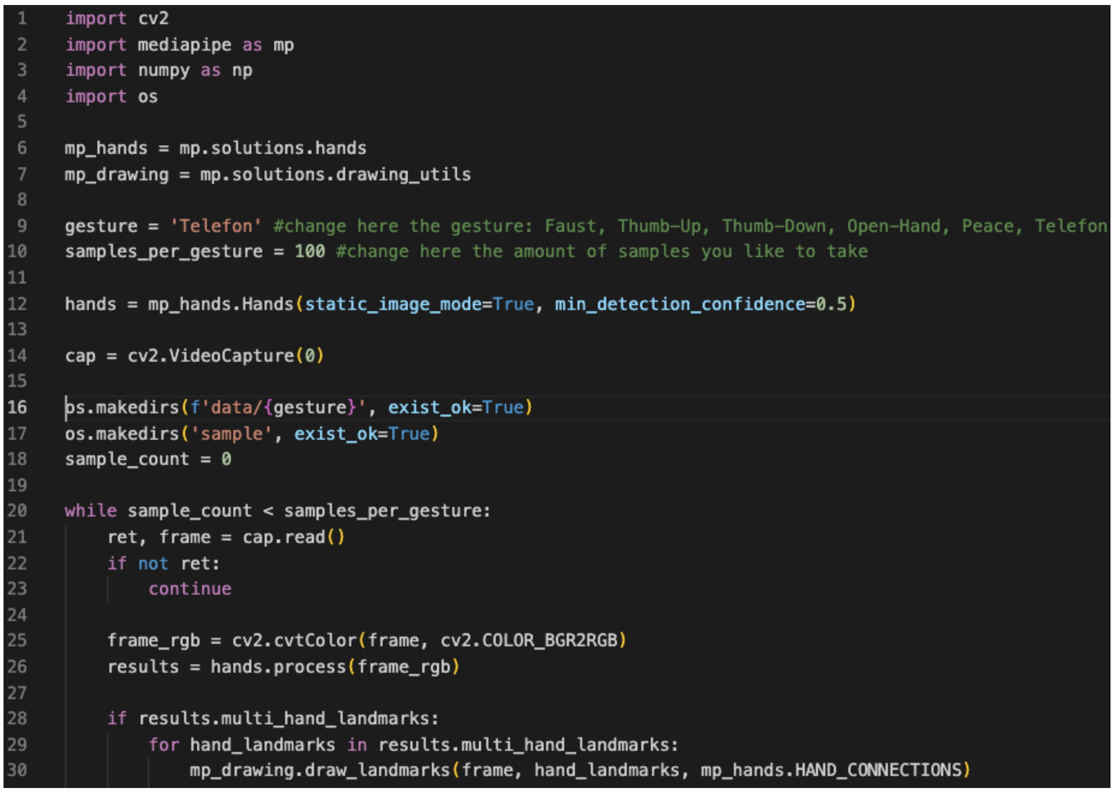
This documentation complements the paper *"Hand Gesture Recognition with MediaPipe and TensorFlow"* by providing additional implementation details. Since the paper covers the problem statement, motivation, state of the art, and results, these aspects are not repeated here.

The following sections briefly describe the implementation of the three models mentioned in the paper, along with the data generation process. Each model is documented in a dedicated Python notebook. Additionally, the evaluation process is outlined at the end of this documentation, while the results can be found in the paper. Also a code-snippet for live-testing is provided at the very end.

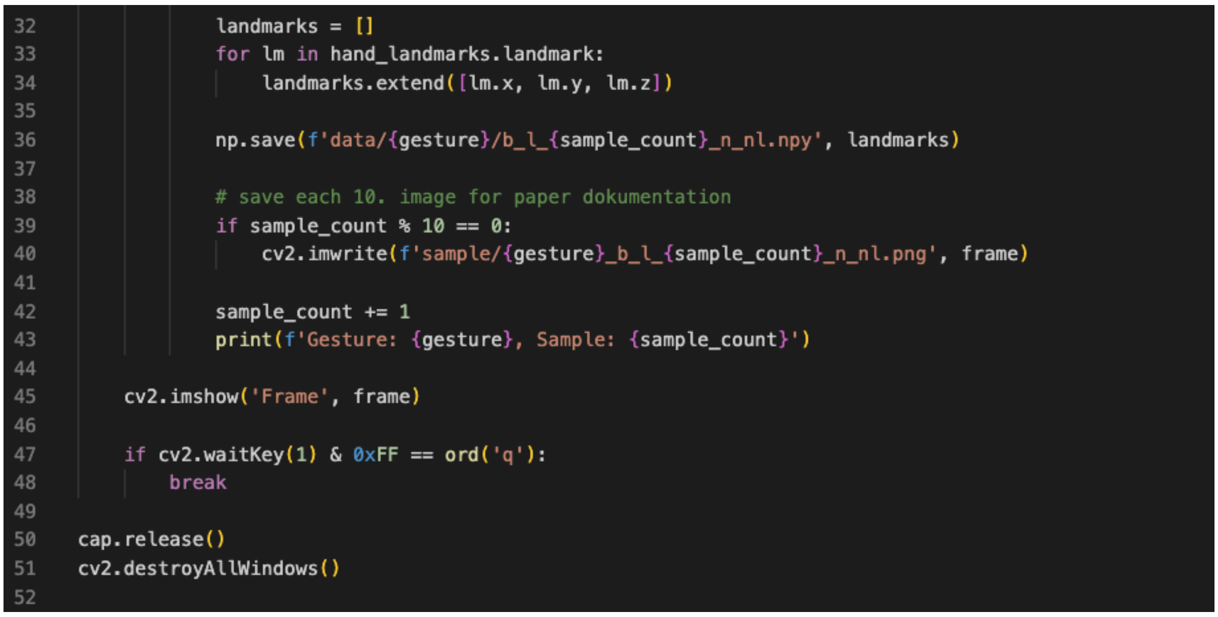
The code and implemented models can be found in this repo https://github.com/Gatze1910/computervision-handgesture-project (if you need access, please write Bernadette -> s2310595001@students.fh-hagenberg.at)

