Erkennung von Deepfakes mittels Convolutional Neural Networks

Projektarbeit im Studiengang Informatik

Vorname Nachname

Matrikelnummer: 1234567

Abgabedatum: 05. Juni 2025

Betreuer: Prof. Dr. Max Mustermann
Zweitkoordinator: Dr. Erika Beispiel

Abstract

Im Rahmen dieser Projektarbeit wird untersucht, wie Convolutional Neural Networks (CNNs) eingesetzt werden können, um Deepfake-Bilder von echten Aufnahmen zu unterscheiden. Dazu werden verschiedene Datensätze (u. a. Kaggle–Deepfake-Datensätze) zusammengeführt, vorverarbeitet und als Trainings- und Validierungssets aufbereitet. Als Basis dient ein vortrainiertes ResNet50-Netzwerk, das durch Transfer Learning auf die Erkennung von Deepfakes adaptiert wird. Der Trainingsprozess umfasst sowohl das Einfrieren der Basislayer als auch ein anschließendes Fine-Tuning. Die Ergebnisse werden anhand von Kennzahlen wie Accuracy, Precision, Recall, F1-Score und AUC evaluiert. Abschließend werden Limitationen und mögliche Verbesserungsstrategien diskutiert.

Inhaltsverzeichnis

1 Einleitung

Deepfakes sind synthetisch erzeugte Medien, bei denen das Gesicht einer Person realistisch in Video- oder Bildmaterial eingefügt wird. Mit wachsender Rechenleistung und verbesserten Algorithmen lassen sich derartige Fälschungen immer schwieriger mit bloßem Auge erkennen. Die damit einhergehenden Risiken reichen von politischer Manipulation bis hin zum Identitätsdiebstahl. Ziel dieser Arbeit ist es, auf Basis moderner Deep-Learning-Techniken automatische Verfahren zur Erkennung von Deepfakes zu entwickeln und zu evaluieren.

In den letzten Jahren wurden zahlreiche Methoden vorgestellt, die tiefe neuronale Netze nutzen, um Artefakte in Deepfake-Bildern zu detektieren (vgl. **agarwal2020detecting**, **rossler2019faceforensics**++). Viele Arbeiten beschränken sich auf bestimmte Arten von Deepfake-Algorithmen (z. B. GAN-basiert), während andere einen breiteren Ansatz wählen, der auf Merkmale wie unnatürliche Augenreflexion, fehlende Gesichtsmerkmale oder Inkonsistenzen in den Spatial-Frequenzen abzielt.

In Kapitel 2 werden die verwendeten Datensätze genauer beschrieben. Kapitel 3 stellt die grundlegende Methodik vor, gefolgt von technischen Details zur Implementierung in Kapitel 4. Die experimentelle Evaluation und die erzielten Ergebnisse sind in Kapitel 5 zusammengefasst. Abschließend erfolgt in Kapitel 6 eine kritische Reflexion der gefundenen Resultate, bevor in Kapitel 7 ein Fazit gezogen wird.

2 Datensätze

Ein zentraler Baustein dieser Arbeit war die sorgfältige Auswahl und systematische Vorverarbeitung verschiedener Deepfake-Bilddatensätze, um eine robuste und realitätsnahe Klassifikation zu ermöglichen. Für das Training und die Evaluierung des Modells wurden insgesamt drei öffentliche Datensätze von Kaggle herangezogen, welche im Folgenden näher beschrieben werden.

2.1 Verwendete Datensätze

- **Deepfake-vs-Real-Classification**[1]: Dieser Datensatz enthält etwa 28.600 echte (*real*) und 28.600 Deepfake-Bilder (*fake*), sodass beide Klassen annähernd gleich stark vertreten sind. Die Bilder liegen im JPG-Format vor und zeigen unterschiedlichste Gesichter in diversen Umgebungen.
- **Detect AI-Generated Faces: High-Quality Dataset**[2]: Ein kleiner, aber qualitativ besonders hochwertiger Datensatz, der speziell für die Unterscheidung von KI-generierten Gesichtern entwickelt wurde. Er enthält 1.001 Fake-Bilder und 2.002 Real-Bilder, jeweils mit klarer Zuordnung.
- **deepfake and real images**[3]: Der größte im Projekt verwendete Datensatz umfasst rund 95.092 als *fake* und 95.213 als *real* gelabelte Bilder, insgesamt also etwa 190.000 Gesichter. Die Bilder sind bereits im Standardformat (256x256 Pixel, JPG) verfügbar.

