Controlling the drone with hand gestures

(Sterowanie dronem za pomocą gestów)

Aleksander Szymański

Praca inżynierska

Promotor: Marek Adamczyk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

09.09.2022

Abstract

Drones are nowadays one of the most popular technical innovations, yet they often need a particular controller for operating (sometimes unique for specific robot models). We will try to overcome this problem by proposing a solution that allows us to operate the drone with the use of static hand gestures, and we will try to expand it with whole motions. Recognizing gestures will be based on a fundamental analysis of the mutual position of points in space and a classification neural network. The whole system will be integrated with ROS2, a framework popular among commercial robots currently available on the market.

Drony są obecnie jedną z bardziej popularnych nowinek technicznych, jednak do operowania nimi często wymagany jest specjalny (często unikatowy dla modelu robota) kontroler. Postaramy się ominąć ten problem, proponując rozwiązanie pozwalające na sterowanie urządzeniem za pomocą statycznych gestów dłoni, a także spróbujemy je rozwinąć o całe jej ruchy. Rozpoznawanie gestów będzie oparte na metodach wykorzystujących podstawową analizę wzajemnego położenia punktów w przestrzeni, jak i o klasyfikującą sieć neuronową. Cały system będzie zintegrowany z ROS2, frameworkiem popularnym wśród komercyjnych robotów dostępnych obecnie na rynku.

Contents

Chapter 1.

Introduction

1.1. Drone steering

Drones are one of the most popular bits of tech nowadays. They are getting cheaper, easier to use, and more accessible for everybody. With high-quality cameras on board, people are starting to be inclined to choose a drone over a camera. But even with special user support like flight assistant, each drone requires a way to operate - some of them need a mobile App, that you have to install on your phone, and others even have a unique controller made just for the model - figure 1.1 shows the new model of DJI RC controller for the latest models of drones from the Mavic series.

Figure 1.1: DJI RC controller. (Source: [1])

Therefore instead of buying a controller, we decided to operate the drone universally - using a simple ps4 controller with our program using the drone's API. After completing this functionality, we thought about if there exists an even more universal solution for this task and got an idea about controlling the drone using bare hands - without any physical device.

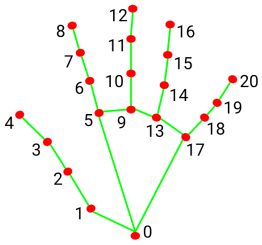
1.2. Drone model

To complete this task, we needed a drone. It wasn't easy to choose a device, considering the number of different products on the market. After a short study, we decided to go with a DJI Ryze Tello (figure 1.2) as it is the cheapest one we could find among the programmable drones. Also, the drone's size is in its favor as it allows us to use it without complicated legal powers.

Figure 1.2: DJI Ryze Tello drone. (Source: [2])

1.3. MediaPipe Hands

Our idea about controlling the drone using bare hands is based on mapping hand gestures (static and dynamic) to robot instructions. But first, we needed some way of recognizing the gestures. The thing that we were lacking was [3] MediaPipe Hands - a pipeline for hand and finger tracking. It uses multiple machine learning models for detecting and tracking even several hands (figure 1.4). Each detection defines a high-quality 3D hand mask consisting of 21 3D points shown in figure 1.3. Moreover, the pipeline is well optimized - it doesn't need a powerful desktop environment, as it can perform real-time hand tracking on mobile devices.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 1.3: 21 points from the hand mask with their IDs. (Source: [4])

The efficiency and ease of use have made us use MediaPipe Hands in all our programs

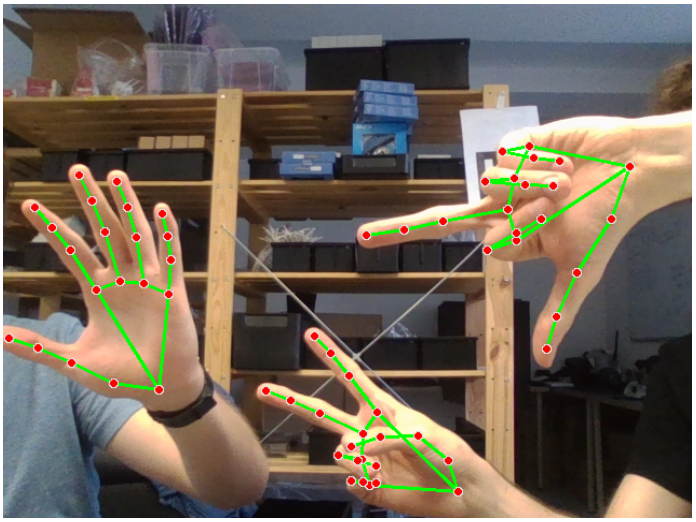


Figure 1.4: Hand masks drawn on the detected multiple hands.

controlling the drone using hand gestures (more details described in chapter 2.).