The dataset for the hand gesture recognition task was constructed with six distinct gestures: "Telephone," "Fist," "Thumb-Up," "Thumb-Down," "Open-Hand," and "Peace." To ensure a diverse and robust dataset, 800 samples per gesture were captured by two individuals (the authors), resulting in 1,600 samples per gesture. The dataset was collected under different conditions, including variations in lighting, hand positions (e.g., rotating the hand, proximity to the camera), and using both hands to perform the gestures. The captured data was processed using MediaPipe up to extract 63 normalized landmarks (x, y, z coordinates) per sample, which represent key points on the hand. This was achieved using the GenerateData.py implementation.

Data Generation) GenerateData.py



```python
import cv2
import mediapipe as mp
import numpy as np
import os

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

gesture = 'Telefon' #change here the gesture: Faust, Thumb-Up, Thumb-Down, Open-Hand, Peace, Telefon
samples_per_gesture = 100 #change here the amount of samples you like to take

hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.5)

cap = cv2.VideoCapture(0)

os.makedirs(f'data/{gesture}', exist_ok=True)
os.makedirs('sample', exist_ok=True)
sample_count = 0

while sample_count < samples_per_gesture:
    ret, frame = cap.read()
    if not ret:
        continue

    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(frame, hand_landmarks, mp_hands.HAND_CONNECTIONS)
```



