HANDWAVE

American Sign Language Recognition System

Aya Khames Khairy

Mohamed Metwalli Noureldin

Supervisor: Prof. Salah Selim

Department of Computer Engineering & Systems

Faculty of Engineering at Alexandria University

July 2023

Contents

Abstract

A real-time sign language translator is an important milestone in facilitating communication between the deaf community and the general public. We hereby present an approach of an American Sign Language (ASL) translator based on a convolutional neural network.
We are going to utilize the pre-trained SSD model architecture for the real-time recognition of the ASL using the concept of transfer learning and to give a try to deal with the dynamic gestures. The main objectives of this project are to develop an accurate and efficient ASL recognition system, to improve communication between the deaf and hearing communities, and to promote inclusivity and accessibility for all. The system has been evaluated on a dataset of ASL gestures and has achieved an accuracy of over 90
In summary, the Real-time American Sign Language Recognition System is a promising solution for bridging the communication gap between the deaf and hearing communities. It provides an efficient, accurate, and accessible means of communicating in ASL, which can significantly improve the quality of life for individuals with hearing impairments. The system can also be extended to include additional features such as gesture recognition for other sign languages.

Acknowledgements

We would like to thank Prof. Salah Selim for his precious guidance, Dr. Marwan Torki for his effort with us in the computer vision course, and Dr. Andrew Ng for his excellent content in the computer vision networks that is available on the internet for everyone and that was one of the main resources.

1 Introducion

1.1 Motivation

Communication between humans is a form of life, as we do this every single day and we may need to communicate with people who do not speak the same language as ours, but despite the many languages, communication has become easier because of translators.
Now, what about the deaf and dumb people, couldn't it be easier to communicate with them..? As a way to help facilitate this problem, we decided to make a software that detects one of the sign languages (ASL - American sign language) and converts the detected signs captured using a real-live camera to its text.

American Sign Language (ASL) is a visual language used by the deaf and hard-of-hearing community in the United States. Despite the large number of individuals who use ASL as their primary mode of communication, there is still a significant communication gap between the deaf and hearing communities. This gap can lead to social isolation, limited access to education and healthcare, and limited job opportunities for individuals with hearing impairments.
Moreover, there is a shortage of qualified interpreters who can accurately translate ASL into spoken or written language. Therefore, there is a critical need for a real-time, accurate, and efficient ASL recognition system that can bridge the communication gap between the deaf and hearing communities.

The proposed Real-time American Sign Language Recognition System aims to address this need by providing an efficient and accurate means of translating ASL into text. By using deep learning techniques, the system can recognize ASL gestures in real-time, which can significantly improve communication between the deaf and hearing communities. The system can also be extended to include additional features such as gesture recognition for other sign languages.

1.2 Scope of Work

We are going to use the SSD architecture by applying transfer learning to one of it's versions over the data we're going to collect for +30 classes. Our software will detect ASL digits, alphabets and some other of its known words' gestures (Yes - No - Thanks - I Love You - Hello).

The scope of this project is to apply the transfer learning technique over one of the object detection models, and evaluate a Real-time American Sign Language Recognition System using deep learning techniques. The system will be able to recognize hand gestures corresponding to ASL signs and translate them into text in real-time.

The project will involve the following tasks:

1. `Literature review:` A comprehensive review of the existing literature on ASL recognition systems and deep learning techniques will be conducted to identify the state-of-the-art methods and best practices.
2. `Dataset collection:` A dataset of ASL gestures will be collected and annotated for training and testing the deep learning model.
3. `Model development:` A deep learning object detection model will be used to apply transfer learning to recognize ASL gestures in real-time. The model will be trained using the collected dataset and optimized for accuracy.
4. `System implementation:` The ASL recognition system will be implemented using the trained model which is integrated into a web application and a camera to capture the hand gestures.
5. `System evaluation:` The system will be evaluated on the collected dataset and on a set of real-world ASL gestures. The performance of the system will be measured in terms of mean average precision (mAP).

The project will be implemented using Python programming language and popular deep learning libraries such as TensorFlow. The system will be deployed on a computer with a webcam and a user-friendly interface will be developed for input and output of text.

1.3 Assumptions

We add dynamic letters of the American Sign Language. All the research works that we read about were excluding them because they were including motion, and we will try to deal with it by taking a frame for every possible pose of the motion of the letters which will increase their detection accuracy.

1.4 Tools to Be Used

- Flask framework.
- Offline Python IDE like Jupyter NoteBook.
- HTML, CSS, JavaScript, with VScode IDE.

2 Background and Related Work

2.1 Background

2.1.1 Machine Learning

Machine learning is a powerful tool for solving problems in computer vision. Supervised learning is the most common approach, where a model is trained on labeled data (input-output pairs) to learn a mapping from inputs to outputs. Unsupervised learning is used when there is no labeled data, and the goal is to learn the underlying structure of the data. Reinforcement learning is used when the goal is to learn a policy for making decisions in an environment based on feedback signals.

The K-Nearest Neighbors (KNN) algorithm is a popular method in machine learning for classification problems. It is based on the idea that objects with similar features tend to belong to the same class. KNN is a non-parametric algorithm, meaning it doesn't make any assumptions about the underlying distribution of the data. Instead, it uses the entire training dataset to make predictions.

In the context of gesture recognition, the KNN approach involves training the model on a dataset of labeled hand gestures. During inference, the system captures an image of a hand and crops it to focus on the hand region. The extracted image is then compared to the training set using a distance metric, and the K nearest neighbors are identified. The class label of the test image is then assigned based on the majority class of the K nearest neighbors.

One of the main advantages of the KNN approach is its simplicity. It doesn't require any training time, and new classes can be easily added to the system. However, one of the main limitations of the KNN approach is its dependence on the quality and size of the training dataset. Additionally, the KNN approach can be computationally expensive, especially when dealing with large datasets or high-dimensional feature spaces. Finally, the KNN approach can be sensitive to the choice of distance metric, and the optimal value of K may vary depending on the problem at hand.

2.1.2 Convolutional Neural Networks (CNNs)

CNNs are a type of neural network that are well-suited for image analysis tasks. They work by applying a series of convolutional filters to an input image, which extract features at different scales and orientations. These features are then passed through a series of pooling and fully connected layers to produce a final output. CNNs can be trained end-to-end on large datasets using backpropagation, which adjusts the weights of the filters to minimize a loss function.

2.1.3 Deep Learning

Deep learning is a subset of machine learning that is based on artificial neural networks with multiple layers. These neural networks are designed to learn representations of data that are increasingly complex as information flows through the layers. Deep learning has revolutionized many fields, including computer vision, natural language processing, and speech recognition.

Deep learning models are typically composed of multiple layers, each of which processes the input data in a different way. The first layer may perform simple operations such as edge detection, while later layers may learn to recognize more complex patterns and objects. The final layer of the network produces the output, which may be a classification label, a regression value, or a sequence of tokens.

Training a deep learning model involves adjusting the weights of the network to minimize a loss function, which measures the difference between the predicted output and the true output. This is typically done using backpropagation, which computes the gradients of the loss function with respect to the weights and updates them using an optimization algorithm such as stochastic gradient descent.

Convolutional Neural Networks (CNNs) are a type of deep learning architecture that are commonly used for computer vision tasks such as image classification, object detection, and segmentation. CNNs use convolutional layers to learn features from the input data, followed by pooling layers to reduce the dimensionality of the representations. Fully connected layers are then used to produce the final output.

Recurrent Neural Networks (RNNs) are another type of deep learning architecture that are commonly used for sequence modeling tasks

such as language modeling, speech recognition, and machine translation. RNNs use recurrent connections to capture dependencies between elements in a sequence, allowing them to model complex temporal patterns.

Generative models, such as Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs), are a type of deep learning model that are used for tasks such as image synthesis, text generation, and data augmentation. These models learn to generate new data samples that are similar to the training data, by learning a distribution over the input data and sampling from it.

Overall, deep learning has had a significant impact on many fields, and continues to be an active area of research with many exciting applications.

2.1.4 Object Detection

Object detection is the task of localizing and classifying objects in an image or video. CNNs are commonly used for this task, often in combination with region proposal algorithms that generate candidate object locations. These algorithms typically use selective search or region proposal networks (RPNs) to generate a set of candidate bounding boxes for each image. The CNN is then trained to classify the contents of each bounding box and refine its location.

Faster R-CNN, YOLO (You Only Look Once), and SSD (Single Shot Detector) are popular object detection models used in computer vision applications.

Faster R-CNN is a two-stage object detection model that uses a region proposal network (RPN) to generate object proposals and a Fast R-CNN network to classify the proposals. The RPN generates a set of object proposals by sliding a small window over the feature map and predicting objectness scores and bounding box offsets for each anchor box. The Fast R-CNN network takes the proposals as input and performs classification and bounding box regression on each proposal.

