Deep Learning project: Face Recognition

The following notebook is about training a face recognition model with deep learning methods. For this purpose, an own dataset with face images is to be created. With these images different models should be trained and finally compared. The project is part of the Deep Learning course at the University of Heilbronn and serves educational purposes.

Author:

Dominik Bücher, Hochschuhle Heilbronn, Automotive System Engineering Master

dbuecher@stud.hs-heilbronn.de

Professor:

Prof. Dr.-Ing. Nicolaj Stache

nicolaj.stache@hs-heilbronn.de

Table of Contents

---

1. Introduction

Before diving into the project's implementation details, let's start with a brief introduction. The objective of this project, as mentioned earlier, is to train a Deep Learning Model capable of recognizing and labeling faces. It is particularly crucial for the model to identify its own creator's face, namely Dominik Bücher, and correctly label it. Any other faces detected should be labeled as "Unknown." To accomplish this, transfer learning is utilized during the training process.

Transfer learning involves retraining a pre-existing model on new data. In this case, a model that has previously been trained on a vast number of images is retrained on a smaller dataset specific to this project. The key advantage of transfer learning is its ability to achieve impressive performance on novel tasks with limited data and computational resources. By leveraging the knowledge acquired during prior training, the model can learn more swiftly and efficiently.

This notebook primarily focuses on training and testing personalized models, offering invaluable hands-on experience in the field. As a result, instead of training a single model, the aim in this project is to train four distinct models. All of these models will utilize transfer learning, although each will have a different foundational model on which transfer learning is applied.

As can be seen from the Table of Contents, the dataset on which the models are trained is presented at the beginning of the project. Subsequently, this dataset is imported and a data augmentation is performed on the images.

---

2. Import of Librarys

In this chapter, the required libraries are first imported, which are important for the use of the notebook. If a library cannot be found, it must be installed with pip (this can be done with this command "!pip install ..." plus the required library). This is crucial, because otherwise the notebook cannot be executed completely.

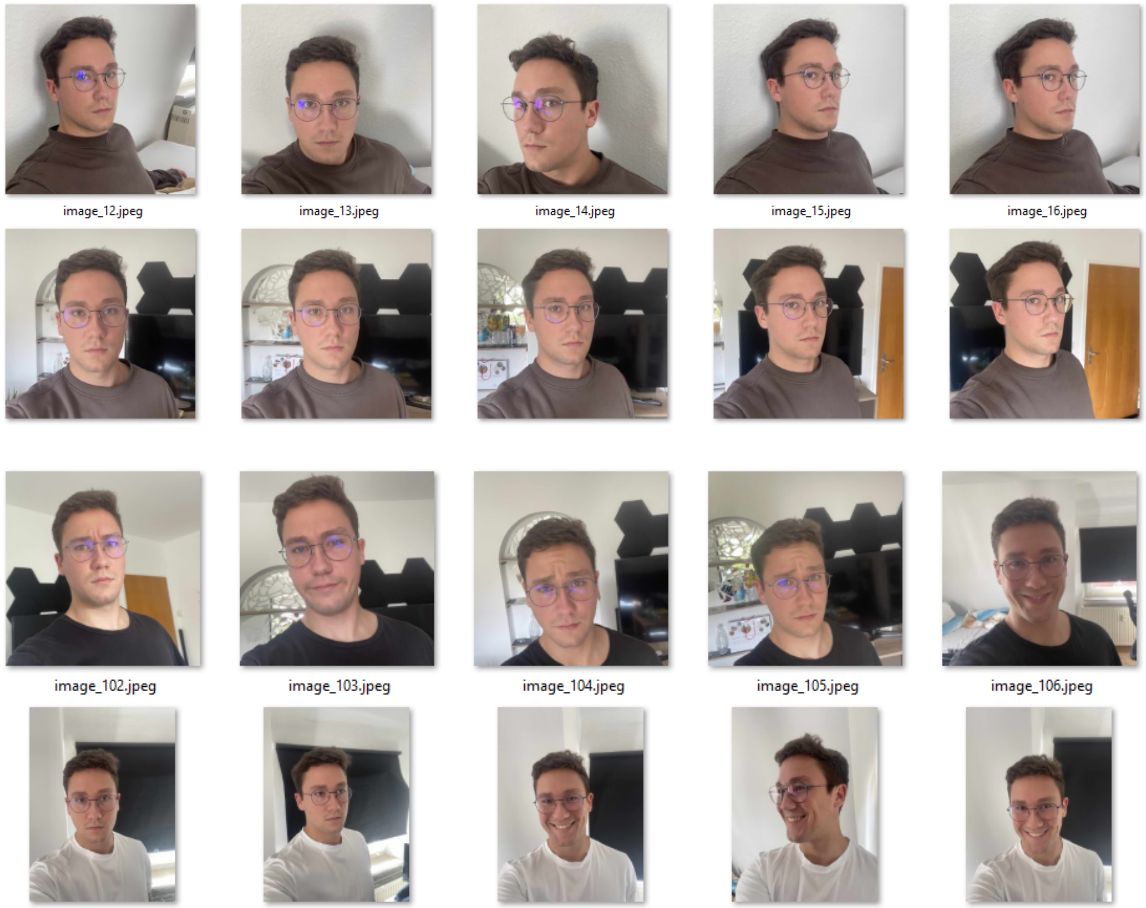
In [ ]:
```
# import the necessary packages
import os
import zipfile
import matplotlib.pyplot as plt
import cv2
import glob
import tensorflow as tf

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dropout
```

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical
from keras.preprocessing.image import img_to_array, array_to_img
from tensorflow.keras.applications import ResNet50, ResNet101, ResNet152, Inception
import numpy as np
from PIL import Image
from PIL import Image, ImageOps
from keras.models import load_model
```

---

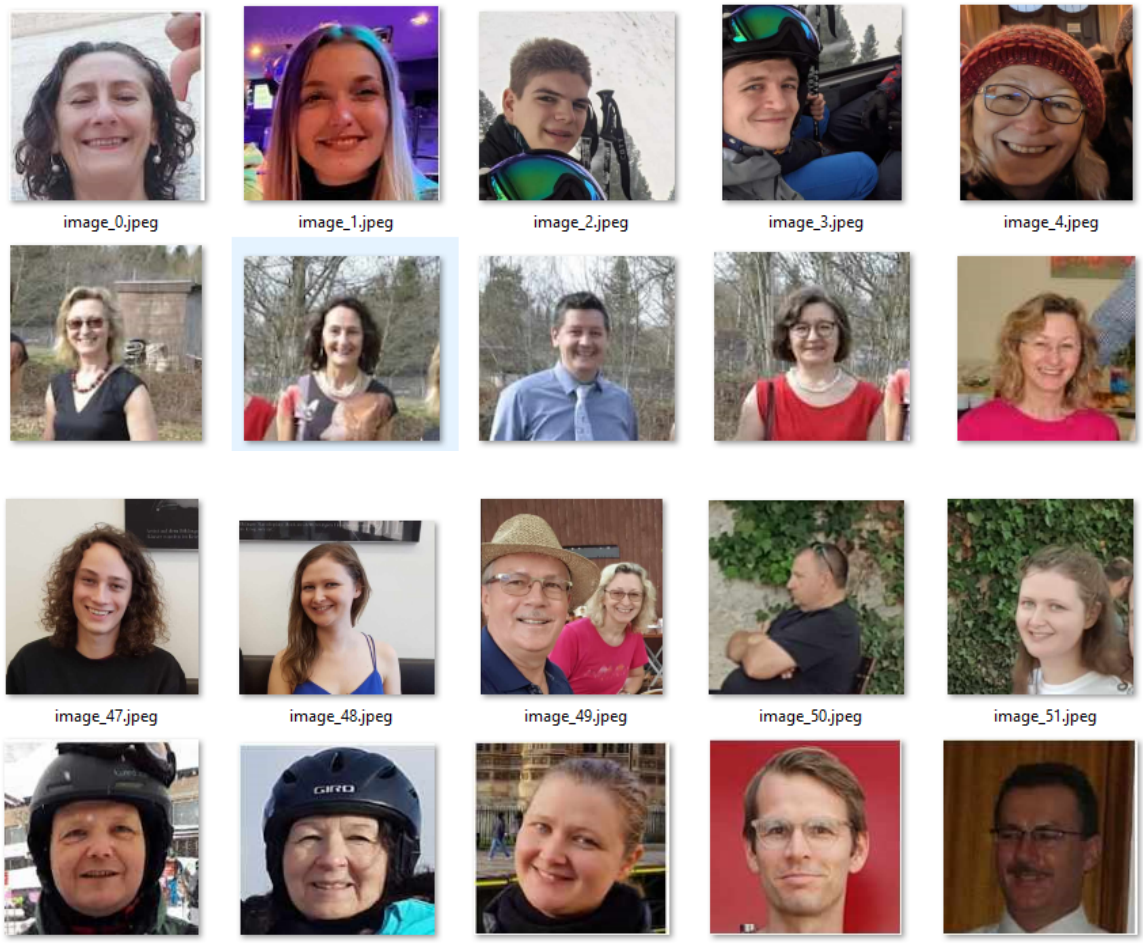
3. Presenting the dataset

- As mentioned in the introduction, this chapter introduces the dataset that will be used later to train the models. Since the dataset is very crucial for the training, it is described here in as much detail as possible.
- The dataset used is a self-created dataset which was created for this project. Since the objective is to recognize the face of Dominik Bücher, the dataset only requires two classes. One class contains images of Dominik, while the other class serves as a reference and is labeled as "Unknown". The "Unknown" class primarily ensures that the models are not trained in a biased manner and that a reference point is available. The images in the class consist of friends and family members of Dominik.
- Specifically, a deliberate effort was made to include a diverse range of individuals in the "Unknown" class to prevent the model from overly focusing on a single additional face. By incorporating numerous different individuals within the "Unknown" class, the model is encouraged to develop a more generalized understanding of what constitutes an unknown face, rather than being biased towards specific individuals.
- The dataset comprises a total of 450 images, with 300 images belonging to the Dominik class and 150 images belonging to the Unknown class. The images used predominantly consist of facial images. Special attention was given to including a diverse range of facial expressions and angles of Dominik's face within the Dominik class. A small selection of example images from the Dominik class can be observed in the following figures:



- The images were carefully selected to include various angles of the face, ensuring a comprehensive representation. Furthermore, in order to introduce a certain level of variance, different outfits were used in the photographs. Additionally, lighting conditions and backgrounds were altered to add diversity to the dataset. These measures help to enhance the model's ability to generalize and perform well in real-world scenarios with varying conditions. The images also vary in terms of facial expressions to include images where various facial expressions, including grimaces, are captured. This allows the model to recognize and classify images with different facial expressions.

- Since the Unknown class serves as a reverence, it is not important to have many different images of one face, but many different images of other faces. It also happens that there is no face at all, so that the model can cope with such images. Below are a number of example images from the Unknown class:



Based on the examples of the images from the two classes, it is clear that the images from Dominik's class usually have a better quality or resolution. This is because the value of these images is clearly more important and care was taken to ensure that they have a reasonably satisfactory quality. Whereas the images of the Unknown class had no quality requirements.

- **Size of the dataset:**
  Coming back to the amount of images in the dataset, this was not chosen purely randomly. First of all, it should be noted that transfer learning does not require a large dataset, which is one of the advantages of this method. Therefore, it is clear why the dataset used does not contain thousands of images. However, it should not be too few, so that the models can learn the features of Dominik's face correctly. For this reason, the size of the dataset was experimented with in the beginning. Thereby it often came to problems if too few images were available for the training. This was due to the fact that the dataset is divided into training, validation and test data, leaving only a few images for training. Therefore, some epochs were canceled from time to time because there were too few images available. For this reason, and through some testing, it was determined that the dataset should contain at least a few hundred images so that sufficient images are also available for validation and testing. Therefore, since the dataset should still remain as small as possible, 300 images were chosen for the Dominik class. For the Unknown class only 150 images were chosen, since these images are not so important and it could be saved thus somewhat at storage place than if again 300

images would be used.

- **Quality of the images:**
  The images in the Dominik class were captured using an iPhone 12, ensuring a high-quality output. However, this also means that the 300 images consume significantly more memory than the maximum available 300 MB. Consequently, a decision was made to reduce the image quality by approximately 50%. This reduction decreased the data size from around 1.65 MB per image to 350 KB. Despite this adjustment, the image quality remains sufficiently good for training the models, as evident in the provided examples. Therefore, the initial high quality plays only a partial role for the training.

Dataset structure:
This section covers the dataset's structure, starting with an explanation of the initial folder structure, followed by the folder structure utilized after loading the dataset.

As depicted in the following figure, the dataset is divided into two folders, representing the distinct classes, with each folder aptly named accordingly. Within each class folder, only images belonging to that particular class are contained. By organizing the dataset in this manner, it becomes easier to navigate and access the images based on their respective classes.

Initial folder structure:

```
- face_dataset
    |- Dominik
        |- image_0.jpeg
        |- image_1.jpeg
        |- image_2.jpeg
        |- ...
    |- Unknown
        |- image_0.jpeg
        |- image_1.jpeg
        |- image_2.jpeg
        |- ...
```

The goal is to store the dataset in a structure as depicted below. There will be three top-level folders containing images for training, validation, and testing, respectively. Within these three folders, there will always be two subfolders representing the two classes.

It is worth mentioning that additional images will be augmented later on, and these augmented images will also be incorporated into the same folder structure.

Folder structure for further processing:

```
- data
    |-train
        |- Dominik
```

```
                |- image_0.jpeg
                |- image_1.jpeg
                |- image_2.jpeg
                |- ...
            |- Unknown
                |- image_0.jpeg
                |- image_1.jpeg
                |- image_2.jpeg
                |- ...
        |-validation
            |- Dominik
                |- image_0.jpeg
                |- image_1.jpeg
                |- image_2.jpeg
                |- ...
            |- Unknown
                |- image_0.jpeg
                |- image_1.jpeg
                |- image_2.jpeg
                |- ...
        |-test
            |- Dominik
                |- image_0.jpeg
                |- image_1.jpeg
                |- image_2.jpeg
                |- ...
            |- Unknown
                |- image_0.jpeg
                |- image_1.jpeg
                |- image_2.jpeg
                |- ...
```

---

4. Import the dataset

Now that the general procedure and the dataset have been presented, we can start with the implementation. For this, the dataset is first loaded from a ZIP file into a temporary directory so that it can be processed later.

To accomplish this, you need to specify the path to the ZIP file and then combine it with the folder path. Once the dataset is stored in the temporary directory, you can easily access the two folders that contain the images for the two classes. Subsequently, the paths of these images are extracted and compiled into a list. These paths can then be utilized to conveniently access the corresponding images.

In [ ]:
```
################ Unzip the dataset in the Colab runtime ##################

# Specify the path to the ZIP file containing the face dataset
zip_file_path = 'data/face_dataset.zip'
```

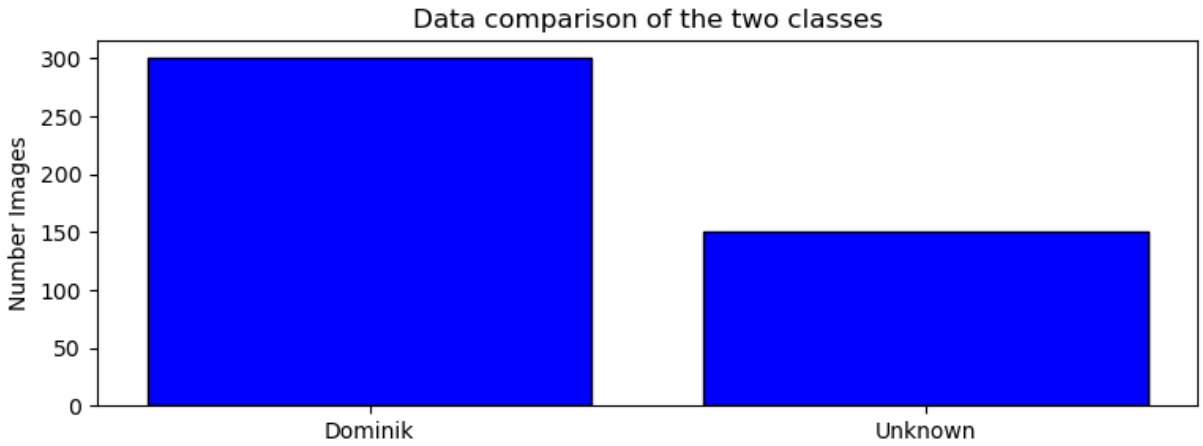
```
# Combine the path with the ZIP file
path_to_zip = os.path.join(zip_file_path)

# Print a message indicating the file being unzipped
print("Unzipping {}".format(path_to_zip))

# Specify the path where the extracted data will be stored
path_to_data = "/tmp"

# Open the ZIP file and extract its contents to the specified path
with zipfile.ZipFile(path_to_zip, 'r') as zip_ref:
    zip_ref.extractall(path_to_data)

# Retrieve the paths of the images for 'Dominik' and 'Unknown' categories
data_dominik = glob.glob(os.path.join(path_to_data, "face_dataset/Dominik/" + "*.jp
data_unknown = glob.glob(os.path.join(path_to_data, "face_dataset/Unknown/" + "*.jp

# Print the number of image paths found for 'Dominik' and 'Unknown' categories
print("Dominik: {} image paths".format(len(data_dominik)))
print("Unknown: {} image paths".format(len(data_unknown)))
```

```
Unzipping data/face_dataset.zip
Dominik: 300 image paths
Unknown: 150 image paths
```

Based on the output of the previous block, it is evident that all 450 images have been successfully located. However, if this is not the case and there are any errors, it is crucial to address and rectify them before proceeding with any further actions.

The number of found images of the two classes are visually represented in the following with this representation. This representation will be used several times in the following to show the development of the dataset after the various preprocessing steps.

In [ ]:
```
# Create a list of class names
names = ['Dominik', 'Unknown']

# Create a list of values representing the number of images in each class
values = [len(data_dominik), len(data_unknown)]

plt.figure(figsize=(9, 3)) # Set up the figure size for the bar chart
plt.bar(names, values, color='blue', edgecolor='black') # Create a bar chart with t
plt.ylabel('Number Images') # Set the label for the y-axis
plt.title('Data comparison of the two classes') # Set the title of the chart
plt.show() # Display the chart
```



In the following code section the images are now loaded with the PIL library as PIL images into the runtime. After this step it is possible to edit the images as desired.

As mentioned before, the images have to be resized again, this is also done in this block. For this the images are scaled down to the format of 224x224 pixels. The 224x224 pixels are a standard size on which most models are trained. Therefore, this format is also chosen for the images in this project.

After these operations, two arrays containing images of the respective class are obtained, and these images are in the PIL image format, each having a dimension of 224x224 pixels.

In [ ]:
```
### Resizing the image of both classes
# Create an empty list to store the resized images for the 'Dominik' class
images_dominik = []

# Iterate over each JPEG file in the '/tmp/face_dataset/Dominik/' directory
for filename in glob.glob("/tmp/face_dataset/Dominik/" + "*.jpeg"):

    # Open the image file using PIL's Image module
    im = Image.open(filename)
    # Perform any necessary image orientation adjustment using ImageOps.exif_transp
    # This is need because the image are sometimes roted while saving
    im = ImageOps.exif_transpose(im)
    # Resize the image to a target size of 224x224 pixels
    resized_im = im.resize([224, 224])
    # Append the resized image to the 'images_dominik' list
    images_dominik.append(resized_im)

# Print the number of resized images for the 'Dominik' class
print("Dominik list:", len(images_dominik))
# Print the first resized image in the 'images_dominik' list
print(images_dominik[0])


# Create an empty list to store the resized images for the 'Unknown' class
images_unknown = []

# Iterate over each JPEG file in the '/tmp/face_dataset/Unknown/' directory
for filename in glob.glob("/tmp/face_dataset/Unknown/" + "*.jpeg"):
```

```
    # Open the image file using PIL's Image module
    im = Image.open(filename)
    # Perform any necessary image orientation adjustment using ImageOps.exif_transp
    # This is need because the image are sometimes roted while saving
    im = ImageOps.exif_transpose(im)
    # Resize the image to a target size of 224x224 pixels
    resized_im = im.resize([224, 224])
    # Append the resized image to the 'images_unknown' list
    images_unknown.append(resized_im)


# Print the number of resized images for the 'Unknown' class
print("Unknown list:", len(images_unknown))
# Print the first resized image in the 'images_unknown' list
print(images_unknown[0])
```

