Signo-lingo

Team: Ong Xiang Qian, Lin Huiqing, Jamie Tan, Glenn Chia

1 INTRODUCTION **3**
1.1 Problem Statement 3
1.2 Dataset 3

2 SOURCE CODE **3**

3 DIRECTORY STRUCTURE **4**

4 INSTRUCTIONS TO RUN THE CODE **4**
4.1 Setting Up the Environment 4
4.1.1 Using conda 4
4.1.2 Using pip 5
4.2 Downloading the Dataset 5
4.3 Running the notebook that trains the final model on the full dataset 6
4.4 Testing the model on some provided samples using a notebook 6
4.5 Running the GUI locally 8
4.5.1 Running on Windows 8
4.5.2 Running on RHEL8 12
4.6 Accessing the GUI remotely 15

5 DATASET **19**
5.1 Dataset Breakdown 19
5.2 Exploration 19
5.3 Preprocessing 23
5.3.1 Fixing 30 Frames Per Video 23
5.3.2 Choosing 10 Classes 26
5.4 Data Augmentation 31

6 MODEL **32**
6.1 State-of-the-Art 32
6.2 Good Practices 34
6.2.1 Early Stopping 34
6.2.2 Learning Rate 35
6.2.3 Saving Checkpoints 36
6.2.4 Graph Generation 37
6.3 Exploration 39
6.3.1 CNN Experiments 40

6.3.1.1 Experiment 1: Effect of Number of Convolutional Layers 40
6.3.1.2 Experiment 2: Effect of Latent Vector Dropout Rate 43
6.3.1.3 Experiment 3: Effect of Weight Decay of Optimizer 44
6.3.2 RNN Experiments 45
6.3.2.1 Experiment 1: Effect of Attention 46
6.3.2.2 Experiment 2: Effect of Bidirectional LSTM 47
6.3.3 Pretrained + RNN/LSTM 48
6.3.4 Data Augmentation 49
6.3.4.1 Masking Data Augmentation 50
6.3.4.2 Pose Estimation Data Augmentation 51
6.3.4.3 Data Augmentation Conclusion 52
6.4 Final Architecture 52

7 RESULTS & EVALUATION 53

8 MODEL FAILURE 57
8.1 Funny Failure 59

9 LEARNING POINTS 61

10 CONCLUSION 61

11 MEMBER CONTRIBUTIONS 62

12 BIBLIOGRAPHY 62

1 INTRODUCTION

1.1 Problem Statement

Sign language is a way of expressing oneself primarily through a series of hand gestures. However, it is dominantly used by the deaf and mute community and may prove to be a challenge for others who do not use the language to understand. Moreover, sign language is not universal - each community develops its own sign language with unique intricacies. As such, this project aims to develop a sign language video classification model that will help others to interpret the semantic meanings behind the language.

Inputs: Videos of gestures representing words in the Turkish Sign Language.

Outputs: Classes that are mapped to a specific meaning, in English.

Deliverables: Source Code (models, weights, logs, Jupyter Notebooks) as well as an URL to a GUI that is hosted on AWS

1.2 Dataset

The dataset chosen for this project is the Turkish Sign Language dataset (AUTSL) which consists of 226 signs performed by 43 different signers and 38,336 isolated sign video samples in total. Samples contain a wide variety of backgrounds recorded in indoor and outdoor environments. On top of that, spatial positions and the postures of signers also vary in the recordings with each sample recorded with Microsoft Kinect v2 and contains color image videos (RGB) and depth videos. The team will be using 10 classes for this project, with more information found in Section 5.3.2.

The dataset can be accessed through the following links:

- Original dataset site: http://chalearnlap.cvc.uab.es/dataset/40/description/
- Zipped file of whole dataset: dataset.zip

2 SOURCE CODE

The source code for this project can be found in the GitHub Repository:
https://github.com/LinHuiqing/signo-lingo

3 DIRECTORY STRUCTURE

In the directory, the components are split into multiple sections as shown in Figure 1.

```
.
├── src/                                # logs gathered from
│   ├── dataset/                        # stores train,val,test dataset
│   ├── dev_model/                      # stores model
│   ├── test_model/                     # store notebook for testing
│   ├── train_final/                    # final notebook
│   ├── Exploratory Data Analysis.ipynb # notebook to run data analysis
│   └── log_file_plots.ipynb            # code for plotting graphs based on log files
├── models/                             # saved models
├── web_gui/                            # code for web GUI
├── Big_Project_Instructions.pdf        # instructions for project
└── README.md
```

Figure 1. Directory Structure

4 INSTRUCTIONS TO RUN THE CODE

4.1 Setting Up the Environment

There are 2 ways of setting up the environment which were prepared.

1. Using conda
2. Using pip

4.1.1 Using conda

1. Configure the conda environment

```
conda env create -f environment.yml
```

2. To use environment

```
conda activate dl-big-proj
```

3. Install pip on conda environment

```
conda install pip
```

4. Install torchinfo with pip

```
pip install torchinfo
```

5. Open jupyter notebook

```
jupyter-notebook
```

4.1.2 Using pip

1. Create a virtual environment

```
virtualenv venv
```

2. To use environment

```
source venv/bin/activate
```

3. Install relevant packages

```
pip install -r requirements.txt
```

4. Open jupyter notebook

```
jupyter-notebook
```

4.2 Downloading the Dataset

Download the dataset from the links provided in Section 1.2. The second link to the zipped file of the whole dataset is recommended as the process of downloading from the original site is quite tedious.

After which, place the extracted contents of the file into the ./src/dataset directory. The directory should resemble Figure 2 where the train, test, val directories with their respective videos are placed in the dataset directory.

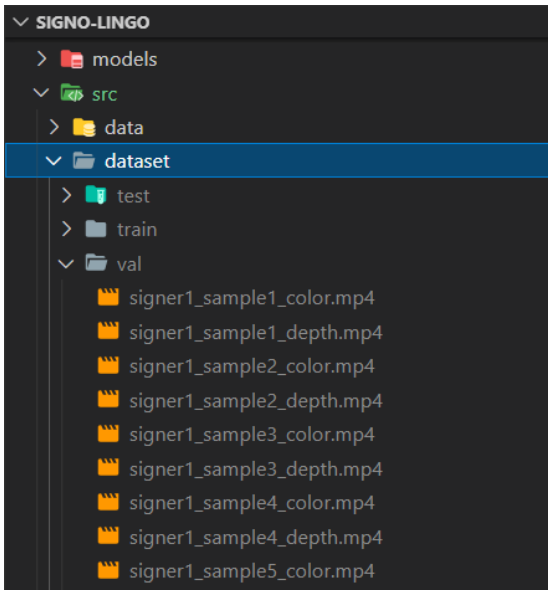

Figure 2. Train, test val directories which are part of the extracted zip file are copied to the dataset directory

4.3 Running the notebook that trains the final model on the full dataset

From the ./src directory, locate the train_final.ipynb notebook as shown in Figure 3.

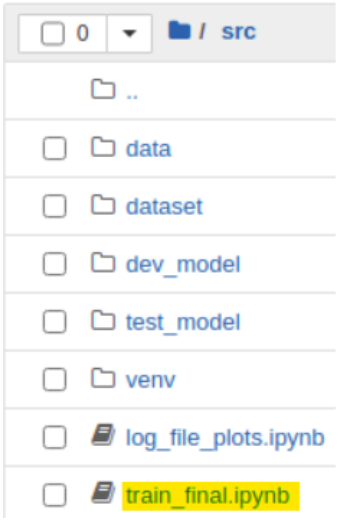

Figure 3. Select the train_final.ipynb notebook

Open the train_final.ipynb notebook. Run the cells sequentially based on their order in the notebook.

4.4 Testing the model on some provided samples using a notebook

From the ./src/test_model directory, locate the load_and_test_model_single_video.ipynb notebook as shown in Figure 4.

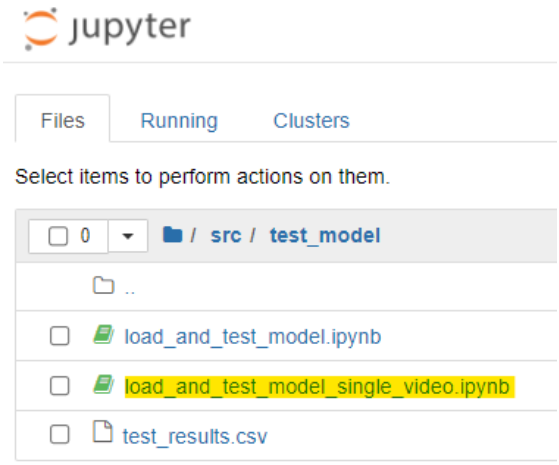

Figure 4. Select the load_and_test_model_single_video.ipynb notebook

Open the load_and_test_model_single_video.ipynb notebook. At the top of the notebook, feel free to change the vid_name variable to any other video name in the test set. In Figure 5, the vid_name chosen is “signer34_sample4”.

Load and Test Model for single video

```
In [1]: # Replace this video name with other video names in the test set.
        # Example: signer34_sample43, signer34_sample145, signer30_sample576
        vid_name = "signer34_sample4"
```

Figure 5. Enter a video name that will be used for prediction

After setting the vid_name, run the whole notebook. The results for the ground truth and the predicted label are displayed in the cell as shown in Figure 6.

```
In [14]: predict_id, predict_label, ground_truth_id, ground_truth_label = load_and_test_video(vid_name)

In [15]: print(f"Ground truth: {ground_truth_label}")
         print(f"Predicted label: {predict_label}")

         Ground truth: champion
         Predicted label: champion
```

Figure 6. Ground truth labels and the prediction label for the chosen video

In addition, a plot to visualize the series of actions is displayed at the bottom of the notebook (see Figure 7).

```
In [17]: visualize_frames(test_dir, vid_name)
```

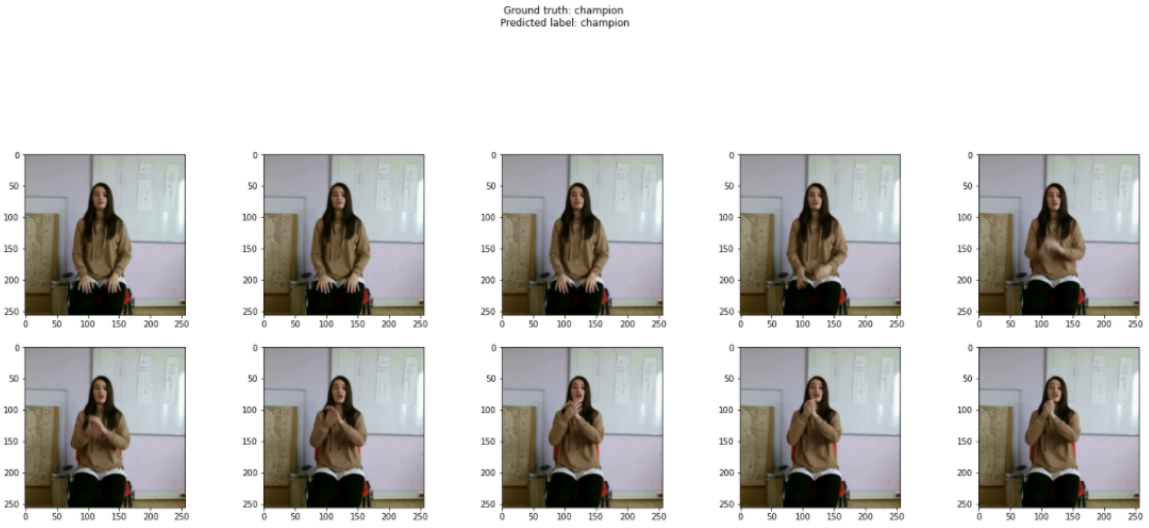

Figure 7. Smaller screenshot of 10 of the 30 displayed frames

4.5 Running the GUI locally

The GUI has been tested on Windows 10 and Red Hat Enterprise Linux 8 (RHEL8) OSes. We also provide an alternative to running the GUI locally as the GUI has also been deployed on AWS at http://3.1.85.200/ (refer to Section 4.6).

4.5.1 Running on Windows

The GUI was tested on Windows 10 running Python 3.8.6.

On Windows, after cloning the repository, change the directory to the web_gui directory.

```
git clone https://github.com/LinHuiqing/signo-lingo.git
cd signo-lingo
cd web_gui
```

Within the web_gui directory, configure the virtual environment, and install the dependencies. If your system is configured with the python3 or pip3 alias, change the commands accordingly.

```
python -m venv venv
venv\Scripts\activate
pip install -r requirements.txt
```

The terminal should resemble the following as shown in Figure 8:

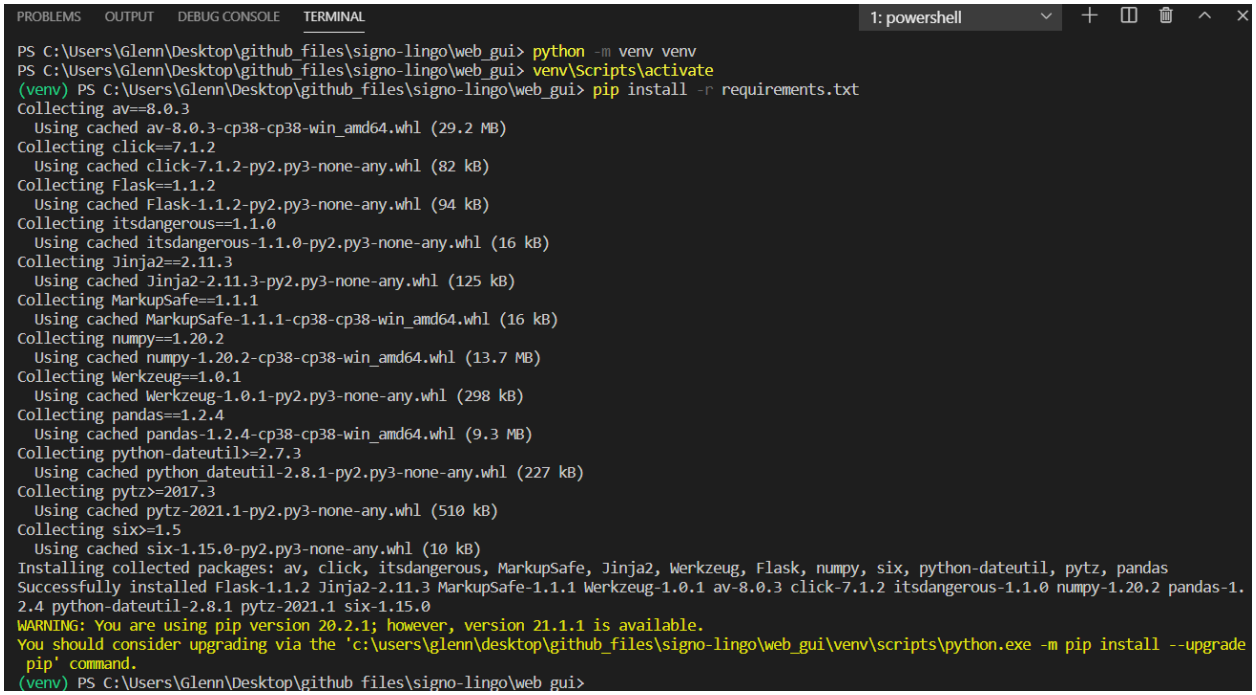

Figure 8. Configuring the python environment and installing dependencies

Note that Torch must be installed separately in another pip or pip3 command. In this system, we install Torch with CPU so that machines without CUDA are able to run the GUI. Visit https://pytorch.org/ and enter the machine configurations to retrieve the installation (refer to Figure 9). An example is shown below:

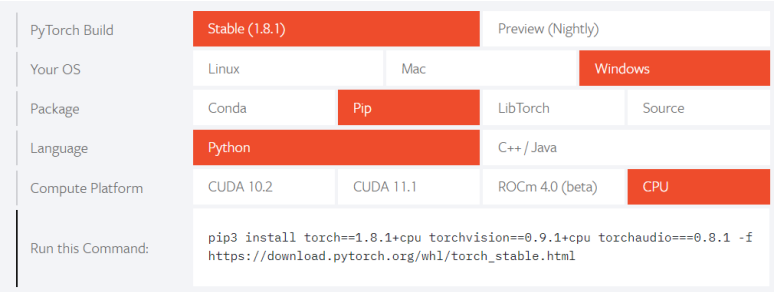

Figure 9. Screenshot of Torch Installation Machine Configurations

Then run the following command:

```
pip install install torch==1.8.1+cpu torchvision==0.9.1+cpu
torchaudio===0.8.1 -f https://download.pytorch.org/whl/torch_stable.html
```

The terminal should resemble the following as shown in Figure 10.

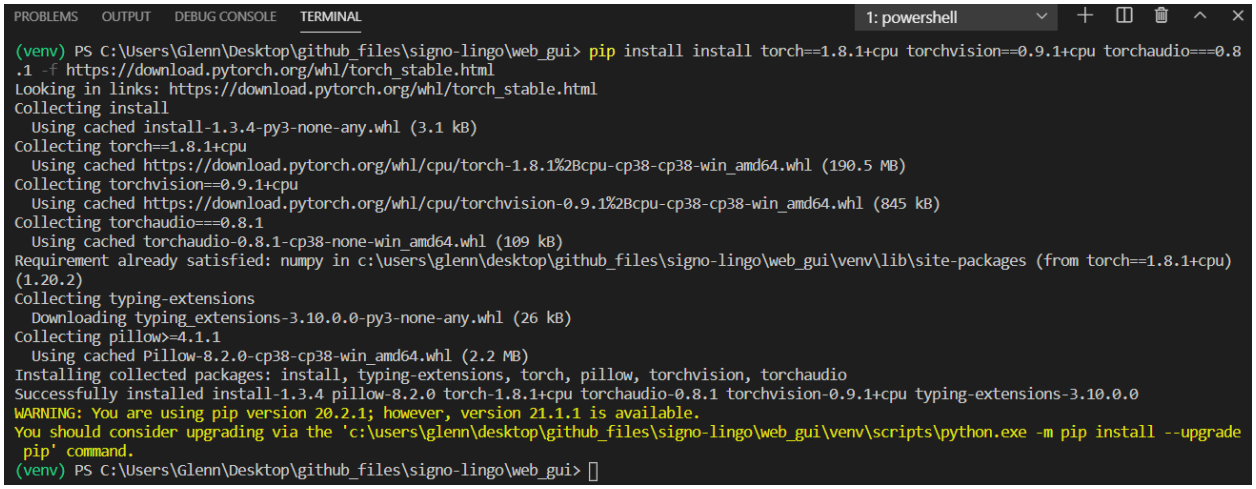

Figure 10. Install torch on windows

After the installations, you are now ready to run the GUI locally. Assuming you are in the web_gui directory, run the following:

```
cd src
flask run
```

If the installations were done correctly, flask will start successfully as shown in Figure 11.

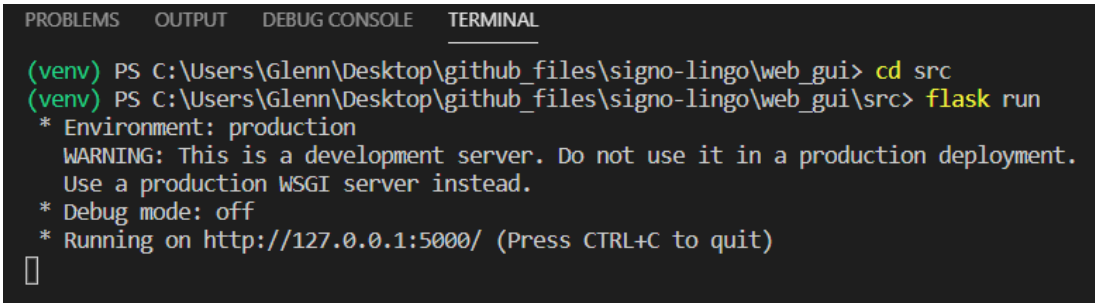

Figure 11. Running flask on windows

Open a web browser and visit http://localhost:5000. This brings up the GUI (see Figure 12). Click on “Choose File” to upload a file.

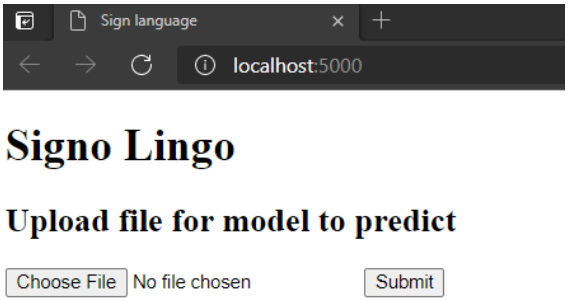

Figure 12. Signo lingo home page running locally

Test one of the sample videos found in the GitHub repository. These are samples taken from the dataset and also includes an example of one of our teammates. Note that these samples have to be formatted in Adobe Premiere Pro as there were issues displaying the videos in various web browsers. After formatting, the videos can be displayed. The “samples” directory provides several examples that can be displayed. Within the “samples” directory, there is the “others” directory which contains examples of misclassifications (see Figure 13). For this example, “signer6_sample86_color.mp4” was selected.