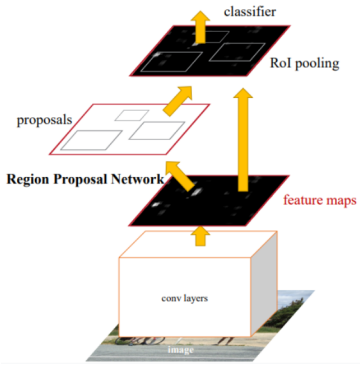

Figure 1: Faster R-CNN is a Two-stage Network, one stage for region proposal and the other one is for detection, Faster than the traditional CNN

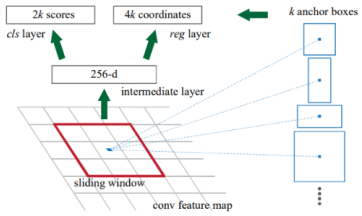

Figure 2: Region Proposal Network RPN

YOLO is a one-stage object detection model that predicts the bounding boxes and class probabilities in a single forward pass. The YOLO model divides the input image into a grid and predicts a fixed number of bounding boxes and class probabilities for each cell in the grid. The YOLO model is fast and can process images in real-time, but it may not perform as well as two-stage models like Faster R-CNN.

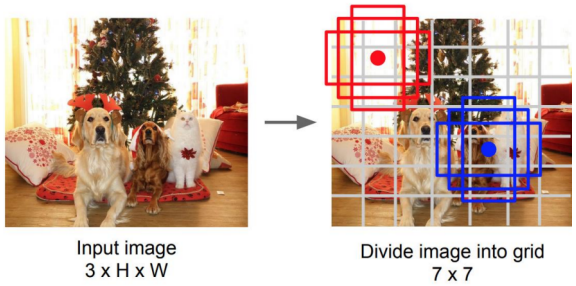

Figure 3: YOLO is a Single-stage Network, depends on the anchor boxes instead of the stage of the region proposal, Faster than the Faster R-CNN model

SSD is another one-stage object detection model that predicts the bounding boxes and class probabilities directly from the feature maps. Instead of using a predefined set of anchor boxes like Faster R-CNN, SSD generates a set of default bounding boxes at different scales and aspect ratios. The model then predicts the offsets and class probabilities for each default box. SSD is fast and accurate, making it a popular choice for real-time object detection applications.

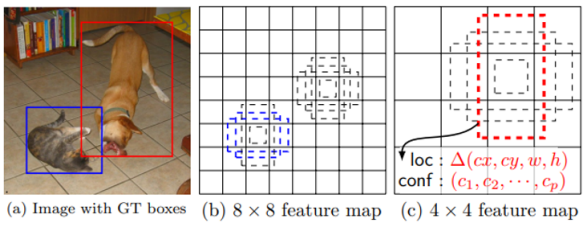

Figure 4: SSD is a Single-stage Network, depends on the anchor boxes instead of the stage of the region proposal, Faster than YOLO

Overall, each object detection model has its strengths and weaknesses, and the choice of model depends on the specific requirements of the application [7].

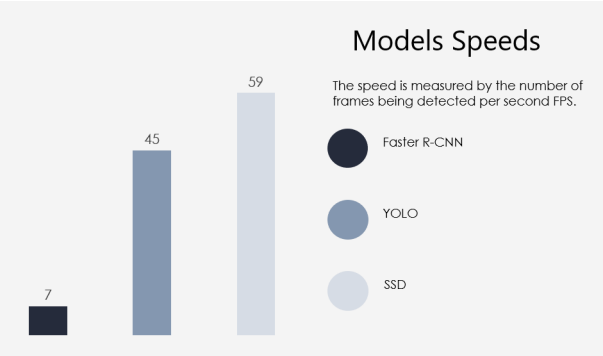

Figure 5: The speed is measured by the number of frames being detected per second FPS.

2.1.5 COCO Dataset vs. Pascal Datset

COCO (Common Objects in Context) and Pascal VOC (Visual Object Classes) [4] are two popular benchmark datasets for object detection and segmentation tasks in computer vision.

COCO is a more recent dataset and is larger than Pascal VOC. It contains over 330,000 images with more than 2.5 million object instances labeled across 80 different object categories. COCO is known for its challenging images with complex backgrounds and occlusions, making it a good benchmark for evaluating object detection and segmentation models' performance under challenging conditions.

Pascal VOC, on the other hand, is an older dataset and contains about 20,000 images labeled across 20 object categories. The dataset is known for its simpler images with less complex backgrounds and occlusions. Pascal VOC has been widely used as a benchmark for object detection and segmentation models and has been the focus of research for many years.

In terms of evaluation metrics, both datasets use the mean Average Precision (mAP) metric to evaluate the performance of object detection and segmentation models. However, the evaluation protocols differ slightly between the two datasets. COCO uses a stricter evaluation protocol that takes into account the overlap between predicted bounding boxes and ground truth boxes, while Pascal VOC uses a looser evaluation protocol that only requires a minimum overlap between predicted and ground truth boxes.

Overall, both COCO and Pascal VOC are widely used benchmark datasets for object detection and segmentation tasks, and the choice of dataset depends on the specific requirements of the application. COCO provides a more challenging dataset with more object categories, while Pascal VOC is simpler and more established.

2.1.6 Training

Training an object detection model involves selecting a suitable architecture, initializing its weights, and optimizing its parameters on a training dataset. Data augmentation techniques, such as random cropping, flipping, and scaling, are often used to increase the size of the training set and improve generalization. Hyper-parameter tuning is used to find the best values for parameters such as learning rate, batch size, and regularization strength.

The training process typically involves the following steps:

1. Data Preparation: Collecting and annotating a large dataset of images or videos is a critical step in training an object detection model. The dataset should include a diverse range of objects and backgrounds, along with accurate annotations for each object's location and class.
2. Model Selection: Choosing an appropriate object detection model architecture is important. There are many different architectures available, each with its strengths and weaknesses. Popular object detection models include Faster R-CNN, YOLO, and SSD.
3. Model Initialization: The model's parameters are usually initialized randomly before training. However, pre-trained models can also be used for transfer learning, which can speed up the training process and improve performance.
4. Training: During training, the model is presented with a batch of images, and the loss is calculated based on the difference between the predicted outputs and the ground truth annotations. The model's parameters are then updated using backpropagation to minimize the loss.
5. Fine-tuning: If the model's performance is not satisfactory, it can be fine-tuned by adjusting the hyper-parameters or by using different data augmentation techniques.

2.1.7 Evaluation

Evaluation involves measuring the accuracy, speed, and robustness of the model on a held-out test set. AP and IoU are commonly used metrics to evaluate object detection performance.

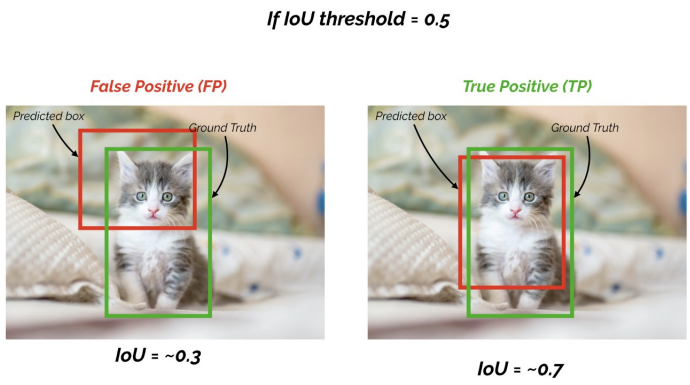

Figure 6: intersection over union (IoU) with 0.5 threshold to determine whether the detection is accepted or not

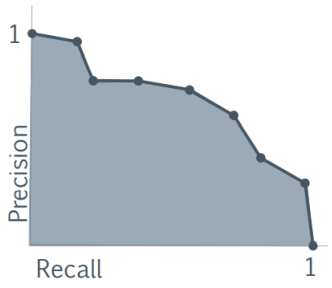

Figure 7: Precision - Recall curve, area under the curve presents the average precision (AP)

We are going to evaluate the model using the mean average precision (mAP), and the number of processed frames per second (FPS). The mean average precision depends on the IoU technique that calculates the area of intersection between the ground truth box and the matching box divided by the union area of both boxes, then apply a threshold over it to decide if it's an accepted match or not

to use these matches later to calculate the precision and the recall.

`Precision` is the number of correct matches over the total number of matches, while the `recall` is the number of the ground truth boxes with correct matches over the number of the ground truth boxes, so if we increased the threshold, we get higher recall and lower precision, and if we reduced it, we get lower recall and higher precision.
For each class the area under the precision-recall curve is called average precision AP, and mAP is the mean of the APs of the classes.

