SLOVAK UNIVERSITY OF TECHNOLOGY
IN BRATISLAVA

FACULTY OF CHEMICAL AND FOOD TECHNOLOGY

Reg. No.: FCHPT-184466-115979

Touchless Drone Navigation via Hand Gestures

BACHELOR THESIS

2024 **Ivana Dukayová**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF CHEMICAL AND FOOD TECHNOLOGY

Reg. No.: FCHPT-184466-115979

Touchless Drone Navigation via Hand Gestures

BACHELOR THESIS

Study programme:	Process Control
Study field:	Cybernetics
Training workspace:	Department of Information Engineering and Process Control
Thesis supervisor:	doc. Ing. Martin Klaučo, PhD.
Consultant:	Ing. Patrik Valábek

2024 **Ivana Dukayová**

Slovak University of Technology in Bratislava
Department of Information Engineering and Process Control

Faculty of Chemical and Food Technology
2023/2024

STU
FCHPT

BACHELOR THESIS TOPIC

Author of thesis:	Ivana Dukayová
Study programme:	process control
Study field:	Cybernetics
Registration number:	FCHPT-184466-115979
Student's ID:	115979
Thesis supervisor:	doc. Ing. Martin Klaučo, PhD.
Head of department:	doc. Ing. Martin Klaučo, PhD.
Consultant:	Ing. Patrik Valábek
Workplace:	Department of Information Engineering and Process Control
Title of the thesis:	**Touchless Drone Navigation via Hand Gestures**
Language of thesis:	English
Topic specifications:	This thesis aims to propose and implement a system for classifying hand gestures for touchless drone control utilizing an embedded camera on the drone. The classification of such gestures relies on image processing from the drone camera and the application of machine learning models. The proposed solution must be sufficiently fast and robust to recognize changing gestures and execute the desired command accurately. Subtasks: 1. implementation of machine learning models for gesture recognition and classification 2. verification of classification accuracy and quality 3. easy implementation of the proposed solution to the drone
Length of thesis:	40
Deadline for submission of thesis:	05. 05. 2024
Approval of assignment of thesis:	**05. 03. 2024**
Assignment of thesis approved by:	**doc. Ing. Juraj Oravec, PhD.** study programme supervisor

Acknowledgment

Abstract

This project focuses on identifying gestures using a drone's camera and controlling the drone based on these gestures. Our target area is to create a reliable system for gesture recognition and drone control without the need for creating complex mathematical models.

Our work includes using the drone's camera to capture gestures and identifying these gestures using the MediaPipe library. We aim to recognize various gestures, including those that control drone movements such as ascents, descents, rotations, and taking photographs. For identification, we have implemented a neural network model that we trained on a dataset containing various hand gestures. In conjunction with the OpenCV library for image processing, the model is capable of recognizing and classifying gestures in real-time based on footage from the drone's camera.

Our work has enabled us to successfully identify various gestures and control the drone based on these recognized gestures. The project also demonstrates how gesture recognition can be a practical and interesting method for interacting with drones, allowing people to intuitively control these devices.

Abstrakt

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As robots continue to become more integrated into our daily lives, researchers are working to improve human-robot interaction. Gesture recognition technology has emerged as a key tool for enabling more natural interactions with robots, replacing traditional human-machine interfaces like keyboards, mice, and joysticks. This application can be a good intervention for people with disabilities. By using hand gestures, they can more easily interact with virtual environments, robotics, home automation, clinical operations, game controls, desktop/tablet applications, delivery services, sign language, and more. Drones are remotely controlled robots can be operated via a remote control device or smartphone. They're a popular tool in many of these applications, often used for sports event coverage, aerial photography, and emergency response. Developing automatic systems that can recognize hand gestures would greatly improve the ability to interact with drones intuitively.
Our goal is to create a real-time hand gesture recognition system for human-drone interaction. In this project, we have achieved touchless interaction between the drone and our hands. We utilized machine learning technology, specifically MediaPipe, for hand tracking and developed a gesture recognition model that underwent rigorous training and testing. The process involves five steps:

I. Drone capturing an image

II. Image processing via OpenCV

III. Gesture recognition

IV. Gesture - command conversion

V. Command execution

Gesture recognition is used in various sectors, including smart home automation and the medical field. It mainly focuses on human-machine interaction. The system receives

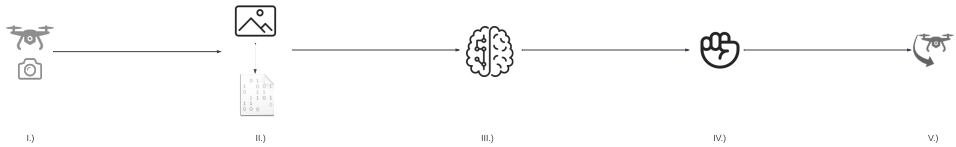

Figure 1.1: Steps of the process

input images from a drone camera connected to the device. The main goal of this project is to propose a real-time system with high accuracy. In summary, the project demonstrates how we can control a drone in an entertaining and educational way while highlighting the potential of gesture recognition technology in various industries and applications.

Chapter 2

Theory

2.1 Machine Learning

Machine learning, or ML for short, is a branch of artificial intelligence (AI) that focuses on creating computer algorithms that learn automatically from data and experience. It allows computers to learn from data and make predictions or decisions without the need for explicit programming. With the rapid advancement of AI, machines are now able to solve complex problems with greater ease. Learning methods can be broadly classified into two main categories - supervised learning and unsupervised learning. In addition to these, there are other methods such as reinforcement learning, semi-supervised learning, and many hybrid approaches.

Supervised learning involves training a computer algorithm with input data that has been labeled for a specific output. To enable it to produce accurate labeling results when presented with never-before-seen data, the model is trained until it can identify the underlying patterns and relationships between the input data and the output labels. Making sense of the data for a particular question is the goal of supervised learning. Classification and regression problems are well-suited for supervised learning.

With classification, one of a known number of categories is represented by the output variable. For instance, "dog" or "cat," "positive" or "negative." Popular techniques used for classification involve a logistic regression, a decision tree, a random forest, or a support vector machine.

The regression's value of the output variable is continuous or actual. For instance, "geographical location", and "price". The following algorithms are frequently employed: linear regression, nonlinear regression, a regression tree, or Bayesian logic.

Unsupervised learning relies on an unlabeled dataset for training, without requiring any correct output values as in supervised learning. Instead, the algorithm identifies

patterns and similarities within the data, independent of external measurements. This approach allows algorithms to explore and uncover unexpected insights that may not have been anticipated by humans.

Two more problem categories for the unsupervised learning algorithm are clustering and association.

With clustering, objects are grouped into clusters, so that those with the greatest similarities stay in that group and have little to no similarities with other objects in the group.

An association rule is used to determine the relationships between variables in a large database. An association rule improves the efficacy of marketing strategy. For example, consumers who purchase X (bread, for example) also frequently buy Y (butter).

Some of the popular unsupervised learning algorithms include K-means clustering, KNN, hierarchal clustering, or anomaly detection.

2.2 Techniques used in Computer Vision

In Computer Vision (CV), ML plays an important role in extracting important information from images. CV successfully contributes to various domains, surveillance systems, optical character recognition, robotics, suspect detection, and many more. The direction of CV research is going towards the healthcare domain, medical imaging (MI) is an emerging technology, that plays a vital role in improving image quality and recognizing critical features of binary medical images, covert original images into grayscale, and set the threshold for segmentation. Within computer vision, three key tasks stand out: segmentation, detection, and classification.

Image Classifiaction

It's a known fact that the image we see as a whole is made up of hundreds to thousands of tiny pixels. Before computer vision can determine and label the image as a whole, it needs to analyze the individual components of the image. That is why image classification techniques analyze a given image in the form of pixels and accomplish this by treating the picture as an array of matrices, the size of which is determined by the image resolution. The pixels of the digital image are taken and grouped into what we know as "classes." From this point on, the procedure will vary depending on the

algorithm. To guarantee that it is not left entirely on the final classifier, the selected algorithm will convert the image into a series of key attributes. These characteristics aid the classifier in identifying the subject matter and class to which the image belongs. We can say that the image classification pipeline looks like this: image pre-processing -> feature extraction -> object classification

Based on the nature of the problem, there are different types of image classification methodologies. They are binary, multiclass, multilabel, and hierarchical.

Binary classification divides unknown data points into two groups using an either-or logic for labeling images. Binary classification is used to handle many different yes/no problems, such as analyzing product quality to determine whether a product has faults, and many more tasks requiring judgment calls.

As the name implies, multiclass divides objects into three or more classes, whereas binary classification divides objects into two classes. It's highly helpful in a variety of fields, including medical diagnostics (disease categorization), NLP (sentiment analysis in situations involving several emotions), etc.

The multilabel method permits an object to be allocated to more than one label, in contrast to multiclass classification, which assigns an image to a single class. For instance, you could have to categorize multiple colors in an image. Taking a picture of a salad, a picture of one will feature red, orange, yellow, purple, and other colors. Consequently, several colors will be used as labels on a single image.

The process of classifying classes into a hierarchical structure based on their similarities is known as hierarchical classification. A lower-level class is more definite and detailed, while a higher-level class represents larger categories. If we're classifying a dog, our first model would recognize a dog vs another animal. If a dog is correctly predicted, another model will be used to classify the breed of the dog into border collie, golden retriever, and poodle. All features of higher-class attributes will be hierarchically contained in the latter ones. Hierarchy allows for effective information transfer between related classes as well as a flexible and interpretable framework for organizing and representing complex visual concepts.

Image Classification correlates one class from the training data with a whole image or video frame regardless of the amount of information present. It involves assigning labels and is suitable when fine-grained information is not necessary. During the model training stage, publically accessible datasets are frequently employed to enable correct data labeling.

