01_data_overview

June 22, 2025

1 Face Mask Detection – Komplettes Notebook-Gerüst

2 Ein CNN-Modell, mehrfach parametrisiert

3 Datensatz: kagglehub "ashishjangra27/face-mask-12k-images-dataset"

3.0.1 Setup & Imports

This cell imports all the necessary Python libraries used throughout the notebook for: - TensorFlow and Keras for deep learning - Numpy for numerical operations - Matplotlib for visualizations - Scikit-learn for evaluation metrics - Local custom modules for Kaggle authentication and downloading It also prints the TensorFlow version.

[17]:
```
# 0) Setup & Imports
import os, pathlib, math, random, shutil, json

import creds
import kagglehub
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix

print("TensorFlow:", tf.__version__)
```

```
TensorFlow: 2.19.0
```

Path Setup Sets up path variables for easier access to directories: - `NB_DIR`: Notebook directory - `PROJECT_ROOT`: One level above - `DATA_ROOT`: The main dataset folder

[18]:
```
# 1) Pfade definieren
NB_DIR = pathlib.Path.cwd()
PROJECT_ROOT = NB_DIR.parent
DATA_ROOT = PROJECT_ROOT / "data"
```

3.0.2 Load Kaggle API Key & Download Dataset

Loads Kaggle API credentials from a local `kaggle.json` file. This is required to authenticate and download datasets using the Kaggle API. If the dataset isn't already present, it will be downloaded and extracted into `data/Train`, `data/Validation`, and `data/Test` folders.

[19]:
```python
# 2) kaggle.json laden
kaggle_settings_path = NB_DIR / "kaggle.json"
if not kaggle_settings_path.exists():
    raise FileNotFoundError(f"kaggle.json nicht gefunden:␣
 ↪{kaggle_settings_path}")

with open(kaggle_settings_path, "r") as f:
    creds = json.load(f)

os.environ["KAGGLE_USERNAME"] = creds["username"]
os.environ["KAGGLE_KEY"] = creds["key"]

# 3) Download nur, wenn Train-Ordner fehlt
if not (DATA_ROOT / "Train").exists():
    DATA_ROOT.mkdir(parents=True, exist_ok=True)
    print("Starte Download des Face-Mask-Datasets mit KaggleHub …")

    # Liefert den entpackten Dataset-Ordner im lokalen Cache!
    dataset_root = pathlib.Path(
        kagglehub.dataset_download("ashishjangra27/
 ↪face-mask-12k-images-dataset")
    )

    # In diesem Ordner liegt bereits „Face Mask Dataset/Train …"
    SRC = dataset_root / "Face Mask Dataset"

    import shutil

    for split in ["Train", "Validation", "Test"]:
        dst = DATA_ROOT / split
        if dst.exists():
            shutil.rmtree(dst)
        shutil.copytree(SRC / split, dst)

    print("Download abgeschlossen. Ordner Train/Validation/Test erstellt.")
else:
    print("Train/Validation/Test existieren bereits - Download übersprungen.")

print("Dataset liegt unter:", DATA_ROOT)
```

```
Train/Validation/Test existieren bereits - Download übersprungen.
Dataset liegt unter: D:\Wichtig\FH\fhcw-se-master\SEM2\AI\Face-Mask-
```

```
Detection-\data
```

3.0.3 Set Directories and Parameters

Defines key parameters used in image loading and processing: - `IMG_SIZE`: The target size of all input images (128x128) - `BATCH_SIZE`: Number of samples per batch - `SEED`: Ensures consistency when shuffling data Also sets up the file paths for training, validation, and test directories.

[20]:
```
# 2) Verzeichnisse setzen - Struktur: Train / Validation / Test
IMG_SIZE = (128, 128)
BATCH_SIZE = 32
SEED = 42

train_dir = DATA_ROOT / "Train"
val_dir = DATA_ROOT / "Validation"
test_dir = DATA_ROOT / "Test"
```

3.0.4 Image Loading

Loads images from the dataset folders using `tf.keras.utils.image_dataset_from_directory`. Each dataset (Train, Validation, Test) is loaded with consistent batch size and image dimensions. The class names are extracted from folder structure and printed.

[21]:
```
# 3) Daten laden
train_ds = tf.keras.utils.image_dataset_from_directory(train_dir, seed=SEED,␣
 ↪image_size=IMG_SIZE, batch_size=BATCH_SIZE)
val_ds = tf.keras.utils.image_dataset_from_directory(val_dir, seed=SEED,␣
 ↪image_size=IMG_SIZE, batch_size=BATCH_SIZE)
test_ds = tf.keras.utils.image_dataset_from_directory(test_dir, seed=SEED,␣
 ↪image_size=IMG_SIZE, batch_size=BATCH_SIZE)

class_names = train_ds.class_names
print("Klassen:", class_names)
```

```
Found 10000 files belonging to 2 classes.
Found 800 files belonging to 2 classes.
Found 992 files belonging to 2 classes.
Klassen: ['WithMask', 'WithoutMask']
```