1.4. Chapters summary

Here you can find a short summary of the following chapters:

- 2.. Gesture recognition - description and analysis of the implemented solutions for gesture recognition.
- 3.. ROS - simple explanation about what Robot Operating System is, description of outsourced packages used, presented ROS environment structure and ROS API documentation of implemented programs.
- 4.. Implementation - overview of the implementation of crucial programs for this project
- 5.. Results - short description of obtained results.
- 6.. Instruction - links to tutorial on running and building the project, and instructions on steering the drone using gestures.
- 7.. Further work - new ideas and changes to apply in the current version of the project to make it more useful.

Chapter 2.

Gesture recognition

As we said in chapter 1., all gesture recognition solutions use hand masks returned by MediaPipe Hands. Some of them use neural networks for analyzing the structure of the hand, and some of them use the mathematical relations between points in 3D space.

2.1. "If-else"

The first implemented solution, and the simplest as well. The name of the subsection is self-explanatory, as the rule of thumb is that the program takes the masks from MediaPipe and analyses 21 positions of the landmarks using if-else statements, and based on the outcome determines what gesture the hand depicts. Each hand sign has its short if-else statements sequence, that describes the gesture as precisely as possible.

Looking back at figure 1.3, we deduced that there is no need to compare every two landmarks. To identify a gesture we need to determine the position of each finger, so for example, for the hand in figure 2.1 (which depicts the gesture "flip-back"), we would just check whether the TIP of each finger is above the finger's PIP (except for the thumb, there we would check the TIP with MCP - according to the IDs on figure 1.3).

Checking fingers' position is enough when you are making the gesture with your hand, but it's prone to false positives as you can rotate your hand and keep fingers in the specified position - which will result in recognizing some gesture when there is none. Also, there occurred a problem when the program was tested by other people. They couldn't twist their hand in positions required for some gestures, such as the thumb pointing precisely downwards with a straight arm.

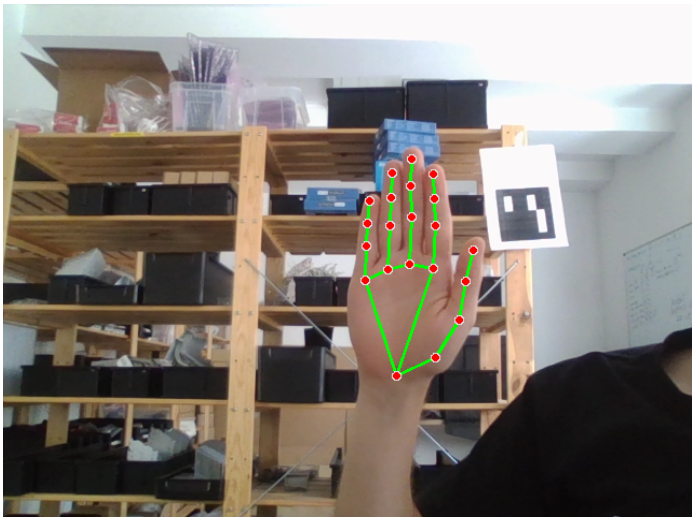


Figure 2.1: Hand mask for hand gesture “flip back”.

2.2. Classification neural network

The problems with the ”if-else” solution were very onerous in the long run, and we had two options for fixing them. We could add more clauses to the ”if” instructions, which might work but would be boring. On the other hand, we could try to use machine learning to recognize our gestures. The second option was much more interesting, so we decided to make a classification neural network trained on hand masks from MediaPipe. Looking again at the shape of returned landmarks (21x3), we decided to treat the masks like 1D images with three color channels and used convolutional layers.

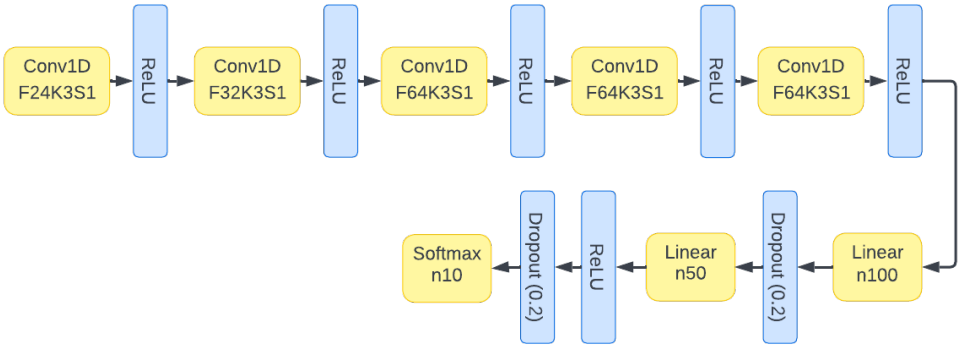


Figure 2.2: Classification neural network architecture for recognizing static hand gestures.