Object Detection

The classification is advanced in object detection. It not only classifies many objects in an image but also provides annotations for the bounding boxes that correspond to each entity's position. The CV model's extra benefits enable its implementation in several practical contexts as these models are only a small portion of detecting, labeling, and annotating objects in images or videos. Faster R-CNN [10], YOLO (You Only Look Once) [27], and SSD (Single Shot MultiBox Detector)[17] are a few popular object detection algorithms. These algorithms are tailored to specific application requirements and differ in terms of speed, accuracy, and trade-offs. An example of an object detection model is a model for human detection. The model would return a bounding box around each person with a label

Cat Cat, Dog

Figure 2.1: Comparison of classification (left) with output: 'Cat' and object detection (right) with output: 'Cat, Dog' and their localization with bounding boxes

Image Segmentation

Since they both serve the same purpose, object detection and image segmentation are comparable. Both techniques identify items in pictures and provide coordinates for locating objects. However, segmentation algorithms produce accurate masks that cover objects at the pixel level, as opposed to creating whole boxes around them.

Annotations for image segmentation include the exact pixel locations of any instances that are present in the image. Image segmentation is better suitable for practical uses because of its accurate results. However, compared to object detection, picture segmentation models are computationally expensive due to algorithm complexity. The example used is in applying visual effects like background blurring and makeup effects to the image. The functionality identifies specific textures, colors, and segments of

object features within image data. It is also widely used in medical imaging, self-driving cars, or satellite imaging.

In figure **??** a U-Net [28], an architecture introduced by Olaf Ronneberger, Philipp Fischer, and Thomas Brox in 2015 is used. It's a convolutional neural network that was specifically designed to be used in Biomedical Imaging. A pre-trained model MobileNetV2 [30] is used as a lightweight, efficient feature extractor for the input image. It would analyze the photo of the dog and create a set of feature maps highlighting important visual attributes like edges, textures, and shapes relevant to the segmentation task. Then, with the feature maps provided by MobileNetV2, U-Net would take on the task of image segmentation. Its architecture, designed to work with fewer data points and to be precise in delineating object boundaries, would use the feature maps to generate a predicted mask. It would attempt to closely replicate the true mask by classifying each pixel as belonging to the dog or the background. A Pix2Pix [13] model is then used to generate a segmented image that tries to match the true mask, learning the mapping from the input image to the segmentation mask during training.

For this task, it is also possible to use interactive segmentation, which divides the image into two regions: the selected object and everything else. It receives a location in an image, calculates the object's boundaries at that location, and returns image data that defines the object's area.

Figure 2.2: Landmarks for pose estimation model by MediaPipe and U-Net segmentation on an image from Oxford-IIIT Pet Dataset (Parkhi et al, 2012).

2.3 Neural Networks

An artificial intelligence technique called a neural network trains computers to process information like that of the human brain. It uses connected nodes or neurons arranged in a layered pattern to mimic the organization of the human brain. Computers can utilize this adaptive approach to learn from their errors and keep getting better. As a result, artificial neural networks try to more accurately answer challenging problems, such as document summarization and face recognition.
Neural networks consist of neurons in the input layer connected to neurons in the output layer of the network. In Figure 3.7, where a general neural network schemes, we can see that they're joined via intermediate neurons. They're placed in hidden layers. 'Hidden' because, in the use of neural networks, only the input and output are of concern to the user.

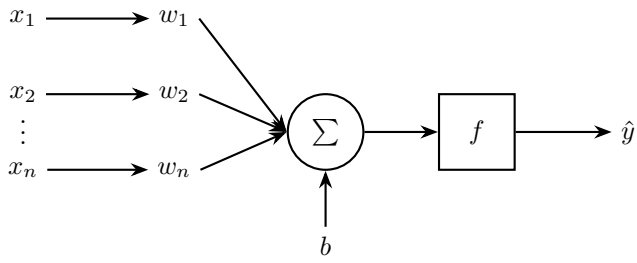

Figure 2.3: Schematic representation of a neuron with inputs, weights, a bias, an activation function, and an output.

Neurons are basic building components of neural networks. Each neuron takes inputs $(x_1, x_2, ..., x_n)$ which are evaluated by weights $(w_1, w_2, ..., w_n)$, which are factors that are changed during training. The weight of the connection affects how much input is passed between neurons. We use the weighted sum of all the inputs, adjusted for the weights of the connections between the inputs and the neuron, to determine the output $\hat{y}$ of the neuron. After summing all inputs multiplied by their weight, a bias b is added. Bias is a constant that shifts the activation function left or right (on x-axis). It tells us how high the weighted sum needs to be before the neuron starts getting meaningfully active. A value that goes into the activation function is then the summation function ϵ,

$$\epsilon = \sum_{i=1}^{n} x_i w_i + b \tag{2.1}$$

where n is is number of inputs.

Activation Functions

A neural network's activation function is an essential component. A neural network that lacks an activation function is just a straightforward linear regression model. This indicates that the neural network's non-linearity is provided by the activation function. When computing a weighted some like in 2.1, we can come out with any number. But for most networks, we want its activations (outputs) to be a value between 0 and 1. There are many types of activation functions, eg. threshold, sigmoid, rectifier, hyperbolic tangent, and softmax. We'll discuss a few of them.

Sigmoid Function

The data science community is familiar with the sigmoid function from its application in logistic regression. Any value can be entered into the sigmoid function, but it will always return a value between 0 and 1. In image classification tasks, the sigmoid function can be used to convert the linear model's output to probability. This probability can be used to predict the binary classification problem (whether there's a cat or a dog in the image). The sigmoid function has the following mathematical definition:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

SoftMax Function

Softmax function is very similar to a sigmoid function. In multiclass classification problems the sigmoid function isn't useful Applying a threshold for positive prediction of 0.5, would say the input data belongs to two classes. The sigmoid function also doesn't take into account the probability of belonging to other classes, when calculating probability for another class.

Softmax converts numbers or logits (outputs of neural networks) into probabilities. It works with relative probabilities.

Just like with the sigmoid function, in equation 2.3, an exponential ensures non-linearity; with z values from the neurons of the output layer for K classes. They are then divided by the sum of exponential values to normalize them before being converted to probabilities. The i-th entry can be thought of as the predicted probability of the test input belonging to class i.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K \tag{2.3}$$

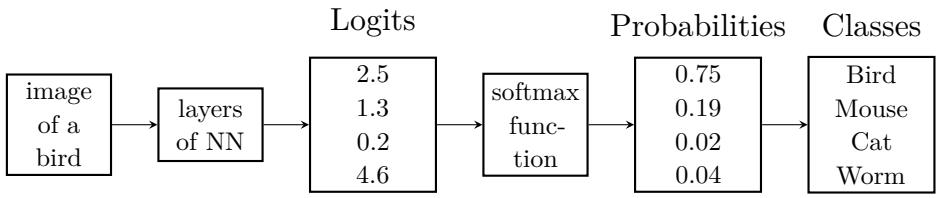

Figure 2.4: A diagram of the NN softmax classification process.

Rectifier Function

Unlike the sigmoid function, the rectifier function does not have the same smoothness property. It is still highly used in the deep learning community. The definition of the rectifier function outputs 0 if the input value is less than 0. The function outputs the input if it isn't. It helps to maintain mathematical stability and keep learned values from being stuck around 0 or jumping into infinity.

$$Relu(x) = max(0, x) \tag{2.4}$$

Rectified Linear Unit activation function, or ReLU for short, is a common term for rectifier function.
While the softmax function works with the output of the last layer of the neural network, the ReLu function is activated in hidden layers.

After applying application function f, the output of the neuron would then be:

$$\hat{y} = f(\epsilon) = f(\sum_{i=1}^{n} x_i w_i + b) \tag{2.5}$$

If we're discussing a neuron in the hidden or output layers, the equation can be rewritten as:

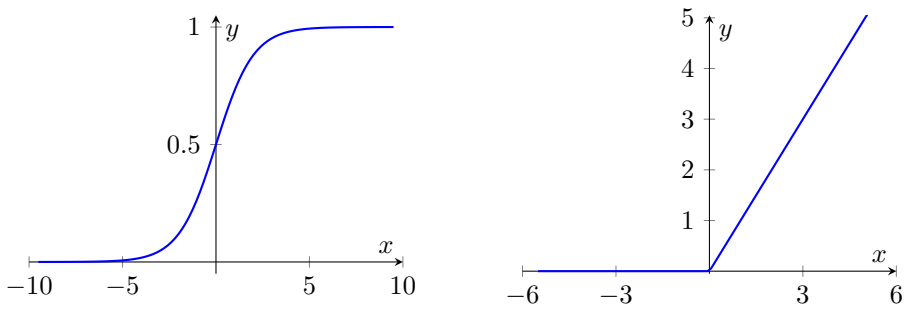

Figure 2.5: Softmax/Sigmoid function and Rectifier function.

$$a_i^k = f\left(\sum_{i=1}^{n} w_{ij}^k a_j^{k-1} + b_i^k\right) \tag{2.6}$$

This output is often called an activation a. It represents the sum of all neurons in the $(k-1)$ layer. The weight w^k is defined for each layer, k. The entries of the weights w^k are just the weights connecting to the k-th layer of neurons, that is, the entry in the i-th row and j-th column is w_{ij}^k.

2.4 Learning

During training, a function is passed from one neuron to another, either through a function 2.5 for hidden and output layers or through a weighted sum function 2.1 for the input layer. This is known as the forward pass. When the end of the network is reached, the feed-forward is also completed.
The used dataset is commonly split into a training set, on which the model trains, and a validation set, that evaluates the output of the model. Our particular model uses 75% as the training data.. As the true results are known during the training phase, the error can be calculated by comparing the actual and predicted values. Cost functions are used to obtain an error value.
There are many loss functions used, based on the nature of the problem. A few of them are: mean squared error, used for regression tasks, binary cross entropy, and categorical cross entropy.