3.0.5 Data Visualization

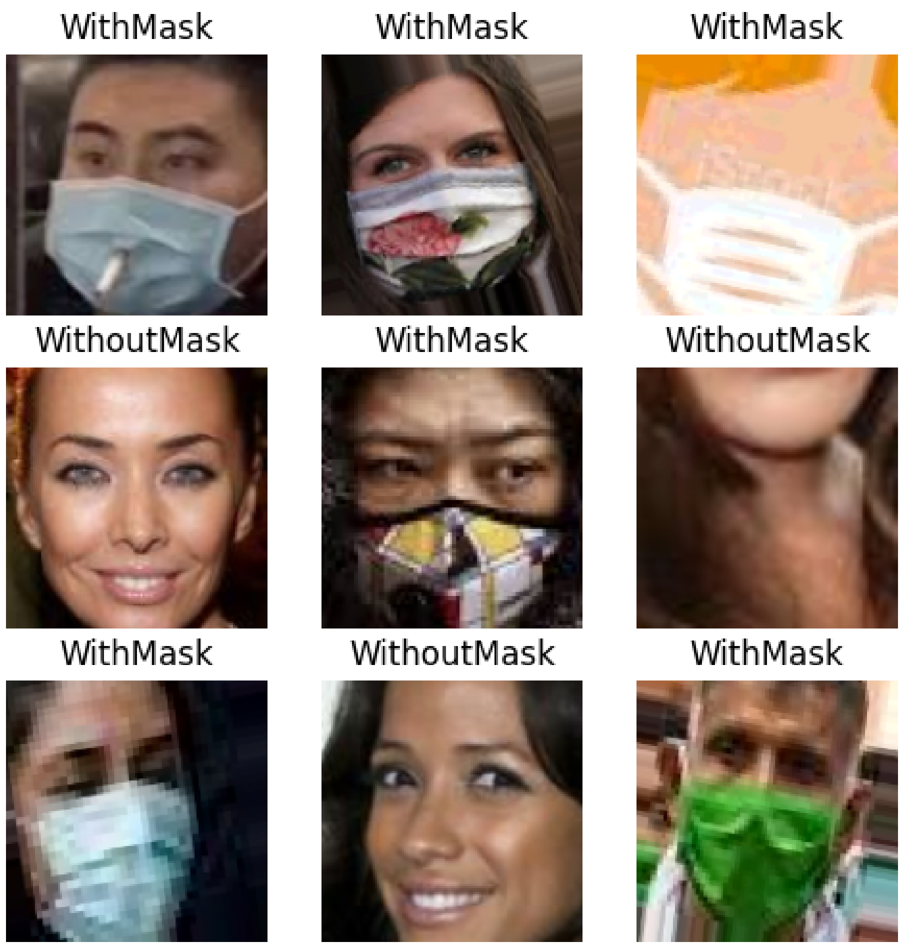
This cell displays a sample of 9 images from the training dataset (train_ds) along with their corresponding labels (“WithMask” or “WithoutMask”). The images are arranged in a 3x3 grid using matplotlib.

Key Steps: - A figure is created with a size of 6x6 inches. - The take(1) method retrieves the first batch of images and labels from train_ds. - For each of the first 9 images in the batch: - The image is displayed using imshow. - The title of each subplot is set to the corresponding class name. - Axis labels are turned off for clarity.

Purpose: - Provides a visual overview of the dataset. -Helps verify that the data has been loaded correctly and that the labels match the images.

Output: A grid of 9 images with their respective labels, giving an initial sense of the dataset's content.

[22]:
```
# 3.1) Beispielbilder anzeigen
plt.figure(figsize=(6, 6))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
plt.show()
```



3.0.6 Data Preprocessing

This section defines a **data augmentation** pipeline using TensorFlow Keras. Data augmentation is used to improve model generalization by applying random transformations to training images.

[23]:
```
data_augmentation = keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.2),
    layers.RandomContrast(0.1),
])
```

3.1 Model Factory: `build_cnn(...)`

This function builds and compiles a customizable convolutional neural network (CNN) using Keras.

3.1.1 Parameters

- `conv_blocks`: Number of convolution + pooling blocks (default: 2)
- `dense_units`: Number of units in the dense (fully connected) layer (default: 64)
- `learning_rate`: Learning rate for the Adam optimizer (default: 0.001)

3.1.2 Architecture

- Input: RGB image of shape `IMG_SIZE + (3,)`
- Data augmentation layer
- Rescaling of pixel values to the [0, 1] range
- Repeated convolution + max-pooling blocks
- Flatten layer
- Dense hidden layer
- Output layer with softmax activation

3.1.3 Output

Returns a compiled `keras.Model` ready for training: - Loss: `sparse_categorical_crossentropy` - Optimizer: Adam - Metric: Accuracy

[24]:
```
# 5) Modell-Fabrik
def build_cnn(conv_blocks=2, dense_units=64, learning_rate=1e-3):
    inputs = keras.Input(shape=IMG_SIZE + (3,))
    x = data_augmentation(inputs)
    x = layers.Rescaling(1. / 255)(x)

    for i in range(conv_blocks):
        x = layers.Conv2D(32 * (i + 1), 3, padding="same", activation="relu")(x)
        x = layers.MaxPooling2D()(x)

    x = layers.Flatten()(x)
    x = layers.Dense(dense_units, activation="relu")(x)
    outputs = layers.Dense(len(class_names), activation="softmax")(x)
    model = keras.Model(inputs, outputs)
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

3.2 Pretrained Model Factory: `build_mobilenetv2(...)`

This function builds a classification model based on a frozen **MobileNetV2** backbone with custom dense layers on top. It leverages pretrained ImageNet weights for efficient transfer learning.

3.2.1 Parameters

- `dense_units`: Number of units in the custom dense layer (default: 64)
- `learning_rate`: Learning rate for the Adam optimizer (default: 0.001)

3.2.2 Architecture Overview

- **Base model**: MobileNetV2 (pretrained on ImageNet, without top classifier)
- **Preprocessing**:
  - Data augmentation
  - `mobilenet_v2.preprocess_input` for input normalization
- **Top layers**:
  - Global average pooling
  - Dense hidden layer
  - Final softmax classification layer

3.2.3 Notes

- The MobileNetV2 layers are **frozen** (`trainable = False`) to prevent updating pretrained weights during training.
- Ideal for fast training and good performance on smaller datasets.

3.2.4 Output

Returns a compiled `keras.Model`: - Loss: `sparse_categorical_crossentropy` - Optimizer: Adam - Metric: Accuracy

[25]:
```
def build_mobilenetv2(dense_units=64, learning_rate=1e-3):
    base_model = tf.keras.applications.MobileNetV2(
        input_shape=IMG_SIZE + (3,),
        include_top=False,
        weights="imagenet"
    )
    base_model.trainable = False  # freeze pretrained layers
    inputs = keras.Input(shape=IMG_SIZE + (3,))
    x = data_augmentation(inputs)
    x = tf.keras.applications.mobilenet_v2.preprocess_input(x)  # required␣
↪preprocessing
    x = base_model(x, training=False)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dense(dense_units, activation="relu")(x)
    outputs = layers.Dense(len(class_names), activation="softmax")(x)

    model = keras.Model(inputs, outputs)