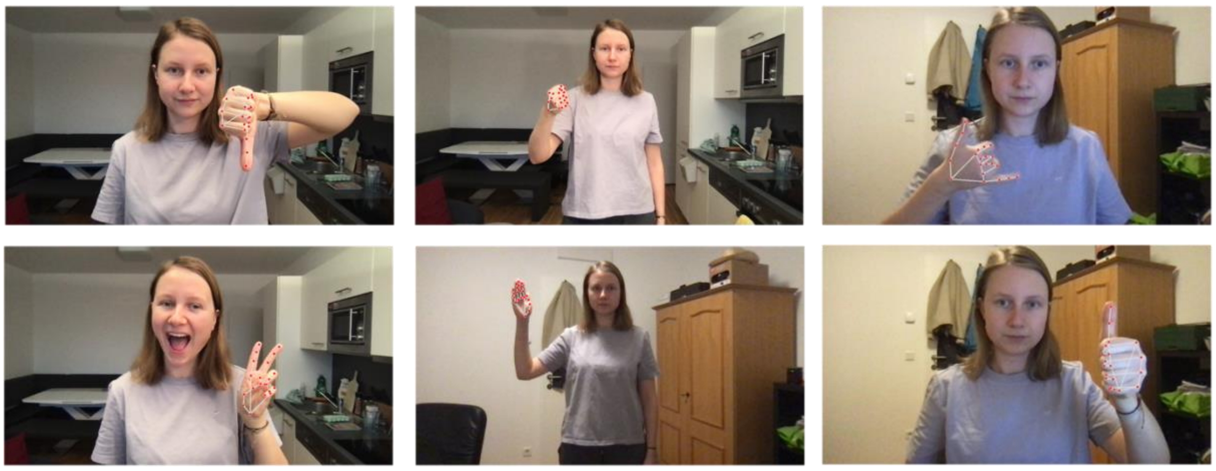
```
            landmarks = []
            for lm in hand_landmarks.landmark:
                landmarks.extend([lm.x, lm.y, lm.z])

            np.save(f'data/{gesture}/b_l_{sample_count}_n_nl.npy', landmarks)

            # save each 10. image for paper dokumentation
            if sample_count % 10 == 0:
                cv2.imwrite(f'sample/{gesture}_b_l_{sample_count}_n_nl.png', frame)

            sample_count += 1
            print(f'Gesture: {gesture}, Sample: {sample_count}')

    cv2.imshow('Frame', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

The data collection process of GenerateData.py uses the MediaPipe framework to detect and capture hand landmarks. It continuously captures frames from the webcam, processes them using MediaPipe's hand tracking model, and saves the landmarks for each gesture. Here's a detailed explanation of how the code works:

1. **Setup**: The script starts by importing necessary libraries, including cv2 (for video capture), mediapipe (for hand tracking), and numpy (for numerical data handling). A directory is created for each gesture to store the captured landmarks.
2. **Hand Detection**: The code initializes the MediaPipe hands model (mp_hands.Hands) with a minimum detection confidence of 0.5. The webcam is opened using cv2.VideoCapture(0).
3. **Data Capture Loop**: Inside the loop, frames are captured from the webcam. The captured frame is converted from BGR to RGB to be compatible with MediaPipe. The hand landmarks are extracted using hands.process(frame_rgb).
4. **Landmark Extraction**: If hand landmarks are detected, they are drawn on the frame using mp_drawing.draw_landmarks(). The landmarks (x, y, z coordinates) are then extracted from the hand's landmarks and saved to a .npy file in the corresponding gesture directory. This allows the data to be easily used for training the machine learning model.
5. **Image Saving for Documentation**: Every 10th frame is saved as a PNG image to provide visual documentation of the data collection process. This is done to ensure that the progression of gesture collection can be tracked visually. Some images of those were put together for the collages in the paper.
6. **Stopping the Capture**: The loop continues until the specified number of samples for each gesture is captured. Pressing the 'q' key will terminate the process.

The landmarks data, saved as .npy files, are used in the subsequent steps of model training and evaluation. This approach ensures that the gesture recognition models are trained on a large and diverse dataset, improving their accuracy and robustness.

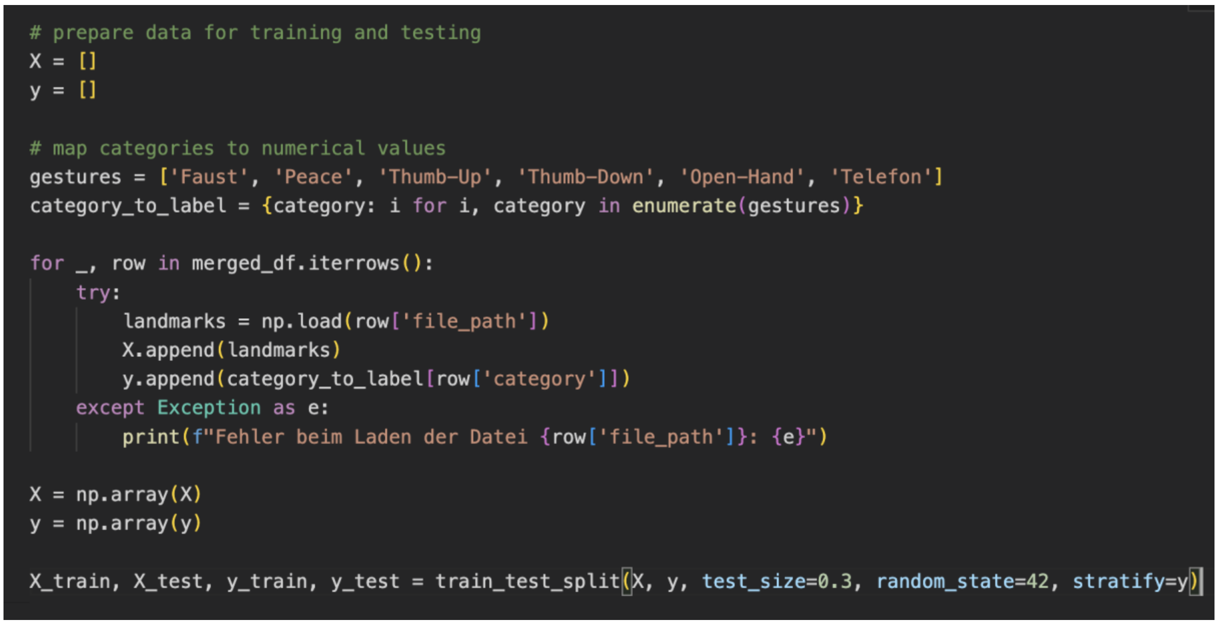
Here are some of the step 5) generated images, to demonstrate how the generation of the data-samples looked like:



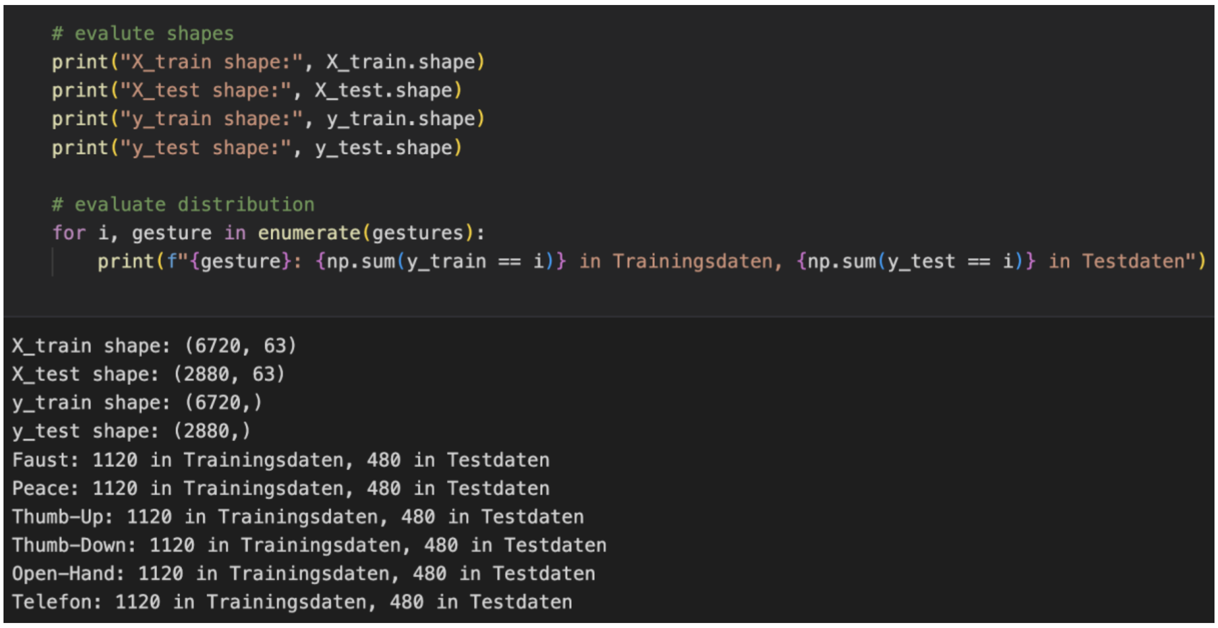
The red-marked-dots are the so-called landmarks.

Three different models were implemented to provide variety and flexibility, allowing for evaluation and comparison of their performance. Each model is documented in a dedicated Python notebook with detailed code comments. Therefore, this section focuses only on the key aspects of each model's structure, which are also explained in the paper.

Input Generated-Data)
The data was then split using train_test_split from sklearn with a 0.3 ratio and each gesture was proper distributed in the training and testing dataset:



```python
# prepare data for training and testing
X = []
y = []