Cost functions

As we train the model, we'll want to evaluate its accuracy using a cost (or loss) function. The cost function tells us about how well our algorithm models our dataset. The loss error is calculated for each training sample, while the loss function represents the whole set of m samples. The cost function represents the average loss error for a sample. It is a function that assesses how well an ML model performs with a given set of data. The error between expected and predicted values is quantified by the cost function and shown as a single real number. Cost functions can be formed in a variety of ways, depending on the nature of the problem.

Mean squared error

For a model using a formula 2.6, the commonly used loss function used for regression tasks, is the mean squared error (MSE). The cost for a single training example x may be written as

$$MSE = \frac{1}{2m} \sum_{i=1}^{m} (y_i - a_i)^2, \tag{2.7}$$

where i represents the index of the sample, a is the predicted outcome, y is the actual value, and m is the number of samples.

Crossentropy function

Other names for cross-entropy loss include logistic loss, log loss, and logarithmic loss. Every predicted class probability is compared to the actual class probability, and a loss function is used to penalize the probability according to its deviation from the actual expected value. Because the penalty is logarithmic, large differences near 1 will result in a large score, and small differences approaching 0 will result in a small score. Cross-entropy loss in a perfect model is 0. What is meant by cross-entropy is:

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}), \tag{2.8}$$

where M is number of classes, y is the binary indicator if class of label c is the correct classification for observation o and p is the predicted probability.

When dealing with labels that are one-hot encoded, such as the 3-class classification problem, where the values are [1,0,0], [0,1,0], and [0,0,1], categorical cross-entropy is

employed. Labels in sparse categorical cross-entropy are encoded with integers, such as [1], [2], and [3] for a 3-class problem.

There are many other types of cost functions used in ML, e.g. Root Mean Square error (RMSE), Binary Cross Entropy Loss Function, Categorical cross-entropy, Hinge Loss, Kullback-Liebler Divergence LOSS (KL-Divergence), and Huber Loss. Choosing a cost function will be influenced by the type of problem, the output activation function, and the network architecture.

With multi-class, single-label classification problems, as we'll deal with, the first choices of activation and loss functions are ReLU for hidden layers, softmax for the last layer, and categorical crossentropy as a loss function.

The training of a neural network begins with the random assignment of weights and biases (usually to 0) between the nodes. After the initial forward pass, the output layer is highly variable and lacks an identifiable pattern. To address this, a cost function is used to determine the error between the output and the desired values.

The goal is to enable the computer to find appropriate settings for all of the parameters - weights and biases. That also means minimizing the cost function.
In ML, minimizing the cost (loss) function is done by optimizers. Common optimizers include gradient descent, stochastic gradient descent, or Adam optimizer.

Gradient Descent is a fundamental optimization algorithm used in classification problems and linear regression. It calculates the first derivatives of the cost function to find the minimum. Each negative gradient component provides two pieces of information. The sign indicates whether the corresponding input vector component should be increased or decreased. Importantly, the relative magnitudes of all the components indicate which changes are more significant for the cost function output. However, Gradient Descent may sometimes be trapped in a local minimum and unable to determine the global minimum.

Stochastic gradient descent algorithms are a variant of the gradient descent method used in ML. In stochastic gradient descent, instead of computing the gradient using all the observations, only a small random sample of the training data is used to estimate the gradient. This method can significantly reduce the computation time, making it a useful approach in many machine learning applications.

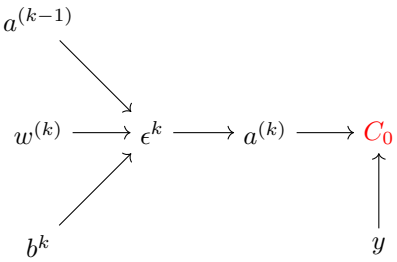

Figure 2.6: An example structure for backpropagation.

Adam is convenient for most convex optimization problems with large datasets. It combines adaptive methods and momentum methods, where it takes into account the moving average of the gradient's first and second-order moments. This allows it to effectively adapt the learning rates for each parameter. It is an extension of stochastic gradient descent.
Minimizing the cost functions means a better performance on all samples. The algorithm for computing the gradient efficiently is called the backpropagation.

Backpropagation

The backpropagation algorithm is the foundation of neural network learning. The goal of backpropagation is to understand how altering the weights and biases in a network alters the cost function.
In backpropagation, the partial derivatives $\partial C/\partial w_{ij}^k$ and $\partial C/\partial b_i^k$ are computed, calling C the cost.
Let's say we work with a cost function (MSE) for one training example in a simple network, such as in Figure 2.6: $C_0(w_1, w_2, \ldots, w_n) = (a^k - y)^2$.

We want to find the effect of the weight change on the cost function. In ML, the chain rule is often considered in the context of networks. The last activation a^k is determined by the weighted sum ϵ^k, or a weight w^k multiplied by the previous neuron's activation $a^{(k-1)}$ plus some bias b^k. After an activation function, $a^k = f(\epsilon^k)$. These parameters, along with a constant y (wanted output), let us compute the cost.
If the question is how sensitive the cost function is to small changes in our weight w^k, we simply calculate the derivative of C_0 with respect to w^k, using the chain rule. The equation is as follows:

$$\frac{\partial C_0}{\partial w^k} = \frac{\partial \epsilon^k}{\partial w^k} \frac{\partial a^k}{\partial \epsilon^k} \frac{\partial C_0}{\partial a^k} \tag{2.9}$$

With the computed derivatives:

$$\frac{\partial C_0}{\partial w^k} = a^{(k-1)} f'(\epsilon^k) 2(a^k - y) \tag{2.10}$$

It's worth noting that the last derivative multiplies the difference between the network's output and the thing we want it to be, so if that output is very different, even slight changes stand to have a big impact on the final cost.
This was computed only for a single training example. Since the full cost function involves averaging together all the cost across many different training examples, its derivative requires averaging the expression over all n training examples:

$$\frac{\partial C}{\partial w^k} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial C_i}{\partial w^k}$$

Of course, that is just one component w^k (in layer k) of the gradient vector ∇C, which itself is built up from partial derivatives of the cost function with respect to all the weights and biases. The sensitivity to bias is almost identical:

$$\frac{\partial C_0}{\partial b^k} = \frac{\partial \epsilon^k}{\partial b^k} \frac{\partial a^k}{\partial \epsilon^k} \frac{\partial C_0}{\partial a^k} = 1 \cdot f'(\epsilon^k) \cdot 2(a^k - y)$$

We want to see how sensitive this cost function is to the activation of the previous layer. The derivative of the weighted sum ϵ with respect to the activation in the previous layer, comes out to be the the weight.

$$\frac{\partial C_0}{\partial a^{(k-1)}} = \frac{\partial \epsilon^k}{\partial a^{(k-1)}} \frac{\partial a^k}{\partial z^k} \frac{\partial C_0}{\partial a^k} = w^k f'(\epsilon^k) 2(a^k - y)$$

When $\partial C / \partial \epsilon_i^k$ is close to 0, we can say that the neuron is already near optimal. Let's call C a cost function for an average weight or a bias, and $-\nabla C$ a negative gradient of the cost function.

$$-\nabla C(w_1, w_2, \ldots, w_n) = \begin{bmatrix} -0.08 \\ +0.12 \\ -1.06 \\ \vdots \\ +0.04 \end{bmatrix} \tag{2.11}$$

In equation 2.11, the negative gradients for each weight (their average), are already calculated. So for the cost function to get closer to a minimum, the weights need to

be changed accordingly. The cost function is sensitive to the weight w_1 of 0.08. After this is computed the weight just needs to be updated in the way:

$$w_1 = w_1 - \alpha \frac{\partial C}{\partial w_1},$$

Where α is a learning rate, which determines the gradient's influence, and $\frac{\partial C}{\partial w_1}$ is the partial derivative of the cost function C with respect to w_1.
In summary, the training can be divided into a few steps:

1. Forward Pass til the final predictions are made.
2. Loss Calculation
3. Backward Pass (Backpropagation)
4. Weight Update

This entire process is repeated for a specified number of epochs or until another stopping criterion is met (like the early stopping callback, which halts training if the validation loss doesn't improve for a set number of epochs).

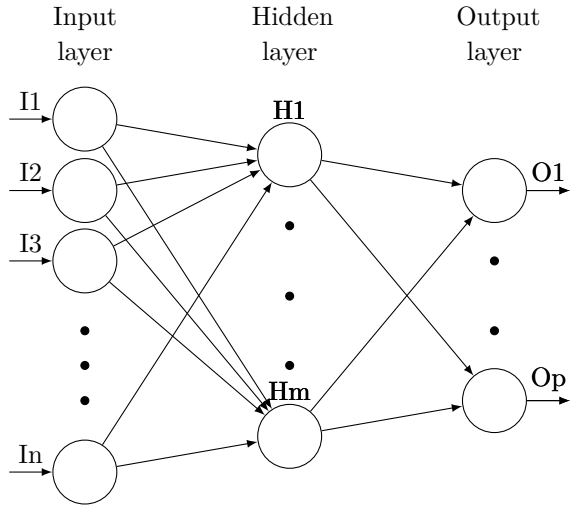

Figure 2.7: Structure of neural network

Hyperparameters of Neural Networks

- Learning Rate is a crucial parameter that controls to which extent the model needs to be modified. It determines the step size at each training iteration. It is usually a value between 0 and 1. We can determine the direction of a loss function's optimum by computing the loss function's gradient. The step size in that direction is determined by the learning rate parameter.

