Hand Pose Recognition using Deep Learning

Dr. Eko Mulyanto Yuniarno

November 2024

Contents

Chapter 1

Introduction

Hand pose are one of the most common communication methods in daily human life. Interaction modalities of user interfaces play a dominant role in the relationship between people and computer technology. The way users interact with interfaces has undergone a significant transformation, with most of the population now using touch devices. In today's technological advancements, human-computer interfaces (HCI) are highly appealing, mainly because they aim to enhance human lifestyles. Alongside these developments, technologies that facilitate interaction with these devices are also required. Initially, interactions were conducted using a mouse and keyboard with computers. However, with the rise of ubiquitous computing, gestures have become more prevalent—for example, smartphone interactions often involve hand gestures. The human body provides a wide range of poses that can be used as computer input. Images captured by cameras exhibit a vast amount of variation. This occurs because images are spatial data where each pixel represents a color at a specific coordinate. For pose classification, many images are needed for training to account for variations in scale, position, and hand orientation within the images. The desired outcome is a feature invariant to scale, position, and orientation, ensuring accurate and robust classification.

1.1 Importance of Hand Pose Recognition

Hand pose recognition is pivotal in advancing human-computer interaction (HCI), enabling intuitive communication between humans and machines. It bridges the gap between natural human gestures and machine understanding, opening new avenues in virtual reality, robotics, and accessibility technologies. For instance, hand gesture recognition is essential for sign language translation, empowering people with hearing impairments to communicate

seamlessly with others. Moreover, the ability to accurately recognize hand poses enhances precision in applications such as gaming, remote robotic control, and augmented reality, where fine motor movements dictate user experience. As technology continues to evolve, the importance of hand pose recognition grows, particularly in creating inclusive, adaptive, and user-friendly systems.

1.2 Challenges in Hand Pose Recognition

Despite its significance, hand pose recognition presents numerous challenges, primarily due to the variability and complexity of human hands. Hands have intricate structures with multiple degrees of freedom, leading to a vast range of poses that can be difficult to capture and interpret accurately. Variations in lighting, occlusions caused by overlapping fingers, and differences in hand shapes across individuals add further complexity. Real-time processing demands also pose a challenge, requiring high computational efficiency without sacrificing accuracy. Additionally, datasets used for training hand pose recognition models often need more diversity, limiting their generalizability in real-world scenarios. Addressing these challenges requires robust algorithms capable of handling variations and noise while maintaining computational efficiency.

1.3 Hand Pose Recognition Implementation in Medical

Hand pose recognition is revolutionizing patient care and therapy methods in the medical field. Surgeons can utilize gesture-based systems to interact with medical imaging during procedures, eliminating the need for physical contact with equipment and ensuring sterility in the operating room. In rehabilitation, hand pose recognition monitors and guides patients recovering from injuries or surgeries involving fine motor skills. For example, it enables therapists to track real-time progress and customize exercises based on a patient's movements. Moreover, hand pose recognition aids in developing assistive devices for individuals with motor impairments, enhancing their ability to perform daily tasks independently. This technology's application in healthcare improves the precision of treatments and fosters patient engagement and empowerment.

Chapter 2

Fundamentals of Hand Pose Recognition with MediaPipe

2.1 Understanding Pose Recognition

Pose recognition is analyzing and interpreting objects' physical positions and movements, particularly the human body or hands, to identify specific gestures or actions. In the context of hand pose recognition, the objective is to track and understand the intricate movements and orientations of the hand in a given space. This involves detecting the hand's position, identifying key landmarks, and classifying the overall hand configuration. Pose recognition systems combine advanced computer vision techniques with machine learning models to make predictions, offering applications in sign language interpretation, virtual reality, and human-computer interaction.

2.2 MediaPipe

MediaPipe is a cross-platform framework by Google that offers efficient pipelines for machine learning and computer vision tasks, such as pose estimation, hand tracking, and facial landmark detection. It supports mobile devices, web, and desktop environments with pre-trained models.

2.3 Real-Time Hand Tracking

Real-time hand tracking is the ability to detect and monitor hand movements instantaneously. This process relies on high-speed algorithms that process visual data from cameras or sensors to pinpoint the location and orientation of

a hand in every frame of a video stream. A significant focus of real-time hand tracking is minimizing latency to ensure the system's responsiveness. Techniques like region-of-interest optimization, GPU acceleration, and lightweight neural networks contribute to achieving this. Real-time tracking is critical for applications such as augmented reality (AR), gaming, and touchless interfaces where seamless interaction is essential.

2.4 Landmark Detection

Landmark detection is a core element of hand pose recognition. It involves identifying key points or nodes on a hand, such as joints, knuckles, or fingertips, collectively defining its structure. These landmarks are typically represented in 2D or 3D coordinates, forming the foundation for analyzing hand gestures and poses. Advanced models like Mediapipe Hand and OpenPose leverage machine learning to perform this task efficiently. Landmark detection is critical for understanding hand dynamics, as it provides the data needed for gesture classification, tracking movements, and reconstructing hand positions in 3D space. The hand landmark can be seen in Figure 2.1.

