TAYLOR'S UNIVERSITY

Wisdom · Integrity · Excellence

TAYLOR'S UWE DUAL AWARDS PROGRAMMES

AUGUST 2022 SEMESTER

MACHINE LEARNING AND PARALLEL COMPUTING

(ITS66604)

Individual Assignment (20%)

DUE DATE: 1st November 2022 via myTIMeS (8pm)

STUDENT DECLARATION

1. *I confirm that I am aware of the University's Regulation Governing Cheating in a University Test and Assignment and of the guidance issued by the School of Computing and IT concerning plagiarism and proper academic practice, and that the assessed work now submitted is in accordance with this regulation and guidance.*
2. *I understand that, unless already agreed with the School of Computing and IT, assessed work may not be submitted that has previously been submitted, either in whole or in part, at this or any other institution.*
3. *I recognise that should evidence emerge that my work fails to comply with either of the above declarations, then I may be liable to proceedings under Regulation.*

Student Name	Student ID	Date	Signature	Score
MD MESBAHUR RAHMAN	0344684	10/10/2022	mesba	

Live Sign Language Detection using Deep Learning

MD MESBAHUR RAHMAN (0344684)

November 6, 2022

1 Abstract

A large portion of the world's citizens are unable to speak due to birth defects or other unfortunate events. Sign language was developed to circumvent the problems that arise from this and allow non-speaking people to communicate just as anyone else would. Unfortunately, only an even smaller segment of the world's populace are able to understand sign language. In this paper, we used computer vision and machine learning techniques to create a model that is able to detect and correctly classify various sign language gestures. We reviewed 2 research articles that have previously tackled the task of sign language detection and studied their approach to the problem. Taking inspiration from their methods, we created our own sign language detection model using deep learning with a Single Shot Detection (SSD) neural network. We achieved a model that is capable of identifying sign language symbols within its dataset at an average speed of 700ms and an accuracy of 100%. We have proven the creation of such a model is possible and at a larger scale would greatly improve the lives of non-speakers, as well society as a whole.

Contents

2 Introduction

Intelligence among living creatures is not rare. What then, has allowed human beings to take over the world the way we have? Some say it's simply superior intelligence, others say that our ability to create and make use of tools has allowed us to accomplish any goal. These are important factors that has allowed humans to thrive, but at the core of it all is our ability to take advantage of the strengths of every individual. To share, pass down information through generations, create organized civilations and governments - we can do all of this because we have the ability to share complex ideas and thoughts through speech.

However, unfortunate events or birth defects cause some of us to be unable or struggle to hear and/or speak. For this reason, sign language was created to allow people with said defects to be able to communicate fluently with the same complexity we have through speech. However by it's nature, sign language - much like regular spoken language - is difficult to learn. It is a daunting task to learn any new language and sign language is no different, and may in many cases be more difficult.

The paper "Handling Sign Language Data: The Impact of Modality" by Quer and Steinbach (2019) states that less than 5-10% of individuals with hearing or speaking disabilities are born to parents with similar conditions and hence are unlikely to receive the proper education required and as a result, less than half of speaking and hearing impaired people are fully capable of communicating, through sign language or otherwise. With over 1.5% of the world's population being born with communication related disabilities and more than 20% of the world being at risk of hearing loss, less than 0.5% are able to communicate fluently through sign language.

Through the use of modern machine learning techniques, we are able to develop computer vision models which are able to detect different patterns in data. These pattern detection methods could be applied to understand and translate sign language to natural language. This paper will focus on and study and demonstrate modern day applications of machine learning to attempt to understand sign language, through the use of computer vision.

2.1 Research Goal

We understand that there are problems with modern implementations of sign language. Most of the population do not understand because it is difficult to learn, and most people are not given a fair opportunity to do so. Modern technology allows the combination of computer vision and machine learning classification techniques to automate the task of reading hand signs, labeling their meaning and making it visible to the user. We attempt to understand the process behind the creation of such models and how they can be applied to solve the problem posed by the lack of sign language understanding amongst the population.

2.2 Objectives

In this paper, we review two other research articles attempting to apply machine learning to match real examples of sign language use, to the training set of pre-labeled images of hand signs used in the specified language. We can adapt these different solutions and demonstrate how we ourselves can create a model that can use a live video feed from a webcamera to be able to classify sign language symbols in real time and assign a confidence level to each symbol.

3 Related Work

3.1 Overview

The papers below discuss different methods to collect and pre-process data, and use the data to train a deep learning model correctly classify static sign language symbols to their appropriate label. These papers were chosen specifically because of their similarity to the chosen topic and the use of recent technological advancements. Both papers discuss the use of a convolutional neural network (CNN) and its construction is described in detail.

3.2 Paper 1 : "Static Sign Language Recognition Using Deep Learning" by Juan et. al. - 2019

The undertaking describes the creation of a machine learning model based on CNNs to create a learning instrument for students seeking to understand American Sign Language (ASL). It is based on differentiating skin textures from non-skin texures in order to detect the signs being formed. The skin tone range was manually predefined by the researchers. Images containing sign language symbols being formed were fed as input to the CNN created using the popular Python library known as Keras. With appropriate illumination, picture quality and background, the model was able to achieve accuracies ranging from 90% to 97.5% for different parts of speech in ASL. The model was built with the goal of achieving fast compute times, allowing for real time sign language recognition - this requirement was satisfied with an average recognition time of 3.93 seconds.