- Batch size tells us how much data is changed during 1 cycle (one epoch) of training. The data is subdivided into batches and each step is computed with respect to a batch. Larger batch sizes could lead to faster training but with a trade-off for lower accuracy or overfitting [34]. The best size will depend on several variables, such as the size of the training dataset, the complexity of the model, and the available computational resources.

- Epochs define the number of times that the learning algorithm will work through the entire training dataset. Every sample in the training dataset has had a chance to update the internal model parameters after one epoch. One or more batches make up an epoch.

Regulations of Neural Network

When talking about the regulation of neural networks we can mention techniques like dropout or early stopping.
Dropout is a regularization strategy for neural networks that, with a given probability, drops a unit (along with connections) during training. It randomly sets a fraction of inputs to 0 at each update during training. The goal is to stop co-adaptation, which occurs when a neural network becomes overly dependent on a single connection and may be an indication of overfitting. It makes intuitive sense to think of dropout as the formation of an implicit neural network ensemble. The result of dropout is shown in Figure 2.8
An intuitive technique for training just enough is early stopping. It prevents the model from learning on a 'noised' dataset. When training, after every epoch, the model is evaluated using a validation dataset. The training process is terminated if the model's performance on the validation dataset begins to drop (for example, if loss starts to rise or accuracy starts to fall).

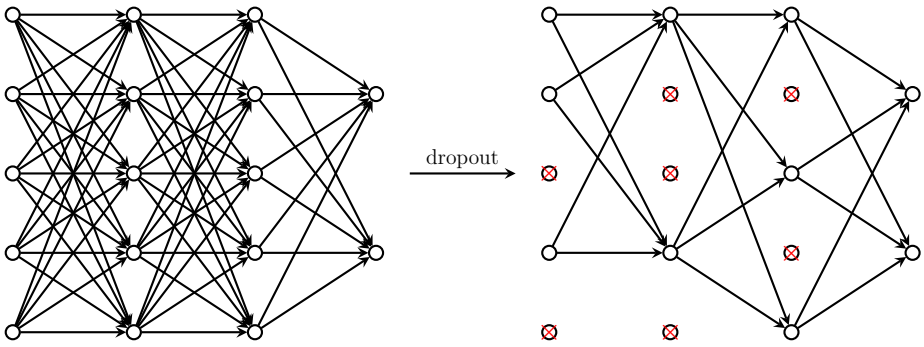

Figure 2.8: Visualization of dropout.

2.5 Gesture Recognition

Gesture recognition is a branch of computer science and language technology that aims to mathematically comprehend human gestures. A gesture can originate from any physical movement or emotion, but it usually appears on the hands or face. [32] Gesture recognition is essential to the development of a desirable alternative to popular methods of human-computer interaction.

Besides computer vision-based gesture recognition techniques, the non-vision-based recognition requires other devices (such as gloves) to gather data. [5] A data glove is a particular type of wired glove that is worn by a person and has tactile switches or sensors that attach to the glove's fingers or joints. To determine whether a hand is open or closed and whether certain finger joints are straight or twisted, optical goniometers, tactile switches, and resistance sensors measure the twisting of joints with basic measurements. Results are sent to a computer, which maps them to gestures and interprets them. One advantage of a basic device is that no kind of pre-processing is required. The technology was used since 1990s with very limited processing power and showed promise regardless of the limitation of manoeuvrability because of cables that were used to connect the data glove to the computer. [25]
In contrast to earlier gloves, some gloves made with wireless technology can now be worn comfortably thanks to technological advancements. Over the years, gloves have been divided into two categories: active data gloves and passive data gloves. An active data glove is made up of numerous sensors that track joint movements, accelerate objects, and communicate with the host computer using either wired or wireless technology. It is known that these gloves limit the wearer's capacity for innovation. In contrast, a passive data glove just has colors or pointers that help cameras and other devices recognize your finger. There are no sensors built into the glove.

CyberGlove II

Because the hand's joints can flex in different ways, the CyberGlove was designed to carry data inputs. The glove has eighteen sensors, as depicted in Figure 2.9. These include four capture sensors, two twist sensors on each finger, and sensors to measure the thumb limit, palm arch, wrist flexion, and wrist capture. A different version of this device has 22 sensors total, including four capture sensors, three flexion sensors for each finger, a palm arch sensor, and sensors for measuring the flexion and capture of the wrist. One glove version allows the user to write, type, and grasp small objects with their fingertips open. CyberGlove motion capture system has been continuously used in numerous applications, including biomechanics, animation, virtual reality, and digital prototype evaluation.

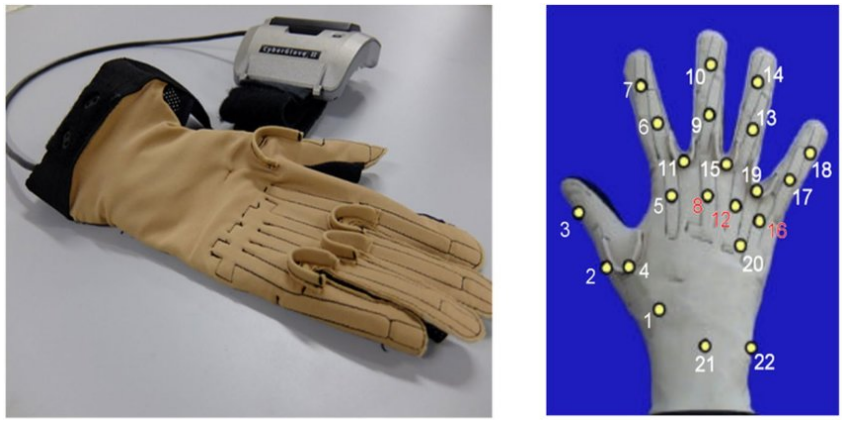

Figure 2.9: Cyberglove II

Chapter 3

Practical Part

In our task, we heavily worked with MediaPipe. We used it for data collection, face recognition, and finally gesture recognition.

3.1 Mediapipe

MediaPipe is a framework used to create machine learning pipelines for time-series data like video and audio. Google initially developed it to process real-time video and audio analysis on YouTube [6]. In 2019, the public release allowed researchers and developers to incorporate MediaPipe into their projects. Unlike other machine learning frameworks that require high computing power, MediaPipe can run efficiently on devices with low power, such as Android and IoT devices. It consists of the MediaPipe framework and MediaPipe solutions. The MediaPipe framework is developed using C++, Java, and Objective C programming. MediaPipe solutions include 16 pre-trained TensorFlow and TensorFlow Lite models built on top of the MediaPipe framework for specific use cases.

MediaPipe Hands

MediaPipe Hands is a solution that tracks hands in real time. The problem of detecting hands is somewhat intricate. The model must be able to recognize hands and it must function across a wide range of hand sizes with a big scale span so it must be robust. It is rather difficult to identify hands based just on their visual traits since hands lack high-contrast patterns.

MediaPipe Hands utilizes a combination of object detection, classification, and regression to recognize and track hands within a given image or video frame. The pipeline consists of two convolutional neural network models: a palm detector and a hand

landmark model.

The palm detector is a single-shot detector model that uses an orientated hand-bounding box to locate palms on a whole input image. This is done before landmark detection because it is easier to estimate bounding boxes of rigid objects like palms than detecting hands with articulated fingers. It ignores many aspect ratios and so the bounding boxes are only squared. The model uses both classification (hands, no hands), and object detection (predicting a bounding box around the detected hand), a focal loss [16] It was trained on data, containing 700 images of 14 geographical subregions, from both men and women.

The Hand Landmark Model runs subsequently with the palm detection model and uses regression to precisely localize 21 3D (x, y, z) coordinates within the hand region, similar to the data glove 2.5. Each landmark has a specific location on the hand that can vary within the pixel grid of an image. To capture the exact position, continuous values are used, allowing the model to indicate precisely where each landmark is located on the hand in terms of its x and y (and possibly z) coordinates within the image. The hands move and rotate freely, leading to a wide range of possible positions and orientations for each landmark. Continuous values are used because they make it possible to track hand gestures and movements more precisely. These gestures and movements may involve tiny changes to position and orientation that would be lost in discretized values. The model is resilient to self-occlusions and partially visible hands, and it learns a consistent internal hand posture representation. The results from the model contain 21 hand landmarks consisting of x, y, and z, a hand flag that indicates the likelihood that a hand is present in the input image, and a binary system of handedness, e.g. left or right hand. The coordinates x and y for each landmark are normalized 3.1 to [0, 1] by image width and height.

The model was trained both on real-world and synthetic datasets, noting that the wrist point was learned only from synthetic images. More than 30,000 manually annotated real-world images were used and the model is very robust, allowing it to detect and map hand landmark points accurately, even on partially visible hands in most cases. In Figure 3.1 my hands are not facing upfront but the model still finds all the landmarks for both hands. If tracking 2 hands, it also shows which one is left or right with different colors.

For encountering a tracking failure, There was also another model output created for tracking failures. It generates the likelihood that the given crop contains a hand that is reasonably aligned. The detector is triggered to reset tracking if the score falls below a predetermined threshold.

Figure 3.1: Hand landmarks model output

Data Normalization

The MediaPipe hand landmarks model provides coordinates for hand landmark points based on the position of pixels containing those points in an image. As a result, the coordinates of two images of the same hand sign with different placements in the frame can have significantly different distances between them. This makes it more challenging to train the model.

To solve this problem, the wrist's landmark point has been considered with coordinates [0,0], and the coordinates of all other landmark points were adjusted accordingly. First, the coordinates' values of the wrist's landmark point are subtracted from all coordinates' values.

Then, the coordinates were normalized to be between 0 and 1 by dividing them by the largest absolute value of the difference. Finally, the normalized coordinates were collected in the landmarks list. The coordinate normalization procedure is shown in Figure 3.2.