2.1.8 Transfer Learning

Transfer learning is a technique in machine learning where a pre-trained model is used as a starting point for a new task. In object detection, transfer learning can be used to speed up the training process and improve the performance of the model.

The basic idea behind transfer learning is that the pre-trained model has already learned useful features from a large dataset, and these features can be reused for a new task with a smaller dataset. For example, a pre-trained model trained on a large dataset like COCO or ImageNet can be used as a starting point for a new object detection task with a smaller dataset.

There are two main approaches to transfer learning in object detection:

1. Fine-tuning: In fine-tuning, the pre-trained model is used as a starting point, and its parameters are adjusted during training to adapt to the new task. The last few layers of the pre-trained model are typically replaced with new layers that are specific to the new task. The new layers are initialized randomly, and the rest of the model's parameters are initialized from the pre-trained model. Fine-tuning can speed up the training process and improve the performance of the model, especially when the new task is similar to the original task the pre-trained model was trained on.
2. Feature extraction: In feature extraction, the pre-trained model is used as a fixed feature extractor, and the new layers are added on top of the pre-trained model. The pre-trained model's parameters are frozen, and only the new layers are trained on

the new dataset. Feature extraction can be useful when the new task is different from the original task the pre-trained model was trained on.

2.2 Related Work

ASL recognition is not a new computer vision problem, it has been implemented using the traditional convolutional neural networks even with the models built and trained from scratch or the use of the pre-trained models by applying transfer learning over a smaller dataset.

2.2.1 Deep Learning and Machine Learning static ASL offline Recognition systems

A Project [5] for the same problem but not real-time implemented it using transfer learning for the pre-trained VGG-16 model, and used self-collected data but with no variety, data splits "training, validation and testing" are almost the same, the data is collected by only one person with no noise in the background or different lighting conditions, or different views for the same letter.
They cropped the section of images that include the hand depending on the contrast, which might fail in the case of having noise or non-flat regions in the image.
because of the huge similarity between the training and testing sets, they had good accuracies approximately 98%.For the same project they tried to use on of the machine learning (ML) techniques, k - Nearest Neighbors (KNN), and got almost the same accuracy but more time consumed for detection.

2.2.2 Deep Learning system using googleNet model and transfere learning

This paper [1] implemented the same problem in the real time using a website, they used the pre-trained GoogLeNet model, and used two different datasets each collected by different five persons with variety in the:

- skin color
- poses of the letter gesture
- lighting conditions and background

Cropping the section of images that include the hand is done using zero padding and random cropping so they reduce the probability of losing a pixel which is related to the hand.
They tested the model in different ways, tested over many sets of letters (a-y, a-k, ..) with different parameters, on the data of the fifth person from each dataset, as the model trained only on the first four persons' data from each, and has been tested at the real-time too, the resulted accuracies were 70% - 98%.

2.2.3 gestures offline Recognition systems using simple linear classifier

While linear classifiers are easy to work with because they are relatively simple models, they require sophisticated feature extraction and preprocessing methods to be successful. Singha and Das obtained accuracy of 96% on 10 classes for images of gestures of one hand using Karhunen-Loeve Transforms These translate and rotate the axes to establish a new coordinate system based on the variance of the data. This transformation is applied after using a skin filter, hand cropping and edge detection on the images. They use a linear classifier to distinguish between hand gestures including thumbs up, index finger pointing left and right, and numbers (no ASL). Sharma et al. use piece-wise classifiers (Support Vector Machines and k-Nearest Neighbors) to characterize each color channel after background subtraction and noise removal [1]. Their innovation comes from using a contour trace, which is an efficient representation of hand contours. They attain an accuracy of 62.3% using an SVM on the segmented color channel model.

2.2.4 Italian gestures offline Recognition systems using 3D gloves

The most relevant work to date is L. Pigou et al's application of CNN's to classify 20 Italian gestures from the ChaLearn 2014 Looking at People gesture spotting competition [1]. They use a Microsoft Kinect on fullbody images of people performing the gestures and achieve a cross-validation accuracy of 91.7%. As in the case with the aforementioned 3-D glove, the Kinect allows capture of depth features, which aids significantly in classifying ASL signs.

2.2.5 Our Approach

To develop a real-time recognition system, it is important to consider the time it takes to recognize objects or to classify a given image. Traditional methods such as using a convolutional neural network (CNN) or two-stage object detection models like Faster-RCNN may not be suitable due to the time-consuming nature of these techniques. Additionally, to make the system more user-friendly, it is desirable to display multiple detections at different scales within the same frame. Therefore, machine learning techniques like KNN may not be preferred because they require cropping the hand out of the frame, which can increase the classification time. Moreover, if there are multiple gestures in the same frame, it can be confusing to deal with.

There are always trade-offs, to get a time-efficient model you need to accept the lower accuracy it provides, to get a highly acurate model you need to accept the higher time spent in recognition or classification.
If you wanted to be effient in both timing and detection accuracy, this requires higher cost to get tools like the 3D gloves.
So we decided to use the SSD pretrained model from tensorflow model zoo that is trained over the COCO dataset and applay the transfere learning technique using the data that we collected our self on this model.

We chose to use the SSD MobileNet model which is pre-trained over the COCO data set because it is lighter than the basic SSD model, making it more suitable for real-time use on mobile devices. This decision was driven by our goal of creating a user-friendly system that requires minimal computational resources while still providing reliable performance.

3 Technical Approach

3.1 SSD Model Architecture

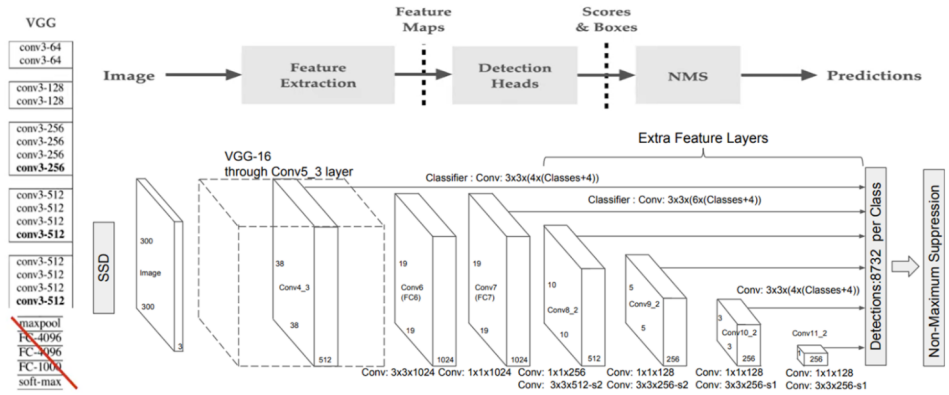

Figure 8: The basic SSD model architecture consists of a base convolutional neural network like VGG-16 which is truncated before the classification stage, followed by feature maps in different scales that decrease progressively in size, these steps are responsible for the feature extraction, followed by detection heads as small size conv. layers to either produce a score for a category or a shape offset relative to the default box coordinates, followed by a non-maximum suppression layer [4].

3.1.1 Base Network (Feature Extraction)

The VGG-16 network is used as a base, but other networks should also produce good results like ResNet, these earlier layers are based on a standard architecture used for high-quality image classification. The VGG is used without the final stage of preparing for classification to get the resulting output as a good starting point for the feature maps.

3.1.2 Multi-Scale Feature Maps for Detection (Feature Extraction)

Convolutional feature layers are added to the end of the truncated base network, they decrease in size progressively and allow predictions of detections at multiple scales, the convolutional model for predicting detections is different for each feature layer.

3.1.3 Convolutional Predictors for Detection (Detection Heads)

Each added feature layer (or optionally an existing feature layer from the base network) of size (m*n*p) can produce a fixed set of detection predictions using a set of convolutional filters of size (3*3*p) these small kernels produce either a score for a category or a shape offset relative to the default box coordinates.

3.1.4 Non-Maximum Suppression (NMS)

In this stage, the multiple detections for the same object are eliminated to one detection.

3.1.5 Default Boxes and Aspect Ratios

A set of default bounding boxes (k) is associated with each feature map cell, for multiple feature maps at the top of the network, at each feature map cell, predict the offsets relative to the default box shapes (4) in the cell, as well as the per-class scores (c) that indicate the presence of a class instance in each of those boxes $\rightarrow$ k(4 + c) for each cell in the different feature maps scales.
Combining predictions from multiple feature maps with different resolutions to naturally handle objects of various sizes and allowing different default box shapes in several feature maps lets us efficiently discretize the space of possible output box shapes.

3.2 Data

Figure 9: shows samples of the self-collected dataset especially for the traditional CNN model, with different lighting conditions and backgrounds, cropped only around the hand.