3.3 Paper 2 : "Deep learning-based sign language recognition system for static signs" by Wadhawan and Kumar - 2019

The paper discusses another model based on the CNN architecture with the goal of recognizing Indian Sign Language (ISL). The model was trained using 35,000 images of 100 different ISL hand signs that were collected from a variety of users. The images are resized to the same desired resolution and used as input to the CNN. 50 different CNNs were built and the best results were shown where the number of layers were decreased - a higher accuracy was seen in a model with 4 layers as compared to a model with 8 layers. Different optimizers were also tested with each of the 50 CNNs and it was seen that a model utilizing 4 layers, 16 filters and the Adam optimizer achieved the best results, with 99.17% accuracy on colored images and 99.80% accuracy on grayscale images.

3.4 Critical Review

The neural network architecture described by Wadhawan and Kumar results in highly accurate models capable of correctly labeling a 100 different types of signs. The drawback to this approach is the required dataset, hardware and training time. The hardware described in the paper consists of industrial data center grade graphics cards and memory units that are not economically viable for the scope of this demonstration. On the other hand, Juan et. al describes a model that can process signs in real time. A side effect of this property is that it allows the model to be lightweight in nature and can be trained on less powerful hardware. The lower accuracy of 90%, compared to the 99% of Wadhawan and Kumar's paper is an acceptable difference for the current research objective. Furthermore, it is not clear if fast recognition of sign language is possible using the Wadhawan-Kumar model at all, regardless of the hardware being used and hence the Juan et al. paper's architecture is more appropriate for this application.

An important concept discussed in the Juan et al. paper is the focus on differentiating skin-tone colors from non-skin-tone colors. This approach is flawed because an image cannot perfectly represent a skin texture. Over or underexposure to light, unseen skin-tones and variation of skin textures may harm the model's detection of symbols being made. The only situation where this approach is superior to others is when there is a background that may be too close in color to the specific skin-tone. However, this situation can be understood to be a much rarer occurence than varying light conditions.

3.5 Takeaways and adaptations to this paper

Convolutional neural networks are strong tools used for computer vision based machine learning projects. They are able to take an image as an input and use primitive techniques to uniquely identify different features of the image, hence performing feature extraction directly from the data without having to manually define these features ourselves. The papers above leverage this strength to create highly accurate models to detect sign language. We will also be using the CNN architecture, however we will be making use of more modern neural network techniques, available to us through the Tensorflow python libraries.

The Juan et. al. paper describes a fast CNN architecture that gives us results in an average of 3.93s, however our paper is more focused towards live detection and hence we need an architecture that is even faster. To accomplish this, we will be making use of an Single Shot Detector (SSD) network. SSDs are a modern approach to real time object detection and are well suited to the objective.

SSDs are based on convolutional networks and are able to extract features from images in much the same way. It takes as input a regular frame from what would be a real time feed and ground truth - acting as labels to be identified by the detector network.

4 Methodology and Implementation

4.1 Business Understanding

A machine learning model capable of detecting and classifying sign language has various applications. As described in the Juan et. al. paper, it can be used as a learning tool for people seeking to study sign language. Traditionally, this learning is done with a peer or teacher who is already well versed. An immediete problem with human teaching is the feedback process. The teacher will need to communicate verbally (something which may not be possible with deaf students) and repeatedly instruct the students on better techniques. This is a tedious and time consuming process. With an automated system, the user will be able to practice fluidly with instant visual feedback - allowing them to quickly determine whether or not their execution is adequate.

A further use of the model can be seen in live translations. People who do not understand sign language would be able to do so with the use of the model as an interpreter. The process would be as simple as pointing a camera to the sign language user and each symbol would be identified and displayed to the user in real time. Communication between sign language users and non-sign language users would be made possible and simple.

The use of a machine learning model makes this all possible without the use of an internet connection. The detection can be used in an offline environment, only requiring a system with a camera and enough processing power to run the model. This requirement is satisfied by a majority of modern devices.

4.2 Analytical Approach

We require a model that is capable of detecting hands making specific gestures, identify and label the specific gesture being made and assign a confidence level to each detection. This must all be done in real time. A machine learning approach that is known to satisfy these criteria is a Single Shot Detector neural network. SSD networks are an extention of CNN architecures, able to extract features in the same way. The focus of this network lies in real time object detection. Labeled objects within larger images are used for the training data and the NN attempts to match features identified within the labeled space to features in the new input data. The strength of the SSD architecture is its speed and capability to run multiple classifications at once or in quick succession.

Pre-existing object detection models using SSD architectures are available in the public Tensorflow research repository on Github. We can use transfer learning techniques to repurpose these models to be specific to detect hand signs defined by our own dataset.

An SSD network satisfies the business requirements and the availability and hence the ease of creating such a model allows us to proceed with model development using the SSD architecture for our neural network.

4.3 Data Requirements

The data requirements depend on our objective and chosen model.

We require images of sufficient quality, depicting the use of sign language as static hand symbols. The SSD network requires this data to be labeled. Hence, a data file will be required alongside each image, containing information about the label being depicted in the image, and the coordinates of the object (hand sign) to be detected and classified.

The coordinates will be required to narrow down the region in which features must be extracted in order to be matched to future inputs. The whole image is still required - this is to allow "context" for each image.

The whole dataset needs to include a sufficient amount of pictures for each symbol. This includes both training and testing data. Variety in the data is required - camera angles, distances from the camera and lighting conditions must be varied in order to allow the model to be able to identify the same signs being made with different surroundings and in different sizes.

For this project, we have chosen to select 6 different static sign language words for testing, they are : Hello, Yes, No, Goodbye, Thank You, I Love You. 15 images are collected for each sign, with 12 being set aside for training and 3 used for testing - giving us a 80:20 train:test ratio.