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )
    return model
```

3.3 Hyperparameter Variants

This dictionary defines different model configurations for experimentation. Each variant includes key hyperparameters for building and training a specific model type.

3.3.1 Structure

Each entry includes: - `type`: Model architecture to use (`"cnn"` or `"mobilenet"`) - `conv_blocks`: Number of convolution blocks (only for CNNs) - `dense_units`: Units in the dense layer - `learning_rate`: Learning rate for the optimizer - `epochs`: Number of training epochs

3.3.2 Defined Variants

- **A_small**: A lightweight CNN with 2 conv blocks and 8 epochs
- **B_medium**: A lightweight CNN with 3 conv blocks and 12 epochs
- **C_large**: A lightweight CNN with 4 conv blocks and 15 epochs
- **D_mobilenet_small**: A MobileNetV2-based model with a custom dense head and 8 epochs
- **E_mobilenet_medium**: A MobileNetV2-based model with a custom dense head and 12 epochs
- **F_mobilenet_large**: A MobileNetV2-based model with a custom dense head and 15 epochs

Other variants (e.g., `"B_medium"`, `"C_large"`) are included in comments and can be activated for further testing.

[26]:
```
# 6) Hyperparameter-Varianten
VARIANTS = {
    "A_small": {"type": "cnn", "conv_blocks": 2, "dense_units": 64,␣
↪"learning_rate": 1e-3, "epochs": 8},
    "B_medium": {"type": "cnn", "conv_blocks": 3, "dense_units": 128,␣
↪"learning_rate": 5e-4, "epochs": 12},
    "C_large":  {"type": "cnn", "conv_blocks": 4, "dense_units": 128,␣
 ↪"learning_rate": 1e-4, "epochs": 15},
    "D_mobilenet_small": {"type": "mobilenet", "dense_units": 64,␣
 ↪"learning_rate": 1e-3, "epochs": 8},
    "E_mobilenet_medium": {"type": "mobilenet", "dense_units": 128,␣
 ↪"learning_rate": 5e-4, "epochs": 12},
    "F_mobilenet_large": {"type": "mobilenet", "dense_units": 128,␣
 ↪"learning_rate": 1e-4, "epochs": 15}
}

histories, models = {}, {}
```

3.4 Training Loop for Variants

This loop iterates over the defined model variants in `VARIANTS` and trains each one based on its configuration.

3.4.1 Process Overview

- For each variant:
  - Print the variant name
  - Select the appropriate model-building function:
    - `build_mobilenetv2` for MobileNet-based models
    - `build_cnn` for CNN-based models
  - Pass hyperparameters such as `dense_units`, `conv_blocks`, and `learning_rate`
  - Train the model using `model.fit` with specified `epochs`
  - Store the training `history` and the trained `model` in dictionaries for later use

3.4.2 Outputs

- `histories`: Stores the training history for each variant
- `models`: Stores the trained model instances

[27]:
```
for name, params in VARIANTS.items():
    print(f"\n=== Training variant {name} ===")

    if params["type"] == "mobilenet":
        model = build_mobilenetv2(
            dense_units=params["dense_units"],
            learning_rate=params["learning_rate"]
        )
    else:
        model = build_cnn(
            conv_blocks=params["conv_blocks"],
            dense_units=params["dense_units"],
            learning_rate=params["learning_rate"]
        )
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=params["epochs"],
        verbose=2,
    )
    histories[name] = history
    models[name] = model
```

```
=== Training variant A_small ===
Epoch 1/8
313/313 - 37s - 118ms/step - accuracy: 0.9167 - loss: 0.2444 - val_accuracy:
0.9750 - val_loss: 0.0875
Epoch 2/8
313/313 - 37s - 117ms/step - accuracy: 0.9566 - loss: 0.1166 - val_accuracy:
0.9837 - val_loss: 0.0592
Epoch 3/8
313/313 - 37s - 118ms/step - accuracy: 0.9677 - loss: 0.0888 - val_accuracy:
0.9900 - val_loss: 0.0287
Epoch 4/8
313/313 - 36s - 115ms/step - accuracy: 0.9752 - loss: 0.0706 - val_accuracy:
0.9875 - val_loss: 0.0314
Epoch 5/8
313/313 - 36s - 116ms/step - accuracy: 0.9770 - loss: 0.0634 - val_accuracy:
0.9837 - val_loss: 0.0390
Epoch 6/8
313/313 - 36s - 114ms/step - accuracy: 0.9774 - loss: 0.0634 - val_accuracy:
0.9900 - val_loss: 0.0236
Epoch 7/8
313/313 - 36s - 115ms/step - accuracy: 0.9801 - loss: 0.0543 - val_accuracy:
0.9925 - val_loss: 0.0166
Epoch 8/8
313/313 - 36s - 115ms/step - accuracy: 0.9812 - loss: 0.0530 - val_accuracy:
0.9925 - val_loss: 0.0230