We have collected data that we were going to use if we implemented the traditional CNN, but as it's shown in [1], it's very slow and needs much processing for the input frames before detection in the real-time like the case of the faster R-CNN [3] that include a region proposal stage, which makes it more slowly, as we decided to use the architecture of the SSD model, so we collected the data in a different way to contain the whole person not only the hands and add the ground truth bounding boxes to it.
To get more data of the data we are going to collect, we will use the original image and its flipped version, as the hand gestures can be made with either the right or the left hand and this is a kind of the data augmentation technique to make the model more robust to these kind of transformations.
We added the dynamic letters(j, z) and try to deal with them by using different poses during the motion per each of them.

Figure 10: shows samples of the self-collected dataset especially for the SSD model transfer learning, this represents the word "No"

Figure 11: shows a sample of the data augmentation like the rotation

The similarity between the letters and digits is very high, so to make differences between them, we decided to enlarge the space between the fingers much more in the digits images.

3.3 Transfer Learning

Models are usually trained over a huge datasets, and the training phase consumes much time and resources. This technique allows to retrain a prtrained model on a smaller amount of data than the data it was pretrained on.
The knowledge transfer is done by firstly, freezing the weights (hyperparameters) of the model, all of them or some of the earlier ones

according to the size of the retraining data, secondly, the fully connected Layers in the original model is chopped off, and replaced with customized Fully-Connected Layers for the new dataset to deal with the new number of classes, finally any unfreezed parameters are going to be tuned during the training.
The primary benefits of such a technique are its less demanding time and data requirements.

3.4 Challenges and Approaches

The challenge in transfer learning comes from the differences between the original data used to train and the new data being classified, as the new data will be having high similarity between classes because they all are different poses for the same object in the old dataset which is the hand object.
we are going to try training only one model over the whole data (letters, numbers and some words) and try to get good detection results, if the results weren't good, we are going to train separate models to reduce the similarities by reducing the number of classes that the model needs to distinguish between.

4 Environment Setup

4.1 Programming Language and Libraries

The first step in setting up a deep learning environment is selecting a programming language and installing the necessary libraries. The most popular programming language for deep learning is Python that we decided to use, and some popular libraries include TensorFlow and others that will be installed properly with other libraries in the upcoming subsection.

We mainly used Python 3.9 and Tensorflow 2.5 .

4.2 IDEs and Editors

You may want to use an Integrated Development Environment (IDE) or editor to write and debug your code. Some popular options include PyCharm, Visual Studio Code, and Sublime Text.

- LabelImg: Open source python application that can be installed to help in data preparation to add annotations to the collected images[6].
- Anaconda: Enables you to build different virtual environments and it's easy to deal with.
- Visual Studio Code (VSCode): The easiest editor to develop the web application and review it using the live server extention.

4.3 Installation

Once the programming language and libraries have been selected, now we need to install them on our machine. This can typically be done using a package manager such as pip or conda. You may also need to install additional dependencies, such as CUDA and cuDNN, if you plan to use GPUs for training as described in the previous subsection.

4.3.1 LabelImg Python Application

LabelImg Application is open source Python Application and used to add labels and boxes which considered as the ground-truth to the collected dataset that facilitates generating annotations. To get it work Proberly follow these steps:

1. Make sure to have Anaconda installed in your machine, and during the installation check the box to add it to the PATH variable, and if you previously installed it you can add it manually, you can find the pathes you need to add using this command "`where conda`" and add them in the PATH variable in the system environment.
2. You need to download the labelImg application from the github repositry [6], this can be done using git clone command.
3. Open the terminal/CMD in the labelImg folder and excute the following commands:
   (a) `conda install pyqt=5`
   (b) `conda install -c anaconda lxml`
   (c) `pyrcc5 -o resources.py resources.qrc`
4. Then move these two files (resources.py, resources.qrc) into the libs folder.
5. Finally you can run the application using this command `python labelImg.py`

4.3.2 GPU Setup

Deep Learning Applications specially the model training phase is a highly resources consumer process, it consumes time, memory, power and CPU cycles in a very high manner, so it's better to get the benefits of your GPU card if you have a suitable one for this project. To get your GPU card working Properly follow these steps:

1. Windows Native - Windows 7 or higher (64-bit) (no GPU support after TF 2.10).
2. Windows WSL2 - Windows 10 19044 or higher (64-bit).
3. Python 3.8 - 3.11
4. Install Visual Studio community edition: We installed VS2019.
5. Install NVIDIA GPU driver, we installed the following:
   (a) `CUDA Toolkit 11.2`
   (b) `cuDNN 8.1.0`
   You need to sign up for Nvidia developer program (free) Extract all folder contents from cudnn you just downloaded to C:/program files/Nvidia GPU computing toolkit/CUDA/v11.0.

6. Install your IDE if you haven't already done so, you may use spyder, jupyter notebook or VSCode.
7. Create conda environment with python = 3.9
   `conda create --name tf python=3.9`
8. Activate the environment in the terminal
   `conda activate tf`
9. Excute the following command:
   `conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0.`
10. Install TensorFlow
    ```
    pip install --upgrade pip
    pip install tensorflow==2.5
    ```
11. Verify the installation with CPU
    ```
    python -c "import tensorflow as tf;
    print(tf.reduce_sum(tf.random.normal([$1000, 1000$])))
    ```
    If a tensor is returned, you've installed TensorFlow successfully.
12. Verify the installation with GPU
    ```
    python -c "import tensorflow as tf;
    print(tf.config.list_physical_devices('GPU'))
    ```
    If a list of GPU devices is returned, you've installed TensorFlow successfully.
13. Or create a new python file and run these lines to test if GPU is recognized by tensorflow.
    ```
    import tensorflow as tf
    tf.test.is_gpu_available(
    cuda_only=False, min_cuda_compute_capability=None
    )
    ```

4.3.3 Anaconda Environment with Tensorflow 2 Object Detection API Setup

We're going to use the SSD MobNet pre-trained object detection model from Tensorflow Model Zoo, so we need basically to setup the our environment to work well with tensorflow object detection API, This part has many compatibility issues, so it's preferred to use Anaconda or Docker in case if you wanted to start again from scratch. Carefully follow these steps:

1. If You skipped the GPU setup part, do the following:
   (a) Download and install Anaconda.

(b) Create new environment with python = 3.9 and activate it
```
conda create --name tf python=3.9
conda activate tf
```

(c) Install Jupyter Notebook and cmd in this environment.

(d) Install TF = 2.5.0
```
pip install --ignore-installed --upgrade tensorflow==2.5.0
```

(e) Verify Installation
```
python -c "import tensorflow as tf;
print(tf.reduce_sum(tf.random.normal([$1000, 1000$])))
```
If a tensor is returned, you've installed TensorFlow successfully.

(f) Download CUDA and cuDNN as described earlier and verify the installation if you wanted to use the GPU.

2. Downloading the TensorFlow Model Garden [2]
either using the `git clone` command or download the repository from GitHub "the Model Garden for TensorFlow" and extract it in a folder named "Tensorflow".

3. Protobuf Installation/Compilation

(a) Download the latest protoc-*-*.zip release and extract it in this "C:/Program Files/Google Protobuf" directory.

(b) Add "C:/Program Files/Google Protobuf/bin" to your Path environment variable.

(c) Open a new CMD from the anaconda environment and go to this path "../Tensorflow/models/research" and run this command:
```
protoc object_detection/protos/*.proto --python_out=.
```

4. COCO API Installation:
As of TensorFlow 2.x, the pycocotools package is listed as a dependency of the Object Detection API. Ideally, this package should get installed when installing the Object Detection API.

(a) Visual C++ 2015 build tools must be installed and on your path.

(b) Open a new CMD and execute the following two commands:
```
pip install cython
pip install git+https://github.com/philferriere/cocoapi.
git#subdirectory=PythonAPI
```

5. Install the Object Detection API: Once you have cloned the TensorFlow models repository, you need to install the Object Detection API. You can do this by running the following command from the TensorFlow models directory:
   (a) Copy the file `setup.py` from "../Tensorflow/models/research/object_detection/packages/tf2" to "../Tensorflow/models/research"
   (b) Install the dependencies in the setup.py by running the following command in a new terminal in the research folder.
   ```
   python -m pip install .
   pip install protobuf==3.20
   ```
6. Test your Installation: To test that the installation was successful, run the following command from a new CMD in the "../Tensorflow/models/research" directory.