Alle Datensätze wurden zunächst unabhängig voneinander gesichtet, geprüft und nach den beiden Zielklassen (*real*, *fake*) kategorisiert.

2.2 Datenorganisation und -aufteilung

Um eine einheitliche und automatisiert verarbeitbare Datenbasis für das Deep-Learning-Modell zu schaffen, wurden sämtliche Bilder zunächst in eine gemeinsame Verzeichnisstruktur überführt:

[1] https://www.kaggle.com/datasets/prithivsakthiur/deepfake-vs-real-60k
[2] https://www.kaggle.com/datasets/shahzaibshazoo/detect-ai-generated-faces-high-quality-dataset
[3] https://www.kaggle.com/datasets/manjilkarki/deepfake-and-real-images

```
Dataset/
|-- train/
|   |-- real/
|   \-- fake/
|-- validation/
|   |-- real/
|   \-- fake/
\-- test/
    |-- real/
    \-- fake/
```

Die Bilder wurden anhand ihrer Ursprungsdatensätze und Labels den jeweiligen Unterordnern `real` und `fake` zugeordnet und dabei auf die drei Datensplits Training, Validierung und Test verteilt.

2.3 Vorverarbeitung und Datenpipeline

Die technische Vorverarbeitung der Bilder erfolgt im Training vollständig automatisiert durch den Einsatz eines `ImageDataGenerator` aus Keras, kombiniert mit der spezifischen Vorverarbeitungsfunktion für ResNet50 (`preprocess_input`). Das Vorgehen ist wie folgt:

- **Bildgröße:** Alle Bilder werden beim Laden auf eine einheitliche Auflösung von 256x256 Pixeln gebracht (per `target_size=(256, 256)`), um optimal mit der Architektur von ResNet50 zu harmonieren.
- **Normalisierung:** Die Pixelwerte werden nicht nur skaliert, sondern gemäß der ResNet50-Konvention kanalweise normalisiert, um einen stabilen Wertebereich für das Netzwerk zu gewährleisten.
- **Datenaugmentation:** Während zu Projektbeginn kurzzeitig verschiedene Augmentationstechniken (z.B. Rotation, Flip, Zoom) getestet wurden, verzichtet der finale Stand vollständig auf künstliche Erweiterungen oder Verzerrungen. Die Trainingsdaten werden also ohne zusätzliche Transformationen verwendet.
- **Batching und Label-Encoding:** Die Bilddaten werden im Trainingsprozess in Batches zu je 64 Bildern geladen. Die Zuordnung zu den Klassen (*real* oder *fake*) erfolgt automatisch anhand der Verzeichnisstruktur und wird als binäre Label (0/1) dem Modell zugeführt.

Die Verwendung der `flow_from_directory`-Funktion gewährleistet, dass alle Splits (Training, Validation, Test) effizient und speicherschonend verarbeitet werden können.

2.4 Zusammenfassung der Datenbasis

Zusammengefasst standen nach Abschluss der Vorverarbeitung und Sortierung folgende Datenmengen für das Projekt zur Verfügung:

- **Deepfake-vs-Real-Classification:** ca. 28.600 Real, 28.600 Fake
- **Detect AI-Generated Faces:** 2.002 Real, 1.001 Fake
- **deepfake and real images:** ca. 95.213 Real, 95.092 Fake

Durch die einheitliche Struktur und konsequente Normalisierung wurden optimale Voraussetzungen geschaffen, um ein robustes Deep-Learning-Modell auf Basis großer und vielseitiger Bilddaten zu trainieren. Auf manuelle oder künstliche Balancierung der Klassen wurde bewusst verzichtet, da die verwendeten Datensätze bereits eine weitgehende Gleichverteilung von echten und gefälschten Bildern aufweisen.