```
Dominik list: 300
<PIL.Image.Image image mode=RGB size=224x224 at 0x184C7BC80D0>
Unknown list: 150
<PIL.Image.Image image mode=RGB size=224x224 at 0x184C80D8AF0>
```

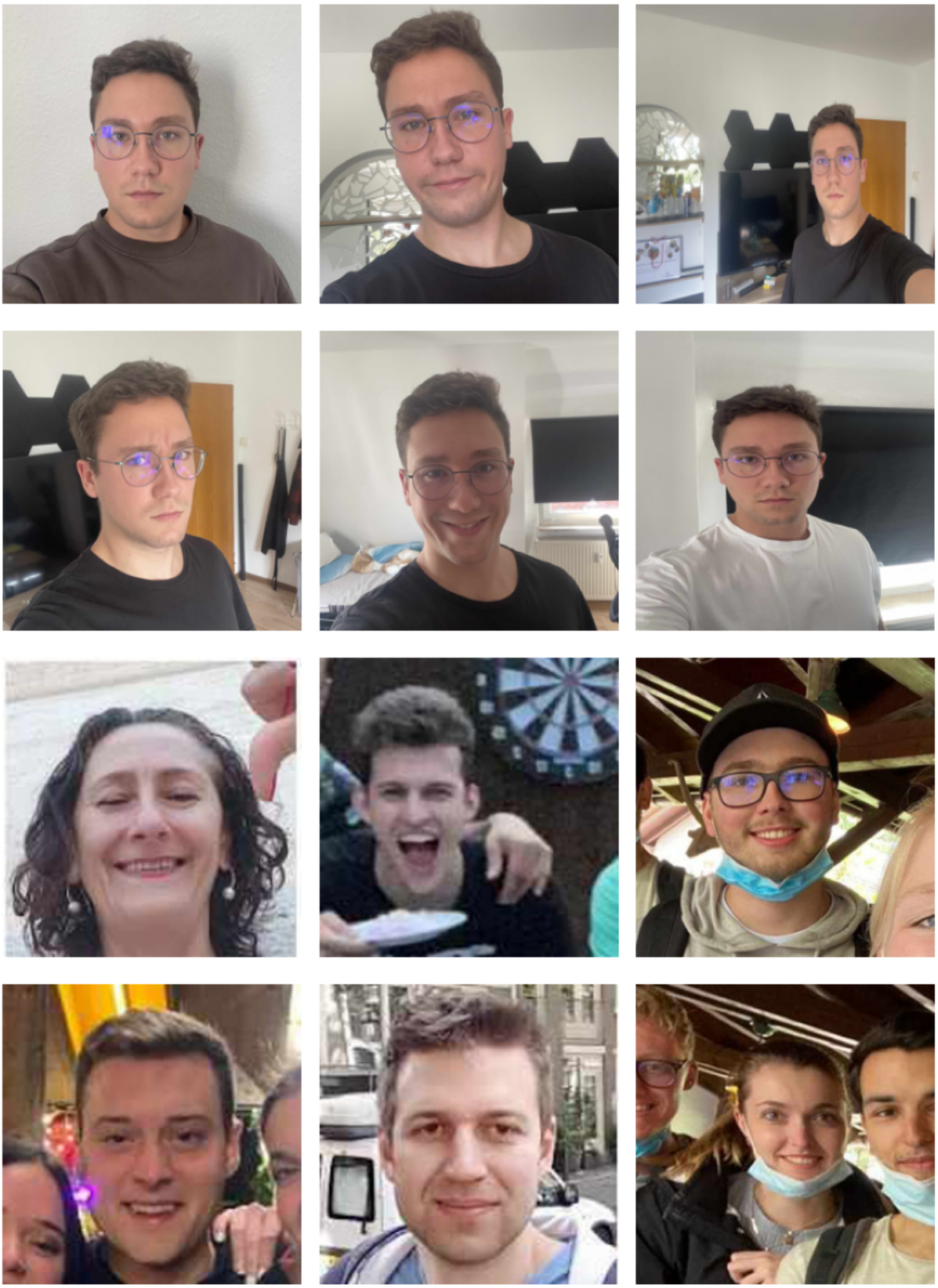
Show examples of the imported images:

Now that the data is in the PIL format, we can proceed to output it. The following two code blocks will display a selection of sample images from the imported data. The first block will show images from the "Dominik" class, while the second block will display images from the "Unknown" class.

In [ ]:
```
def printImages(images):
    for i in range(0,6,3):
        # Create a figure and axes
        fig, ax = plt.subplots(1,3, figsize=(8, 4))

        # Display the image
        ax[0].imshow(images[i])
        ax[1].imshow(images[i+5+1])
        ax[2].imshow(images[i+10+2])

        # Add a legend
        legend_text = ""
        ax[0].text(10, 10, legend_text, color='white', backgroundcolor='black')
        ax[1].text(10, 10, legend_text, color='white', backgroundcolor='black')
        ax[2].text(10, 10, legend_text, color='white', backgroundcolor='black')
        # ax[2].text(10, 10, legend_text, color='white', backgroundcolor='black')

        # Remove the axis labels
        ax[0].axis("off")
        ax[1].axis("off")
        ax[2].axis("off")

        # Adjust the layout
        plt.tight_layout()

        # Show the plot
```

```
        plt.show()
```

In [ ]:
```
printImages(images_dominik)
printImages(images_unknown)
```



Normalize images:

In this step the images are normalized and written into a new array. The purpose of normalizing images before training is to ensure consistency and improve the effectiveness of the training process. Normalization involves transforming pixel values to a standardized range. The benefits of image normalization include maintaining a consistent scale, faster convergence during training, addressing feature correlation, and promoting generalization to unseen data.

Alternatively, it would have been possible to perform the image normalization step directly after resizing the images. However, it was intentionally deferred to a later stage in order to provide clarity and emphasize the individual steps involved in the data processing pipeline.

For this step, the images are first converted into arrays, then normalized, and finally written back as PIL images into a list.

In [ ]:
```
# Create an empty list to store the normalized images for the 'Dominik' class
images_dominik_norm = []

# Iterate over each resized image in the 'images_dominik' list
for i in range(len(images_dominik)):
    # Convert the resized image to an array
    temp_data = img_to_array(images_dominik[i])
    # Normalize the image array using cv2.normalize, scaling the values between 0 a
    img_norm = cv2.normalize(temp_data, None, alpha=0, beta=1, norm_type=cv2.NORM_M
    # Convert the normalized array back to an image and append it to the 'images_do
    images_dominik_norm.append(array_to_img(img_norm))

# Print the number of normalized images for the 'Dominik' class
print("Dominik normalized list:", len(images_dominik_norm))
# Print the first normalized image in the 'images_dominik_norm' list
print(images_dominik_norm[0])


# Create an empty list to store the normalized images for the 'Unknown' class
images_unknown_norm = []

# Iterate over each resized image in the 'images_unknown' list
for i in range(len(images_unknown)):
    # Convert the resized image to an array
    temp_data = img_to_array(images_unknown[i])
    # Normalize the image array using cv2.normalize, scaling the values between 0 a
    img_norm = cv2.normalize(temp_data, None, alpha=0, beta=1, norm_type=cv2.NORM_M
    # Convert the normalized array back to an image and append it to the 'images_un
    images_unknown_norm.append(array_to_img(img_norm))

# Print the number of normalized images for the 'Unknown' class
print("Unknown normalized list:", len(images_unknown_norm))
# Print the first normalized image in the 'images_unknown_norm' list
print(images_unknown_norm[0])
```

```
Dominik normalized list: 300
<PIL.Image.Image image mode=RGB size=224x224 at 0x184C77A2670>
Unknown normalized list: 150
<PIL.Image.Image image mode=RGB size=224x224 at 0x18478D38E20>
```

Split the dataset:

As previously mentioned, the data must be split into a training, validation, and testing dataset. This is very important to ensure effective model training, evaluation, and performance estimation. The training subset is used for model learning, the validation subset helps fine-tune the model and assess its performance during training, and the testing subset provides an unbiased evaluation of the final model's ability to generalize to unseen data. This process aids in preventing overfitting, obtaining reliable performance estimates, and building robust models capable of handling real-world scenarios.

The dataset can be efficiently split using the train_test_split() function, which requires the images and a corresponding list of labels as input. To create the labels list, you can generate two separate lists, each containing the class name for every image within that class. Since images within an array belong to the same class, this approach simplifies label creation. In addition to the images and labels, train_test_split() expects a value between 0 and 1, representing the desired data split ratio. In this scenario, we have designated 60% of the data for the training dataset, which is slightly below the commonly used default value for such splits. The reason behind this choice lies in the data augmentation process, where two augmented images are created for each original training image, while only one augmented image is used for validation images. This decision was made thoughtfully to address the possibility of augmented images appearing very similar, potentially impacting the model validation.

To ensure a reliable validation process, I opted for a cautious approach, utilizing more original images for validation. This safeguards against any bias introduced by highly similar augmented images. However, to maintain a sufficiently robust training dataset, we increase the number of augmented images generated during later stages of training. This balanced approach ensures the model benefits from augmented data without compromising the validation integrity, ultimately leading to more accurate and dependable evaluation of its performance. The remaining 40% is then allocated for the validation and testing datasets.

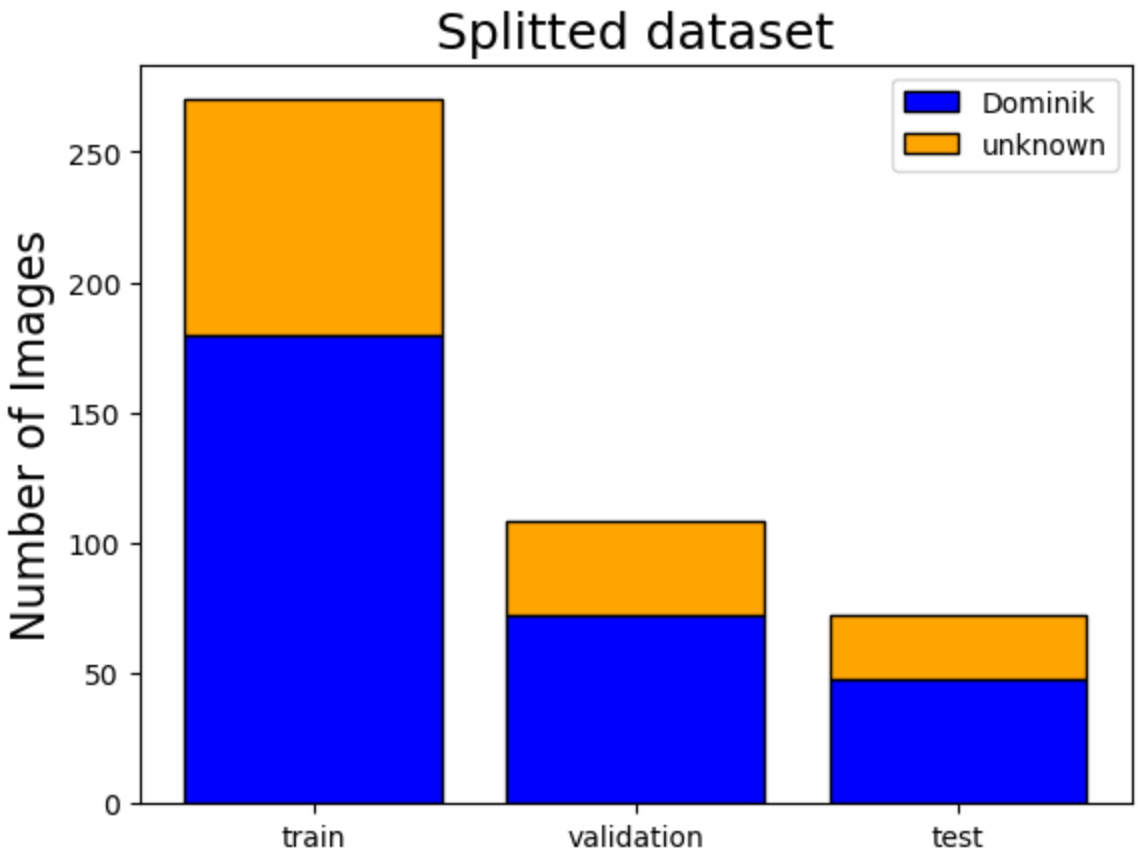
Subsequently, the remaining 40% of the total data is further divided into validation and test datasets. For this partitioning, 60% of the remaining data is assigned to the validation dataset, while the remaining 40% is assigned to the test dataset. These split ratios fall within the standard range commonly used for such divisions, making them suitable choices.

In [ ]:
```
# Create empty lists to store the labels for the 'Unknown' and 'Dominik' classes
y_unknown = []
y_dominik = []

# Append 'unknown' labels to the 'y_unknown' list for each normalized image in the
# This is needed for the 'train_test_split' function -> every image needs a lable
for i in range(len(images_unknown_norm)):
    y_unknown.append('unknown')

# Append 'Dominik' labels to the 'y_dominik' list for each normalized image in the
```

```
# This is needed for the 'train_test_split' function -> every image needs a lable
for i in range(len(images_dominik_norm)):
    y_dominik.append('Dominik')

# Split the 'Unknown' class data into training and remaining datasets using train_t
X_train_unknown, X_rem_unknown, y_train_unknown, y_rem_unknown = train_test_split(i
# Split the 'Dominik' class data into training and remaining datasets using train_t
X_train_dominik, X_rem_dominik, y_train_dominik, y_rem_dominik = train_test_split(i

# Split the remaining 'Unknown' data into validation and test datasets using train_
X_valid_unknown, X_test_unknown, y_valid_unknown, y_test_unknown = train_test_split
# Split the remaining 'Dominik' data into validation and test datasets using train_
X_valid_dominik, X_test_dominik, y_valid_dominik, y_test_dominik = train_test_split
```

To provide a visual representation of the sizes of the individual datasets, the following diagram depicts the three datasets along with the distribution of the two classes within the dataset.

In [ ]:
```
# Define the splits and data origins
split = ['train', 'validation', 'test']
data_origin = ['Dominik', 'unknown']

# Create an array of positions for the bars
pos = np.arange(len(split))

# Create lists of values for the 'Dominik' class for each split
values_dominik = [len(X_train_dominik), len(X_valid_dominik), len(X_test_dominik)]
# Create lists of values for the 'unknown' class for each split
values_unknown = [len(X_train_unknown), len(X_valid_unknown), len(X_test_unknown)]

# Create a bar plot for the 'Dominik' class
plt.bar(pos, values_dominik, color='blue', edgecolor='black')
# Create a bar plot for the 'unknown' class, stacking on top of the 'Dominik' class
plt.bar(pos, values_unknown, color='orange', edgecolor='black', bottom=values_domin
# Set the x-axis ticks to the split names
plt.xticks(pos, split)
# Set the y-axis label
plt.ylabel('Number of Images', fontsize=16)
# Set the title of the plot
plt.title('Splitted dataset', fontsize=18)
# Add a legend to differentiate the data origins
plt.legend(data_origin)
# Display the plot
plt.show()
```



Create a directory for the dataset:

To prepare for the training process, two directories are required: one for the training images and another for the validation images. Therefore, the following code blocks demonstrate the organization of the images into separate folders for each class within the training, validation, and testing datasets. The images are stored in the folder structure to allow easy access and retrieval later in the process.

In [ ]:
```
# Save the 'Unknown' class training images
for i in range(len(X_train_unknown)):
    X_train_unknown[i].save(r'data\train\unknown\image_' + str(i) + '.jpg', 'JPEG')

# Save the 'Unknown' class validation images
for i in range(len(X_valid_unknown)):
    X_valid_unknown[i].save(r'data\validation\unknown\image_' + str(i) + '.jpg', 'J

# Save the 'Unknown' class test images
for i in range(len(X_test_unknown)):
    X_test_unknown[i].save(r'data\test\unknown\image_' + str(i) + '.jpg', 'JPEG')
```

In [ ]:
```
# Save the 'Dominik' class training images
for i in range(len(X_train_dominik)):
    X_train_dominik[i].save(r'data\train\Dominik\image_' + str(i) + '.jpg', 'JPEG')

# Save the 'Dominik' class validation images
for i in range(len(X_valid_dominik)):
    X_valid_dominik[i].save(r'data\validation\Dominik\image_' + str(i) + '.jpg', 'J
```

```
# Save the 'Dominik' class test images
for i in range(len(X_test_dominik)):
    X_test_dominik[i].save(r'data\test\Dominik\image_' + str(i) + '.jpg', 'JPEG')
```

---

5. Data augmentation

After successfully loading the images into memory and preprocessing them to a point where they are ready for training in their respective directories, we can now begin with data augmentation.

Data augmentation refers to the process of artificially expanding the dataset by applying various transformations to the existing images. These transformations can include rotation, scaling, flipping, cropping, or color changes, among others. By applying data augmentation techniques, we can make the dataset more diverse, better preparing the model to handle different variations of input data. This helps improve model generalization and reduces overfitting. It is important to carefully choose data augmentation techniques to maintain the realism of the images and consider the relevance of the transformations for the specific application domain.

But there are also downsides for data augmentation, for example having to much augmented images can lead to issues such as overfitting, increased computational costs, unrealistic representations, and confusion for the model. It is important to strike a balance in data augmentation, considering the dataset's characteristics and the desired generalization ability of the model. Applying moderate and meaningful augmentation techniques can enhance model performance without introducing the drawbacks associated with excessive augmentation. For this reason, each training image is augmented only twice during data augmentation, even though it would be possible to generate many more augmented images. As mentioned before, only one augmented image per original image is created for validation to ensure the integrity of the validation dataset. This approach helps prevent overfitting and maintains the realism of the data. By augmenting each training image twice and each validation image once, a moderate variation of the existing images is achieved, improving the model's robustness.

Before the images can be augmented, they need to be transformed into arrays. This conversion is necessary to enable efficient processing and manipulation of the image data. In addition, data augmentation typically requires an additional dimension in the array structure. To accommodate this requirement, each array is expanded by adding an extra dimension with a value of 0. This is done in the follwing block:

In [ ]:
```
# Convert images to arrays and expand dimensions for the 'Unknown' class training a
train_unknown_array = []
for i in range(len(X_train_unknown)):
    train_unknown_array.append(np.expand_dims(img_to_array(X_train_unknown[i]), axi
```

```
valid_unknown_array = []
for i in range(len(X_valid_unknown)):
    valid_unknown_array.append(np.expand_dims(img_to_array(X_valid_unknown[i]), axi


# Convert images to arrays and expand dimensions for the 'Dominik' class training a
train_dominik_array = []
for i in range(len(X_train_dominik)):
    train_dominik_array.append(np.expand_dims(img_to_array(X_train_dominik[i]), axi

valid_dominik_array = []
for i in range(len(X_valid_dominik)):
    valid_dominik_array.append(np.expand_dims(img_to_array(X_valid_dominik[i]), axi
```

Create the ImageDataGenerator:

In this block, the actual data augmentation takes place using the ImageDataGenerator() function. This function is a powerful tool provided by TensorFlow that allows for real-time data augmentation during model training. In this case, however, augmentation is performed before training so that the resulting images can be stored and analyzed. Otherwise, it would not be possible to review the images before the training.

The ImageDataGenerator() function can be customized by passing various parameters to control the augmentation process. Here are the parameters that are used in this example:

- rotation_range: Specifies the range, in degrees, within which the images can be randomly rotated.
  - Normally this value of the rotation_range is between 0 and 45°, but since the images should not deviate too much from the real images (Setting a very high rotation range can introduce unnatural or implausible rotations that may not reflect real-world scenarios), it is set to 35° in this case.
- width_shift_range: Defines the range, as a fraction of the total width, within which the images can be randomly shifted horizontally.
  - The width_shift_range parameter was set to 0.2. Since there is only one face in the center of the images, it does not matter much if the image is shifted by up to 20%, since no information is lost at the edges.
- height_shift_range: Specifies the range, as a fraction of the total height, within which the images can be randomly shifted vertically.
  - For the height_shift_range the same applies as for the width_shift_range parameter.
- zoom_range: Specifies the range for random zooming in or out of the images.
  - A zoom_range value of 0.2 is chosen to enhance the visibility of facial features in the augmented images. This allows the model to focus more prominently on the details of the face by zooming in on the image.
- horizontal_flip: Enables random horizontal flipping of the images.
  - The horizontal_flip parameter is set to True in order to introduce more variance and increase the diversity of the training data.
- vertical_flip: Enables random vertical flipping of the images.

    - The vertical_flip is set to False because otherwise there are images where the face is displayed upside down.
- brightness_range: Defines the range for randomly adjusting the brightness of the images.
    - The brightness_range is between [0.2, 1.4], these are default values. This makes the image neither too bright nor too dark.
- fill_mode: Specifies the strategy for filling in newly created pixels during image transformations.
    - The fill_mode is set to 'reflect', so the edges that would normally be black when moving the images are filled with the reflected image to simulate a natural transition.

By applying these augmentation parameters, the ImageDataGenerator() function can dynamically generate augmented images.

(TensorFlow. "ImageDataGenerator Class - TensorFlow API." Accessed July 18, 2023. URL: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator)

In [ ]:
```
# Data augmentation parameters
rotation_range = 35
width_shift_range = 0.2
height_shift_range = 0.2
zoom_range = 0.2
brightness_range = [0.2, 1.4]
fill_mode = 'reflect'