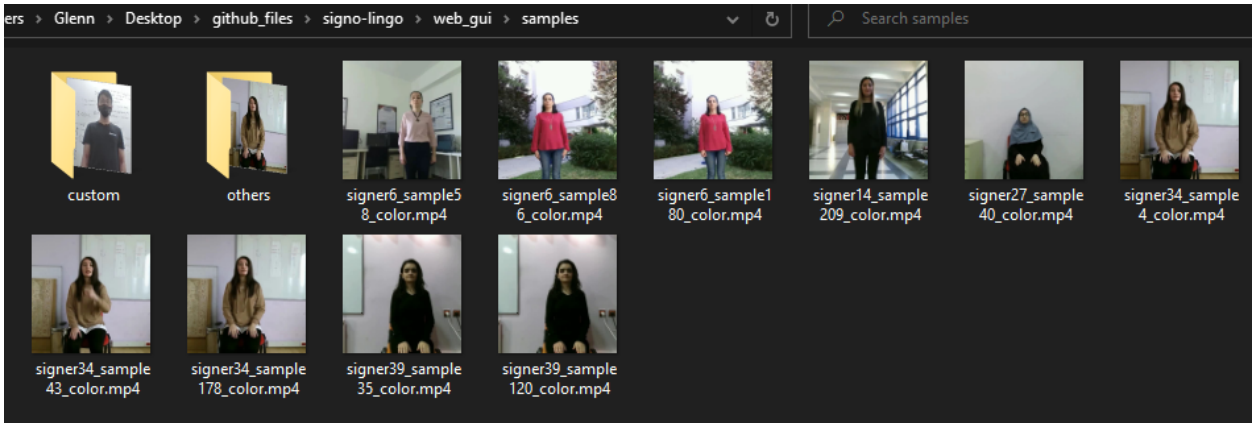

Figure 13. Select a video to upload to the GUI

After selecting the file, click “Submit” to send the video for prediction (see Figure 14).

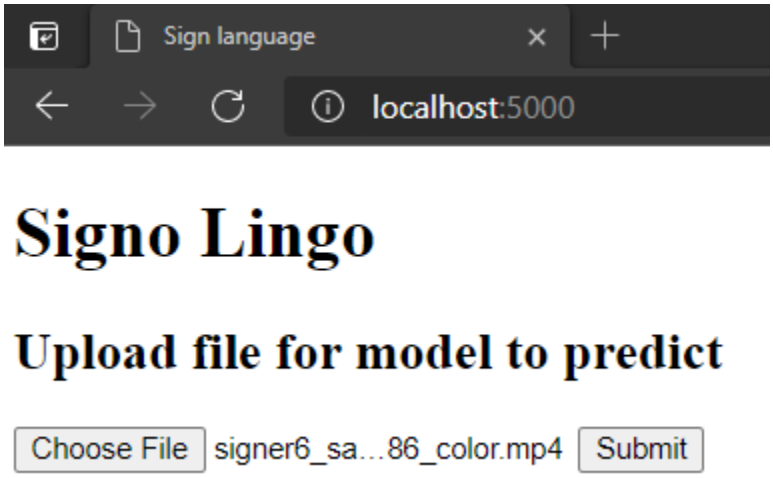

Figure 14. After choosing the file, click submit

The ground truth and predicted labels are displayed. In addition, the video can be played on the GUI. For this video, the ground truth is “married” and the predicted label is also “married”. The model has made a correct prediction. This is illustrated in Figure 15.

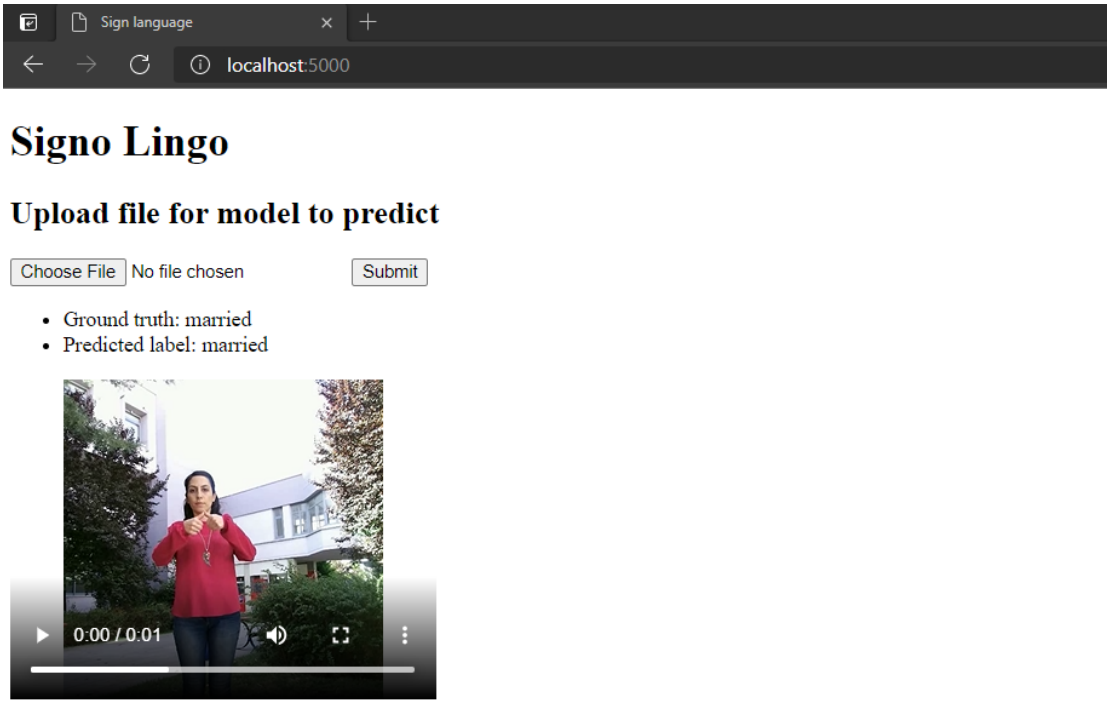

Figure 15. GUI displays the ground truth and prediction for data from the test set

4.5.2 Running on RHEL8

Install Python 3.8 (see Figure 16), create the virtual environment, and activate the environment, and install the dependencies (see Figure 17 and Figure 18).

```
sudo dnf install python3.8 -y
python3 -m venv venv
source venv/bin/activate
pip3 install -r requirements.txt
```

```
[root@ip-172-31-35-195 opt]# sudo dnf install python3.8 -y
Last metadata expiration check: 2:15:03 ago on Sun 02 May 2021 08:01:53 AM UTC.
Dependencies resolved.
================================================================================
 Package                    Architecture          Version
================================================================================
Installing:
 python38                   x86_64                3.8.3-3.module+el8.3.0+7680+79e
Installing dependencies:
 python38-libs              x86_64                3.8.3-3.module+el8.3.0+7680+79e
 python38-pip-wheel         noarch                19.3.1-1.module+el8.3.0+7187+a2
 python38-setuptools-wheel  noarch                41.6.0-4.module+el8.3.0+7187+a2
Installing weak dependencies:
```

Figure 16. Install python 3.8

```
[root@ip-172-31-35-195 opt]# python3 -m venv venv
[root@ip-172-31-35-195 opt]# source venv/bin/activate
(venv) [root@ip-172-31-35-195 opt]#
```

Figure 17. Create and activate the virtual environment

```
(venv) [root@ip-172-31-35-195 sign-language-ml-gui]# ls
LICENSE  README.md  requirements.txt  src
(venv) [root@ip-172-31-35-195 sign-language-ml-gui]# pip3 install -r requirements.txt
Collecting av==8.0.3
  Downloading https://files.pythonhosted.org/packages/d8/1a/0c76fb4fa6dd7b1bd8f4e74a76a
B)
     |                                | 38.2MB 289kB/s
Collecting click==7.1.2
  Downloading https://files.pythonhosted.org/packages/d2/3d/fa76db83bf75c4f8d338c2fd15c
     |                                | 92kB 15.8MB/s
Collecting Flask==1.1.2
  Downloading https://files.pythonhosted.org/packages/f2/28/2a03252dfb9ebf377f40fba6a78
     |                                | 102kB 9.0MB/s
Collecting itsdangerous==1.1.0
  Downloading https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d10
Collecting numpy==1.20.2
  Downloading https://files.pythonhosted.org/packages/75/f6/60e7d3a1da53a9979f37931d3cc
5.4MB)
     |                                | 15.4MB 45.1MB/s
Collecting pandas==1.2.4
  Downloading https://files.pythonhosted.org/packages/c8/3b/50dd12d901d5ec760d5ee169dfc
B)
     |                                | 9.7MB 53.1MB/s
Collecting Jinja2==2.11.3
  Downloading https://files.pythonhosted.org/packages/7e/c2/1eece8c95ddbc9b1aeb64f5783a
     |                                | 133kB 63.1MB/s
Collecting MarkupSafe==1.1.1
  Downloading https://files.pythonhosted.org/packages/80/16/98afa5c19296aaf7b16d1eb3c7e
1
Collecting Werkzeug==1.0.1
  Downloading https://files.pythonhosted.org/packages/cc/94/5f7079a0e00bd6863ef8f1da638
     |                                | 307kB 53.9MB/s
Collecting python-dateutil>=2.7.3
  Downloading https://files.pythonhosted.org/packages/d4/70/d60450c3dd48ef87586924207ae
     |                                | 235kB 59.2MB/s
Collecting pytz>=2017.3
  Downloading https://files.pythonhosted.org/packages/70/94/784178ca5dd892a98f113cdd923
     |                                | 512kB 49.4MB/s
Collecting six>=1.5
  Downloading https://files.pythonhosted.org/packages/ee/ff/48bde5c0f013094d729fe4b0316
Installing collected packages: av, click, MarkupSafe, Jinja2, Werkzeug, itsdangerous,
Successfully installed Flask-1.1.2 Jinja2-2.11.3 MarkupSafe-1.1.1 Werkzeug-1.0.1 av-8.0
WARNING: You are using pip version 19.3.1; however, version 21.1.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
(venv) [root@ip-172-31-35-195 sign-language-ml-gui]#
```

Figure 18. Install the base dependencies

Note that Torch must be installed separately in another pip or pip3 command. In this system, we install Torch with CPU so that machines without CUDA are able to run the GUI. Visit https://pytorch.org/ and enter the machine configurations to retrieve the installation (refer to Figure 19).

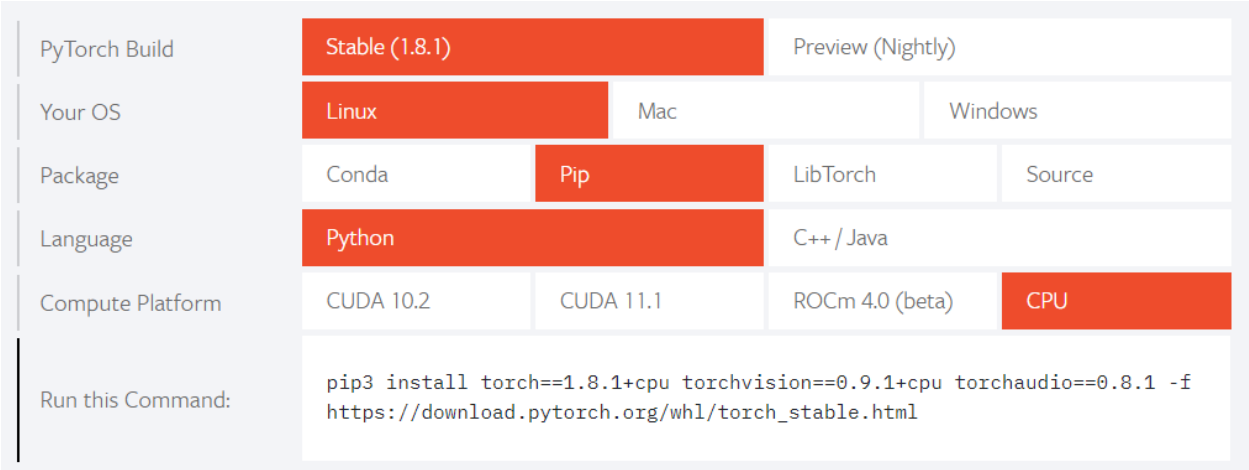

Figure 19. Install torch on linux

Then run the following command:

```
pip3 install torch==1.8.1+cpu torchvision==0.9.1+cpu torchaudio==0.8.1 -f
https://download.pytorch.org/whl/torch_stable.html
```

The terminal should resemble the following as shown in Figure 20.

Figure 20. Verify that torch installs

After the installations, you are now ready to run the GUI locally. Assuming you are in the web_gui directory, run the following:

```
cd src
flask run
```