3 Methodik

3.1 Experimentumgebung

Die Durchführung sämtlicher Trainings- und Evaluationsläufe erfolgte auf einer lokalen Workstation mit NVIDIA RTX 2070 Super GPU (8 GB VRAM) unter Windows 10. Für das gesamte Deep-Learning-Framework kam TensorFlow (Version 2.x) in Verbindung mit Keras zum Einsatz. Der Trainingsprozess wurde über Visual Studio Code gesteuert und sowohl GPU- als auch CPU-Implementierungen wurden automatisch erkannt und genutzt. Insbesondere die Aktivierung des Mixed Precision Trainings (`mixed_float16`) trug dazu bei, die GPU-Auslastung zu optimieren und die Speichereffizienz sowie Trainingsgeschwindigkeit deutlich zu steigern. Typischerweise betrug die Dauer eines Trainingsschritts (bei einer Batchgröße von 64) im Mittel ca. 385 ms.

3.2 Verwendete Datensätze

Für das Training und die Validierung des Deepfake-Klassifikators wurden mehrere öffentlich verfügbare Bilddatensätze von Kaggle integriert (siehe Kapitel 2). Im Fokus standen:

- **Deepfake and Real Images:** Hauptdatensatz mit 190.000 Bildern (256×256 px, JPG), aufgeteilt in 140.000 Trainingsbilder, 39.400 Validierungsbilder und 10.905 Testbilder.
- **Detect AI-Generated Faces (High Quality):** Ergänzend hinzugefügt, um die Varianz im Training zu erhöhen, mit 1.001 Fake- und 2.202 Real-Bildern.

Alle Daten wurden auf einen konsistenten Verzeichnisbaum (`train/`, `validation/`, `test/`, jeweils mit Unterordnern `real`/`fake`) gebracht, um eine automatisierte, fehlerfreie Datenzufuhr während des Trainings zu gewährleisten.

3.3 Modellarchitektur

Das eingesetzte neuronale Netz basiert auf einem vortrainierten `ResNet50`-Backbone (ImageNet-Gewichte, ohne Top-Layer). Diese Architektur wurde gewählt, da sie in der Literatur als sehr robust für Bildklassifikationsaufgaben gilt und in eigenen Vorversuchen anderen Modellen wie EfficientNetB0 überlegen war.

Der eigentliche Klassifikationskopf besteht aus den folgenden Schichten:

- `GlobalAveragePooling2D()` als Übergang von Feature Maps zu einem Vektor.
- `Dense(128)`-Schicht mit ReLU-Aktivierung und L2-Regularisierung ($\lambda = 0{,}002$), um Overfitting zu begegnen.
- `Dropout(0,6)` zur weiteren Reduzierung von Überanpassung.
- `Dense(1)` mit Sigmoid-Aktivierung und explizitem `float32`-Output, um trotz Mixed Precision stabile und exakte Ausgaben zu gewährleisten.

Im finalen Modell ist der gesamte ResNet50-Block von Beginn an trainierbar (`base_model.trainable = True`). Auf ein stufenweises Fine-Tuning einzelner Layer wird somit verzichtet; alle Schichten werden gleichberechtigt im Gradientenabstieg aktualisiert.

3.4 Vorverarbeitung und Datenpipeline

Die Bildvorverarbeitung und das Datenhandling erfolgen vollautomatisch durch den `ImageDataGenerator` aus `tensorflow.keras.preprocessing.image`. Als Preprocessing wird die `preprocess_input`-Funktion von ResNet50 verwendet, die neben der Skalierung auf 256×256 Pixel auch eine kanalweise Normalisierung vornimmt.

Im Gegensatz zu frühen Projektphasen wird im finalen Ansatz komplett auf künstliche Datenaugmentation (Rotation, Flip, Zoom etc.) verzichtet, da empirische Tests gezeigt haben, dass diese im Gesamtsystem keine Vorteile bringen, sondern die Stabilität sogar vermindern können.

Bilder werden batchweise (Batchgröße 64) geladen und die Labels (`real` / `fake`) automatisch aus der Ordnerstruktur als binäre Klassen extrahiert.