# Data augmentation for the 'Unknown' class training and validation sets
datagen = ImageDataGenerator(width_shift_range=width_shift_range, height_shift_rang
```

Create the augmented data:

After the ImageDataGenerator has been created with the previously mentioned parameters, separate loops are run through for the training and validation data of the two classes, in which the augmentation takes place. In each of these four loops, there are augmented image created from each training and validation image. These are then saved directly into the corresponding folders after a normailization. It should be noted that no images of the test dataset are specifically augmented here to ensure the integrity and representativeness of the test dataset. The test dataset is used to evaluate and validate the actual performance of the model on unknown data.

If test images are also augmented, this could lead to overrepresentation of certain augmentation patterns and bias the evaluation of the model. Test data should reflect reality as closely as possible to evaluate the performance of the model in a real environment.

In [ ]:
```
images_train_aug_dominik = []
images_val_aug_dominik = []
```

```
images_train_aug_unknown = []
images_val_aug_unknown = []

# Generate augmented images for the 'Dominik' class training set
for n in range(2):
    for i in range(len(train_dominik_array)):
        train_generator_dominik = datagen.flow(train_dominik_array[i], batch_size=1
        # Generate a batch of augmented images
        batch = train_generator_dominik.next()
        # normalize the generated images
        img_norm = cv2.normalize(batch[0], None, alpha=0, beta=1, norm_type=cv2.NOR
        im = array_to_img(img_norm)
        images_train_aug_dominik.append(im)
        # save the augmented images in the same folder as the original images of th
        im.save(r'data\train\Dominik\augmented_image_' + str(i)+'_'+ str(n) + '.jpg

# Generate augmented images for the 'Dominik' class validation set

for i in range(len(valid_dominik_array)):
    valid_generator_dominik = datagen.flow(valid_dominik_array[i], batch_size=1)
    # Generate a batch of augmented images
    batch = valid_generator_dominik.next()
    # normalize the generated images
    img_norm = cv2.normalize(batch[0], None, alpha=0, beta=1, norm_type=cv2.NORM_MI
    im = array_to_img(img_norm)
    images_val_aug_dominik.append(im)
    # save the augmented images in the same folder as the original images of the Do
    im.save(r'data\validation\Dominik\augmented_image_' + str(i)+ '.jpg')

# Generate augmented images for the 'Unknown' class training set
for n in range(2):
    for i in range(len(train_unknown_array)):
        train_generator_unknown = datagen.flow(train_unknown_array[i], batch_size=1
        # Generate a batch of augmented images
        batch = train_generator_unknown.next()
        # normalize the generated images
        img_norm = cv2.normalize(batch[0], None, alpha=0, beta=1, norm_type=cv2.NOR
        im = array_to_img(img_norm)
        images_train_aug_unknown.append(im)
        # save the augmented images in the same folder as the original images of th
        im.save(r'data\train\unknown\augmented_image_' + str(i)+'_'+ str(n) + '.jpg

# Generate augmented images for the 'Unknown' class validation set

for i in range(len(valid_unknown_array)):
    valid_generator_unknown = datagen.flow(valid_unknown_array[i], batch_size=1)
    # Generate a batch of augmented images
    batch = valid_generator_unknown.next()
    # normalize the generated images
    img_norm = cv2.normalize(batch[0], None, alpha=0, beta=1, norm_type=cv2.NORM_MI
    im = array_to_img(img_norm)
    images_val_aug_unknown.append(im)
    # save the augmented images in the same folder as the original images of the un
    im.save(r'data\validation\unknown\augmented_image_' + str(i)+ '.jpg')
```

Visualize the results:

The following code blocks showcase sample images from the training and validation datasets of the two classes. These images demonstrate the various transformations applied using the specified augmentation parameters. Some transformations may appear more pronounced, while others may be subtle. However, it is evident that the images have undergone modifications.

In [ ]:
```
# print some example images
print("Augmented images of the class Dominik and the dataset train: ")
# Call the function to print the augmented images in the training dataset for the c
printImages(images_train_aug_dominik)
```

```
Augmented images of the class Dominik and the dataset train:
```

In [ ]:
```
# print some example images
print("Augmented images of the class Dominik and the dataset validation: ")
# Call the function to print the augmented images in the validation dataset for the
printImages(images_val_aug_dominik)
```

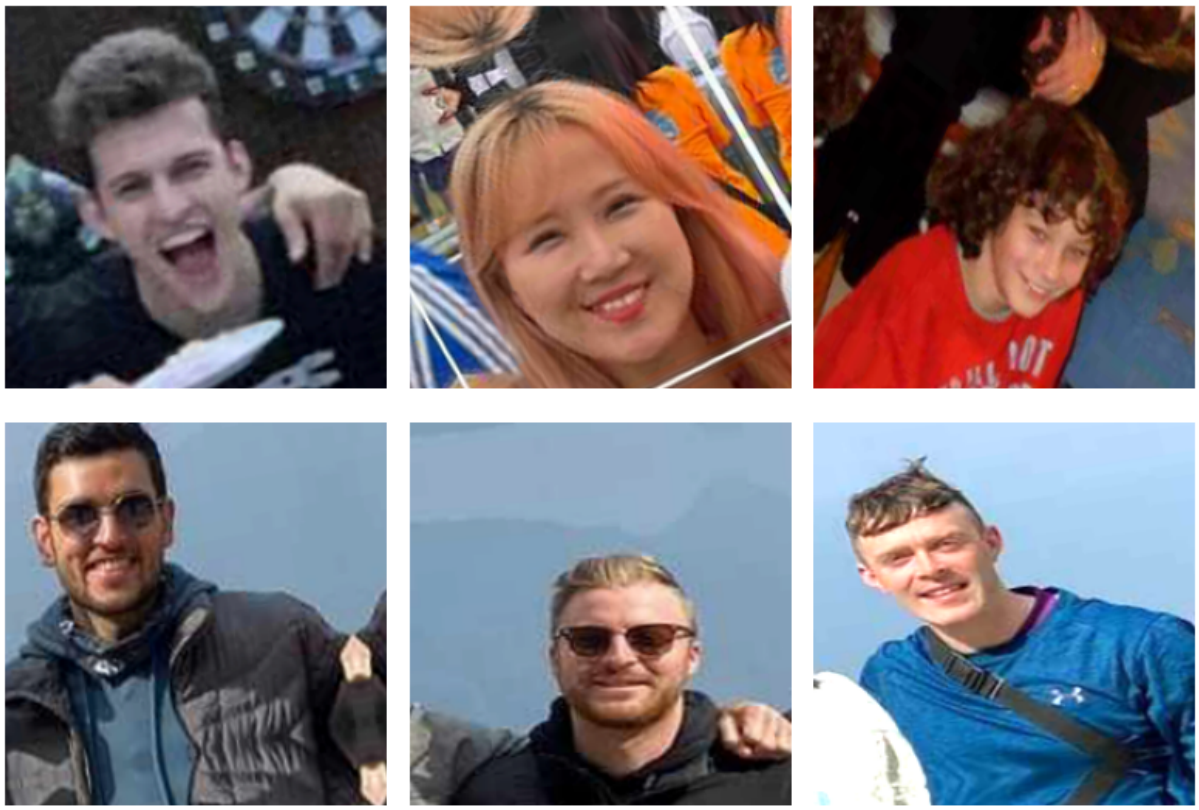
```
Augmented images of the class Dominik and the dataset validation:
```

In [ ]:
```
# print some example images
print("Augmented images of the class Dominik and the dataset train: ")
# Call the function to print the augmented images in the training dataset for the c
printImages(images_train_aug_unknown)
```

```
Augmented images of the class Dominik and the dataset train:
```

In [ ]:
```
# print some example images
print("Augmented images of the class Dominik and the dataset validation: ")
# Call the function to print the augmented images in the validation dataset for the
printImages(images_val_aug_unknown)
```

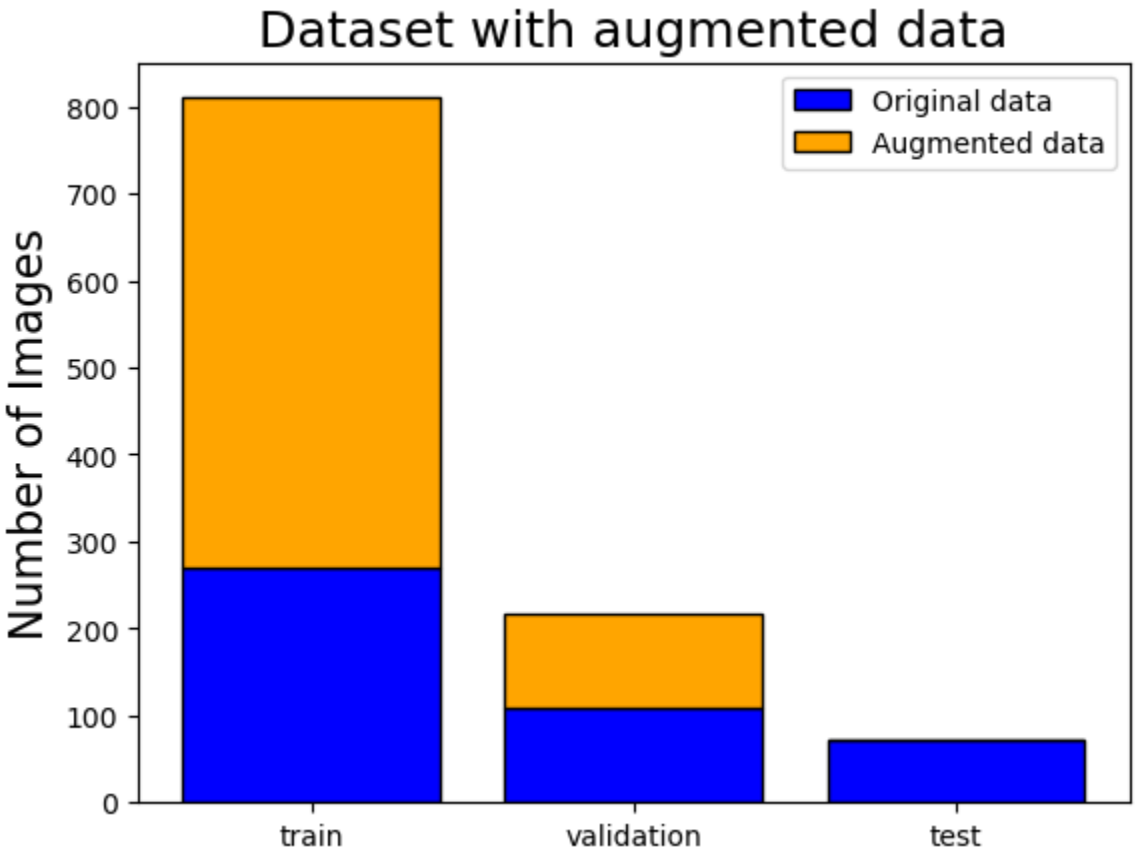
```
Augmented images of the class Dominik and the dataset validation:
```



In the following, a visual representation is created to illustrate the composition of the three datasets, which include both augmented and original data. The figure provides an overview of the training, validation, and test sets, highlighting the inclusion of augmented images alongside the original ones.

In [ ]:
```
# Define the splits and data origins for the plot
split = ['train', 'validation', 'test']
data_origin = ['Original data', 'Augmented data']

# Create an array of positions based on the number of splits
pos = np.arange(len(split))

# Prepare the values for the 'Original data' bars
values_og = [len(train_dominik_array+train_unknown_array), len(valid_dominik_array+

# Prepare the values for the 'Augmented data' bars
values_augmented = [len(images_train_aug_dominik+images_train_aug_unknown), len(ima

# Plotting the bar chart
plt.bar(pos, values_og, color='blue', edgecolor='black')
plt.bar(pos, values_augmented, color='orange', edgecolor='black', bottom=values_og)
plt.xticks(pos, split)
plt.ylabel('Number of Images', fontsize=16)
```

```
plt.title('Dataset with augmented data', fontsize=18)
plt.legend(data_origin)
plt.show()
```



---

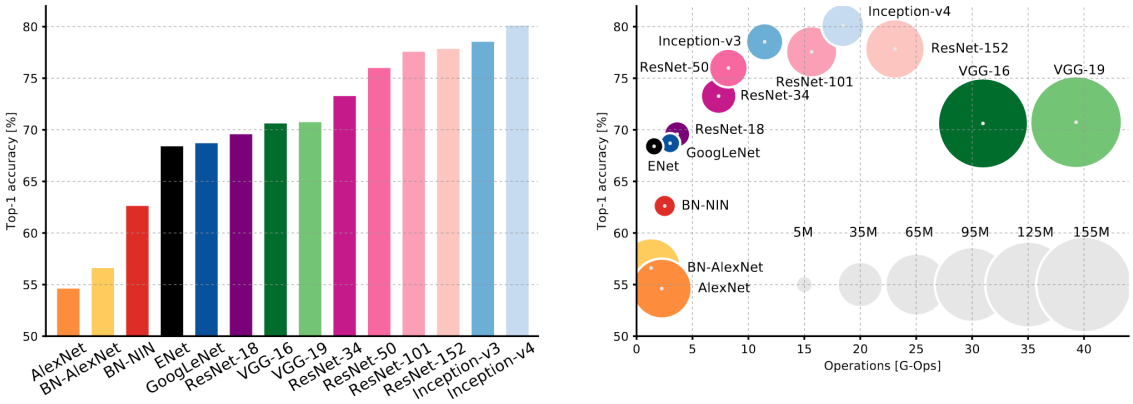
6. Introducing the used Models

Now that the images have been pre-processed and moreover the augmented images have been created, the preparation of the data is over. The data can now be used for training as it is in the directory.

Therefore, it is now time to select the models that serve as the basis for transfer learning. For this, the two diagrams from the following figure are considered, these were already discussed in the lecture. It goes without saying that a model should be chosen which has a good top-1 accuracy. However, care should also be taken that the model is not too large. The size of the model is measured by the number of parameters. Using a model that is excessively large for transfer learning can lead to overfitting, as it adapts too closely to the training data and struggles to generalize to new examples. Larger models also require more computational resources, resulting in longer training times and increased memory usage. For this reason, a balance should be found between the accuracy and the size of the model.

As depicted in the figure, several models are available for the intended purpose, including ResNet-50, Inception-v3, ResNet-101, Inception-v4, and ResNet-152. These models exhibit

high accuracy while maintaining a relatively compact size compared to other Deep learning models like the VGG16 model. Consequently, the subsequent sections focus on training, testing, and comparing these models using the provided dataset. However, it's worth noting that the Inception-v4 model is not utilized due to its unavailability in TensorFlow's official implementation. This is attributed to its recent emergence and complex architecture. Therefore, transfer learning will be performed using the three ResNet models (ResNet-50, ResNet-101, and ResNet-152) alongside the Inception-v3 model.



(Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017)

Set parameter:

Since now that the models that will be used for the training have been selected, they must be created. For this, a few parameters are set first. This includes the input format which is the previously mentioned 224x224 pixels. The weights are initilized with the imagenet weights, which means that the pre-trained weights of the model should be initialized on the ImageNet dataset. ImageNet is a large dataset with millions of images and thousands of classes. By using the pre-trained weights from ImageNet, the model can benefit from the already learned features and structures trained on a wide variety of images.

The dropout rate plays a crucial role in reducing overfitting and enhancing the robustness of the model. By randomly deactivating neurons during training, the model becomes less dependent on specific features and gains resilience against noise. However, it's important to strike a balance. A dropout rate of 0.5 was chosen to prevent excessive dropout while finding a suitable middle ground that mitigates overfitting without compromising the model's capacity. This value ensures a reasonable trade-off between regularization and preserving the model's ability to capture complex patterns in the data. (Vijay, Upendra. "How Does Dropout Help to Avoid Overfitting in Neural Networks." Medium, Medium, 18 Oct. 2019, https://medium.com/@upendravijay2/how-does-dropout-help-to-avoid-overfitting-in-neural-networks-91b90fd86b20)

The last parameter that is set here is the EarlyStopping. EarlyStopping is a technique used during model training to stop the training process based on a predefined criterion. It monitors a chosen metric, typically on a validation dataset, and stops training if the metric

does not improve or worsens consistently over a certain number of epochs. EarlyStopping helps prevent overfitting by ending training early when the model's performance on unseen data is not improving. It optimizes the training process, saves time, and ensures the model is stopped at the point of best validation performance. The metric used here for monitoring is the validation loss, and the patience value is set to 3. This indicates after how many epochs without improvement the training will be stopped. ("EarlyStopping Callback." Keras API Reference, TensorFlow, n.d., https://keras.io/api/callbacks/early_stopping/)

In [ ]:
```
# Define the input shape for the model
input_shape = (224, 224, 3)

# Specify the weights to be used with 'imagenet'
weights = 'imagenet'

dropout_rate = 0.5  # Adjust the dropout rate as desired

# Configure Early Stopping callback
patience = 4  # Number of epochs with no improvement after which training will be s
early_stopping = EarlyStopping(monitor='val_loss', patience=patience)
```

Creating the models:

ResNet50:

In the next 5 code blocks the respective models are created, starting with the ResNet-50 model. For the creation of the model, a base model of the ResNet50 architecture is first created from the tensorflow.keras.applications library. The previously initialized weights and the input_shape are passed as parameters. In addition, the model is informed that the upper parts (top layer) are not included. The top layers (or fully connected layers) are not included in this case because the model is to be developed for a specific classification task with only two classes.

The ResNet50 model was originally developed for the ImageNet challenge, which includes thousands of classes. The top layers in ResNet50 are specifically designed to handle these many classes. If these top layers were retained, they would have to be re-trained for the new classification task with only two classes. This requires a significant amount of training data and computational resources to achieve good results. However, since the given code is only for a classification with two classes, the pre-trained top layers are not used and instead custom classification layers are created. Adding a new classification layer allows the number of outputs to be adjusted to the number of classes and the top layers to be trained specifically for this task without affecting the remaining weights of the base model. This is an important step in the transfer learning process. The same applies to the next step, in which the base layers of the model are frozen. The pre-trained base layers of the ResNet50 model are frozen to preserve the general visual features of the model that have already been learned. When using a pre-trained model such as ResNet50 that has been trained on large and general datasets, it already contains many useful features that are transferable to different visual tasks. By freezing the base layers, one prevents their weights from being

updated during training and one reduces training effort, prevents overfitting, and leverages transfer learning benefits for better performance results with limited training data.

After the base model ResNet50 has been created, further steps are taken to adapt it for a specific image classification task with two classes. For this purpose, a new Sequential model (ResNet50_model) is created consisting of the base model, a GlobalAveragePooling2D layer and a Dropout layer. The GlobalAveragePooling2D layer is used to reduce the spatial dimensions of the output of the base model by averaging over all feature maps. This will allow the model to efficiently handle images of different sizes. The dropout layer is added to reduce overfitting. As previously mentioned, layers are randomly disabled during training for this purpose. Finally, a dense layer with two outputs (classes) is added to perform the final classification. The softmax activation function is used to calculate the probabilities for the two classes, where the sum of these probabilities equals 1.

The final ResNet50_model is therefore a deep neural network with a classification layer trained to classify images of two different classes. It benefits from the pre-trained ResNet50 architecture by using the general visual features of the base model and adapting them to the specific classification task.

Afterward, the exact same model is created again, with the purpose of using two different optimizers later in the process. Having two models of each network architecture is necessary to compare the performance when using different optimizers.

Since these steps are exactly the same for all upcoming models, they will not be explained again. The only difference is that a different model architecture is used. This is described again briefly before the blocks.

In [ ]:
```
# Create a base model using ResNet50 architecture
ResNet50_base_model = ResNet50(include_top=False, input_shape=input_shape, weights=

# Freeze the layers of the base model
for layer in ResNet50_base_model.layers:
    layer.trainable = False

# Create a model for the Adam optimizer and one for the RMSprop optimizer.
# Add a classification head on top of the base model
ResNet50_model_adam = tf.keras.Sequential([
    ResNet50_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])

# Add a classification head on top of the base model
ResNet50_model_rmsprop = tf.keras.Sequential([
    ResNet50_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
```

```
])
```

ResNet101:

In this code, the ResNet101 model is constructed using the ResNet101 architecture from the tensorflow.keras.applications library.

In [ ]:
```
# Create the ResNet101 model
ResNet101_base_model = ResNet101(include_top=False, input_shape=input_shape, weight

# Freeze the layers of the base model
for layer in ResNet101_base_model.layers:
    layer.trainable = False

# Create a model for the Adam optimizer and one for the RMSprop optimizer.
# Add a classification head on top of the base model
ResNet101_model_adam = tf.keras.Sequential([
    ResNet101_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])

# Add a classification head on top of the base model
ResNet101_model_rmsprop = tf.keras.Sequential([
    ResNet101_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])
```

ResNet152:

In this code, the ResNet152 model is constructed using the ResNet152 architecture from the tensorflow.keras.applications library.

In [ ]:
```
# Create the ResNet101 model
ResNet152_base_model = ResNet152(include_top=False, input_shape=input_shape, weight