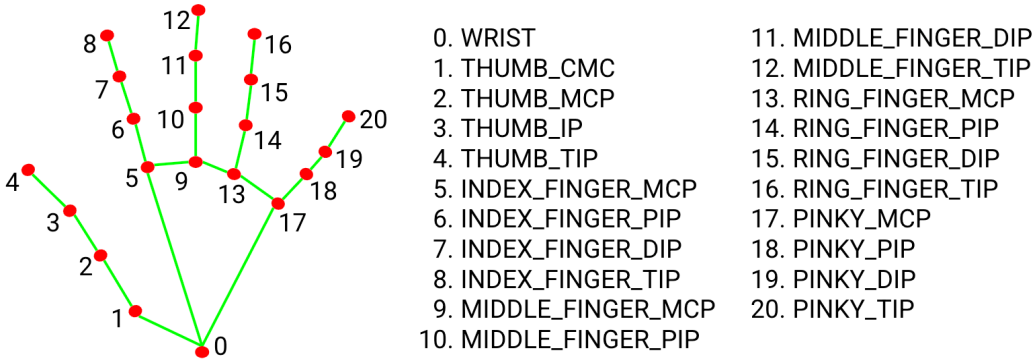

Figure 2.1: Example of Hand Landmark obtained from Mediapipe

2.5 Model architecture

The model architecture for hand pose recognition typically combines convolutional neural networks (CNNs) for feature extraction with regression or classification layers for predicting hand poses. State-of-the-art models often utilize encoder-decoder frameworks to process input images and generate

heatmaps or landmark coordinates. Mediapipe, for instance, uses a pipeline that first detects the hand region and then applies a specialized model to predict precise landmarks. Lightweight and efficient architectures are crucial to ensure real-time performance without compromising accuracy. These models are trained on large datasets to generalize across different hand shapes, orientations, and lighting conditions.

2.6 Pose Estimation in 3D

Pose estimation in 3D extends the capabilities of 2D models by incorporating depth information to reconstruct hand poses in a three-dimensional space. This process often integrates data from depth sensors or multi-view camera setups to triangulate landmark positions. 3D pose estimation provides more accurate representations of hand gestures, especially in scenarios involving rotations or occlusions. It is beneficial in applications like robotics, where precise hand positioning is required, or virtual reality, where realistic interactions with virtual objects depend on spatial accuracy. Each landmark consists of the following:

1. x and y: Landmark coordinates normalized to $[0.0, 1.0]$ by the image width and height respectively.

2. z: Represents the landmark depth with the depth at the midpoint of hips being the origin, and the smaller the value the closer the landmark is to the camera. The magnitude of z uses roughly the same scale as x.

3. visibility: A value in $[0.0, 1.0]$ indicating the likelihood of the landmark being visible (present and not occluded) in the image.

2.7 Multi-Hand Support

Multi-hand support is the capability of a hand pose recognition system to track and analyze multiple hands within the same frame simultaneously. This requires the model to distinguish between hands, assign unique identifiers, and accurately detect the landmarks for each hand. Challenges in multi-hand support include managing occlusions (where one hand partially covers the other) and handling varying hand orientations. Robust systems, like Mediapipe, employ efficient region segmentation and tracking algorithms to maintain consistent performance in multi-hand scenarios. Multi-hand support is essential for collaborative applications like multi-user gaming or shared virtual workspaces.

Chapter 3

Hand Pose Feature Representation

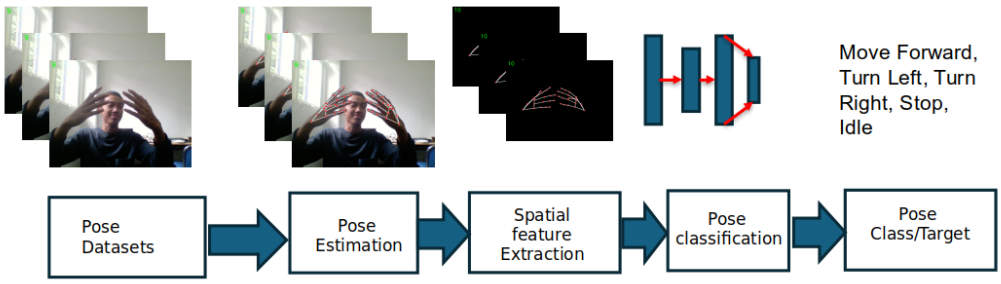

Figure 3.1: Hand Recognition Pipeline

3.1 Geometric Feature

Images captured from cameras exhibit a vast amount of variation. This is because images are spatial data, where each pixel represents a color at a specific coordinate. For pose classification, many images are required for training to account for differences in scale, position, and orientation of the hand in the images. The proposed geometric features were extracted from the coordinates of the hand joints. We used Google's MediaPipe Hands framework to track hand positions using joint coordinate landmarks. The framework provides three outputs: coordinates of landmark positions, a detection score that indicates the model's confidence in hand detection, and Handedness Classification, which identifies whether it is the left or right hand.