Figure 1: Set of Sign Language gestures used in the dataset

4.4 Data Collection

The data was collected using a 720p webcam. As described in the requirements, 15 pictures of each of the 6 static sign language symbols were performed infront of the camera and were captured, giving the dataset a total size of 90 images. The requirements of the contents of the image were followed closely, making sure to each image depicting the same symbol were captured in different positions, some being slightly similar while others were drastically different. An example is shown below.

Figure 2: Image variation within same class of sign language symbol

Figure 2 depicts three different images used to show the symbol "Goodbye". The first image shows a high quality image with a typical use of the symbol. The second image shows the same symbol being made with the opposite hand. The third image shows a lower quality image with a varied hand position. These variations in image quality, and body/hand position are repeated throughout the dataset for the purpose of data variety.

4.5 Data Understanding

The key feature of each image is of course the class of the hand sign being depicted. Other important features are the dimensions (and hence the quality) of the image, the relative amount of light present in each picture, the position of the sign within the picture and the relative size of the symbol in the picture (relates to distance from the camera, the sign being large indicates it being closer to the camera).

These features have been varied throughout the different images. This is because in a real time video feed, there will be constant changes in the environment and lighting, along with signs being seen from various angles and distances from the camera. The similarities in the features of the hand symbol despite the differences of the mentioned features is the problem that will be tackled by the SSD network. These features are not explicitly defined or annotated by us, but are extracted through the neural network, directly from the image. The only exception being the hand sign itself - we manually annotate and label the region containing the hand sign.

4.6 Data Preperation

The first stage in data preperation is to appropriately annotate and label the images as necessary. For this process, we use a python program known as "label image". The program allows us to annotate specific areas of an image, and save this data into a seperate file.

Figure 3: Area annotated using LabelImage

goodbye.6b16bba8-583d-11ed-8e45-309c23af27cd.xml

```xml
<annotation>
    <folder>collectedimages</folder>
    <filename>goodbye.6b16bba8-583d-11ed-8e45-309c23af27cd.jpg</filename>
    <path>D:\Conda Projects\test\RealTimeObjectDetection\Tensorflow\workspace\images\collectedimages\goodbye.6b16bba8-583d-1
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>1920</width>
        <height>1080</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>Goodbye</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>1245</xmin>
            <ymin>287</ymin>
            <xmax>1529</xmax>
            <ymax>595</ymax>
        </bndbox>
    </object>
</annotation>
```

Figure 4: XML file containing the data associated with the annotation made in the previous image

The figures above show an example of how one image would be prepared. The image is taken and the section containing the hand sign is annotated with the name of the label and the coordinates of the boundaries of the sections. The data is saved in the format of an XML file as shown in Figure 4.

The data is then seperated into training and test data. We use 80% for training and 20% for testing. This gives us 12 images for training and 3 for testing for each sign, giving us a total of 72 images for our training set and 18 images for our test set.

For processing by the model, a label map is required. The label map assigns each label a specific id. This is created with the follow code.

In [ ]:

```
labels = [
    {'name':'Goodbye', 'id':1},
    {'name':'Hello', 'id':2},
    {'name':'I Love You', 'id':3},
    {'name':'No', 'id':4},
    {'name':'Thank You', 'id':5},
    {'name':'Yes', 'id':6}]

with open('Tensorflow\workspace\annotations\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')
```

Figure 5: Code used to generate label map

This gives us the follow file.

label_map - Notepad

File Edit Format View Help

```
item {
        name:'Goodbye'
        id:1
}
item {
        name:'Hello'
        id:2
}
item {
        name:'I Love You'
        id:3
}
item {
        name:'No'
        id:4
}
item {
        name:'Thank You'
        id:5
}
item {
        name:'Yes'
        id:6
}
```