If the installations were done correctly, flask will start successfully (see Figure 21).

```
(venv) [root@ip-172-31-35-195 src]# flask run
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Figure 21. Running Flask on linux

4.6 Accessing the GUI remotely

Installing Torch on some OSes and incompatible python versions can be challenging. Hence, we deployed the GUI on AWS. This GUI can be accessed via the following URL: http://3.1.85.200/. Note that the URL is using the **HTTP** protocol.

This brings up the homepage of the GUI (refer to Figure 22)

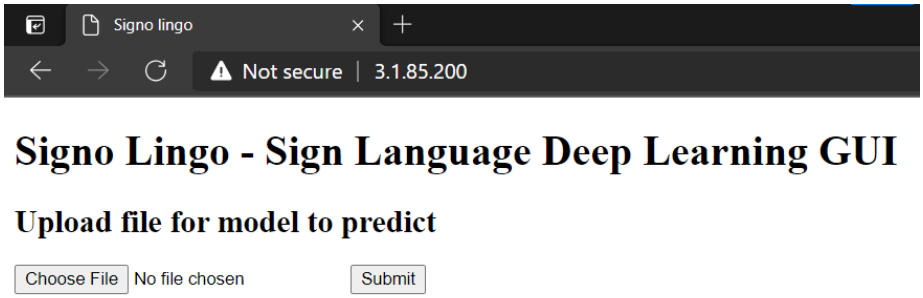

Figure 22. Screenshot of GUI Homepage

To test some of the sample videos and display them on the GUI, download the video samples from the GitHub repository (see Figure 23). An example is shown in Figure 24, where we download “signer27_sample40_color.mp4”.

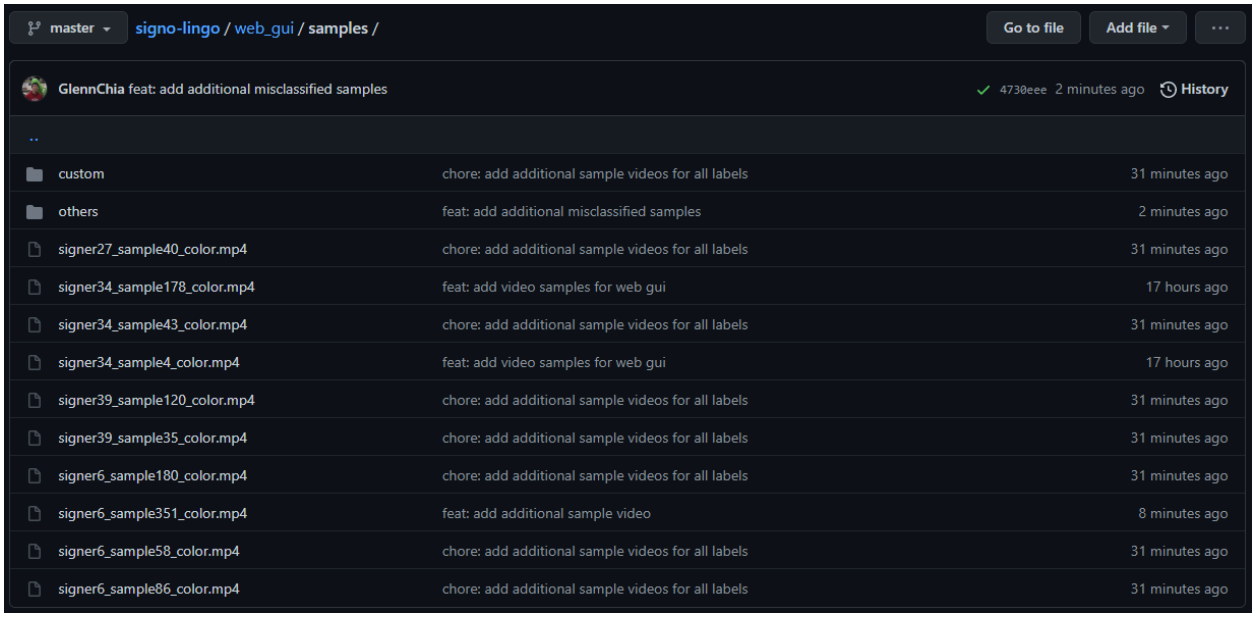

Figure 23. Sample videos that can be downloaded from the GitHub repository

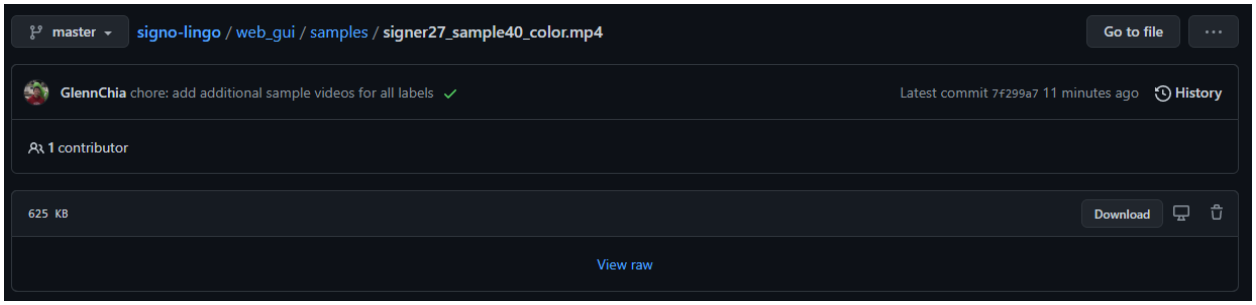

Figure 24. Selecting a video gives an option to download it

Back on the GUI, select a video to upload to the GUI by clicking on the “Choose File” button. For this example, “signer27_sample40_color.mp4” was selected as shown in Figure 25.

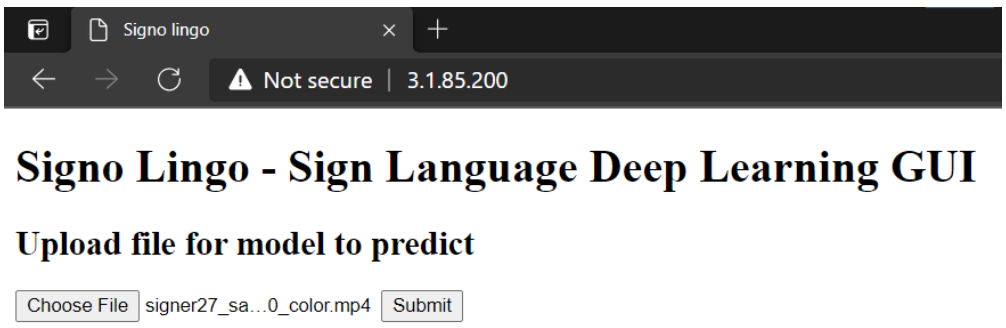

Figure 25. Selecting a video displays the name of the file

After selecting the file, click “Submit” to send the video for prediction. For this video, the ground truth is “potato” and the predicted label is also “potato”. The model has made a correct prediction. This is illustrated in Figure 26.

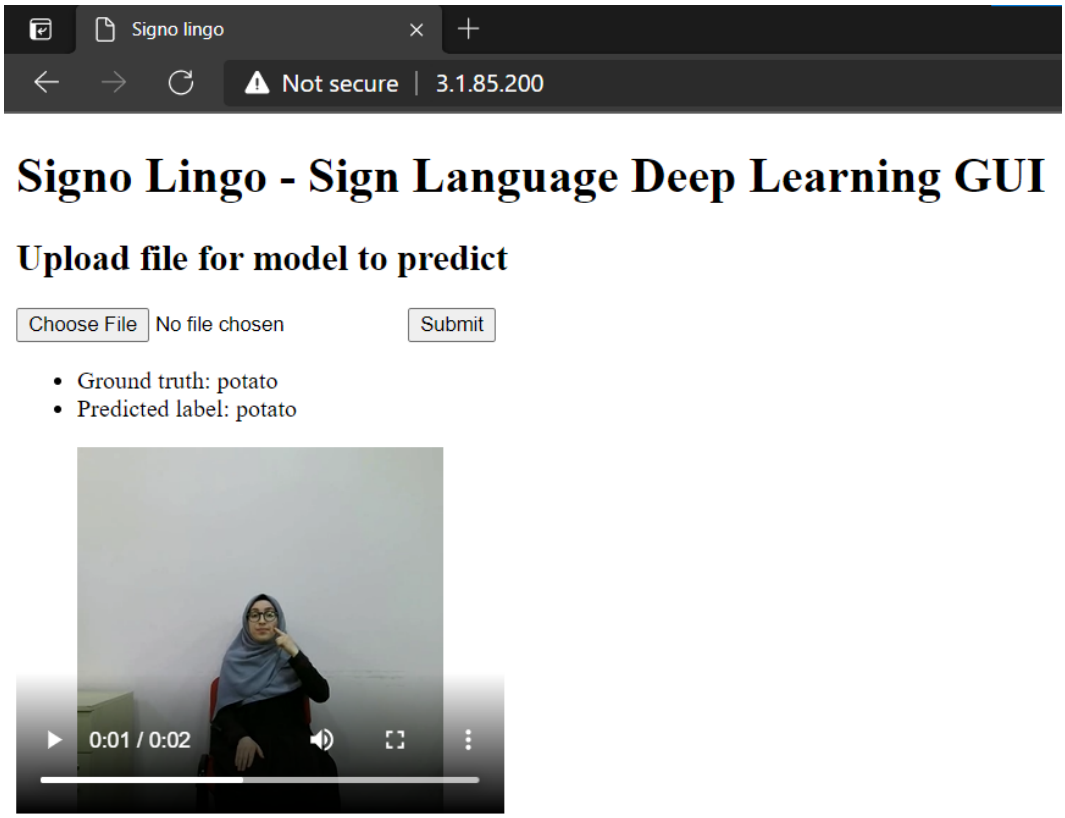

Figure 26. Model predicts correctly on the deployed gui

In addition, the GUI also accepts custom videos. Within the Github repository, there is a video called “xiangqian2.mp4” in the directory called “custom” (see Figure 27). Download this video.

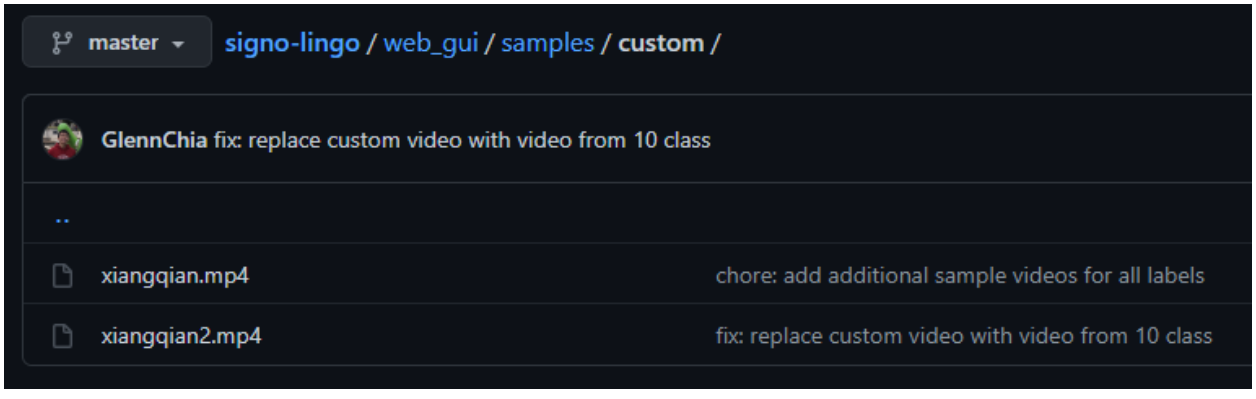

Figure 27. two custom videos of one of the team members can be downloaded from the custom directory on GitHub

Back on the GUI, select click on the “Choose File” button and select “xiangqian2.mp4” as shown in Figure 28.

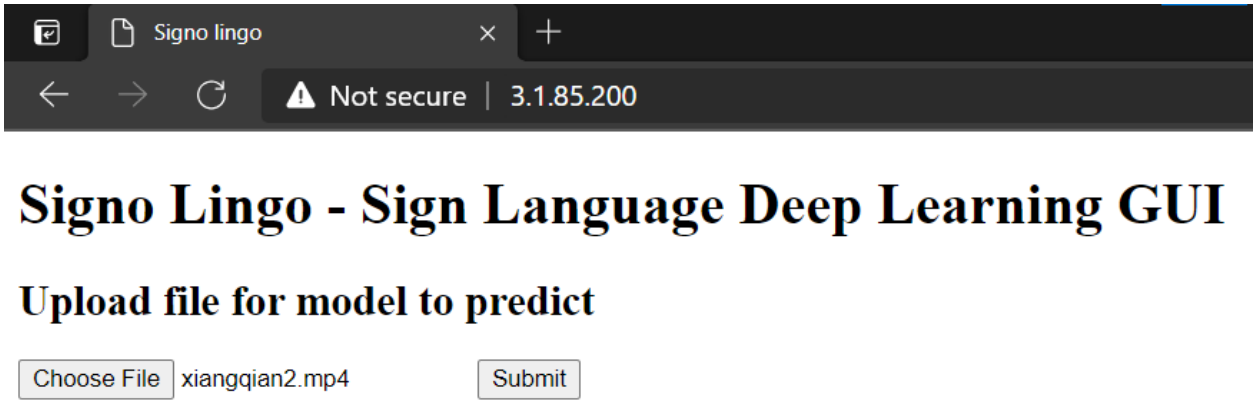

Figure 28. Select the custom video

In the GUI, select “xiangqian2.mp4”. This video is sent to the GUI for prediction. Based on our recording, the desired action was “child”, however, the model makes a mistake here and classifies it as “champion”. Since this is a custom data input, there is no mapping to the ground truth label and the GUI will display “This uploaded video was not from the dataset” (see Figure 29).

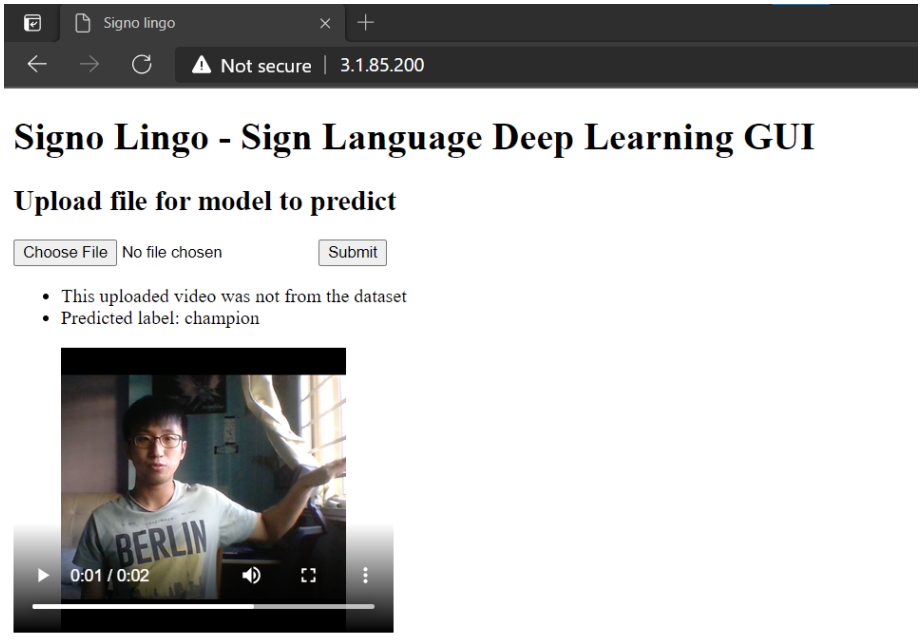

Figure 29. Model can perform a prediction but makes a wrong prediction

5 DATASET

5.1 Dataset Breakdown

This dataset is broken down into 3 folders - training (76% of dataset), validation (13% of dataset) and test (11% of dataset). Training and validation dataset will be used during experiments and training of models. The test dataset will be used in the evaluation result as it has the least data.

The labels for each folder are saved in the CSV files, each file named after its respective dataset (e.g. train.csv/ test.csv/ val.csv). SignList_ClassId_TR_EN.csv holds the information to convert a label number to its turkish and english meaning. filtered_ClassId.csv holds the 10 labels that will be used in this project.

5.2 Exploration

Diving deeper into the dataset, we performed exploratory data analysis in the AUTSL. This gives us a clearer picture on the type of data we are handling and the type of preprocessing that could be performed.

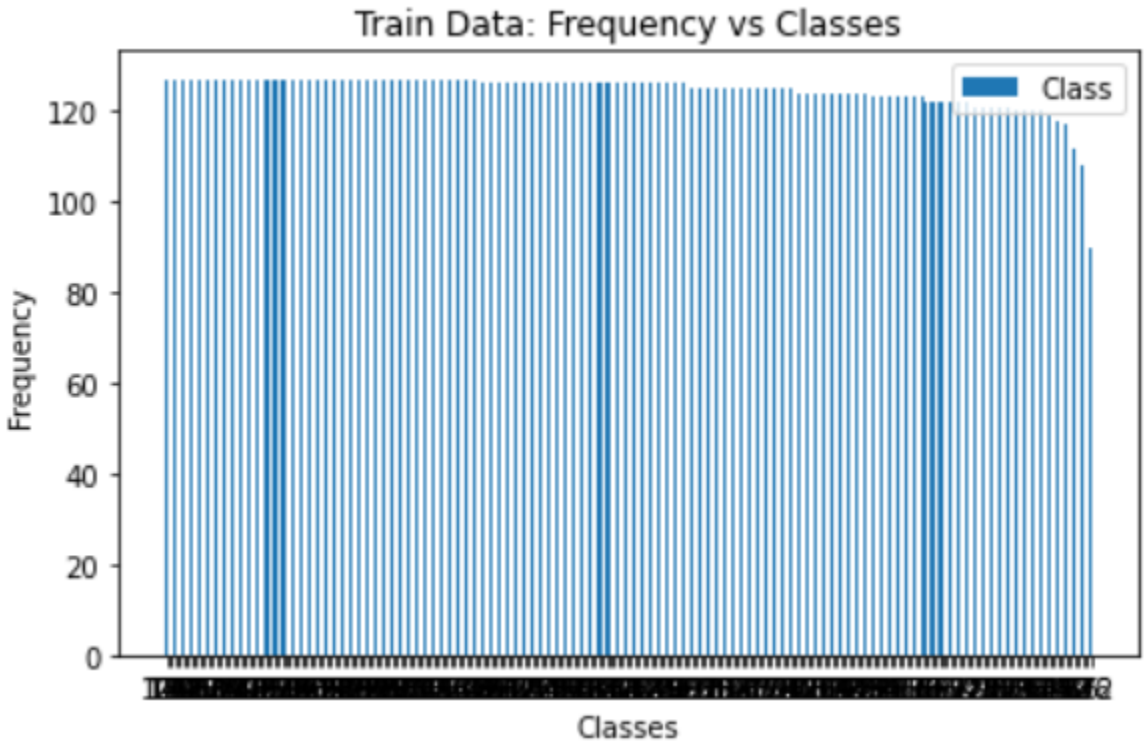

Figure 30. Train dataset: Labels against Frequency Graph, all classes

Figure 30 presents the entire dataset with x-axis as the classes and y-axis as the classes' frequency, or the number of times it appears in the dataset. It can be observed that the majority of the classes are evenly distributed with the exception of some classes that are found at the right tail of the figure with frequency below 110. Most classes have 127 videos in the dataset. Due to the limitations of the graph size, the 226 class labels on the x-axis are overlapping each other.

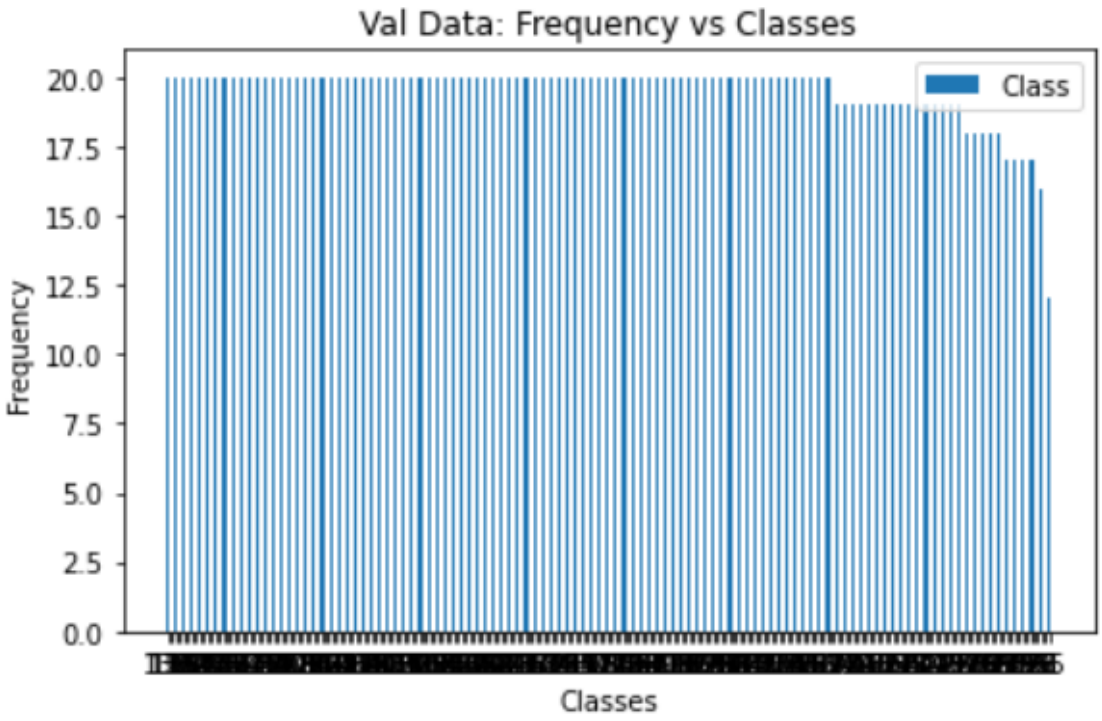

Figure 31. Validation dataset: Labels against Frequency Graph, all classes

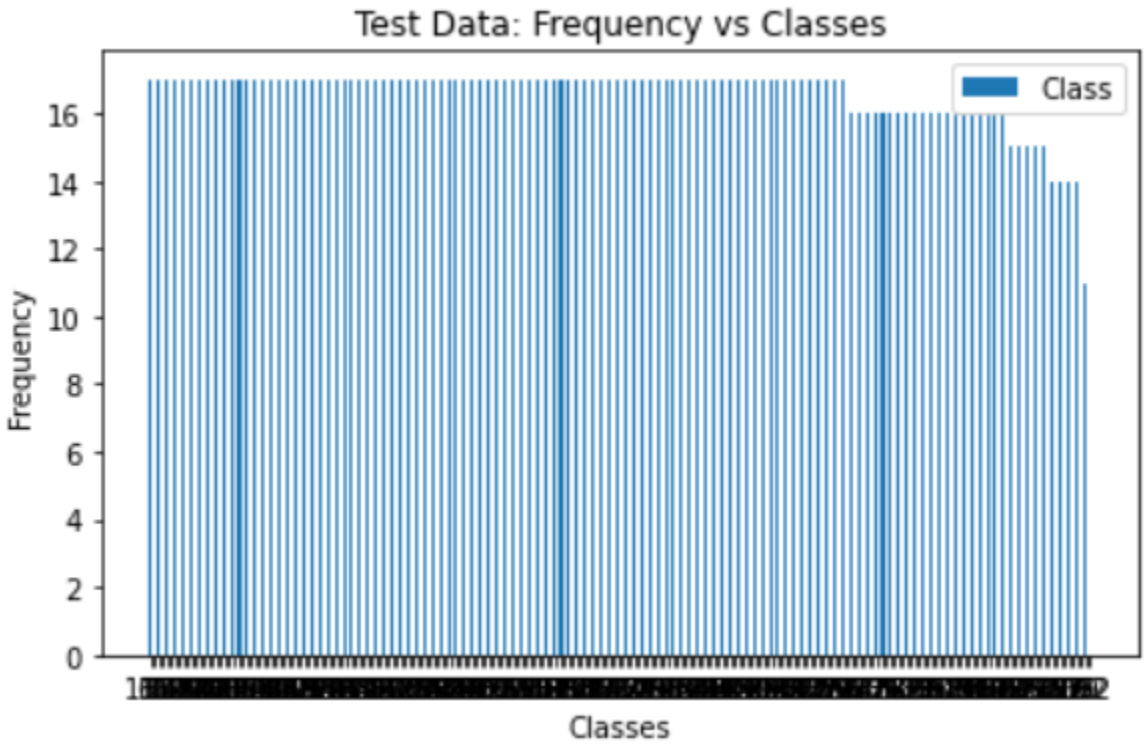

Figure 32. Test dataset: Labels against Frequency Graph, all classes

The same observation can be made in the validation dataset as seen in Figure 31, as well as in the test set, seen in Figure 32. Common validation frequency is around 20 and common test frequency is about 18.

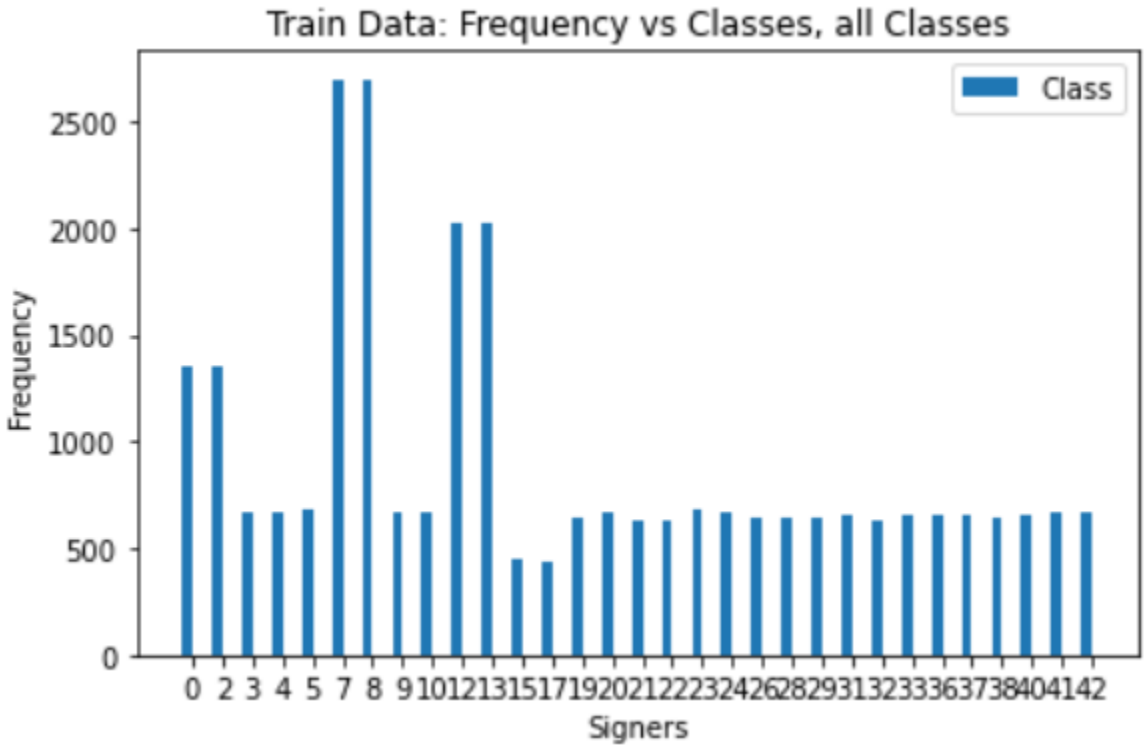

Figure 33. Train dataset: Signers against Frequency Graph, all classes

The videos performed by signers were investigated and categorized according to the signer's ID as seen in Figure 33. It is evident that signers 7 and 8 appeared the most number of times, followed by 12 and 13, and then 0 and 2. The rest of the signers have an even frequency distribution found in the train dataset.

Finally, the frame rate distribution of the train dataset was examined.

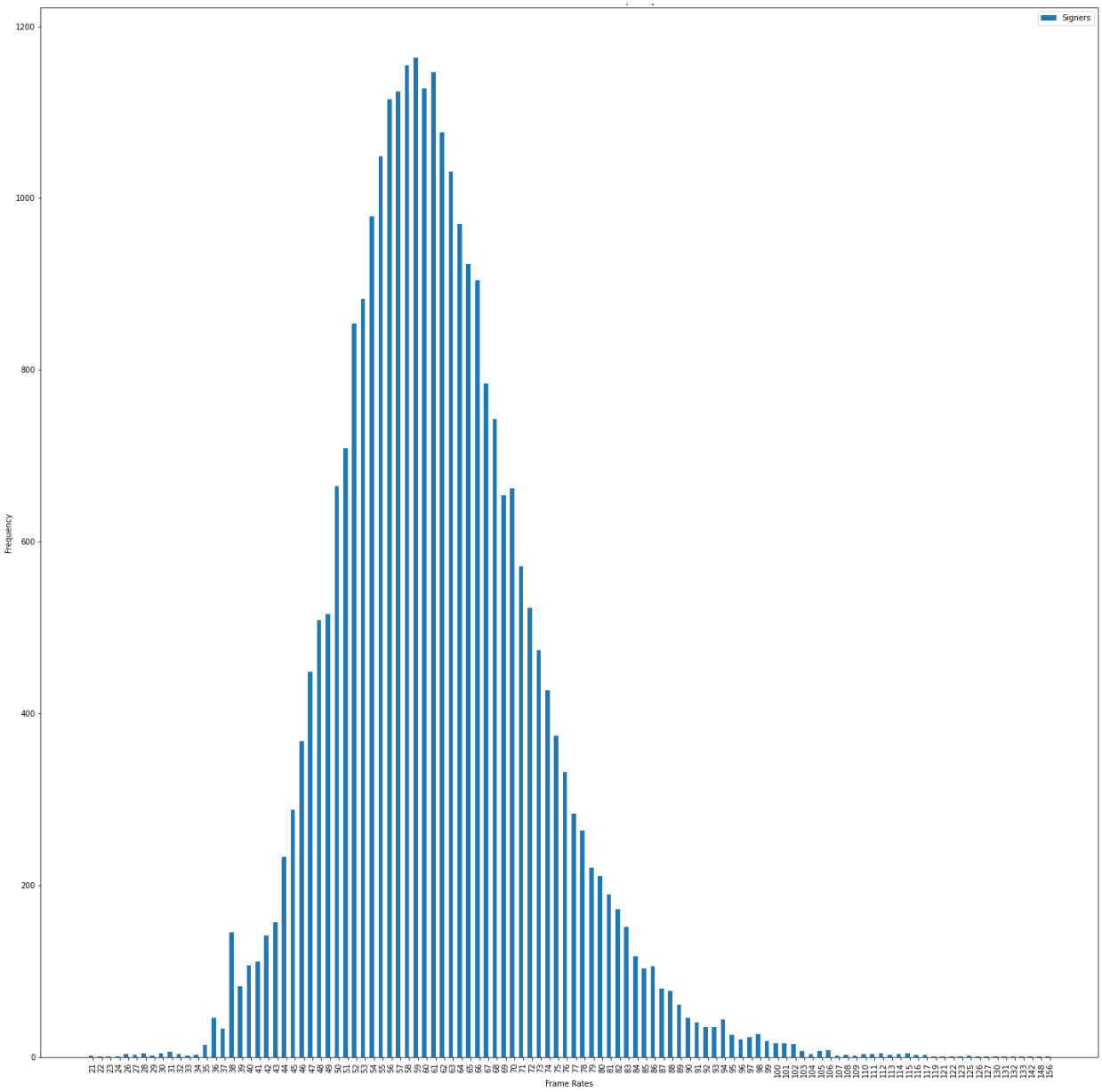

Figure 34. Train dataset: Graph of Frequency against Number of Frames, all classes

From Figure 34, there is a normal distribution of the train dataset, with the highest occurring number of frames at 59 frames. The x-axis is the number of frames and the y-axis is the number of videos that contain that particular frame rate.

5.3 Preprocessing

5.3.1 Fixing 30 Frames Per Video

From our initial data exploratory analysis, we notice that the video frames rates differ from videos to videos as seen in Figure 34. In addition, we are limited by the resources at hand, specifically

access to large GPU clusters. Having varying frame rates would cause the Nvidia GPU to raise memory error should it load a video with large frame rate during the training. This is an error we would like to avoid.

Hence, the number of frames per video is capped at 30 frames, for every video in training, validation, and test dataset. 30 frames are sufficient for forming a coherent short video that makes sense. To ensure that 30 frames for a video still captures the key actions of the sign language for classification, our team proceeded to analyse 30 frames of a sample video.

To do so, each video was opened with the av library and the number of frames in the video (`n_frames`) was extracted. After which, the regular intervals between frames (`interval`) were calculated with the total number of frames divided by the limit of frames set (`frames_cap`). To further space out the frames to extract, the remainder of this division is calculated as well (`remainder`). A variable is then set to keep track of the next frame to take from the video (`take_frame_idx`). For each frame number, if the frame number is not the same as `take_frame_idx`, the next frame is looked at instead. Otherwise, the frame is transformed and appended to an array (`vid_arr`), after which, the next frame's index is computed by adding the regular interval calculated, and if there is still `remainder`, the next frame's index is further incremented while the remainder is decremented. This process is repeated until all the frames are extracted. The code for this is as shown below.