Looking at the whole net’s architecture (figure 2.2), there are five, 1D convolutional layers at the beginning with the number of kernels respectively: 24, 32,

64, 64, 64 (the first three having kernel size three, and the last two having kernel size two). All convolutional layers have also a stride equal to one, and ReLU as an activation function. Then the output from the last convolution is flattened and passed through two dense layers and dropouts (both 20% rate). In the end, there is one more dense layer, with softmax as an activation function, that outputs the probability distribution for the ten classified gestures (listed in chapter 6.2.).
After training, we converted the model to a TensorFlow Lite, to increase the performance.

As we wanted custom gestures to be recognized, we had to create our dataset. Therefore we implemented the *hand_mask_recorder* node - a program that records the detected hand mask for a given time and saves stored data into a pickle format file (chapter 4.4.). For every gesture, we recorded three files, so we had the split for train, validation, and test sets ready. Data distribution for the created dataset is presented in figure 2.3.

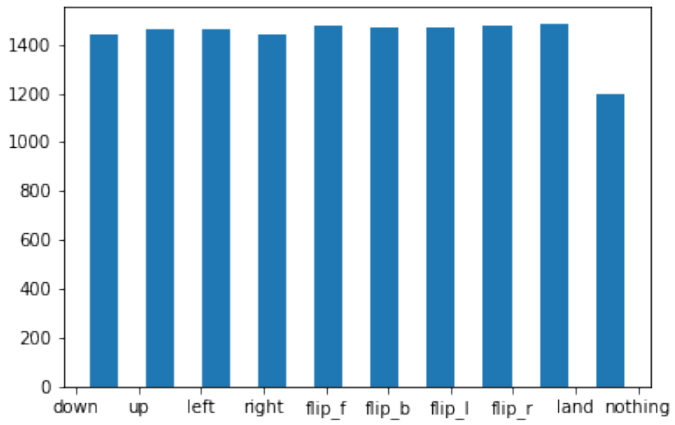


Figure 2.3: Data distribution of samples per class for the hand gestures dataset.

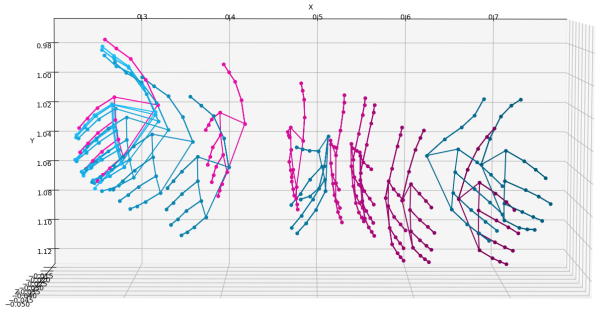
2.3. Timeseries classification neural network

With satisfying results for gesture recognition, we decided to go even further and try recognizing the whole movement instead of a static pose of the hand. Completing this task could give additional possibilities for defining instructions for the robot that are hard to show with a single gesture.

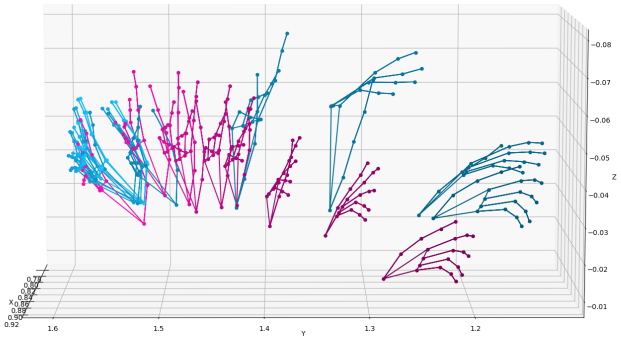
The intuitive approach for recognizing movement would be to use recurrent neural networks however, after reading an article from Keras ([5]), we decided to treat the problem as timeseries classification and use the same network architecture as in figure 2.2.

In contrast to recognizing gestures, creating a dataset for movement recognition

wasn't an easy task. It is possible to record sliding the hand in the same way at least 500 times, and this is only for one gesture, but obviously, it would be very time-consuming. Therefore we had to come up with a better solution. With the use of the *hand_mask_recorder* (section 4.4.), we recorded a few hand movements for each movement type we wanted to recognize. Then after inspection, we chose the one depicting the motion in the best way. Next, we used the selected files for generating a specified number of samples of a given movement using the *data_generator.py* (section 4.5.). Figure 2.4 presents examples of the generated data.