Figure 6: Label Map, saved as labelmap.pbtxt

Another requirement by the model is known as a Tensorflow record. The following script is provided by the Tensorflow library to create these records.

```
[ ]: #SAVED LOCALLY AS generate-tfrecord.py

""" Sample TensorFlow XML-to-TFRecord converter

usage: generate_tfrecord.py [-h] [-x XML_DIR] [-l LABELS_PATH] [-o OUTPUT_PATH]␣
 ↪[-i IMAGE_DIR] [-c CSV_PATH]

optional arguments:
  -h, --help            show this help message and exit
```

```
  -x XML_DIR, --xml_dir XML_DIR
                        Path to the folder where the input .xml files are stored.
  -l LABELS_PATH, --labels_path LABELS_PATH
                        Path to the labels (.pbtxt) file.
  -o OUTPUT_PATH, --output_path OUTPUT_PATH
                        Path of output TFRecord (.record) file.
  -i IMAGE_DIR, --image_dir IMAGE_DIR
                        Path to the folder where the input image files are␣
 ↪stored. Defaults to the same directory as XML_DIR.
  -c CSV_PATH, --csv_path CSV_PATH
                        Path of output .csv file. If none provided, then no file␣
 ↪will be written.
"""

import os
import glob
import pandas as pd
import io
import xml.etree.ElementTree as ET
import argparse

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'    # Suppress TensorFlow logging (1)
import tensorflow.compat.v1 as tf
from PIL import Image
from object_detection.utils import dataset_util, label_map_util
from collections import namedtuple

# Initiate argument parser
parser = argparse.ArgumentParser(
    description="Sample TensorFlow XML-to-TFRecord converter")
parser.add_argument("-x",
                    "--xml_dir",
                    help="Path to the folder where the input .xml files are␣
 ↪stored.",
                    type=str)
parser.add_argument("-l",
                    "--labels_path",
                    help="Path to the labels (.pbtxt) file.", type=str)
parser.add_argument("-o",
                    "--output_path",
                    help="Path of output TFRecord (.record) file.", type=str)
parser.add_argument("-i",
                    "--image_dir",
                    help="Path to the folder where the input image files are␣
 ↪stored. "
                         "Defaults to the same directory as XML_DIR.",
                    type=str, default=None)
```

```
parser.add_argument("-c",
                    "--csv_path",
                    help="Path of output .csv file. If none provided, then no␣
 ↪file will be "
                         "written.",
                    type=str, default=None)

args = parser.parse_args()

if args.image_dir is None:
    args.image_dir = args.xml_dir

#label_map = label_map_util.load_labelmap(args.labels_path)
#label_map_dict = label_map_util.get_label_map_dict(label_map)

label_map_dict = label_map_util.get_label_map_dict(args.labels_path)

def xml_to_csv(path):
    """Iterates through all .xml files (generated by labelImg) in a given␣
 ↪directory and combines
    them in a single Pandas dataframe.

    Parameters:
    ----------
    path : str
        The path containing the .xml files
    Returns
    -------
    Pandas DataFrame
        The produced dataframe
    """

    xml_list = []
    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        for member in root.findall('object'):
            value = (root.find('filename').text,
                     int(root.find('size')[0].text),
                     int(root.find('size')[1].text),
                     member[0].text,
                     int(member[4][0].text),
                     int(member[4][1].text),
                     int(member[4][2].text),
                     int(member[4][3].text)
                     )
            xml_list.append(value)
```

```
    column_name = ['filename', 'width', 'height',
                   'class', 'xmin', 'ymin', 'xmax', 'ymax']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    return xml_df


def class_text_to_int(row_label):
    return label_map_dict[row_label]


def split(df, group):
    data = namedtuple('data', ['filename', 'object'])
    gb = df.groupby(group)
    return [data(filename, gb.get_group(x)) for filename, x in zip(gb.groups.
 ↪keys(), gb.groups)]


def create_tf_example(group, path):
    with tf.gfile.GFile(os.path.join(path, '{}'.format(group.filename)), 'rb')␣
 ↪as fid:
        encoded_jpg = fid.read()
    encoded_jpg_io = io.BytesIO(encoded_jpg)
    image = Image.open(encoded_jpg_io)
    width, height = image.size

    filename = group.filename.encode('utf8')
    image_format = b'jpg'
    xmins = []
    xmaxs = []
    ymins = []
    ymaxs = []
    classes_text = []
    classes = []

    for index, row in group.object.iterrows():
        xmins.append(row['xmin'] / width)
        xmaxs.append(row['xmax'] / width)
        ymins.append(row['ymin'] / height)
        ymaxs.append(row['ymax'] / height)
        classes_text.append(row['class'].encode('utf8'))
        classes.append(class_text_to_int(row['class']))

    tf_example = tf.train.Example(features=tf.train.Features(feature={
        'image/height': dataset_util.int64_feature(height),
        'image/width': dataset_util.int64_feature(width),
        'image/filename': dataset_util.bytes_feature(filename),
        'image/source_id': dataset_util.bytes_feature(filename),
```

```
        'image/encoded': dataset_util.bytes_feature(encoded_jpg),
        'image/format': dataset_util.bytes_feature(image_format),
        'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
        'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
        'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label': dataset_util.int64_list_feature(classes),
    }))
    return tf_example


def main(_):

    writer = tf.python_io.TFRecordWriter(args.output_path)
    path = os.path.join(args.image_dir)
    examples = xml_to_csv(args.xml_dir)
    grouped = split(examples, 'filename')
    for group in grouped:
        tf_example = create_tf_example(group, path)
        writer.write(tf_example.SerializeToString())
    writer.close()
    print('Successfully created the TFRecord file: {}'.format(args.output_path))
    if args.csv_path is not None:
        examples.to_csv(args.csv_path, index=None)
        print('Successfully created the CSV file: {}'.format(args.csv_path))


if __name__ == '__main__':
    tf.app.run()
```

We create Tensorflow records for the training and testing data with the following commands executed on the command line.

[ ]:
```
!python {'Tensorflow/scripts' + '/generate_tfrecord.py'} -x {'Tensorflow/
 ↪workspace/images/train'} -l {'Tensorflow/workspace/annotations/label_map.
 ↪pbtxt'} -o {'Tensorflow/workspace/annotations/train.record'}
!python {'Tensorflow/scripts' + '/generate_tfrecord.py'} -x{'Tensorflow/
 ↪workspace/images/test'} -l {'Tensorflow/workspace/annotations/label_map.
 ↪pbtxt'} -o {'Tensorflow/workspace/annotations/test.record'}
```

All required data is prepared and we proceed to building our neural network.

4.7 Modeling

We will be using an SSD network, along with transfer learning techniques in order to apply pre-trained object detection models to help train our sign language detection model.