MediaPipe Face

With support for multiple faces and six landmarks (left eye, right eye, nose tip, mouth, left eye region, and right eye region), MediaPipe Face Detection is a quick solution for face detection. Mediapipe BlazeFace [4] is a compact and efficient face detector designed for mobile GPU inference, that serves as its foundation, detecting only a few landmarks on the face and drawing a bounding box. Because of its real-time performance, the detector can be used with any live viewfinder experience that needs

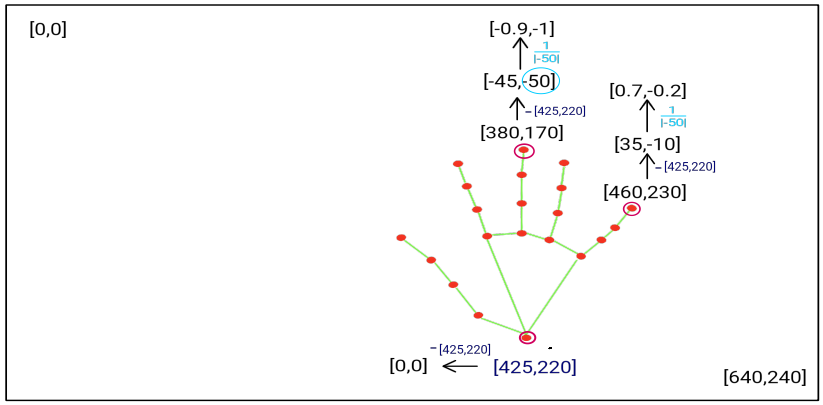

Figure 3.2: Process of normalization of landmark coordinates

a precise facial region of interest as an input for other task-specific models, like face region segmentation, facial feature or expression classification, and 3D facial geometry estimation (like MediaPipe Face Mesh). BlazeFace leverages a resolution technique as an alternative to a GPU-friendly anchor mechanism adapted from Single Shot MultiBox Detector (SSD) [18] and a lightweight feature extraction network that is similar to MobileNetV1/V2 [12]. With BlazeFace in use, we can get outputs shown in Figure 3.3, also outputting the confidence of the face.

Figure 3.3: Outputs of face detection model with different lightings

3.2 Data Collection

The flow of this project is as follows: the drone flies off, and we process the image that is captured by the drone's camera. Using multiple models for detecting, classifying, and localizing features in the image, after detecting a face, our model returns a prediction of a gesture and the drone executes the corresponding command in real time. The logic of the process is presented in Figure 3.4

To make predictions, we had to train our model with data. Since we decided to use hand gestures, we had to collect the data ourselves. After finishing the training, we established drone commands and tested our program.

Although many publicly accessible datasets contain pictures of hand gestures, we chose to collect our data using the MediaPipe hand-tracking model. To do this, we recorded video footage of various hand gestures being held for a specific duration and moved around to increase diversity. Because the MediaPipe models are open-sourced and many developers use them for various projects, there are also numerous programs available for capturing and labeling gestures, like [33]. During the recording process, we labeled each gesture by pressing a key. Our dataset includes over 5000 samples, including images of both right and left hands, palms facing forward or backward toward the camera, and various degrees of hand positions captured by the camera.

Our final dataset has 8 total classes. It demonstrates the efficacy of the model and minimizes the risk of error commands that could lead to unsafe situations. The gestures are shown in Figure 3.5.

The hand-tracking model used for data collection, outputs x, y, and z coordinates of hand landmark points from images, but only x and y coordinates are necessary for training the final model. Therefore, we eliminated the z coordinates. After normalization 3.1, we stored the remaining x and y coordinates of hand landmarks for various hand signs in a csv file for each hand landmark. The coordinates displayed in Figure 3.6 represent the data points used to generate the final dataset and train the final model. The first column denotes the gesture label, the next two columns indicate the coordinates of landmark 0 at the wrist, and so on. Our csv file contains many combinations of landmark coordinates for each gesture.

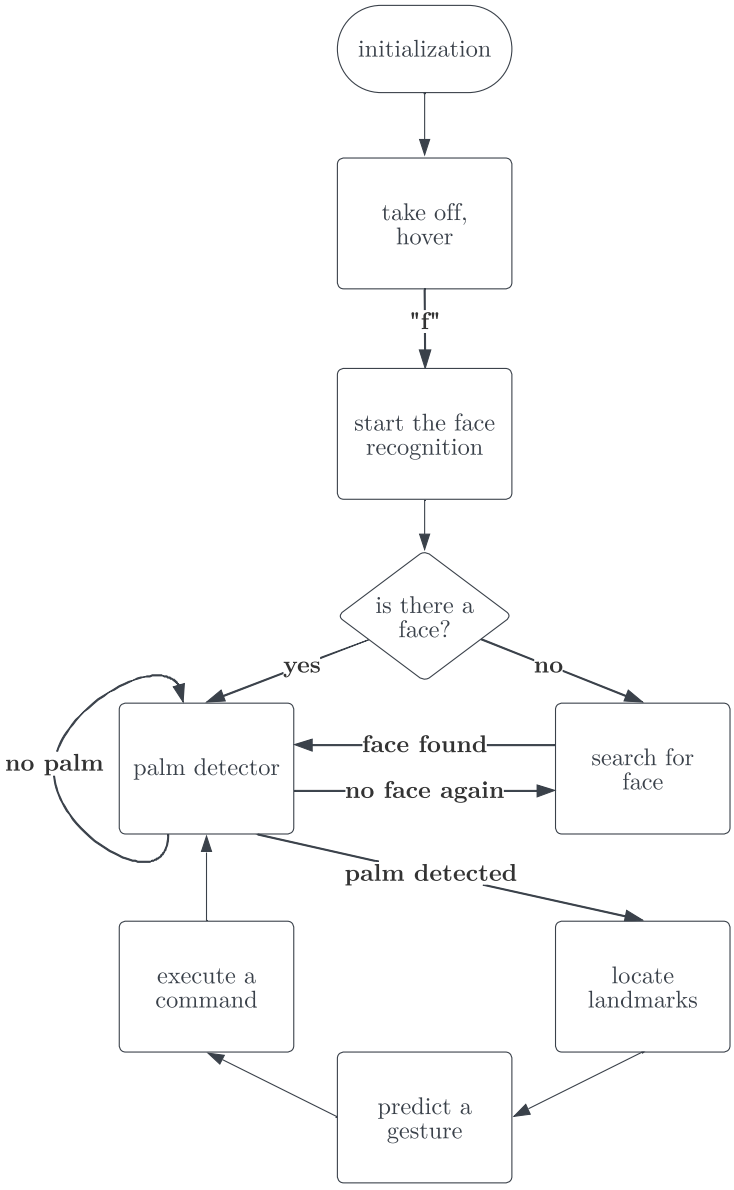

Figure 3.4: Logic in the recognition process.

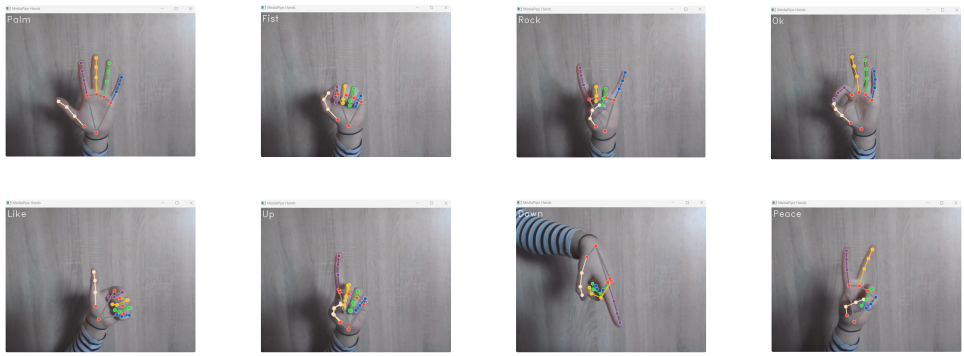

Figure 3.5: Gestures used in this project.

Gesture	lm0.x	lm0.y	lm1.x	lm1.y	...	lm20.x	lm20.y
0	0	0	0.19444	-0.07639	...	-0.19444	-0.77083
1	0	0	-0.42857	0.08929	...	0.12500	-0.44643
3	0	0	0.23077	-0.14530	...	0.28205	-0.85470
4	0	0	0.16667	-0.10526	...	-0.02632	-0.30702
5	0	0	0.21698	-0.26415	...	0.26415	-0.05660
6	0	0	-0.12791	-0.12209	...	0.13953	-0.16279
7	0	0	-0.22973	0.11486	...	0.04054	0.41216

Figure 3.6: A selection of the saved csv file

3.3 Model

The model we used for gesture recognition is a fully connected feedforward neural network. The model is adjusted to work with data output by KeyPointClassifier. KeyPointClassifier is a class that processes data of MediaPipe Hands models. The data are then sent to our gesture classifier.