(a) Movement left



(b) Movement forward (shown from the side)

Figure 2.4: Examples of the generated data for the movement dataset. The blue color depicts the recorded sample, and the purple generated data. The movement direction is marked with a color gradient - a bright shade of color means the beginning of the movement, and a darker one means the end.

Although the model trained well and is capable of recognizing some movements, we didn't manage to wrap it up into a working program. Therefore this approach

wasn't used after all, as there were too many problems with making the model work with the drone, and recognize all the movement types.

Chapter 3.

ROS

ROS (Robot Operating System) is a set of open-source libraries and tools for building robot applications. It's a popular framework used on (or is easy to integrate with) most of the robots available on the market. It was started in 2007, and since then had a lot of updates hence ROS2 was made and is under development.

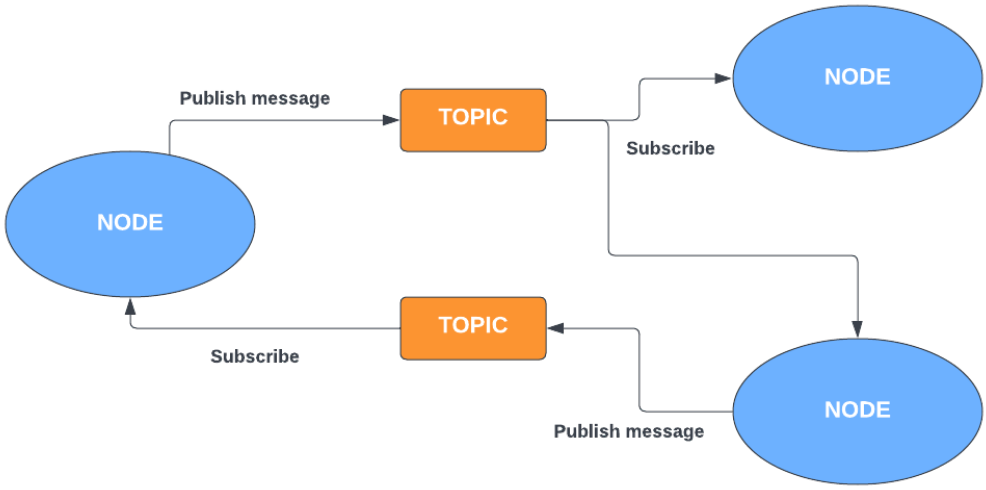


Figure 3.1: Simple ROS structure.

In simplification, ROS works like this - executable programs are called "nodes" and can communicate through topics. Each topic has a type (determines the type of messages that the topic accepts) and a unique name. When one node publishes something on a given topic, every node that listens to that topic (has a subscriber on that topic) will receive the message. Each node can listen to and publish messages on many topics. Simple visualization of the description in figure 3.1.

As the project runs on a robot, we decided to use the framework and organize the programs in ROS packages, so it'll be easy to integrate with other devices for

potential future users. Having experience with ROS1 and being a newbie to ROS2, we chose to develop our project for the newer version of the framework to gain experience.

3.1. tello_ros

Since the ROS community is large, and most of the tools are open source, before implementing our integration of the DJI Tello drone into the ecosystem, we decided to check the internet. We found that there already exists a ROS abstraction on the Tello SDK (API for programming the DJI Tello drone), written for ROS2. The repository is called "tello_ros" and was made by Clyde McQueen. It contains four ROS packages that provide topics and services for essential communication with the drone via ROS. Full documentation on [6].