3.5 Trainingskonfiguration und Hyperparameter

Die Modellkompilierung erfolgt mit dem Adam-Optimierer (Standardparameter), *binary_ crossentropy* als Verlustfunktion und *accuracy* als zentrale Trainingsmetrik. Wesentliche Trainingsparameter:

- Input-Größe: $256 \times 256 \times 3$
- Batchgröße: 64
- Trainingsdauer: 50 Epochen (mit früherem Abbruch durch Callback möglich)
- Mixed Precision Policy: `mixed_float16`

Callbacks für Monitoring und Steuerung:

- `ModelCheckpoint`: Speichert das jeweils beste Modell (nach Validierungs-Accuracy, Datei: `best_model_initial.h5`).
- `ReduceLROnPlateau`: Reduziert die Lernrate bei Stagnation der Validierungsverluste (Faktor 0,2, Patience 10, min_lr 1e−7).
- `TensorBoard`: Logging aller Trainingsmetriken und Lernverläufe zur späteren Analyse.

3.6 Ablauf des Trainings und der Evaluation

Der vollständige Trainingsprozess lässt sich wie folgt beschreiben:

1. Initialisierung des Modells und Laden der Trainings- und Validierungsdaten über die ImageDataGenerator-Pipeline.
2. Training über bis zu 50 Epochen mit Echtzeit-Monitoring aller Metriken (Trainings-/Validierungs-Accuracy und -Loss).
3. Speichern des besten Modells während des Trainings per Callback.
4. Nach Abschluss des Trainings erfolgt die Evaluation auf dem separaten Test-Set:
   - Berechnung von **Loss** und **Accuracy** mittels `model.evaluate(...)`
   - Erstellung eines detaillierten **Klassifikationsreports** (Precision, Recall, F1-Score) mittels `sklearn.metrics.classification_report`
   - Darstellung der **Konfusionsmatrix** mittels `sklearn.metrics.confusion_matrix`
5. Das finale Modell wird als `model_final.h5` gespeichert.

3.7 Versuchsplanung und Entwicklungsschritte

Im Verlauf des Projekts wurden zahlreiche Alternativen und Erweiterungen getestet, darunter:

- **Datenaugmentation:** Zu Beginn wurden diverse Augmentations (Rotation, Flip, Zoom) zur Erhöhung der Varianz erprobt. Sie wurden jedoch im finalen Ansatz entfernt, da sie zu schwankenden und weniger stabilen Trainingsergebnissen führten.
- **Stufenweises Fine-Tuning:** Ein schrittweises Training mit initial eingefrorenem Feature-Extractor und nachfolgend sukzessiv freigegebenen Layern (z. B. 10/50/100 Schichten) brachte keinen konsistenten Vorteil gegenüber dem sofortigen End-to-End-Training.

- **Hyperparameter-Tuning:** Verschiedene Lernraten ($\eta = 10^{-4}$ bis 10^{-6}), Regularisierungsstärken (L2, Dropout), Batchgrößen und Optimierer wurden evaluiert. Die finale Konfiguration orientiert sich an den stabilsten und am besten generalisierenden Settings.
- **Architekturvergleich:** Alternative Backbones wie EfficientNetB0 wurden verglichen, letztlich fiel die Entscheidung jedoch klar zugunsten von ResNet50 aus.
- **Datensatzdiversifizierung:** Das Einbinden zusätzlicher Datensätze (wie „Detect AI-Generated Faces“) führte zu einer Erhöhung der Varianz und geringfügigen Verbesserungen im Recall, jedoch nicht zu signifikant besseren Gesamtergebnissen.

3.8 Zusammenfassung

Die beschriebene Methodik gewährleistet ein reproduzierbares, transparentes und effizientes Training eines Deep-Learning-Modells zur Deepfake-Erkennung, wobei sämtliche Entscheidungen datengestützt und iterativ empirisch begründet wurden.

4 Implementierung

4.1 Datensätze und Use-Case

Zu Beginn wurde eine ausführliche Recherche durchgeführt, um geeignete Datensätze für die Bild-basierte Deepfake-Erkennung zu identifizieren. Da der Use-Case auf Einzelbildern und nicht auf Videos basiert, fiel die Wahl auf den Kaggle-Datensatz „Deepfake and Real Images“[1]:

- **Inhalt:** 190.000 JPG-Bilder in 256×256 Pixel.
- **Aufteilung:** Training (140.000), Validation (39.400), Test (10.905).
- **Gründe:** ausreichendes Volumen für Deep-Learning, standardisierte Auflösung, annähernde Klassenbalance.

