IKT213 - Machine Vision

# Project report

Autumn 2023

Morten Eidet, Ole-Johan Øvreås, Sivert Espeland Husebø

December 31, 2023

## Mandatory Group Declaration

Each student is solely responsible for familiarizing themselves with the legal aids, guidelines for their use, and rules regarding source usage. The declaration aims to raise awareness among students of their responsibilities and the consequences of cheating. Lack of declaration does not exempt students from their responsibilities.

| | | |
|---|---|---|
| 1. | We hereby declare that our submission is our own work and that we have not used other sources or received any help other than what is mentioned in the submission. | Yes |
| 2. | **We further declare that this submission:** <br>• Has not been used for any other examination at another department/university/-college domestically or abroad. <br>• Does not reference others' work without it being indicated. <br>• Does not reference our own previous work without it being indicated. <br>• Has all references included in the bibliography. <br>• Is not a copy, duplicate, or transcription of others' work or submission. | Yes |
| 3. | We are aware that violations of the above are considered to be cheating and can result in cancellation of the examination and exclusion from universities and colleges in Norway, according to the Universities and Colleges Act, sections 4-7 and 4-8 and the Examination Regulation, sections 31. | Yes |
| 4. | We are aware that all submitted assignments may be subjected to plagiarism checks. | Yes |
| 5. | We are aware that the University of Agder will handle all cases where there is suspicion of cheating according to the university's guidelines for handling cheating cases. | Yes |
| 6. | We have familiarized ourselves with the rules and guidelines for using sources and references on the library's website. | Yes |
| 7. | We have in the majority agreed that the effort within the group is notably different and therefore wish to be evaluated individually. Ordinarily, all participants in the project are evaluated collectively. | Yes |

## Publishing Agreement

Authorization for Electronic Publication of Work The author(s) hold the copyright to the work. This means, among other things, the exclusive right to make the work available to the public (Copyright Act. §2).

Theses that are exempt from public access or confidential will not be published.

| | |
|---|---|
| We hereby grant the University of Agder a royalty-free right to make the work available for electronic publication: | Yes |
| Is the work confidential? | No |
| Is the work exempt from public access? | No |

## Preface

This report documents the work made on our project in the course IKT213G-23H (Machine Vision). We are a group of three computer science students who decided to make a communication tool for people using sign language.

The assignment was given by The University of Agder, Institute for Science and Technology.

Artificial intelligence has been used as a debugging tool, formulating sentences, helping with grammar and gathering sources. Specifically openai chatgpt 3.5[1] and 4.0[2] and phind GPT-4 [3] model has been used during the project

# Contents

# 1 Introduction

In IKT213 Machine Vision, we were tasked with a project to create software which utilizes machine learning and cameras in some way. This report will take you through the development process to give an insight into how we started with an idea, to the actual application we ended up creating. In the next subsections we will describe our project idea and how our product stands out from other similar products to make it successful.

## 1.1 Project idea

The idea for this project stems from a thought experiment that we did in order to come up with a good plan for what type of product we should end up with. We had some constraints as we wanted to use some sort of machine learning process and utilize optical interfaces such as cameras. Other than that we were free to come up with anything.

After considering a multitude of ideas, we landed on the creation of a program that could translate sign language to text in real time. Effective communication is essential in our daily lives and this goes for deaf and mute people also. However, due to the fact that normal people do not (usually) know sign language, communication between the two groups can be somewhat of a hassle. The primary goal of this program is to bridge that communication gap.

## 1.2 Existing sing language translation systems

The project idea is not necessarily revolutionary, but neither an overused one. In terms of sign-language-to-text translators, there already exists a few. However, as for spoken languages, sign language differs in both languages and dialects [4].

After conducting some research, it became evident that there were no Norwegian-based sign language to text translators or applications available. Given the existence of well-documented translators in various languages and the absence of such translators in Norwegian, this provided us with an opportunity to undertake a distinctive project while having an established framework to guide our work.

A common factor for all the open source sign language to text translators was that they utilized an A.I. model in some way. The types of models used varied greatly and this meant that there was flexibility in which approach we decided to use. We will discuss the model we ended up using further in chapter 3.

# 2 Theoretical background

## 2.1 Sign language and communication

Sign language is a visual-gestural language used primarily by deaf and hard-of-hearing individuals for communication. It is a complete linguistic system with its own grammar, vocabulary, and syntax. Sign language is not a universal language, but rather diverse and region-specific, with variations such as American Sign Language (ASL), British Sign Language (BSL), and many others worldwide. Sign languages primarily use handshapes and movements, but facial expressions and body postures are also used to convey meaning, making them a rich and expressive mode of communication [5].

Sign language is vital for the deaf, hard of hearing, and the mute community, allowing them to express thoughts, feelings, and ideas. For many Deaf individuals, sign language is their first language, and it serves as the primary mode of communication within their communities. Sign language enables deaf individuals to participate fully in education, employment, and social interactions[5].

While sign language is an effective mode of communication among the deaf community, it poses challenges when interacting with those who do not understand it. This communication gap can result in exclusion, miscommunication, and difficulties accessing essential services. Sign language interpreters are often used to solve this issue, but their availability may be limited, and communication may not always be as seamless as desired[5].

### 2.1.1 Sign language linguistics

Sign language linguistics is a specialized field that explores the linguistic properties of sign languages used by deaf and hard-of-hearing communities. As mentioned, sign language is a visual-gestural form of communication, with its own grammar, syntax, phonology, morphology, and lexicon [5]. Understanding these linguistic characteristics is essential for developing accurate and effective sign language recognition and translation systems.

## 2.2 Preliminary system design and setup

### 2.2.1 Hardware components

A crucial piece of hardware utilized to train the model was a dedicated GPU, specifically an RTX 3070. Training deep learning models, particularly neural networks made for image or video data is computationally intense and by using a dedicated GPU instead of a CPU, we manage to train the models much more efficiently. However, for tasks like image pre-processing, image capture, and other operations, we relied on a CPU. For model deployment and testing, we used a laptop and we used both an external webcam and built-in webcams for capturing sign language gestures in real-time.

### 2.2.2 Software components

The software used for the solution is Python [6] with OpenCV [7] and Tensorflow [8]. However, the specific version the libraries varied on how the images was processed and how the model was trained, in term of hardware. For example, the images and model were processed and trained using a dedicated GPU, which required a specific version of Tensorflow, depending on the GPU and the used operating system, which in our case was Windows 10. Additional libraries, like cuDNN [9], tensorflow-gpu, the Cuda Toolkit [10] and Keras [11] were also used.

# 3 Implementation

## 3.1 Data collection and dataset used

### 3.1.1 Original planned dataset

Our original plan was to use Noroff's open source "Data-set for recognition of Norwegian Sign Language" [12]. This data-set contained 900 images for each static sign of each letter of the alphabet. For each letter directory, 300 images are with a clean background, 300 images are with noise, and 300 images are done with a female participant. The data-set is tailored to be used to make a Convoluted Neural Network (CNN) model [12].

After a lot of discussion and contemplation, we decided not to move forward with using this data-set. This was because the data-set has a lot of glaring issues. First of all the data-set only contains 900 images per letter which is not enough to make a complex and well-working model. Furthermore, the photos are taken using almost exclusively the same background as well as only using 1-2 participants. Using a data-set with varying participants, lighting conditions, and noise will lead to a much better project. Hence, we have decided to make our own Norwegian Sign Language data-set from scratch.