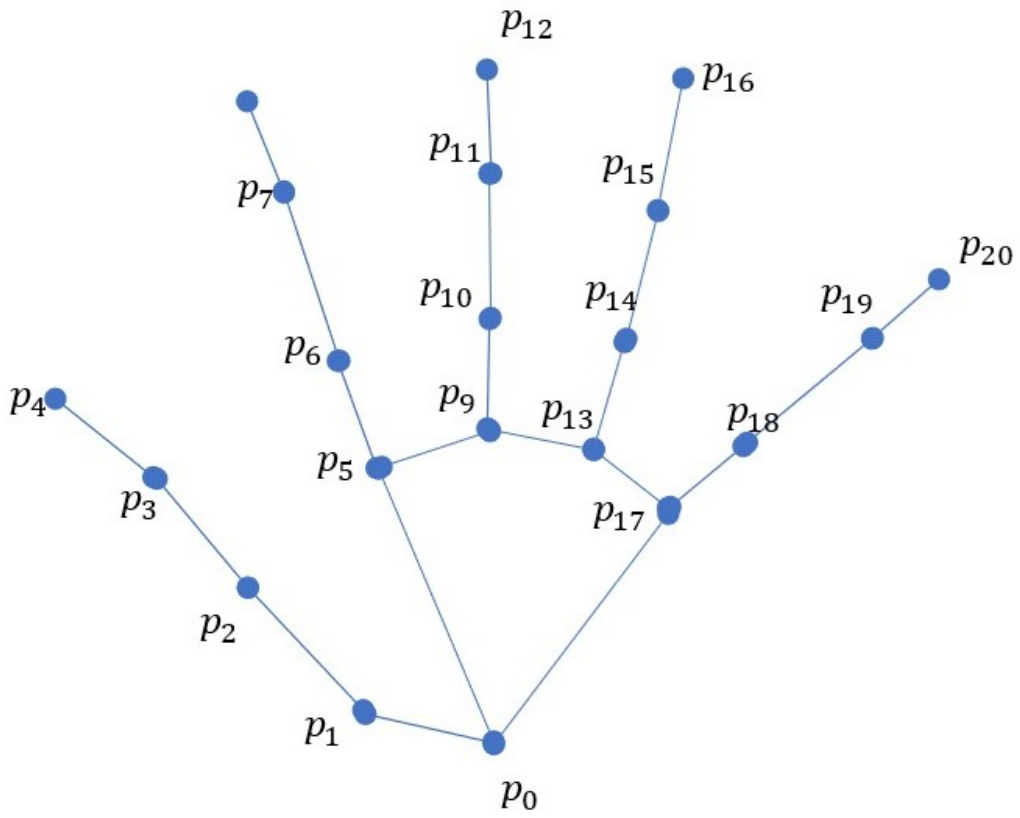

Figure 3.2: The Feature on Hand Landmark

The geometric features developed for this study were derived from the angles between the hand and joints. The selected hand joint angles are shown in Figure 3.2.

3.2 Normalized Geometric Feature

In hand pose recognition, landmarks represent specific points on the hand, such as fingertips, joints, and the wrist. These landmarks are crucial geometric features used to describe the hand's pose. However, raw landmark coordinates can vary significantly depending on factors like hand position in the frame, distance from the camera, and individual hand size. To address this, normalization relative to landmark 0 (the wrist) and landmark 5 (the base of the index finger) ensures consistency and robustness. Normalization concerning landmark 0 and landmark 5 makes the model invariant to translation, scale, and individual hand size. By anchoring all landmarks to the

wrist (landmark 0), the model ignores the absolute position of the hand in the image, focusing instead on its relative geometry. Scaling based on the distance between landmarks 0 and 5 compensates for variations in hand size or camera distance, ensuring that the features represent proportional relationships across individuals. This process reduces input variance, simplifies learning for the model, and ensures consistency in recognizing poses regardless of external conditions, such as hand position or size changes. The Figure 3.3 represents the normalized geometric features.6

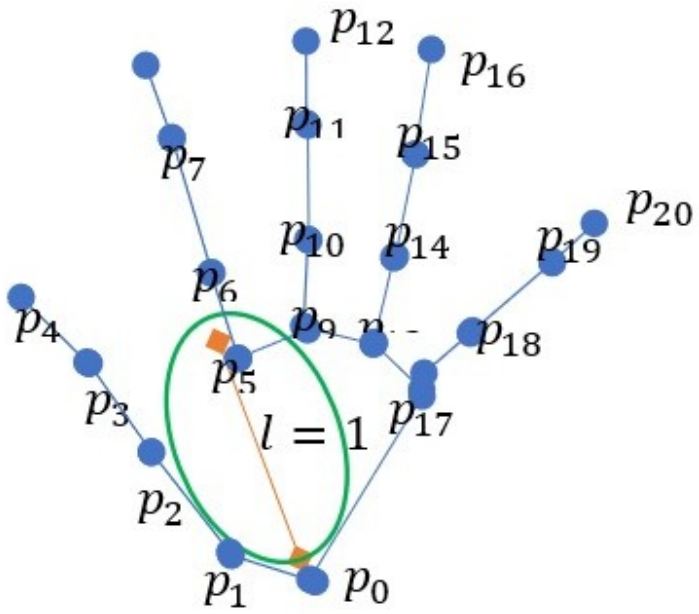

Figure 3.3: The Normalized Features on Hand Landmark

3.3 Geometric Feature Joint

In hand pose recognition, landmarks for the left and right hands are processed and combined to create a unified geometric representation. In the code, two arrays (`left_hand_landmarks` and `right_hand_landmarks`) are initialized to store the normalized landmarks for each hand. The function `process_landmarks` normalizes the landmarks relative to the wrist (landmark 0) and scales them using the distance to landmark 5. This makes the landmarks invariant to translation, scale, and hand size. When multiple hands are detected, the code uses the handedness attribute from Mediapipe

to identify whether the landmarks belong to the left or right hand. The processed landmarks are stored in their respective arrays. After normalization, both sets of landmarks are flattened into 1D arrays and concatenated using `np.concatenate`. This creates a single feature vector that includes both hands' landmarks. Combining the landmarks into one representation allows the model to analyze interactions and spatial relationships between the hands. If only one hand is detected, the unused hand's array remains at zero, ensuring a consistent feature size for the model. This approach makes the representation robust and ready for gesture or sign language recognition tasks.