3.2. ROS structure

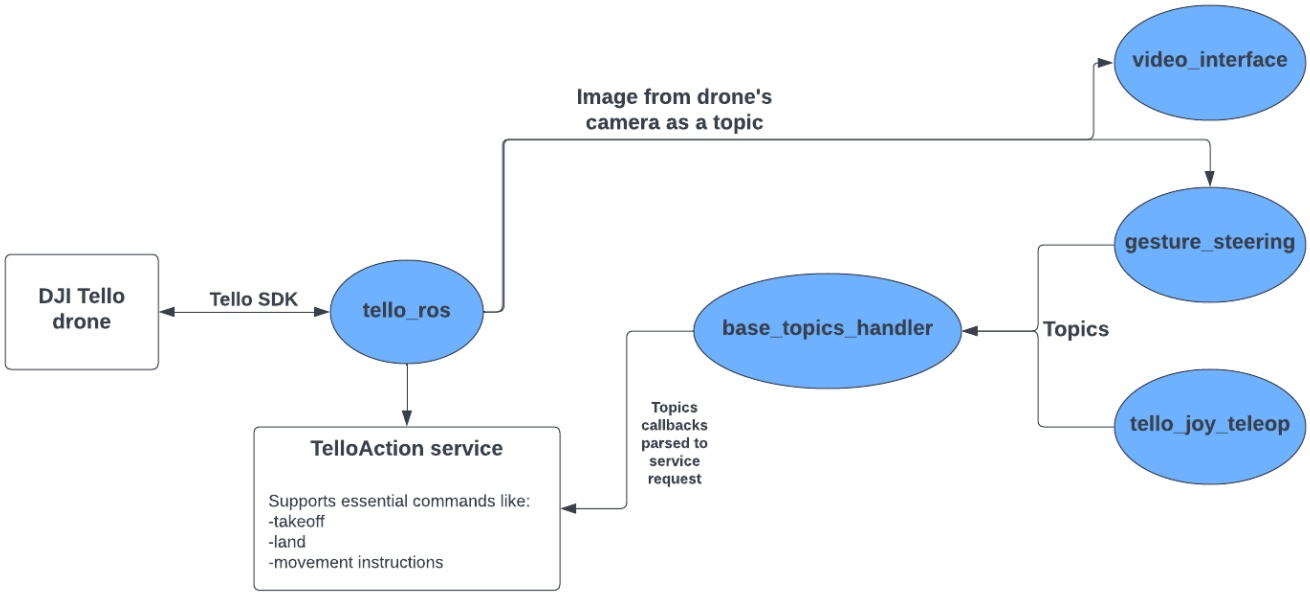


Figure 3.2: Simplified ROS structure of the project.

Figure 3.2 depicts a simplified structure of the project's ROS architecture. Direct communication with the drone goes through Tello SDK and is managed by an outsourced package. The package provides a camera image from the drone and a TelloAction service whose request takes command from the SDK as strings. Then we have the *base_topics_handler* node which communicates with the drone using the service and provides topics for every needed command. When a message on a given

topic comes, it is parsed to a correct request for the service. The topics are used by other nodes like *tello_joy_teleop* (responsible for operating the drone with ps4 controller), *video_interface* (adding UI to the image from the drone's camera), and gesture steering (operating the drone with bare hands). For a detailed communication diagram, see figure 3.3. Each node there is marked as an ellipse and has the published/subscribed topics presented as lines with an arrowhead on the endpoint.

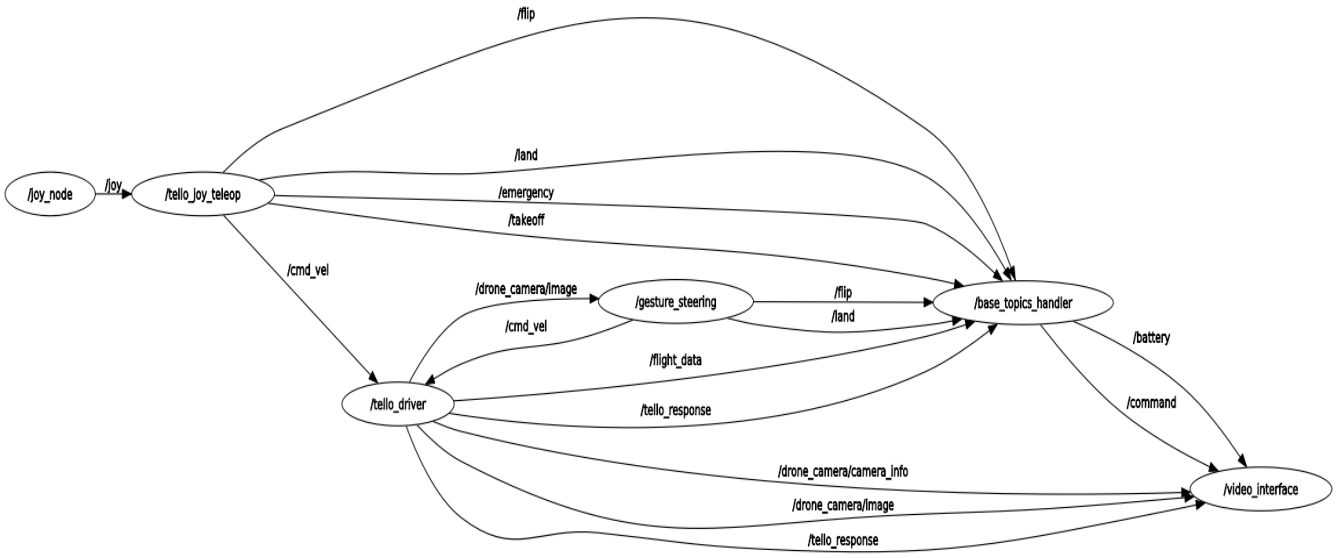


Figure 3.3: Complex ROS structure with all active nodes and topics.

3.3. ROS API

This section contains technical description of ROS interface for nodes not described in chapter 4..