=== Training variant B_medium ===
Epoch 1/12
313/313 - 46s - 148ms/step - accuracy: 0.9278 - loss: 0.1833 - val_accuracy:
0.9750 - val_loss: 0.0802
Epoch 2/12
313/313 - 44s - 141ms/step - accuracy: 0.9708 - loss: 0.0787 - val_accuracy:
0.9850 - val_loss: 0.0359
Epoch 3/12
313/313 - 44s - 141ms/step - accuracy: 0.9749 - loss: 0.0644 - val_accuracy:
0.9900 - val_loss: 0.0297
Epoch 4/12
313/313 - 44s - 141ms/step - accuracy: 0.9810 - loss: 0.0572 - val_accuracy:
0.9812 - val_loss: 0.0550
```

```
Epoch 5/12
313/313 - 45s - 144ms/step - accuracy: 0.9819 - loss: 0.0508 - val_accuracy:
0.9925 - val_loss: 0.0265
Epoch 6/12
313/313 - 44s - 141ms/step - accuracy: 0.9830 - loss: 0.0454 - val_accuracy:
0.9912 - val_loss: 0.0283
Epoch 7/12
313/313 - 45s - 144ms/step - accuracy: 0.9852 - loss: 0.0436 - val_accuracy:
0.9950 - val_loss: 0.0154
Epoch 8/12
313/313 - 44s - 141ms/step - accuracy: 0.9848 - loss: 0.0395 - val_accuracy:
0.9937 - val_loss: 0.0195
Epoch 9/12
313/313 - 44s - 141ms/step - accuracy: 0.9877 - loss: 0.0353 - val_accuracy:
0.9962 - val_loss: 0.0170
Epoch 10/12
313/313 - 44s - 140ms/step - accuracy: 0.9889 - loss: 0.0321 - val_accuracy:
0.9925 - val_loss: 0.0150
Epoch 11/12
313/313 - 44s - 139ms/step - accuracy: 0.9886 - loss: 0.0293 - val_accuracy:
0.9950 - val_loss: 0.0231
Epoch 12/12
313/313 - 44s - 139ms/step - accuracy: 0.9899 - loss: 0.0300 - val_accuracy:
0.9950 - val_loss: 0.0220

=== Training variant C_large ===
Epoch 1/15
313/313 - 49s - 156ms/step - accuracy: 0.9039 - loss: 0.2338 - val_accuracy:
0.9712 - val_loss: 0.0895
Epoch 2/15
313/313 - 47s - 149ms/step - accuracy: 0.9614 - loss: 0.1102 - val_accuracy:
0.9812 - val_loss: 0.0647
Epoch 3/15
313/313 - 46s - 148ms/step - accuracy: 0.9731 - loss: 0.0805 - val_accuracy:
0.9825 - val_loss: 0.0519
Epoch 4/15
313/313 - 46s - 148ms/step - accuracy: 0.9764 - loss: 0.0670 - val_accuracy:
0.9912 - val_loss: 0.0329
Epoch 5/15
313/313 - 46s - 148ms/step - accuracy: 0.9813 - loss: 0.0513 - val_accuracy:
0.9912 - val_loss: 0.0249
Epoch 6/15
313/313 - 46s - 148ms/step - accuracy: 0.9827 - loss: 0.0522 - val_accuracy:
0.9912 - val_loss: 0.0238
Epoch 7/15
313/313 - 47s - 149ms/step - accuracy: 0.9832 - loss: 0.0484 - val_accuracy:
0.9937 - val_loss: 0.0236
Epoch 8/15
```

```
313/313 - 46s - 148ms/step - accuracy: 0.9841 - loss: 0.0426 - val_accuracy:
0.9950 - val_loss: 0.0198
Epoch 9/15
313/313 - 46s - 148ms/step - accuracy: 0.9855 - loss: 0.0424 - val_accuracy:
0.9950 - val_loss: 0.0188
Epoch 10/15
313/313 - 46s - 148ms/step - accuracy: 0.9868 - loss: 0.0387 - val_accuracy:
0.9962 - val_loss: 0.0187
Epoch 11/15
313/313 - 46s - 148ms/step - accuracy: 0.9872 - loss: 0.0344 - val_accuracy:
0.9912 - val_loss: 0.0202
Epoch 12/15
313/313 - 47s - 149ms/step - accuracy: 0.9865 - loss: 0.0360 - val_accuracy:
0.9900 - val_loss: 0.0295
Epoch 13/15
313/313 - 47s - 149ms/step - accuracy: 0.9879 - loss: 0.0340 - val_accuracy:
0.9987 - val_loss: 0.0152
Epoch 14/15
313/313 - 46s - 147ms/step - accuracy: 0.9892 - loss: 0.0308 - val_accuracy:
0.9950 - val_loss: 0.0176
Epoch 15/15
313/313 - 46s - 148ms/step - accuracy: 0.9897 - loss: 0.0267 - val_accuracy:
0.9937 - val_loss: 0.0199

=== Training variant D_mobilenet_small ===
Epoch 1/8
313/313 - 33s - 106ms/step - accuracy: 0.9810 - loss: 0.0504 - val_accuracy:
0.9987 - val_loss: 0.0049
Epoch 2/8
313/313 - 29s - 94ms/step - accuracy: 0.9914 - loss: 0.0220 - val_accuracy:
0.9987 - val_loss: 0.0041
Epoch 3/8
313/313 - 30s - 96ms/step - accuracy: 0.9934 - loss: 0.0208 - val_accuracy:
0.9975 - val_loss: 0.0069
Epoch 4/8
313/313 - 29s - 94ms/step - accuracy: 0.9942 - loss: 0.0168 - val_accuracy:
0.9962 - val_loss: 0.0092
Epoch 5/8
313/313 - 30s - 94ms/step - accuracy: 0.9930 - loss: 0.0204 - val_accuracy:
1.0000 - val_loss: 0.0021
Epoch 6/8
313/313 - 30s - 97ms/step - accuracy: 0.9956 - loss: 0.0119 - val_accuracy:
1.0000 - val_loss: 0.0014
Epoch 7/8
313/313 - 30s - 97ms/step - accuracy: 0.9962 - loss: 0.0112 - val_accuracy:
1.0000 - val_loss: 0.0012
Epoch 8/8
313/313 - 29s - 93ms/step - accuracy: 0.9959 - loss: 0.0116 - val_accuracy:
```