Chapter 4

Hand Pose Recognition using Deep Learning

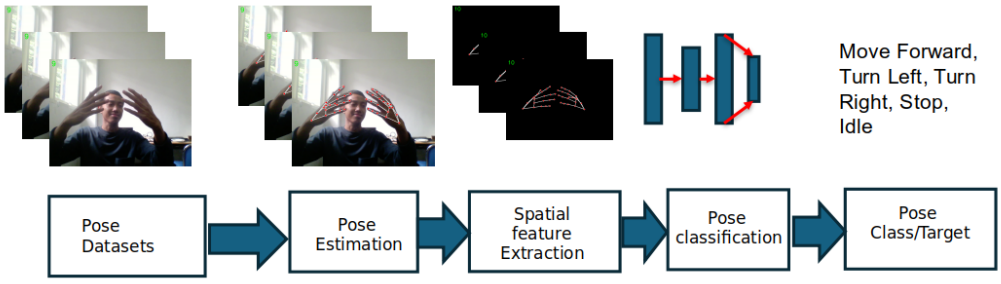

Figure 4.1: Hand Recognition Pipeline

4.1 Dataset Acquisition

This code provides a systematic way to acquire a dataset for hand pose recognition using Mediapipe and OpenCV. It captures hand landmarks and saves the corresponding frames from a webcam feed, enabling the creation of a labeled dataset for tasks like training gesture recognition models.

The process begins with the initialization and setup of necessary components. Mediapipe's Hands model is used for detecting and tracking hand landmarks, while OpenCV handles video capture and frame display. Command-line arguments allow customization of parameters such as the dataset path, label name, frame saving rate, maximum frames to save, and a delay before data collection starts. The script ensures that a directory for the dataset

is created, organizing the captured data based on the provided label name. The Figure 4.2 shows the sample of data acquisition.

Figure 4.2: Sample of Data Acquisition Process

Hand detection and landmark extraction are key steps in the process. The Mediapipe Hands model detects 21 landmarks on each hand, including knuckles and fingertips, and supports tracking up to two hands simultaneously. For every frame captured, the script converts it to RGB format and processes it using Mediapipe. Landmark coordinates are normalized relative to specific reference points, such as the wrist and a finger joint, making the data invariant to variations in hand size and position. The normalized landmarks for both hands are then concatenated into a single array, representing the frame's hand pose data.

4.2 Pose Estimation

The 'create_dataset' function is a crucial part of the pose estimation process. It facilitates the acquisition of hand pose data by capturing frames from a webcam feed, extracting hand landmarks using Mediapipe's 'Hands' solution, and organizing the data into labeled directories. This function serves as the foundation for building datasets tailored to pose estimation tasks, enabling further applications like gesture recognition and motion analysis.

Pose estimation involves detecting and interpreting key hand landmarks in real-time. The function initializes Mediapipe's 'Hands' module, which is capable of identifying 21 specific landmarks on each hand, such as fingertips, joints, and the wrist. These landmarks represent the structural configuration of the hand and are essential for understanding its pose. By converting the captured video frames to RGB format and processing them through Mediapipe, the function extracts these landmarks with high precision.

To normalize the landmark data, the function computes positions relative to key reference points—namely, the wrist and a finger joint. This normalization step ensures that the data remains invariant to hand size and camera

perspective, enhancing its generalizability for machine learning models. The landmark coordinates for both hands are flattened into a single array, representing the spatial configuration of the hands in each frame.

Beyond pose estimation, the 'create_dataset' function also incorporates real-time feedback and organization. Frames are captured and saved at user-defined intervals, allowing control over the frame rate and total number of frames collected. Each frame is stored in a labeled directory, organized by the name of the gesture or pose being recorded. The overlay of the detected landmarks on the video feed, along with a live counter of saved frames, enhances the usability of the function by providing visual confirmation and progress tracking.

This function exemplifies the integration of pose estimation techniques with practical data collection workflows. It not only extracts meaningful pose information but also structures the data in a format ready for subsequent analysis and model training. This makes it a powerful tool for researchers and developers working on applications in gesture recognition, human-computer interaction, and beyond.

4.2.1 Import required Python libraries

```
1     import cv2
2     import mediapipe as mp
3     import numpy as np
4     from datetime import datetime
5     import os
6     import time
7     import argparse
```

4.2.2 MediaPipe Initialization

```
1     mp_hands = mp.solutions.hands
2     hands = mp_hands.Hands(static_image_mode=False,
3     max_num_hands=2,
4     min_detection_confidence=0.5,
5     min_tracking_confidence=0.5)
```

This part initializes the Mediapipe Hands solution, which is responsible for detecting hand landmarks in video frames. The configuration allows real-time processing of up to two hands per frame, with parameters controlling the confidence thresholds for detection and tracking. This setup ensures accurate and stable landmark recognition during data collection.

4.2.3 Webcam Access and Directory Setup

```
    cap = cv2.VideoCapture(0)
    save_dir = os.path.join(direktori_path, nama_label)
    if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

The webcam is opened using OpenCV's VideoCapture, which serves as the data source for real-time frame capture. The save_dir is created dynamically based on the dataset path and label name, ensuring an organized directory structure for storing the collected data.

4.2.4 Landmark Extraction

```
    def ekstraksi_fitur(frame):
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)
    height, width, _ = frame.shape
    ...
    # Normalization and concatenation of landmarks
    return np.concatenate((left_hand_landmarks.flatten(),
        right_hand_landmarks.flatten()))
```

This function extracts and processes hand landmarks from each frame:

1. Conversion to RGB: Mediapipe requires frames in RGB format for processing.

2. Landmark Processing: If hand landmarks are detected, their coordinates are normalized relative to reference points (e.g., the wrist and a finger joint) to ensure invariance to hand size and position.

3. Concatenation: The landmarks for both hands are flattened and concatenated into a single array, representing the full pose information for the frame.

4.3 Geometric Feature Extraction

4.3.1 Function Definition and Initialization

```
    def create_dataset(direktori_path, nama_label,
        frameratesave=2, maxframe=100, start_delay=5):