The following is a report on training our neural network with the data. We utilized a model, which comprises one input layer and one output layer, along with five hidden layers consisting of three dense layers and two dropout layers. The sequential application of dropout and dense layers with ReLU activation acts as a mechanism to prevent overfitting while allowing the model to make nonlinear mappings. The output layer of the neural network comprises neurons equivalent to the number of hand gestures it can recognize - 8. The configuration of the model is depicted in Figure 3.7. The was trained with a dataset split into training and testing sets and was designed

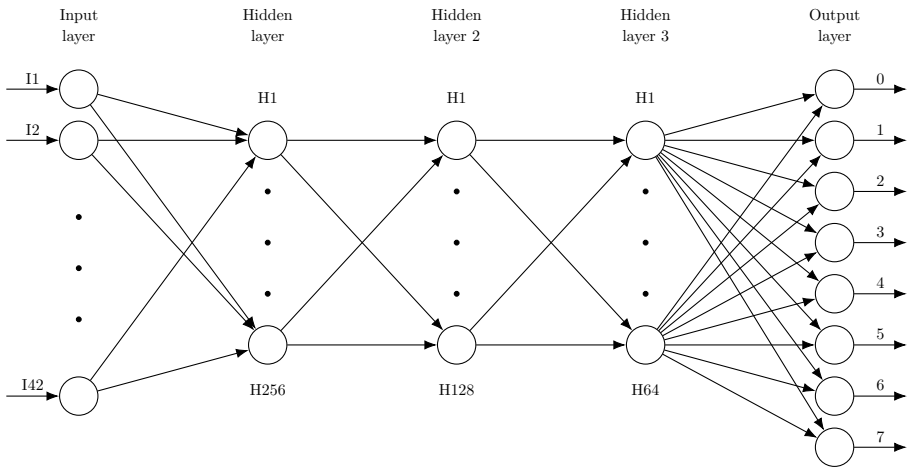

Figure 3.7: Neural network architecture.

using the Adam optimizer, which is efficient and appropriate for training models with large data and parameters. The Sparse Categorical Crossentropy loss function was utilized to evaluate the loss between predicted and actual labels, while accuracy was used as the evaluation metric to determine how frequently the prediction matches the actual label. The training was halted by early stopping when the validation loss stopped to decrease and then converted into TensorFlow Lite to reduce its size. Initially, the training model was set to have only four hidden layers (2 dense layers). However, the training accuracy failed to exceed 90%. Consequently, we updated the model to have five hidden layers (with 3 dense layers), without any other modifications and obtained satisfactory results. The training accuracy achieved was 99.62%, and the calculated loss was 0.0309.

3.4 Results

To perform a quantitative analysis of the test dataset, we employed the classification report and confusion matrix libraries from scikit-learn. The classification report library produced an assessment report of our model with accuracy, precision, recall, and F1 score matrices. Additionally, the support matrix represents the model's real-time recognition performance.

Classification Report

	precision	recall	f1-score	support
0	0.99	1.00	0.99	622
1	1.00	0.99	1.00	508
2	1.00	0.98	0.99	556
3	1.00	0.99	1.00	759
4	1.00	1.00	1.00	621
5	1.00	1.00	1.00	673
6	0.98	1.00	0.99	666
7	1.00	1.00	1.00	603
accuracy			0.99	5008
macro avg	1.00	0.99	0.99	5008
weighted avg	0.99	0.99	0.99	5008

Figure 3.8: Classification report of the model

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations.

The accuracy matrix calculates the number of correctly predicted labels by the model from the entire dataset (Eq.3.1). The precision matrix measures the model's accuracy out of the predicted positives. It calculates the number of actual positives in the predicted positives and is an excellent measure to consider when the False Positive (FP) cost is high. Equation 3.2 depicts the mathematical formula of the precision matrix.

The recall matrix measures the number of predictions our model correctly labels as positives. It is a measure considered when false negatives have high costs. The mathematical formulation of the recall matrix is Equation 3.3.

The F1 score is calculated by combining both precision and recall, as shown in Equation 3.4. It is their harmonic mean.

Support is the number of samples in each class. The macro-average of precision, recall and F1-score demonstrates the system's average performance across all classes, while the weighted average takes into account the class imbalance by weighting the metrics based on the number of samples in each class.
Additionally, the loss value of 3.09% (not shown) indicates how efficiently the model is minimizing its errors during training, while overall accuracy stands at 99.62%

The classification report of the implemented model in detail is shown in Figure 3.8.

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{3.2}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{3.3}$$

$$\text{F1 score} = \frac{2PR}{P + R} \tag{3.4}$$

In the equations (3.1), (3.2), (3.3), and (3.4), TP, TN, FP, and FN represent True Positive, True Negative, False Positive, and False Negative, respectively.

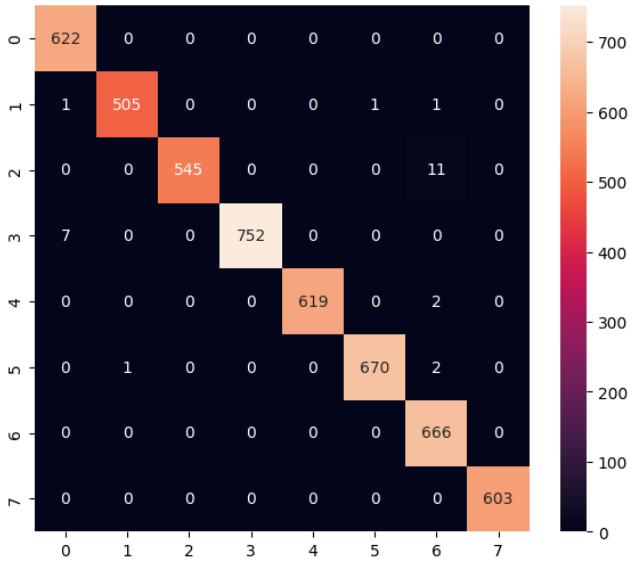

Figure 3.9: Confusion matrix of the model

The confusion matrix is a key performance metric used in machine learning, particularly in classification tasks. By utilizing the scikit-learn library in Python, one can create a confusion matrix. To conduct our experiment on predicting hand gestures, we obtained

and pre-processed the necessary datasets. The confusion matrix was then utilized to evaluate the accuracy of our model, which is illustrated in Figure 3.9. The x-axis represents predicted labels, while the y-axis represents actual labels.

Through the confusion matrix, we can pinpoint which gestures are being incorrectly recognized. For instance, we observed that gesture 2 was misinterpreted as gesture 6 a total of 11 times. We can further calculate the recall of gesture 2 utilizing Equation 3.5:

$$\text{recall}(2) = \frac{545}{545 + 11} \tag{3.5}$$

The system's performance is satisfactory, with almost perfect precision, recall, and F1-score in most classes, as well as high accuracy and weighted average.

To find the model's performance from a loss validation standpoint, we can plot its progress throughout the epochs.

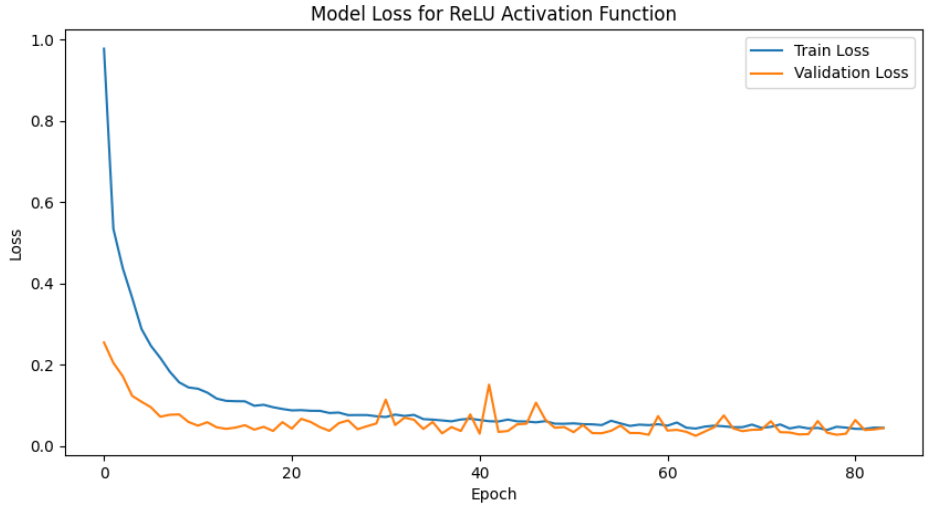

Figure 3.10: Training and validation loss curves for a neural network model utilizing the ReLU activation function.

The graph indicates a rapid decrease in loss in the initial epochs followed by a stable, low variance loss, demonstrating good model convergence without overfitting.

Comparisons

As we wanted to optimize our neural network, we conducted a comparison of the Stochastic Gradient Descent (SGD) and Adam optimizers. Figure 3.11 shows that Adam achieves faster convergence, reduces loss swiftly, and maintains high accuracy, making it suitable for complex neural network training where time and resource optimization are as critical as model accuracy. On the other hand, SGD shows more gradual improvement and requires more epochs to reach comparable performance.

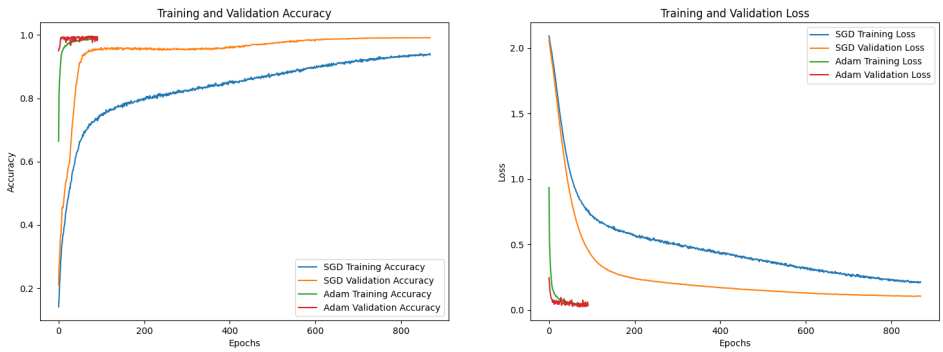

Figure 3.11: Comparative analysis of different optimizers.

We studied the impact of initial learning rates on Adam optimizer's performance. The initial learning rate has a significant effect on the convergence speed and training stability, as demonstrated in figure 3.12.
The model trained with a higher initial learning rate of 0.001 (as used in our case) converges rapidly, as indicated by a swift decline in loss and a quick rise in accuracy. However, the models with lower initial rates of 0.0001 and $1e^{-5}$ demonstrate a more gradual improvement, with the lowest rate showcasing the most stable but slowest progression.
The initial learning rate serves as a multiplier to the adaptively computed rates and determines the starting point for adjustments.

