambda_8 = 1 \cdot 10^{-5}$

2.4 Validierung

Neben den Testdaten (siehe Tabelle 1) werden für eine aussagekräftige Evaluation geeignete Metriken benötigt. Diese müssen für Multi-Klassen-Probleme geeignet sein und insbesondere mit unausgewogenen Klassen umgehen können, deshalb sind *accuracy* und *area under curve* ungeeignet.

Precision: Misst den Anteil der korrekt vorhergesagten positiven Instanzen im Verhältnis zu allen Instanzen, die als positiv vorhergesagt wurden:

$$P_i = \frac{\mathrm{TP}_i}{(\mathrm{TP}_i + \mathrm{FP}_i)} \text{ bzw. } P_{\text{micro}} = \frac{\sum_{i=1}^{C} \mathrm{TP}_i}{\sum_{i=1}^{C} (\mathrm{TP}_i + \mathrm{FP}_i)} \text{ (aggregierter Durchschnittswert)}$$

Recall: Misst den Anteil der korrekt vorhergesagten positiven Instanzen im Verhältnis zu allen tatsächlichen positiven Instanzen.

$$R_i = \frac{\mathrm{TP}_i}{(\mathrm{TP}_i + \mathrm{FN}_i)} \text{ bzw. } R_{\text{micro}} = \frac{\sum_{i=1}^{C} \mathrm{TP}_i}{\sum_{i=1}^{C} (\mathrm{TP}_i + \mathrm{FN}_i)} \text{ (aggregierter Durchschnittswert)}$$

F1-Score: Ist das harmonische Mittel von Precision und Recall und bietet eine ausgewogene Metrik zur Bewertung der Modellleistung. Bei unbalancierten Daten können die F1-Scores von Klassen mit vielen Instanzen höhere Gewichte erhalten, wenn sie auf einen Wert reduziert werden sollen.

$$F_i = \frac{2 \cdot P_i \cdot R_i}{P_i + R_i} \text{ bzw. } F_{\text{weighted}} = \frac{\sum_{i=1}^{C} w_i \cdot F_i}{\sum_{i=1}^{C} w_i} \text{ mit } w_i = \frac{\#\text{Instanzen Klasse } i}{\#\text{Instanzen gesamt}}$$

2. ImageNet, `https://www.image-net.org/`

3 Implementierung

Das in dieser Untersuchung entwickelte Modell (siehe Abs. 2.2) wurde mit TensorFlow in Python implementiert. Als Paket- und Umgebungsmanagement wurde MiniForge[3] mit Mamba verwendet. Die folgenden Tools bzw. Versionen müssen in der virtuellen Umgebung installiert sein:

- Python 3.12.9
- tensorflow 2.18.0
- numpy 2.0.2
- keras 3.8.0
- NVIDIA CUDA 12.5.82

PyTorch vs. TensorFlow:
Für die Implementierung des Modells wurden die beiden ML-Frameworks PyTorch und TensorFlow betrachtet:
PyTorch ist sehr dynamisch bzw. erweiterbar und deshalb für Forschung und Experimente prinzipiell geeignet. Durch die flexiblen Nutzungsmöglichkeiten ergibt sich allerdings auch eine steilere Lernkurve. Der Einsatz von PyTorch ist deshalb vor allem dann sinnvoll, wenn mehr Kontrolle über das Modell(training) benötigt wird.
TensorFlow hingegen bietet durch die große Anzahl von High-Level-Schnittstellen und die umfangreiche Dokumentation einen einfacheren Einstieg. Es ist zudem sehr gut dazu geeignet, mit wenig Code einen lauffähigen Prototypen mit automatischen Optimierungen zu erstellen. Da im Rahmen dieser Arbeit keine neuen Komponenten für Netze entwickelt werden, sondern lediglich bereits bestehende zu einem neuen Modell kombiniert werden, und zudem wenig praktisches Vorwissen vorhanden war, ist TensorFlow als Framework die bessere Wahl.