### 3.1.2 Data collection

For the data collection of our new data-set, we used a Python script utilizing the OpenCV library to capture images through a webcam [2][10]. We then used another script to generate data-sets for training, testing, and validation [4]. When initiated the user can specify the number of images to capture for each letter as well as the delay between the automatic captures. Once you run the program it will let you take the chosen amount of images for each letter of the alphabet.

In the program, we defined a 256x256 Region of Interest (ROI) that was positioned in the center of the camera frame. Participants were instructed to position their hand signs within this region. This region was visualized by a green square. The script was tailored to our specific data needs as we made it capture images for the static letters in A-Å while excluding the dynamic ones ('H' and 'Å').

Each time the user pressed the 'c' key, the script would begin the image-capturing process for the current letter. These images were saved in the designated directory with a naming convention denoting the letter as well as the sequential number of the image. Real-time feedback was provided to the user on the screen, which displayed how many images had been captured for the current letter, with the total count updating until the specified number of images was reached. For our data collection, we had a standard of taking 500 images per letter with a 50-ms delay between each image.

### 3.1.3 Data-set used

For our project, we reached out and got 10 participants which included ourselves. Each participant was tasked with capturing 500 images for each specified letter leading to a comprehensive and diverse dataset. We made sure that every participant used a different background with varying lighting conditions and noise.

In the end, we managed to get 5,000 images for each letter. We then split the dataset up into training, testing, and validation sets with an 80-10-10 split, using script [4]. This means we had 4,000 images for our training data-set and 500 each for our testing and validation data-sets. For our images, we made sure that about 70% of the images had little to no noise and the remaining 30% had a lot of noise. Our application is mostly going to be used indoors with clean backgrounds so we did not need to train it with a lot of noise. For our end product model, we did not use the entirety of the data-set to train the model, instead, we randomly selected some of the images for training. This is due to the approach we ended up taking towards training a model. More about this later. Still, our data collection approach gave us a very good and comprehensive data-set for future use.

## 3.2 Technical implementation of the solution

As mentioned in the introduction, we wanted to use some sort of machine learning to implement our solution. This was deemed a near necessity due to the complex nature of "guessing" hand signs from a live camera-feed. After trying to create our own model, we ended up implementing an approach based on transferred learning. We will talk about the technical details of our implementation here, while discussing the reasons later in chapter 5.

### 3.2.1 Software used

**Anaconda**
To implement our project and keeping our implementation in line with the knowledge gained from the IKT213, we utilized anaconda environments. This allowed us to experiment with different versions of libraries and packages without having to install them system wide. The full list of packages that we ended up installing in our environment can be found on bitbucket, link here: packagelist.txt.

**Tensorflow**
Tensorflow is a free and open source library developed by Google. It primarily serves as a toolkit for machine learning and other A.I. related tasks by providing abstractions and resources such as pre-trained models, processing scripts, gpu-compatability etc [8].

If not the most important part of Tensorflow, one crucial aspect in our use case are its numerous APIs. One of which is the Object-Detection API [13] which we heavily utilized throughout our project.

**OpenCV**
OpenCV is another free and open-source library that we utilized when implementing our solution [7]. It is mainly developed for real-time computer vision and allowed us to preprocess images, access the camera and print out the predictions of our model to the screen - effectively serving as the interface of our program.

**Other software**
In addition to Tensorflow and OpenCV, our project required the support of specific software to optimize performance. The CUDA Toolkit [10] and cuDNN libraries [9] were crucial for this. CUDA enables us to use a GPU when training our model and cuDNN provides GPU-acceleration when working with deep neural networks.

### 3.2.2 Workspace setup and model training

In order to begin working with tensorflow and all the other components, we first had to set up the workspace by installing all required packages and tools mentioned above. While adding libraries and packages through anaconda, either through its built in packet manager or by using pip, is very straight forward, setting up other software such as cuDNN and the Object Detection API from Tensorflow can be tricky, especially on windows. The Object Detection API has a great tutorial that we utilized for this and can be found here: Tensorflow Object Detection API Tutorial.

Now that we had our workspace successfully set up and all the dependencies installed, we had to prepare the data-set. To do this, we started by creating a label map which would contain all of our labels from A to Ø, excluding H and Å, as they are dynamic signs. The code below shows how we created said label map.

```
labels = [{'name':'A', 'id':1}, {'name':'B', 'id':2} .... {'name':'Ø', 'id':27}]

with open(ANNOTATION_PATH + '\label_map.pbtxt', 'w') as f:
    for label in labels:
        f.write('item { \n')
        f.write('\tname:\'{}\'\n'.format(label['name']))
        f.write('\tid:{}\n'.format(label['id']))
        f.write('}\n')
```

Listing 1: Script to create label map

We then labeled the images using a tool called LabelImg [14]. This process involved going through each image and marking areas of interest which covered the signs that we wanted our model to recognize. While our dataset consisted of a total of 135 000 images, we only needed to use a fraction as we decided to use a pre-trained model, which then enabled us to label them all. Once all the images were labeled, we moved on to generating TFRecords. TFRecords are files that Tensorflow uses in the training process to link images with their labels and the bounding box area that we specified in LabelImg. The script we used to create the TFRecords can be found here: TFRecord generation script. With our data-set now prepared and formatted correctly we were ready to configure the training pipeline.

As mentioned earlier we decided to utilize transferred learning to train our model. This means that we use a pre-trained model and resume its training from a checkpoint but now using our data-set as its training material. For our implementation we used the *SSD MobileNet V2 FPNLite 640x640* model from the Tensorflow Model Zoo [15]. We discuss the reason behind this choice in greater detail in chapter 5. The model was downloaded from the repository and extracted into a local folder in our workspace. We then had to modify the configuration file by manually adjusting the different paths to, among other things, the TFRecords we just made.

Now that the workspace was set up, dependencies installed, data-set processed and training pipeline configured, there was only one thing left to do; train the model. This was done simply by running the *model_main_tf2.py*[16] script which was handily provided to us by the Object Detection API. The model then began training for the specified amount of steps, resuming original training progression from the last available checkpoint of the pre-trained model and saves new checkpoints along the way. Image pre-processing and augmentation was model specific and predetermined by the models configuration. As such we did not perform any image manipulation ourselves.

During the training process, we used Tensorboard [17] to visualize the training progression by keeping an eye on the loss and validation metrics. Once the model completed training for the total specified steps, it saved its progression as a checkpoint which we could use to perform inference.

Our implementation was greatly inspired by a mask detection tutorial found here: [18].

### 3.2.3 Model usage

The final step in our implementation was using the trained model for real-time predictions. We loaded the last checkpoint saved by the model and then used OpenCV to access the live camera feed before processing each frame individually and using the model to predict which, if any, signs it could decipher. If the model detected any signs, it would draw a box around the sign it detected and display what sign it thought it was. It would also display how confident it was in the top left corner.

As this project was supposed to let users write by using sign language to the camera, we implemented some simple logic as a proof of concept that made the program type the sign it detected to the terminal once it had detected the sign continuously for 2 seconds without interruptions. It would then not type it again until it re-detected the same sign after an interruption in order to avoid duplicate text 9.