We researched different batch sizes, ranging from 16 to 1024, to find the right one for our model. We found that smaller batch sizes, such as 16 and 64, decrease the loss quickly, but they are less stable and can make the convergence process more complicated. On the other hand, larger batch sizes, such as 512 and 1024, are more stable but converge slower, which means they may not use the training data as efficiently. The model's performance on each batch size is shown in figure 3.13

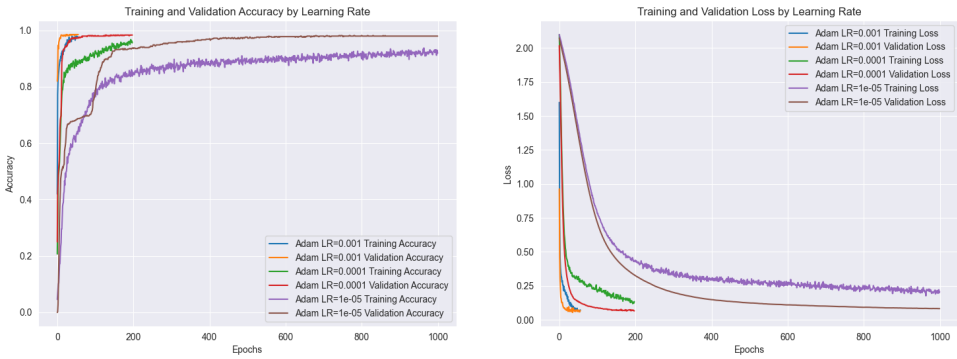

Figure 3.12: Training and validation accuracy over epochs for different initial learning rates using the Adam optimizer.

After considering all the options, we chose a batch size of 128. This size balances fast convergence with stable loss decrease and efficiently learns from the data. It is a good compromise between the noise observed in smaller batch sizes and the slow convergence observed in larger ones. The loss curve for a batch size of 128 is consistent, which means that our model has strong generalization performance. Therefore, we selected this batch size for our final model training.

Figure 3.14 shows the model's loss over 1000 epochs when using the ReLU activation function. The training loss decreases rapidly in the initial epochs, indicating the model is learning quickly and stabilizing. However, the validation loss fluctuates significantly throughout the training process, particularly in later epochs. This suggests that the model may be capturing noise or overfitting to the training data, continuing to learn beyond achieving minimal loss on the validation set.

The importance of implementing early stopping as a regularization technique (like in figure 3.10) is underscored by this extended training. Without it, the model risks wasting computational resources and reducing its ability to generalize. The increasing divergence between training and validation loss after several hundred epochs reinforces the need for early stopping to halt training when the model ceases to improve in generalization to validation data.

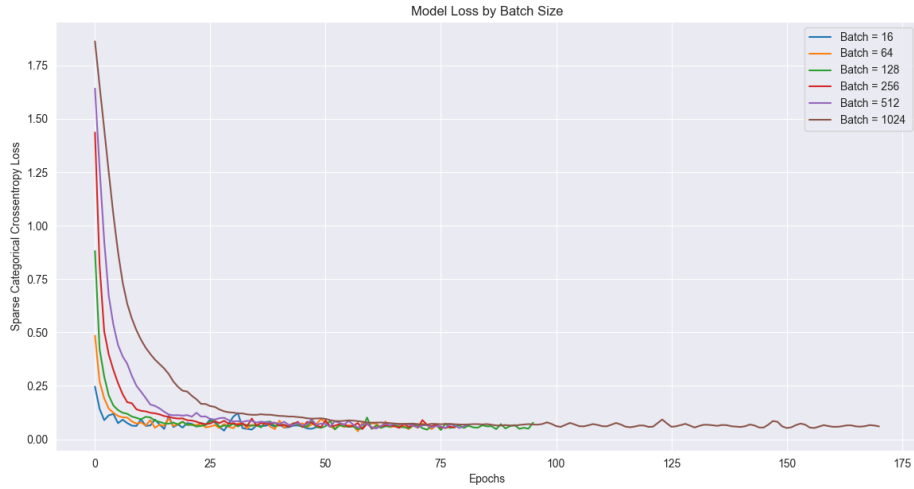

Figure 3.13: The effect of batch size on model loss during training and validation.

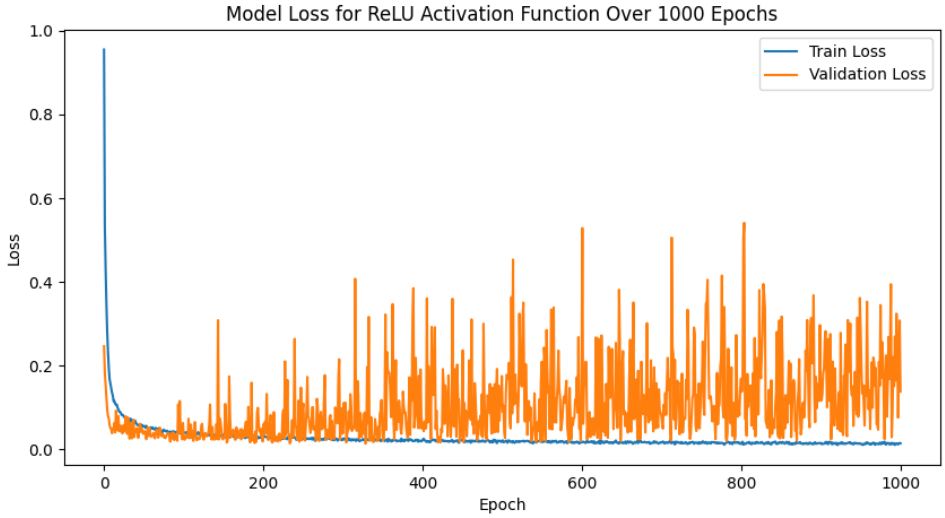

Figure 3.14: Model loss using the ReLU activation function plotted over 1000 epochs without early stopping.

3.5 Drone Implementation

The Tello drone is a compact quadcopter that boasts an advanced vision positioning system and a high-resolution (8 mp) onboard camera, designed to capture stunning aerial photographs and videos. It offers versatile operating options, allowing users to control it through a laptop computer or a smartphone. The drone connects to devices using Wi-Fi, which facilitates a stable and fast connection essential for real-time video streaming and responsive flight control. Additionally, it supports the 2.4 GHz frequency band for remote control, a widely used standard that ensures reliable communication over distances. The drone has other built-in functions like a range finder, barometer, LED, or vision system. It is fitted with a single accelerometer and 3-axis magnetometer (which can get the drone orientation), a pressure, and an IR-based altitude detector. The Tello drone is compatible with a dedicated smartphone application available on both Android and iOS platforms. This app unlocks several intuitive flight modes, easy-to-use controls, and creative functions.

The Tello library is a toolkit crafted to seamlessly integrate with Python. It's specifically designed for developers and hobbyists, offering easy access and control over the Tello drone's functionalities. With a comprehensive array of built-in functions, the library manages communication protocols with the drone, allowing for real-time transmission of control commands and reception of status updates. This ensures a smooth and responsive piloting experience.

Moreover, the Tello library efficiently handles state changes, adapting to various flight conditions and drone responses, such as battery levels, speed, and altitude. For example, if the drone is low on battery, it doesn't flip. This feature is essential for creating applications that require precise control and feedback from the drone, providing a foundation for developing flight patterns and maneuvers.

The library's event-driven control allows the execution of code in response to specific events, such as takeoff, landing, or moving in a direction with a designated speed.

To operate a drone, we can use specific hand gestures as commands. This technique is chosen due to its ease of use and intuitive nature. Additionally, Tello provides a Python API that streamlines the process of drone operation. This API eliminates the need for direct manipulation of the motor hardware, allowing us to focus on perfecting our hand gestures and achieving more precise control of the drone. For this to work, we just set each gesture ID to a command.

Table 3.1 lists the commands used in our project, which were assigned to the labels of

Figure 3.15: Drone used in this project.

our gestures. The gestures we utilized can be found in Figure 3.5.

Gesture Label	Command
0 - Palm	Move along the y-axis (forward velocity)
1 - Fist	Move along -y-axis (backward velocity)
2 - Rock	Flip forward (upward velocity)
3 - OK	Take-off or land (depending on whether in flight or landed)
4 - Peace	Takes a picture (saves in png)
5 - Like	Rotate 360°
6 - Up	Move up (ascending velocity)
7 - Down	Move down (descending velocity)

Table 3.1: Gesture commands for drone control

Our program prompts the Tello drone to begin executing commands after a key press and identifies a gesture every 5 seconds. This allows for the comfortable execution of a command before recognizing the next one. After tweaking speeds and sleep times, the Tello drone executed the received commands rapidly and consistently.

In Figure 3.16 we can observe the various gestures identified by the drone's camera. Upon recognizing the 'Palm' gesture, the drone would proceed to move backward. In the subsequent image, we see the drone on the ground, and upon recognizing the 'Ok' gesture, it would take off to a predetermined altitude. In addition, if we are not recognizing any gesture, the drone is ready to land on the palm. Because the drone can execute most of the commands only while flying, it takes off at the start of the program. It can also take off with gesture 4, so the program doesn't have to be reset.