    # Initialize Mediapipe Hands and Drawing
```

```
    mp_hands = mp.solutions.hands
    mp_drawing = mp.solutions.drawing_utils
    hands = mp_hands.Hands(static_image_mode=False,
    max_num_hands=2,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5)
```

The create_dataset function begins with its definition, which includes several customizable parameters: the dataset directory (direktori_path), the label for the dataset (nama_label), the frame saving rate (frameratesave), the maximum number of frames to save (maxframe), and the delay before starting the capture (start_delay). Inside the function, the Mediapipe Hands model is initialized. This model is set up to dynamically process video streams (static_image_mode=False) and detect up to two hands simultaneously. Detection and tracking confidence thresholds are set to 0.5, balancing accuracy and performance. The DrawingUtils module is also initialized for later use in visualizing detected landmarks.

4.3.2 Webcam Initialization and Directory Setup

```
    # Open webcam
    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
    print("Failed to open webcam.")
    return

    # Create directory path
    save_dir = os.path.join(direktori_path, nama_label)
    if not os.path.exists(save_dir):
    os.makedirs(save_dir)
    print(f"Directory created: {save_dir}")
```

Next, the function initializes the webcam using OpenCV's VideoCapture. If the webcam cannot be opened, an error message is printed, and the function exits gracefully. The function then prepares the directory for storing the dataset. If the directory does not already exist, it is created using os.makedirs. The dataset is organized by the label provided (nama_label), ensuring clear structure and traceability for the saved frames.

4.3.3 Countdown Timer and Frame Processing

Before data collection begins, the function sets up a countdown timer `start_delay` using `time.time()`. The `ekstraksi_fitur` function processes each frame. It converts the frame to RGB (as required by Mediapipe) and detects hand

landmarks using the `hands.process` method. If landmarks are detected, they are normalized relative to specific reference points (e.g., the wrist and a finger joint) to account for hand size and positioning. The normalized coordinates for the left and right hands are stored in separate arrays and concatenated for a unified representation.

```
1
2       time_before = datetime.now()
3       start_time = time.time()
4       counter = 0
5
6       def ekstraksi_fitur(frame):
7       # Convert to RGB
8       frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
9
10      # Process frame with Mediapipe
11      results = hands.process(frame_rgb)
12
13      # Frame dimensions
14      height, width, _ = frame.shape
15
16      # Initialize numpy arrays for left and right hands
17      left_hand_landmarks = np.zeros((21, 2))
18      right_hand_landmarks = np.zeros((21, 2))
19
20      # Helper function to process landmarks
21      def process_landmarks(hand_landmarks, width, height):
22      landmarks = [(lm.x * width, lm.y * height) for lm in
            hand_landmarks.landmark]
23      landmark_0 = np.array(landmarks[0])
24      landmark_5 = np.array(landmarks[5])
25      normalized_landmarks = [
26      ((x - landmark_0[0]) / (landmark_5[0] - landmark_0[0] + 1
            e-6),
27      (y - landmark_0[1]) / (landmark_5[1] - landmark_0[1] + 1e
            -6))
28      for x, y in landmarks
29      ]
30      return np.array(normalized_landmarks)
31
32      # If hands are detected
33      if results.multi_hand_landmarks and results.
            multi_handedness:
34      for hand_landmarks, hand_handedness in zip(results.
            multi_hand_landmarks, results.multi_handedness):
35      # Identify hand as left or right
36      handedness = hand_handedness.classification[0].label
```

```
    processed_landmarks = process_landmarks(hand_landmarks,
        width, height)
    if handedness == 'Left':
    left_hand_landmarks = processed_landmarks
    elif handedness == 'Right':
    right_hand_landmarks = processed_landmarks

    # Draw landmarks on the frame
    mp_drawing.draw_landmarks(frame, hand_landmarks, mp_hands
        .HAND_CONNECTIONS)

    # Concatenate left and right hand landmarks
    concatenated_landmarks = np.concatenate((
        left_hand_landmarks.flatten(), right_hand_landmarks.
        flatten()))
    return concatenated_landmarks
```

4.3.4 Frame Capture and Saving

```
    while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
    print("Failed to read frame from webcam")
    break

    # Countdown before saving starts
    elapsed_time = time.time() - start_time
    if elapsed_time < start_delay:
    countdown_text = f"Starting in {int(start_delay -
        elapsed_time)} seconds"
    cv2.putText(frame, countdown_text, (10, 50), cv2.
        FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
    cv2.imshow('Hand Landmark Detection', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
    break
    continue

    # Check framerate and save frame
    time_now = datetime.now()
    if (time_now - time_before).total_seconds() > 1 /
        frameratesave:
    # Generate file name
    file_name = datetime.now().strftime("%Y%m%d%H%M%S%f")
        [:-3] + ".jpg"
    file_path = os.path.join(save_dir, file_name)

    # Save frame
```

```
25      cv2.imwrite(file_path, frame)
26      counter += 1
27
28      # Update the previous time
29      time_before = time_now
30
31      # Display counter on frame
32      cv2.putText(frame, f"Files saved: {counter}", (10, 50),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
33
34      # Process landmarks and draw them on the frame
35      concatenated_landmarks = ekstraksi_fitur(frame)
36      print("Concatenated Landmarks:", concatenated_landmarks)
37
38      # Display the frame with landmarks
39      cv2.imshow('Hand Landmark Detection', frame)
40
41      # Stop saving after maxframe is reached
42      if counter >= maxframe:
43      print(f"Reached maximum of {maxframe} frames. Exiting..."
            )
44      break
45
46      # Handle key press
47      key = cv2.waitKey(1) & 0xFF
48      if key == ord('q'):  # Exit on 'q'
49      break
```

The webcam feed is processed frame by frame. During the countdown period, a message informs the user of the time remaining before data collection begins. Once the countdown ends, frames are saved at intervals defined by the `frameratesave` parameter. Each frame is assigned a timestamp-based filename and stored in the appropriate directory. The landmarks are processed in real-time, printed to the console, and visualized on the webcam feed using Mediapipe's drawing utilities.

4.3.5 Cleanup