```
def extract_frames(vid_path, transforms=None, frames_cap=None):
   """Extract and transform video frames

   Parameters:
   vid_path (str): path to video file
   frames_cap (int): number of frames to extract, evenly spaced
   transforms (torchvision.transforms, optional): transformations to apply to frame

   Returns:
   list of numpy.array: vid_arr

   """
   vid_arr = []
   with av.open(vid_path) as container:
       stream = container.streams.video[0]
       n_frames = stream.frames
       if frames_cap:
           remainder = n_frames % frames_cap
           interval = n_frames // frames_cap
           take_frame_idx = 0
           if interval < 1:
               raise ValueError(f"video with path '{vid_path}' is too short, please
make sure that video has >={frames_cap} frames")
       for frame_no, frame in enumerate(container.decode(stream)):
```

```
            if frames_cap and frame_no != take_frame_idx:
                continue
            img = frame.to_image()
            if transforms:
                img = transforms(img)
            vid_arr.append(np.array(img))
            if frames_cap:
                if remainder > 0:
                    take_frame_idx += 1
                    remainder -= 1
                take_frame_idx += interval
    return vid_arr
```

Figure 35. Fix 30 frame rate

From Figure 35, 30 frames are extracted from a video with 71 frames. The figure can be interpreted from left to right, top to bottom. Judging from the 30 frames, the series of actions looks plausible, similar to the video when it had 71 frames.

5.3.2 Choosing 10 Classes

On top of that, given the amount of time and scale of the problem, our team decided to pick 10 random classes out of the 226 given classes. Random sampling was performed so as to get a representative sample of the full dataset.

This is done programmatically by the documented code in `sample_data_classes.py`, and the code is explained below.

At the start of the script, arguments entered through the command line are parsed. These arguments are used to specify the number of classes to be filtered (`--n_classes`), the dataset directory (`--dataset_dir`), and the output directory of the processed CSVs (`--output_dir`).

```
parser = argparse.ArgumentParser(description='Decrease number of classes to a
chosen size.')
parser.add_argument('--n_classes',
                    type=int,
                    help='number to classes to filter to',
                    required=True)
parser.add_argument('--dataset_dir', nargs='?', default='dataset',
                    help='dataset directory')
parser.add_argument('--output_dir', nargs='?', default='data',
                    help='output directory for filtered csvs')
args = parser.parse_args()
```

If the specified output directory does not exist, it is created with the code below.

```
if not os.path.exists(args.output_dir):
   os.mkdir(args.output_dir)
```

The CSV with the different classes and their corresponding translations are then loaded through pandas.

```
class_df = pd.read_csv(f'{args.dataset_dir}/SignList_ClassId_TR_EN.csv')
u_len_label = len(class_df['ClassId'].unique())
print("total unique label:", u_len_label)
```

If the number of classes to be filtered is less than 1 or more than the total number of classes, an error is raised as it is an invalid input.

```
if args.n_classes < 1 or args.n_classes > u_len_label:
       raise ValueError(f"please use an int which is between 1 and the
number of all classes ({u_len_label})")
```

The following function is then defined to randomly choose the labels.

```
def choose_labels(class_id_df, n_classes, random_state=420):
    """ Chooses n_classes from all classes in class_id_df.

        Returns:
            class_id_df - dataframe of class ids
            n_classes - number of classes to filter
```

```
    """
    class_id_df =  class_id_df.sample(n_classes, random_state=random_state)
    return class_id_df, class_id_df["ClassId"].to_list()
```

With the help of the above function, the classes were then extracted and saved to a new CSV in the output directory as shown below.

```
class_df, sampled_class_ids = choose_labels(class_df, args.n_classes)
class_df = class_df.reset_index(drop=True)
class_df["oldClassId"] = class_df["ClassId"]
class_df["ClassId"] = class_df.index
class_df.to_csv(f"{args.output_dir}/filtered_ClassId.csv", index=False)
```

All the train/val/test sets were then loaded into data frames and saved into a dictionary.

```
df_store = {}
df_store["train"] = pd.read_csv(f'{args.dataset_dir}/train_labels.csv',
                                header=None)
df_store["val"] = pd.read_csv(f'{args.dataset_dir}/val_ground_truth.csv',
                              header=None)
df_store["test"] = pd.read_csv(f'{args.dataset_dir}/test_ground_truth.csv',
                               header=None)
```

Through a for loop, the data frames are looped through and the relevant examples are extracted. The new data frames are then saved into CSVs in the output directory specified.

```
for dataset_type, df in df_store.items():
   df = df[df[1].isin(sampled_class_ids)]
   df[1] = df[1].map(dict(zip(class_df["oldClassId"], class_df["ClassId"])))
   print(f"filtered {dataset_type} set has {len(df)} rows")
   df.to_csv(f"{args.output_dir}/{dataset_type}.csv",
             header=False,
             index=False)
```

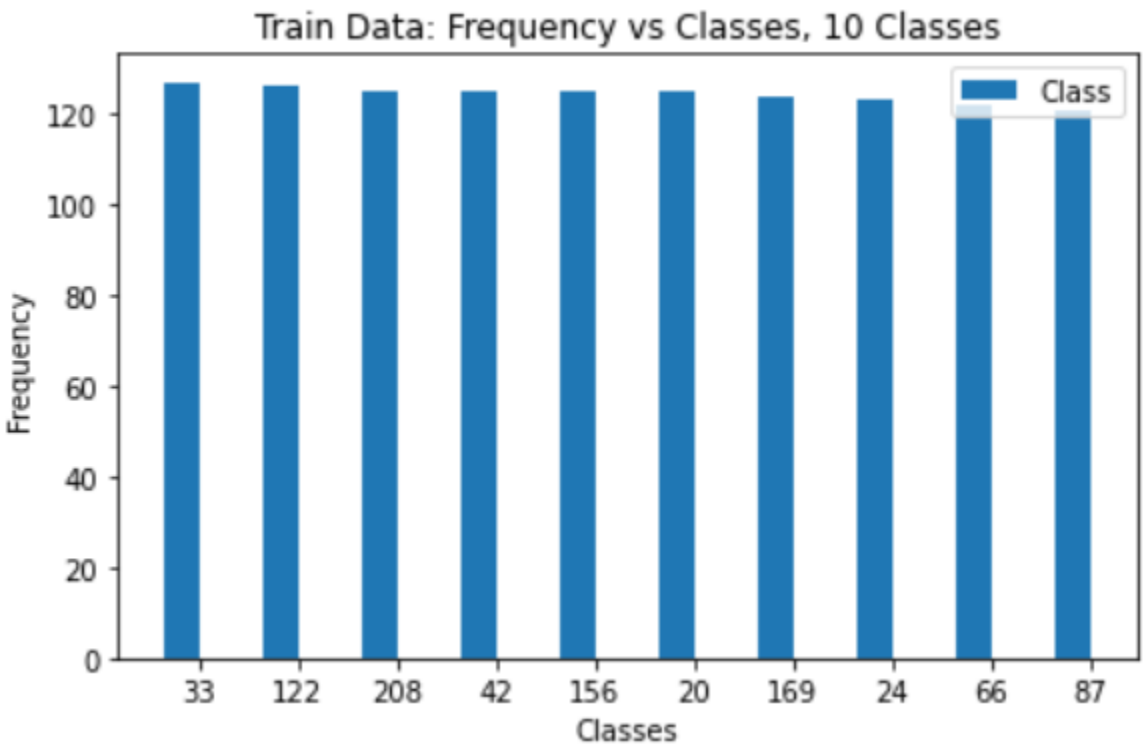

Figure 36. Train dataset: Class against Frequency Graph, Chosen 10 classes

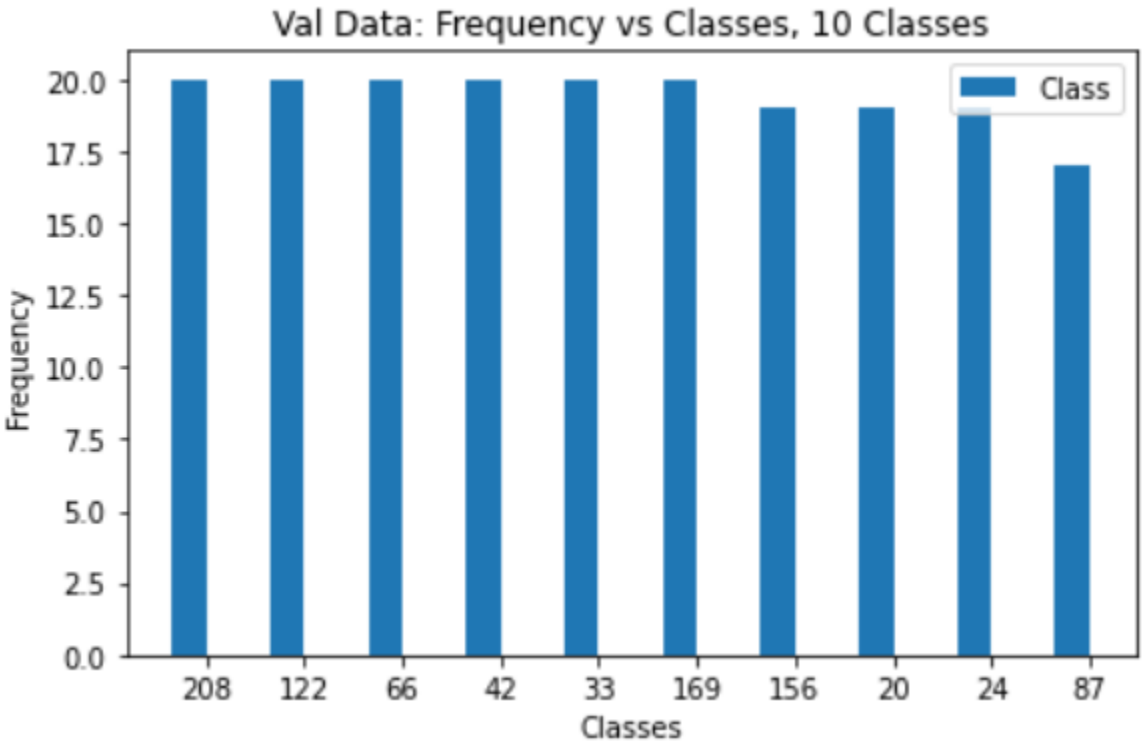

Figure 37. Validation dataset: Class against Frequency Graph, Chosen 10 classes

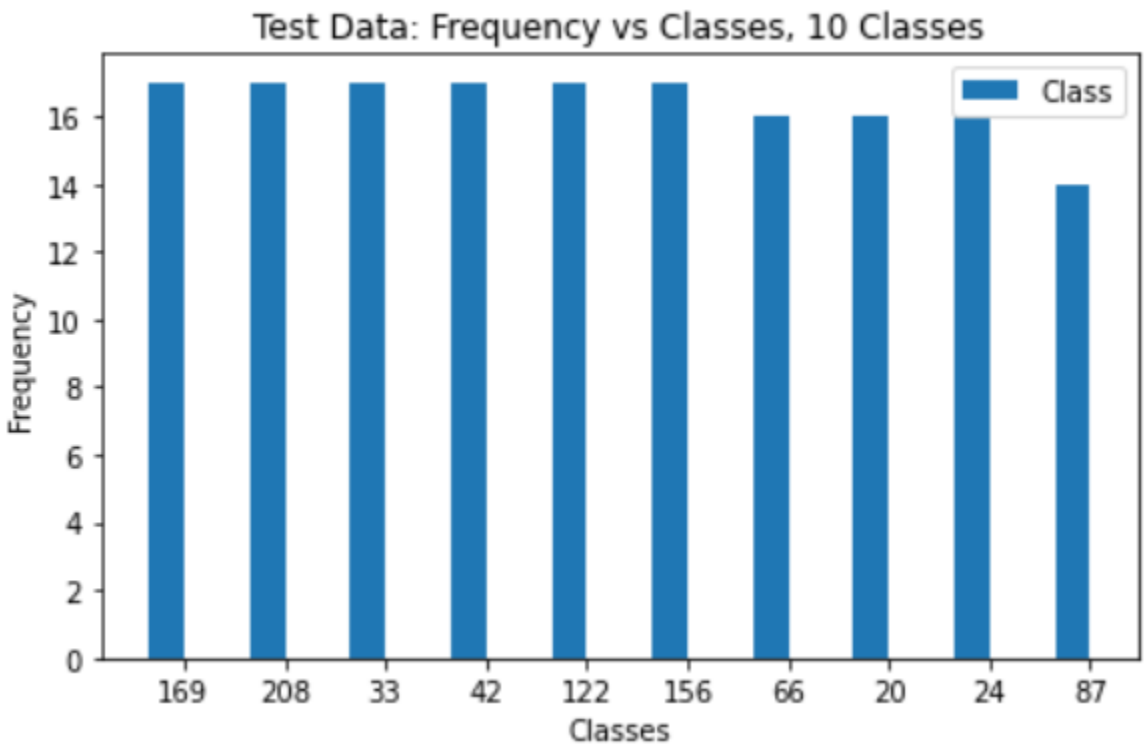

Figure 38. Test dataset: Class against Frequency Graph, Chosen 10 classes

From Figure 36, Figure 37 and Figure 38, the classes are largely evenly distributed for our team's chosen 10 classes for the train, validation, and test sets respectively. This ensures that the effect of class imbalance would be minimal, and that the dataset with 10 classes is representative of the full dataset.

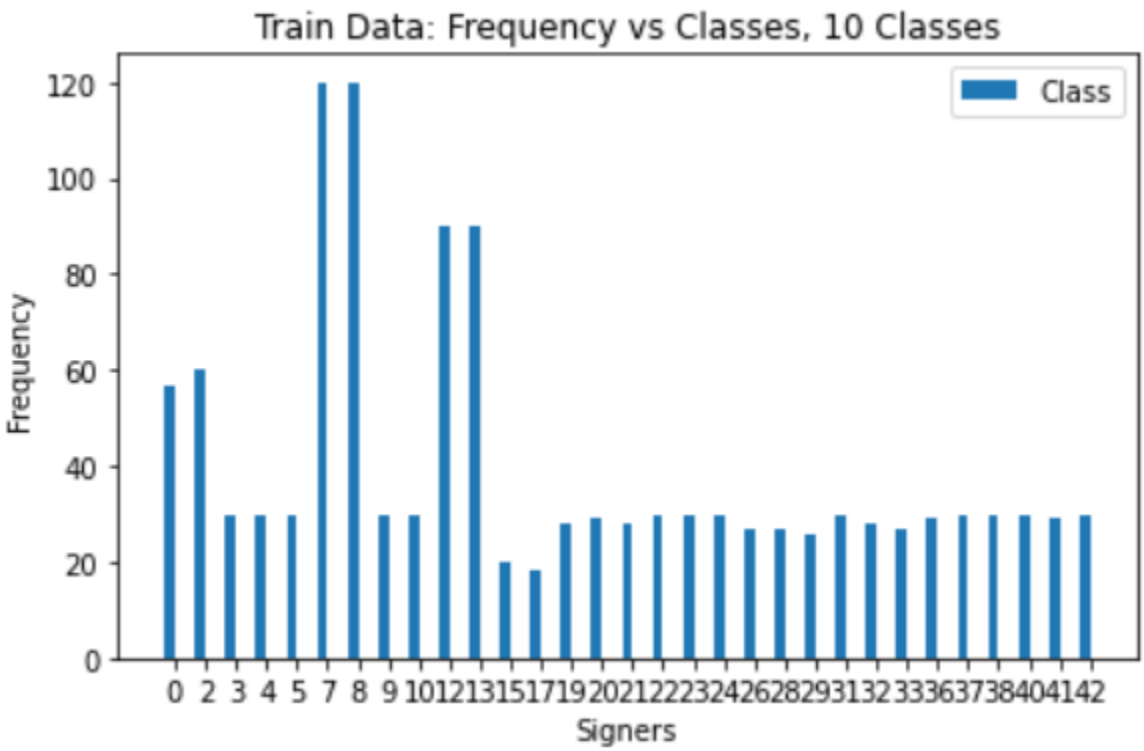

Figure 39. Train dataset: Signers against Frequency Graph, Chosen 10 classes

As seen in Figure 39, the distribution of signers in the dataset with the 10 sampled classes is of roughly the same distribution as the full dataset, as seen in Figure 33.

5.4 Data Augmentation

Since the dataset is based on the full-body movement that involves hands, arms, and face gestures, one possible data augmentation technique to explore is pose-estimation. Pose estimation involves finding points along the body that when connected with one another, forms an outline that looks like a skeleton of the human body. This could potentially aid the model in training as pose estimation captures key features of the human body necessary for classifying the actions of sign language.

Another possible data augmentation method is to use the depth images as masking on the RGB images. By masking the RGB, it removes background pixels from the original image. Masking could potentially improve training as it allows the model to focus on the important aspect, which is the signers themselves.

The combination of the two data augmentation techniques discussed above is illustrated in Figure 40 below. The background is replaced with black pixels as part of masking. The face, shoulders, arms and hands are outlined with pink lines and green dots.

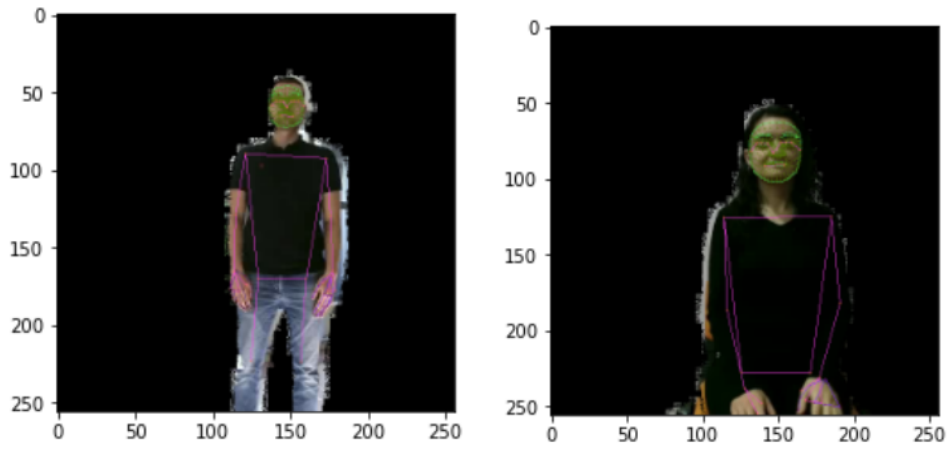

Figure 40. Pose Estimation with Masking Examples

6 MODEL

6.1 State-of-the-Art

Video frames can be extracted as static images to be input into CNNs before averaging the predictions. However, this naive approach does not fully encapsulate the complete storyline of the video and incomplete information can result in misclassifications. To overcome this limitation, feature pooling can be used to learn a global descriptor of a video's temporal evolution. Aside from its ability to overcome the vanishing or exploding gradient problem during backpropagation, RNNs such as Long Short Term Memory (LSTMs) that have a memory cell use frame-level sequences of CNN activations and can thus learn how to integrate information over time (Ng et al., 2015). Furthermore, both architectures maintain a constant number of parameters by sharing parameters over time.

Leveraging on the advantages of LSTM, long-term recurrent convolutional networks (LRCNs) that consist of convolutional layers and long-range temporal recursions have also been proposed as shown in Figure 41 (Donahue et al., 2015). In the case of sign language recognition (SLR), a film frame can be passed through an encoder or visual feature transformation parametrized by weight V, which involves the activations of some deep CNN layer. This creates a fixed-length latent representation that can be input into a decoder sequence model with parameter W. The gradients with respect to V and W can be updated using the back propagation in stochastic gradient descent (SGD) with momentum, thereby allowing the model to perform "end-to-end" learning, i.e. V can identify visual features that contribute to the sequential classification.

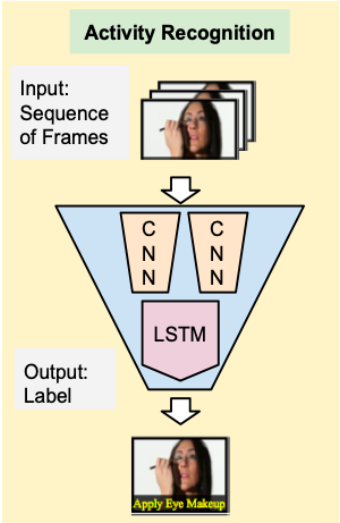

Figure 41. Brief Architecture of LRCN (Source: Donahue et al., 2015)

Research has found that batch-normalisation in the implementation of LSTM allows for faster convergence and better generalization, as compared to a normal LSTM model (Cooijmans et al., 2017). In standardizing activations using their mean and standard deviation, model parameters can learn based on varying input distributions and reduce internal covariate shift, whereby changing parameters of a layer affects the input distribution for layers above it.

Recently, a Graph Convolution Network (GCN) approach was adopted for SLR. This entails converting whole-body poses to reduced graphs that contain only the critical information needed for SLR (Figure 42). The use of pose estimation captures the interactions between a signer's hands and other body parts, and intricacies of body motion while making hand gestures. A multi-modal ensemble with 4 modes - skeleton, RGB, features, and optical flow - weighted based on ensemble validation accuracy, was found to give the best accuracy (Jiang et al., 2021).

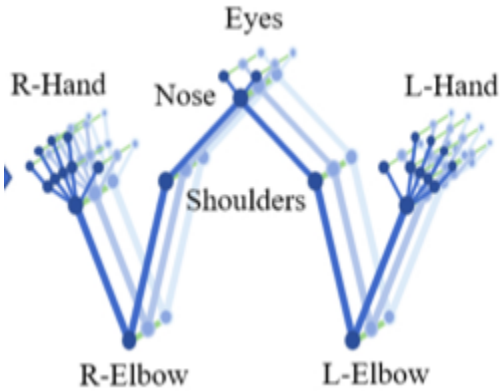

Figure 42. Pose Graph After Reduction to Key Nodes (Source: (Jiang et al., 2016))

6.2 Good Practices

6.2.1 Early Stopping

To determine when to stop the model from training, an early stopping module was added. This ensures the model only stops training when the early stopper detects overfitting, by checking that the validation loss has increased as epochs were trained, which is common practice (Stack Exchange - Data Science, 2018).

This is done by first setting the patience and delta values of choice. Patience is the consecutive number of epochs where the model's validation loss does not decrease below the minimum validation loss achieved, and the default for this was set as 10 because the common practice is to set it between 10 and 1000 (Stack Exchange - Cross Validated, 2016). The delta value is the buffer within which increases in validation loss is acceptable and the model would not be considered as overfitting.

At each epoch, the validation loss is passed into the `stop` function to check if the latest validation loss is above the threshold value. If so, increase `overfit_count` and print a message, otherwise, reset `overfit_count`. Then, update the minimum validation loss so far. After which, if the `overfit_count` is greater than or equals the `patience` value set, return `True`. Otherwise, return `False`.

Implemented code for early stopping is shown below. An object of this class is then initialized at the start of the training function, and then checked at the end of each epoch.