```
1.0000 - val_loss: 5.6703e-04

=== Training variant E_mobilenet_medium ===
Epoch 1/12
313/313 - 34s - 109ms/step - accuracy: 0.9795 - loss: 0.0507 - val_accuracy:
0.9962 - val_loss: 0.0154
Epoch 2/12
313/313 - 29s - 93ms/step - accuracy: 0.9926 - loss: 0.0228 - val_accuracy:
0.9975 - val_loss: 0.0066
Epoch 3/12
313/313 - 29s - 93ms/step - accuracy: 0.9925 - loss: 0.0224 - val_accuracy:
0.9987 - val_loss: 0.0042
Epoch 4/12
313/313 - 30s - 94ms/step - accuracy: 0.9937 - loss: 0.0177 - val_accuracy:
0.9975 - val_loss: 0.0036
Epoch 5/12
313/313 - 29s - 93ms/step - accuracy: 0.9954 - loss: 0.0128 - val_accuracy:
0.9962 - val_loss: 0.0111
Epoch 6/12
313/313 - 29s - 93ms/step - accuracy: 0.9940 - loss: 0.0161 - val_accuracy:
1.0000 - val_loss: 8.7356e-04
Epoch 7/12
313/313 - 29s - 93ms/step - accuracy: 0.9965 - loss: 0.0102 - val_accuracy:
1.0000 - val_loss: 9.4687e-04
Epoch 8/12
313/313 - 29s - 93ms/step - accuracy: 0.9968 - loss: 0.0095 - val_accuracy:
1.0000 - val_loss: 0.0011
Epoch 9/12
313/313 - 29s - 93ms/step - accuracy: 0.9960 - loss: 0.0127 - val_accuracy:
1.0000 - val_loss: 0.0011
Epoch 10/12
313/313 - 29s - 94ms/step - accuracy: 0.9978 - loss: 0.0079 - val_accuracy:
1.0000 - val_loss: 0.0018
Epoch 11/12
313/313 - 29s - 94ms/step - accuracy: 0.9961 - loss: 0.0120 - val_accuracy:
0.9987 - val_loss: 0.0056
Epoch 12/12
313/313 - 29s - 93ms/step - accuracy: 0.9967 - loss: 0.0096 - val_accuracy:
0.9987 - val_loss: 0.0022

=== Training variant F_mobilenet_large ===
Epoch 1/15
313/313 - 34s - 108ms/step - accuracy: 0.9677 - loss: 0.0884 - val_accuracy:
0.9937 - val_loss: 0.0208
Epoch 2/15
313/313 - 29s - 92ms/step - accuracy: 0.9904 - loss: 0.0272 - val_accuracy:
1.0000 - val_loss: 0.0083
Epoch 3/15
```

```
313/313 - 29s - 93ms/step - accuracy: 0.9934 - loss: 0.0204 - val_accuracy:
1.0000 - val_loss: 0.0072
Epoch 4/15
313/313 - 29s - 93ms/step - accuracy: 0.9929 - loss: 0.0197 - val_accuracy:
0.9987 - val_loss: 0.0051
Epoch 5/15
313/313 - 29s - 93ms/step - accuracy: 0.9948 - loss: 0.0152 - val_accuracy:
0.9987 - val_loss: 0.0055
Epoch 6/15
313/313 - 29s - 94ms/step - accuracy: 0.9948 - loss: 0.0160 - val_accuracy:
0.9975 - val_loss: 0.0062
Epoch 7/15
313/313 - 29s - 94ms/step - accuracy: 0.9955 - loss: 0.0141 - val_accuracy:
0.9975 - val_loss: 0.0064
Epoch 8/15
313/313 - 30s - 96ms/step - accuracy: 0.9962 - loss: 0.0102 - val_accuracy:
1.0000 - val_loss: 0.0036
Epoch 9/15
313/313 - 30s - 96ms/step - accuracy: 0.9952 - loss: 0.0130 - val_accuracy:
0.9950 - val_loss: 0.0066
Epoch 10/15
313/313 - 29s - 93ms/step - accuracy: 0.9963 - loss: 0.0126 - val_accuracy:
0.9987 - val_loss: 0.0037
Epoch 11/15
313/313 - 29s - 93ms/step - accuracy: 0.9971 - loss: 0.0089 - val_accuracy:
1.0000 - val_loss: 0.0026
Epoch 12/15
313/313 - 29s - 93ms/step - accuracy: 0.9971 - loss: 0.0089 - val_accuracy:
1.0000 - val_loss: 0.0021
Epoch 13/15
313/313 - 29s - 94ms/step - accuracy: 0.9961 - loss: 0.0105 - val_accuracy:
1.0000 - val_loss: 0.0023
Epoch 14/15
313/313 - 29s - 93ms/step - accuracy: 0.9973 - loss: 0.0085 - val_accuracy:
1.0000 - val_loss: 0.0012
Epoch 15/15
313/313 - 29s - 93ms/step - accuracy: 0.9969 - loss: 0.0088 - val_accuracy:
1.0000 - val_loss: 0.0014
```

3.5 Selecting the Best Model Variant

This code identifies the best-performing model variant based on validation accuracy.

3.5.1 Process

- Iterates through all training histories stored in `histories`
- Finds the variant with the highest recorded validation accuracy (`val_accuracy`)
- Retrieves the corresponding trained model from `models`

- Prints the name of the best variant

3.5.2 Outputs

- `best_name_cnn`: The name of the variant with the highest validation accuracy of the CNN models
- `best_model_cnn`: The trained model instance corresponding to `best_name_cnn`
- `best_name_mobilenet`: The name of the variant with the highest validation accuracy of the MobileNet models
- `best_model_mobilenet`: The trained model instance corresponding to `best_name_cnn`

[42]:
```
# 8) Beste Variante ermitteln
best_name_cnn = max(
    (name for name in histories if name[0].upper() in "ABC"),
    key=lambda n: max(histories[n].history["val_accuracy"])
)

best_name_mobilenet = max(
    (name for name in histories if name[0].upper() in "DEF"),
    key=lambda n: max(histories[n].history["val_accuracy"])
)