# map categories to numerical values
gestures = ['Faust', 'Peace', 'Thumb-Up', 'Thumb-Down', 'Open-Hand', 'Telefon']
category_to_label = {category: i for i, category in enumerate(gestures)}

for _, row in merged_df.iterrows():
    try:
        landmarks = np.load(row['file_path'])
        X.append(landmarks)
        y.append(category_to_label[row['category']])
    except Exception as e:
        print(f"Fehler beim Laden der Datei {row['file_path']}: {e}")

X = np.array(X)
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```



```python
# evalute shapes
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)

# evaluate distribution
for i, gesture in enumerate(gestures):
    print(f"{gesture}: {np.sum(y_train == i)} in Trainingsdaten, {np.sum(y_test == i)} in Testdaten")
```

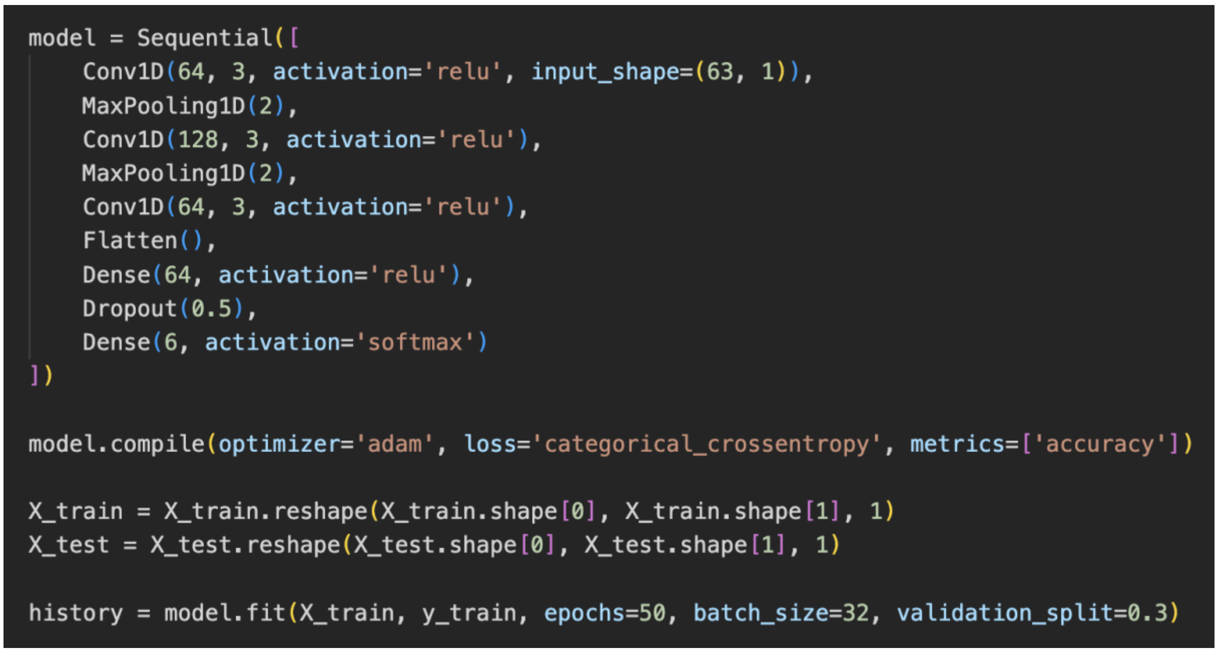
```
X_train shape: (6720, 63)
X_test shape: (2880, 63)
y_train shape: (6720,)
y_test shape: (2880,)
Faust: 1120 in Trainingsdaten, 480 in Testdaten
Peace: 1120 in Trainingsdaten, 480 in Testdaten
Thumb-Up: 1120 in Trainingsdaten, 480 in Testdaten
Thumb-Down: 1120 in Trainingsdaten, 480 in Testdaten
Open-Hand: 1120 in Trainingsdaten, 480 in Testdaten
Telefon: 1120 in Trainingsdaten, 480 in Testdaten
```

Model 1) Handgesture_Model_1.ipynb

Model 1 utilizes a convolutional neural network (CNN) to extract features from the input data. The architecture consists of three convolutional layers with ReLU activation, followed by max-pooling layers. After flattening the features, two dense layers are applied, with a dropout layer for regularization. The output layer employs a softmax activation to classify six hand gestures. The detailed architecture is as follows:

• Input shape: (63, 1)
• Conv1D (64 filters, kernel size 3, ReLU activation)
• MaxPooling1D (pool size 2)
• Conv1D (128 filters, kernel size 3, ReLU activation)
• MaxPooling1D (pool size 2)
• Conv1D (64 filters, kernel size 3, ReLU activation)
• Flatten layer
• Dense (64 units, ReLU activation)
• Dropout (rate 0.5)
• Dense (6 units, softmax activation)

The model was trained using the Adam optimizer and categorical cross-entropy loss, with a batch size of 32 and 50 epochs as shown in the following graphic.