3.3.1. base_topics_handler

Subscribed topics

- *flip* (std_msgs/String)
  Topic responsible for flip commands; takes one of four flip directions ("l", "r", "u", "p").
- *land* (std_msgs/Empty)
  Topic responsible for land command; takes empty messages - works like a trigger.
- *emergency* (std_msgs/Empty)
  Topic responsible for emergency command (power off the drone); takes empty

messages - works like a trigger.

- *takeoff* (std_msgs/Empty)
  Topic responsible for takeoff command; takes empty messages - works like a trigger.

- *flight_data* (tello_msgs/FlightData)
  Topic responsible for getting info about drone state.

Published topics

- *battery* (std_msgs/Int16)
  Info about the current battery level.

- *command* (std_msgs/String)
  The current command sent to the drone; is needed for the video interface.

3.3.2. tello_joy_teleop

Subscribed topics

- *joy* (sensor_msgs/Joy)
  Topic responsible for getting data from joystick about the buttons and axes states.

Published topics

- *cmd_vel* (geometry_msgs/Twist)
  Moving and rotating the drone.

- *emergenvy* (std_msgs/Empty)
  Power off the drone; works like a trigger.

- *flip* (std_msgs/String)
  Performing flip in one of given directions ("l" - left, "r" - right, "f" - front, "b" - back).

- *land* (std_msgs/Empty)
  Landing the drone; works like a trigger.

- *takeoff* (std_msgs/Empty)
  Starting the drone; works like a trigger.

3.3.3. video_interface

Subscribed topics

- *battery* (std_msgs/Int16)
  Displaying info on the interface about the current battery level.
- *command* (std_msgs/String)
  Displaying on the interface the current command.
- *drone_camera/image* (sensor_msgs/Image)
  Displaying the view from the drone camera.
- *tello_response* (tello_msgs/TelloResponse)
  Displaying frame upon getting a response for sent command.

Chapter 4.

Implementation

This section describes the implementation of programs connected with gesture steering and dataset building. As the project was developed for ROS2, the code has a characteristic structure for the framework. Each program is either a single ROS node or a module, so every file contains one class (ready to be imported from other files) and works on callbacks.

4.1. hand_tracking.py

This file contains one class (*HandTracker*) which is used by other programs for hand mask detection. The class implements the use of the MediaPipe module and provides methods for getting normalized landmarks or mapped to the image size (for drawing purposes).

4.2. math_gesture_interpreter.py

Serves as another python module used in other programs. It implements the solution for gesture recognition described in section three. The main class in this program defines an independent method for each allowed gesture to check if the given sign is present on the detected landmarks. Each function has the if clauses sequence and returns a boolean specifying whether or not the gesture is depicted.

4.3. gesture_steering

The program that uses both modules described in sections 4.1. and 4.2.. It's a ROS node, which gives the drone instructions based on recognized gestures. It supports choice of interpreting method (explained in sections 2.1. and 2.2.) as a command line argument. All needed subscribers and publishers are made at the start of the

program (*__init__* method), and all the logic is covered with the use of callback methods.

4.4. hand_mask_recorder

Another ROS node. It was made just for the creation of the dataset for the neural network. Gets the image from the drone, runs hand detection on the received image, and stores the detected hand masks. When the given time is up, it saves the stored data into a pickle file specified by the path. The program handles all the configuration parameters as duration time and output file name as command line arguments.

4.5. data_generator.py

The Data Generator is a standalone program made just for the task of creating the movement dataset. The whole generation is based on the cubic bezier curves and recorded motion samples. The program calculates the bezier formula for each of 21 landmarks from the hand mask which makes it 21 bezier curves needed to generate hand movement in total. The points for the formula are sampled from the recorded movement (except start and end points, those are always given as the first and last hand from the recorded data). Once we have the formula, we can generate any number of points on the curve. We do that for each landmark point and get a generated hand movement. To not produce the same sample every time, to each hand on the curve is applied random offset. Therefore each generated data is unique but still depicts the same movement type.

Chapter 5.

Results

In general, the project ended with success - there is a working gesture steering system, with two interpreters, that perform well (figure 5.1 presents training and validation loss and accuracy for trained neural network). It's hard to measure the effectiveness of such a system as there are no metrics or other validation methods. The system was checked in field tests done by a few people, who tested the program for running smoothness, hand detection, and accuracy of gesture interpretation. Feedback from those tests was very positive - both interpreters work fast enough to make the drone perform smooth changes in the movement direction. The interpretation process doesn't affect the frame rate, so all the time, the image from the drone is of the highest possible quality.