# Get the models
best_model_cnn = models[best_name_cnn]
best_model_mobilenet = models[best_name_mobilenet]

# Print the results
print("Beste Variante CNN:", best_name_cnn)
print("Beste Variante MobileNet:", best_name_mobilenet)
```

```
Beste Variante CNN: C_large
Beste Variante MobileNet: D_mobilenet_small
```

3.6 Model Evaluation Function

This function evaluates a trained model on a given dataset by computing detailed classification metrics.

3.6.1 Description

- Iterates over the dataset batches
- Collects true labels and model predictions
- Uses `classification_report` to print precision, recall, F1-score, and support for each class
- Prints the confusion matrix to show prediction accuracy across classes

3.6.2 Inputs

- `model`: The trained Keras model to evaluate
- `dataset`: A TensorFlow dataset containing images and labels for evaluation

3.6.3 Outputs

- Prints a detailed classification report
- Prints the confusion matrix for the dataset

[43]:
```
# 9) Evaluation
def evaluate(model, dataset):
    y_true, y_pred = [], []
    for imgs, labels in dataset:
        preds = model.predict(imgs, verbose=0)
        y_true.extend(labels.numpy())
        y_pred.extend(tf.argmax(preds, axis=1).numpy())
    print(classification_report(y_true, y_pred, target_names=class_names))
    print("Confusion Matrix:")
    print(confusion_matrix(y_true, y_pred))
```

3.6.4 CNN Model Evaluation

[44]:
```
evaluate(best_model_cnn, test_ds)
```

```
              precision    recall  f1-score   support

    WithMask       0.99      1.00      0.99       483
 WithoutMask       1.00      0.99      0.99       509

    accuracy                           0.99       992
   macro avg       0.99      0.99      0.99       992
weighted avg       0.99      0.99      0.99       992

Confusion Matrix:
[[481   2]
 [  7 502]]
```

3.6.5 MobileNet Model Evaluation

[45]:
```
evaluate(best_model_mobilenet, test_ds)
```

```
              precision    recall  f1-score   support

    WithMask       1.00      1.00      1.00       483
 WithoutMask       1.00      1.00      1.00       509

    accuracy                           1.00       992
   macro avg       1.00      1.00      1.00       992
weighted avg       1.00      1.00      1.00       992