4 Evaluation

Insgesamt wurden 6 Modellkonfigurationen für jeweils 9 Epochen trainiert (siehe Abs. 2.3). Mit den in Abs. 2.4 aufgeführten Metriken ergeben sich die folgenden Zahlen zur Performance der Modelle:

Tabelle 3: Auswertung der Klassifikationsergebnisse nach 9 Epochen Training

Modellkonfiguration (FE-Sequenzlänge)	**Zeit** pro Item	**Precision**				**Recall**				**F1-Score**			
		`OR`	`FS`	`FR`	micro	`OR`	`FS`	`FR`	micro	`OR`	`FS`	`FR`	weighted
ResNet50-08	44 ms	0.8449	0.9160	0.8853	0.8938	0.9980	0.8263	0.9318	0.8929	0.9126	0.8687	0.9082	0.8924
ResNet50-12	67 ms	0.8310	0.9292	0.8904	0.9003	0.9990	0.8280	0.9417	0.8990	0.9032	0.8759	0.9155	0.8988
ResNet50-16	103 ms	0.8573	0.9282	0.8833	0.8977	0.9980	0.8225	0.9412	0.8964	0.9189	0.8725	0.9114	0.8961
EfficientNetB0-08	38 ms	0.7099	0.9049	0.8433	0.8531	0.9920	0.7422	0.9150	0.8531	0.8230	0.8174	0.8782	0.8505
EfficientNetB0-12	59 ms	0.7235	0.8893	0.8759	0.8689	0.9820	0.8020	0.9002	0.8651	0.8263	0.8437	0.8880	0.8665
EfficientNetB0-16	90 ms	0.6801	0.9071	0.8484	0.8540	0.9830	0.7455	0.9137	0.8495	0.7958	0.8197	0.8799	0.8507

Zunächst kann festgestellt werden, dass die ResNet50-Modelle in allen Konfigurationen besser abschneiden als EfficientNetB0, was vermutlich auf die höhere Parameteranzahl von ResNet50 zurückzuführen ist, was sich dementsprechend auch in der Detektionszeit wiederspiegelt. Innerhalb derselben Baseline-Architektur steigen die durchschnittlichen Werte für Precision, Recall und F1-Score leicht mit der Sequenzlänge, wobei speziell die Modelle mit einer Länge von 12 geringfügig besser abschneiden.
Die besten Precision-Werte werden in der Klasse `FS` mit ResNet50 erreicht, hingegen liefert EfficientNetB0 besonders bei der Klasse `OR` relativ viele False-Positives. Dies deckt sich mit den Recall-Werten; die besten Werte wurden von ResNet50 in der Klasse `OR` erzielt, die meisten False-Negatives treten allerdings bei EfficientNetB0 in der Klasse `FS` auf. Das bedeutet, dass EfficientNetB0 bei der Trennung der Klassen `OR` und `FS` mehr Probleme hat als ResNet50, wodurch zu viele `OR`-Instanzen fälschlicherweise zugeordnet und einige relevante `FS`-Items zu übersehen wurden.
Die etwas niedrigeren F1-Scores bei `FS` deuten darauf hin, dass ResNet50 bei dieser Klasse mehr Probleme hat, eine optimale Balance zwischen Precision und Recall zu erreichen. Der geringfügig

3. MiniForge3, `https://github.com/conda-forge/miniforge/`

höheren F1-Scores bei `FR` legen nahe, dass EfficientNetB0 vor allem bei dieser Klasse eine gute Balance zwischen Precision und Recall erzielt.
Zusammenfassend kann gesagt werden, dass die Modelle insgesamt recht dicht beieinander liegen. Die besten Werte konnten für das ResNet50-Modell mit einer Sequenzlänge von 12 erreicht werden (ResNet50-12), was einen guten Trade-Off zwischen Performance und Rechenleistung darstellt.

5 Zusammenfassung

5.1 Fazit

Im Rahmen dieser Arbeit wurde eine Architektur zur Klassifikation von DeepFake-Bildsequenzen entwickelt und evaluiert. Die Modelle bestehen aus einem Merkmalsextraktor (ResNet50 oder EfficientNetB0) als Baseline-Klassifikator mit einem dahinterliegenden BiLSTM und unterscheiden die Klassen *original*, *face-swap* und *face-reenactment* recht gut. Dabei konnte gezeigt werden, dass mit ResNet50 insgesamt leicht bessere Ergebnisse erzielt werden können. Obwohl die Leistung aller Modelle mit zunehmender Sequenzlänge geringfügig zunimmt, scheint bei Sequenzen der Länge 12 ein Sweet-Spot bezogen auf die verwendeten Trainingsparameter erreicht zu sein. Die ResNet50-Modelle haben zwar Schwierigkeiten mit der `FS`-Klasse, das größte Problem der EfficientNetB0-Modelle bleibt aber die Trennung der Klassen `OR` und `FS`.