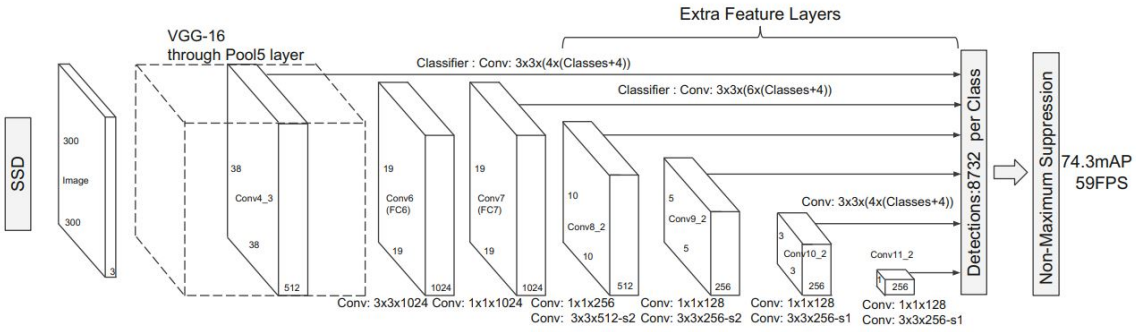


Figure 7: SSD Network Layer Architecture

Figure 7 shows the layer architecture of the SSD network used by the Tensorflow library for object detection. It consists of a pooling layer, and several convolutional layers. The depth of these convolutional layers depend on the number of classes the model is being trained for.

Different aspects of the model are configured using the configuration file. Each relevant section of the config is explained in detail below.

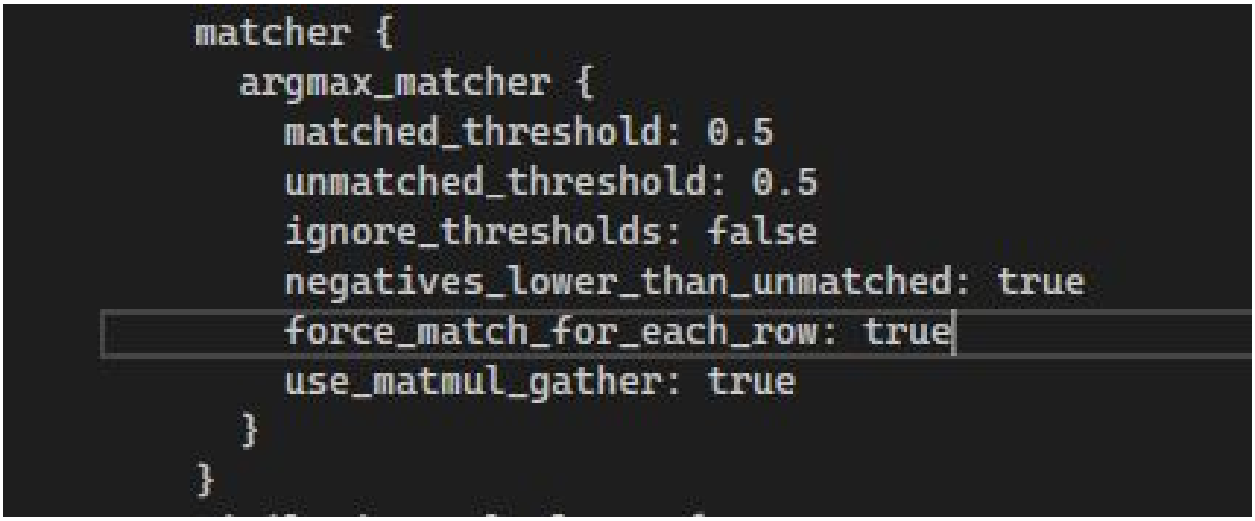
pipeline.config

```
model {
  ssd {
    num_classes: 6
    image_resizer {
      fixed_shape_resizer {
        height: 320
        width: 320
      }
    }
```

This shows the base configuration of the SSD network. We set the number of classes to be identified to be 6. Each image is resized to the dimensions 320x320 (height x width).

```
feature_extractor {
  type: "ssd_mobilenet_v2_fpn_keras"
  depth_multiplier: 1.0
  min_depth: 16
  conv_hyperparams {
    regularizer {
      l2_regularizer {
        weight: 3.9999998989515007e-05
      }
    }
    initializer {
      random_normal_initializer {
        mean: 0.0
        stddev: 0.009999999776482582
      }
    }
    activation: RELU_6
    batch_norm {
      decay: 0.996999979019165
      scale: true
      epsilon: 0.0010000000474974513
    }
  }
  use_depthwise: true
  override_base_feature_extractor_hyperparams: true
  fpn {
    min_level: 3
    max_level: 7
    additional_layer_depth: 128
  }
}
```

These are the specified configurations of the section of the neural network used to extract features from each image. Here we define the type of feature extractor and minimum depth. We also define the parameters of the convolutional network - the type of regulizer used, the initializer and the activation type. The specific settings are shown in the figure.



We set the configurations for the matcher. We set the minimum confidence level for a match being seen to be 50%

```
box_predictor {
  weight_shared_convolutional_box_predictor {
    conv_hyperparams {
      regularizer {
        l2_regularizer {
          weight: 3.99999989895150 07e-05
        }
      }
      initializer {
        random_normal_initializer {
          mean: 0.0
          stddev: 0.009999999776482582
        }
      }
      activation: RELU_6
      batch_norm {
        decay: 0.996999979019165
        scale: true
        epsilon: 0.0010000000474974513
      }
    }
    depth: 128
    num_layers_before_predictor: 4
    kernel_size: 3
    class_prediction_bias_init: -4.599999904632568
    share_prediction_tower: true
    use_depthwise: true
  }
}
```