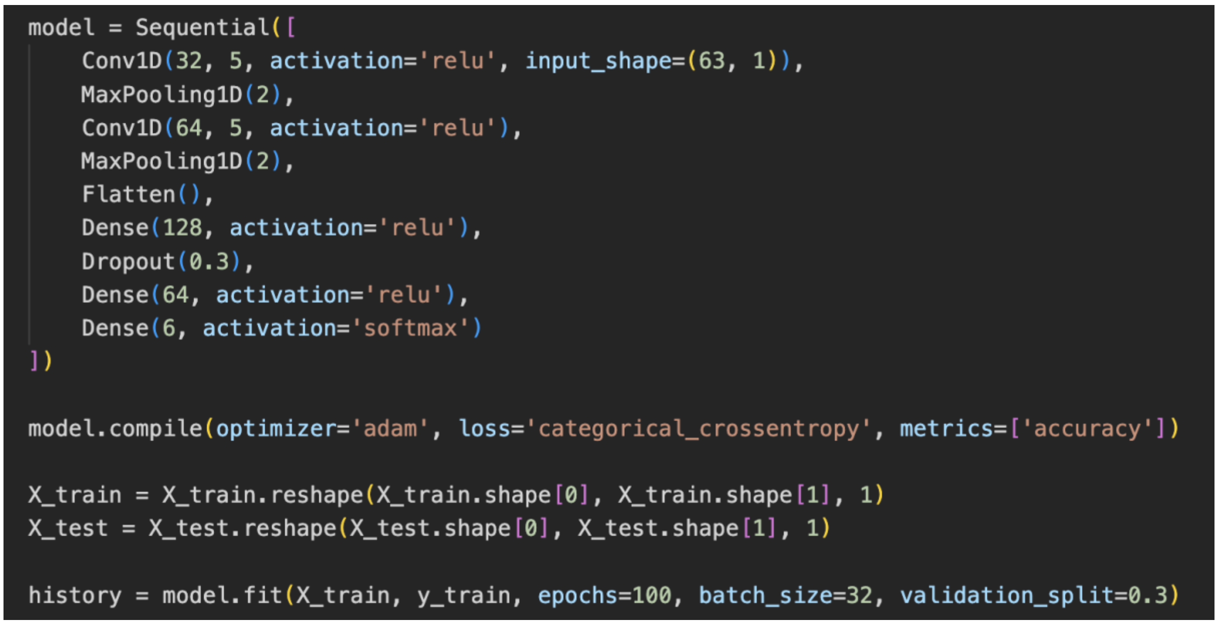
```
model = Sequential([
    Conv1D(64, 3, activation='relu', input_shape=(63, 1)),
    MaxPooling1D(2),
    Conv1D(128, 3, activation='relu'),
    MaxPooling1D(2),
    Conv1D(64, 3, activation='relu'),
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(6, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.3)
```

Model 2) Handgesture_Model_2.ipynb

Model 2 refines the CNN approach by using different configurations for kernel sizes, number of filters, and dropout rates. The architecture is optimized for better generalization and performance. Key features include:

• Input shape: (63, 1)
• Conv1D (32 filters, kernel size 5, ReLU activation)
• MaxPooling1D (pool size 2)
• Conv1D (64 filters, kernel size 5, ReLU activation)
• MaxPooling1D (pool size 2)
• Flatten layer
• Dense (128 units, ReLU activation)
• Dropout (rate 0.3)
• Dense (64 units, ReLU activation)
• Dense (6 units, softmax activation)

Similar to Model 1, this model was trained with the Adam optimizer and categorical cross-entropy loss, but with 100 epochs for improved performance as shown in the following graphic.



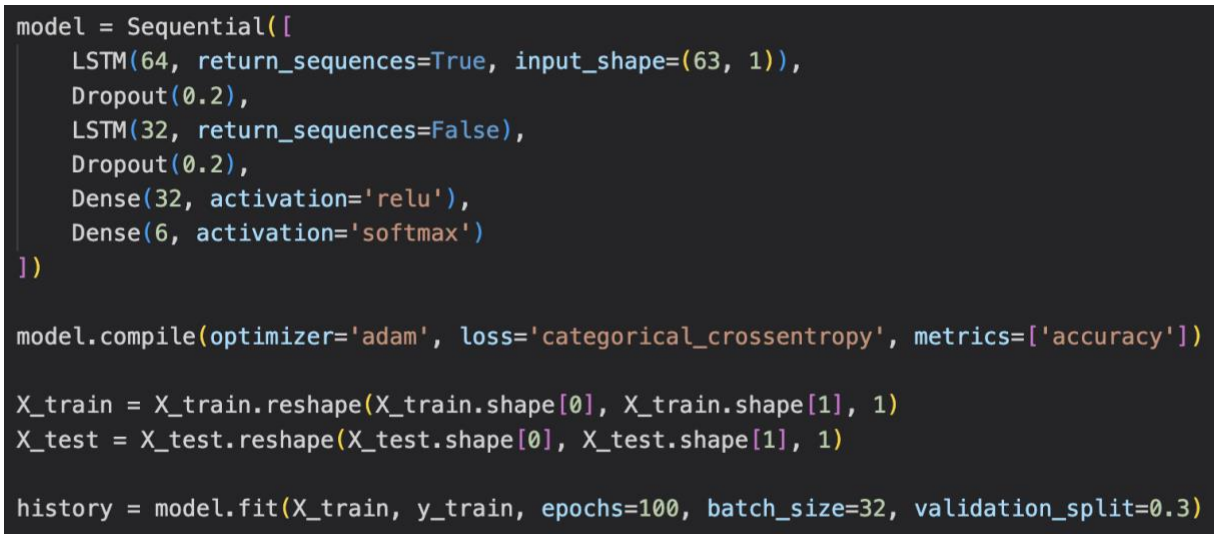
```
model = Sequential([
    Conv1D(32, 5, activation='relu', input_shape=(63, 1)),
    MaxPooling1D(2),
    Conv1D(64, 5, activation='relu'),
    MaxPooling1D(2),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dense(6, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.3)
```

Model 3) Handgesture_Model_3.iypnb

Model 3 employs a model with Long Short-Term Memory (LSTM) layers to capture temporal dependencies in the input sequence data. The architecture consists of two LSTM layers, followed by dense layers for classification. Key components are:

• Input shape: (63, 1)
• LSTM (64 units, return sequences: True)
• Dropout (rate 0.2)
• LSTM (32 units, return sequences: False)
• Dropout (rate 0.2)
• Dense (32 units, ReLU activation)
• Dense (6 units, softmax activation)

This model was also trained with the Adam optimizer and categorical cross-entropy loss, using a batch size of 32 and 100 epochs, as shown in the below figure.