Confusion Matrix:
[[481   2]
 [  1 508]]
```

3.7 Plotting Training and Validation Curves

This code visualizes model performance over epochs using accuracy and loss curves for all trained variants.

3.7.1 Description

- Plots training and validation accuracy curves in one figure:
  - Training accuracy with dashed lines
  - Validation accuracy with solid lines
- Plots training and validation loss curves in a separate figure:
  - Training loss with dashed lines
  - Validation loss with solid lines

3.7.2 Purpose

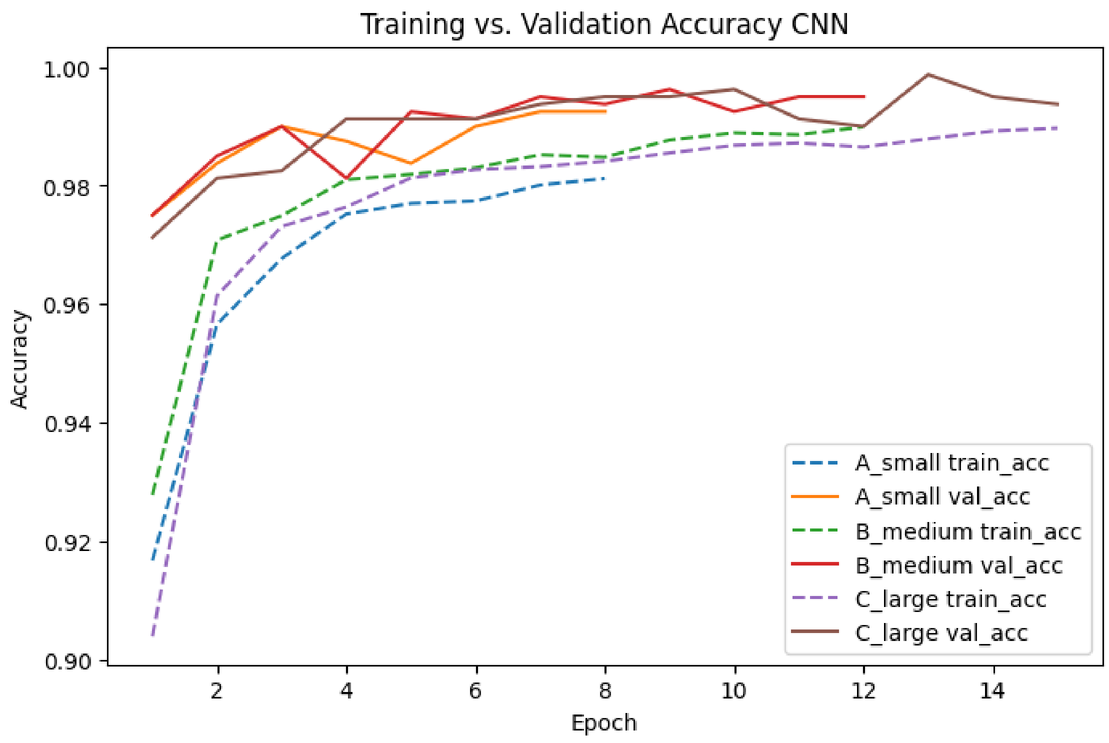
- Helps to monitor overfitting and underfitting
- Provides visual comparison between different model variants

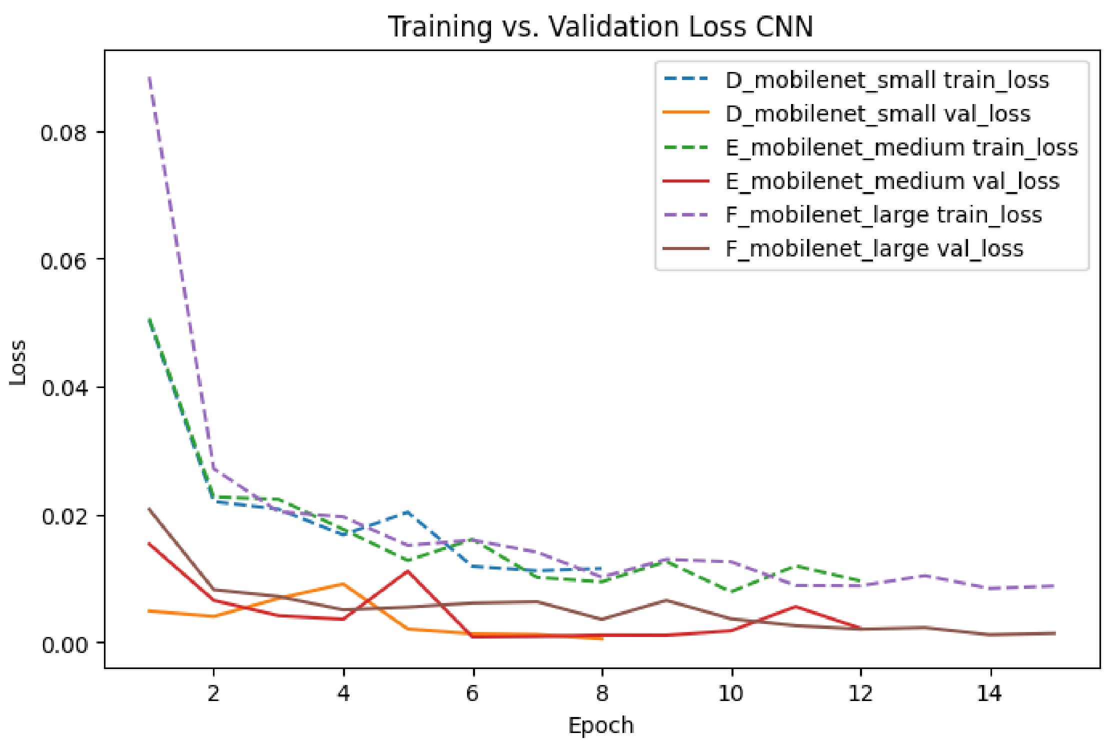
3.7.3 Visualization Details

- X-axis: Epoch number
- Y-axis: Accuracy or Loss values
- Includes legends to distinguish between training/validation and model variants

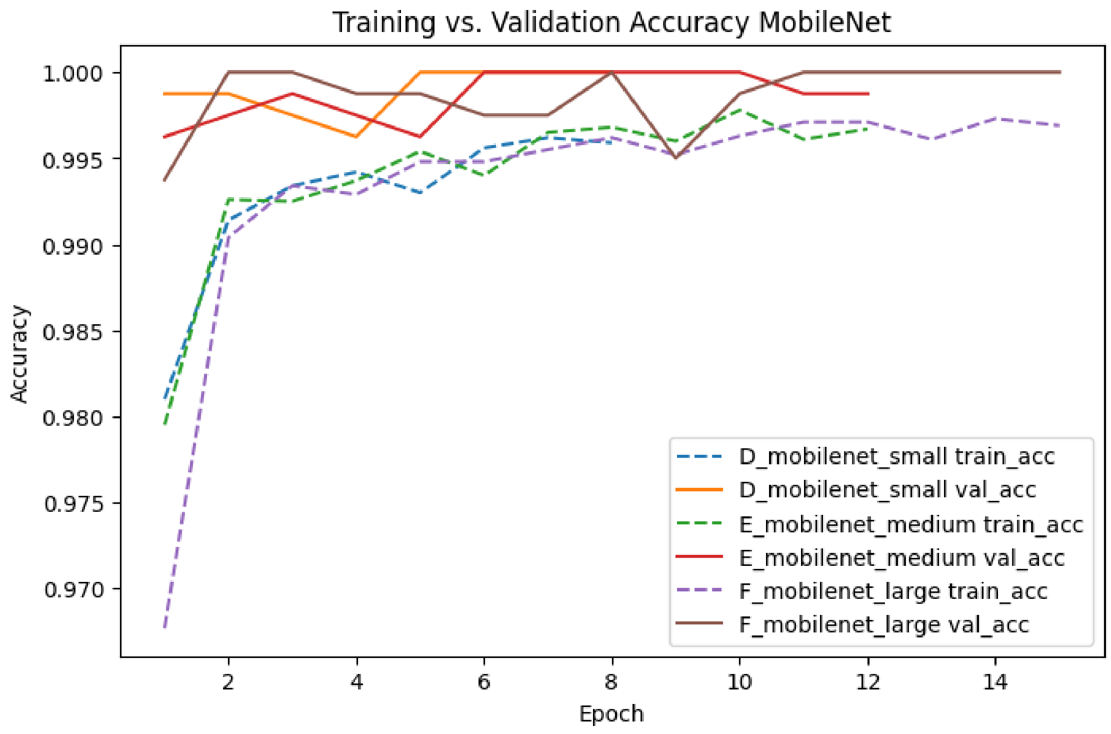
[46]:
```
# 10) Trainings- und Validierungskurven
import matplotlib.pyplot as plt