```
class EarlyStopping:
   """Keep track of loss values and determine whether to stop model."""
   def __init__(self,
                patience:int=10,
                delta:int=0,
                logger=None):
       self.patience = patience
       self.delta = delta
       self.logger = logger

       self.best_score = float("inf")
       self.overfit_count = 0

   def stop(self, loss):
       """Update stored values based on new loss and return whether to stop
model."""
       threshold = self.best_score + self.delta
```

```
        # check if new loss is mode than threshold
        if loss > threshold:
            # increase overfit count and print message
            self.overfit_count += 1
            print_msg = f"Increment early stopper to {self.overfit_count}
because val loss ({loss}) is greater than threshold ({threshold})"
            if self.logger:
                self.logger.info(print_msg)
            else:
                print(print_msg)
        else:
            # reset overfit count
            self.overfit_count = 0

        # update best_score if new loss is lower
        self.best_score = min(self.best_score, loss)

        # check if overfit_count is more than patience set, return value
accordingly
        if self.overfit_count >= self.patience:
            return True
        else:
            return False
```

6.2.2 Learning Rate

Based on the experimentation and the dataset's paper, it was found that the learning rate for this problem should be set to relatively low values for better training (Sincan & Keles, 2020). As such, the learning rate is set to start at 1e-5.

Adam is chosen as the choice of learning rate optimizer. Similar to RMSProp and Adadelta, Adam is able to compute adaptive learning rate for each parameter by storing an exponentially decaying average of past squared gradients. In addition, it keeps an exponentially decaying average of past gradients, similar to momentum (Ruder, 2016). These properties help the model to speed up learning when there is a long flat plateau while also helping the model to slow down when there is a sudden steep slope of gradient. Furthermore, it has been shown that the bias correction property in Adam allows it to perform slightly better than RMSProp towards the end of optimization when gradients are more sparse (Kingma & Ba, 2014).

The Adam function is then initialised in the train function as seen below, with `optimizer_lr` set as 1e-5 throughout the experiments detailed in this report.

```
optimizer = optim.Adam(model.parameters(),
                       lr=optimizer_lr,
                       weight_decay=weight_decay)
```

A learning rate scheduler was used as well to adjust the learning rate as the model trains (Lau, 2018). This is so that the model would be able to take smaller steps as the model gets closer to an optimum so that better weights can be learnt by the model over epochs. The `ReduceLROnPlateau` is used to decrease the learning rate when the validation loss plateaus. Patience is set at a third of the patience for the early stopper so as to allow the model ample time to improve its performance with decreased learning rate before the model training is stopped by the early stopper.

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                 patience=patience//3,
                                                 verbose=True)
```

6.2.3 Saving Checkpoints

Since each epoch takes a significant time to train, it will be costly to retrain a model from scratch should there be any issues. This was especially critical since half the team was training using the school's GPU cluster, which is known to have unpredictable availability issues. Hence, we implemented checkpoint saving to ensure that we could resume training for a partially trained model. Within this checkpoint, the most important variables to store would be the model weights, epoch number, optimizer state, and scheduler state since we elected to use a learning rate scheduler.

```
if save_checkpoint:
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'scheduler_state_dict': scheduler.state_dict(),
        'train_loss': train_loss,
        'val_loss': val_loss,
        'train_acc': train_acc,
        'val_acc': val_acc,
    }, f"{save_dir}/{epoch}-checkpoint.pt")
```

Within the training function, there is also a flag that users can configure if they want to load a checkpoint to resume training. This loads the four most important training variables.

```
if load_checkpoint and load_dir and load_epoch:
    # load model from checkpoint
    checkpoint = torch.load(f"{load_dir}/{load_epoch}-checkpoint.pt")
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    scheduler.load_state_dict(checkpoint['scheduler_state_dict'])
    start_epoch = checkpoint['epoch'] + 1
```

6.2.4 Graph Generation

Several experiments were conducted to optimise our proposed Deep Learning model (to be discussed in section 6.3.1 CNN Experiments). Log files were created to store the results for each experiment, and can be found in the ./src/dev_model/log. For data visualisation purposes, code was written to extract and plot the data from the log files to observe the effect of independent variables on model performance.

An `extract_numbers_from_file(filename)` function was first written to parse through a log file line by line and extract numbers based on the occurrence of keywords like “Training Loss” and “Validation Loss”. For example, if “Training Loss” appears in a line, split the line into a list and retrieve the values for epoch number, training loss and accuracy based on their index in the list. Knowledge on the indexes for these values can be assumed because the log files have been formatted the same way for different experiments. This function returns the number of epochs, training losses, training accuracies, validation losses and validation accuracies as separate lists.

```
def extract_numbers_from_file(filename):
    file = open(filename, "r")
    epochs = []
    train_losses = []
    train_accs = []
    val_losses = []
    val_accs = []
    for line in file.readlines():
        train = re.findall("Training Loss.*|$", line)
        if len(train) > 2:
            train = train[0].split()
            epoch_no = float(train[4][:-1])
            epochs.append(epoch_no)
            train_loss = float(train[5])
            train_losses.append(train_loss)
            train_acc = float(train[8][:-1])
            train_accs.append(train_acc)

        val = re.findall("Validation Loss.*|$", line)
```

```
        if len(val) > 2:
            val = val[0].split()
            val_loss = float(val[5])
            val_losses.append(val_loss)
            val_acc = float(val[8][:-1])
            val_accs.append(val_acc)
    return epochs, train_losses, train_accs, val_losses, val_accs
```

A second `plot_graphs(general_filepath, experiment_name)` function was written to show 4 subplots for training and validation losses and accuracies based on the numbers extracted from each log file. The function takes in the experiment name and general file name shared by the files belonging to the same experiment, to generate a list of log files to parse through using the above `extract_numbers_from_file(filename)` function. A dictionary stored within the function contains the legend labels for each experiment. Graphs for models with different parameters are superposed onto the 4 subplots for easy comparison.

```
def plot_graphs(general_filepath, experiment_name):
    dictionary = {'No. of Convo Layers in CNN': [2,3,4,5,6],
                  'Dropout Rate of Latent Vector': [0.1, 0.25, 0.50, 0.80],
                  'Weight Decay of Optimizer': [0, '1e^-5', '1e^-6',
'1e^-7', '1e^-8'],
                  'Attention (CNN-LSTM)': ['w/o attention', 'w/ attention'],
                  'Bidirectional': ['not bidirectional', 'bidirectional'],
                  'Attention (VGG11-LSTM)': ['w/o attention', 'w/
attention']}
    files = glob.glob(general_filepath, recursive = True)
    files = sorted(files)
    if experiment_name == 'Dropout Rate of Latent Vector':
        files = files[:4]
    elif experiment_name == 'Weight Decay of Optimizer':
        files = sorted(files[-4:]+[files[0]])
    elif experiment_name == 'Attention (CNN-LSTM)':
        files = files[:2]
    print(files)
    fig, axs = plt.subplots(2, 2, figsize = (15, 15))
    fig.suptitle(experiment_name, fontsize = 25, fontweight = 'bold')
    for file in files:
        index = files.index(file)
        legend = dictionary[experiment_name][index]
        epochs, train_losses, train_accs, val_losses, val_accs =
extract_numbers_from_file(file)
        axs[0, 0].plot(epochs, train_losses, label = legend)
        axs[0, 0].set_ylabel("average training loss", fontsize = 15)
```

```
        axs[0, 0].set_xlabel("epoch number", fontsize = 15)
        axs[0, 0].legend(loc="upper right")
        axs[0, 1].plot(epochs, train_accs, label = legend)
        axs[0, 1].set_ylabel("average training accuracy", fontsize = 15)
        axs[0, 1].set_xlabel("epoch number", fontsize = 15)
        axs[0, 1].legend(loc="upper right")
        axs[1, 0].plot(epochs, val_losses, label = legend)
        axs[1, 0].set_ylabel("average validation loss", fontsize = 15)
        axs[1, 0].set_xlabel("epoch number", fontsize = 15)
        axs[1, 0].legend(loc="upper right")
        axs[1, 1].plot(epochs, val_accs, label = legend)
        axs[1, 1].set_ylabel("average validation accuracy", fontsize = 15)
        axs[1, 1].set_xlabel("epoch number", fontsize = 15)
        axs[1, 1].legend(loc="upper right")
```

Full documentation with docstrings and comments can be found in the log_file_plots.ipynb python script.

6.3 Exploration

Based on state-of-the-art research, the CNN-LSTM architecture was chosen as a starting point due to the requirements of this project (e.g. needing to implement a CNN-RNN model), relative ease of implementation, and several advantages that both CNN and LSTM have which were explained earlier.

For the CNN architectures used, convolution blocks were used due to inspiration from state-of-the-art models such as ResNet. These consist of convolution layers, batch normalization, activation functions, and a max pool layer as seen in Figure 43. These blocks form the basis of our proposed CNN models.

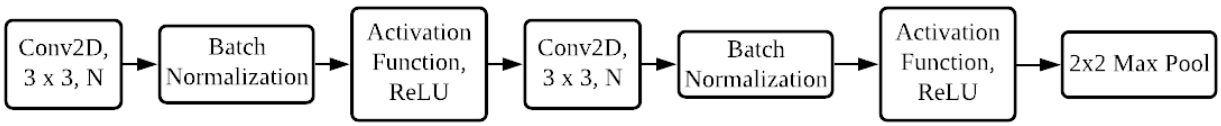

Figure 43. Structure of Convolution Block, Channel Size = N

As a starting point, a CNN-LSTM model with 2 convolution blocks and a single layer LSTM was constructed. The details of the architecture is as seen in Figure 44 below.

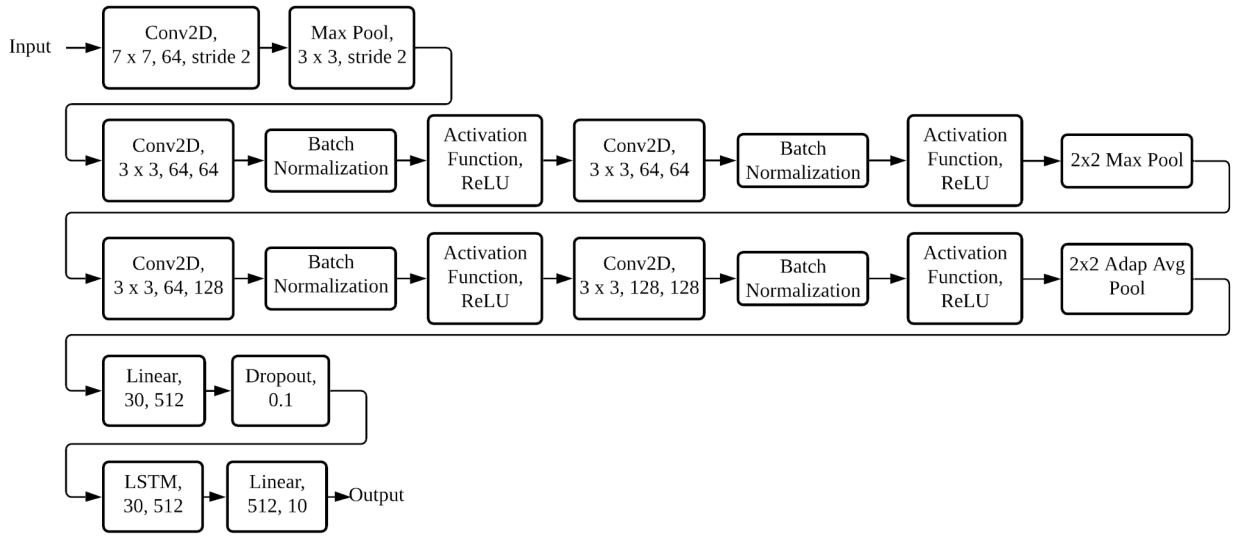

Figure 44. Architecture of Starting Model

Batch normalization layers are placed after convolution layers and before activation layers as it has been shown to allow for higher learning rates, more flexible initialization, and apply regularisation (Ioffe & Szegedy, 2015).

The activation function of ReLU is used as an entry point as it , with other options like LeakyReLU and PReLU coded to be easily included for experiments as well.

Max pooling is used between convolution blocks to better differentiate brighter and darker colours. Adaptive average pooling is used at the end of the convolution blocks to smooth out differences in values while keeping a consistent input size of the next fully connected layer.

Dropout is added after the encoder to ensure that all the features learnt by the encoder would be useful.

Note: All models trained in the following experiments are saved in this link: saved_models.

6.3.1 CNN Experiments

Mentioned earlier, experiments were conducted to decide an optimal design for our convolutional neural network (CNN). Log files used to extract the experimental results can be found in the ./src/dev_model/log directory.

6.3.1.1 Experiment 1: Effect of Number of Convolutional Layers

For this experiment, the number of convolution blocks were varied from 2 to 6 based on the architectures seen in Table 1. For all models, there were a few layers that were kept constant,

namely the first 7x7 convolution block, the 3x3 max pooling layer with stride 2, the last layers of average pooling and a 512 dimensional fully connected layer.