# Freeze the layers of the base model
for layer in ResNet152_base_model.layers:
    layer.trainable = False

# Create a model for the Adam optimizer and one for the RMSprop optimizer.
# Add a classification head on top of the base model
ResNet152_model_adam = tf.keras.Sequential([
    ResNet152_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])

# Add a classification head on top of the base model
ResNet152_model_rmsprop = tf.keras.Sequential([
    ResNet152_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
```

```
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])
```

Inception-v3:

In this code, the Inception-v3 model is constructed using the Inception-v3 architecture from the tensorflow.keras.applications library.

In [ ]:
```
# Create the InceptionV3 model
InceptionV3_base_model = InceptionV3(include_top=False, input_shape=input_shape, we

# Freeze the layers of the base model
for layer in InceptionV3_base_model.layers:
    layer.trainable = False

# Create a model for the Adam optimizer and one for the RMSprop optimizer.
# Add a classification head on top of the base model
InceptionV3_model_adam = tf.keras.Sequential([
    InceptionV3_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])

# Add a classification head on top of the base model
InceptionV3_model_rmsprop = tf.keras.Sequential([
    InceptionV3_base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    Dropout(dropout_rate),  # Add Dropout layer with the specified rate
    tf.keras.layers.Dense(2, activation='softmax')
])
```

---

7. Training

After creating the models, the next crucial step is training them. To begin the training process, an ImageDataGenerator is required. Specifically, two separate ImageDataGenerators are created for the training data and the validation data. Notably, no additional parameters are defined in either generator. The reason behind this is that no further data augmentation is intended at this stage. Instead, the ImageDataGenerators are utilized purely for training purposes. While it is indeed possible to apply real-time data augmentation during training to address data scarcity, in this case, data augmentation was already performed earlier, rendering these parameters unnecessary.

The data generator is used here because it loads the images incrementally instead of all at once, which reduces memory usage and requires fewer computational resources.

In [ ]:
```
# if you want more augmented data you can uncomment the parameters in the ImageData
train_datagen = ImageDataGenerator(
```

```
    # rotation_range=20,
    # width_shift_range=0.2,
    # height_shift_range=0.2,
    # shear_range=0.2,
    # zoom_range=0.2,
    # horizontal_flip=True
)

validation_datagen = ImageDataGenerator()
```

Create the data generator:

In this step, the two data generators are created, which are used for training the models, one for the training images and one for the validation images. Since the data is located in the directory, the function flow_from_directory() is used for this. The flow_from_directory function is a method in the Keras library that allows images or other data to be loaded directly from a directory and used in batches for training or evaluating a model. The function receives as input values the path to the directory of the training or validation data, the batch size and the class mode. The batch size is set to 16. This is a common value for such a project. Since the dataset is not very large, this value is sufficient and it is not necessary to use 32. Also, it is a sufficient number of data points to provide stable gradient estimates without using too much memory.

The class_mode parameter is set to 'categorical', which means that the class labels of the data should be interpreted as One-Hot encoded vectors. Significantly, through the utilization of One-Hot encoding, neural networks, particularly those employing softmax outputs, can effectively handle categorical data and articulate predictions in the form of distinct class probabilities. Additionally the image size is passed and the shuffle is set to true, so that the images are shuffled.

In [ ]:
```
# Create data generators for training and validation data

# Specify the directories containing the training and validation data
train_dir = "data/train/"
val_dir = "data/validation/"

# Set the batch size for each iteration during training
batch_size = 16

# Define the desired size for input images
IMAGE_SIZE = (224, 224)

class_mode='categorical'


# Create a data generator for training data
train_generator = train_datagen.flow_from_directory(
    train_dir,                          # Directory containing the training data
    batch_size=batch_size,              # Number of samples per batch
    target_size=IMAGE_SIZE,             # Resize input images to the specified si
    shuffle=True,                       # Shuffle the order of images during trai
```

```
    class_mode=class_mode                # Generate one-hot encoded labels for mul
)

# Create a data generator for validation data
validation_generator = validation_datagen.flow_from_directory(
    val_dir,                             # Directory containing the validation dat
    batch_size=batch_size,               # Number of samples per batch
    target_size=IMAGE_SIZE,              # Resize input images to the specified si
    shuffle=True,                        # Shuffle the order of images during vali
    class_mode=class_mode                # Generate one-hot encoded labels for mul
)
```

```
Found 810 images belonging to 2 classes.
Found 216 images belonging to 2 classes.
```

Compile the models:

Before the training can begin, all models must be compiled once. For this the optimizer is passed, as well as the loss and the metric.

For the optimizer, as mentioned before, two different variants are tested. These are the Adam Optimizer and the RMSprop Optimizer. Adam and RMSprop are both adaptive optimization methods with an adaptive learning rate for training Deep Learning models. Adam combines the advantages of Adagrad and SGD, while RMSprop is well suited for non-stationary or ill-conditioned problems. Both optimizers adjust the learning rate to each weight in the model and use a moving average of squared gradients. Adam can converge faster and is suitable for large datasets and complex models, while RMSprop is robust to widely varying gradient values. The choice between the two depends on the specific model and dataset conditions, and it is recommended to test both to obtain the best performance for the given problem.

When compiling models, the parameter metrics specifies which metric should be used to evaluate the model during training and evaluation. In this case, the Accuracy metric is selected. Accuracy is a commonly used metric in classification problems. It indicates how many predictions of the model match the actual class labels.

The third parameter passed is the loss, this parameter uses the Categorical Cross Entropy as a loss function during training. This loss function is often used for multi-class classification problems, where each data sample belongs to exactly one of several classes.

After comiling, there are two models for each network architecture, one with the Adam Optimizer and the other with the Rmsprop opmimizer. These models are now ready to be trained in the next step.

In [ ]:
```
# Compile the models
# Set the optimizer, loss function, and metrics for evaluation
optimizer_rms = 'rmsprop'
optimizer_adam = 'adam'
loss = 'categorical_crossentropy'
metrics = ['accuracy']
```

```
# Compile each model with the adam/rmsprop optimizer, loss, and metrics
ResNet50_model_adam.compile(optimizer=optimizer_adam, loss=loss, metrics=metrics)
ResNet101_model_adam.compile(optimizer=optimizer_adam, loss=loss, metrics=metrics)
ResNet152_model_adam.compile(optimizer=optimizer_adam, loss=loss, metrics=metrics)
InceptionV3_model_adam.compile(optimizer=optimizer_adam, loss=loss, metrics=metrics

ResNet50_model_rmsprop.compile(optimizer=optimizer_rms, loss=loss, metrics=metrics)
ResNet101_model_rmsprop.compile(optimizer=optimizer_rms, loss=loss, metrics=metrics
ResNet152_model_rmsprop.compile(optimizer=optimizer_rms, loss=loss, metrics=metrics
InceptionV3_model_rmsprop.compile(optimizer=optimizer_rms, loss=loss, metrics=metri
```

Train the models:

For the training different parameters are needed which are defined before. This is on the one hand the number of epochs, this is set here to 15. An epoch represents a single run of the entire training dataset during the training process. Since this is transfer learning, not so many epochs are needed in general. In addition, the new problem with only two classes is not particularly complicated, which also means that fewer epochs are needed. Finally, the dataset is also limited in size, so the number of epochs is limited. For these reasons and after some testing the value 15 turned out to be a good value.

For the further procedure the total amount of validation data is needed, this is determined here. With the total number of validation images the validation_steps are determined, this variable specifies how many steps (or batches) are run through during an epoch for the validation. The same procedure is used to determine the steps_per_epoch. This parameter specifies how many steps (or batches) are passed during an epoch for training. However, these should actually be determined using the total amount of data divided by the batch size and not using the validation data. Since there were problems with the training because too much data was used for only one epoch, it was decided to calculate the same as with the validation_steps. Thus one epoch is smaller, but more epochs can be trained.

For the actual training the fit() function is used now. The two previously created data generators are passed to it, as well as the epochs. In addition, the steps_per_epoch and the validation_steps are passed. As last parameter the early_stopping parameter is passed, this was already explained before. With the parameters which are passed, there are no differences between the different models.

Since there are a lot of models, the trainnig can take a while, but since all models are used for the evaluation, none of the models should be bracketed. The history of each training is written to a varaible, which will become important later.

In [ ]:
```
# Train the models
# Calculate the total number of validation images
total_val = len(images_val_aug_dominik + images_val_aug_unknown + valid_dominik_arr

# Set the number of epochs for training
epochs = 15
```

```
# Calculate the steps per epoch and validation steps
steps_per_epoch = total_val // batch_size
validation_steps = total_val // batch_size

# Train each model and store the training history
print("################################################################################")
# ResNet50 model
print('ResNet50 (optimizer: adam):')
history_ResNet50_adam = ResNet50_model_adam.fit(train_generator, epochs=epochs, ste
print('ResNet50 (optimizer: rmsprop):')
history_ResNet50_rmsprop = ResNet50_model_rmsprop.fit(train_generator, epochs=epoch

print("################################################################################")
# ResNet101 model
print('ResNet101 (optimizer: adam):')
history_ResNet101_adam = ResNet101_model_adam.fit(train_generator, epochs=epochs, s
print('ResNet101 (optimizer: rmsprop):')
history_ResNet101_rmsprop = ResNet101_model_rmsprop.fit(train_generator, epochs=epo

print("################################################################################")
# ResNet152 model
print('ResNet152 (optimizer: adam):')
history_ResNet152_adam = ResNet152_model_adam.fit(train_generator, steps_per_epoch=
print('ResNet152 (optimizer: rmsprop):')
history_ResNet152_rmsprop = ResNet152_model_rmsprop.fit(train_generator, steps_per_

print("################################################################################")
# InceptionV3 model
print('InceptionV3 (optimizer: adam):')
history_InceptionV3_adam = InceptionV3_model_adam.fit(train_generator, steps_per_ep
print('InceptionV3 (optimizer: rmsprop):')
history_InceptionV3_rmsprop = InceptionV3_model_rmsprop.fit(train_generator, steps_
```

```
################################################################################
ResNet50 (optimizer: adam):
Epoch 1/15
13/13 [==============================] - 13s 1s/step - loss: 0.0371 - accuracy: 0.99
04 - val_loss: 0.0131 - val_accuracy: 1.0000
Epoch 2/15
13/13 [==============================] - 13s 1s/step - loss: 0.0424 - accuracy: 0.98
51 - val_loss: 0.0117 - val_accuracy: 1.0000
Epoch 3/15
13/13 [==============================] - 13s 1s/step - loss: 0.0467 - accuracy: 0.98
51 - val_loss: 0.0112 - val_accuracy: 1.0000
Epoch 4/15
13/13 [==============================] - 14s 1s/step - loss: 0.0376 - accuracy: 0.99
04 - val_loss: 0.0099 - val_accuracy: 1.0000
Epoch 5/15
13/13 [==============================] - 13s 1s/step - loss: 0.0248 - accuracy: 0.99
01 - val_loss: 0.0171 - val_accuracy: 0.9952
Epoch 6/15
13/13 [==============================] - 13s 1s/step - loss: 0.0272 - accuracy: 0.99
52 - val_loss: 0.0079 - val_accuracy: 1.0000
Epoch 7/15
13/13 [==============================] - 13s 1s/step - loss: 0.0285 - accuracy: 0.99
52 - val_loss: 0.0081 - val_accuracy: 1.0000
Epoch 8/15
13/13 [==============================] - 13s 1s/step - loss: 0.0174 - accuracy: 1.00
00 - val_loss: 0.0111 - val_accuracy: 1.0000
Epoch 9/15
13/13 [==============================] - 13s 1s/step - loss: 0.0366 - accuracy: 0.98
08 - val_loss: 0.0072 - val_accuracy: 1.0000
Epoch 10/15
13/13 [==============================] - 13s 1s/step - loss: 0.0227 - accuracy: 1.00
00 - val_loss: 0.0068 - val_accuracy: 1.0000
Epoch 11/15
13/13 [==============================] - 13s 1s/step - loss: 0.0275 - accuracy: 0.98
56 - val_loss: 0.0070 - val_accuracy: 1.0000
Epoch 12/15
13/13 [==============================] - 13s 1s/step - loss: 0.0268 - accuracy: 0.98
56 - val_loss: 0.0059 - val_accuracy: 1.0000
Epoch 13/15
13/13 [==============================] - 13s 1s/step - loss: 0.0118 - accuracy: 1.00
00 - val_loss: 0.0057 - val_accuracy: 1.0000
Epoch 14/15
13/13 [==============================] - 13s 1s/step - loss: 0.0208 - accuracy: 1.00
00 - val_loss: 0.0054 - val_accuracy: 1.0000
Epoch 15/15
13/13 [==============================] - 13s 1s/step - loss: 0.0147 - accuracy: 0.99
52 - val_loss: 0.0059 - val_accuracy: 1.0000
ResNet50 (optimizer: rmsprop):
Epoch 1/15
13/13 [==============================] - 13s 1s/step - loss: 0.0153 - accuracy: 0.99
52 - val_loss: 0.0053 - val_accuracy: 1.0000
Epoch 2/15
13/13 [==============================] - 13s 1s/step - loss: 0.0182 - accuracy: 0.99
50 - val_loss: 0.0052 - val_accuracy: 1.0000
Epoch 3/15
13/13 [==============================] - 13s 1s/step - loss: 0.0306 - accuracy: 0.99
```

```
04 - val_loss: 0.0042 - val_accuracy: 1.0000
Epoch 4/15
13/13 [==============================] - 13s 1s/step - loss: 0.0238 - accuracy: 0.99
04 - val_loss: 0.0040 - val_accuracy: 1.0000
Epoch 5/15
13/13 [==============================] - 13s 1s/step - loss: 0.0178 - accuracy: 0.99
52 - val_loss: 0.0040 - val_accuracy: 1.0000
Epoch 6/15
13/13 [==============================] - 13s 1s/step - loss: 0.0093 - accuracy: 1.00
00 - val_loss: 0.0048 - val_accuracy: 1.0000
Epoch 7/15
13/13 [==============================] - 13s 1s/step - loss: 0.0076 - accuracy: 1.00
00 - val_loss: 0.0039 - val_accuracy: 1.0000
Epoch 8/15
13/13 [==============================] - 13s 1s/step - loss: 0.0166 - accuracy: 0.99
50 - val_loss: 0.0066 - val_accuracy: 1.0000
Epoch 9/15
13/13 [==============================] - 13s 1s/step - loss: 0.0290 - accuracy: 0.99
04 - val_loss: 0.0032 - val_accuracy: 1.0000
Epoch 10/15
13/13 [==============================] - 13s 1s/step - loss: 0.0124 - accuracy: 0.99
04 - val_loss: 0.0028 - val_accuracy: 1.0000
Epoch 11/15
13/13 [==============================] - 13s 1s/step - loss: 0.0108 - accuracy: 0.99
52 - val_loss: 0.0026 - val_accuracy: 1.0000
Epoch 12/15
13/13 [==============================] - 13s 1s/step - loss: 0.0216 - accuracy: 0.99
04 - val_loss: 0.0025 - val_accuracy: 1.0000
Epoch 13/15
13/13 [==============================] - 13s 1s/step - loss: 0.0044 - accuracy: 1.00
00 - val_loss: 0.0022 - val_accuracy: 1.0000
Epoch 14/15
13/13 [==============================] - 13s 1s/step - loss: 0.0095 - accuracy: 0.99
52 - val_loss: 0.0018 - val_accuracy: 1.0000
Epoch 15/15
13/13 [==============================] - 13s 1s/step - loss: 0.0131 - accuracy: 0.99
50 - val_loss: 0.0027 - val_accuracy: 1.0000
################################################################################
ResNet101 (optimizer: adam):
Epoch 1/15
13/13 [==============================] - 24s 2s/step - loss: 0.1130 - accuracy: 0.97
12 - val_loss: 0.0522 - val_accuracy: 0.9808
Epoch 2/15
13/13 [==============================] - 24s 2s/step - loss: 0.0971 - accuracy: 0.96
15 - val_loss: 0.0331 - val_accuracy: 1.0000
Epoch 3/15
13/13 [==============================] - 25s 2s/step - loss: 0.0702 - accuracy: 0.97
60 - val_loss: 0.0226 - val_accuracy: 0.9952
Epoch 4/15
13/13 [==============================] - 24s 2s/step - loss: 0.0651 - accuracy: 0.97
03 - val_loss: 0.0227 - val_accuracy: 0.9952
Epoch 5/15
13/13 [==============================] - 24s 2s/step - loss: 0.0479 - accuracy: 0.98
08 - val_loss: 0.0201 - val_accuracy: 0.9952
Epoch 6/15
13/13 [==============================] - 24s 2s/step - loss: 0.0733 - accuracy: 0.97
```

```
60 - val_loss: 0.0273 - val_accuracy: 0.9952
Epoch 7/15
13/13 [==============================] - 24s 2s/step - loss: 0.0631 - accuracy: 0.97
12 - val_loss: 0.0194 - val_accuracy: 0.9952
Epoch 8/15
13/13 [==============================] - 24s 2s/step - loss: 0.0593 - accuracy: 0.97
60 - val_loss: 0.0170 - val_accuracy: 0.9952
Epoch 9/15
13/13 [==============================] - 24s 2s/step - loss: 0.0368 - accuracy: 0.99
04 - val_loss: 0.0167 - val_accuracy: 0.9952
Epoch 10/15
13/13 [==============================] - 24s 2s/step - loss: 0.0645 - accuracy: 0.98
56 - val_loss: 0.0150 - val_accuracy: 0.9952
Epoch 11/15
13/13 [==============================] - 24s 2s/step - loss: 0.0472 - accuracy: 0.98
51 - val_loss: 0.0143 - val_accuracy: 0.9952
Epoch 12/15
13/13 [==============================] - 24s 2s/step - loss: 0.0411 - accuracy: 0.98
51 - val_loss: 0.0151 - val_accuracy: 0.9952
Epoch 13/15
13/13 [==============================] - 24s 2s/step - loss: 0.0232 - accuracy: 0.99
52 - val_loss: 0.0135 - val_accuracy: 0.9952
Epoch 14/15
13/13 [==============================] - 24s 2s/step - loss: 0.0448 - accuracy: 0.97
60 - val_loss: 0.0136 - val_accuracy: 0.9952
Epoch 15/15
13/13 [==============================] - 24s 2s/step - loss: 0.0393 - accuracy: 0.99
01 - val_loss: 0.0130 - val_accuracy: 0.9952
ResNet101 (optimizer: rmsprop):
Epoch 1/15
13/13 [==============================] - 25s 2s/step - loss: 0.6367 - accuracy: 0.78
37 - val_loss: 0.1663 - val_accuracy: 0.9567
Epoch 2/15
13/13 [==============================] - 24s 2s/step - loss: 0.2693 - accuracy: 0.90
38 - val_loss: 0.0869 - val_accuracy: 0.9856
Epoch 3/15
13/13 [==============================] - 24s 2s/step - loss: 0.1994 - accuracy: 0.93
75 - val_loss: 0.0621 - val_accuracy: 0.9856
Epoch 4/15
13/13 [==============================] - 24s 2s/step - loss: 0.1755 - accuracy: 0.92
31 - val_loss: 0.0398 - val_accuracy: 0.9904
Epoch 5/15
13/13 [==============================] - 24s 2s/step - loss: 0.1572 - accuracy: 0.94
71 - val_loss: 0.0605 - val_accuracy: 0.9856
Epoch 6/15
13/13 [==============================] - 24s 2s/step - loss: 0.1348 - accuracy: 0.93
56 - val_loss: 0.0313 - val_accuracy: 0.9952
Epoch 7/15
13/13 [==============================] - 24s 2s/step - loss: 0.0671 - accuracy: 0.98
02 - val_loss: 0.0247 - val_accuracy: 0.9904
Epoch 8/15
13/13 [==============================] - 24s 2s/step - loss: 0.0523 - accuracy: 0.97
03 - val_loss: 0.0220 - val_accuracy: 0.9952
Epoch 9/15
13/13 [==============================] - 24s 2s/step - loss: 0.0385 - accuracy: 0.99
04 - val_loss: 0.0155 - val_accuracy: 1.0000
```