```
1       # Release resources
2       cap.release()
3       cv2.destroyAllWindows()
4       hands.close()
```

After reaching the specified frame limit or upon user interruption, the function releases the webcam and closes all OpenCV windows. The Mediapipe Hands object is also closed, ensuring that all resources are properly deallo-

cated. This function effectively combines pose estimation, real-time feedback, and dataset organization, making it a comprehensive tool for collecting hand pose data.

4.4 Model Architecture

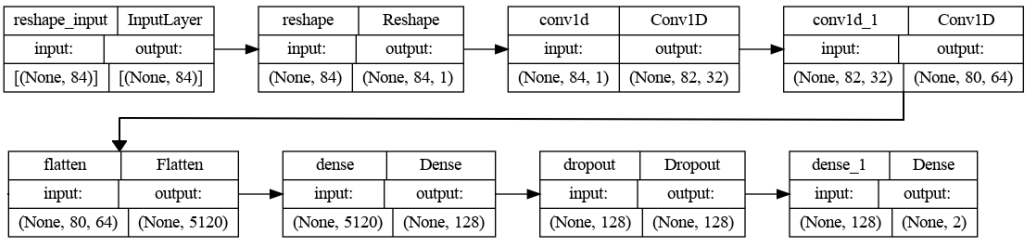

Figure 4.3: Model Architecture

This model represents a simple feed-forward neural network designed for tasks such as classification, particularly for hand pose recognition. The architecture utilizes Convolutional Neural Network (CNN) model implemented in TensorFlow, specifically designed for tasks involving 1D data, such as time-series analysis, signal processing, or structured data (like hand pose landmarks).

```
1    Reshape((input_size, 1), input_shape=(input_size,))
```

This layer reshapes the 1D input vector of size `(input_size,)` into a 2D array with dimensions `(input_size, 1)`. This transformation is necessary because Conv1D layers require a 2D input format where one dimension represents the sequence length `(input_size)`, and the other represents the number of channels (here, a single channel).

The reshaped input is then passed to the first Conv1D layer:

```
1    Conv1D(filters=32, kernel_size=3, activation='relu')
```

This layer applies 32 convolutional filters with a kernel size of 3 to extract local patterns from the data. The `relu` activation introduces non-linearity, enabling the model to learn complex features. The next layer is another Conv1D layer:

```
1    Conv1D(filters=64, kernel_size=3, activation='relu')
```

This layer increases the number of filters to 64, allowing it to capture more complex patterns and higher-level features from the output of the previous

layer. Both convolutional layers reduce the dimensions of the input while retaining its essential features.

After the convolutional layers, a Flatten layer:

```
Flatten()
```

is used to convert the multi-dimensional output from the Conv1D layers into a 1D vector. For example, if the output from the second Conv1D layer is of shape (80, 64), the Flatten layer transforms it into a vector of size (80 * 64 = 5120). This step prepares the data for the subsequent fully connected layers.

The flattened output is fed into a Dense layer:

```
Dense(128, activation='relu')
```

which consists of 128 neurons. Each neuron applies a linear transformation followed by a `relu` activation function to learn combinations of features extracted by the convolutional layers. This dense layer reduces the feature space while focusing on the most significant patterns.

To prevent overfitting, the model includes a Dropout layer:

```
Dropout(0.5)
```

which randomly deactivates 50% of the neurons during training. This regularization technique ensures that the model does not rely too heavily on specific neurons, improving its ability to generalize to unseen data.

Finally, the model ends with an output Dense layer:

```
Dense(num_classes, activation='softmax')
```

This layer has a number of neurons equal to the number of target classes (`num_classes`). The `softmax` activation function converts the raw outputs into probabilities for each class, ensuring that the sum of all probabilities equals 1. This makes it suitable for multi-class classification tasks.

Overall, the model starts with reshaping the input, extracts meaningful features using convolutional layers, flattens the extracted features, and uses dense layers for classification, with dropout for regularization. The final softmax layer outputs class probabilities, making it effective for tasks like hand pose classification or time-series analysis.

Chapter 5

The Experiments of Hand Pose Recognition

5.1 Hand Pose Data Training Preparation

Function: `read_landmarks`

The `read_landmarks` function is designed to extract hand pose landmarks from a dataset of labeled images using Mediapipe's `Hands` module. It processes all images in subdirectories corresponding to different labels, extracts features, and prepares the data for machine learning tasks.

```
def read_landmarks(dataset, labels):
    # Initialize Mediapipe Hands and Drawing
    hands = mp_hands.Hands(static_image_mode=False,
    max_num_hands=2,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5)

    kelas = np.eye(len(labels))
    y = []

    images_by_label = {}
    X = []
    for i, label in enumerate(labels):
        # Construct the directory path
        directory_path = os.path.join(dataset, label)

        # Check if the directory exists
        if not os.path.exists(directory_path):
            print(f"Directory '{directory_path}' does not
                exist.")
            images_by_label[label] = []
```

```
            continue

    # List all '.jpg' files in the directory
    filenames = [f for f in os.listdir(directory_path) if f.
        endswith('.jpg')]