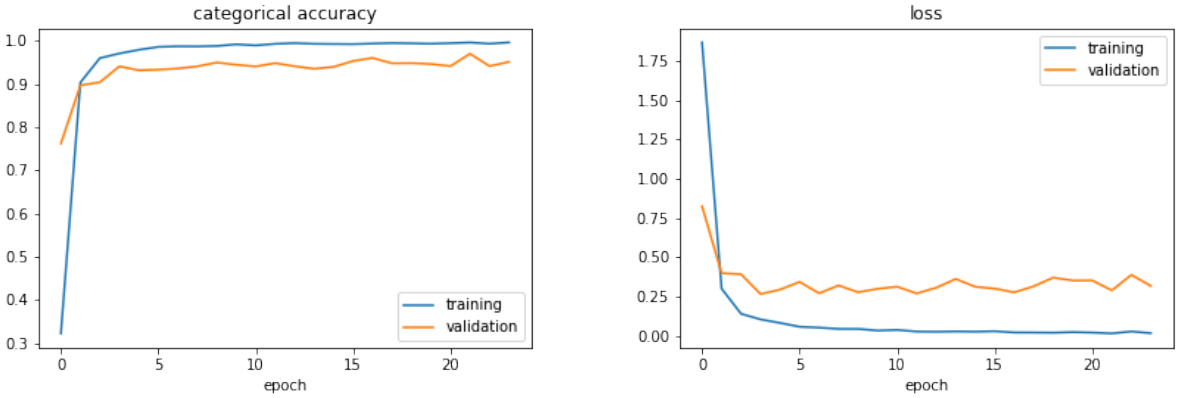


Figure 5.1: Loss and accuracy for neural network model classificating hand gestures.

Although the system works well, there is a part that doesn't work. Making a system that recognizes the specific motions and maps them to instructions turned out to be more difficult than we assumed. The main problem was the integration of the neural network model in the ROS environment. The first issue regarding the integration was the approach we used for the classification of the timeseries. The use of a convolutional neural network required holding a buffer with detected landmarks.

It was problematic what to do after recognition - whether or not to empty the buffer, start collecting hand masks from scratch or continue to append data to the filled buffer. The first option didn't work because of the wrong classifications - most of the model's outputs were wrong even though the training went well (figure 5.2). Once the model returned a decision, the recorded movement in a buffer was lost, so the model didn't even have a chance to recognize the motion correctly. On the other hand, without emptying the buffer, the model was making decisions too often.

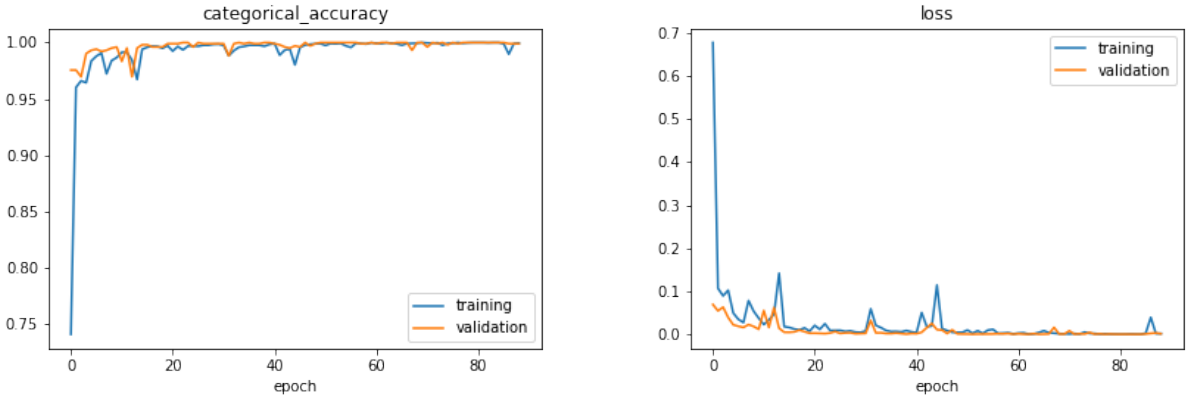


Figure 5.2: Loss and accuracy for neural network model classificating timeseries (motion).

Chapter 6.