```
Epoch 10/15
13/13 [==============================] - 24s 2s/step - loss: 0.0381 - accuracy: 0.98
56 - val_loss: 0.0133 - val_accuracy: 1.0000
Epoch 11/15
13/13 [==============================] - 24s 2s/step - loss: 0.0676 - accuracy: 0.98
02 - val_loss: 0.0124 - val_accuracy: 1.0000
Epoch 12/15
13/13 [==============================] - 24s 2s/step - loss: 0.0471 - accuracy: 0.98
56 - val_loss: 0.0210 - val_accuracy: 0.9952
Epoch 13/15
13/13 [==============================] - 24s 2s/step - loss: 0.0497 - accuracy: 0.98
56 - val_loss: 0.0282 - val_accuracy: 0.9952
Epoch 14/15
13/13 [==============================] - 24s 2s/step - loss: 0.0436 - accuracy: 0.99
04 - val_loss: 0.0093 - val_accuracy: 1.0000
Epoch 15/15
13/13 [==============================] - 24s 2s/step - loss: 0.0285 - accuracy: 0.99
04 - val_loss: 0.0089 - val_accuracy: 1.0000
################################################################################
ResNet152 (optimizer: adam):
Epoch 1/15
13/13 [==============================] - 37s 3s/step - loss: 0.8436 - accuracy: 0.61
54 - val_loss: 0.3252 - val_accuracy: 0.8558
Epoch 2/15
13/13 [==============================] - 36s 3s/step - loss: 0.3990 - accuracy: 0.84
62 - val_loss: 0.1056 - val_accuracy: 0.9808
Epoch 3/15
13/13 [==============================] - 35s 3s/step - loss: 0.1854 - accuracy: 0.91
58 - val_loss: 0.0675 - val_accuracy: 0.9856
Epoch 4/15
13/13 [==============================] - 35s 3s/step - loss: 0.1696 - accuracy: 0.93
27 - val_loss: 0.0472 - val_accuracy: 0.9904
Epoch 5/15
13/13 [==============================] - 35s 3s/step - loss: 0.1557 - accuracy: 0.92
79 - val_loss: 0.0344 - val_accuracy: 0.9904
Epoch 6/15
13/13 [==============================] - 35s 3s/step - loss: 0.0631 - accuracy: 0.98
51 - val_loss: 0.0304 - val_accuracy: 0.9904
Epoch 7/15
13/13 [==============================] - 35s 3s/step - loss: 0.0985 - accuracy: 0.97
03 - val_loss: 0.0288 - val_accuracy: 0.9904
Epoch 8/15
13/13 [==============================] - 35s 3s/step - loss: 0.0514 - accuracy: 0.99
04 - val_loss: 0.0227 - val_accuracy: 0.9952
Epoch 9/15
13/13 [==============================] - 35s 3s/step - loss: 0.0717 - accuracy: 0.97
60 - val_loss: 0.0231 - val_accuracy: 1.0000
Epoch 10/15
13/13 [==============================] - 35s 3s/step - loss: 0.0773 - accuracy: 0.97
03 - val_loss: 0.0240 - val_accuracy: 0.9904
Epoch 11/15
13/13 [==============================] - 35s 3s/step - loss: 0.0663 - accuracy: 0.98
02 - val_loss: 0.0237 - val_accuracy: 0.9904
Epoch 12/15
13/13 [==============================] - 36s 3s/step - loss: 0.0622 - accuracy: 0.98
08 - val_loss: 0.0243 - val_accuracy: 0.9904
```

```
ResNet152 (optimizer: rmsprop):
Epoch 1/15
13/13 [==============================] - 60s 5s/step - loss: 0.7275 - accuracy: 0.72
12 - val_loss: 0.1864 - val_accuracy: 0.9519
Epoch 2/15
13/13 [==============================] - 58s 4s/step - loss: 0.2856 - accuracy: 0.90
38 - val_loss: 0.1089 - val_accuracy: 0.9808
Epoch 3/15
13/13 [==============================] - 58s 4s/step - loss: 0.2570 - accuracy: 0.89
90 - val_loss: 0.0615 - val_accuracy: 1.0000
Epoch 4/15
13/13 [==============================] - 58s 4s/step - loss: 0.2280 - accuracy: 0.90
87 - val_loss: 0.0399 - val_accuracy: 1.0000
Epoch 5/15
13/13 [==============================] - 59s 5s/step - loss: 0.1262 - accuracy: 0.96
15 - val_loss: 0.0356 - val_accuracy: 1.0000
Epoch 6/15
13/13 [==============================] - 58s 4s/step - loss: 0.0852 - accuracy: 0.97
03 - val_loss: 0.0230 - val_accuracy: 1.0000
Epoch 7/15
13/13 [==============================] - 58s 4s/step - loss: 0.0810 - accuracy: 0.98
08 - val_loss: 0.0222 - val_accuracy: 0.9952
Epoch 8/15
13/13 [==============================] - 58s 4s/step - loss: 0.0647 - accuracy: 0.99
04 - val_loss: 0.0140 - val_accuracy: 1.0000
Epoch 9/15
13/13 [==============================] - 58s 4s/step - loss: 0.0580 - accuracy: 0.97
60 - val_loss: 0.0145 - val_accuracy: 1.0000
Epoch 10/15
13/13 [==============================] - 57s 4s/step - loss: 0.0319 - accuracy: 0.99
01 - val_loss: 0.0160 - val_accuracy: 0.9952
Epoch 11/15
13/13 [==============================] - 58s 4s/step - loss: 0.0570 - accuracy: 0.98
56 - val_loss: 0.0103 - val_accuracy: 1.0000
Epoch 12/15
13/13 [==============================] - 58s 4s/step - loss: 0.0270 - accuracy: 0.98
56 - val_loss: 0.0067 - val_accuracy: 1.0000
Epoch 13/15
13/13 [==============================] - 59s 5s/step - loss: 0.0772 - accuracy: 0.97
12 - val_loss: 0.0128 - val_accuracy: 1.0000
Epoch 14/15
13/13 [==============================] - 58s 4s/step - loss: 0.0578 - accuracy: 0.98
08 - val_loss: 0.0080 - val_accuracy: 1.0000
Epoch 15/15
13/13 [==============================] - 58s 4s/step - loss: 0.0649 - accuracy: 0.97
60 - val_loss: 0.0164 - val_accuracy: 1.0000
################################################################################
InceptionV3 (optimizer: adam):
Epoch 1/15
13/13 [==============================] - 15s 1s/step - loss: 26.3148 - accuracy: 0.5
396 - val_loss: 11.8151 - val_accuracy: 0.6635
Epoch 2/15
13/13 [==============================] - 14s 1s/step - loss: 12.8106 - accuracy: 0.6
250 - val_loss: 8.0830 - val_accuracy: 0.6971
Epoch 3/15
13/13 [==============================] - 14s 1s/step - loss: 11.8177 - accuracy: 0.6
```

```
980 - val_loss: 5.5396 - val_accuracy: 0.7596
Epoch 4/15
13/13 [==============================] - 14s 1s/step - loss: 10.0622 - accuracy: 0.6
442 - val_loss: 3.3924 - val_accuracy: 0.8365
Epoch 5/15
13/13 [==============================] - 15s 1s/step - loss: 5.7305 - accuracy: 0.71
63 - val_loss: 4.0325 - val_accuracy: 0.8221
Epoch 6/15
13/13 [==============================] - 14s 1s/step - loss: 9.3734 - accuracy: 0.72
12 - val_loss: 3.6488 - val_accuracy: 0.8365
Epoch 7/15
13/13 [==============================] - 14s 1s/step - loss: 9.7096 - accuracy: 0.64
42 - val_loss: 4.6019 - val_accuracy: 0.8269
Epoch 8/15
13/13 [==============================] - 14s 1s/step - loss: 7.1086 - accuracy: 0.71
15 - val_loss: 3.4262 - val_accuracy: 0.8558
InceptionV3 (optimizer: rmsprop):
Epoch 1/15
13/13 [==============================] - 14s 1s/step - loss: 17.5035 - accuracy: 0.5
594 - val_loss: 3.5128 - val_accuracy: 0.7548
Epoch 2/15
13/13 [==============================] - 14s 1s/step - loss: 11.2736 - accuracy: 0.6
635 - val_loss: 11.5836 - val_accuracy: 0.7404
Epoch 3/15
13/13 [==============================] - 13s 1s/step - loss: 10.3800 - accuracy: 0.6
971 - val_loss: 6.8041 - val_accuracy: 0.7837
Epoch 4/15
13/13 [==============================] - 14s 1s/step - loss: 11.6438 - accuracy: 0.6
683 - val_loss: 3.2874 - val_accuracy: 0.8365
Epoch 5/15
13/13 [==============================] - 13s 1s/step - loss: 8.3308 - accuracy: 0.69
31 - val_loss: 5.4727 - val_accuracy: 0.7933
Epoch 6/15
13/13 [==============================] - 14s 1s/step - loss: 9.3341 - accuracy: 0.71
15 - val_loss: 3.0507 - val_accuracy: 0.8462
Epoch 7/15
13/13 [==============================] - 13s 1s/step - loss: 9.4684 - accuracy: 0.74
52 - val_loss: 2.6788 - val_accuracy: 0.8558
Epoch 8/15
13/13 [==============================] - 14s 1s/step - loss: 8.6323 - accuracy: 0.72
12 - val_loss: 3.5541 - val_accuracy: 0.8558
Epoch 9/15
13/13 [==============================] - 13s 1s/step - loss: 6.9179 - accuracy: 0.79
33 - val_loss: 9.9621 - val_accuracy: 0.7548
Epoch 10/15
13/13 [==============================] - 13s 1s/step - loss: 9.0640 - accuracy: 0.72
12 - val_loss: 2.3784 - val_accuracy: 0.8606
Epoch 11/15
13/13 [==============================] - 13s 1s/step - loss: 7.1247 - accuracy: 0.75
96 - val_loss: 2.7918 - val_accuracy: 0.8654
Epoch 12/15
13/13 [==============================] - 13s 1s/step - loss: 6.7140 - accuracy: 0.74
52 - val_loss: 5.7625 - val_accuracy: 0.8173
Epoch 13/15
13/13 [==============================] - 13s 1s/step - loss: 6.5652 - accuracy: 0.77
40 - val_loss: 3.3885 - val_accuracy: 0.8606
```

```
Epoch 14/15
13/13 [==============================] - 13s 1s/step - loss: 6.1852 - accuracy: 0.77
88 - val_loss: 8.2463 - val_accuracy: 0.8029
```

Save the models:

Once the models are trained, they can be used for testing or evaluation. But to make sure that the models do not have to be trained again each time, they are saved first.

There are two options for saving a model: either only the weights can be stored, or the entire model can be saved. Saving only the weights has the advantage of being more memory-efficient, as it only requires storing the model parameters. Moreover, loading the weights is faster compared to saving the whole model. However, when only the weights are saved, the model cannot be directly used for predictions or inference on new data, as the model architecture must first be rebuilt and reconstructed before it can be utilized.

On the other hand, saving the entire model allows for direct use in making predictions on new data without the need for rebuilding. It contains all the necessary information, including the model architecture, weights, and configuration. Additionally, saving the entire model enables seamless reuse, including any custom layers, functions, or other components incorporated into the architecture. Nonetheless, this method requires significantly more storage space.

In this example, both methods are utilized to showcase their functionality.

In [ ]:
```
# Save the weights of the trained models
ResNet50_model_adam.save_weights('models/ResNet50_model_weights_adam.h5')
ResNet101_model_adam.save_weights('models/ResNet101_model_weights_adam.h5')
ResNet152_model_adam.save_weights('models/ResNet152_model_weights_adam.h5')
InceptionV3_model_adam.save_weights('models/InceptionV3_model_weights_adam.h5')

ResNet50_model_rmsprop.save_weights('models/ResNet50_model_weights_rmsprop.h5')
ResNet101_model_rmsprop.save_weights('models/ResNet101_model_weights_rmsprop.h5')
ResNet152_model_rmsprop.save_weights('models/ResNet152_model_weights_rmsprop.h5')
InceptionV3_model_rmsprop.save_weights('models/InceptionV3_model_weights_rmsprop.h5

# Save the entire trained models
ResNet50_model_adam.save('models/ResNet50_model_adam.h5')
ResNet101_model_adam.save('models/ResNet101_model_adam.h5')
ResNet152_model_adam.save('models/ResNet152_mode_adaml.h5')
InceptionV3_model_adam.save('models/InceptionV3_model_adam.h5')

ResNet50_model_rmsprop.save('models/ResNet50_model_rmsprop.h5')
ResNet101_model_rmsprop.save('models/ResNet101_model_rmsprop.h5')
ResNet152_model_rmsprop.save('models/ResNet152_model_rmsprop.h5')
InceptionV3_model_rmsprop.save('models/InceptionV3_model_rmsprop.h5')
```

Load the saved models (if needed):

If the models have already been trained once and do not want to be trained again, they can be loaded here. This saves time if you want to view the results of the models again. For this,

only the lower lines in the block must be commented out.

In [ ]:
```
# Load the .h5 models
model_path_ResNet50_adam = 'models/ResNet50_model_adam.h5'
model_path_ResNet50_rmsprop = 'models/ResNet50_model_rmsprop.h5'
model_path_ResNet101_adam = 'models/ResNet101_model_adam.h5'
model_path_ResNet101_rmsprop = 'models/ResNet101_model_rmsprop.h5'
model_path_Resnet152_adam = 'models/Resnet152_model_adam.h5'
model_path_Resnet152_rmsprop = 'models/Resnet152_model_rmsprop.h5'
model_path_InceptionV3_adam = 'models/InceptionV3_model_adam.h5'
model_path_InceptionV3_rmsprop = 'models/InceptionV3_model_rmsprop.h5'

# ResNet50_model_adam = load_model(model_path_ResNet50_adam)
# ResNet50_model_rmsprop = load_model(model_path_ResNet50_rmsprop)
# ResNet101_model_adam = load_model(model_path_ResNet101_adam)
# ResNet101_model_rmsprop = load_model(model_path_ResNet101_rmsprop)
# ResNet152_model_adam = load_model(model_path_Resnet152_adam)
# ResNet152_model_rmsprop = load_model(model_path_Resnet152_rmsprop)
# InceptionV3_model_adam = load_model(model_path_InceptionV3_adam)
# InceptionV3_model_rmsprop = load_model(model_path_InceptionV3_rmsprop)
```

---

8. Evaluation

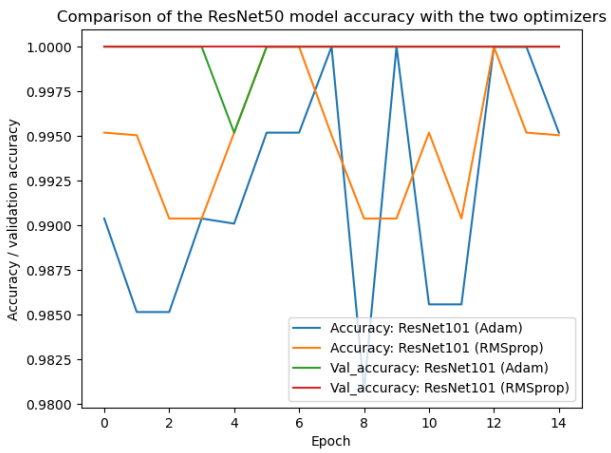
This chapter deals with the evaluation of the models, it will be discussed which of the two optimizers gives the better result and how good the model architectures are among each other.

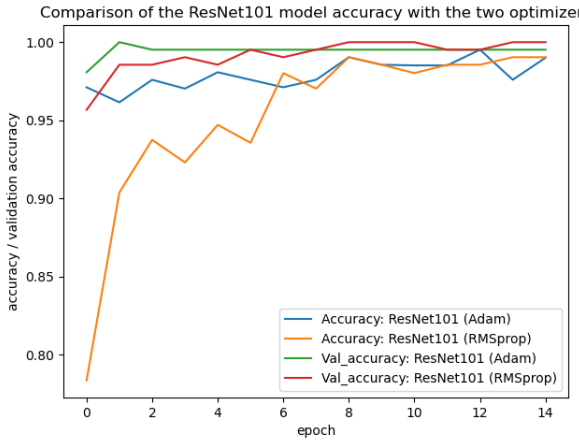
Comparison of the Accuracy and the Loss:
In the following block, the four models architectures are not directly compared with each other, but the respective model is compared with the two different optimizers. In order to get an overview which of the optimizers delivers a better result. For this purpose, the training accuracy and the validation accuacy of the respective models are displayed and compared in a plot. It should be mentioned that the validation accuracy is much more important for the comparison, but to get a feeling for the model it makes sense to look at the training accuracy as well. The second block for each model is the comparison of the trainings loss and the validation loss. Both plots are important to get an understanding of the model and to be able to make statements about the models.

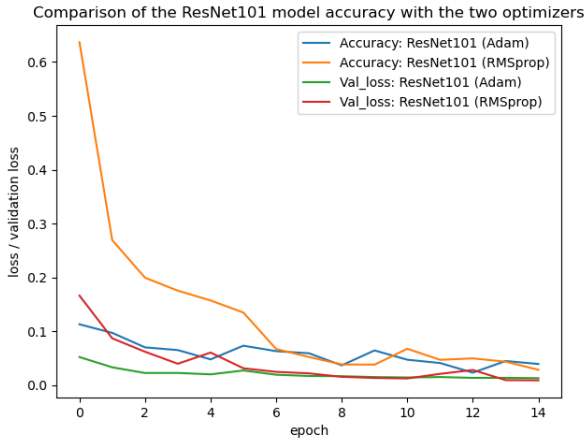
In [ ]:
```
# Comparison of the Accuracy of the individual models with adam and with RMSprop op

# ResNet50
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.plot(history_ResNet50_adam.history['accuracy'], label="ResNet50 (optimizer: Ada
plt.plot(history_ResNet50_rmsprop.history['accuracy'], label="ResNet50 (optimizer:
plt.plot(history_ResNet50_adam.history['val_accuracy'], label="ResNet50 (optimizer:
plt.plot(history_ResNet50_rmsprop.history['val_accuracy'], label="ResNet50 (optimiz
plt.title('Comparison of the ResNet50 model accuracy with the two optimizers')
```

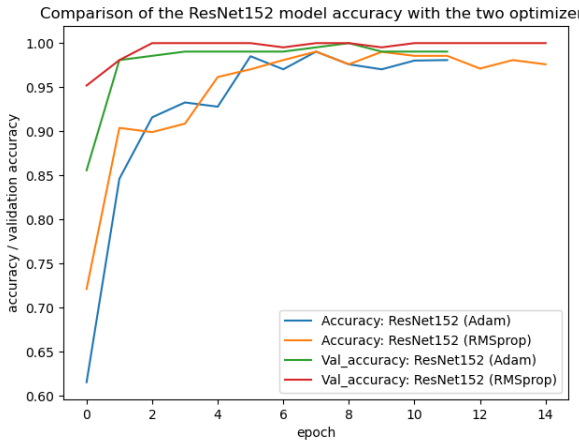
```
plt.ylabel('Accuracy / validation accuracy')
plt.xlabel('Epoch')
plt.legend(['Accuracy: ResNet101 (Adam)', 'Accuracy: ResNet101 (RMSprop)', 'Val_acc

plt.subplot(1, 2, 2)
plt.plot(history_ResNet50_adam.history['loss'], label="ResNet50 (optimizer: Adam)")
plt.plot(history_ResNet50_rmsprop.history['loss'], label="ResNet50 (optimizer: RMSp
plt.plot(history_ResNet50_adam.history['val_loss'], label="ResNet50 (optimizer: Ada
plt.plot(history_ResNet50_rmsprop.history['val_loss'], label="ResNet50 (optimizer:
plt.title('Comparison of the ResNet50 model loss with the two optimizers')
plt.ylabel('Loss / validation loss')
plt.xlabel('Epoch')
plt.legend(['Loss: ResNet50 (Adam)', 'Loss: ResNet50 (RMSprop)', 'Val_loss: ResNet5

plt.tight_layout(pad=5.0)
plt.show()

##################################################

# ResNet101
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.plot(history_ResNet101_adam.history['accuracy'], label = "ResNet101 (optimizer:
plt.plot(history_ResNet101_rmsprop.history['accuracy'], label = "ResNet101 (optimiz
plt.plot(history_ResNet101_adam.history['val_accuracy'], label = "ResNet101 (optimi
plt.plot(history_ResNet101_rmsprop.history['val_accuracy'], label = "ResNet101 (opt
plt.title('Comparison of the ResNet101 model accuracy with the two optimizers')
plt.ylabel('accuracy / validation accuracy')
plt.xlabel('epoch')
plt.legend(['Accuracy: ResNet101 (Adam)', 'Accuracy: ResNet101 (RMSprop)', 'Val_acc

plt.subplot(1, 2, 2)
plt.plot(history_ResNet101_adam.history['loss'], label = "ResNet101 (optimizer: Ada
plt.plot(history_ResNet101_rmsprop.history['loss'], label = "ResNet101 (optimizer:
plt.plot(history_ResNet101_adam.history['val_loss'], label = "ResNet101 (optimizer:
plt.plot(history_ResNet101_rmsprop.history['val_loss'], label = "ResNet101 (optimiz
plt.title('Comparison of the ResNet101 model accuracy with the two optimizers')
plt.ylabel('loss / validation loss')
plt.xlabel('epoch')
plt.legend(['Accuracy: ResNet101 (Adam)', 'Accuracy: ResNet101 (RMSprop)', 'Val_los
plt.tight_layout(pad=5.0)
plt.show()