    # Read images and store them as frames
    # frames = []
    for filename in filenames:
        file_path = os.path.join(directory_path, filename)
        frame = cv2.imread(file_path)  # Read the image with
            OpenCV
        fitur = feature_extract(frame, hands)
        X.append(fitur)
        y.append(kelas[i])

    hands.close()
    return np.array(X), np.array(y)
```

Listing 5.1: The `read_landmarks` function

Explanation

The `read_landmarks` function performs the following key tasks:

1. Initialize Mediapipe Hands

- `static_image_mode=False`: Allows the function to dynamically process images or frames.
- `max_num_hands=2`: Configured to detect up to two hands in an image.
- `min_detection_confidence=0.5`: Sets the minimum confidence threshold for hand detection.
- `min_tracking_confidence=0.5`: Ensures reliable tracking of hand landmarks once detected.

2. One-Hot Encode Labels

A one-hot encoded matrix (`kelas`) is created for the labels using:

```
    kelas = np.eye(len(labels))
```

For example, if there are three labels:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The list `y` is initialized to store the corresponding one-hot encoded labels for each image.

3. Process Each Label Directory

The function iterates through the given labels:

```
    for i, label in enumerate(labels):
    directory_path = os.path.join(dataset, label)
```

It constructs the full directory path for each label's subdirectory. If the directory does not exist, a warning is printed, and an empty list is assigned to that label.

4. Read and Process Images

For each valid directory, the function lists all filenames ending with `.jpg`:

```
    filenames = [f for f in os.listdir(directory_path) if f.
       endswith('.jpg')]
```

It then reads each image using OpenCV:

```
    frame = cv2.imread(file_path)
```

Hand pose features are extracted using the `feature_extract` function:

```
    fitur = feature_extract(frame, hands)
```

The extracted features are appended to the `X` list, and the corresponding one-hot encoded label is added to the `y` list.

5. Close Mediapipe Resources and Return Data

Once all images are processed, the Mediapipe `Hands` module is closed:

```
    hands.close()
```

Finally, the function returns `X` (a NumPy array of extracted features) and `y` (a NumPy array of one-hot encoded labels).

Output

- `X`: A NumPy array containing the extracted hand pose features for all images.
- `y`: A NumPy array of one-hot encoded labels corresponding to each feature in `X`.

Practical Use Case

The `read_landmarks` function is particularly useful in preprocessing datasets for hand pose recognition tasks. By extracting and normalizing geometric features from images, it prepares the data for downstream machine learning models.

The function `read_landmarks` is designed to process a dataset of hand pose images, extract features using Mediapipe, and prepare the data for machine learning models. Specifically, it:

1. Traverses a dataset organized into subdirectories, where each subdirectory represents a label.
2. Reads all `.jpg` images within each subdirectory.
3. Extracts hand pose features from each image using Mediapipe's `Hands` module.
4. Prepares feature data (`X`) and one-hot encoded labels (`y`) for training or testing a machine learning model.

5.1.1 Function Parameters

- `dataset (str)`: The root directory containing subdirectories, each corresponding to a label.
- `labels (list)`: A list of label names, where each label matches a subdirectory name within `dataset`.

Step 1: Initialize Mediapipe Hands

The function initializes Mediapipe's `Hands` module to process the images:

- `static_image_mode=False`: Operates in dynamic mode for handling multiple frames or videos.

- `max_num_hands=2`: Configured to detect and process up to two hands per image.
- `min_detection_confidence=0.5`: Specifies the minimum confidence threshold for detecting a hand in an image.
- `min_tracking_confidence=0.5`: Specifies the minimum confidence required to track hand landmarks reliably.

Step 2: One-Hot Encode Labels

```
    kelas = np.eye(len(labels))
    y = []
```

A one-hot encoded matrix (`kelas`) is created for the labels. For instance, if there are three labels, the one-hot encoded representation would look like:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The `y` list is initialized to store the labels corresponding to each image.

Step 3: Iterate Over Labels

The function loops through each label and constructs the full directory path for each subdirectory:

```
    for i, label in enumerate(labels):
    directory_path = os.path.join(dataset, label)
```

Step 4: List All `.jpg` Files

To retrieve all the image files within each subdirectory, the function uses:

```
    filenames = [f for f in os.listdir(directory_path) if f.
       endswith('.jpg')]
```

This collects only filenames with a `.jpg` extension from the directory.

Step 5: Read and Process Images

The function processes each image file in the directory:

```
    for filename in filenames:
    file_path = os.path.join(directory_path, filename)
    frame = cv2.imread(file_path)
    fitur = ekstraksi_fitur(frame, hands)
    X.append(fitur)
    y.append(kelas[i])
```

For every image:

1. The full file path is constructed using `os.path.join`.
2. The image is read using OpenCV's `cv2.imread`.
3. The `ekstraksi_fitur` function is called to extract hand pose features from the image using Mediapipe.
4. The function normalizes hand landmarks for left and right hands.
5. The extracted features are appended to `X`, and the corresponding one-hot encoded label is appended to `y`.

Output

The function returns the following:

- `X`: A NumPy array containing the extracted features for all images in the dataset.
- `y`: A NumPy array of one-hot encoded labels corresponding to each set of features in `X`.

Purpose of Concatenating Hand Features

In the feature extraction step, landmarks from both the left and right hands are concatenated into a single vector. This unified representation captures spatial relationships between the hands, enabling the model to understand interactions between the two hands (e.g., in sign language or complex gestures). If only one hand is detected, the unused hand's feature vector is set to zeros, ensuring consistency in the feature size. This approach simplifies model training and ensures robustness in handling frames with varying numbers of detected hands.

Example Usage of `read_landmarks`

The following example demonstrates how to use the `read_landmarks` function to prepare a dataset for training and testing a machine learning model. It includes the steps for defining the dataset and labels, calling the function, and splitting the data into training and testing sets.