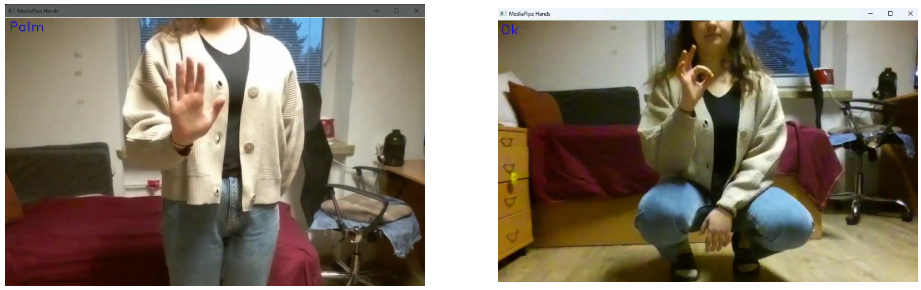

Figure 3.16: Gesture recognition shots from the drone camera.

3.6 Future Work

Moving forward with our work, there is ample room for the expansion of the project. We have developed a good gesture classifier that can be utilized for other applications besides drones, such as smart home control, sign language translation, multimedia control, or security systems. For example, a previous study [7] used a gesture recognizer and ESP8266 microcontroller to control home appliances using gestures to adjust a simple LED light, turning it on/off, setting its brightness and color.

If we decide to implement face recognition, there are many popular face recognition systems available that we can use. Facial recognition systems are widely used in modern life, making authentication processes easier without the need for passwords or physical IDs. Some examples of algorithms used for face recognition include Eigenfaces [37], Local Binary Patterns Histograms (LBPH) [2], Fisherfaces, Scale-Invariant Feature Transform (SIFT) [14], and Speed Up Robust Features (SURF) [3]. The Haar Cascade classifier by Paul Viola and Michael Jones [2] is the first choice for face detection alone.

Since we are using Mediapipe, we opted to use their Face Landmarker model to extract 478 3-dimensional face landmarks. This model works in conjunction with the BlazeFace detection model and the Face mesh model. Figure 3.17 shows the difference in outputs of each model. We applied these models in a similar way as we did with the hand models, using spatial positions of the landmarks. Firstly, we collected our data, labeling each capture of the face with a label. Then, we trained a simple KNN model to predict the person in the picture. However, since our database is not large (as we only tested with two people, and the prediction is real-time), it is difficult to conclude whether the model is sufficient for higher usage. Nevertheless, it demonstrates an alternative way for face recognition using only Mediapipe models. The face recognition model is saved in our repository [11].

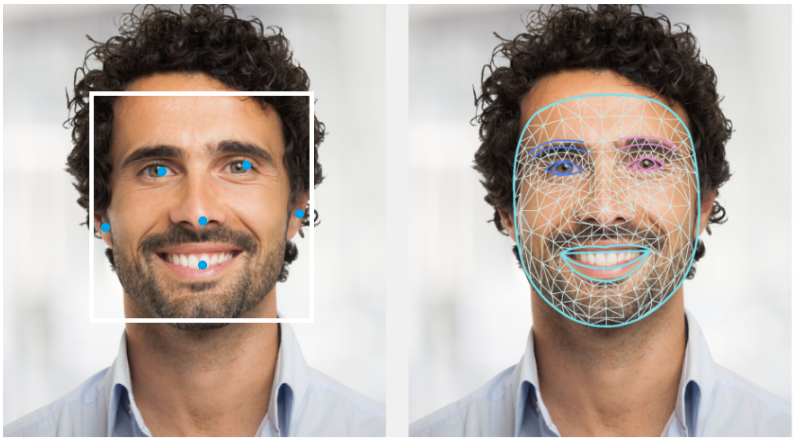

Figure 3.17: Outputs of FaceBlaze and Face Landmarker model.

Chapter 4

Conclusions

This thesis aimed to design and develop a reliable system that classifies hand gestures for controlling a drone through an embedded camera system. This was achieved by integrating machine learning models for real-time image processing. MediaPipe, an open-source framework, was utilized for hand tracking, while TensorFlow, an open-source machine learning framework, was used for developing the model. To classify gestures efficiently, a neural network was designed based on deep learning research. The model was designed to handle diverse hand orientations and lighting conditions encountered during drone operation. The backbone of this achievement was the fact, that the model wasn't trained on just images processed with classical methods, but rather extracted features represented 21 landmarks on each hand.

To evaluate the efficiency of the prediction model, it was empirically tested, and an accuracy of 99.62% was achieved on the testing set. The model's ability to generalize well to new, unseen data was a critical requirement for real-world applications, and this was achieved through a classification report that yielded high precision, recall, and F1 scores across all classes. The model's architecture is based on deep learning research and employs dropout layers and dense networks with ReLU activations.

The model successfully demonstrated its ability to accurately distinguish between nuanced hand gestures, providing reliable navigation commands to the drone. The minimal number of misclassifications is a testament to the model's precision in differentiating between the hand gestures. In this study, it was demonstrated that the implementation of the model using TensorFlow Lite resulted in an efficient translation of computational power and performance in the context of an embedded system, like the Tello drone. This successful implementation highlights the practicality of incorporating advanced machine-learning algorithms in environments with limited resources.

In conclusion, the thesis aims to design, develop, and implement a machine learning system that provides accurate, efficient, and reliable gesture-based control of a drone.

Through extensive testing, the practical applications of this system were validated, and its adaptability across a range of gestures was proved. This achievement lays the groundwork for future advancements in touchless drone control interfaces, or work with authentication, and opens new doors for exploration in the field of human-computer interaction.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Timo Ahonen, Abdenour Hadid, and Matti Pietikäinen. Face recognition with local binary patterns. volume 3021, pages 469–481, 05 2004.

[3] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran, and Matthias Grundmann. Blazeface: Sub-millisecond neural face detection on mobile gpus, 2019.

[5] M. K. Bhuyan, M. K. Bhuyan, and A. Gogoi. A review on hand gesture recognition techniques, challenges, and applications. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 10(2):175–190, 2017.

[6] Google AI Blog. Mediapipe: On-device, real time hand tracking, 2019. Blog Post Accessed: December, 2023.

[7] Sheng-Kuan Wang Victor R.L. Shen Yi-Chih Tung Frank H.C. Shen Cheng-Ying Yang, Yi-Nan Lin and Chun-Hsiang Huang. Smart control of home appliances using hand gesture recognition in an iot-enabled system. *Applied Artificial Intelligence*, 37(1):2176607, 2023.

[8] Ahmed Eid and Friedhelm Schwenker. Visual static hand gesture recognition using convolutional neural network. *Algorithms*, 16(8):361, 2023. Journal Article DOI: 10.3390/a16080361.

[9] D. F. Escoté. djitellopy, 2021. GitHub Repository.

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[11] Hedviga. Touchless drone control, 2024. Project Repository on GitHub Accessed: December 2023.

[12] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[13] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[14] Ladislav Lenc and Pavel Král. Automatic face recognition system based on the sift features. *Computers & Electrical Engineering*, 46, 02 2015.

[15] Y Li and P Zhang. Static hand gesture recognition based on hierarchical decision and classification of finger features. *Science Progress*, 105(1), 2022.

[16] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.

[17] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. *SSD: Single Shot MultiBox Detector*, page 21–37. Springer International Publishing, 2016.

[19] Carmen López-Casado, Enrique Bauzano, Irene Rivas-Blanco, Carlos J Pérez-del Pulgar, and Víctor F Muñoz. A gesture recognition algorithm for hand-assisted laparoscopic surgery. *Sensors*, 19(23):5182, 2019.

[20] MediaPipe. Mediapipehands, 2023. MediaPipeHands Source Code Accessed: November 2023.

[21] Joe Minichino. Opencv mediapipe hand gesture recognition, 2022. Project Repository on GitHub Accessed: December 2023.

[22] Michael A. Nielsen. Neural networks and deep learning, 2018.

[23] JungSoo Park, Hyo-Rim Choi, JunYoung Kim, and TaeYong Kim. Hand pose recognition by using masked zernike moments. In *2014 International Conference on Computer Vision Theory and Applications (VISAPP)*, volume 1, pages 551–556. IEEE, 2014.

[24] P. Premaratne. *Chapter 2: Human Computer Interaction Using Hand Gestures*, chapter 2. Springer, Singapore, 2014 edition, 2014.

[25] P. Premaratne. *Human Computer Interaction Using Hand Gestures*. Springer, Singapore, 2014 edition, 2014.

[26] S Padam Priyal and Prabin Kumar Bora. A robust static hand gesture recognition system using geometry based normalizations and krawtchouk moments. *Pattern Recognition*, 46(8):2202–2219, 2013.

[27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[28] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[29] RYZE. Tello specs, 2023. TELLO Specifications Accessed: December 2023.

[30] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[31] P. K. Sharma and S. Sharma. Evolution of hand gesture recognition: A review. *International Journal of Engineering and Computer Science*, 4(1):9963, Jan 2015.

[32] J. D. Smith and A. B. Johnson. Hand gesture recognition using computer vision techniques. *International Journal of Computer Science*, 10(2):123–145, 2022.

[33] techfort. OpenCV-MediaPipe Hand Gesture Recognition. `https://github.com/techfort/opencv-mediapipe-hand-gesture-recognition`, 2023.

[34] Yang You, Yuhui Wang, Huan Zhang, Zhao Zhang, James Demmel, and Cho-Jui Hsieh. The limit of the batch size. *CoRR*, abs/2006.08517, 2020.

[35] Tong Zhang, Huifeng Lin, Zhaojie Ju, and Chenguang Yang. Hand gesture recognition in complex background based on convolutional pose machine and fuzzy gaussian mixture models. *International Journal of Fuzzy Systems*, 22, 03 2020.

[36] F. Zhao, V. Bazarevsky, A. Vakunov, et al. Mediapipe hands: On-device real-time hand tracking, 2020. Research Paper Google Research, USA.

[37] M.üge Çarıkçı and Figen Özen. A face recognition system based on eigenfaces method. *Procedia Technology*, 1:118–123, 2012. First World Conference on Innovation and Computer Sciences (INSODE 2011).