##################################################

# ResNet152
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.plot(history_ResNet152_adam.history['accuracy'], label = "ResNet152 (optimizer:
plt.plot(history_ResNet152_rmsprop.history['accuracy'], label = "ResNet152 (optimiz
plt.plot(history_ResNet152_adam.history['val_accuracy'], label = "ResNet152 (optimi
plt.plot(history_ResNet152_rmsprop.history['val_accuracy'], label = "ResNet152 (opt
plt.title('Comparison of the ResNet152 model accuracy with the two optimizers')
plt.ylabel('accuracy / validation accuracy')
plt.xlabel('epoch')
plt.legend(['Accuracy: ResNet152 (Adam)', 'Accuracy: ResNet152 (RMSprop)', 'Val_acc
```

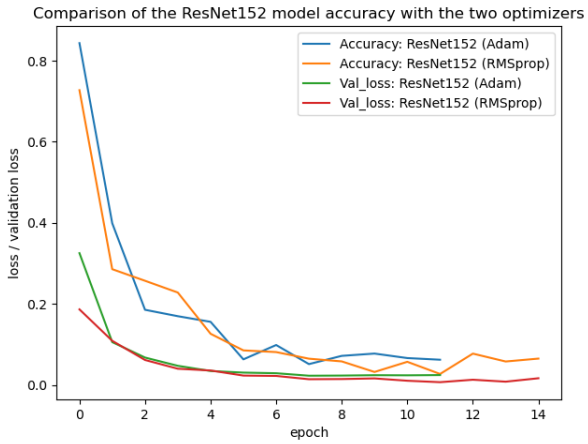
```
plt.subplot(1, 2, 2)
plt.plot(history_ResNet152_adam.history['loss'], label = "ResNet152 (optimizer: Ada
plt.plot(history_ResNet152_rmsprop.history['loss'], label = "ResNet152 (optimizer:
plt.plot(history_ResNet152_adam.history['val_loss'], label = "ResNet152 (optimizer:
plt.plot(history_ResNet152_rmsprop.history['val_loss'], label = "ResNet152 (optimiz
plt.title('Comparison of the ResNet152 model accuracy with the two optimizers')
plt.ylabel('loss / validation loss')
plt.xlabel('epoch')
plt.legend(['Accuracy: ResNet152 (Adam)', 'Accuracy: ResNet152 (RMSprop)', 'Val_los
plt.tight_layout(pad=5.0)
plt.show()

##################################################

# InceptionV3
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.plot(history_InceptionV3_adam.history['accuracy'], label = "InceptionV3 (optimi
plt.plot(history_InceptionV3_rmsprop.history['accuracy'], label = "InceptionV3 (opt
plt.plot(history_InceptionV3_adam.history['val_accuracy'], label = "InceptionV3 (op
plt.plot(history_InceptionV3_rmsprop.history['val_accuracy'], label = "InceptionV3
plt.title('Comparison of the InceptionV3 model accuracy with the two optimizers')
plt.ylabel('accuracy / validation accuracy')
plt.xlabel('epoch')
plt.legend(['Accuracy: InceptionV3 (Adam)', 'Accuracy: InceptionV3 (RMSprop)', 'Val

plt.subplot(1, 2, 2)
plt.plot(history_InceptionV3_adam.history['loss'], label = "InceptionV3 (optimizer:
plt.plot(history_InceptionV3_rmsprop.history['loss'], label = "InceptionV3 (optimiz
plt.plot(history_InceptionV3_adam.history['val_loss'], label = "InceptionV3 (optimi
plt.plot(history_InceptionV3_rmsprop.history['val_loss'], label = "InceptionV3 (opt
plt.title('Comparison of the InceptionV3 model accuracy with the two optimizers')
plt.ylabel('loss / validation loss')
plt.xlabel('epoch')
plt.legend(['Accuracy: InceptionV3 (Adam)', 'Accuracy: InceptionV3 (RMSprop)', 'Val
plt.tight_layout(pad=5.0)
plt.show()
```

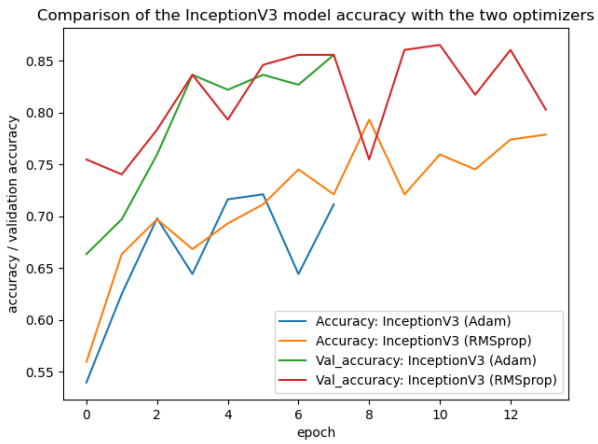












Comparison of the two optimizers:

In this section, the previously presented results are briefly discussed and explained. For this purpose, each model architecture is considered individually. A summary of the results is given at the end.

At the beginning you will notice that not all lines in the graph are the same length, this is not because there was a mistake, but because the training was stopped because of the early stopping criterion introduced earlier. Notably, the green and red lines, especially those representing the accuracies, play a pivotal role in comprehending the models' progression throughout the learning process. So that also with another training result is comprehensible about which graphs is spoken, in each case pictures are linked. They show the graphs to

which the text below refers.

ResNet50:
First, the performance of the ResNet50 model is analyzed.

- Upon observation of the figure, it becomes evident that the models using different optimizers exhibited minimal discrepancies. Although various fluctuations were observed during training, they were relatively small, resulting in nearly identical outcomes. RMSprop demonstrated a more stable accuracy, with fewer fluctuations compared to the Adam optimizer. This suggests that the weights were adjusted to a lesser extent in the RMSprop model.

- In terms of validation accuracy, the ResNet50 (Adam) model initially displayed better values. However, as training progressed, its performance declined, eventually being surpassed by the ResNet50 (RMSprop) model, which achieved better results. Both models, nevertheless, yielded highly satisfactory outcomes, approaching close to 100% accuracy.

- With respect to the trained epochs, it can be seen that the ResNet50 (RMSprop) was stopped somewhat earlier due to the early stopping procedure. This is done to avoid overfitting, early stopping monitors the performance of the model on a separate validation dataset during training. Training is stopped when performance on the validation dataset stops improving or even deteriorates. At this point, it is assumed that the model has reached the point where it can be best generalized and thus cannot be improved any further.

- Another good point is that the accuarcy and the valdiation accuarcy did not differ much. A small difference between the two metrics indicates that the model generalizes well and performs similarly well on unknown data as it does on training data.

- The loss as well as the validation loss of both models looks very good. The values have an almost permanent decrease, where there are only small fluctuations. Furthermore, the validation loss is as good as the loss, which indicates that the model has good generalization capabilities and is not overfitted. Similar good performance on the validation data compared to the training data indicates that the model is able to predict effectively on unknown data and is not just rote learning.

- (Notebook_images\ResNet50-accuracy-loss.PNG)

ResNet101:
The next model is the ResNet101

- Like with the ResNet50, the models develop very well over the epochs. However, it is noticeable that the valdation accuracy of the model with adam optimizer delivers worse results than the training accuracy.

- In addition, the ResNet101 (RMSprop) model delivers slightly better results, even if the difference is not particularly large.
- It can already be seen that the ResNet101 gives slightly worse results than the ResNet50, since the graphs do not come as close to the 100% Accuarcy.
- It is also noticeable that this time the ResNet101 (Adam) model was aborted a bit earlier during training. Therefore, no pattern can be seen with the optimizers yet.
- With the loss and the validation loss there is again only a small difference, which is good. However, there are more fluctuations during the training, but they subside again towards the end of the training. It should be mentioned that the fluctuation of the loss and the validation loss during the training can mean that the model has not yet converged stably and the training has not yet been completed. This should therefore not be underestimated.
- (Notebook_images\ResNet101-accuracy-loss.PNG)

ResNet152:

- Also for the ResNet152 model, the development of accuracy and loss is very similar to the previous models.
- There is again an earlier stopping of the training in the model with adam optimizer. But results at the end are again almost the same as with the ResNet152 (RMSprop) model.
- The model delivers very good results and also the loss and the valuation loss deliver only little fluctuations and are very close to each other, which speaks for a well-generalized model.
- (Notebook_images\ResNet152-accuracy-loss.PNG)

InceptionV3:
The last model is the InceptionV3

- The first major changes are seen in the InceptionV3 models. Here you can see that the inceptionV3 (Adam) model stops much earlier with the training. This model also delivers significantly better results than the model with RMSprop optimizer.
- It is also noticeable that the accuracy and the validation accuracy are significantly different. If the validation accuracy is significantly better than the training accuracy, this may indicate overfitting, erroneous data or problems with model complexity. To address this imbalance, measures such as reducing model complexity, applying regularization techniques, or improving training data quality should be considered. The goal is to obtain a model that performs well on both training data and new data and provides reliable predictions.

- Both models do not perform well, with accuracies ranging from about 70 to 90 percent. Compared to the other models, these are very poor results. This will also be noticeable when testing the models.
- Possible reasons why the InceptionV3 models perform so poorly include random initialization of model weights, inappropriate model architecture, incorrectly chosen hyperparameters, susceptibility to overfitting, or faulty implementation.
- (Notebook_images\InceptionV3-accuracy-loss.PNG)

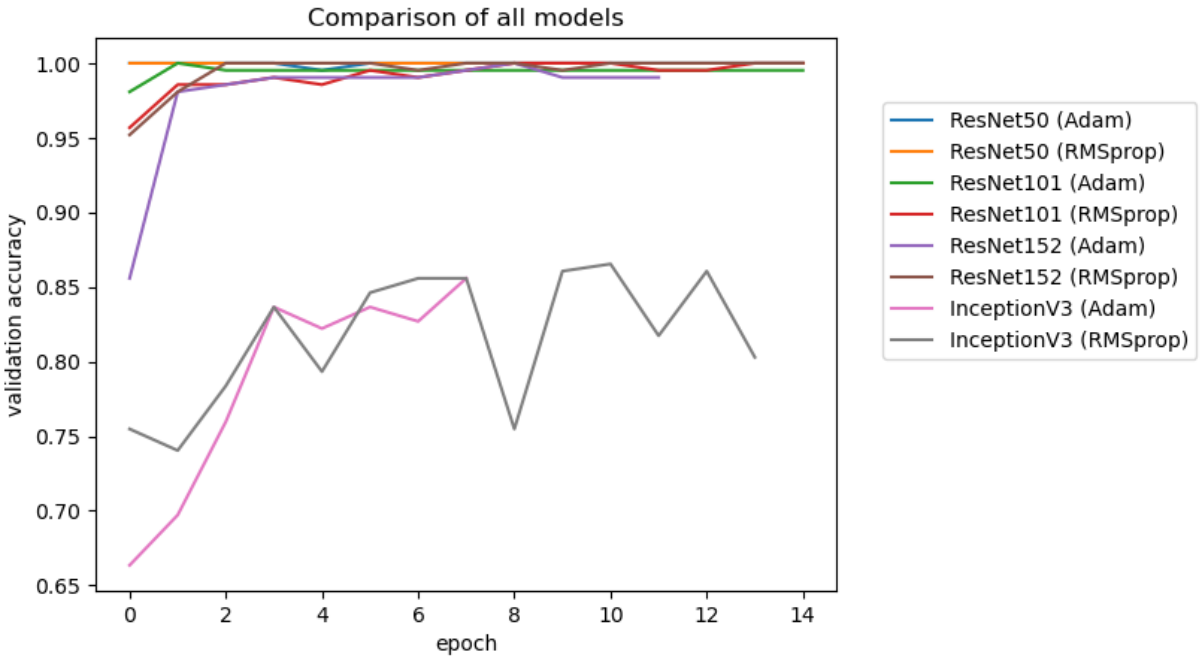
Summary and conclusion:

- Of the eight models shown here, almost all deliver very good results, only the two InceptionV3 models deliver less good results.
- To come back to the comparison between the optimizers, here you can often see only slight differences. It can be said that the Adam optimizer was more often interrupted earlier by the early stopping. There is also a very small difference in performance, where the models with RMSprop optimizer perform slightly better. For the further course, however, all models are used, since the performance difference between the optimizers is simply too small to play a role. The InceptionV3 models will also be used to see if the current results are mirrored in testing.

Comparison of the four model architectures:

In the following, the four model architectures are compared. These include ResNet50, ResNet101, ResNet152 and InceptionV3. For each architecture there are two models that are compared, one with Adam and the other with RMSprop optimizer.

In [ ]:
```
#  "Accuracy"
plt.plot(history_ResNet50_adam.history['val_accuracy'], label = "ResNet50 (optimize
plt.plot(history_ResNet50_rmsprop.history['val_accuracy'], label = "ResNet50 (optim
plt.plot(history_ResNet101_adam.history['val_accuracy'], label = "ResNet101 (optimi
plt.plot(history_ResNet101_rmsprop.history['val_accuracy'], label = "ResNet101 (opt
plt.plot(history_ResNet152_adam.history['val_accuracy'], label = "ResNet152 (optimi
plt.plot(history_ResNet152_rmsprop.history['val_accuracy'], label = "ResNet152 (opt
plt.plot(history_InceptionV3_adam.history['val_accuracy'], label = "InceptionV3 (op
plt.plot(history_InceptionV3_rmsprop.history['val_accuracy'], label = "InceptionV3
plt.title('Comparison of all models')
plt.ylabel('validation accuracy')
plt.xlabel('epoch')
plt.legend(['ResNet50 (Adam)', 'ResNet50 (RMSprop)', 'ResNet101 (Adam)', 'ResNet101
plt.show()
```



Compare all models with each other:

As previously observed, most models exhibit very similar performance. There are only two significant outliers, which are the Inception models, as mentioned earlier.

The best-performing models in this scenario are those that use the RMSprop optimizer. However, no model stands out significantly, as the differences are minimal. In this run, the ResNet152 (RMSprop) and ResNet50 (RMSprop) achieved the highest accuracies on the validation dataset.

It should be noted that the results may vary slightly depending on the run, especially when the performance differences are so small.

Preparing the test images:

In the following block, the test images of the two classes are loaded from the directory. They are normalized and resized directly to have the same format as the other images. A label with "Dominik" or "Unknown" is then created for the respective images. To bring some varraiation into the dataset the images are shuffled, the same happens with the labels so that they still fit to the images.

In [ ]:
```
# Load and prepare the test images
test_path_dominik = 'data/test/Dominik'
test_path_unknown = 'data/test/unknown'

def getImages(test_image_folder):
    test_images = []
    for image_path in os.listdir(test_image_folder):
        image = Image.open(os.path.join(test_image_folder, image_path))
        temp_data = img_to_array(image)
        img_norm = cv2.normalize(temp_data, None, alpha=0, beta=1, norm_type=cv2.NO
```

```
        image_norm = array_to_img(img_norm)

        image = image_norm.resize((224, 224))  # Adjust the image size according to
        image = np.array(image)# / 255.0  # Normalize the image pixels to the range
        test_images.append(image)

    # Convert the list of images to a Numpy array
    test_images = np.array(test_images)
    return test_images

test_images_dominik = getImages(test_path_dominik)
test_images_unknown = getImages(test_path_unknown)

test_labels_dominik = np.array(['Dominik'] * len(test_images_dominik))
test_labels_unknown = np.array(['Unknown'] * len(test_images_unknown))


all_test_labels = np.concatenate((test_labels_dominik, test_labels_unknown))
all_test_images = np.concatenate((test_images_dominik, test_images_unknown))

# Get the indices for shuffling
indices = np.arange(len(all_test_labels))

# Shuffle the indices
np.random.shuffle(indices)

# Shuffle the arrays using the shuffled indices
shuffled_labels = all_test_labels[indices]
shuffled_images = all_test_images[indices]
```

Apply models to test data:

Now that the test data is available, the models can be applied to the test data. For this purpose, the labels are converted to the one-hot encoded format. With the function evaluate() the models can be tested. The function returns the test_loss and the test_accuracy. The test_accuracy is particularly important, because it indicates how well the model performs on images that have never been seen before. This value should be as high as possible to get a good result. The results of this test are compared in the next block with the other values in a table.

In [ ]:
```
# Convert labels to numerical values
label_encoder = LabelEncoder()
numerical_labels_dom = label_encoder.fit_transform(shuffled_labels)

# Convert numerical labels to one-hot encoded format
one_hot_labels_dom = to_categorical(numerical_labels_dom, num_classes=2)

# Evaluate the model on the test dataset
ResNet50_adam_test_loss, ResNet50_adam_test_accuracy = ResNet50_model_adam.evaluate
ResNet101_adam_test_loss, ResNet101_adam_test_accuracy = ResNet101_model_adam.evalu
ResNet152_adam_test_loss, ResNet152_adam_test_accuracy = ResNet152_model_adam.evalu
InceptionV3_adam_test_loss, InceptionV3_adam_test_accuracy = InceptionV3_model_adam

ResNet50_rmsprop_test_loss, ResNet50_rmsprop_test_accuracy = ResNet50_model_rmsprop
```

```
ResNet101_rmsprop_test_loss, ResNet101_rmsprop_test_accuracy = ResNet101_model_rmsp
ResNet152_rmsprop_test_loss, ResNet152_rmsprop_test_accuracy = ResNet152_model_rmsp
InceptionV3_rmsprop_test_loss, InceptionV3_rmsprop_test_accuracy = InceptionV3_mode
```

```
5/5 [==============================] - 3s 523ms/step - loss: 0.0524 - accuracy: 0.98
61
5/5 [==============================] - 5s 960ms/step - loss: 0.0774 - accuracy: 0.98
61
5/5 [==============================] - 7s 1s/step - loss: 0.0657 - accuracy: 0.9861
5/5 [==============================] - 2s 348ms/step - loss: 2.9224 - accuracy: 0.80
56
5/5 [==============================] - 2s 482ms/step - loss: 0.0727 - accuracy: 0.98
61
5/5 [==============================] - 5s 909ms/step - loss: 0.1179 - accuracy: 0.97
22
5/5 [==============================] - 7s 1s/step - loss: 0.0329 - accuracy: 0.9861
5/5 [==============================] - 2s 351ms/step - loss: 7.8372 - accuracy: 0.75
00
```

Comparison of the models based on the obtained values:

In order to better display the collected values, a table is created below. For this, the last loss and accuracy value of the training is passed. Also the validation loss and the validation accuracy. In addition, the previously selected test loss and the test accuracy. However, the validation accuracy and especially the test accuracy are decisive here. The other values have only been taken over out of interest.

In [ ]:
```python
import pandas as pd

# Liste der Modellnamen
model_names = ['ResNet50', 'ResNet101', 'ResNet152', 'InceptionV3']

# Liste der Optimierer
optimizers = ['adam', 'rmsprop']

# Daten sammeln
data = []

# Funktion, um den letzten Wert einer Metrik für ein gegebenes Modell und Optimiere
def get_last_metric_value(history, metric_name):
    last_epoch = len(history.history[metric_name]) - 1
    last_value = history.history[metric_name][last_epoch]
    return last_value

# Schleife über alle Modelle und Optimierer
for model_name in model_names:
    for optimizer in optimizers:
        # Erstellen Sie den passenden Schlüssel für den Zugriff auf den Trainingsve
        history_key = f'history_{model_name}_{optimizer}'

        # Den letzten Loss-Wert ausgeben
        last_loss = get_last_metric_value(eval(history_key), 'loss')

        # Den letzten Validierung-Loss-Wert ausgeben
        last_val_loss = get_last_metric_value(eval(history_key), 'val_loss')
```

```
        # Den letzten Accuracy-Wert ausgeben
        last_accuracy = get_last_metric_value(eval(history_key), 'accuracy')

        # Den letzten Validierung-Accuracy-Wert ausgeben
        last_val_accuracy = get_last_metric_value(eval(history_key), 'val_accuracy'

        # Testverlust und Testgenauigkeit für jedes Modell abrufen
        test_loss_key = f'{model_name}_{optimizer}_test_loss'
        test_accuracy_key = f'{model_name}_{optimizer}_test_accuracy'

        # Daten für die Tabelle sammeln
        data.append([model_name, optimizer, last_loss, last_val_loss, eval(test_los
                     eval(test_accuracy_key)])

# Erstellen der DataFrame-Tabelle
df = pd.DataFrame(data, columns=['Model', 'Optimizer', 'Last Loss', 'Last Validatio
# Tabellarische Darstellung
print(df)
```

```
        Model Optimizer  Last Loss  Last Validation Loss  Test Loss  \
0    ResNet50      adam   0.014653              0.005864   0.052355
1    ResNet50   rmsprop   0.013103              0.002741   0.072729
2   ResNet101      adam   0.039311              0.012979   0.077384
3   ResNet101   rmsprop   0.028483              0.008923   0.117923
4   ResNet152      adam   0.062166              0.024298   0.065717
5   ResNet152   rmsprop   0.064939              0.016426   0.032908
6  InceptionV3      adam   7.108573              3.426212   2.922408
7  InceptionV3   rmsprop   6.185216              8.246324   7.837189