Commissioning instructions

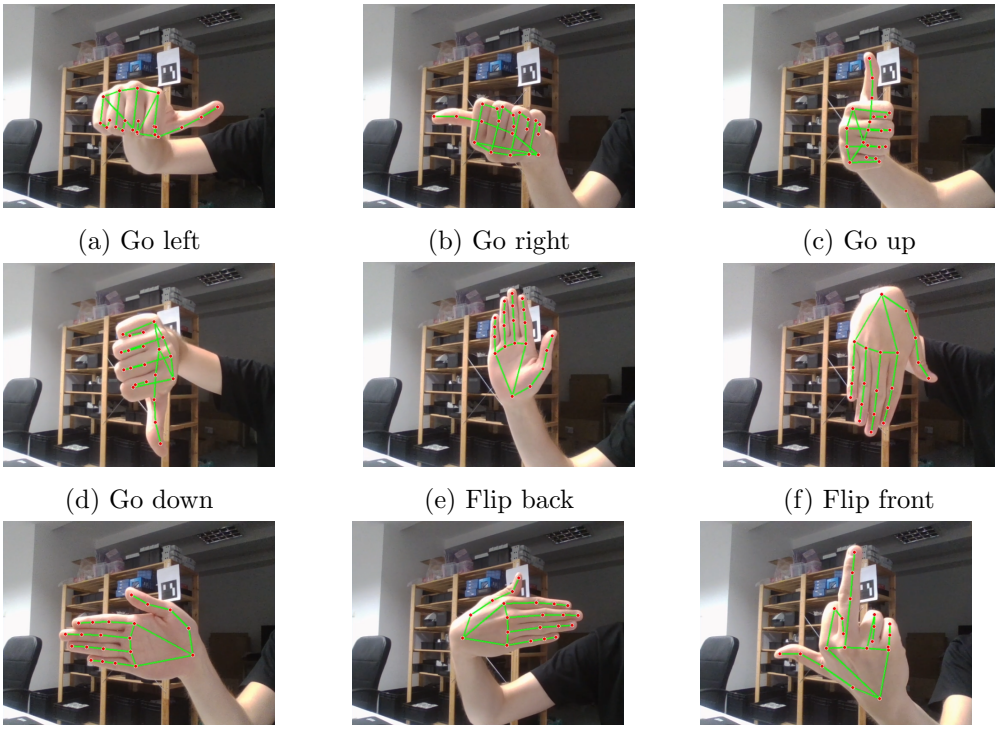
All code (including notebooks) from this project is available on [7].

6.1. Installation and running

For instructions on the installation and running of the project check out the README.md file on [7].

6.2. Gestures

Although the model recognizing the static gestures can distinguish ten different classes, only nine gestures are for operating the drone (the tenth class is for "no-gesture" recognition). The hand signs cover moving in vertical and horizontal axes, executing flips in four directions (left, right, front, back), and landing. All gestures are depicted (using the right hand), with descriptions in figure 6.1.



(a) Go left (b) Go right (c) Go up

(d) Go down (e) Flip back (f) Flip front

(g) Flip right (h) Flip left (i) Land

Figure 6.1: Gestures for the drone control.

Chapter 7.

Further work

Results and feedback show that the system is working. But the current version is very case specific and only presents that it's possible to develop a practical project for operating a robot (not necessarily a drone) with hand gestures.

Such a system would need to have a universal way to map hand signs to instructions, and that's the first thing we would change in the future. Currently, each gesture has simple instructions pre-assigned, and it's impossible to change them. The universal method would provide a set of recognizable gestures, and each person would have a way to customize instructions for each sign.
One solution is to make instruction methods universal enough, to specify the actions with the arguments that would be passed by as easily changeable parameters. This seems easy to complete but still limits the user to order only movement instructions to the robot.
To make it fully customizable - like allowing the robot to perform sequences or more complex instructions - then it needs the use of ROS services and requires users to write some code.

Another thing is the number and type of recognizable gestures. For some users, nine may be great, but in more complex systems it might be not enough. As the number of static gestures you can do with your hand is not infinite, we would try to find a way to make 2.3. work.
There is some place for improvement, like using RNN instead of a convolutional network, changing the current architecture of the model, generating more or better data, and whole integration with the ROS environment (the most challenging part).

One extra thing is adding recognition of gestures made by other body parts or even the whole torso (example in figure 7.1). Once the drone is away from a person, the detected hand becomes very small impeding the model to make a correct classification. In such a case, gestures made with a whole arm or body posture would be much easier to recognize.

Figure 7.1: Sample skeleton pose estimation. (Source: [8])

Having those changes applied, the project would serve more as a configurable interface for operating a robot with hand signs and motions instead of a case-specific solution for moving it, making it much more practical.

Bibliography

[1] https://www.dji.com/pl/rc.

[2] https://store.dji.com/pl/product/tello?vid=38421.

[3] Fan Zhang et al. *MediaPipe Hands: On-device Real-time Hand Tracking*. 2020. DOI: 10.48550/ARXIV.2006.10214. URL: https://arxiv.org/abs/2006.10214.

[4] https://google.github.io/mediapipe/solutions/hands.html.

[5] https://keras.io/examples/timeseries/timeseries_classification_from_scratch/.

[6] Clyde McQueen. https://github.com/clydemcqueen/tello_ros.

[7] https://github.com/Bitterisland6/tello_gesture_steering.

[8] https://learnopencv.com/deep-learning-based-human-pose-estimation-using-opencv-cpp-python/.