4.2 Projektstruktur

```
Projekt/
|-- data/
|   |-- train/     % 140.000 Trainingsbilder
|   |-- val/       % 39.400 Validierungsbilder
|   \-- test/      % 10.905 Testbilder
|-- models/
|   |-- best_model_initial.h5     % nach Phase 1
|   |-- best_model_finetuned.h5   % nach Phase 2
|   \-- model_final.h5            % finales Modell
\-- src/
    |-- train_model.py    % Modell-Definition, Training & Fine-Tuning
    \-- evaluate.py       % Evaluation, Klassifikationsreport, Konfusionsmatrix
```

[1] https://www.kaggle.com/datasets/manjilkarki/deepfake-and-real-images

4.3 Modelldefinition

Der Kern des Trainingsskripts ist eine ResNet50-basierte Architektur, die Transfer Learning mit ImageNet-Gewichten nutzt. Anfangs wurden alle Basis-Layer eingefroren, später Teil-Fine-Tuning durchgeführt.

Listing 4.1: Definition in `train_model.py`

```
import tensorflow as tf
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras import mixed_precision

# Mixed Precision aktivieren
mixed_precision.set_global_policy('mixed_float16')

def create_model(freeze_base: bool = True, l2_factor: float = 0.001, dropout_r
    base = ResNet50(weights='imagenet', include_top=False, input_shape=(256,25
    base.trainable = not freeze_base

    x = GlobalAveragePooling2D()(base.output)
    x = Dense(128, activation='relu', kernel_regularizer=l2(l2_factor))(x)
    x = Dropout(dropout_rate)(x)
    out = Dense(1, activation='sigmoid', dtype='float32')(x)

    model = tf.keras.Model(inputs=base.input, outputs=out)
    return model
```

Erklärung wichtiger Komponenten

- **Mixed Precision:** Float16 für schnellere GPU-Auslastung.
- **GlobalAveragePooling2D statt Flatten:** Reduziert Überanpassung.
- **L2-Regularisierung (0.001):** Verhindert zu große Gewichte.
- **Dropout (0.5):** Weitere Regularisierung.

4.4 Training und Fine-Tuning

Das Training erfolgte in zwei Phasen, jeweils mit Callbacks zur Steuerung und Optimierung.

4.4.1 Phase 1: Initialtraining (10 Epochen)

Listing 4.2: Initialtraining in `train_model.py`

```
model = create_model(freeze_base=True, l2_factor=0.001, dropout_rate=0.5)
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
    loss='binary_crossentropy',
    metrics=['accuracy', tf.keras.metrics.AUC()]
)

# Data Augmentation: Rotation, Flip, Zoom etc. wird in train_dataset integriert
callbacks_phase1 = [
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_b
    tf.keras.callbacks.ModelCheckpoint('models/best_model_initial.h5', monitor=
    tf.keras.callbacks.TensorBoard(log_dir='logs/fit/initial', update_freq='epo
]

model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=callbacks_phase1
)
# Speichern des finalen Modells
model.save('models/model_initial_final.h5')
```

Phase 2: Fine-Tuning (10 Epochen)

Listing 4.3: Fine-Tuning in `train_model.py`

```
# Freigabe der obersten 10 Basisschichten
for layer in model.layers[-10:]:
    layer.trainable = True

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
    loss='binary_crossentropy',
    metrics=['accuracy', tf.keras.metrics.AUC()]
)

callbacks_phase2 = [
    tf.keras.callbacks.ModelCheckpoint('models/best_model_finetuned.h5', monito
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2, patien
```

```
    tf.keras.callbacks.TensorBoard(log_dir='logs/fit/finetune', update_freq='ep
]

model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=callbacks_phase2
)
# Speichern als finales Modell
model.save('models/model_final.h5')
```

4.5 Ergebnisse und iterative Anpassungen

Nach jeder Phase wurde auf dem Testset evaluiert (Klassifikationsreport aus `sklearn.metrics` und Konfusionsmatrix mit `matplotlib`).