5.2 Ausblick für zukünftige Arbeiten

Weitere Klassen:
Im DF40-Datensatz sind neben *face-swap*- und *face-reenactment*-Sequenzen auch Erzeugnisse von *entire face synthesis*- oder *face edit*-Tools vorhanden. In einer weiterführenden Untersuchung kann geprüft werden, ob die Modelle auch für 4 bzw. 5 Klassen ausreichend gute Ergebnisse erzielen.

Vision Transformer als Merkmalsextraktor:
Die entwickelten Modelle nutzen aktuell beide CNN-Architekturen, entweder ResNet50 oder EfficientNetB0 als Baseline-Klassifikator. Vision Transformer stellen einen alternativen Ansatz zur Merkmalsextraktion mittels *self attention* dar. In einer weiterführenden Arbeit könnte ein entsprechend angepasstes Transformer-Modell mit der Performance der Modelle dieser Arbeit verglichen werden.

Data Augmentation:
In dieser Arbeit wurden Klassengewichte genutzt, um unausgewogene Klassen zu balancieren. Alternativ könnte mit einer Datenaugmentation eine fortgeschrittene Oversampling-Technik eingesetzt werden, um zusätzliche Datenpunkte zu erstellen. Eine gute Datenaugmentation hat das Potenzial, das Generalisierungsvermögen der Modelle weiter zu steigern.

Zusätzliche Validierungsdaten:
Der DF40-Datensatz wird zwar bereits mit einem Validierungssplit ausgeliefert, allerdings sollte noch mindestens ein weiteres Datenset (z.B. DeepSpeakV1[4]) hinzugezogen werden, um die Generalisierung auf einer breiteren Datenbasis zu bestätigen.

Erweiterte Vorverarbeitung:
Das Modell kann aktuell nur Bildsequenzen mit Framegröße `(224, 224, 3)` klassifizieren. Deshalb sollte in Framework um das Modell herum implementiert werden, welches geeignete Frames zunächst aus einem zu überprüfenden Video extrahiert (*face detection*) und zuschneidet (*cropping*). Erst mit der extrahierten Sequenz kann das Modell praktisch genutzt werden.

Auflösung erhöhen:
Zu guter Letzt verarbeiten die aktuell verwendeten Merkmalsextraktoren ResNet50 und EfficientNetB0 Bilder der Größe `(224, 224, 3)`. Da die Bilder im DF40-Datensatz eine Größe von `256x256x3` haben, gehen durch die *resizing*-Operation einige Informationen verloren. Durch die Verwendung gesteigerter Netztiefen und -breiten könnten die Ergebnisse eventuell weiter verbessert werden.

4. DeepSpeakV1, `https://huggingface.co/datasets/faridlab/deepspeak_v1`

Literatur

[1] Kaiming He u. a. „Deep Residual Learning for Image Recognition“. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[2] Mingxing Tan und Quoc V. Le. „EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks“. In: *CoRR* abs/1905.11946 (2019). arXiv: 1905.11946. URL: http://arxiv.org/abs/1905.11946.

[3] Zhiyuan Yan u. a. „DF40: Toward Next-Generation Deepfake Detection“. In: *arXiv preprint arXiv:2406.13495* (2024).

Anhang

Listing 1: Ordnerstruktur des DF40-Datensets im `io/`-Verzeichnis

```
df40/
    test/
        original/
            ffpp/frames/
                <item>/*.png
        face_swap/
            <tool>/frames/
                <item>/*.png
        face_reenact/
            <tool>/frames/
                <item>/*.png
    train/
        original/
            ffpp/frames/
                <item>/*.png
        face_swap/
            <tool>/frames/
                <item>/*.png
        face_reenact/
            <tool>/frames/
                <item>/*.png
```