2-block	3-block	4-block	5-block	6-block
7x7, 64, stride 2				
3x3 max pool, stride 2				
$[\frac{3x3, 64}{3x3, 64}]\ x\ 1$	$[\frac{3x3, 64}{3x3, 64}]\ x\ 1$	$[\frac{3x3, 64}{3x3, 64}]\ x\ 1$	$[\frac{3x3, 64}{3x3, 64}]\ x\ 2$	$[\frac{3x3, 64}{3x3, 64}]\ x\ 2$
$[\frac{3x3, 128}{3x3, 128}]\ x\ 1$	$[\frac{3x3, 128}{3x3, 128}]\ x\ 1$	$[\frac{3x3, 128}{3x3, 128}]\ x\ 1$	$[\frac{3x3, 128}{3x3, 128}]\ x\ 1$	$[\frac{3x3, 128}{3x3, 128}]\ x\ 2$
	$[\frac{3x3, 256}{3x3, 256}]\ x\ 1$	$[\frac{3x3, 256}{3x3, 256}]\ x\ 1$	$[\frac{3x3, 256}{3x3, 256}]\ x\ 1$	$[\frac{3x3, 256}{3x3, 256}]\ x\ 1$
		$[\frac{3x3, 512}{3x3, 512}]\ x\ 1$	$[\frac{3x3, 512}{3x3, 512}]\ x\ 1$	$[\frac{3x3, 512}{3x3, 512}]\ x\ 1$
average pool, 512-d fc				

Table 1. Architecture of CNN models with Different No. of Convolution Blocks

First, the number of convolutional blocks was increased from 2 to 6, and the training and validation accuracies and losses were recorded (refer to Figure 45). While the CNN with 3 layers achieved the lowest validation loss, it achieves the highest training loss, suggesting that the model was not training. Although the CNN with 6 layers might not have resulted in the lowest validation loss, it achieved the lowest training loss and highest training and validation accuracies. As such, 6 convolutional layers were used in the subsequent experiments.

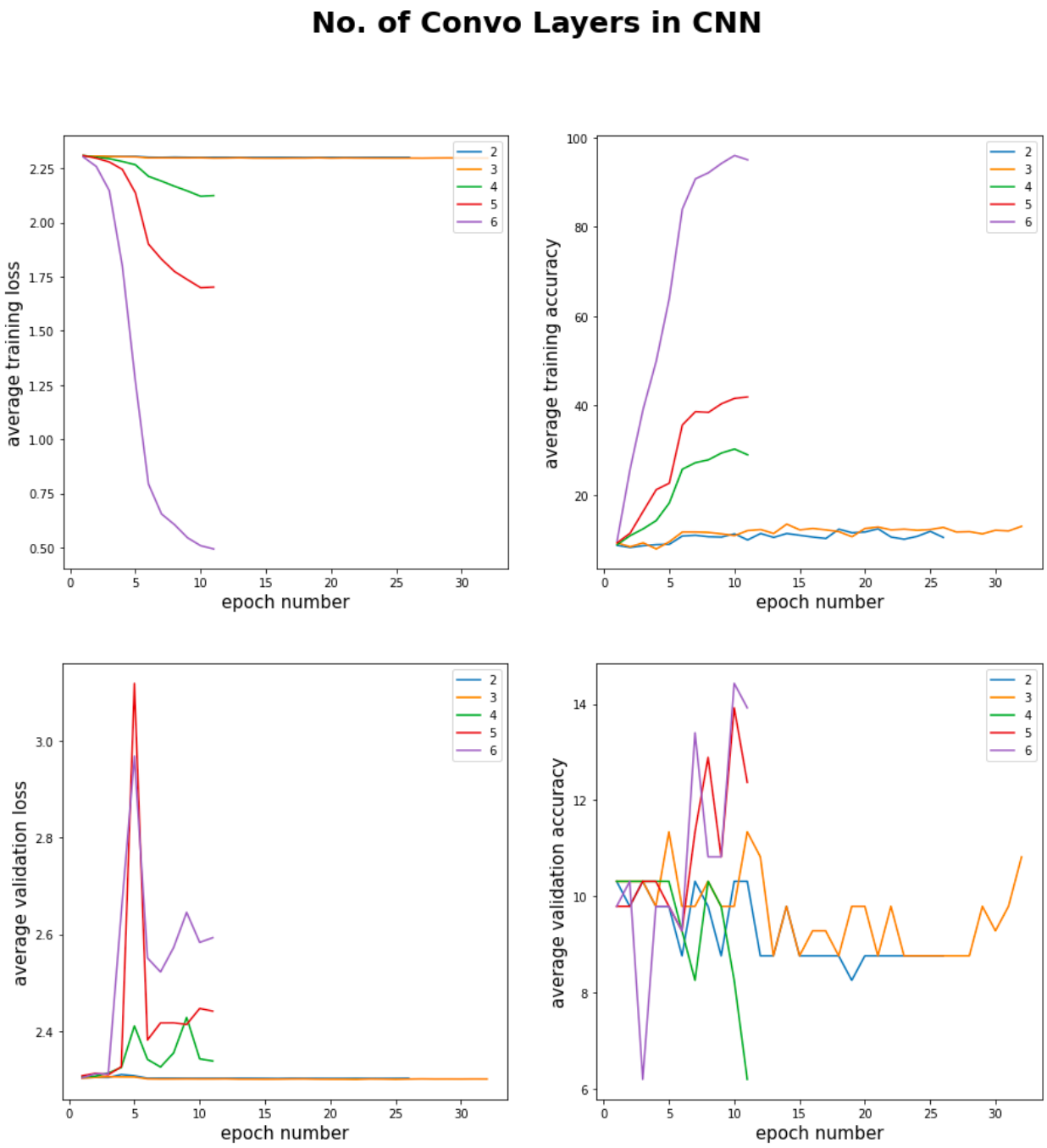

Figure 45. Graphs for Experiment on Varying Number of Convolutional Layers in CNN

It can be observed from the graphs, however, that overfitting occurs since training loss is still on a decreasing trend when validation loss is increasing. This demands the need for regularisation through methods such as dropout and L2 regularisation. Specifically, two other experiments were carried out to vary regularisation, which prevents overfitting: the first varied the dropout rate of the latent vector, the second changed the weight decay of the optimizer.

6.3.1.2 Experiment 2: Effect of Latent Vector Dropout Rate

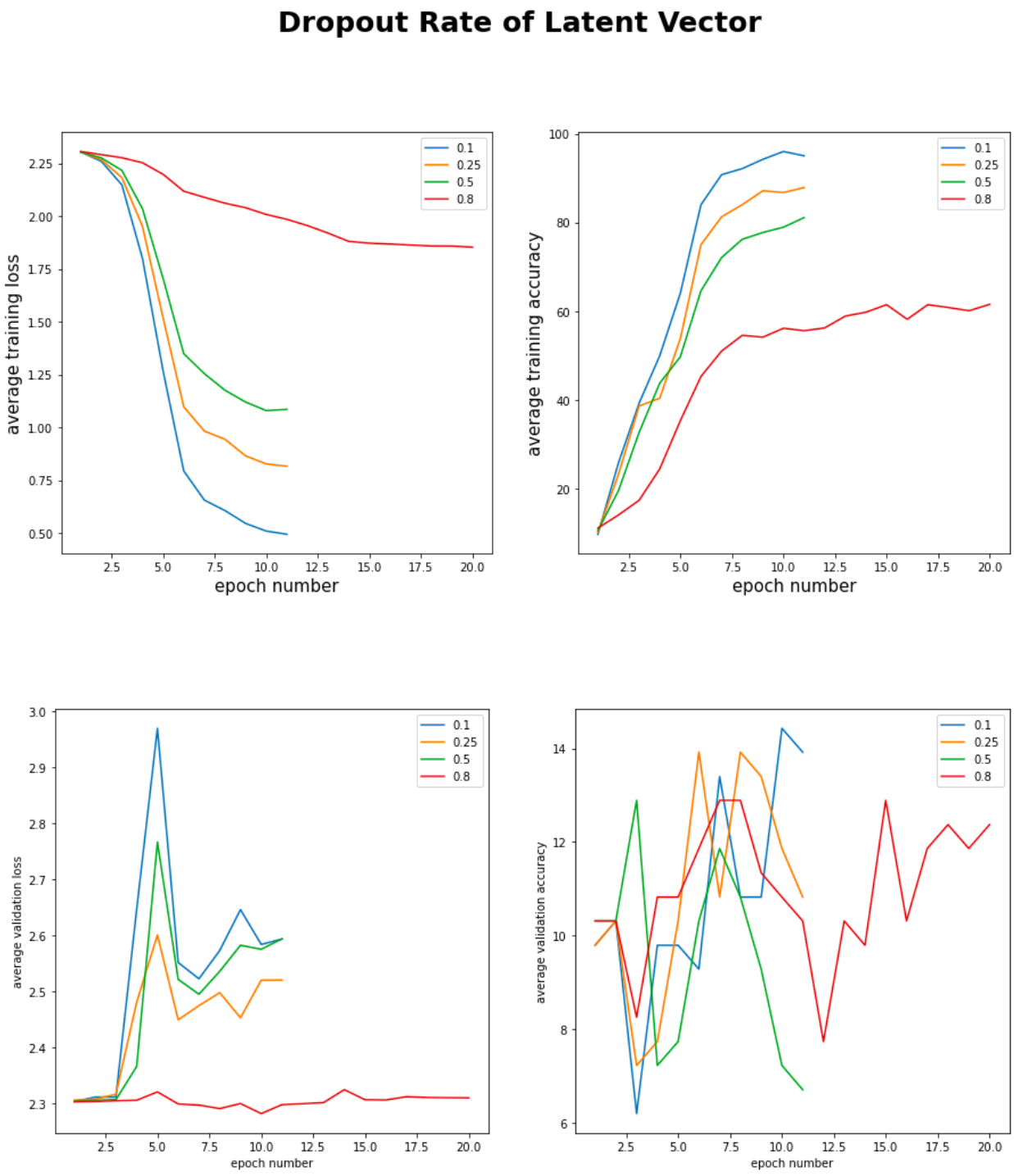

Figure 46. Graphs for Experiments on Changing Dropout Rate of Latent Vector

The dropout rate of the latent vector specifies the probability to keep a neuron active. Early stopping was implemented for the model to stop training when the losses do not decrease further. While a dropout rate of 0.8 produces the highest average training loss and lowest training accuracy, it achieves the lowest validation loss. Thus, a dropout rate of 0.80 was used for the final experiment as shown in Figure 46.

6.3.1.3 Experiment 3: Effect of Weight Decay of Optimizer

In the third experiment, the learning optimizer with a weight decay of 1^e-7 achieved the lowest validation loss as depicted in Figure 47. However, the effect of weight decay seems to be negligible with respect to when no weight decay was used. As such, weight decay was not used in further experiments.

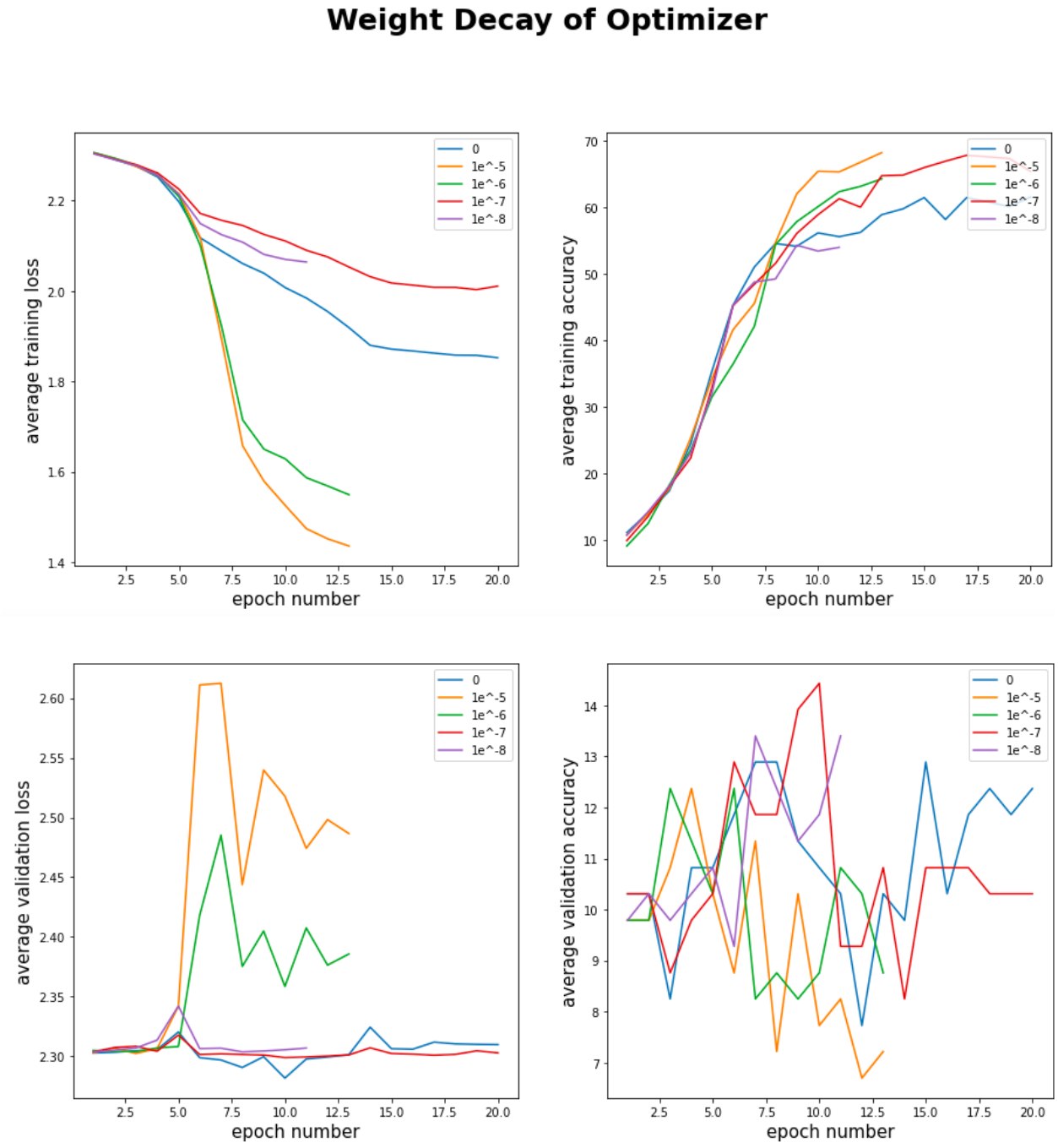

Figure 47. Graphs for Experiments on Varying Weight Decay of Optimizer

6.3.2 RNN Experiments

A series of experiments was also conducted on the RNN part of the model to further improve the performance of the model.

6.3.2.1 Experiment 1: Effect of Attention

An attention layer was first implemented because it enhances the important parts of the data and fades out the less important parts. This ensures that compute power can be devoted to the most important parts of the data. The effect of the attention layer on performance can be observed in Figure 48.

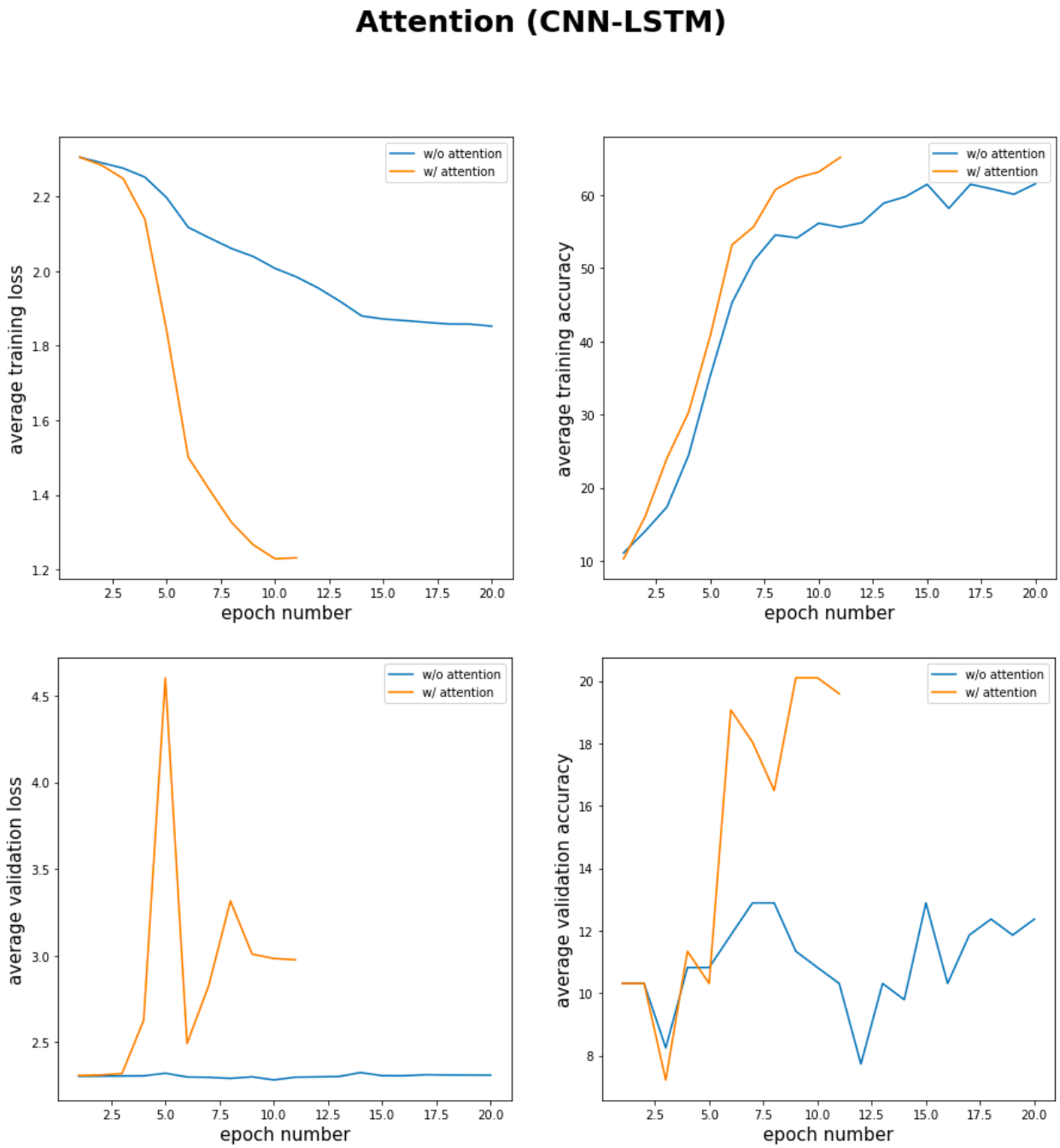

Figure 48. Experimental results of attention layer

Although the training loss is drastically decreased with the attention layer, validation loss is higher. The addition of the attention layer is found to increase both training and validation accuracy. As such, the attention layer will be included.

6.3.2.2 Experiment 2: Effect of Bidirectional LSTM

The effect of bidirectional LSTM on model performance was also measured.

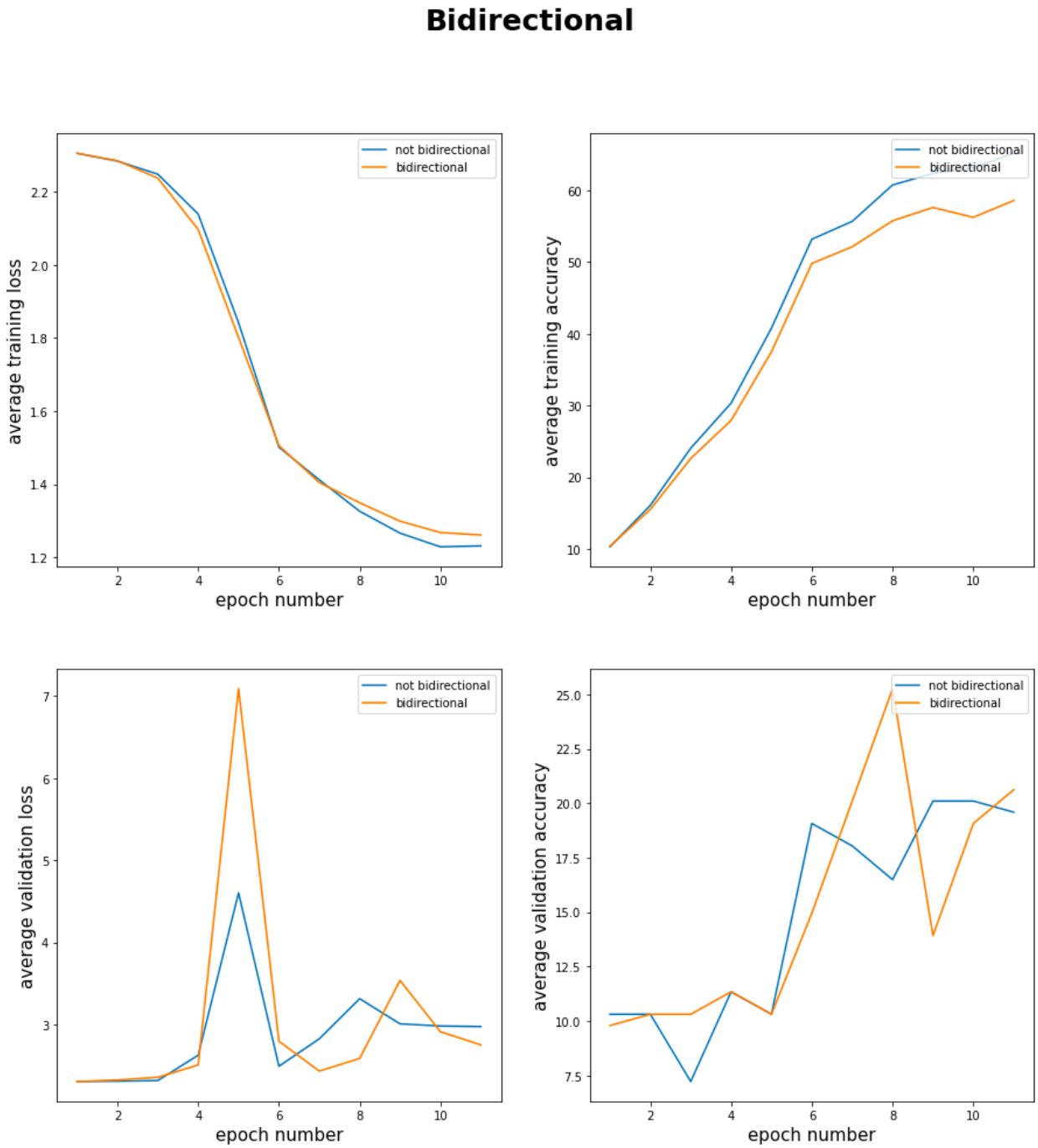

Figure 49. Experimental results of bidirectional LSTM

Based on the results plotted in Figure 49, contrary to what was expected, the model with the LSTM that is not bidirectional obtained lower training loss and higher training accuracy. However, as the bidirectional LSTM achieves the lowest validation loss and higher validation accuracy, it is implemented in our final model.