We set the specific parameters of the box predictor. Once again we use a regulizer, intializer and activation.

```
train_config {
  batch_size: 4
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_crop_image {
      min_object_covered: 0.0
      min_aspect_ratio: 0.75
      max_aspect_ratio: 3.0
      min_area: 0.75
      max_area: 1.0
      overlap_thresh: 0.0
    }
  }
  sync_replicas: true
  optimizer {
    momentum_optimizer {
      learning_rate {
        cosine_decay_learning_rate {
          learning_rate_base: 0.07999999821186066
          total_steps: 50000
          warmup_learning_rate: 0.026666000485420227
          warmup_steps: 1000
        }
      }
      momentum_optimizer_value: 0.8999999761581421
    }
    use_moving_average: false
  }
  fine_tune_checkpoint: "Tensorflow/workspace/pre-trained-models/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8/checkpoint/ckpt-0"
  num_steps: 50000
  startup_delay_steps: 0.0
  replicas_to_aggregate: 8
  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  fine_tune_checkpoint_type: "detection"
  fine_tune_checkpoint_version: V2
}
```

Here we set the configuration for the required training steps. We define the batch size, which is the number of samples we process before making changes to the model. A higher number will require more computing and vice versa. As the size of our dataset is relatively low, we set this to be 4.

Data augmentation settings are defined to assume how much the the data should be allowed to change. We set that an object's features being flipped horizontally should still be detected, furthermore we define the different aspect ratios and size of the object relative to it's original size. The chosen optimizer is a momentum optimizer and configured with a specified base learning rate and learning rate decay.

We set the options for fine tuning to be of the detection type and use a pretrained model checkpoint to fine tune detections made appropriately.

```
train_input_reader {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/train.record"
  }
}
eval_config {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/test.record"
  }
}
```

Here we define the input to be read. We feed in our previously defined label map and tensorflow record file created for our training data.

We also setup the evaluation metrics to be shown during training and similarly to our training data, feed in our label map and tensorflow record for our test data.

The configurations for the model are prepared and we can begin training the model. This is done with the following command.

```
D:\University Stuff\Sem 5\MLPC\RealTimeObjectDetection>python Tensorflow/models/research/object_detection/model_main_tf2.py --model_dir=Tensorflow/workspace/models/my_ssd_mobnet
--pipeline_config_path=Tensorflow/workspace/models/my_ssd_mobnet/pipeline.config --num_train_steps=10000
```

Figure 8: Command used to begin the training process, using the settings defined in pipeline.config

```
You may not need to update to CUDA 11.1; cherry-picking the ptxas binary is often sufficient.
INFO:tensorflow:Step 100 per-step time 0.419s
I1102 03:30:32.864053 19824 model_lib_v2.py:705] Step 100 per-step time 0.419s
INFO:tensorflow:{'Loss/classification_loss': 0.4709408,
 'Loss/localization_loss': 0.39284348,
 'Loss/regularization_loss': 0.15411115,
 'Loss/total_loss': 1.0178955,
 'learning_rate': 0.0319994}
```

Figure 9: Training Started, intial loss metrics

We can see that the model has begun training with the initial loss metrics being shown.

```
INFO:tensorflow:{'Loss/classification_loss': 0.11631052,
 'Loss/localization_loss': 0.025639325,
 'Loss/regularization_loss': 0.09921524,
 'Loss/total_loss': 0.24116509,
 'learning_rate': 0.07352352}
I1102 03:47:40.543819 19824 model_lib_v2.py:708] {'Loss/classification_loss': 0.11631052,
 'Loss/localization_loss': 0.025639325,
 'Loss/regularization_loss': 0.09921524,
 'Loss/total_loss': 0.24116509,
 'learning_rate': 0.07352352}
```

Figure 10: Training Ended, final loss metrics

The model has concluded training and the final loss metrics are displayed.

4.8 Results and Evaluation

The model is used to predict the classes seperated into the test data. The following results were observed in the following confusion matrix.