Ergebnisse vom 14.04.2025

Tabelle 4.1: Performance nach 10/10 Epochen

Modell	Acc.	Prec.	Rec.	F1	AUC	Epochen (init/ft)
best_model_initial.h5	0.7051	0.7710	0.6053	0.6782	–	10 / –
best_model_finetuned.h5	0.7026	0.7706	0.5990	0.6741	–	10 / 10
model_final.h5 (v1)	0.7026	0.7706	0.5990	0.6741	–	10 / 10

Anpassungen zum 20.04.2025

- Epochen auf 20 erhöht
- Fine-Tuning auf 50 Layer
- LR initial: 1×10^{-6}
- L2-Regularisierung: 0.002
- Dropout: 0.6
- ReduceLROnPlateau: factor=0.2, patience=2, min_lr=1e-7

Tabelle 4.2: Performance nach 20/20 Epochen, 50 Layer

Modell	Acc.	Prec.	Rec.	F1	AUC	Epochen (init/ft)
best_model_initial.h5*	0.7030	0.7866	0.5785	0.6667	–	20 / –
best_model_finetuned.h5*	0.6914	0.7762	0.5608	0.6511	–	20 / 20
model_final.h5 (v2)	0.6914	0.7762	0.5608	0.6511	–	20 / 20

Tabelle 4.3: Performance nach 20/20 Epochen, 100 Layer

Modell	Acc.	Prec.	Rec.	F1	AUC	Epochen (init/ft)
best_model_initial.h5**	0.7048	0.7678	0.6094	0.6795	–	20 / –
best_model_finetuned.h5**	0.6669	0.7640	0.5082	0.6104	–	20 / 20
model_final.h5 (v3)	0.6654	0.7658	0.5017	0.6063	0.7244	20 / 20

Weitere Versuche zum 22.04.2025

- Fine-Tuning auf 100 Layer

Neuer Ansatz zum 24.04.2025

- Rückkehr zu 10 Layer Fine-Tuning + Thresholding (0.5)
- EfficientNetB0 als Basis getestet

Tabelle 4.4: Performance EfficientNetB0 (20/20, 10 Layer)

Modell	Acc.	Prec.	Rec.	F1	AUC	Epochen (init/ft)
best_model_initial.h5***	0.7025	0.7682	0.6022	0.6752	–	20 / –
best_model_finetuned.h5***	0.7027	0.7614	0.6131	0.6793	–	20 / 20
model_final.h5 (v4)	0.7027	0.7614	0.6131	0.6793	–	20 / 20

4.5.1 Radikale Umstrukturierung

Um Overfitting weiter zu reduzieren, wurde:

- **Data Augmentation entfernt**, um verrauschte Gradienten durch starke Transformationen zu vermeiden.
- **LR-Patience auf 10 erhöht**, um dem Modell Zeit für langsame Anpassungen zu geben.
- **Fine-Tuning komplett ausgelassen**, um nur den vorkonfigurierten Kopf zu trainieren.
- **Zusätzlicher Datensatz** von Shahzaibshazoo[2] integriert, um die Bildvarianz und Robustheit zu steigern.

[2] https://www.kaggle.com/datasets/shahzaibshazoo/detect-ai-generated-faces-high-quality-dataset

Diese Maßnahmen führten zu stabileren Lernkurven, höherem Recall und geringerer Overfitting-Tendenz. Insgesamt wurden 1.000 Epochen durchgeführt (Trainingsdauer etwa 3,5 Tage).

Klassifikationsreport

	precision	recall	f1-score	support
Fake	0.67	0.79	0.73	5825
Real	0.76	0.63	0.69	6147
accuracy			0.71	11972

Konfusionsmatrix

$$\begin{pmatrix} 4606 & 1219 \\ 2246 & 3901 \end{pmatrix}$$

4.6 Letzte Optimierung

Abschließend wurde das gesamte ResNet50-Modell freigegeben und 50 Epochen trainiert, was eine sehr hohe Trennschärfe ermöglichte.

Klassifikationsreport

	precision	recall	f1-score	support
Fake	0.83	0.97	0.89	5825
Real	0.96	0.81	0.88	6147
accuracy			0.89	11972

Konfusionsmatrix

$$\begin{pmatrix} 5643 & 182 \\ 1149 & 4998 \end{pmatrix}$$

4.7 Ausblick

Die erzielten Ergebnisse sind sehr zufriedenstellend. Weitere sinnvolle Erweiterungen:

- **MobileNetV2-Integration** für ressourcenarme Umgebungen.
- **Gesichts-Extraktionspipeline** zur Fokussierung auf relevante Bildbereiche.
- **Zusätzliche Datensatz-Erweiterungen**, z. B. Deepfake vs. Real 60k, um Robustheit weiter zu steigern.