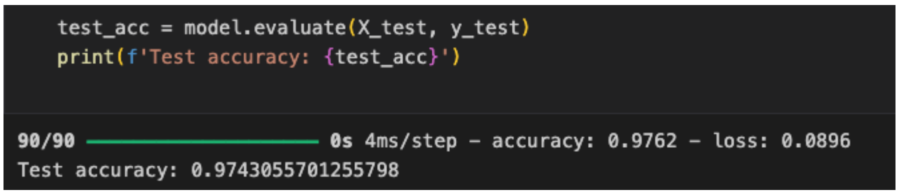


```
model = Sequential([
    LSTM(64, return_sequences=True, input_shape=(63, 1)),
    Dropout(0.2),
    LSTM(32, return_sequences=False),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dense(6, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.3)
```

Evaluation)
The results are discussed in the paper section VI. Results.

The evaluation was approached through two distinct methodologies.

The first involved utilizing the train_test_split function during each model implementation and subsequent performance measurement. (each model-python-notebook includes it, an example is shown in the following graphic)



```
test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')

90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9762 - loss: 0.0896
Test accuracy: 0.9743055701255798
```

The second approach, designed to assess model efficacy in predicting hand gestures for novel individuals, was conducted as follows: (HandgesturesValidation.ipynb)

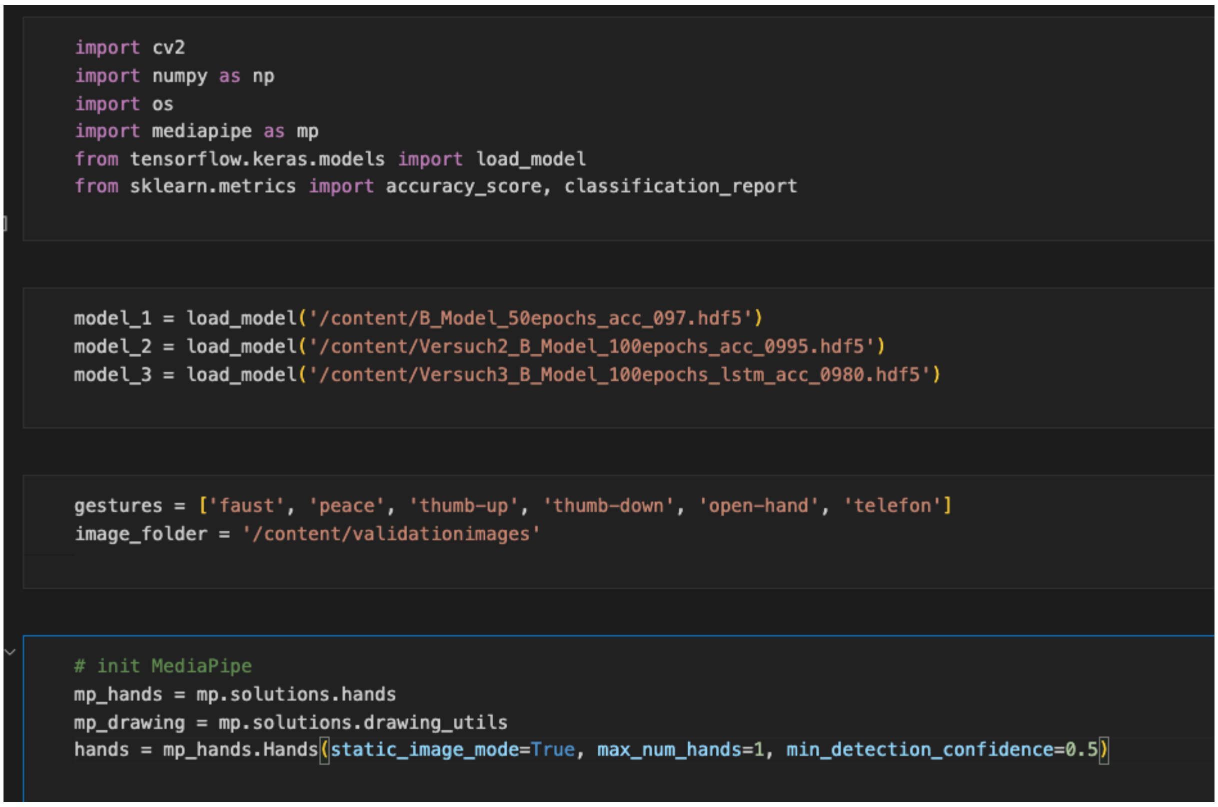
An evaluation was performed using a small, carefully curated validation set. This set comprised two individuals (one male and one female) intentionally selected to ensure their data had not been previously exposed to the models during the training phase.

The evaluation script assesses three models using a validation dataset consisting of 12 images. It employs MediaPipe Hands for hand landmark detection and subsequently predicts the gesture displayed in each image based on these landmarks for each model.

Process Flow:
1. Model Loading: The three models are loaded using TensorFlow.
2. Hand Detection: MediaPipe is utilized to extract hand landmarks from the images.
3. Image Processing: OpenCV is employed to load images, convert them to RGB format, and process hand landmarks.
4. Prediction: Each model predicts the gesture based on the extracted landmarks.
5. Evaluation: Predictions are compared with actual labels (derived from filenames), and accuracy is calculated.
6. Results Analysis: Metrics such as precision, recall, and F1-score are reported, along with a misclassification analysis.

Conclusion:
The script provides an objective evaluation of model performance on previously unseen data, thereby facilitating the identification of areas for potential improvement.