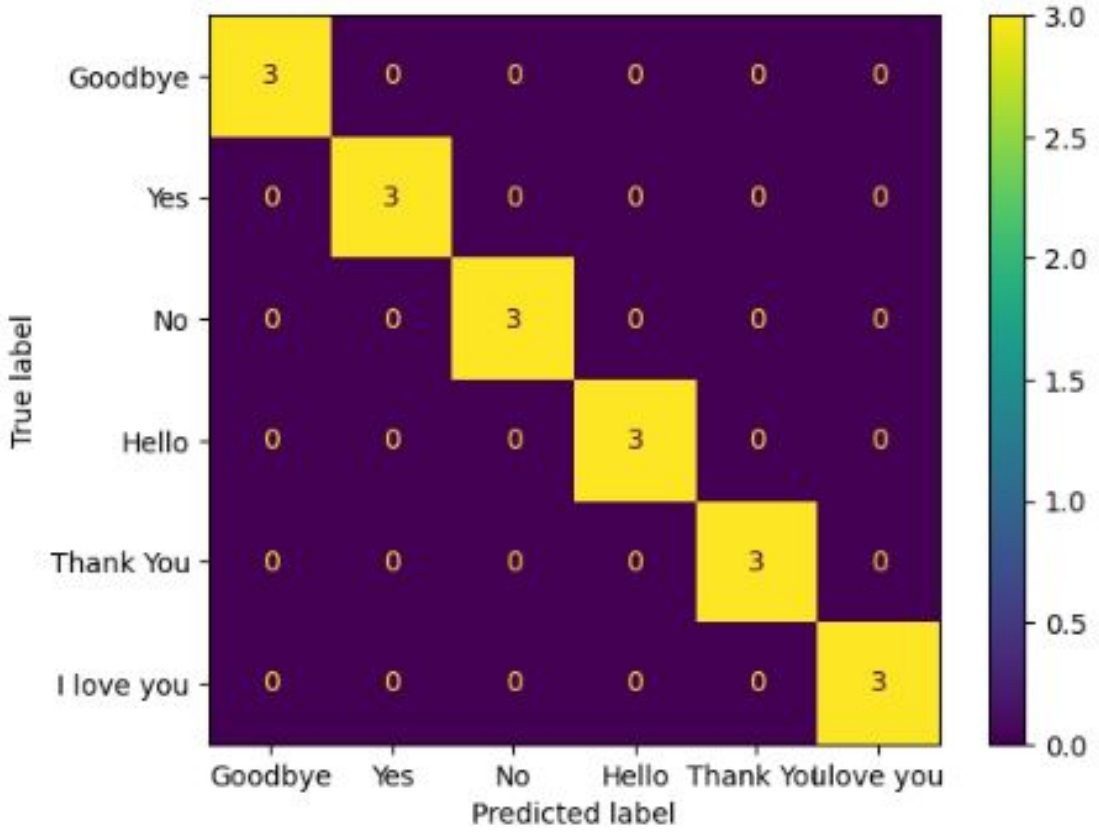


Figure 11: Confusion Matrix: Predicted vs Actual Label

Common metrics for computer vision based machine learning models are the average precision and recall. These are calculated as follows.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Where, TP = True Positive, FP = False Positive, FN = False Negatives.

Therefore, we calculate

Average Precision = 1 Recall = 1

The model has correctly classified each of the sign language classes depicted in the images and hence the evaluation metrics give us a perfect precision and recall score of 1. We can conclude that the model is able to accurately predict each sign language class.