5 Evaluation

Nach dem Training wurden die Modelle auf dem Testset (ca. 10 % der Daten) evaluiert. Es wurden folgende Metriken ausgewertet:

- **Accuracy:** Anteil korrekt klassifizierter Bilder.
- **Precision / Recall / F1-Score:** Aus dem `sklearn.metrics`-Klassifikationsreport.
- **AUC:** Area Under the ROC-Kurve (verwaltet durch Keras während des Trainings).
- **Konfusionsmatrix:** Bild nicht gefunden, deshalb hier ausgelassen.

5.1 Quantitative Ergebnisse

Tabelle 5.1: Test-Ergebnisse verschiedener Modellvarianten

Modell	Acc.	Prec.	Rec.	F1-Score	AUC	Fine-Tune-Layer
best_initial (20/0)	0.7030	0.7866	0.5785	0.6667	-	-
best_finetuned (20/20-50)	0.6914	0.7762	0.5608	0.6511	-	50

5.2 Konfusionsmatrix

Warnung: Bild nicht gefunden: `figures/confusion_matrix.png`
Die Konfusionsmatrix wurde in dieser Version weggelassen.

5.3 Analyse der Ergebnisse

Die Ergebnisse zeigen, dass das Fine-Tuning mit zu vielen freigegebenen Layern (z. B. 50) zu Overfitting führte, während ein vorsichtiger Feinschliff (10 Layer) die beste Mischung aus Precision und Recall erreichte. Die AUC-Kurve (Abbildung **??**) bestätigt, dass das Modell selbst bei ungleichen Klassenverteilungen noch einigermaßen trennscharf bleibt.

Warnung: Bild nicht gefunden: `figures/roc_curve.png`
Die ROC-Kurve konnte nicht angezeigt werden.

6 Diskussion

Obwohl das initiale Training mit eingefrorenem ResNet50 breite Generalisierung lieferte (Accuracy 0,70), zeigt sich in den Feintuning-Durchläufen, dass zu umfangreiches Fine-Tuning (50 Layer) zu einer Reduktion der Recall-Rate führt (0,50 gegenüber 0,61). Mögliche Ursachen:

- **Datenheterogenität:** Unterschiedliche Bildquellen (verschiedener Kaggle-Datensätze) weisen nicht dieselben Kameraeigenschaften auf. Das Feintuning kann lokale Artefakte überanpassen.
- **Overfitting:** Trotz starker L2-Regularisierung und erhöhtem Dropout (0,6) zeigten die TensorBoard-Kurven nach Epochen 15–20 einen deutlichen Divergenz-Effekt zwischen Training und Validierung (vgl. Anhang, Abbildung A.1).
- **Fehlende große Testdaten:** Die finale Test-Stichprobe umfasste nur ca. 10

6.1 Limitationen der aktuellen Arbeit

- Systematisch wurden weder *Video-Deepfakes* noch Audio-Manipulationen berücksichtigt.
- Die Datensätze enthalten primär Gesichter in neutraler Mimik – extreme Gesichtsausdrücke (Lachen, Grimassen) wurden kaum repräsentiert.
- Die Trainingsdauer (je Lauf 4–6 Stunden auf einer RTX 2080 Ti) begrenzte das Experimentieren mit noch tieferen Netzwerken (z. B. EfficientNetV2-Large).

7 Fazit

Diese Projektarbeit zeigt, dass ein auf ResNet50 basierendes Transfer-Learning-Ansatz in der Lage ist, Deepfake-Bilder von echten Aufnahmen mit einer Accuracy von ca. 0,70 zu unterscheiden. Durch gezieltes Fine-Tuning (Freigabe nur der letzten 10 Layer) konnte ein guter Kompromiss zwischen Precision und Recall gefunden werden. Die Integration von L2-Regularisierung und erhöhtem Dropout verhindert zumindest teilweise Overfitting.

Für zukünftige Arbeiten wäre eine Ausweitung auf Video-Deepfakes sinnvoll, ebenso wie die Integration von multimodalen Ansätzen (z. B. Audio+Bild) und die Erforschung von Ensemble-Methoden.

Insgesamt leistet diese Arbeit einen Beitrag zur automatisierten Erkennung von Deepfakes und kann als Grundlage für weiterführende Projekte dienen.