   Last Accuracy  Last Validation Accuracy  Test Accuracy
0       0.995192                  1.000000       0.986111
1       0.995049                  1.000000       0.986111
2       0.990099                  0.995192       0.986111
3       0.990385                  1.000000       0.972222
4       0.980769                  0.990385       0.986111
5       0.975962                  1.000000       0.986111
6       0.711538                  0.855769       0.805556
7       0.778846                  0.802885       0.750000
```

Analyzing the obtained values:

In the previous charts, it was indeed possible to analyze how the model developed over different epochs. However, it is not straightforward to read precise data in the decimal places. Therefore, the actual values will be compared again here for a more accurate assessment.

It is evident from the analysis that the previous statement is confirmed – both the ResNet50 and ResNet152 models with the RMSprop optimizer deliver the best results. This is evident from the fact that these models correctly labeled all test images, which is highly impressive. Additionally, the same models with the Adam optimizer also yield equally good results. The six ResNet models had validation accuracies ranging from 97.5% to 99%, and these excellent results are reflected in the test data as well.

However, as observed before, the InceptionV3 models only achieve an accuracy of 82% and 85% on the test data. This is a rather poor outcome, considering there are only two classes to be predicted from.

---

9. Testing

In this chapter, the test dataset is again transferred to the models, but this time not only the numerical values are output, but the images with the actual labels. This makes it possible to see better how well the models actually performed.

To do this, first the predict() function is used which takes a dataset as input and the result is a matrix or numpy array containing the predictions of the model for each input image in the dataset. This procedure is performed for each model

In [ ]:
```
# Get the Results of the models predicting the testing images
predictions_dominik_ResNet50_model_adam = ResNet50_model_adam.predict(test_images_d
predictions_dominik_ResNet50_model_rmsprop = ResNet50_model_rmsprop.predict(test_im

predictions_unknown_ResNet50_model_adam = ResNet50_model_adam.predict(test_images_u
predictions_unknown_ResNet50_model_rmsprop = ResNet50_model_rmsprop.predict(test_im

predictions_dominik_ResNet101_model_adam = ResNet101_model_adam.predict(test_images
predictions_dominik_ResNet101_model_rmsprop = ResNet101_model_rmsprop.predict(test_

predictions_unknown_ResNet101_model_adam = ResNet101_model_adam.predict(test_images
predictions_unknown_ResNet101_model_rmsprop = ResNet101_model_rmsprop.predict(test_

predictions_dominik_ResNet152_model_adam = ResNet152_model_adam.predict(test_images
predictions_dominik_ResNet152_model_rmsprop = ResNet152_model_rmsprop.predict(test_

predictions_unknown_ResNet152_model__adam = ResNet152_model_adam.predict(test_image
predictions_unknown_ResNet152_model_rmsprop = ResNet152_model_rmsprop.predict(test_

predictions_dominik_InceptionV3_model_adam = InceptionV3_model_adam.predict(test_im
predictions_dominik_InceptionV3_model_rmsprop = InceptionV3_model_rmsprop.predict(t

predictions_unknown_InceptionV3_model_adam = InceptionV3_model_adam.predict(test_im
predictions_unknown_InceptionV3_model_rmsprop = InceptionV3_model_rmsprop.predict(t
```

```
WARNING:tensorflow:5 out of the last 37 calls to <function Model.make_predict_functi
on.<locals>.predict_function at 0x000001847EC62DC0> triggered tf.function retracing.
Tracing is expensive and the excessive number of tracings could be due to (1) creati
ng @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3)
passing Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option
that relaxes argument shapes that can avoid unnecessary retracing. For (3), please r
efer to https://www.tensorflow.org/tutorials/customization/performance#python_or_ten
sor_args and https://www.tensorflow.org/api_docs/python/tf/function for  more detail
s.
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_predict_functi
on.<locals>.predict_function at 0x00000184884DC3A0> triggered tf.function retracing.
Tracing is expensive and the excessive number of tracings could be due to (1) creati
ng @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3)
passing Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option
that relaxes argument shapes that can avoid unnecessary retracing. For (3), please r
efer to https://www.tensorflow.org/tutorials/customization/performance#python_or_ten
sor_args and https://www.tensorflow.org/api_docs/python/tf/function for  more detail
s.
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make_predict_functi
on.<locals>.predict_function at 0x00000184694BBAF0> triggered tf.function retracing.
Tracing is expensive and the excessive number of tracings could be due to (1) creati
ng @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3)
passing Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option
that relaxes argument shapes that can avoid unnecessary retracing. For (3), please r
efer to https://www.tensorflow.org/tutorials/customization/performance#python_or_ten
sor_args and https://www.tensorflow.org/api_docs/python/tf/function for  more detail
s.
```

In the next step, the previously determined predictions are determined with a selectively created function. Since the predictions are specified in numerical values, it is not so clear which class actually belongs to the label. Therefore we look in the function which class has the higher probability, the class with the higher probability is then assigned to the image. Thus there is then for each model a list with labels for the respective images. The images of the two classes are intentionally not merged into one dataset, so that in the following it can be seen how well the model performs on the respective class. This makes it easier to judge which class the model has more difficulties with.

In [ ]:
```python
# Function to get labels from predictions
def getLabel_from_prediction(predicitons):
    labels = []
    for i in range(len(predicitons)):
        if predicitons[i][0] > predicitons[i][1]:
            labels.append("Dominik")
        else:
            labels.append("Unknown")
    return labels

labels_dominik_ResNet50_adam = getLabel_from_prediction(predictions_dominik_ResNet5
labels_dominik_ResNet50_rmsprop = getLabel_from_prediction(predictions_dominik_ResN

labels_unknown_ResNet50_adam = getLabel_from_prediction(predictions_unknown_ResNet5
```

```
labels_unknown_ResNet50_rmsprop = getLabel_from_prediction(predictions_unknown_ResN

labels_dominik_ResNet101_adam = getLabel_from_prediction(predictions_dominik_ResNet
labels_dominik_ResNet101_rmsprop = getLabel_from_prediction(predictions_dominik_Res

labels_unknown_ResNet101_adam = getLabel_from_prediction(predictions_unknown_ResNet
labels_unknown_ResNet101_rmsprop = getLabel_from_prediction(predictions_unknown_Res

labels_dominik_ResNet152_adam = getLabel_from_prediction(predictions_dominik_ResNet
labels_dominik_ResNet152_rmsprop = getLabel_from_prediction(predictions_dominik_Res

labels_unknown_ResNet152_adam = getLabel_from_prediction(predictions_unknown_ResNet
labels_unknown_ResNet152_rmsprop = getLabel_from_prediction(predictions_unknown_Res

labels_dominik_InceptionV3_adam = getLabel_from_prediction(predictions_dominik_Ince
labels_dominik_InceptionV3_rmsprop = getLabel_from_prediction(predictions_dominik_I

labels_unknown_InceptionV3_adam = getLabel_from_prediction(predictions_unknown_Ince
labels_unknown_InceptionV3_rmsprop = getLabel_from_prediction(predictions_unknown_I
```

Show testing results:

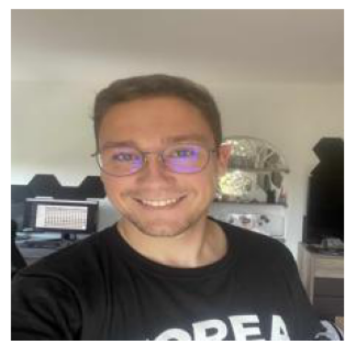
In the following section, the images from the test dataset are displayed. In addition, the labels created by the respective models for the individual images are also displayed. This makes it possible to see when a model has incorrectly labeled an image. Initially, only the images from the Dominik class are considered.

In [ ]:
```
images_dominik = []
for i in range(len(test_images_dominik)):
    image = array_to_img(test_images_dominik[i])
    images_dominik.append(image)


for i in range(0, 30,3):
    # Create a figure and axes
    fig, ax = plt.subplots(1,3, figsize=(12, 5))

    # Display the image
    ax[0].imshow(images_dominik[i])
    ax[1].imshow(images_dominik[i+1])
    ax[2].imshow(images_dominik[i+2])

    # Add a legend
    legend_text = ""
    ax[0].text(10, 10, legend_text, color='white', backgroundcolor='black')
    ax[1].text(10, 10, legend_text, color='white', backgroundcolor='black')
    ax[2].text(10, 10, legend_text, color='white', backgroundcolor='black')
    # ax[2].text(10, 10, legend_text, color='white', backgroundcolor='black')

    # predictions

    text_ResNet50_adam = "ResNet50 (Adam): " + labels_dominik_ResNet50_adam[i]
    text_ResNet50_rmsprop = "ResNet50 (RMSprop): " + labels_dominik_ResNet50_rmspro
    text_ResNet101_adam = "ResNet101 (Adam): " + labels_dominik_ResNet101_adam[i]
```

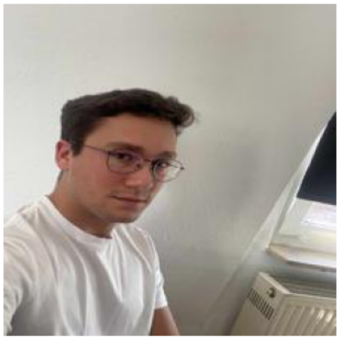
```
    text_ResNet101_rmsprop = "ResNet101 (RMSprop) " + labels_dominik_ResNet101_rmsp
    text_ResNet152_adam = "ResNet152 (Adam): " + labels_dominik_ResNet152_adam[i]
    text_ResNet152_rmsprop = "ResNet152 (RMSprop): " + labels_dominik_ResNet152_rms
    text_InceptionV3_adam = "InceptionV3 (Adam): " + labels_dominik_InceptionV3_ada
    text_InceptionV3_rmsprop = "InceptionV3 (RMSprop): " + labels_dominik_Inception


    # Add text under the image
    text_lines = ["Predictions: ", text_ResNet50_adam, text_ResNet50_rmsprop, text_
    text_position = (image.width // 2, image.height + 20)
    for line in text_lines:
        ax[0].text(*text_position, line, color='black', ha='center')
        ax[1].text(*text_position, line, color='black', ha='center')
        ax[2].text(*text_position, line, color='black', ha='center')
        text_position = (text_position[0], text_position[1] + 20)


    # Remove the axis labels
    ax[0].axis("off")
    ax[1].axis("off")
    ax[2].axis("off")

    # Adjust the layout
    plt.tight_layout()
    plt.show()
```

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



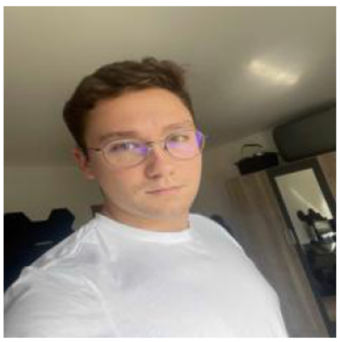
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

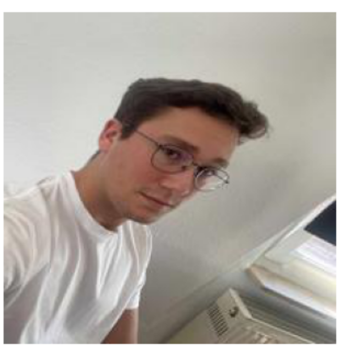
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



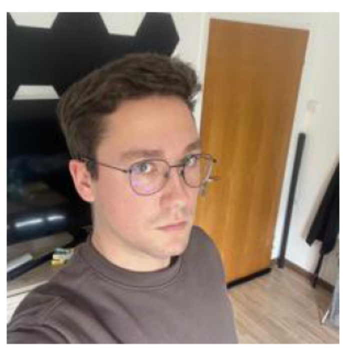
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

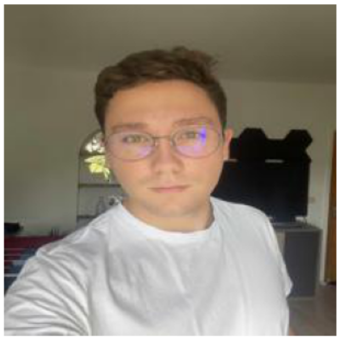


Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

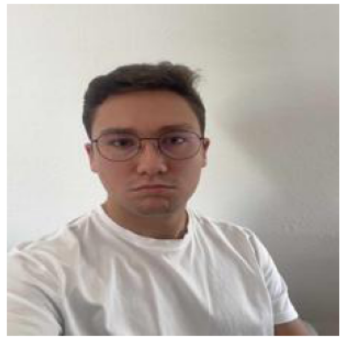


Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

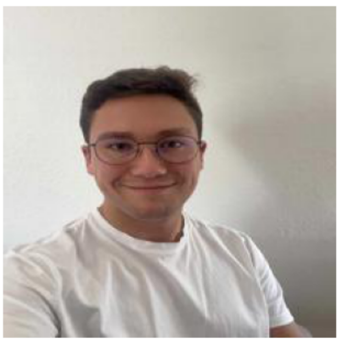


Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

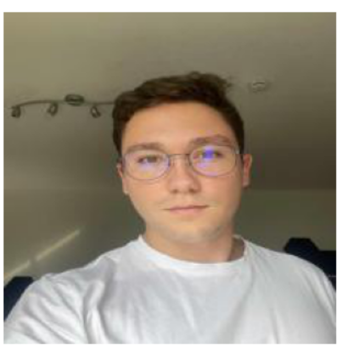
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



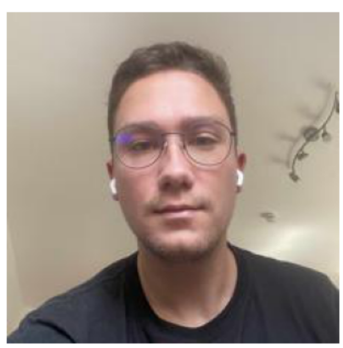
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

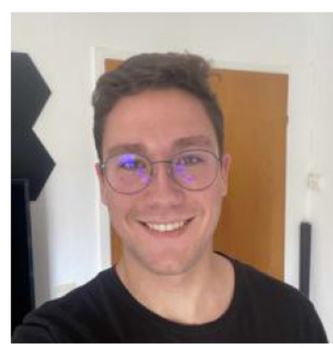
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

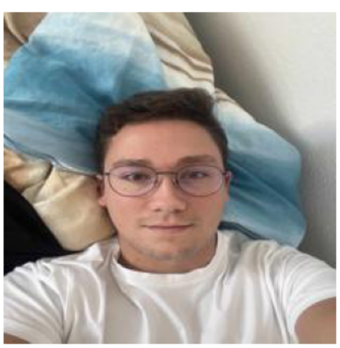
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

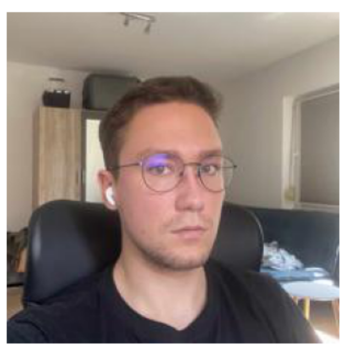
Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Dominik
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik



Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Dominik
ResNet50 (RMSprop): Dominik
ResNet101 (Adam): Dominik
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Dominik
ResNet152 (RMSprop): Dominik
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Show testing results:

Here, as before, the images are shown with the corresponding laben of the models. This time, however, for the Unknown Test images.

In [ ]:
```
images_unknown = []
for i in range(len(test_images_unknown)):
    image = array_to_img(test_images_unknown[i])
    images_unknown.append(image)


for i in range(0, 24,3):
    # Create a figure and axes
    fig, ax = plt.subplots(1,3, figsize=(12, 5))

    # Display the image
    ax[0].imshow(images_unknown[i])
    ax[1].imshow(images_unknown[i+1])
    ax[2].imshow(images_unknown[i+2])

    # Add a legend
    legend_text = ""
    ax[0].text(10, 10, legend_text, color='white', backgroundcolor='black')
    ax[1].text(10, 10, legend_text, color='white', backgroundcolor='black')
    ax[2].text(10, 10, legend_text, color='white', backgroundcolor='black')

    # predictions
   # predictions
    text_ResNet50_adam = "ResNet50 (Adam): " + labels_unknown_ResNet50_adam[i]
    text_ResNet50_rmsprop = "ResNet50 (RMSprop): " + labels_unknown_ResNet50_rmspro
    text_ResNet101_adam = "ResNet101 (Adam): " + labels_unknown_ResNet101_adam[i]
    text_ResNet101_rmsprop = "ResNet101 (RMSprop) " + labels_unknown_ResNet101_rmsp
    text_ResNet152_adam = "ResNet152 (Adam): " + labels_unknown_ResNet152_adam[i]
    text_ResNet152_rmsprop = "ResNet152 (RMSprop): " + labels_unknown_ResNet152_rms
    text_InceptionV3_adam = "InceptionV3 (Adam): " + labels_unknown_InceptionV3_ada
    text_InceptionV3_rmsprop = "InceptionV3 (RMSprop): " + labels_unknown_Inception


    # Add text under the image
    text_lines = ["Predictions: ", text_ResNet50_adam, text_ResNet50_rmsprop, text_
    text_position = (image.width // 2, image.height + 20)
    for line in text_lines:
        ax[0].text(*text_position, line, color='black', ha='center')
        ax[1].text(*text_position, line, color='black', ha='center')
        ax[2].text(*text_position, line, color='black', ha='center')
        text_position = (text_position[0], text_position[1] + 20)