6.3.3 Pretrained + RNN/LSTM

Finally, the CNN portion of the model was replaced with a pretrained VGG with 11 layers to test if it would be better at extracting more meaningful features from the frames. The hypothesis was that VGG11, being trained on a larger dataset should be able to perform better for feature extraction. VGG11 was chosen instead of the original VGG16 used in the paper as we are training on a small dataset with only 10 classes, which larger models like VGG16 are likely to overfit.

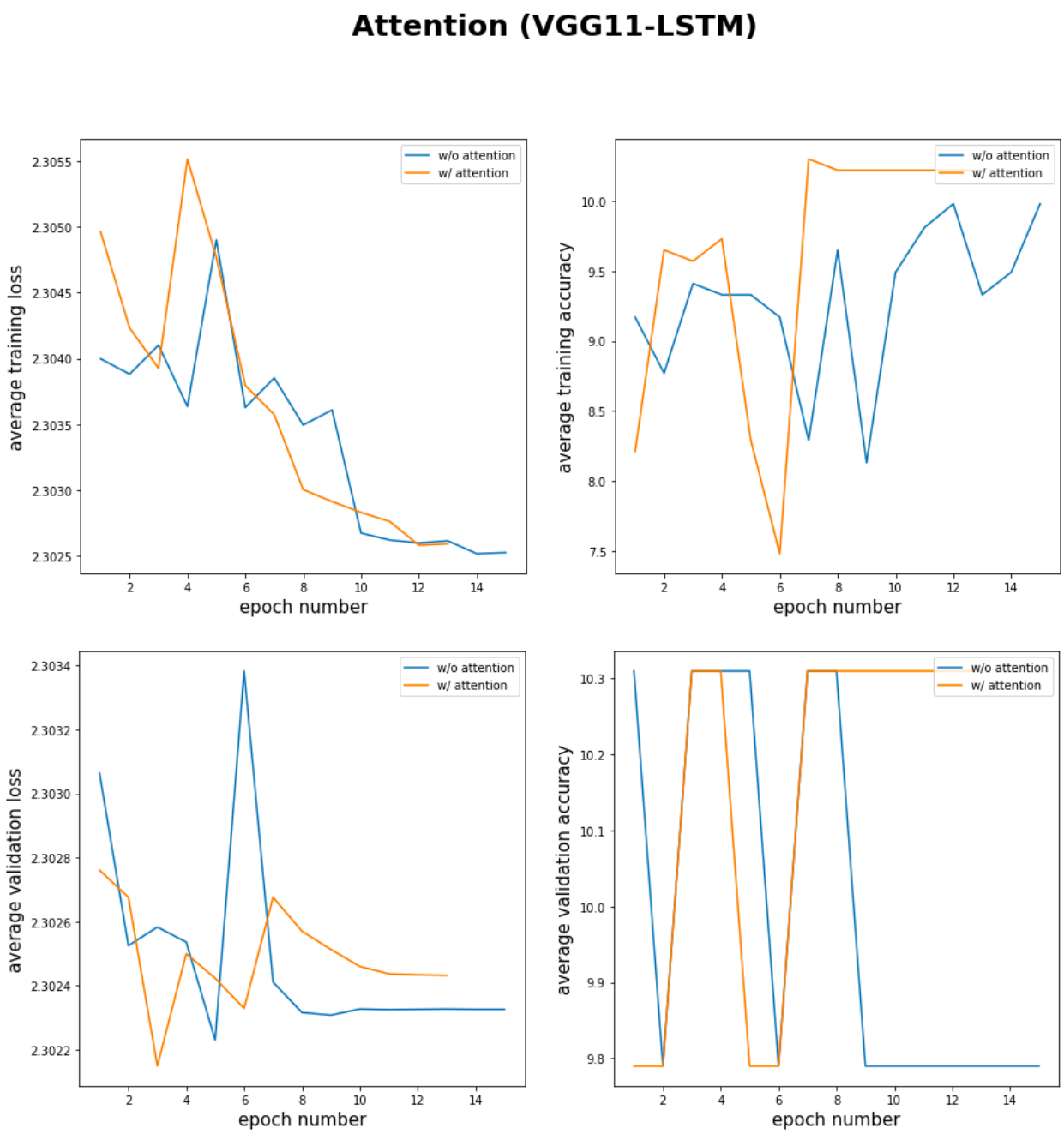

Figure 50. Experimental results of VGG model for CNN + LSTM

From the figures above, our team managed to achieve 25% in validation score for the attention-based CNN + LSTM architecture, seen in Figure 48. This is compared to the VGG 11 + LSTM architecture that did not perform as well with roughly 10% accuracy in validation which is considerable to random guessing, shown in Figure 50. As such, we implemented the CNN + LSTM model for the final architecture.

6.3.4 Data Augmentation

Finally, the models are trained with two kinds of data augmentation: masking and pose estimations, besides the regular dataset. The aim is to check if minor changes in the video data could further improve the accuracy for the model.

6.3.4.1 Masking Data Augmentation

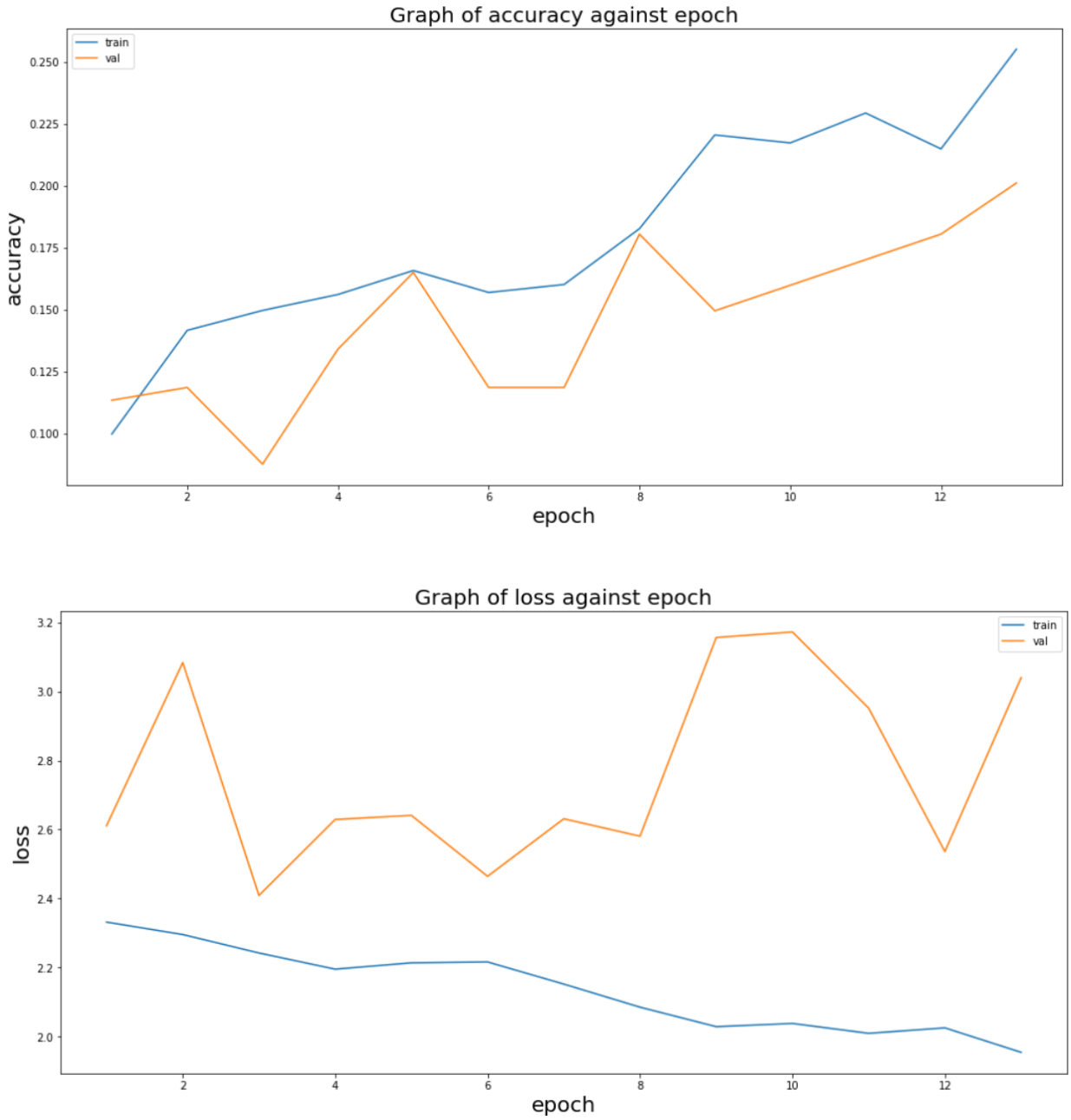

Figure 51. Experimental results of masking for CNN + LSTM

As seen in Figure 51, data augmentation with masking does not improve the accuracy of the training. The validation accuracy peaks at 20.3% before triggering the early stopping as

compared to the accuracy without data augmentation. Furthermore, using data augmentation on the fly while training takes up computation time. If there are no inherent benefits of augmenting data, our team will not be using it in our final pipeline.

6.3.4.2 Pose Estimation Data Augmentation

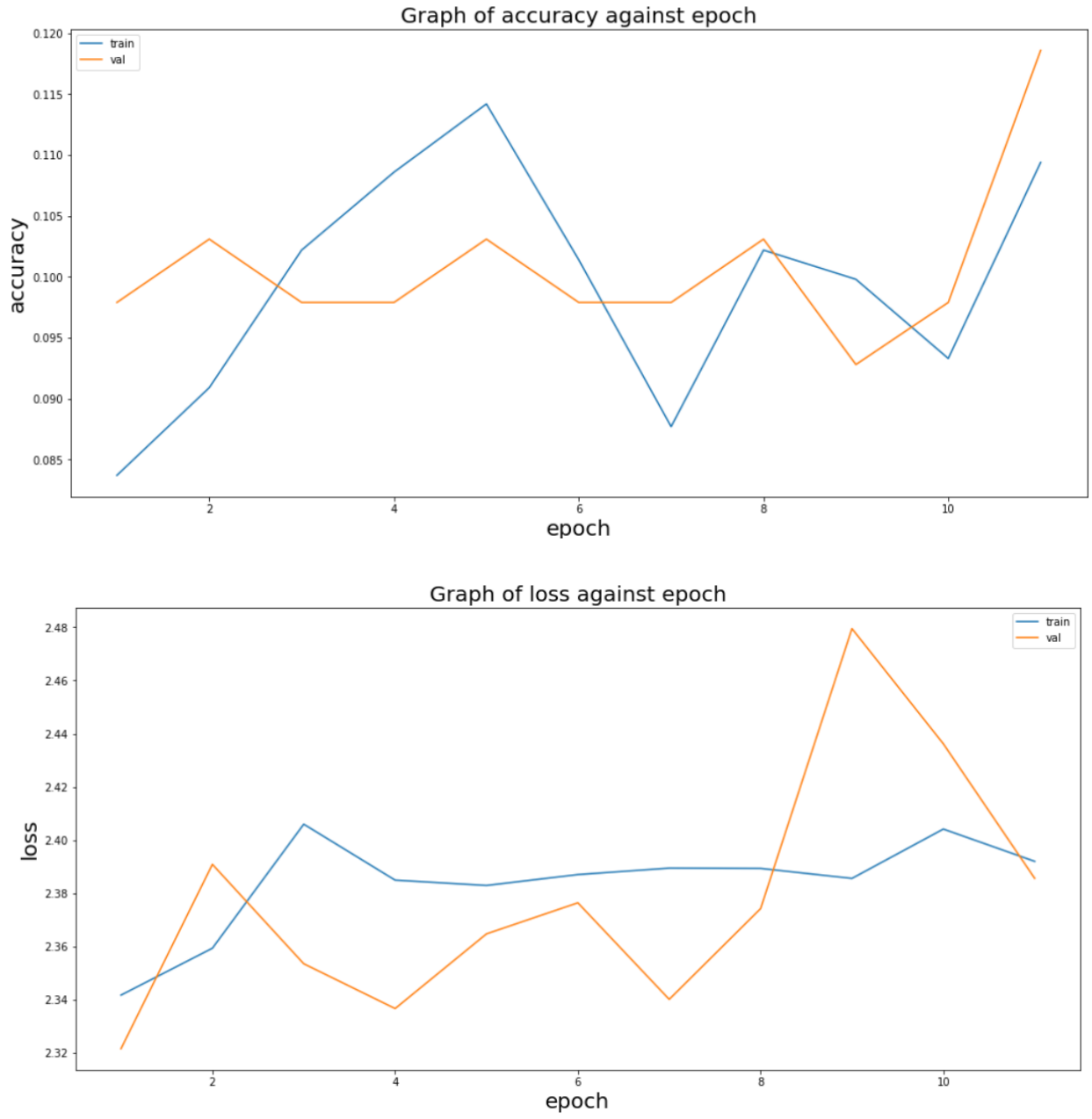

Figure 52. Experimental results of Pose Estimation for CNN + LSTM

Based on Figure 52, the validation accuracy trained on pose estimation data augmentation was critically low at around 12.5%. On top of that, creating pose estimation from frames of images

takes up a large amount of computation and time. The time taken to train 1 epoch was almost 3 times the time required to train without pose estimation data augmentation.

6.3.4.3 Data Augmentation Conclusion

From the experimental results above, data augmentation does not improve the training and validation accuracy significantly. In fact, it took longer per epoch as augmentation is computation intensive. As such, the final pipeline will be done without data augmentation.

6.4 Final Architecture

Figure 53 illustrates the architecture of the final model distilled from the experiments done above in Section 6.3.

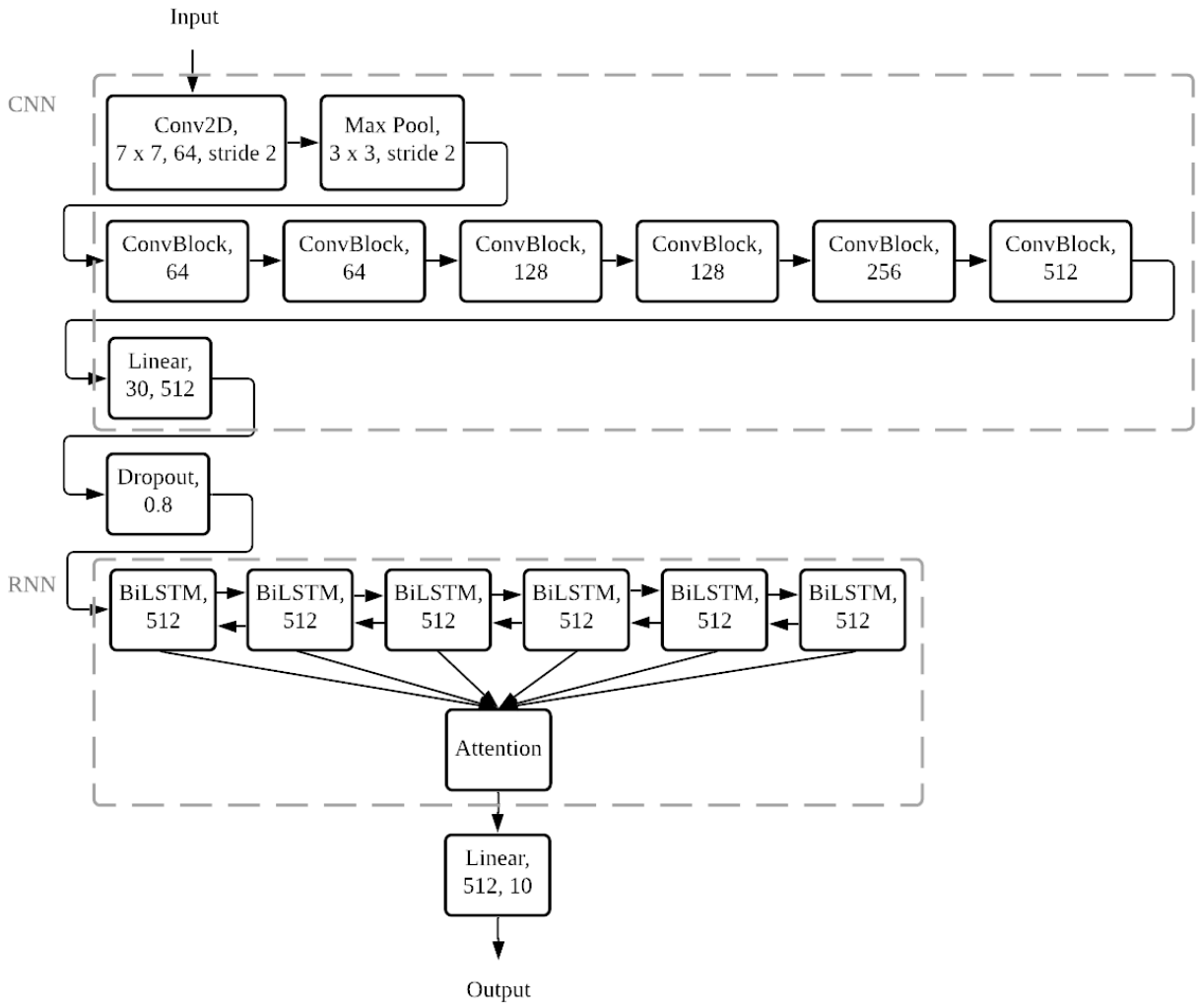

Figure 53. Final Model Architecture Diagram: CNN + LSTM + Attention

7 RESULTS & EVALUATION

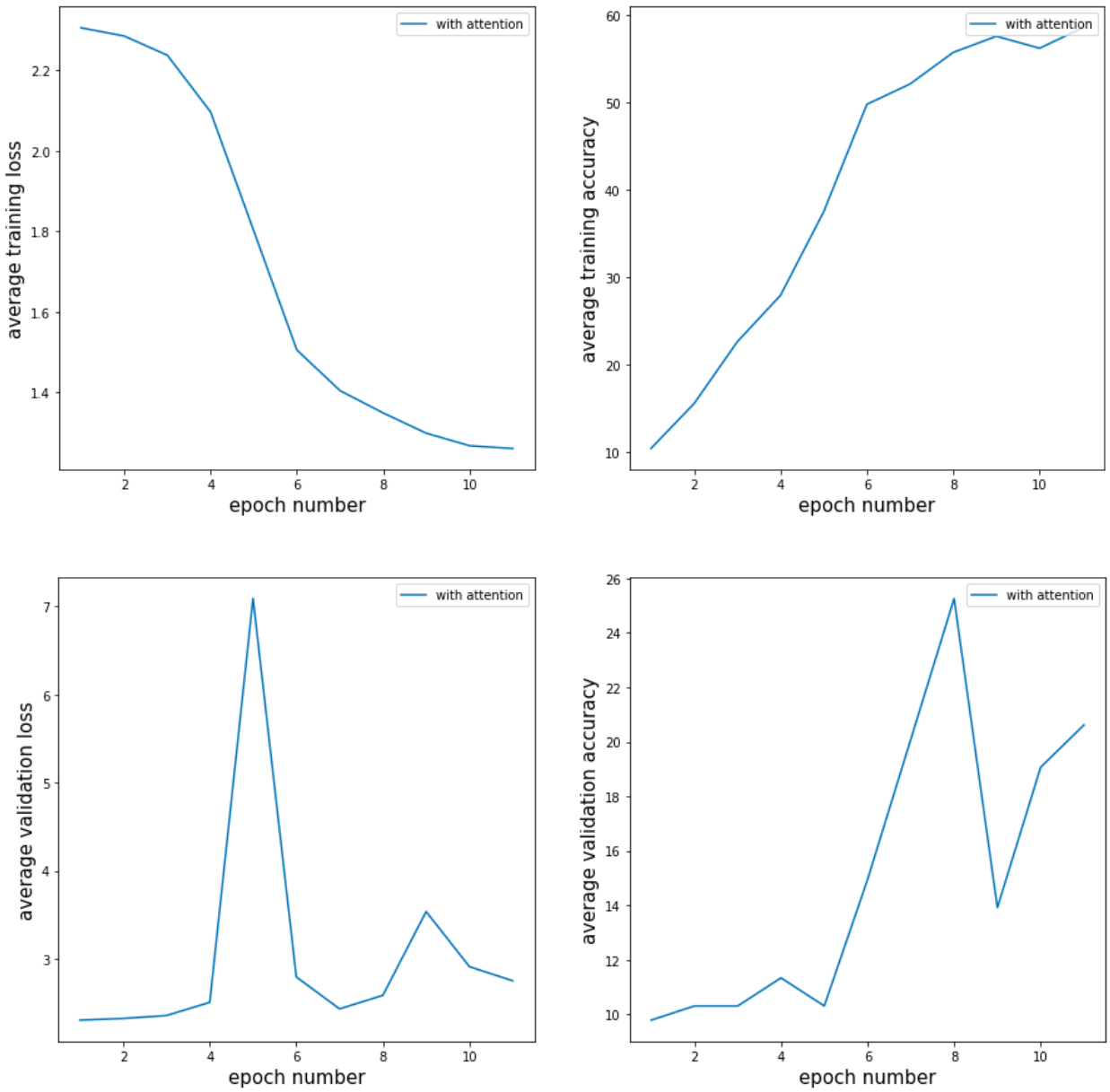

Figure 54. Train and Validation Statistics for Final Model

Based on Figure 54, we can see that the final model has a peak validation accuracy of 25.5% and training accuracy of close to 60%.