Here is the process flow shown:

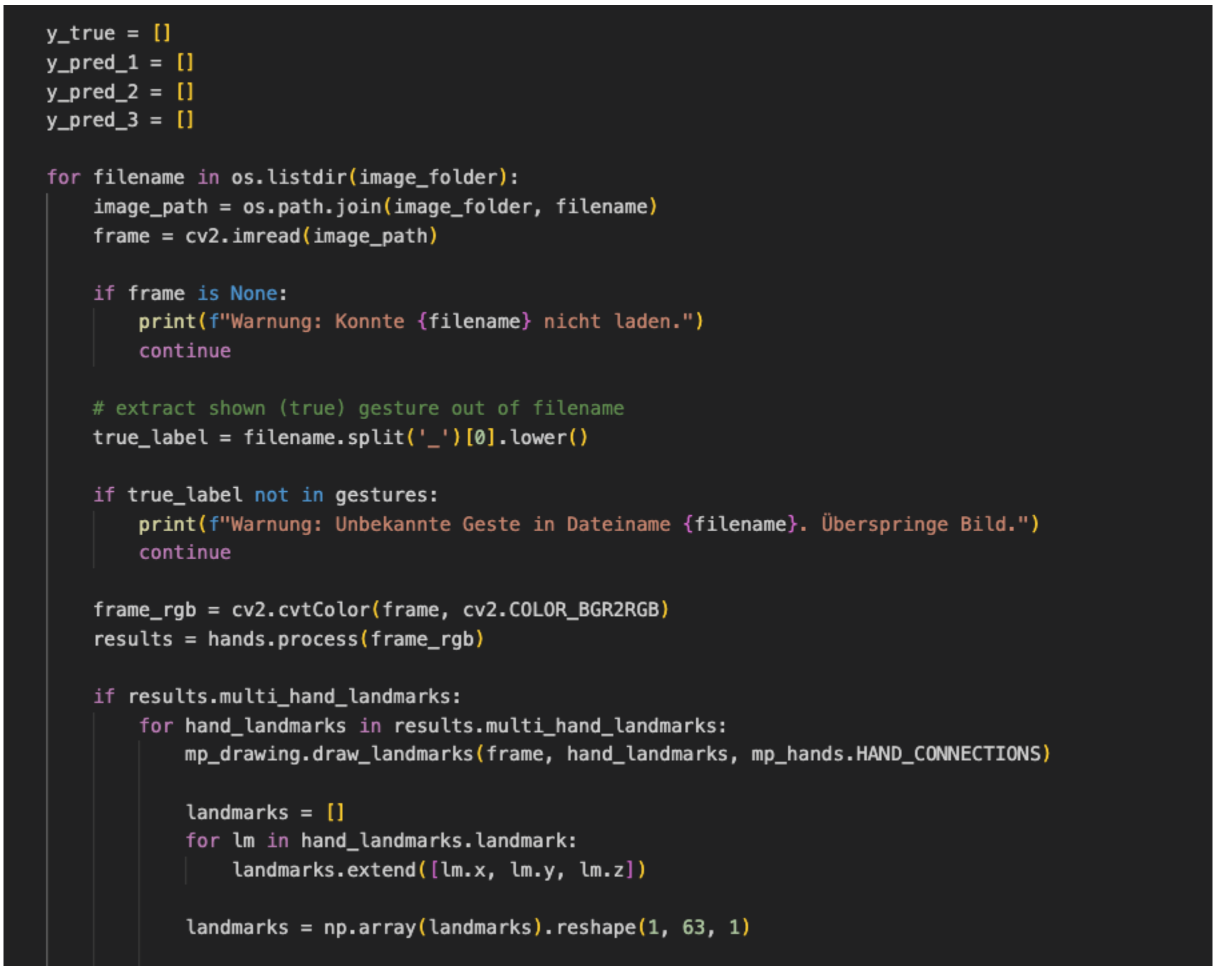


```python
import cv2
import numpy as np
import os
import mediapipe as mp
from tensorflow.keras.models import load_model
from sklearn.metrics import accuracy_score, classification_report
```