histories_cnn = {name: hist for name, hist in histories.items() if name[0].
  ↪upper() in "ABC"}
histories_mobilenet = {name: hist for name, hist in histories.items() if␣
  ↪name[0].upper() in "DEF"}
```

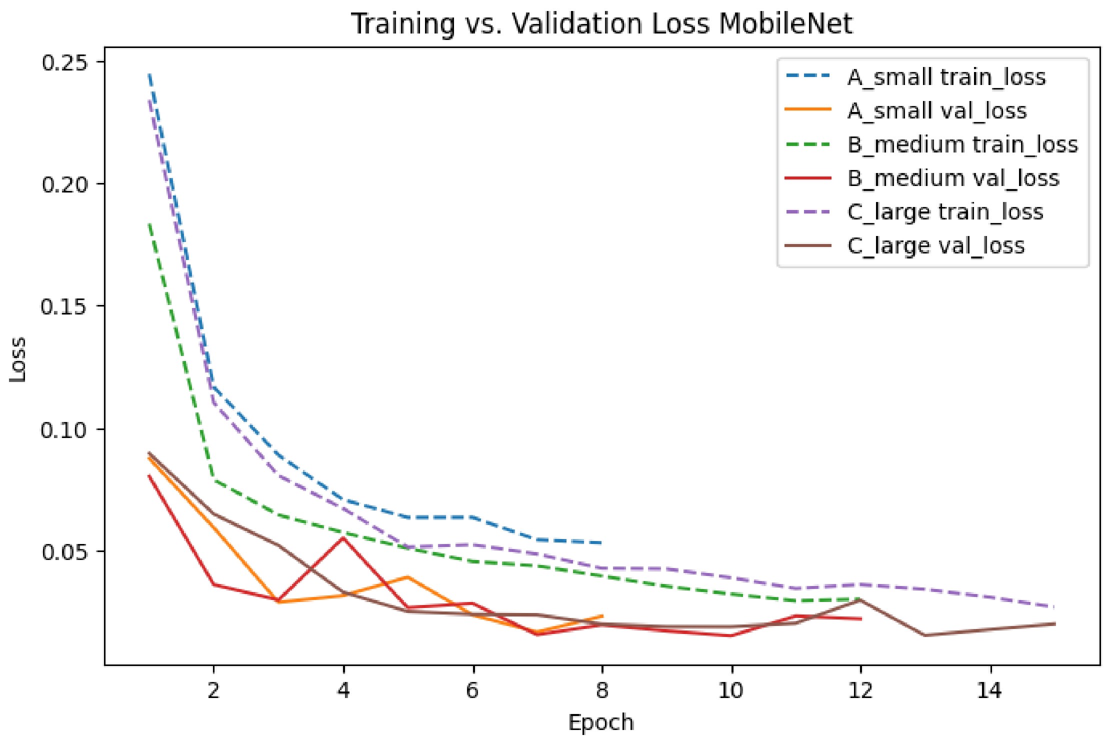
[47]:
```
plt.figure(figsize=(8, 5))
for name, hist in histories_cnn.items():
    epochs = range(1, len(hist.history["accuracy"]) + 1)

    # -- Accuracy --
    plt.plot(epochs,
             hist.history["accuracy"],
             linestyle="--",  # gestrichelt = Training
             label=f"{name} train_acc")

    plt.plot(epochs,
             hist.history["val_accuracy"],
             label=f"{name} val_acc")

plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Training vs. Validation Accuracy CNN")
plt.legend()
plt.show()

#  Loss-Kurven in einem separaten Plot
plt.figure(figsize=(8, 5))
for name, hist in histories_mobilenet.items():
    epochs = range(1, len(hist.history["loss"]) + 1)
    plt.plot(epochs, hist.history["loss"],
             linestyle="--",
             label=f"{name} train_loss")
    plt.plot(epochs, hist.history["val_loss"],
             label=f"{name} val_loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training vs. Validation Loss CNN")
plt.legend()
plt.show()
```





[48]:
```python
plt.figure(figsize=(8, 5))
for name, hist in histories_mobilenet.items():
    epochs = range(1, len(hist.history["accuracy"]) + 1)

    # -- Accuracy --
    plt.plot(epochs,
             hist.history["accuracy"],
             linestyle="--",  # gestrichelt = Training
             label=f"{name} train_acc")

    plt.plot(epochs,
             hist.history["val_accuracy"],
             label=f"{name} val_acc")

plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Training vs. Validation Accuracy MobileNet")
plt.legend()
plt.show()

#  Loss-Kurven in einem separaten Plot
plt.figure(figsize=(8, 5))
for name, hist in histories_cnn.items():
    epochs = range(1, len(hist.history["loss"]) + 1)
    plt.plot(epochs, hist.history["loss"],
             linestyle="--",
             label=f"{name} train_loss")
    plt.plot(epochs, hist.history["val_loss"],
             label=f"{name} val_loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training vs. Validation Loss MobileNet")
plt.legend()
plt.show()
```




Training vs. Validation Loss MobileNet
Loss
Epoch
0.25
0.20
0.15
0.10
0.05
2
4
6
8
10
12
14
A_small train_loss
A_small val_loss
B_medium train_loss
B_medium val_loss
C_large train_loss
C_large val_loss