Table 3: Recognition rates (%) of our models using RGB+Depth data.

Method	Balanced Test Set			Imbalanced Test Set		
	top-1	top-3	top-5	top-1	top-3	top-5
CNN + LSTM	39.31	59.13	66.64	37.84	57.68	65.30
CNN + FPM + LSTM	41.26	60.60	68.76	39.45	58.50	66.55
CNN + LSTM + Attention	57.80	76.24	82.57	54.55	62.79	70.82
CNN + FPM + LSTM + Attention	60.02	78.00	**83.93**	56.99	75.79	**82.22**
CNN + FPM + BLSTM + Attention	**62.02**	**78.11**	83.45	**59.24**	**76.03**	81.60

Table 4: Recognition rates (%) of our models using only RGB data.

Method	Balanced Test Set			Imbalanced Test Set		
	top-1	top-3	top-5	top-1	top-3	top-5
CNN + LSTM	23.00	37.03	43.66	22.80	36.94	43.61
CNN + LSTM + Attention	42.14	61.83	71.21	40.89	60.68	69.42
CNN + FPM + LSTM + Attention	44.89	64.24	72.26	43.69	62.79	70.72
CNN + FPM + BLSTM + Attention	**49.22**	**68.89**	**75.78**	**47.62**	**67.38**	**73.89**

Figure 55. Results from Baseline Models Implemented by Paper

The results of baseline models implemented by the paper publishing the AUTSL dataset, which is being used in this project, is shown in Figure 55. For the CNN + LSTM + Attention model that takes in RGB data from the balanced test set, peak accuracy of 42.14 % for top-1, 61.83% for top-3 and 71.21% for top-5.

The high accuracy can be attributed to the fact that they used a VGG11 pre-trained model which was pre-trained on the large dataset of ImageNet. In addition, while our learning rate was fixed at 1e-5, their learning rate was set to 1e−5 and then reduced to 2e−6 if validation accuracy does not improve for ten epochs. The training is only terminated if there is no improvement for another ten epochs. Decreasing learning rate based on model performance during validation ensures that the poor performance is not because the model is converging fast to a suboptimal solution.

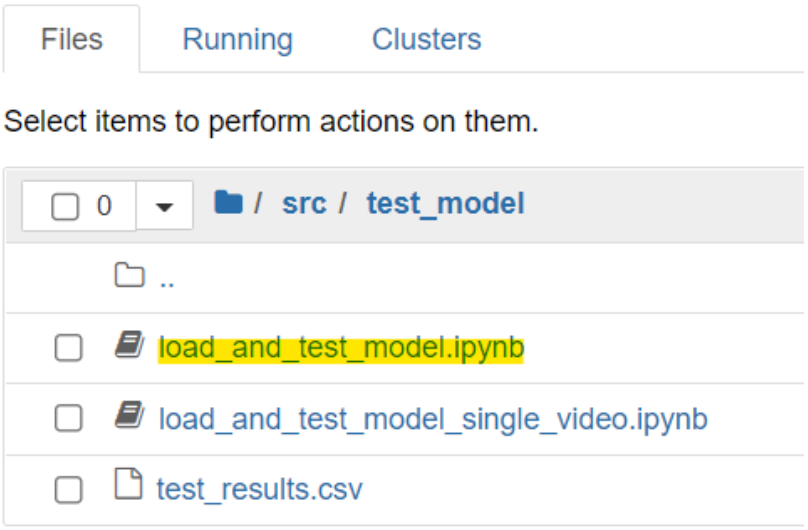

Figure 56. Select the load_and_test_model.ipynb in the ./src/test_model directory

	precision	recall	f1-score	support
champion	0.15	0.12	0.13	17
glass	0.17	0.06	0.09	16
wrong	0.31	0.24	0.27	17
bad	0.25	0.24	0.24	17
married	0.44	0.50	0.47	16
potato	0.09	0.18	0.12	17
congratulations	0.00	0.00	0.00	14
child	0.19	0.59	0.29	17
inform	0.50	0.12	0.19	17
father	0.50	0.12	0.20	16
accuracy			0.22	164
macro avg	0.26	0.22	0.20	164
weighted avg	0.26	0.22	0.20	164

Figure 57. Test Results from the Final Model

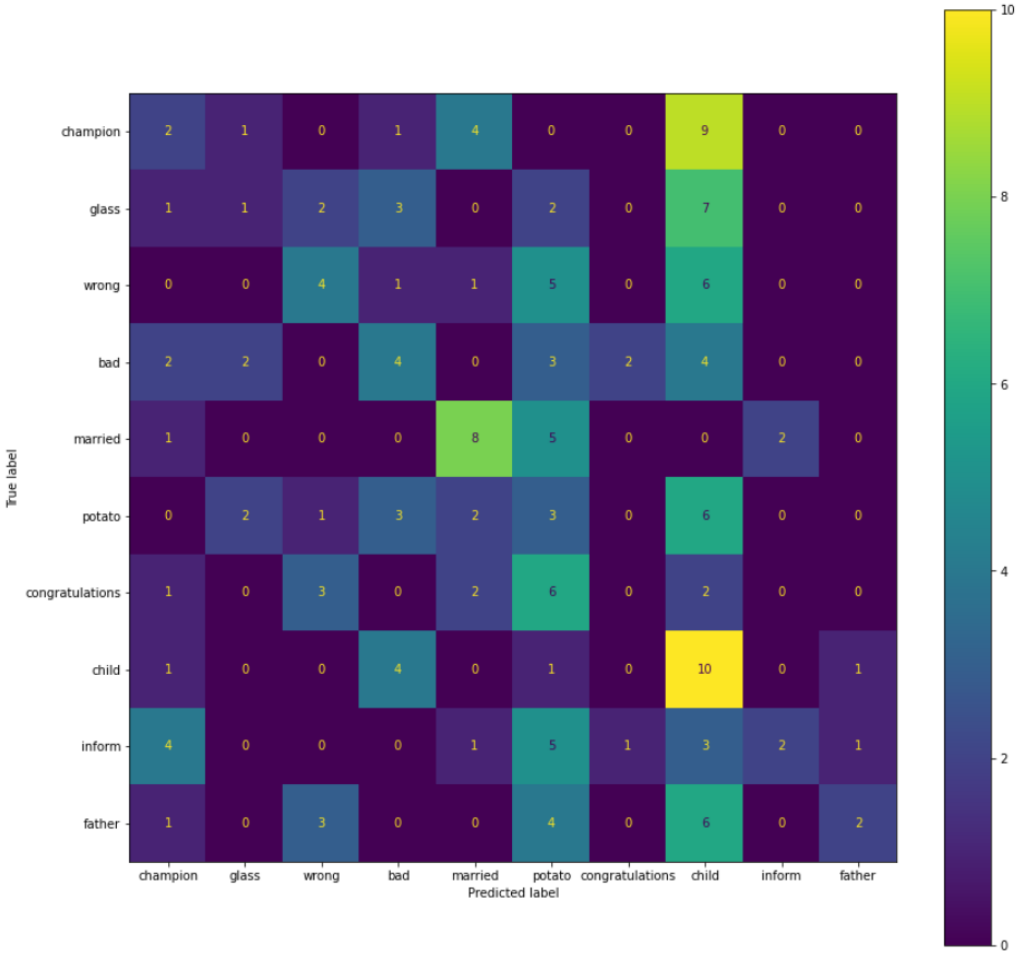

Figure 58. Confusion Matrix from the Final Model

The code to generate the classification report and confusion matrix can be found in the load_and_test_model.ipynb notebook which can be found in ./src/test_model directory (See Figure 56). For the final model results as seen in Figure 57, our proposed model scored 22% on the test set, with a weighted average of precision score of 26% and weighted average recall score of 22%. Based on Figure 58, it can be seen that the model is biased towards predicting the child label, followed by the potato label as those 2 columns are the most lit.

Considering that the CNN layers were not trained on huge dataset before, the score is relatively decent after training on close to 12 epochs with early stopping. However, our model performance still pales in comparison to 98.42% recognition rate achieved by the state-of-the-art SLR-GCN (Jiang et al., 2021) as shown in Figure 59. Unlike the state-of-the-art SLR-GCN, our model lacked a successful implementation of pose-estimation. For example, the SLR-GCN model performed graph reduction to learn only from the key points that were most critical in SLR, whereas the skeleton of the whole-body was used in our experiments above. The use of whole-body pose estimation might have impeded the model's ability and progress to learn due to unnecessary

keypoints, i.e. hand gestures are unlikely to affect the lower body, thereby only pose-estimation for the upper half of the body is necessary. Furthermore, in our implementation of pose-estimation, pose lines were simply overlaid onto the images and the model had to learn to interpret the pose instead of directly using the different key points as input. It is also important to note that our model failed to consider other factors like optical flow and features unlike the multi-modal ensemble model. Generally, the poor model performance can be attributed to the simplistic model used for a complex task like SLR.

	Finetune	Track	Top-1
Baseline	-	RGB	49.23
Baseline	-	RGB-D	62.03
Ensemble	No	RGB	97.51
Ensemble	No	RGB-D	97.68
Ensemble	w/ Val	RGB	**98.42**
Ensemble	w/ Val	RGB-D	**98.53**

Figure 59. Screenshot of Test Results Using SLR-GCN on AUTSL Dataset (Source: (Jiang et al., 2021))

8 MODEL FAILURE

One of the ways in which the model has failed is predicting videos that look like mirror images. In the examples below, both signers perform the sign for ‘champion’. However, one signer is performing the action towards the left while the other signer is performing the action towards the right.

The two videos are broken down into its constituent 30 frames as shown the figures below.

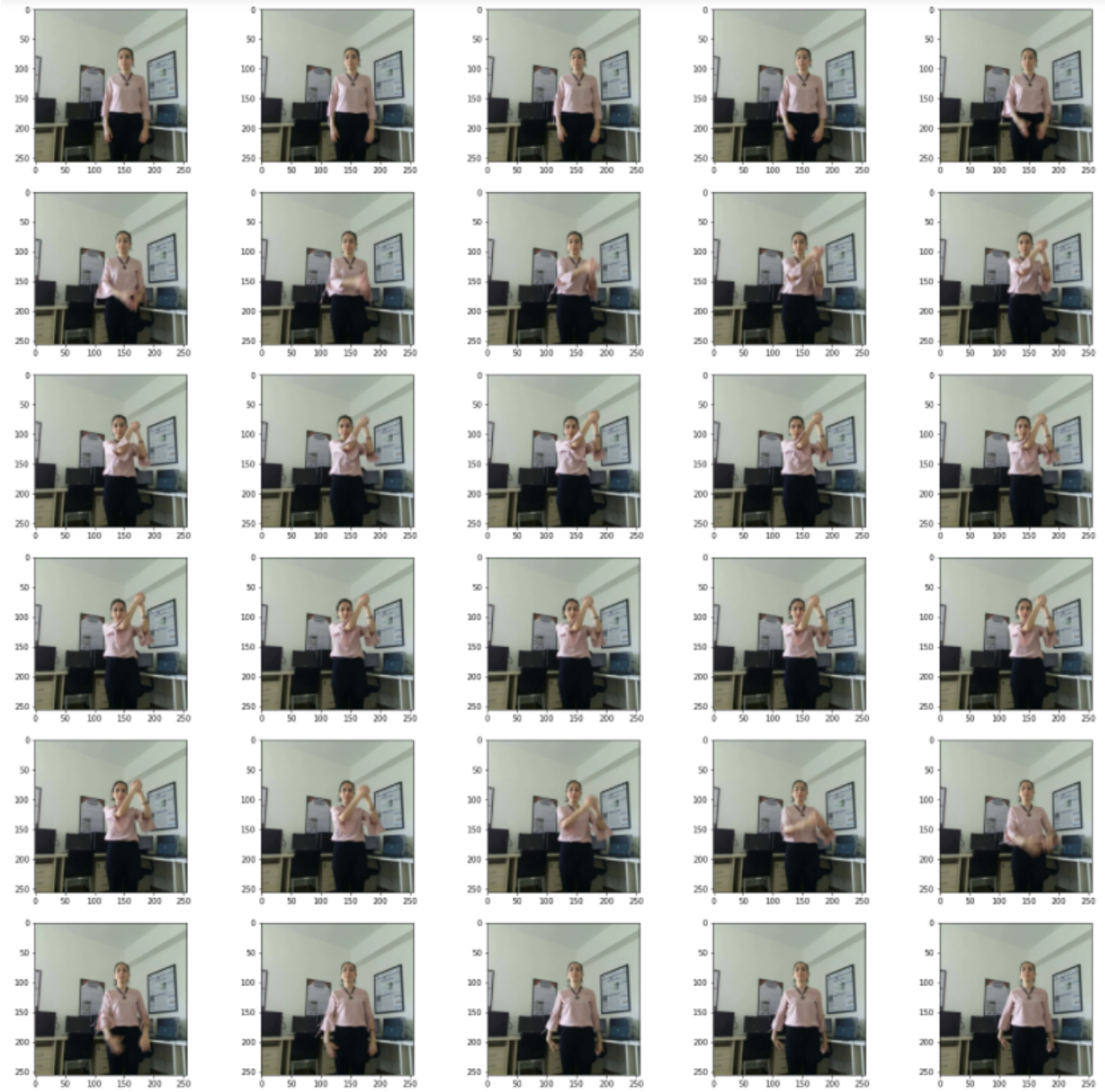

Figure 60. Example 1 (left); Label: Champion | Predicted: Married

Figure 61. Example 2 (right); Label: Champion | Predicted: Champion

From Figure 60 and Figure 61, we can see that although the actions are similar, one of the examples was misclassified from champion to married.

8.1 Funny Failure

One of our courageous members decided to put the failure to the test to see if the failure was persistent against different signers. He decided to perform the Turkish Sign Language for “champion” towards the left side.

Figure 62. Member (left); Label: Champion | Predicted: Married

Unfortunately, from Figure 62, the failure still exists even though our member is not even married. Poor lad, he had his hopes high.

9 LEARNING POINTS

Some problems faced by the team include encountering GPU Cuda Memory Error which was resolved by fixing the number of frames and reducing the number of classes used for the problem.

Contrary to popular belief, hyperparameters are very important for the model to train well. Initially, when the models were tried, they were not able to achieve accuracies of 11% or more. Given that the data being used to train is limited to 10 classes, this means that the models were not learning as the accuracy achieved was roughly equivalent to random guessing. However, when the learning rate was decreased from the default of 0.001 to 1e-5, the models showed improvements in training accuracy.

Using pretrained models such as VGG or ResNet does not necessarily mean better performance as the models were trained on general image data, meaning that the embeddings would be generalised as well. However, this did not work well during our experiments as the chosen problem is quite specific. Furthermore, the pretrained models were too big to be custom trained properly, especially when considering the small dataset size being used.

Based on the test results published in the paper, it can be concluded that this is a difficult problem, with a naive CNN-LSTM model achieving only test accuracy of 39.31% (refer to Figure 55). This is because the dataset is made to be difficult, with different signers, different clothes worn, different backgrounds, different angles, and even different ways of performing the same sign. Due to the many variations built into the dataset, it could be possible that training with only a small fraction of the dataset with 10 classes could have increased the difficulty of the problem as the model might not be able to properly pick up the important parts of the image with all the unimportant parts of the frames which are changing.

10 CONCLUSION

To sum up, there were multiple challenges faced by the group that ranged from downsampling the dataset and reducing the frame rate to testing out different architectures that produce results that are satisfactory.

However, the team is able to find ways around the problem and ultimately produce a model as well as an interactive GUI that is able to classify Turkish Sign Language videos into English meanings.

11 MEMBER CONTRIBUTIONS

Jamie: Problem Statement, State-Of-The-Art Research, Graph Generation Source Code, Explanations for Experimental Results, Comparison of Proposed Model to State-Of-The-Art

Xiang Qian: Dataset and Dataloader Source Codes, Exploratory Data Analysis and Data Augmentation Experiments

Huiqing: Model Hyperparameter Tuning, Model Training, Model Source Code, Experiments on Different Architectures and Comparison.

Glenn: Notebooks for model test and analysis, local GUI, deployed GUI to AWS, experiments on video frame counts, and pretrained vgg testing.

12 BIBLIOGRAPHY

Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç, & Courville, A. (2017). Recurrent Batch Normalization. *ICLR 2017*.

Donahue, J., Hendricks, L. A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Darrell, T., & Saenko, K. (2015). Long-term recurrent convolutional networks for visual recognition and description. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2015.7298878

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *International Conference on Machine Learning*, 448–456. https://arxiv.org/pdf/1502.03167.pdf

Jiang, S., Sun, B., Wang, L., Bai, Y., Li, K., & Fu, Y. (2021, March 26). *Skeleton Aware Multi-modal Sign Language Recognition*. ArVix.Org. https://arxiv.org/pdf/2103.08833v4.pdf

Kingma, D. P., & Ba, J. (2014, December 22). Adam: A Method for Stochastic Optimization. ArXiv.Org. https://arxiv.org/abs/1412.6980

Lau, S. (2018, June 20). *Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning*. Medium.

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

Ng, J. Y., Hausknecht, M., Vijayanarasimhan, S., Vinyals, O., Monga, R., & Toderici, G. (2015). Beyond short snippets: Deep networks for video classification. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2015.7299101

Ruder, S. (2016, January 19). An overview of gradient descent optimization algorithms. Sebastian Ruder. https://ruder.io/optimizing-gradient-descent/index.html

Sincan, O. M., & Keles, H. Y. (2020). AUTSL: A Large Scale Multi-Modal Turkish Sign Language Dataset and Baseline Methods. *IEEE Access*, *8*, 181340–181355. https://doi.org/10.1109/access.2020.3028072

Stack Exchange - Cross Validated. (2016, August 22). *How to use early stopping properly for training deep neural network?* https://stats.stackexchange.com/questions/231061/how-to-use-early-stopping-properly-for-training-deep-neural-network

Stack Exchange - Data Science. (2018, August 20). *Early stopping on validation loss or on accuracy?* https://datascience.stackexchange.com/questions/37186/early-stopping-on-validation-loss-or-on-accuracy