```
# Example usage
dataset = "../dataset"                    # Base dataset directory
labels = ["One", "Two", "Three"]          # List of labels (subdirectories)

# Call the function
X, y = read_landmarks(dataset, labels)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=40)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

Listing 5.2: Example Usage of `read_landmarks`

Explanation

- `dataset`: Specifies the base directory containing the dataset. Each label should have a corresponding subdirectory in this path containing the images for that label.
- `labels`: A list of subdirectory names representing the classes (e.g., "One", "Two", "Three"). Each label corresponds to a unique class in the dataset.

The `read_landmarks` function is called with the dataset directory and the list of labels as arguments. It returns two outputs:

- `X`: A NumPy array containing the extracted features for all images in the dataset.
- `y`: A NumPy array containing the one-hot encoded labels for each image.

The dataset is then split into training and testing subsets using `train_test_split`, a function from the `sklearn.model_selection` module. The arguments are as follows:

- `test_size=0.2`: Specifies that 20% of the data is reserved for testing.

- `random_state=40`: Ensures reproducibility by fixing the random seed.

Finally, the shapes of the resulting datasets are printed:

- `X_train.shape`: The shape of the training feature set.
- `y_train.shape`: The shape of the training label set.
- `X_test.shape`: The shape of the testing feature set.
- `y_test.shape`: The shape of the testing label set.

This example shows how to preprocess the dataset and prepare it for use in a machine learning workflow.

5.2 Model Training and Evaluation

The following code demonstrates the process of configuring, training, and evaluating a machine learning model for hand pose recognition. It includes the initialization of hyperparameters, model creation, compilation, training, evaluation, and saving.

```
    # Parameters
    input_size = X_train.shape[1]  # Number of features (
        flattened landmarks)
    num_classes = len(labels)      # Number of labels
    batch_size = 32                # Batch size for
        DataLoader
    learning_rate = 0.01           # Learning rate
    num_epochs = 200               # Number of epochs

    from keras.optimizers import SGD

    # Create model, loss function, and optimizer
    model = get_model(input_size=input_size, num_classes=
        num_classes)

    # Display the model summary
    model.summary()

    # Compile the model
    model.compile(optimizer=SGD(learning_rate=learning_rate),
    loss='categorical_crossentropy',
    metrics=['accuracy'])

    # Train the model
    history = model.fit(
```

```
23          X_train, y_train,
24          validation_split=0.2,
25          epochs=num_epochs,
26          batch_size=batch_size,
27          verbose=1
28      )
29
30      # Evaluate the model
31      test_loss, test_accuracy = model.evaluate(X_test, y_test,
            verbose=0)
32      print(f"Test Accuracy: {test_accuracy:.2f} %")
33
34      output_path = "outputs"
35      model_name = "model.keras"
36      if not os.path.exists(output_path):
37          os.makedirs(output_path)  # Create directory if it
              doesn't exist
38          print(f"Directory '{output_path}' created.")
39      model.save(os.path.join(output_path, model_name))
```

Listing 5.3: Model Training and Evaluation

Explanation

The process is divided into multiple steps:

1. Define Hyperparameters

- `input_size = X_train.shape[1]`: Determines the number of input features, which corresponds to the number of flattened landmarks extracted from the dataset.
- `num_classes = len(labels)`: Defines the total number of classes in the dataset based on the length of the `labels` list.
- `batch_size = 32`: Specifies the number of samples to be processed in one training batch.
- `learning_rate = 0.01`: Sets the learning rate for the optimizer, which controls the step size during weight updates.
- `num_epochs = 200`: Specifies the number of complete passes through the training dataset.

2. Create and Compile the Model

The model is created using the `get_model` function, which takes the `input_size` and `num_classes` as parameters. The model summary is displayed using:

```
model.summary()
```

This provides a detailed view of the model's architecture, including the number of layers, parameters, and input/output shapes.

The model is then compiled using the Stochastic Gradient Descent (SGD) optimizer:

```
model.compile(optimizer=SGD(learning_rate=learning_rate),
loss='categorical_crossentropy',
metrics=['accuracy'])
```

- `SGD`: A simple optimizer that updates weights using gradients calculated from the loss function.
- `categorical_crossentropy`: The loss function used for multi-class classification problems.
- `metrics=['accuracy']`: Specifies accuracy as the evaluation metric during training.

3. Train the Model

The model is trained using the `fit` method:

```
history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=num_epochs,
    batch_size=batch_size,
    verbose=1
)
```

- `X_train, y_train`: The training data and corresponding labels.
- `validation_split=0.2`: Reserves 20% of the training data for validation.
- `epochs=num_epochs`: The model is trained for the specified number of epochs.
- `batch_size=batch_size`: Processes samples in batches of size 32.
- `verbose=1`: Enables detailed logging of the training progress.

4. Evaluate the Model

The trained model is evaluated on the test dataset using:

```
    test_loss, test_accuracy = model.evaluate(X_test, y_test,
        verbose=0)
```

The test loss and accuracy are calculated, and the accuracy is printed:

```
    print(f"Test Accuracy: {test_accuracy:.2f} %")
```

5. Save the Model

The trained model is saved to a directory:

```
    if not os.path.exists(output_path):
    os.makedirs(output_path)  # Create directory if it doesn'
        t exist
    model.save(os.path.join(output_path, model_name))
```

- `output_path`: The directory where the model will be saved.
- `model_name`: The name of the saved model file.
- `os.makedirs`: Ensures that the directory is created if it does not already exist.

Practical Use Case

This process is designed for training and evaluating a deep learning model for hand pose recognition. The hyperparameters can be adjusted based on the complexity of the dataset, and the model can be reused for predictions by loading the saved file.

5.3 Hand Pose Classification

5.4 Hand Pose Inference