    # Remove the axis labels
    ax[0].axis("off")
    ax[1].axis("off")
    ax[2].axis("off")

    # Adjust the layout
    plt.tight_layout()
```

```
# Show the plot
plt.show()
```

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Dominik
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Predictions:
ResNet50 (Adam): Unknown
ResNet50 (RMSprop): Unknown
ResNet101 (Adam): Unknown
ResNet101 (RMSprop) Unknown
ResNet152 (Adam): Unknown
ResNet152 (RMSprop): Unknown
InceptionV3 (Adam): Unknown
InceptionV3 (RMSprop): Dominik

Evaluation of the test results on the test dataset:

Based on the concrete example provided, it can now be confidently stated that the six ResNet models perform exceptionally well. They demonstrate almost perfect performance, with only one or two mislabeled images observed for the ResNet101 models. This indicates their high accuracy and reliability in predicting labels.

On the other hand, the Inception models show significantly lower performance. While they still manage to correctly label most of Dominik's images, they struggle with unknown images, mislabeling approximately half of them. This is a very poor result, as it is not much better than random guessing.

Considering this, it does not make sense to use the Inception models for any practical applications, given their inadequate performance on unknown images. In contrast, the ResNet models have proven to be more robust and suitable for various tasks.

---

10. Discussion of the results

After evaluating the models, let's discuss the results briefly. The remarkable performance of the ResNet models is truly impressive, and it appears that the combination of Dropout, Early Stopping, and Data Augmentation has proven to be highly effective. On the other hand, the poor results of the InceptionV3 model are quite surprising. It struggled significantly to handle the data, possibly because its pre-trained weights were not well adapted to the specific dataset.

Evaluating these different models has provided valuable insights into the most effective techniques and architectures for this specific application. It highlights the significance of continually optimizing models to enhance their performance and achieve better results in the future.

In the beginning, two optimizers were defined for comparison since it was challenging to decide which one to use. However, it turned out to be unnecessary as the models performed almost identically, regardless of the chosen optimizer. The differences in performance between the two optimizers are so minimal that both can be confidently used for these models. This suggests that the model architecture and other hyperparameters have a more significant impact on performance than the optimizer choice. As a result, either optimizer can be employed with confidence for these specific models.

---

11. Live face recognition

This chapter is a small extra, which implements a live face recognition mode. All of the

previously trained models can be used for this, but it must be decided in advance which of the models will be used and which must be commented out. It should be ensured that the remaining models are commented, so that a wrong model is not used by mistake. In addition, it must be said that this mode can only be used if a camera is available on the computer.

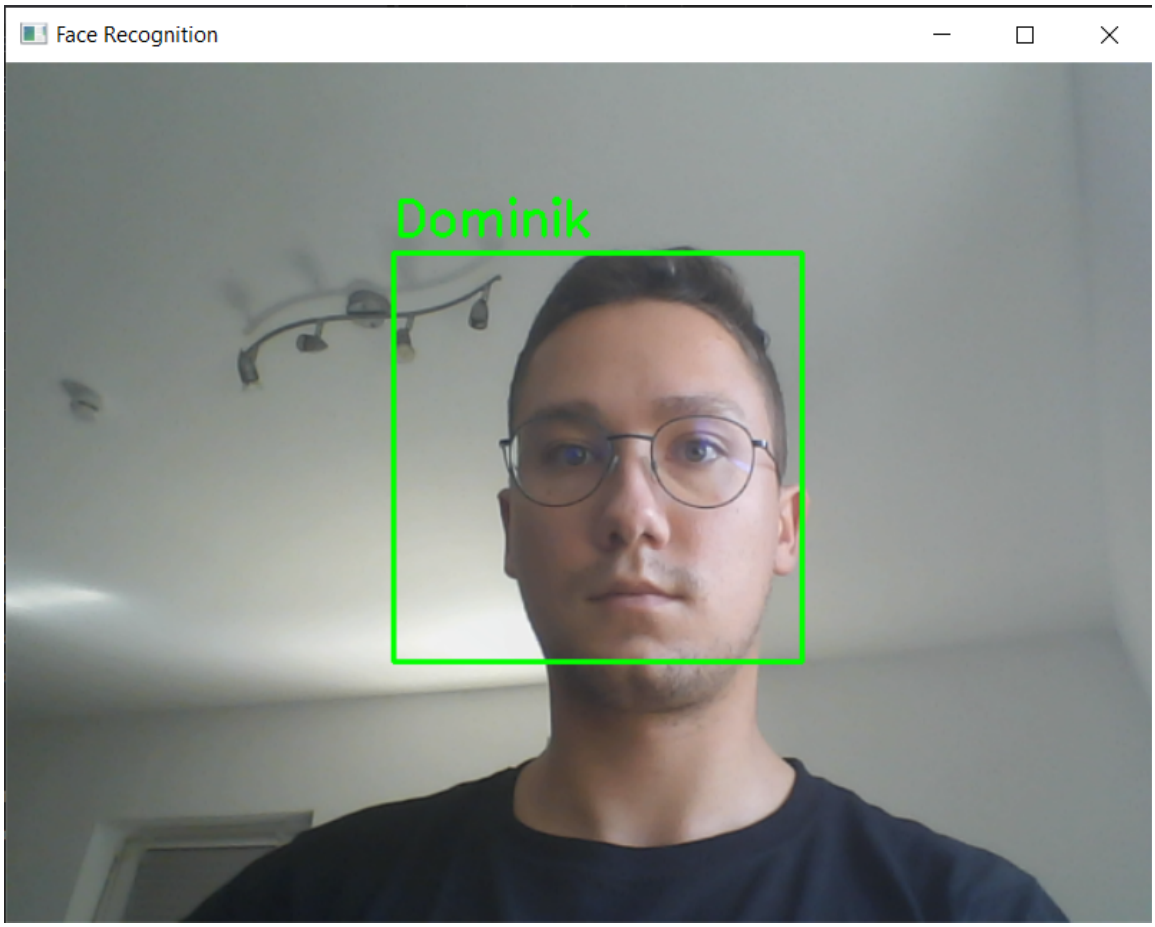
In the live mode, images are captured at very short intervals and passed to the function. These images are then analyzed using the haarcascade_frontalface_default to detect faces. When a face is recognized, it is marked with a green rectangle, and the area containing the face is cropped and sent to the trained model for further analysis. The model then labels the cropped face, and the label is returned. The live representation of the camera displays the labeled face with the rectangular box around it, providing real-time feedback on face recognition. This process enables the model to continuously analyze and label faces in the live video feed.

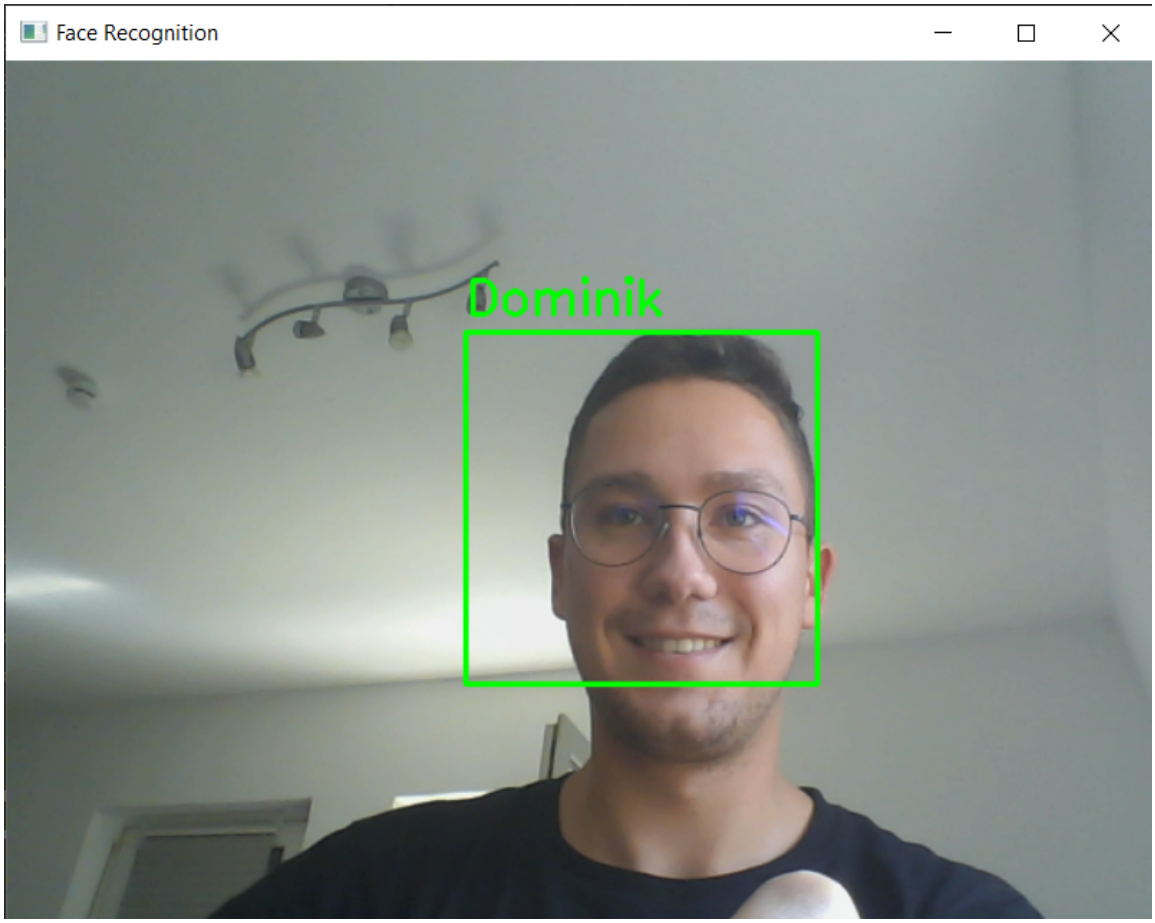
This is a great mode to see the self-trained model in action.

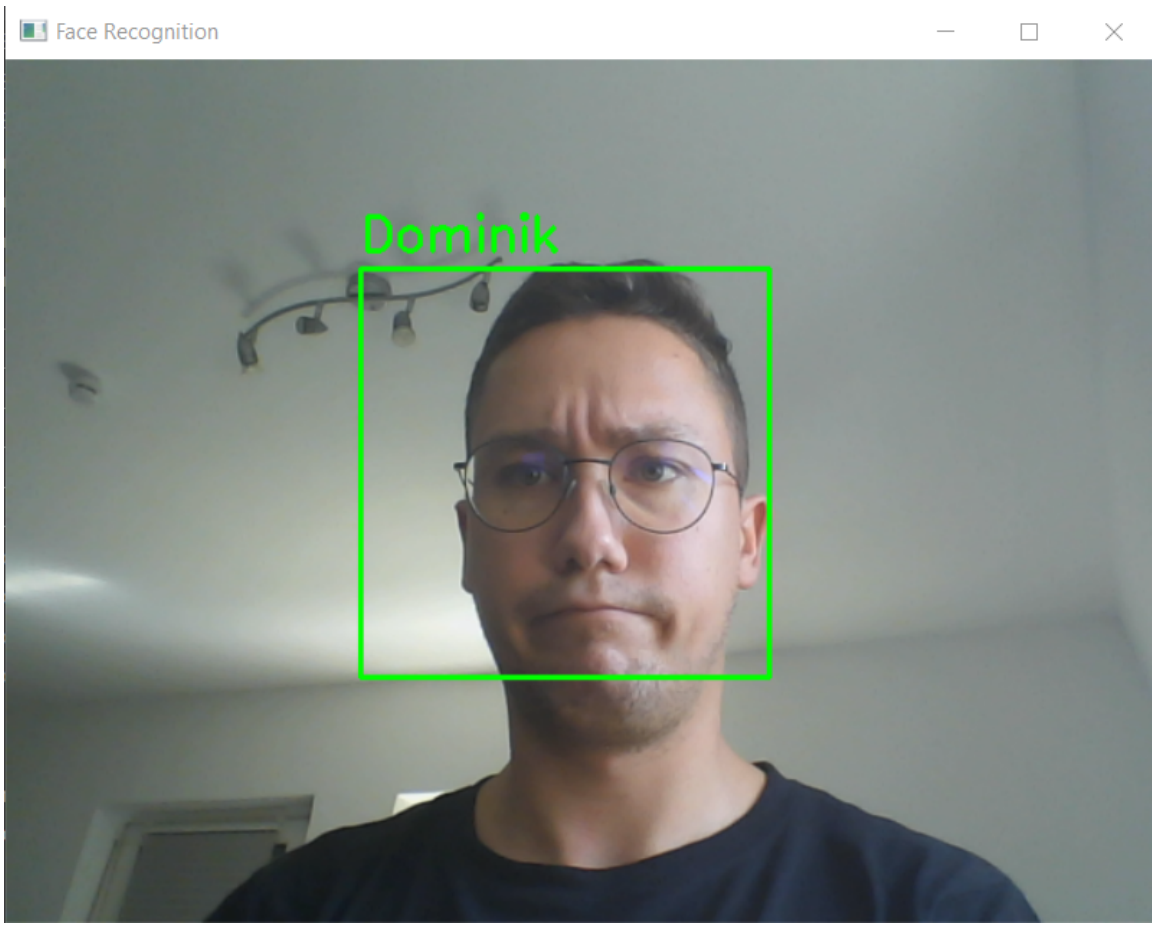
In [ ]:
```
# Paths to the face recognition model and cascade files

# Load the .h5 models
model_path_ResNet50_adam = 'models/ResNet50_model_adam.h5'
model_path_ResNet50_rmsprop = 'models/ResNet50_model_rmsprop.h5'
model_path_ResNet101_adam = 'models/ResNet101_model_adam.h5'
model_path_ResNet101_rmsprop = 'models/ResNet101_model_rmsprop.h5'
model_path_Resnet152_adam = 'models/Resnet152_model_adam.h5'
model_path_Resnet152_rmsprop = 'models/Resnet152_model_rmsprop.h5'
model_path_InceptionV3_adam = 'models/InceptionV3_model_adam.h5'
model_path_InceptionV3_rmsprop = 'models/InceptionV3_model_rmsprop.h5'


# Select one of the eight pre-trained models, while keeping the remaining models co
################################################################################
# model = load_model(model_path_ResNet50_adam)
# model = load_model(model_path_ResNet50_rmsprop)
# model = load_model(model_path_ResNet101_adam)
# model = load_model(model_path_ResNet101_rmsprop)
model = load_model(model_path_Resnet152_adam)
# model = load_model(model_path_Resnet152_rmsprop)
# model = load_model(model_path_InceptionV3_adam)
# model = load_model(model_path_InceptionV3_rmsprop)
################################################################################

# Paths to the cascade file
face_cascade_path = cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'
# Create cascade classifier for face detection
face_cascade = cv2.CascadeClassifier(face_cascade_path)

classes = { 0:'Dominik',
            1:'Unknown'}

# Function to detect faces
def detect_faces(image):
```

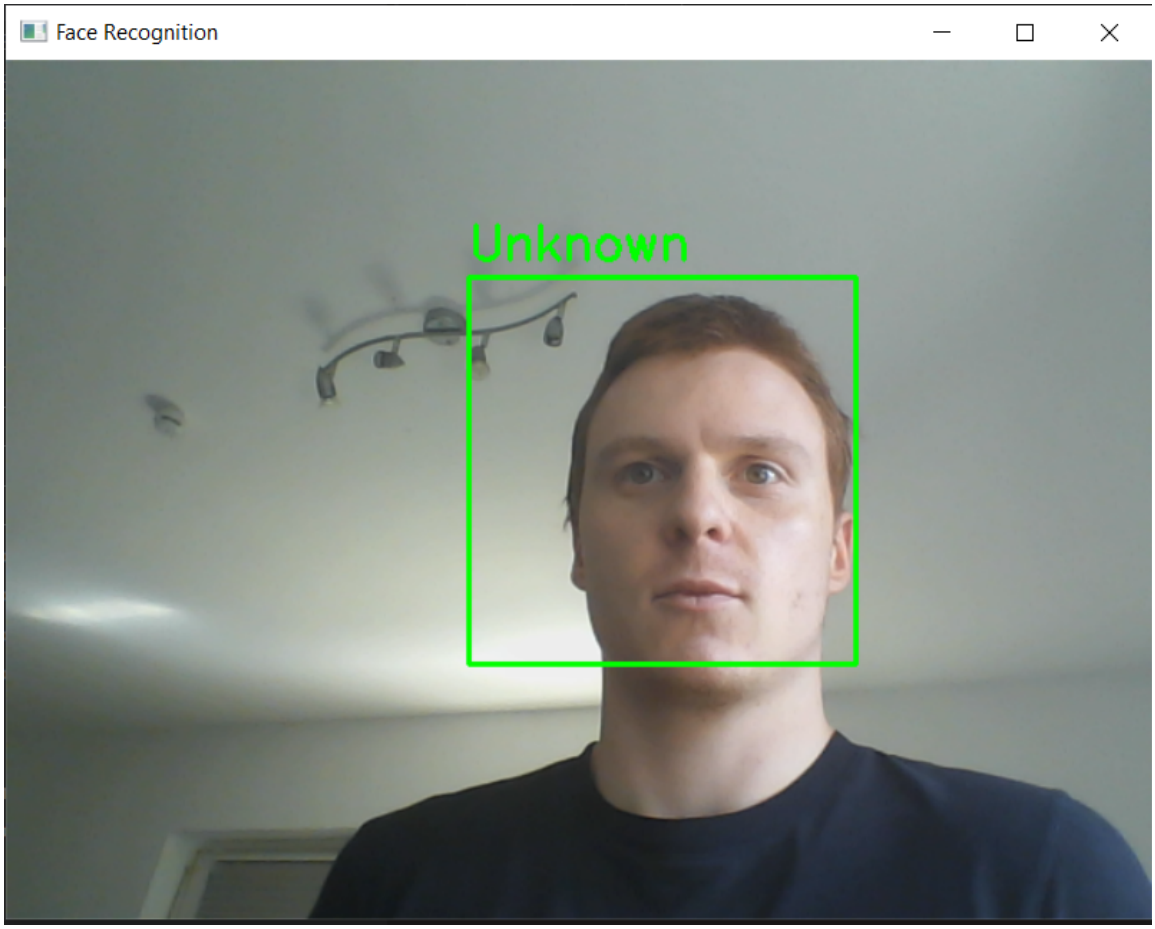
```
    rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(rgb, scaleFactor=1.2, minNeighbors=5, min
    return faces

# Initialize webcam
cap = cv2.VideoCapture(0)


# Check if the webcam is available
if not cap.isOpened():
    print("No webcam found or access denied.")
else:
    # Infinite loop for face recognition
    while True:
        # Capture frame from the webcam
        ret, frame = cap.read()
        # Detect faces
        faces = detect_faces(frame)

        # Make predictions for each detected face
        for (x, y, w, h) in faces:

            # Expand the frame of the face
            val = 40
            x = max(0, x - val)
            y = max(0, y - val)
            w = min(frame.shape[1], val+x+w) - x
            h = min(frame.shape[0], val+y+h) - y
            face_image = frame[y:y+h, x:x+w]

            #face_image = np.expand_dims(face_image, axis=0)
            face_image = img_to_array(face_image)
            # normalize the image
            face_image = cv2.normalize(face_image, None, alpha=0, beta=1, norm_type
            face_image = array_to_img(face_image)
            face_image = face_image.resize((224, 224))
            face_image = np.expand_dims(face_image, axis=0)
            face_image = np.array(face_image)

            # Perform face classification
            # get the label from the highest prediction
            predictions = np.argmax(model.predict(face_image), axis=1)[0]
            cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)
            cv2.putText(frame, classes[predictions], (x, y-10), cv2.FONT_HERSHEY_SI

        # Display the frame
        cv2.imshow('Face Recognition', frame)

        # Exit the loop by pressing 'q' key
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

# Release the webcam stream
cap.release()
cv2.destroyAllWindows()
```

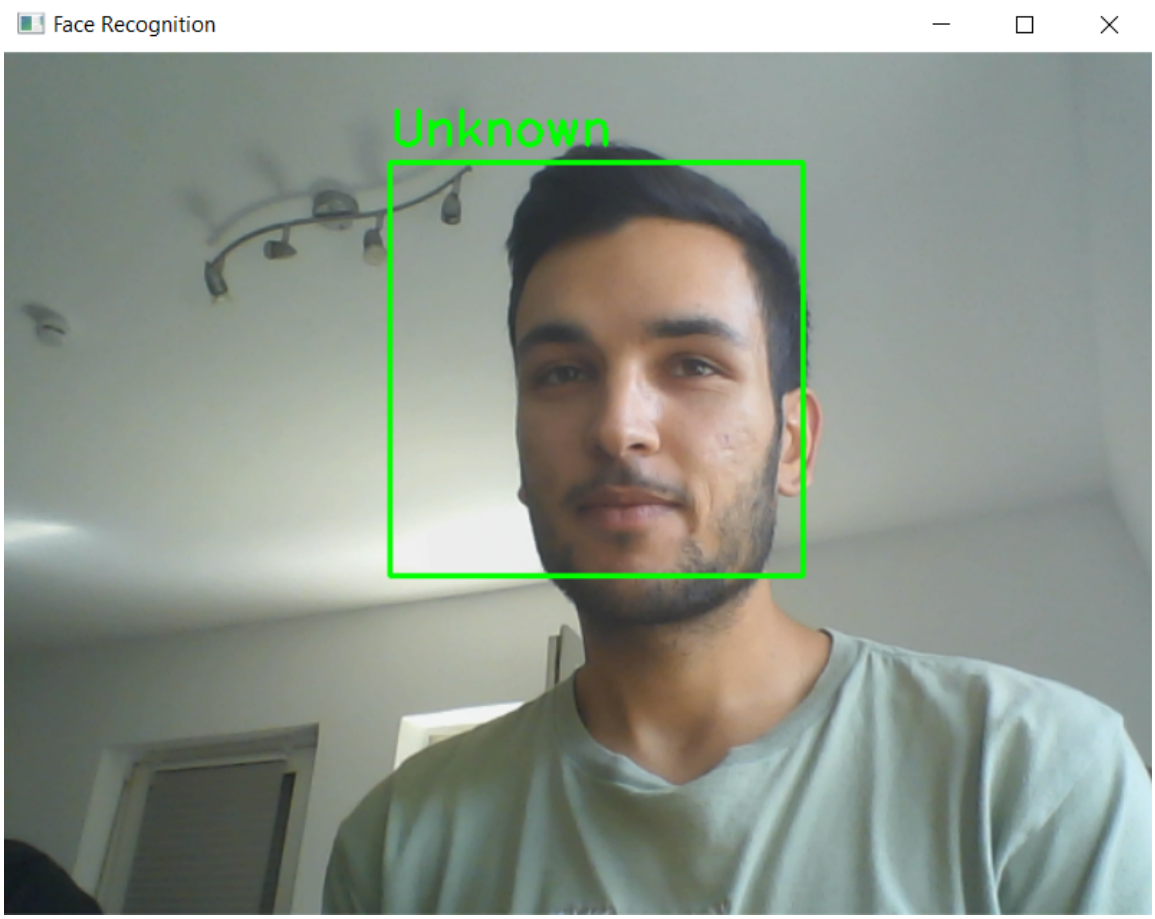
If there is no possibility to test the live mode, here are a few pictures of the mode while running. The first three pictures are from Dominik Bücher and the last two from two diffrent friends.




Face Recognition
Dominik


Face Recognition
Dominik


Face Recognition
Unknown


Face Recognition
Unknown

Live mode evaluation:

- Basically, the live mode works very well, the frame and the label are projected into the window in real time.
- Also the recognition of Dominik and strangers is quite good for most models. There are of course differences in speed and quality depending on the model.
- It is recommended to run the live mode with a powerful PC, so that the image can be generated smoothly.
- Problems with recognition were more common in the Unknown class, where several people were often shown as Dominik. This suggests that the dataset was perhaps distributed too unevenly.
- In the pictures and of course directly in live mode, it is noticeable that the frame around the face does not always sit so well. This can greatly impair the quality of the label, since sometimes only half the face is transferred to the model. This should be improved in the future, if you continue to work with the live mode.

Create a product from it:

To create an actual product from the trained models, various approaches can be pursued. The Live Mode is already a great entry point, which could run in the background on the PC and offer specific functionalities exclusively to the owner. However, it's important to address the current errors to create a reliable product. Additionally, the models would need to be retrained for each individual owner. Despite this, given the widespread usage of facial recognition in numerous devices, it would be meaningful to further develop this product.

To achieve that, it would be beneficial to use more data of the desired individual for model training. Moreover, a better user interface is required for deploying the model. The interface provided here is only for testing purposes and not suitable for actual application.