   ```
   python object_detection/builders/model_builder_tf2_test.py
   ```

   If the installation was successful, you should see a list of test results.
7. Now we need to edit some file in the installed pycocotools.
   (a) Go to the Folder of pycocotools installed in the previously created environment in anaconda (for example, "C:/users/USERNAME/anaconda3/envs/tf/libs/site-packages/pycocotools").
   (b) Open the file named `cocoeval.py`.
   (c) replace the lines 378,379 with the following:
   ```
   tp_sum = np.cumsum(tps, axis=1).astype(dtype=float)
   fp_sum = np.cumsum(fps, axis=1).astype(dtype=float)
   ```

   (d) replace the lines 506, 507 with the following:
   ```
   self.iouThrs = np.linspace(.5, 0.95, 10, endpoint=True)
   self.recThrs = np.linspace(.0, 1.00, 101, endpoint=True)
   ```

   (e) replace the lines 517, 518 with the following:
   ```
   self.iouThrs = np.linspace(.5, 0.95, 10, endpoint=True)
   self.recThrs = np.linspace(.0, 1.00, 101, endpoint=True)
   ```

4.3.4 VSCode Setup with Flask to Run The Web Application

Simply you need to install the requirement in the requirements.txt file in our repository in GitHub (not in the environment to avoid con-

flicts with flask dependencies and others) using this command `pip install -r requirements.txt` and then run the following command to run the flask project in VSCode `python app.py` and open the localhost to review the web application.

If you faced any problems with the "object_detection" module, use the `setup.py` file by running this command `python -m pip install .`, or take the folder named "object_detection" in the same directory where requirements.txt and setup.py file exists, and copy it to the site-packages of your python directory.

4.4 Data preparation

Now we need to prepare our collected images of the 40 classes, firstly, use the LabelImg application to easily draw a box around each hand gesture in the image and give this box the suitable label of the class represented by the gesture, then split the whole data for the training and testing.

- Training: 90%
- Testing: 10%

4.5 Hardware and Infrastructure

Deep learning models can require a lot of computational resources, particularly when training on large datasets. You may need to use specialized hardware, such as GPUs or TPUs, to speed up training. You may also want to use cloud services such as Amazon Web Services or Google Cloud Platform to access more powerful hardware and to easily scale up or down your infrastructure as needed. We were going to use Google Colab to avoid the GPU setup, but it puts limitations on the provided resources which may lead to lose all the efforts spent in the training if these limits are exceeded.

The following GPU-enabled devices are supported:
NVIDIA GPU card with CUDA architectures:

- 3.5
- 5.0
- 6.0
- 7.0 or 7.5
- 8.0 and higher

5 Implementation Details

5.1 Project Organization

5.1.1 data_collector.ipynb

The objective of this code is to collect a set of labeled images for training an object detection model that can detect hand gestures. Specifically, the code captures a series of images from a video camera for each label in a predefined set of labels, where each label corresponds to a hand gesture.

The code uses the following methodology to collect the labeled images:

1. First, the code defines a variable IMAGES_PATH that specifies the directory where the labeled images will be stored. It also defines a list of labels, where each label corresponds to a hand gesture.
2. For each label in the list of labels, the code creates a subdirectory in the IMAGES_PATH directory with the same name as the label.
3. The code then creates a VideoCapture object to capture frames from the video camera.
4. For each label, the code enters a loop that captures a series of images from the video camera. Specifically, the loop captures number_imgs images for each label, where number_imgs is a predefined constant.
5. In each iteration of the loop, the code captures a frame from the video camera using the read method of the VideoCapture object.
6. The code then saves the captured frame as an image in the subdirectory corresponding to the current label. The name of the image file is constructed by concatenating the label name, the image number, and the file extension .jpg.
7. The code displays the captured frame in a window using the imshow function of the cv2 module.
8. The code waits for 2 seconds before capturing the next image to allow the user to adjust their hand gesture.

9. If the user presses the 'q' key during the loop, the loop is terminated and the program exits.

`Results:` The output of the code is a set of labeled images stored in subdirectories of the IMAGES_PATH directory. Each subdirectory corresponds to a label, and contains number_imgs images of hand gestures corresponding to that label.

5.2 Images Labeling with LabelImg

Run the LabelTmg application as described in the installations section, then open each image and draw a box exactly around the hand gesture, then give this box the suiable label of the class which will produce an xml file with the same name as the labeld image and describes the boxes in the image with the labels given to them.

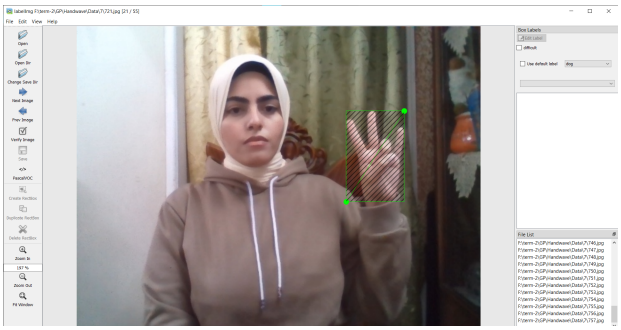

Figure 12: shows an image with the boundary box that is considered as a ground-truth, labeled using the LabelImg python application

5.2.1 main.ipynb

The objective of this notebook is to prepare the previously collected images and get them in the suuitable format to the object detection model using TensorFlow's Object Detection API, This API provides a collection of pre-trained models for object detection, but it can also be used to train custom models using user-provided data, to apply the transfer learning on it after updating its pipeline configurations file according to our needs in the system.

1. `Resources Used:` This provides the link of the tensorflow 2 object detection API tutorial for the installation to be done properly.

2. `Setup Paths:` The code first sets up the paths for various directories and files required for the object detection pipeline. The workspace path is set to 'Tensorflow/workspace', which is the root directory for the project. The scripts path is set to 'Tensorflow/scripts', which contains scripts used for generating TFRecord files, evaluating models, and other tasks. The API model path is set to 'Tensorflow/models', which contains TensorFlow's Object Detection API. The annotation path is set to 'Tensorflow/workspace/annotations', which is the directory where the annotations for the dataset will be stored. The image path is set to 'Tensorflow/workspace/images', which is the directory where the images for the dataset will be stored. The model path is set to 'Tensorflow/workspace/models', which is the directory where the trained model will be saved. The pre-trained model path is set to 'Tensorflow/workspace/pre-trained-models', where pre-trained models can be stored if needed. The configuration path is set to 'Tensorflow/workspace/models/my_ssd_mobnet/pipeline.config', which is the configuration file for the model.

3. `Creating The Label Map:` Next, a list of labels is defined. Each label is represented by a dictionary containing its name and ID. The label names include letters from A to Z, some common words such as 'yes', 'no', 'thanks', and 'hello', and digits from 1 to 9. The label IDs are assigned in increasing order from 1 to 40.
   Then, a label map file is created in the annotation path. This file contains a mapping between the label names and their corresponding IDs in the form of a protocol buffer text (pbtxt) file. The 'with' statement is used to open the file, and then each label is written to the file by iterating over the labels list and writing its name and ID in the required format. This label map file will be used later in the pipeline to map the label names to their corresponding IDs.

4. `Helpers To Merge The Data:` Merging the data collected by both of us, is't straight forward after the labeling process, because in the annotation file the image file associated with it is written inside, so we needed to add some prefix to the files names before merging them.

   First, a list of labels is defined, which includes the names of the different classes that the images belong to. The variable

'dst_path' is set to the destination directory where all the images will be copied to.
The 'main' function is defined, which takes a path as input and iterates over all the files in that directory. For each file, it renames the file by adding the prefix "Aya_" to the original filename using the 'os.rename' function.
Next, a loop is defined over the labels, and for each label, it copies all the image files from the corresponding directory in the source path to the destination directory using the 'shutil.copy' function.
Then, another loop is defined over the labels, and for each label, it iterates over all the XML files in the corresponding directory in the source path. For each XML file, it loads the file using the 'ET.parse' function, gets the root element of the XML tree using the 'tree.getroot' function, finds the 'filename' element using the 'root.find' function, replaces the file extension with ".jpg" using the 'replace' function, and saves the modified XML file using the 'tree.write' function.

Overall, this code helps to prepare the data for training an object detection model by merging the images and annotations from different directories into a single directory. The image files are renamed, and the XML annotation files are modified to match the new filenames.

5. `Split the merged data for training 90% and testing 10% manually.`

6. `Create TF Records:` Two TFRecord files are generated for the training and test datasets using the 'generate_tfrecord.py' script from the scripts path. TFRecord is a format used by TensorFlow to store and read data efficiently. The script takes the path to the image directory, label map file, and output file path as inputs. The script reads the annotations and images from the image directory and label map file, respectively, and creates a TFRecord file for the dataset. The 'generate_tfrecord.py' script is called twice, once for the training dataset and once for the test dataset. The '-x' flag is used to specify the path to the image directory, the '-l' flag is used to specify the path to the label map file, and the '-o' flag is used to specify the output file path. The output files are saved in the annotation path with the names 'train.record' and 'test.record' for the training and test datasets, respectively.

The generated TFRecord files are saved in the annotation path. These files will be used as inputs to train the object detection model.

7. `Download TF Models Pretrained Models from Tensorflow Model Zoo:` This code downloads a pre-trained object detection model from the TensorFlow Model Zoo and extracts its files to a specified directory using the 'wget' and 'tar' commands.
   The TensorFlow Model Zoo contains a collection of pre-trained models for various computer vision tasks, including object detection. These pre-trained models can be used as a starting point for training custom object detection models on new datasets, saving significant time and effort.
   The 'wget' command is used to download the pre-trained model from the specified URL and save it to the current working directory. The 'move' command is then used to move the downloaded model file to the 'PRETRAINED_MODEL_PATH' directory, which is a user-defined variable that specifies the path to the directory where pre-trained model files will be stored.
   The 'tar' command is used to extract the model files from the downloaded archive to the 'PRETRAINED_MODEL_PATH' directory. The 'tar' command is a Unix utility that is used to create and manipulate archive files in the tar format. In this case, the 'zxvf' options are used to decompress the archive, extract the files, and display the progress of the extraction process.

8. `Copy Model Config to Training Folder:` create a new directory for a custom object detection model and copy the configuration file from a pre-trained model to the new directory using Windows commands.
   This part of the code defines a variable 'CUSTOM_MODEL_NAME' to a string value representing the name of the custom model to be created.
   Then Creates a new directory for the custom model with the name specified in the 'CUSTOM_MODEL_NAME' variable. This is done using the 'mkdir' command in Windows, which creates a new directory with the specified name.
   After that copies the configuration file of a pre-trained model to the newly created directory for the custom model. The 'copy' command is used to copy the file from the source directory to the destination directory. The source directory is the path to the pre-trained model's configuration file, and the destination

directory is the path to the directory created in the previous step with the name specified in the 'CUSTOM_MODEL_NAME' variable.

9. `Update Config For Transfer Learning:` This code updates the configuration file for a custom object detection model by modifying various parameters and saving the updated configuration to the same file.
   This firstly imports the necessary packages from the TensorFlow Object Detection API and the Google Protobuf library.

   Then defines a variable 'CONFIG_PATH' to the path of the configuration file for the custom model.

   Next uses the 'config_util' function from the Object Detection API to load the configurations from the pipeline file specified in the 'CONFIG_PATH' variable.

   After that creates a new pipeline configuration object and reads the configuration file specified in the 'CONFIG_PATH' variable using the 'tf.io.gfile.GFile' function.

   And update various parameters of the pipeline configuration to customize it for the specific object detection task. For example, the number of classes is set to 40, the batch size is set to 5, and the paths to the label map and input data files are updated, then converts the updated pipeline configuration object to a text format using the 'text_format.MessageToString' function.

   Finally save the updated configuration to the same file specified in the 'CONFIG_PATH' variable using the 'tf.io.gfile.GFile' function.

   Overall, this code updates various parameters of the pipeline configuration for a custom object detection model and saves the updated configuration to the same file. This updated configuration file is then used to configure and train the custom model on a new dataset.

10. `Train The Model:` This code runs the TensorFlow Object Detection API's 'model_main_tf2.py' script to train a custom object detection model.
    It defines a 'command' variable that specifies the command to be executed. This command executes the 'model_main_tf2.py' script with the following arguments and then excute using '!':

    - The path to the TensorFlow Object Detection API's 'model_main_tf2.py' script is specified using the 'APIMODEL_PATH' variable.

- The directory where the model checkpoints and other outputs will be saved is specified using the 'MODEL_PATH' and 'CUSTOM_MODEL_NAME' variables.
- The path to the pipeline configuration file for the model is specified using the 'MODEL_PATH' and 'CUSTOM_MODEL_NAME' variables.
- The total number of training steps is set to 50,000 using the '–num_train_steps' argument.

11. `Evaluate The Model:` This code runs the TensorFlow Object Detection API's 'model_main_tf2.py' script to evaluate a trained custom object detection model.
    It defines a 'command' variable that specifies the command to be executed. This command executes the 'model_main_tf2.py' script with the following arguments and then excute using '!':

    - The path to the TensorFlow Object Detection API's 'model_main_tf2.py' script is specified using the 'APIMODEL_PATH' variable.
    - The directory where the model checkpoints and other outputs will be saved is specified using the 'MODEL_PATH' and 'CUSTOM_MODEL_NAME' variables.
    - The path to the pipeline configuration file for the model is specified using the 'MODEL_PATH' and 'CUSTOM_MODEL_NAME' variables.
    - The directory where the model checkpoints are saved during training is specified using the 'MODEL_PATH' and 'CUSTOM_MODEL_NAME' variables with the 'checkpoint_dir' argument.

    For the training and evaluation steps, they results some files in the (model_directory/eval) and (model_directory/train) that can be later used to visualize the analysis graphs and results like the training loss and the mAP metric, to do that use this command in those folders:
    `tensorboard --logdir=..`

12. `Load Train Model From Checkpoint:` This code defines a function to perform object detection using a trained custom object detection model.
    Firstly load the pipeline configuration and build a detection model using the 'get_configs_from_pipeline_file' and 'model_builder.build' functions from the TensorFlow Object Detection API. The 'is_training' parameter is set to False to indicate that the model is not being

trained.
Then restores the checkpoint of the trained model using the 'tf.compat.v2.train.Checkpoint' function. The 'ckpt.restore' method is used to restore the checkpoint file located at the specified path.
Then define a 'detect_fn' function that takes an input image and performs object detection using the trained model. The function first pre-processes the input image using the 'detection_model.preprocess' method. Then, the 'detection_model.predict' method is used to generate predictions for the pre-processed image. The 'detection_model.postprocess' method is then used to post-process the predictions and generate the final detections. The function is defined with the '@tf.function' decorator, which indicates that the function should be compiled into a TensorFlow graph for improved performance.

13. `Detect in Real-Time:` This code performs real-time object detection using the trained custom object detection model and OpenCV to capture and display the video feed and using the detect_fn to prepare the detections to be then displayed on the frames captured.

5.2.2 Handwave_Flask

This is the Flask web application, the friendly-user interface, with the trained model checkpoint integrated into it to allow detection through the website.

The main files in this flask application are the following:

- index.html: The welcome page considered as a starter.
- home.html: The model page where you can start the detection process.
- app.py: The python script that controls the responses to the requests coming from the front-end.

You can easily run this using `python app.py` command.

6 Analysis and Results

6.1 Training Analysis and Evaluation

Tensorboard Visualization was extremely helpful to analyse the training and evaluate the resulted models using the visualization of metrics like training loss and mAP.
We trained the model many times on different classes(40 classes include dynamic and static, 38 classes include static gestures only) with increasing the number of images per class step by step to avoid over-fitting.

6.1.1 40 Class 10 Images per Class – 15000 Step – Batch Size 10

Not enough data per class for the training the resulted total loss is almost 20%, so increase the number of images per class.

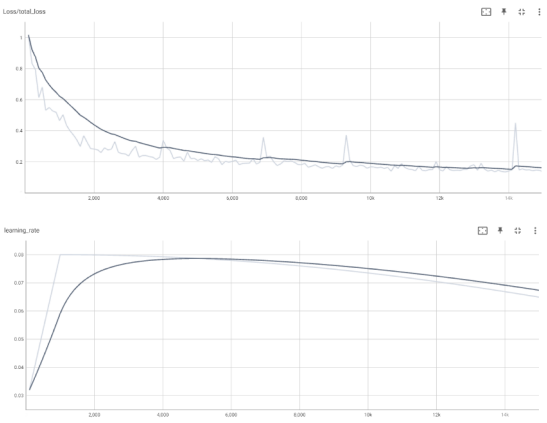

Figure 13: 40 Class 10 Images per Class – 15000 Step – Batch Size 10, shows the total training loss and the learning rate

6.1.2 40 Class 100 Images per Class – 10000 Step – Batch Size 5

Much higher total loss almost 40%, didn't converge and needs more steps to converge.

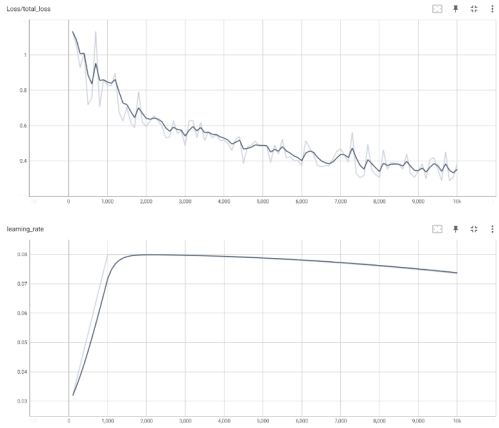

Figure 14: , shows the total training loss and the learning rate

6.1.3 40 Class 50 Images per Class – 15000 Step – Batch Size 5

Better total loss but still didn't converge and need much more steps, so increase both steps and the number of images per class.

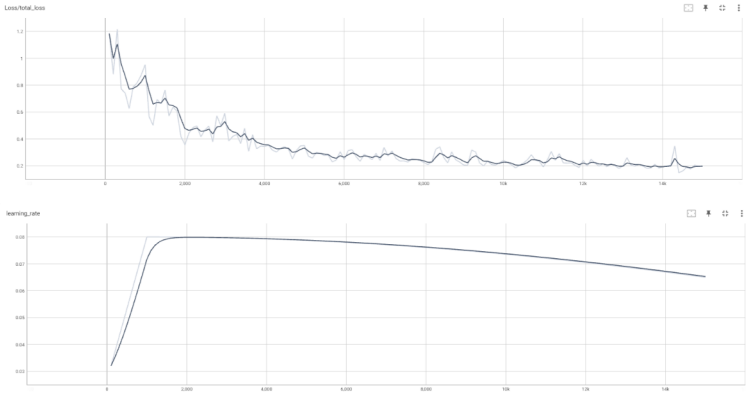

Figure 15: , shows the total training loss and the learning rate

6.1.4 40 Class 100 Images per Class – 50000 Step – Batch Size 5

Resulted a well-performing model with static gestures and mAP over 90%.

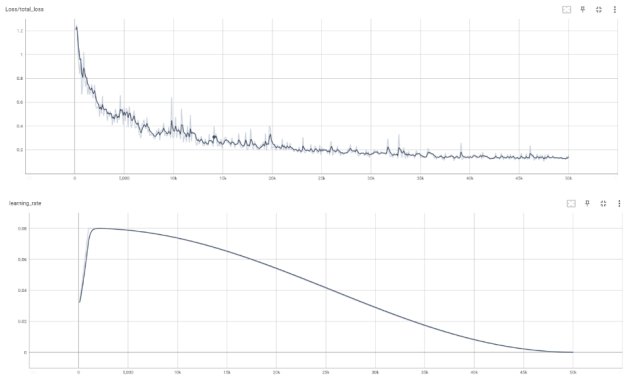

Figure 16: 40 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the total training loss and the learning rate

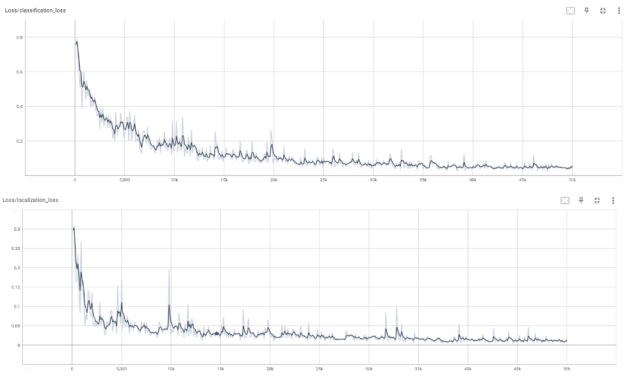

Figure 17: 40 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the classification loss and the localization loss

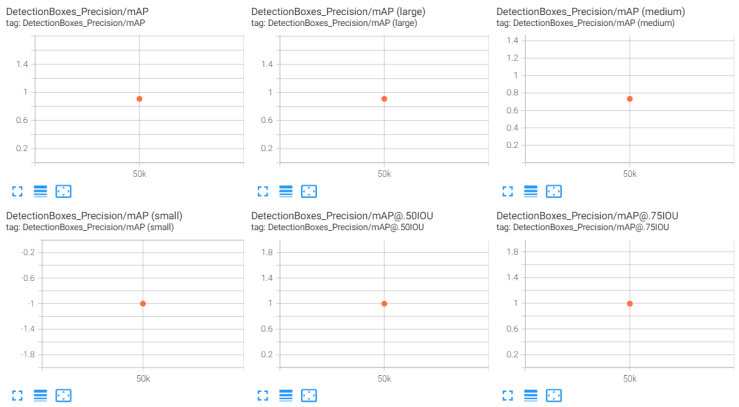

Figure 18: 40 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the evaluation metrics precision/mAP

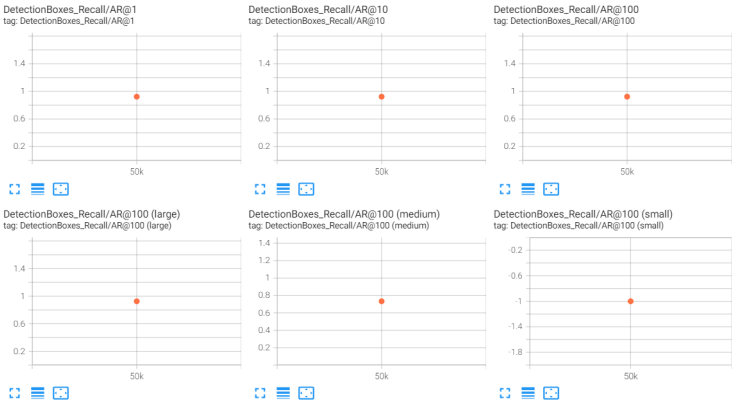

Figure 19: 40 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the evaluation metrics recall/mAP

6.1.5 38 Class 100 Images per Class – 50000 Step – Batch Size 5

The removal of the dynamic letters produced higher mAP up to 98%.

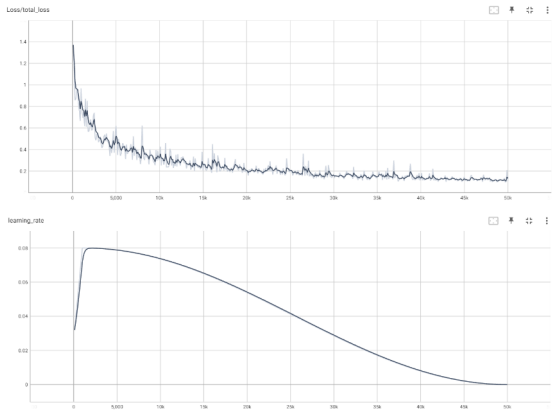

Figure 20: 38 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the total training loss and the learning rate

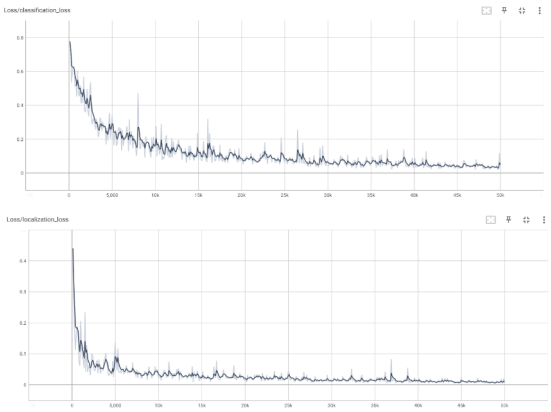

Figure 21: 38 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the classification loss and the localization loss

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.896
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.986
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.986
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.767
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.896
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.922
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.922
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.922
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.767
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.923
```

Figure 22: 38 Class 100 Images per Class – 50000 Step – Batch Size 5, shows the evaluation metrics precision/mAP and recall/mAP

6.2 Results

Dynamic gestures involve a continuous movement of the body or limbs, whereas static gestures involve holding a particular pose or position. Detection and recognition of dynamic gestures are generally more challenging than static gestures because they involve analyzing a sequence of frames rather than a single image.
When dealing with dynamic gestures, it is necessary to consider the temporal aspect of the motion. This means that the system needs to capture and analyze a series of frames to understand the motion and identify the gesture. This requires more computational resources and can be more difficult to implement compared to the detection of static gestures, which only require the analysis of a single image.
Furthermore, dynamic gestures are prone to variations in speed, scale, and orientation, which can make it more challenging to identify and classify them accurately. These variations can also make it more difficult to develop accurate and robust gesture recognition models.
In addition, dynamic gestures are more context-dependent than static gestures. The meaning of a dynamic gesture can change depending on the context in which it is used. For example, a wave can be a simple greeting or a signal for help, depending on the situation. Therefore, it is essential to consider the context in which the gesture is performed when developing a dynamic gesture recognition system.
The ASL has many similarities between its gestures which increases the model confusion.
The model trained over the 40 classes, that includes the dynamic letters, resulted garbage detection boxes for these dynamic letters.
To reduce the confusion, we trained the model only over the static gestures "38 class not the 40 class" which reduced the garbage detections, and the resulted mAP was over 95%

6.3 Model Limitations

- SSD model is not that good in detecting the small objets.
- The model doesn't deal well with the dynamic letters.

7 Conclusion and Future Work

7.1 Conclusion

The ASL detection and recognition project presented in this study is a real-time web system that aims to detect and recognize American Sign Language (ASL) gestures from captured frames and display the corresponding text on the captured frame in a web page. The system utilizes computer vision techniques and transfer learning to accurately identify and classify ASL gestures, with a mean Average Precision score between 90% - 98%.

The system is designed to help bridge the communication gap between deaf and hearing individuals by providing an efficient and effective means of capturing and interpreting ASL gestures. The use of self-collected data, carefully designed and prepared, enables the system to tailor the dataset to the specific characteristics and variations of ASL gestures, which contributes to the high level of accuracy achieved.

The project is based on the SSD MobileNet v2 model from the Tensorflow model zoo, which is fine-tuned on a dataset of ASL gestures. The system uses a combination of feature extraction techniques and deep learning algorithms to accurately detect and recognize ASL gestures in real-time. The recognized gestures are then converted into corresponding text and displayed, allowing both deaf and hearing individuals to communicate effectively.

The success of the project is due to the careful evaluation and adaptation of the pre-trained models to the specific requirements and characteristics of ASL recognition. The use of transfer learning and self-collected data enables the system to effectively capture and interpret ASL gestures, which has the potential to significantly improve the communication and accessibility of deaf and hard-of-hearing individuals in various settings.

Overall, the ASL detection and recognition project is a significant accomplishment that demonstrates the effectiveness of computer vision techniques and transfer learning in detecting and recognizing ASL gestures. The system has the potential to improve the accessibility and communication of deaf and hard-of-hearing individuals in a variety of settings, including education, healthcare, and social interactions.

7.2 Future Work

7.2.1 Motion Tracking to Deal with Dynamic Gestures

Dealing with the dynamic letters to work efficiently would be great. The detection of dynamic gestures is more challenging than the detection of static gestures due to the temporal aspect of the motion, variations in speed, scale, and orientation, and the context-dependent nature of dynamic gestures. These challenges require more sophisticated algorithms and techniques to accurately identify and classify dynamic gestures. Using videos data sets will be useful.

7.2.2 Additional Languages

Additional work to be done is to add more languages, not only the American language, by retraining the model over the new data with our data too which is known as (transfer learning), or to make a model for each language to get better accuracies as it reduces the similarities between the classes, or to improve our model (architecture and parameters) to get better detection accuracy.

7.2.3 Text Prediction and Concatenation

Adding algorithms that takes the continuous detections of letters and digits to results a meaningful text to have a complete translator for the ASL that may be widely used.

7.2.4 Both Sides Translator

Not only translating the ASL into normal English text, but also translating the plain English text into ASL, with limitations on the characters in that text and a good visualization of the gestures to be smooth and clear not only correct.

Appendix

Models Architectures Comparison

	Faster R-CNN	YOLOv3	SSD
Phases	RPN + Fast R-CNN detector	Concurrent bounding-box	regression and classification
Neural Network Type	Fully convolutional	Fully convolutional	Fully convolutional
Backbone Feature Extractor	VGG-16 or other feature extractors	Darknet-53 (53 convolutional layers)	VGG-16 or other feature extractors
Location Detection	Anchor-based	Anchor-Based	Prior boxes/Default boxes
Anchor Box	9 default boxes with different scales and aspect ratios	K-means from coco and VOC, 9 anchors boxes with different sizes	A fixed number of bounding boxes with different scales and aspect ratios in each feature map
IOU Thresholds	Two (at 0.3 and 0.7)	One (at 0.5)	One (at 0.5)
Loss Function	Softmax loss for classification; Smooth L1 for regression	Binary cross-entropy loss	Softmax loss for confidence; Smooth L1 Loss for localization

ASL Alphabets

SSD Objective Function

The overall objective loss function is a weighted sum of the confidence loss L_{conf} and the localization loss L_{loc}:

$$L(x,\ c,\ l,\ g)\ =\ \tfrac{1}{N}(L_{\text{conf}}(x,\ c)\ +\alpha L_{\text{loc}}(x,\ l,\ g))$$

where N is the number of matched default boxes, α is set to 1 by cross validation.
The confidence loss is the softmax loss over multiple classes' confidences(c).

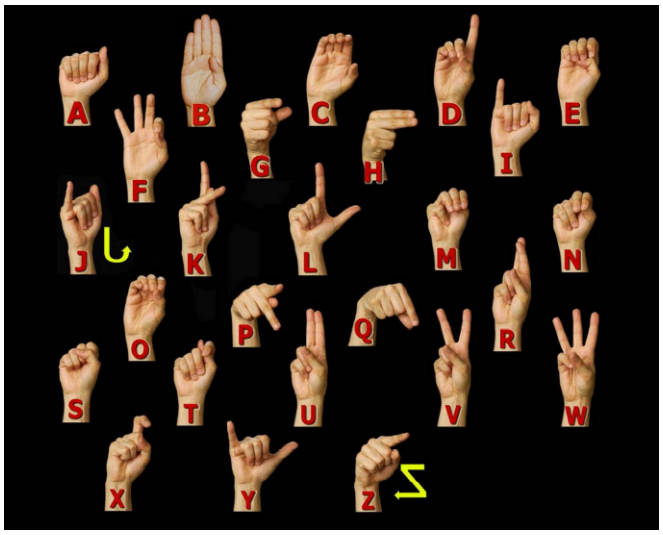

Figure 23: American sign language fingerspelling alphabet.

$$L_{\text{conf}}(x,\ c) \ = \ -x_{\text{ij}}^{p} log(\hat{C}_{i}^{p}) - log(\hat{C}_{i}^{0})$$

$$\hat{C}_{i}^{p} \ = \ \frac{e^{c_i^p}}{e^{c_i^p}}$$

Where $x_{\text{ij}}^{p} = \{0,\ 1\}$ is an indicator for matching the i-th default box to the j-th ground truth box of category p.

Localization loss is a Smooth L1 loss between the predicted box (l) and the ground truth box (g) parameters, we regress to offsets for the center (cx, cy) of the default bounding box (d) and for its width (w) and height (h).

$$L_{\text{loc}}(x,\ l,\ g) \ = \ (\ x_{\text{ij}}^{k}\ \text{smooth}_{L1}(l_{i}^{m} \ - \ \hat{g}_{i}^{m}))$$
$$\hat{g}_{j}^{\text{cx}} \ = \ \frac{g_{j}^{\text{cx}} \ - \ d_{i}^{\text{cx}}}{d_{i}^{w}}\ ,\ \hat{g}_{j}^{\text{cy}} \ = \ \frac{g_{j}^{\text{cy}} \ - \ d_{i}^{\text{cy}}}{d_{i}^{h}}\ ,\ \hat{g}_{j}^{w} \ = \ log(\frac{g_{j}^{w}}{d_{i}^{w}}),\ \hat{g}_{j}^{h} \ = \ log(\frac{g_{j}^{h}}{d_{i}^{h}})$$

List of Figures

List of Abbreviations

American Sign Language ..... ASL

Single Shot Detector ..... SSD

You Only Look Once ..... YOLO

Visual Geometry Group ..... VGG

Region-based Convolutional Neural Network ..... R-CNN

Recurrent Neural Networks ..... RNN

Generative Adversarial Networks ..... GANs

Variational Autoencoders ..... VAEs

Graphics Processing Unit ..... GPU

Central Processing Unit ..... CPU

Tensor Processing Unit ..... TPU

Convolutional Neural Network ..... CNN

Common Objects in Context ..... COCO

Visual Object Classes ..... VOC

Intersection over Union ..... IoU

Integrated Development Environment ..... IDE

Application Programming Interface ..... API

Machine Learning ..... ML

Region Proposal Network ..... RPN

K - Nearest Neighbor ..... KNN

Average Precision ..... AP

mean Average Precision ..... mAP

Frames Per Second ..... FPS

Non-Maximum Suppression ..... NMS

Visual Studio Code ..... VSCode

Compute Unified Device Architecture ..... CUDA

CUDA Deep Neural Network library ..... cuDNN

Command Prompt ..... CMD

Tensorflow ..... TF

References

[1] Brandon Garcia and Sigberto Alarcon Viesca. Real-time american sign language recognition with convolutional neural networks. *Convolutional Neural Networks for Visual Recognition*, 2(225-232):8, 2016.

[2] Mark Daoust Chen Chen Vishnu Banna Hongkun Yu, Neal Wu. tensorflow/models, 2022.

[3] Min Li, Zhijie Zhang, Liping Lei, Xiaofan Wang, and Xudong Guo. Agricultural greenhouses detection in high-resolution satellite images based on convolutional neural networks: Comparison of faster r-cnn, yolo v3 and ssd. *Sensors*, 20(17):4938, 2020.

[4] 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.

[5] Zach Carlson Andrew Napolitano Tyler Beard, Adam Bennion. Asl image recognition, 2022.

[6] Jorlogicus Jorlogicus Stefan Breunig Madhava Jay Tzutalin Darrenl, Rflynn. Labelimg, 2018.

[7] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.