# 4 Results

## 4.1 Overview

In this section of the result, we will present the results we achieved in the development of our Norwegian sign language detector. Our focus was primarily on getting static signs to work which encompassed mostly one-handed alphabet signs with some two-handed alphabet signs. Some of our key achievements are the ability to recognize multiple signs at once, evaluating the accuracy of each sign, and the ability to print the signs to the terminal.

## 4.2 Static sign recognition

### 4.2.1 One-handed alphabet signs

For our project we successfully implemented the ability to recognize all static one-handed alphabet signs. The following one-handed letters were included in our detection model:

- A, B, C, D, E, F, G, I, K, L, M, N, O, R, S, T, V, W, Y, Æ

For each of these signs, our model was trained to identify their specific hand shape and position. The recognition process involved capturing these static signs live through the use of a camera interface. This live camera feed then preprocessed the images live for better detection. Our model became very efficient at distinguishing the different letters, and would reliably guess the correct signs. All of the one-handed signs we implemented worked well, with some slight differences in performance based on the sign. Here is an example of our model predicting a one-handed sign:

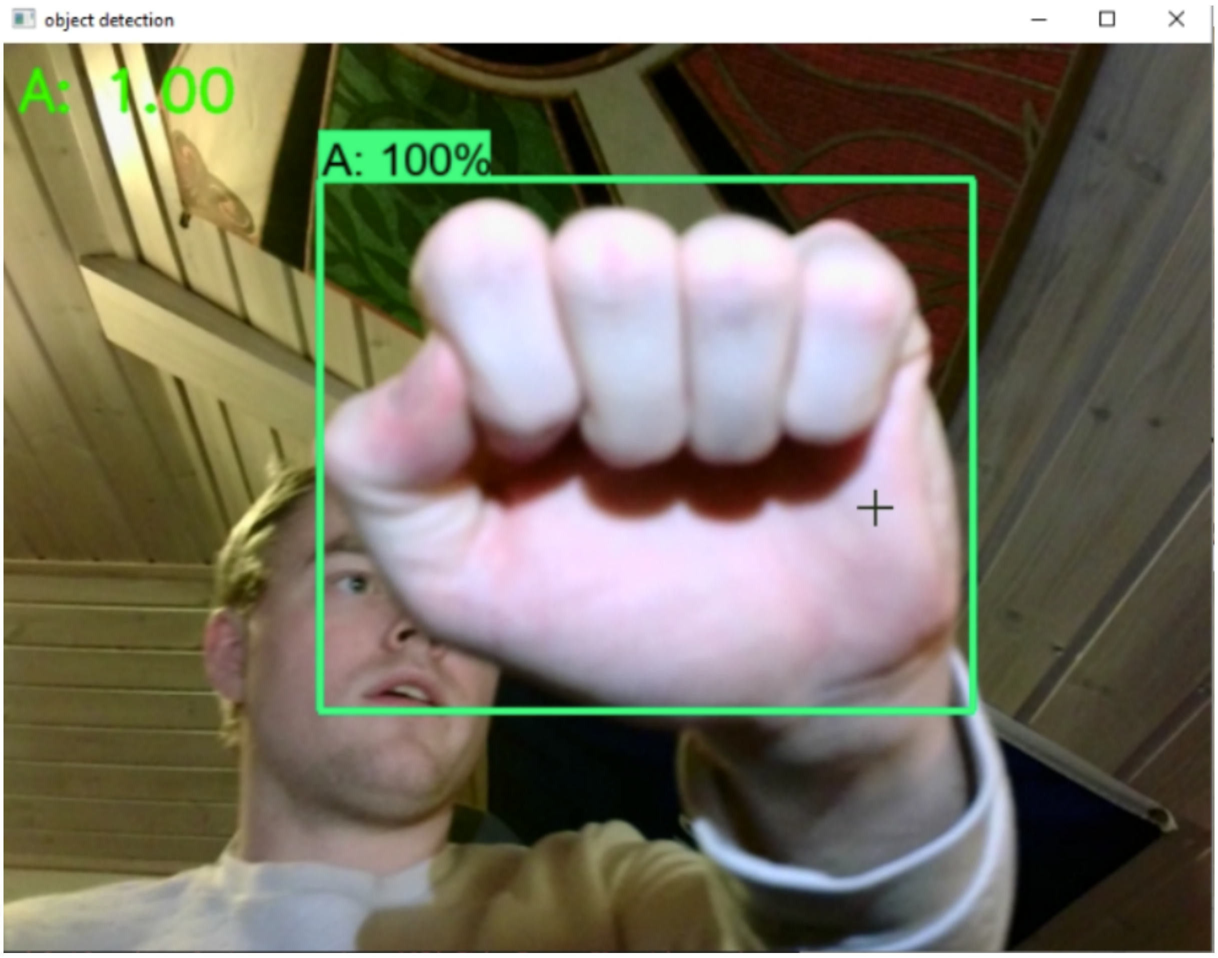


Figure 1: One-handed sign recognition example

#### 4.2.2 Two-handed alphabet signs

In addition to one-handed sign recognition our project also supported some two-handed signs. These signs were used when their one-handed counterpart were dynamic. The two-handed letters we included were:

- J, P, Q, U, X, Z, Ø

Two-handed signs are more complex than one-handed signs as they require more coordinated movement and positioning of both hands. These complexities were both a gift and a curse. Creating the images, and training the model was more complex for our model, but when actually using our model live it was better at detecting two-handed signs than one-handed signs. The uniqueness of the two-handed signs gave the model an easier time accurately recognizing them. We successfully implemented and recognized all the two-handed letters we wanted to use. Here is an example of our model predicting a two-handed sign:

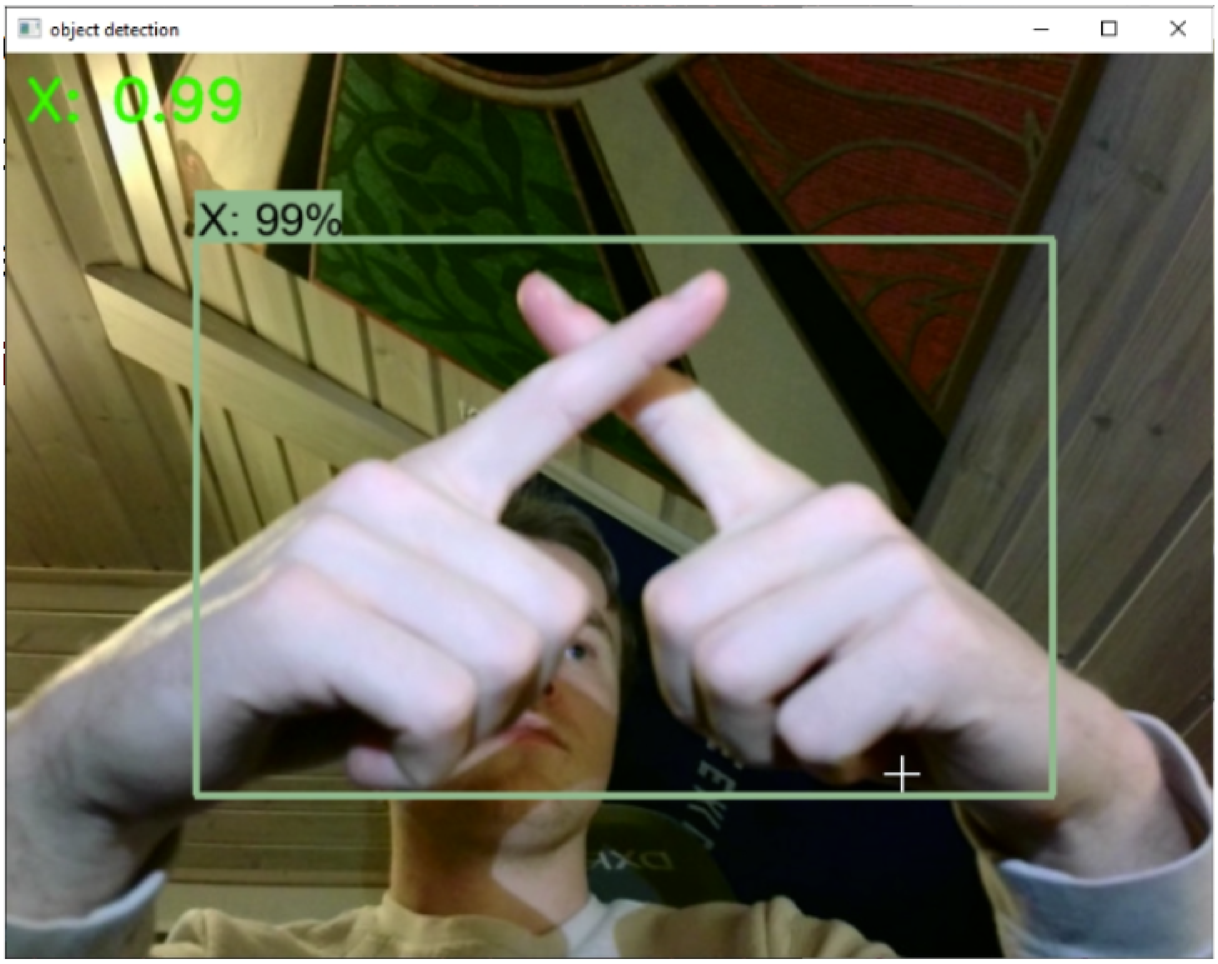


Figure 2: Two-handed sign recognition example.

## 4.3 Multiple sign recognition

One of the accomplishments of our sign language detector is its ability to recognize multiple signs at the same time. This feature is significant as we have the groundwork that lets us move beyond the recognition of isolated signs. There is no limit to the number of signs our model can detect at once. In theory, the only limitation is the number of different signs you can fit into the web camera feed. This feature gives our application the ability to be used by multiple users at the same time with no detriments. Here is an example of our application recognizing multiple signs at once:

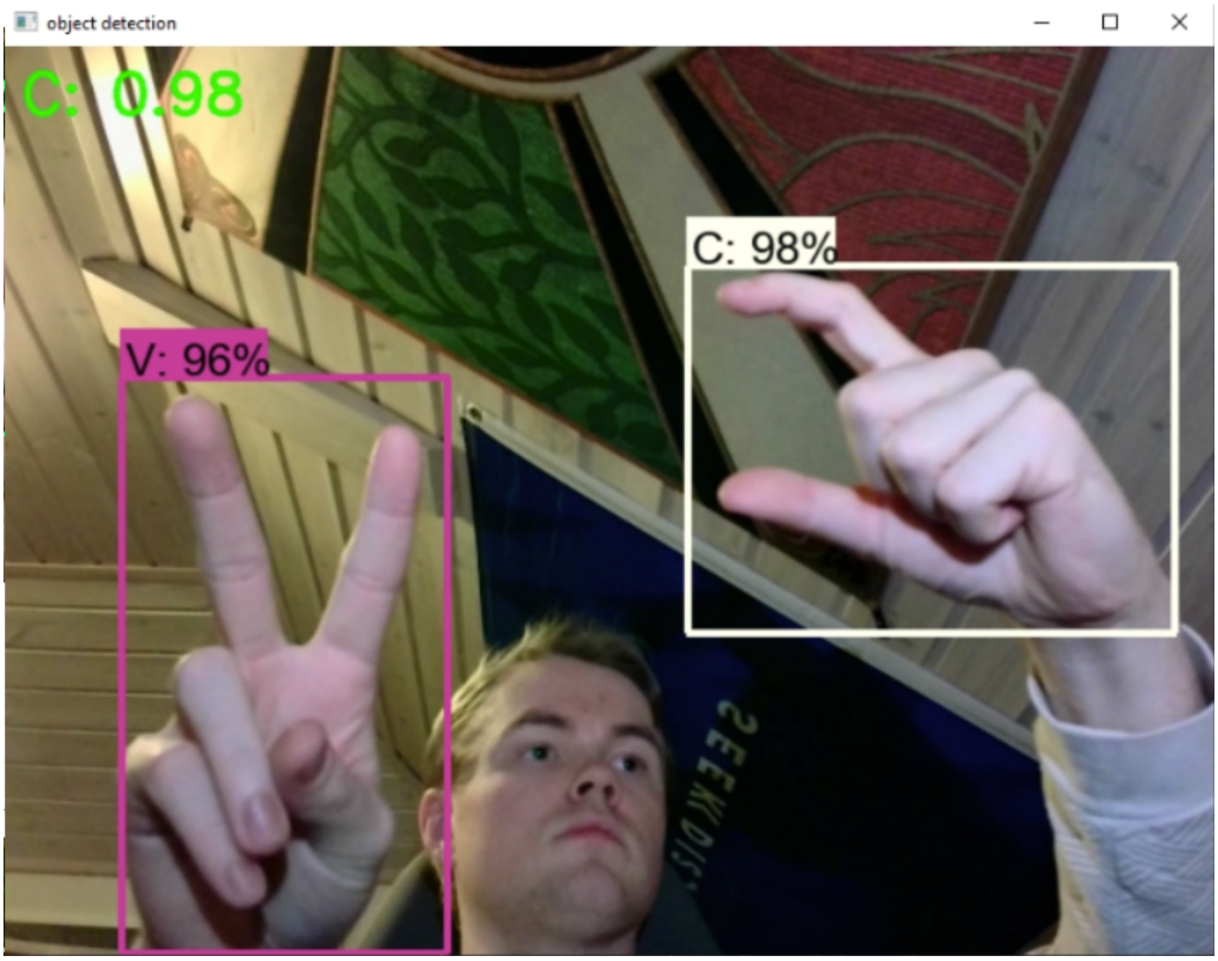


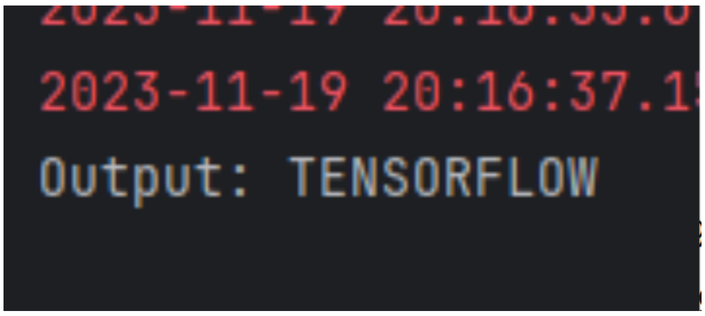
Figure 3: Multiple signs at once example

## 4.4 Accuracy score

As seen in the images above our model gives a percentage accuracy score for each prediction it makes. This score is basically how confident the model is for each prediction. Whenever a sign is recognized on the screen there is calculated an accuracy score for every possible sign, and the sign that has the highest score is the one the model will guess. This score measure reflects how well our system can recognize and interpret signs correctly. This is a nice achievement as it both serves as a tool for evaluating the performance of the detector, and it helps us in refining our model.

## 4.5 Printing sign recognition

A crucial achievement for our Norwegian sign language detector is its ability to print recognized signs to a user interface like the terminal. To make our application work as an actual translation tool we managed to implement a feature where the system prints the recognized sign to the terminal after it has been consistently detected for two or more seconds. This ensures usability and enhances the user experience. A user can keep writing to the terminal by showing different signs to the terminal. If the user wants the same letter two times in a row he just has to remove his hand and perform the same sign again. This functionality is essential for making our application an actual tool for converting sign language to text, which enables communication and translation in real-time. Here is how the printed signs will look in the terminal:



```
2023-11-19 20:16:37.1
Output: TENSORFLOW
```

Figure 4: Writing to the terminal example

# 5 Discussion

## 5.1 Challenges or obstacles to implementing planned tasks.

As in all coding related projects, there were a few bumps along the road. The challenges ranged from learning sign language, figuring out what to prioritize for the project in terms of signs, to more technical aspects like choice of model and hardware related issues.

### 5.1.1 Sign language

Although we all agreed that a sign language translator was an exciting idea, we quickly realized that none of us actually knew sign language. This realization meant that we had to decide where to start and define our goals for learning the necessary steps. Once the goals were set, and a plan was made, we had to collect a dataset. We used 10 candidates to gather a total of approximately 135,000 pictures. However, this implied not only learning certain signs but also teaching them to others. The result was a reasonably good dataset, not to say that it was perfect. Some of the signs in the images turned out to be incorrect and had to be retaken, while others lacked in quality.

### 5.1.2 Image quality and diversity

The quality of the images is a significant factor in the ability to train a model based on them. The first reason why good-quality images are essential is for proper feature extraction. Feature extraction from the hands is perhaps the most crucial aspect in the images; we need sufficient quality to be able to distinguish between different signs when training the model. On the other hand, as this product is intended to be used by any type of camera, including lower-end cameras such as cheap webcams, we saw the necessity of limiting the quality of our images so that the model would not get accustomed to high quality images. Training a model on a ”poor quality” data-set would still allow it to work on images taken by higher-end cameras while it is more tricky the other way around.

The dataset also needed to have a certain degree of diversity. Each sign had to be captured from different angles to achieve the best possible end result. A factor here that could have affected the final product was that some angles for different signs could have made the sign impossible to be seen by the camera and were not filtered out. The end result was that the model trained on a few ’false’ images.

Another issue related to the images was the overall lack of variety within the dataset. This problem originated during the creation of the dataset. During image capture, a 50 ms delay was used between each image, resulting in images that were too similar. Consequently, the dataset exhibited too many similarities, impacting the validation process of the model training. Furthermore, when allocating 10% of the images for letter validation during training, we encountered challenges with an overfitted model. The overfitting issue persisted until the very end, prompting us to consider adopting a pre-trained model.

### 5.1.3 Choice of model

Due to the nature of our project, implementing object recognition, we determined that a *convolutional neural network* was the best-suited model. The discussion, however, revolved around whether to create our own or use a pre-trained model. From the start, we were determined to create our own model. The training process was a time-consuming ordeal that did not necessarily yield the best results. Another problem was our uncertainty about why the model did not perform as expected. Was it the images, the preprocessing, augmentation, the balance between training-validation-testing data, or the model itself that was the issue? The reasons could be numerous, and training time was lengthy. Therefore, after careful consideration, we chose to transition from creating our own model to using a pre-trained one.

When using a pre-trained model, we have to employ a technique called transferred learning. Transferred learning involves taking a model that is partially trained and adapting it to a different but related task. For us, it meant finding a model trained on object recognition and continue training it on our data [19].

TensorFlow 2 offers a variety of different models through their model zoo [15]. A common factor for all the models is that they are trained on the same dataset, COCO 17. The first implementation of a transferred learning model was done by following the official tutorial [20], which used the *SSD ResNet50 V1 FPN 640x640 (RetinaNet50)*[18]. However, this model did not provide a satisfactory results for our use case.

The next model we experimented with was *SSD MobileNet V2 FPNLite 320x320*. Compared to RetinaNet50, MobileNet is a more lightweight convolutional neural network specifically designed for mobile edge devices, such as cellphones and web cameras. It also has a better-balanced pre-trained model in terms of speed and accuracy [15]. After training the MobileNet model, we observed a remarkable difference in terms of performance. However, the results were still too inaccurate for our liking. One reason for this could be the reduction in image resolution, which was reduced from 640x640 to 320x320.

Due to the differences in results between RetinaNet50 and MobileNet 320x320, we once again changed the model to *SSD MobileNet V2 FPNLite 640x640*, which is the same MobileNet model but with the resolution changed to 640x640. The results after changing were by far the best and ended up being the model that we chose to work with. The reasoning for changing model to one with higher resolution, was that the images would provide more spatial information. More spatial information would further help the model detect and learn smaller and more intricate differences in the data-set compared to the MobileNet model with its lower resolution. Another key couple of factors for the choice of model for the project was the improved localization accuracy and better feature representation. The improved feature extraction is a key aspect in the model due to the similarity between some of the signs.

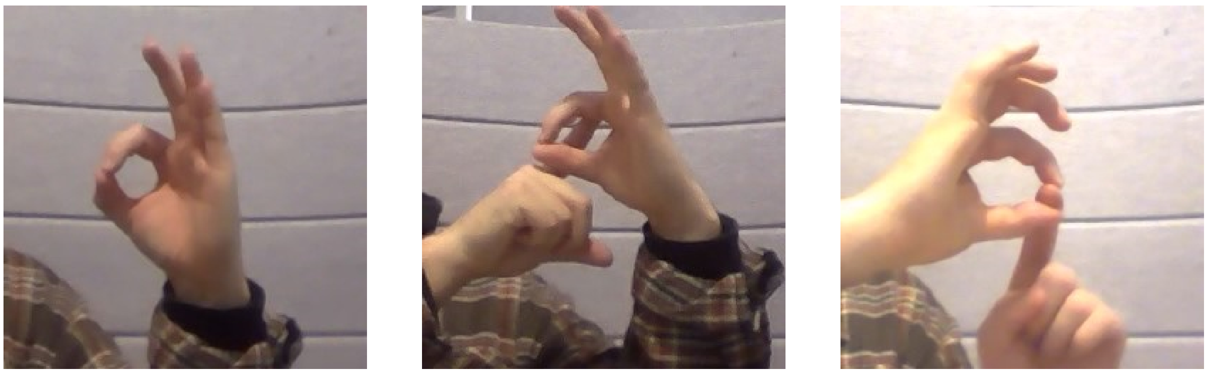


Figure 5: Illustration of similarities in different signs; O, Ø and Q

As illustrated in figure 5.1.3, the differences between some signs are minimal, even in a controlled environment. Based on these finite nuances between the signs, we quickly concluded that MobilNet 640x640 would be a better fit for us than the 320x320 version.

#### 5.1.4 Hardware from UIA

The time it took from pre-processing to a trained model was a long, tedious process that could take several hours. The waiting time could have been reduced by utilizing UiA's hardware. Not only could we have saved time on processing and augmenting the images, but also on training the model. Additionally, we could have skipped the process of configuring all the necessary packages for our separate hardware, specifically for the GPU. The reasoning behind why we did not utilize this was due to the fact that we were not aware of it being a possibility until after we had completed the project. Although we are happy with the current result of the project one can only wonder where we would be with better, more powerful hardware.

### 5.2 Use-case scenarios

1. **Scenario**: Anna, a deaf student, enrolls in a public school where none of the teachers or students understand Norwegian Sign Language. The school administration is concerned about how Anna will be able to interact and communicate effectively with her peers and educators.

   **Solution**: The school decides to utilize our real-time sign language translation software in classrooms. A camera is set up, and whenever Anna wishes to ask or answer a question, the system translates her signed sentences into text, which is then projected onto the smartboard for everyone to see. This enables seamless communication between Anna and her teachers and classmates, ensuring that she is not left behind academically and can actively participate in classroom discussions.

   **Outcome**: Anna's peers become curious and show interest in understanding sign language themselves. While they start picking up some signs, the translation system remains a crucial bridge for comprehensive communication. Anna feels included, engaged, and on par with her classmates, fostering an inclusive educational environment.

2. **Scenario**: Erik, a company executive who is mute, is attending a regional conference with potential clients and stakeholders. Most attendees do not understand sign language, and Erik's presentation is pivotal for the company's future partnerships.

   **Solution**: Erik, aware of the communication barrier, sets up a laptop with a built-in camera and our sign language translation software. As he starts his presentation and signs, the system translates his signing into real-time text that is projected onto a screen for all attendees to follow. Not only does this solution allow Erik to communicate his points effectively, but it also impresses the audience with the seamless integration of technology to bridge communication gaps.

   **Outcome**: The potential clients and stakeholders are impressed by Erik's presentation and the company's innovative approach to inclusivity. This not only results in successful partnerships but also raises awareness about the need for such solutions in diverse workplace settings. Erik's success story becomes an inspiration for other businesses to foster inclusivity and utilize technology to its fullest potential.

## 5.3 Model evaluation

When we look at and discuss the performance of our sign language detection model we have to consider both the successes and limitations we encountered. Our model had a good level of accuracy in recognizing most of the Norwegian sign language alphabet signs we trained it on. We are satisfied with the overall performance of our model, especially when you take into account the complexity of sign language detection. The model was able to correctly interpret the entirety of the static alphabet signs we trained it on with pretty high accuracy.

The biggest challenge we encountered was the overlap in recognition of signs that were visually similar. Certain signs like 'M' and 'N' have a close resemblance to each other and led to some instances of misinterpretation by the model. Moreover, some signs like 'D' needed to be angled and rotated in a certain way to have the best results. Throughout our project, we tested our models using different backgrounds, lightning conditions, angles, rotations, and the distance from our hand to the camera to determine where our model had the best performance. These tests helped us in figuring out how to best improve our model, but they also showed us the complexities involved in real-world sign language translation.

These challenges and kinks gave us an indication that our model need further refinement before it can be used as an actual tool. The model should be enhanced to capture finer nuances in hand movements and positions. It should also have good accuracy for all signs in all kinds of environments. Still, all in all we are very satisfied with the outcome of our model and we believe that it shows great promise in the development of a working Norwegian sign language detector.

## 5.4 Future work

The development of our Norwegian Sign Language detector is a good step towards helping the deaf and dumb community. However, there still exists a vast amount of potential for the future enhancement and expansion of our application. For our future work, we would like to improve and extend several capabilities of our product.

### 5.4.1 Dynamic signs

Our current Norwegian hand sign detector model is limited to only being able to recognize static hand signs. We have functionality for most of the static signs in the Norwegian sign language, but to make our detector even better we plan to integrate the ability to recognize dynamic signs. As most of the sign language deaf and dumb people use on a daily basis is dynamic this ability would be crucial for making an actual usable application. This task would involve capturing and integrating frames of hand movements over time instead of using singular static images.

### 5.4.2 Completing the Alphabet

For our application to have as much functionality as possible we mixed the use of the one-handed and two-handed Norwegian alphabet. This gave us a static version of most of the alphabet excluding 'H' and 'Å'. In the future, we would like the completely finish the entire Norwegian alphabet for both the one-handed and two-handed signing. This would make our application and model more complete and useful for sign language users.

### 5.4.3 Implementing Words and Numbers

Once we have managed to start implementing dynamic signs we would like to start incorporating entire words from the Norwegian sign language dictionary. Individuals who actively speak using sign language use entire words instead of just letters to have effective communication. Our implementation of dynamic signs would then naturally evolve into incorporating more and more commonly used words in our application. Furthermore, the implementation of numbers would also increase the usability.

### 5.4.4 Optimizing for Real-Time signing

As we want our application to be a useful tool for live sign language translation we would like to optimize our model for real-time interpretation. The point of this would be to make our application as user-friendly and efficient as possible for actual users. This would involve improving both the speed and accuracy of the model so users can receive immediate feedback and translation while signing.

## 6 Conclusion

We set out to create a sign language-to-text translator and chose to focus on the static one-handed alphabet, using two-handed signs instead of their one-handed dynamic counterparts. In terms of the project vision, we were mostly successful. We managed to create a model that could recognize every letter of the Norwegian alphabet, disregarding the dynamic letters with a high percentage of certainty. That being said, there are improvements to be made; the model is too dependent on having the right angle to interpret some of the signs or may confuse them with others. A proposed solution for this could be to use larger and more varied parts of our dataset for training the model.

However, challenges emerged in the course of our project. While making our own model, we had troubles with overfitting, regardless of the letter in the alphabet. This emphasizes the importance of a good dataset with enough variations to avoid the issue.

Furthermore, a lot of time was lost due to hardware limitations. The efficiency and speed of processing the dataset, augmenting it, and then training our model on the augmented dataset were tied to the available computational resources. Or so we thought; during the project presentation, we were informed that we could have borrowed hardware resources from UIA. If we had known this from the start, we would probably have saved hours in just processing and training, and might have included the dynamic letters of the alphabet. These are only speculations, of course.

For the future work on this project, we would like to fine-tune the model on static signs, making it more accurate, as well as finishing the alphabet. This implies that we have to train the model on dynamic signs, which, in turn, could be the foundation for implementing words and phrases.

In conclusion, we are happy with the end result and were able to implement the static alphabet, which was the goal. During the project period, we had to overcome a few challenges related to overfitting, practical implementation hurdles, and hardware limitations. We did overcome the problems and are now ready to implement the next steps toward a fully functioning translator.

## References

[1] Open Ai, “Chatgpt 3.5,” *Large language model*, 2022.

[2] Open Ai, “Chatgpt 4.0,” *Large language model*, 2023.

[3] Phind, “Phind GPT-4,” *Large language model*, 2023.

[4] *What are the different types of sign language?* [Online; accessed 12. Oct. 2023], May 2023. [Online]. Available: `https://www.signsolutions.uk.com/what-are-the-different-types-of-sign-language/`.

[5] *Sing language*, [Online; accessed 12. Oct. 2023]. [Online]. Available: `https://en.wikipedia.org/wiki/Sign_language`.

[6] *Welcome to Python.org*, [Online; accessed 24. Nov. 2023], Nov. 2023. [Online]. Available: `https://www.python.org`.

[7] *Home*, [Online; accessed 24. Nov. 2023], Nov. 2023. [Online]. Available: `https://opencv.org`.

[8] *TensorFlow*, [Online; accessed 24. Nov. 2023], Nov. 2023. [Online]. Available: `https://www.tensorflow.org`.

[9] *CUDA Deep Neural Network*, [Online; accessed 24. Nov. 2023], Nov. 2023. [Online]. Available: `https://developer.nvidia.com/cudnn`.

[10] *CUDA Toolkit - Free Tools and Training*, [Online; accessed 24. Nov. 2023], Nov. 2023. [Online]. Available: `https://developer.nvidia.com/cuda-toolkit`.

[11] K. Team, *Keras: Deep Learning for humans*, [Online; accessed 24. Nov. 2023], Nov. 2023. [Online]. Available: `https://keras.io`.

[12] S. Kadry and B. Svendsen, “A Dataset for recognition of Norwegian Sign Language,” *International Journal of Mathematics, Statistics, and Computer Science*, vol. 2, Aug. 2023, ISSN: 2704-1077. DOI: `10.59543/ijmscs.v2i.8049`.

[13] *models/research/object_detection at master · tensorflow/models*, [Online; accessed 25. Nov. 2023], Nov. 2023. [Online]. Available: `https://github.com/tensorflow/models/tree/master/research/object_detection`.

[14] *labelImg*, [Online; accessed 25. Nov. 2023], Nov. 2023. [Online]. Available: `https://pypi.org/project/labelImg`.

[15] vighneshbirodkar, *TensorFlow 2 Detection Model Zoo*, [Online; accessed 25. Nov. 2023], 2021. [Online]. Available: `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md`.

[16] *models/research/object_detection at master · tensorflow/models*, [Online; accessed 25. Nov. 2023], Nov. 2023. [Online]. Available: `https://github.com/tensorflow/models/blob/master/research/object_detection/model_main_tf2.py`.

[17] *TensorBoard | TensorFlow*, [Online; accessed 25. Nov. 2023], Oct. 2023. [Online]. Available: `https://www.tensorflow.org/tensorboard`.

[18] *RealTimeObjectDetection/Tutorial.ipynb at main · nicknochnack/RealTimeObjectDetection*, [Online; accessed 25. Nov. 2023], Nov. 2023. [Online]. Available: `https://github.com/nicknochnack/RealTimeObjectDetection/blob/main/Tutorial.ipynb`.

[19] *Transfer learning*, [Online; accessed 25. Nov. 2023]. [Online]. Available: `https://en.wikipedia.org/wiki/Transfer_learning`.

[20] *Tensorflow 2 Object Detection-API tutorial*, [Online; accessed 18. Nov. 2023]. [Online]. Available: `https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html`.

## List of Figures

```
import cv2
import os
import string


def capture_images(cap, dataset_type, class_name, num_images, delay, start_x, start_y, end_x, end_y):
    base_dir = os.path.join(os.getcwd(), '../datasets')
    target_dir = os.path.join(base_dir, dataset_type, class_name.upper())
    os.makedirs(target_dir, exist_ok=True)

    existing_files = os.listdir(target_dir)
    count = max([get_image_count(f, class_name) for f in existing_files] + [0])
    start_count = count + 1

    capturing = False
    while True:
        ret, frame = cap.read()
        if not ret:
            print("Error: Couldn't read from the camera.")
            return

        roi = frame[start_y:end_y, start_x:end_x].copy()
        cv2.rectangle(frame, (start_x, start_y), (end_x, end_y), (0, 255, 0), 2)

        if capturing:
            instruction_text1 = f'Capturing for {class_name}...'
            img_name = f"{class_name.upper()}_{count + 1}.png"
            img_path = os.path.join(target_dir, img_name)
            cv2.imwrite(img_path, roi)
            print(f"{img_name} written!")
            count += 1
            cv2.waitKey(delay)
        else:
            instruction_text1 = f'Press "c" to start capturing for letter {class_name}.'

        instruction_text2 = f'Captured {count - start_count + 1}/{num_images} for {class_name}.'
        cv2.putText(frame, instruction_text1, (10, 30), cv2.FONT_HERSHEY_SIMPLEX,
        0.7, (0, 255, 0), 2, cv2.LINE_AA)

        cv2.putText(frame, instruction_text2, (10, 60), cv2.FONT_HERSHEY_SIMPLEX,
        0.7, (0, 255, 0), 2, cv2.LINE_AA)

        cv2.imshow('Capture Images', frame)

        k = cv2.waitKey(1)
        if k == ord('c') and not capturing:
            capturing = True

        if (count - start_count + 1) >= num_images:
            break

def get_image_count(filename, class_name):
    try:
        if filename.startswith(class_name.upper() + '_'):
            return int(filename.split('_')[-1].split('.')[0])
        return 0
    except ValueError:
        return 0
```

Listing 2: Script to capture images - Part 1

```python
if __name__ == "__main__":
    dataset_type_dict = {
        1: "training_data",
        2: "testing_data",
        3: "validation_data"
    }

    dataset_type_input = 0
    while dataset_type_input not in dataset_type_dict:
        try:
            dataset_type_input = int(
                input("Choose the dataset type: \n1 for training_data\n2 for testing_data
                \n3 for validation_data\n")
            )
        except ValueError:
            print("Please enter a valid number (1, 2, or 3).")
    dataset_type = dataset_type_dict[dataset_type_input]

    num_images = int(input("Enter the number of images you want to capture for each letter: "))
    delay = int(input("Enter the delay between automatic captures (in milliseconds): "))

    cap = cv2.VideoCapture(0)
    ret, frame = cap.read()
    if not ret:
        print("Error: Couldn't read from the camera.")
        cap.release()
        exit()

    height, width, _ = frame.shape
    center_x, center_y = width // 2, height // 2
    start_x, start_y = center_x - 128, center_y - 128
    end_x, end_y = center_x + 128, center_y + 128

    # Define custom letter sequence
    letter_sequence = list(string.ascii_uppercase)
    letter_sequence.remove('H')  # Remove H
    letter_sequence.extend(['AE', 'OE'])  # Add AE and OE

    for letter in letter_sequence:
        capture_images(cap, dataset_type, letter, num_images, delay, start_x, start_y, end_x, end_y)

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

Listing 3: Script to capture images - Part 2

```python
import os
import shutil
import random

base_dir = "../datasets"  # Replace with the path to your dataset folder
training_dir = os.path.join(base_dir, "training_data")
validation_dir = os.path.join(base_dir, "validation_data")
testing_dir = os.path.join(base_dir, "testing_data")

for letter in os.listdir(training_dir):
    letter_dir = os.path.join(training_dir, letter)

    if not os.path.isdir(letter_dir):
        continue

    os.makedirs(os.path.join(validation_dir, letter), exist_ok=True)
    os.makedirs(os.path.join(testing_dir, letter), exist_ok=True)

    images = os.listdir(letter_dir)
    random.shuffle(images)  # Shuffle to randomly select images

    for i in range(500):
        shutil.move(os.path.join(letter_dir, images[i]), os.path.join(validation_dir, letter))
    for i in range(500, 1000):
        shutil.move(os.path.join(letter_dir, images[i]), os.path.join(testing_dir, letter))

print("Images moved successfully.")
```

Listing 4: Script to create the data-sets

```python
import cv2
import numpy as np
from pathlib import Path
from tensorflow.keras.preprocessing.image import ImageDataGenerator


def preprocess_image(img_path: Path, output_dir: Path) -> np.array:
    img = cv2.imread(str(img_path))
    if img is None:
        raise ValueError(f"Image not found at {img_path}")

    img = cv2.resize(img, (224, 224))

    img = img / 255.0

    # ---- Skin Masking ----
    # gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # lower_bound = np.array([0, 40, 30], dtype="uint8")
    # upper_bound = np.array([43, 255, 254], dtype="uint8")

    # skin_mask = cv2.inRange(hsv, lower_bound, upper_bound)

    # skin = cv2.bitwise_and(gray, gray, mask=skin_mask)

    # ---- Thresholding ----
    # _, thresh = cv2.threshold(skin, 127, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    # ---- Canny Edge Detection ----
    # edges = cv2.Canny(skin_mask, 100, 200)

    cv2.imwrite(str(output_dir / img_path.name), img)


def apply_augmentation(img: np.array, output_dir: Path, img_file: Path):
    datagen = ImageDataGenerator(
        rescale=1. / 255,
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        horizontal_flip=True
    )

    img = img.reshape((1,) + img.shape + (1,))  # Reshape the image to (1, 128, 128, 1)
    for i, batch in enumerate(datagen.flow(img, batch_size=1)):
        augmented_img = batch.squeeze()
        augmented_img_path = output_dir / f"{img_file.stem}_aug_{i}{img_file.suffix}"
        cv2.imwrite(str(augmented_img_path), augmented_img * 255)
        if i == 4:
            break
```

Listing 5: Script to preprocess images - Part 1

```python
def process_dataset(dataset_type: str):
    input_dir = base_input_dir / dataset_type
    output_dir = base_output_dir / dataset_type
    output_dir.mkdir(parents=True, exist_ok=True)

    for class_dir in input_dir.iterdir():
        if class_dir.is_dir():
            output_class_dir = output_dir / class_dir.name
            output_class_dir.mkdir(exist_ok=True)

            for img_file in class_dir.iterdir():
                if img_file.suffix in ['.jpg', '.jpeg', '.png']:
                    preprocess_image(img_file, output_class_dir)

                    # if dataset_type == 'training_data':
                    # apply_augmentation(img, output_class_dir, img_file)


base_input_dir = Path('../datasets')
base_output_dir = Path('processed_datasets')

datasets = ['training_data', 'testing_data', 'validation_data']

for dataset in datasets:
    process_dataset(dataset)

print("Preprocessing and Augmentation Completed!")
```

Listing 6: Script to preprocess images - Part 2

```
import os
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
from pathlib import Path

os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'

print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))

base_dir = Path('processed_datasets')

train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2
)

validation_datagen = ImageDataGenerator(rescale=1. / 255)

test_datagen = ImageDataGenerator(rescale=1. / 255)

image_size = (224, 224)

train_generator = train_datagen.flow_from_directory(
    base_dir / 'training_data',
    target_size=image_size,
    batch_size=32,
    class_mode='categorical'
)

validation_generator = validation_datagen.flow_from_directory(
    base_dir / 'validation_data',
    target_size=image_size,
    batch_size=32,
    class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    base_dir / 'testing_data',
    target_size=image_size,
    batch_size=32,
    class_mode='categorical'
)
```

Listing 7: Script to train our model - Part 2

```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
    layers.BatchNormalization(),
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(256, (3, 3), activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(256, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),

    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),

    layers.Dense(train_generator.num_classes, activation='softmax')
])

optimizer = optimizers.Adam(learning_rate=0.00005)

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

early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = model.fit(
    train_generator,
    epochs=30,
    validation_data=validation_generator,
    callbacks=[early_stopping]
)

test_loss, test_acc = model.evaluate(test_generator)
print(f"Test Accuracy: {test_acc * 100:.2f}%")

model.save('my_model.h5')

print("Model Training Completed!")
```

Listing 8: Script to train our model - Part 1

```python
import os
import time

from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils
from object_detection.builders import model_builder
import cv2
import numpy as np
import tensorflow as tf
from object_detection.utils import config_util

# Paths redacted

# Load pipeline config and build a detection model
configs = config_util.get_configs_from_pipeline_file(CONFIG_PATH)
detection_model = model_builder.build(model_config=configs['model'], is_training=False)

# Restore checkpoint
ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
ckpt.restore(os.path.join(CHECKPOINT_PATH, 'ckpt-31')).expect_partial()


@tf.function
def detect_fn(image):
    image, shapes = detection_model.preprocess(image)
    prediction_dict = detection_model.predict(image, shapes)
    detections = detection_model.postprocess(prediction_dict, shapes)
    return detections


category_index = label_map_util.create_category_index_from_labelmap(ANNOTATION_PATH + '/label_map.pbtxt')

# Setup capture
cap = cv2.VideoCapture(0)
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

last_detected_label = None
detection_start_time = None
detection_interval = 2  # seconds
can_print = True
init_print = True
```

Listing 9: Script to use model to perform inference - Part 1

```
while True:
    ret, frame = cap.read()
    frame = cv2.flip(frame, 1)
    image_np = np.array(frame)

    input_tensor = tf.convert_to_tensor(np.expand_dims(image_np, 0), dtype=tf.float32)
    detections = detect_fn(input_tensor)

    num_detections = int(detections.pop('num_detections'))
    detections = {key: value[0, :num_detections].numpy()
                  for key, value in detections.items()}
    detections['num_detections'] = num_detections

    # detection_classes should be ints.
    detections['detection_classes'] = detections['detection_classes'].astype(np.int64)

    label_id_offset = 1
    image_np_with_detections = image_np.copy()

    # Find the detection with the highest score
    max_score_index = np.argmax(detections['detection_scores'])
    max_score = detections['detection_scores'][max_score_index]
    max_class = detections['detection_classes'][max_score_index] + label_id_offset

    if init_print:
        print("Output: ", end='')
        init_print = False

    if max_score > 0.1:  # Only consider detections with score greater than 0.5
        # Get the label of the class with the highest score
        label = category_index[max_class]['name']

        # Check if the label is continuously detected
        if label == last_detected_label:
            if can_print and (time.time() - detection_start_time) >= detection_interval:
                print(label, end='')
                can_print = False  # Prevent further prints until label is redetected
        else:
            last_detected_label = label
            detection_start_time = time.time()
            can_print = True  # Reset printing permission for a new label

        # You can also display this information on the window
        cv2.putText(image_np_with_detections, f'{label}: {max_score:.2f}',
                    (10, 35), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2, cv2.LINE_AA)

    viz_utils.visualize_boxes_and_labels_on_image_array(
        image_np_with_detections,
        detections['detection_boxes'],
        detections['detection_classes'] + label_id_offset,
        detections['detection_scores'],
        category_index,
        use_normalized_coordinates=True,
        max_boxes_to_draw=5,
        min_score_thresh=.5,
        agnostic_mode=False)

    cv2.imshow('object detection', cv2.resize(image_np_with_detections, (800, 600)))

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

Listing 10: Script to use model to perform inference - Part 2