4.9 Deployment

The model is deployed with a real time feed from a webcam with the following code.

```
[ ]: import os
     import tensorflow as tf
     import cv2
     import numpy as np
     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
     from object_detection.utils import config_util
     from object_detection.protos import pipeline_pb2
     from google.protobuf import text_format


     configs = config_util.get_configs_from_pipeline_file('RealTimeObjectDetection/
      ↪Tensorflow/workspace/models/my_ssd_mobnet/pipeline.config')
```

```
detection_model = model_builder.build(model_config=configs['model'],␣
↪is_training=False)

ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
ckpt.restore(os.path.join('RealTimeObjectDetection/Tensorflow/workspace/models/
↪my_ssd_mobnet', 'ckpt-11')).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('RealTimeObjectDetection/Tensorflow/
↪workspace/annotations/label_map.pbtxt')

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

while True:
    ret, frame = cap.read()
    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

    detections['detection_classes'] = detections['detection_classes'].astype(np.
↪int64)

    label_id_offset = 1
    image_np_with_detections = image_np.copy()

    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, (1920,␣
↪1080)))

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

This launches a window with a webcam feed, annotating the sign language symbols as they are detected on screen.

Correctly matched "Hello"

Correctly matched "Yes"

Correctly matched "No"

Correctly matched "I Love You"

Correctly matched "Goodbye"

It can be seen that the deployed model correctly detects and identifies hand signs made in a live video.

5 Analysis

5.1 "Static Sign Language Recognition Using Deep Learning" by Juan et. al. - 2019

A strength of the model seen in the Jaun et. al. paper is the model's speed. It is able to achieve an adequately high accuracy. An element of the design that can be considered both a flaw and a strength, is seen in the design's explicit focus on static backgrounds, lighting and predefined skin-tone ranges. The results show an average of 93.67% accuracy in its predictions but this is only within the ideal conditions set by the researchers. The model may perform poorly outside of these conditions. The predefined skin-tone may not be able to properly detect hands with skin tones outside or at the edges of the thresholds and furthermore it may be influenced by the background. A background with a color matching the skin tone of the user may reveal problems with the detection model. To conclude, the model works well in a controlled environment, but with the intention of being used as a learning tool, more often than not, the background, lighting, camera angle and the user's skin tone will not be perfectly desirable to the model.

5.2 "Deep learning-based sign language recognition system for static signs" by Wadhawan and Kumar - 2019

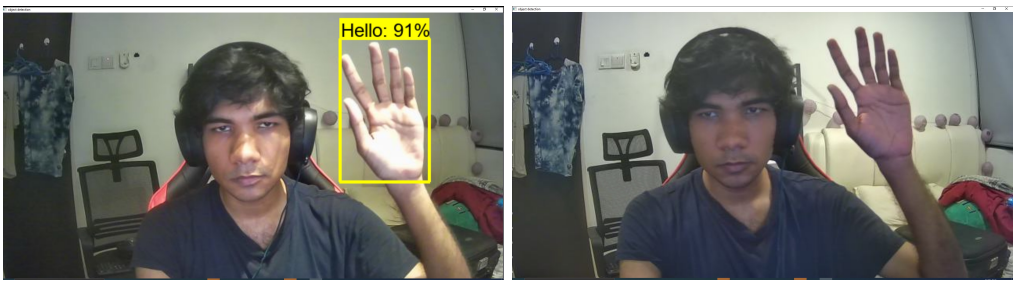
In contrast, the model observed in the Wadhawan and Kumar paper sacrifices speed in preference to accuracy. 50 different models were constructed and the best model was chosen strictly on its ability to correctly identify the label regardless of the computation time, achieving an accuracy of 99.9% on greyscale images. Such a high evaluation requires us to investigate the possibility of overfitting. The model has been created with a dataset including 35,000 images of 100 different static signs. A dataset with a high number of records does not directly imply overfitting has occurred, but it this case this judgement is appropriate because with the addition of more and more images, the dataset does become more complex with an increasing amount of features and scenarios. Thus we can say that overfitting has occured and the model's evaluation is inflated. Although the model's evaluation may not be accurate, a strength of such a complex dataset is that it is able to capture a large variety of situations. There are few situations which are unseen and hence in real world application would run better than a model with a smaller dataset.

5.3 Our implementation

We end up with a model with a 100% accuracy, according to evaluation against our test data. This is of course, too good to be true. In our demonstration, we notice certain false positives with the model detecting certain sections of the background as one of the sign language symbols, which is of course incorrect. This can be attributed to multiple factors regarding our dataset. The size of the dataset only consists of 6 different sign language symbols with 15 images depicting each label. Comparing to the datasets used in the Juan et. al and Wadhawan and Kumar papers, we know that this dataset is very small. The model has few examples to learn from and match to. Hence overfitting is the likely culprit of the high accuracy.

This problem is further made clear in the model deployment. We attempted to change the lighting in the room to see if the model would still be able to detect the signs being made and it was seen that the model struggled to detect any signs at all with only occasional detections which were not at all accurate. This tells us that the dataset used was inadequete in both size and variety.

Figure 12: Comparing lighting used in training dataset vs new lightingl



The above figure shows that in new lighting, the model is unable to detect the same hand sign in a similar position.

There is a noticable delay in the hand signs being formed on the video and being detected and classified in the box predictors. The average delay recorded was about 700ms. This is a noticable delay but nearly 6 times faster than the CNN model discussed in the Juan et. al paper. 700ms is acceptable for individual signs, however multiple signs stringed together may cause a problem as some signs would not have enough time to be properly detected and classified.

6 Recommendations

An immediete remedy to our implementation's overfitting and inability to detect labels under different circumstances would be with an increased dataset size. We have confirmed that the model is able to detect the same hand sign in different positions and camera angles. However, our dataset did not account for different lightings or backgrounds. A dataset could be constructed with larger considerations for variation and with a larger amount of entries, the model would be able to overcome these imperfections.

Another workaround to this problem was seen in the Jaun et. al., where certain features of the subject are not left to the model's feature extractor and instead manually set as a predefined parameter. This however would take much more manual labor and potentially lead to less than favorable results as compared to increasing the quality and size of our dataset.

Our model detects signs faster than the above papers and yet the delay is still noticeable. SSDs are fast detection networks but there are comparable neural network architectures that rival and in certain cases surpass SSDs. These are known as Faster Regional CNN models. They are capable of combining different regions into similar regions and running fast detection algorithms that are also popular for object detection models. The alternate neural network architecture of Faster R-CNN models would be a suitable alternative to the SSD network used in this implementation.

The model detects sign language gestures individually but does not concern itself with gestures made previously as context to predict the current or future symbols. This is a problem that is tackled by language models, using probability distributions to calculate the probability of a certain word given a number of previous words. This can be implemented into our system using a Hidden Markov Model (HMM) using Bayes' Theorem to calculate the probability of the current symbol being detected, along with the base detection algorithm. This is especially useful when the user is attempting to form sentences using sign language gestures in fluid succession. With this additional context, the model may be more accurate in its classifications.

7 Conclusion

Communication is a core building block of society and the progress of humanity and unfortunately a large portion of the population struggle to participate due to birth defects or other unfortunate circumstances. Sign language provides individuals unable to speak to still be able to conveniently communicate just as effectively as anyone else. Just like any other spoken language however, it must be learned and practiced for years on end. Furthermore, even if one were to take the time to learn sign language to an effective level, it would not be usable in regular life unless everyone they were trying to communicate to also understood this language. Unfortunately, not everyone has the time for such an undertaking.

In regular languages, machine learning techniques have been used with natural language processing to understand the meaning of a body of text in one language and hence form a sentence with the same meaning in another language. This form of translation is not possible with sign language as we have no text to work with, however, we do have computer vision. We have created a model which is able to look at any sign language gestures within the dataset and instantaneously display its meaning on screen.

This can be used by both speaking and non-speaking people to use as a tool to ease the process of learning sign language. Furthermore, it can be used as a direct translation tool, foregoing the learning process entirely only requiring a video input from any device.

The creation of such a model has greater implications for the progress of society. The progress and impact of a country on the world can largely depend on its population. The advent of an instant translation tool for such a large portion of the world's population is equivalent to unlocking huge untapped potential. Many non-speaking people struggle to find jobs or participate in activities that increase the productivity and strength of any given field and this tool may allow them to do so.

8 Main References

Tolentino, L., Juan, R., Thio-ac, A., Pamahoy, M., Forteza, J., & Garcia, X. (2019). Static Sign Language Recognition Using Deep Learning. International Journal of Machine Learning and Computing. Retrieved 9 October 2022, from https://www.researchgate.net/publication/337285019_Static_Sign_Language_Recognition_Using_Deep_Learn

Wadhawan, A., & Kumar, P. (2019). Deep learning-based sign language recognition system for static signs. Retrieved 9 October 2022, from https://doi.org/10.1007/s00521-019-04691-y.

9 Other References

Shapiro, L. (2018). Computer vision: the last 50 years. Taylor & Francis. Retrieved 9 October 2022, from https://doi.org/10.1080/17445760.2018.1469018.

Quer, J., & Steinbach, M. (2019). Handling Sign Language Data: The Impact of Modality. Retrieved 9 October 2022, from https://pubmed.ncbi.nlm.nih.gov/30914998/.