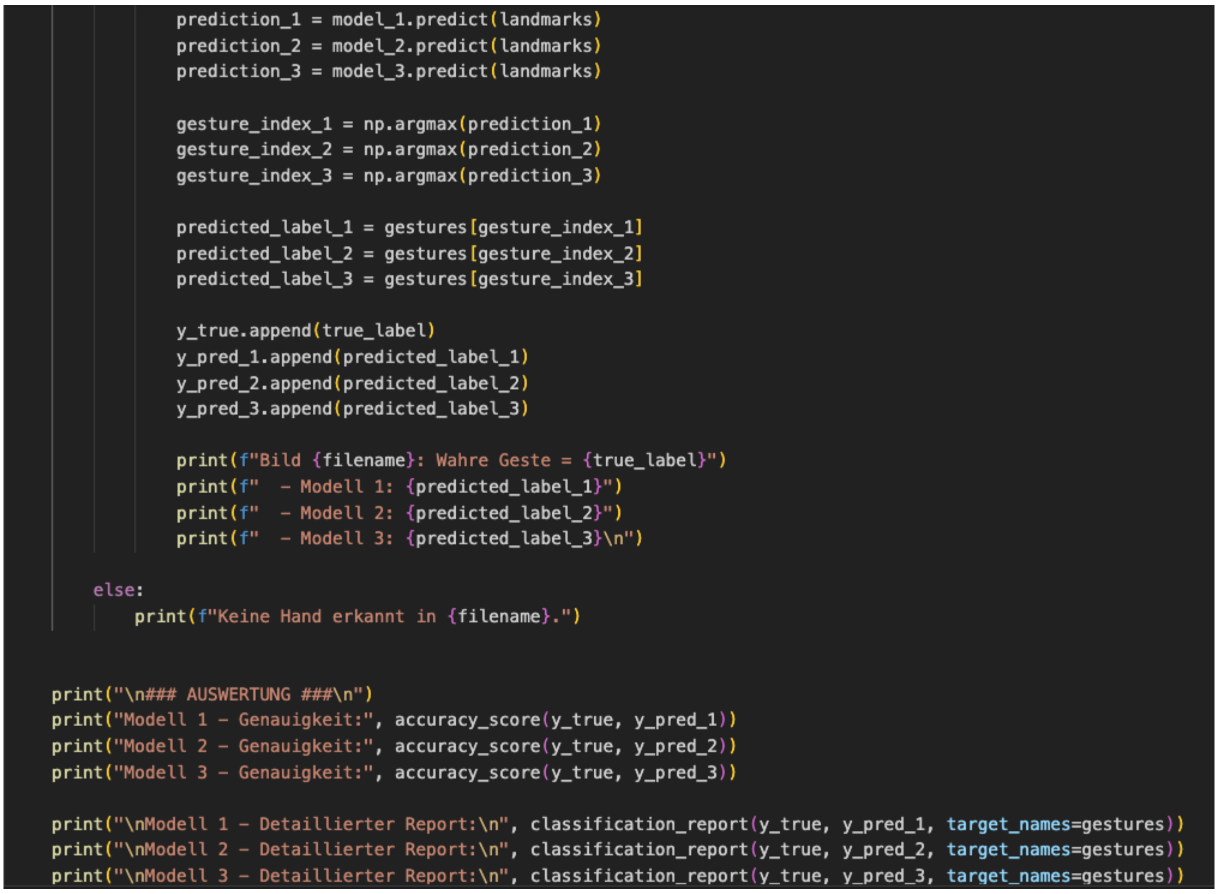
```python
model_1 = load_model('/content/B_Model_50epochs_acc_097.hdf5')
model_2 = load_model('/content/Versuch2_B_Model_100epochs_acc_0995.hdf5')
model_3 = load_model('/content/Versuch3_B_Model_100epochs_lstm_acc_0980.hdf5')
```

```python
gestures = ['faust', 'peace', 'thumb-up', 'thumb-down', 'open-hand', 'telefon']
image_folder = '/content/validationimages'
```

```python
# init MediaPipe
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
hands = mp_hands.Hands(static_image_mode=True, max_num_hands=1, min_detection_confidence=0.5)
```



```python
y_true = []
y_pred_1 = []
y_pred_2 = []
y_pred_3 = []

for filename in os.listdir(image_folder):
    image_path = os.path.join(image_folder, filename)
    frame = cv2.imread(image_path)

    if frame is None:
        print(f"Warnung: Konnte {filename} nicht laden.")
        continue

    # extract shown (true) gesture out of filename
    true_label = filename.split('_')[0].lower()

    if true_label not in gestures:
        print(f"Warnung: Unbekannte Geste in Dateiname {filename}. Überspringe Bild.")
        continue

    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(frame, hand_landmarks, mp_hands.HAND_CONNECTIONS)

            landmarks = []
            for lm in hand_landmarks.landmark:
                landmarks.extend([lm.x, lm.y, lm.z])

            landmarks = np.array(landmarks).reshape(1, 63, 1)
```



```python
                prediction_1 = model_1.predict(landmarks)
                prediction_2 = model_2.predict(landmarks)
                prediction_3 = model_3.predict(landmarks)

                gesture_index_1 = np.argmax(prediction_1)
                gesture_index_2 = np.argmax(prediction_2)
                gesture_index_3 = np.argmax(prediction_3)

                predicted_label_1 = gestures[gesture_index_1]
                predicted_label_2 = gestures[gesture_index_2]
                predicted_label_3 = gestures[gesture_index_3]

                y_true.append(true_label)
                y_pred_1.append(predicted_label_1)
                y_pred_2.append(predicted_label_2)
                y_pred_3.append(predicted_label_3)

                print(f"Bild {filename}: Wahre Geste = {true_label}")
                print(f"  - Modell 1: {predicted_label_1}")
                print(f"  - Modell 2: {predicted_label_2}")
                print(f"  - Modell 3: {predicted_label_3}\n")

        else:
            print(f"Keine Hand erkannt in {filename}.")


    print("\n### AUSWERTUNG ###\n")
    print("Modell 1 - Genauigkeit:", accuracy_score(y_true, y_pred_1))
    print("Modell 2 - Genauigkeit:", accuracy_score(y_true, y_pred_2))
    print("Modell 3 - Genauigkeit:", accuracy_score(y_true, y_pred_3))

    print("\nModell 1 - Detaillierter Report:\n", classification_report(y_true, y_pred_1, target_names=gestures))
    print("\nModell 2 - Detaillierter Report:\n", classification_report(y_true, y_pred_2, target_names=gestures))
    print("\nModell 3 - Detaillierter Report:\n", classification_report(y_true, y_pred_3, target_names=gestures))
```

Live-Testing) Testing.py
All three models, can be tested live, using the following script, where only the pathtomodel needs to be set correctly.

```python
import cv2
import mediapipe as mp
import numpy as np
from tensorflow.keras.models import load_model


mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils


model = load_model(f'./pathtomodel')
gestures = ['faust', 'peace', 'thumb-up', 'thumb-down', 'open-hand', 'telefon']


cap = cv2.VideoCapture(0)
hands = mp_hands.Hands(static_image_mode=False, max_num_hands=1, min_detection_confidence=0.5)


while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        continue

    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(frame, hand_landmarks, mp_hands.HAND_CONNECTIONS)

            landmarks = []
            for lm in hand_landmarks.landmark:
                landmarks.extend([lm.x, lm.y, lm.z])

            landmarks = np.array(landmarks).reshape(1, 63, 1)
            prediction = model.predict(landmarks)
            gesture_index = np.argmax(prediction)
            gesture = gestures[gesture_index]

            cv2.putText(frame, gesture, (10, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

    cv2.imshow('Hand Gesture Recognition', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

The script will open your laptop camera and evaluate in live each shown gesture – for example: