ASLNet: Towards Real Time Sign Language Alphabet Interpretation

Takumi Miyawaki
Department of Engineering, 2021
NYU Abu Dhabi
Abu Dhabi, UAE
tm2904@nyu.edu

Munachiso Nwadike
Department of Computer Science, 2019
NYU Abu Dhabi
Abu Dhabi, UAE
msn307@nyu.edu

Romeno Fernando Wenogk
Department of Computer Science, 2021
NYU Abu Dhabi
Abu Dhabi, UAE
wenogk@nyu.edu

***Abstract*—The lack of communication infrastructure for those with hearing impairments is a clear problem that must be tackled in today's society. The paper, explores methods in which machine learning can be applied in order to ameliorate this case. the different machine learning models, explored in this paper include, Convolutional Neural Network, Faster-RCNN, and YOLO v3. From these models, we used the YOLO v3 model, to implement a model, which localized and classifies ASL Hand Gestures of the Alphabet. Further, we incorporated the model into an android application, in order to produce a working prototype that can be used in everyday life.**

I. Introduction

Our work was motivated by a desire to help with a problem we found while researching smart city development. We sought to help ameliorate the lack of communication infrastructure for people with hearing impairment. One particular finding that motivated us to work towards this objective was the minimal number of sign language interpreters in city courts. This made us wonder, if they are unable to sufficiently accommodate the deaf community in such an official setting of a city court, how is the infrastructure keeping up in other areas. Through this we found that mental health issues are especially common in the deaf community. Some mental health issues include, depression, anxiety, bipolar disorder and schizophrenia. According to a study conducted in the University of West Florida, the likelihood of depression among deaf students compared to hearing students were nearly double. A large cause of this is said to be due to communication barrier. This is further amplified through the lack of counselling facilities for the deaf community. One small study involving 54 people found that more than half hadn't been able to find mental health services that they, as deaf people, could use.This is because they are not able to effectively communicate their symptoms. From this, we decided to make a project that focuses on assisting the deaf community.

II. Approach

One of the priorities for our product was accessibility and convenience, so that the user can use it in any case, with ease. Also, since our focus is to give a wider option for deaf people to communicate, it will enable the user to input sign language, and the product will output letters. Since, this is a starting point, we will just have inputs as letters. From these objectives we reached the idea of ASLNet, a mobile app that uses the camera and allows the swift recognition of ASL letters by finger spelling. The goal is to build a neural network capable of classifying images of ASL hand gestures into letters. This neural network would be integrated with a mobile application system which could be used to capture fingerspelling gestures in real time and translates them from the ASL into standard English Alphabet. One of the factors that leads to a good machine learning model is good data, and so it was vital for us to find a suitable dataset for our use case.

III. Dataset

We decided to use the ASL alphabet dataset on Kaggle which contains 87,000 200x200 images classified into 29 classes including the letters from A-Z, and three classes for space, delete and nothing. Each letter has 3000, raw images and post-processed images. We chose this dataset because, images are similar to what we would expect from a mobile phone camera as they are RGB images and it has sufficient amount of data to train our model. Some sample images of the dataset are shown in figure 1.

We have used a second dataset. This data set was a custom dataset we have produced, consisting of 100 images per letter. This dataset consisted of the labeling for the classification as well as regional labelling for the bounding boxes around the hand for each picture. The dataset include closeup pictures which only included the hand as well as full body images. Examples of the images can be seen in figure 2. further details on the reason why this dataset was used will be explained further.

Fig. 1. ASL alphabet A, B and C. This was the dataset we initially built a custom CNN for, which eventually inspired us to build our own custom dataset

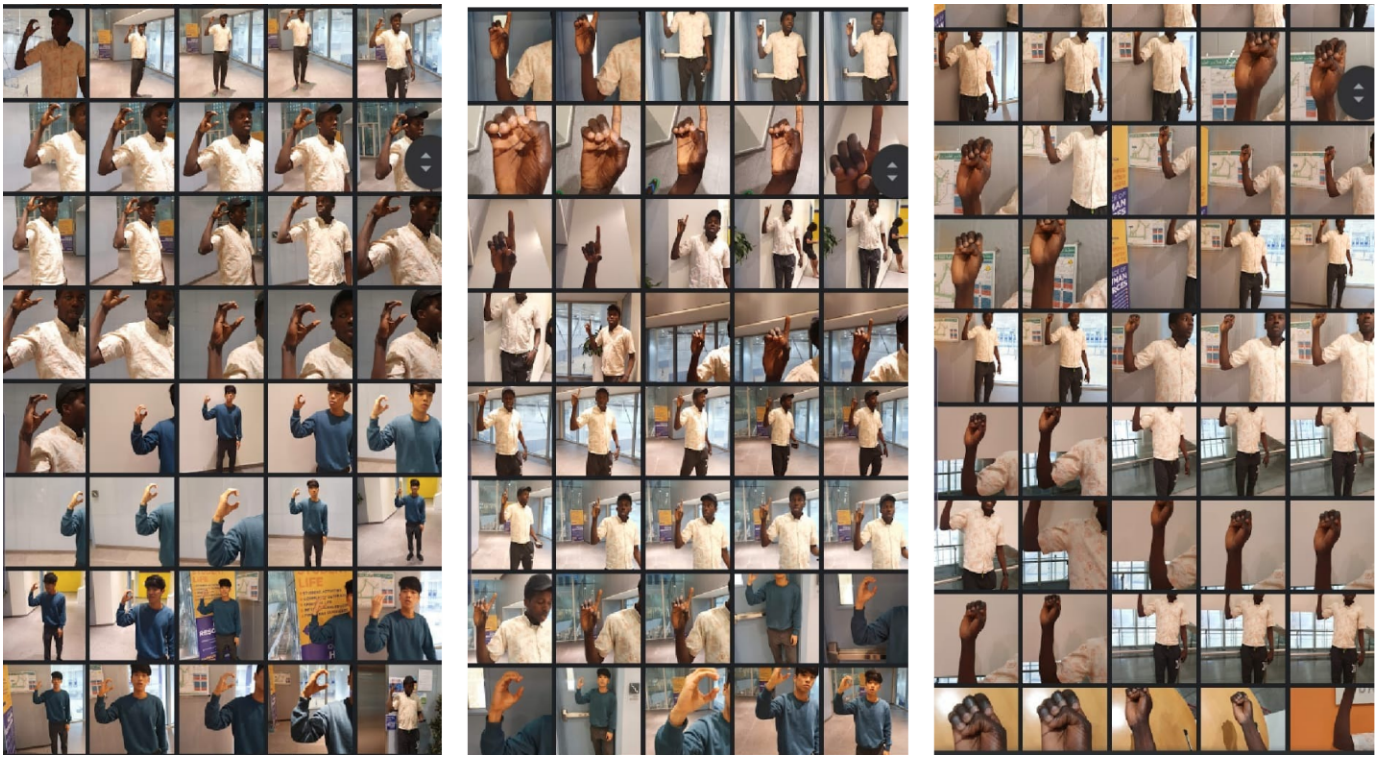

Fig. 2. Collage of images from our custom dataset. We created 500 images of the ASL alphabet A to E, obtaining 100 images per letter. Each image was manually hand-labeled

IV. Previous Works and Background on CNN

A team at UCSD did work on a neural network architecture that made use of CNNs to classify hand gestures on Fingerspelling Recognition [1]. Using a dataset of 31,000 depth maps, this architecture achieved an accuracy greater than 90%. It differs from our model because it requires sophisticated technology such as Kinect or 3D Camera to obtain depth maps. We aim to perform classification with merely a smartphone camera. Figure 3 shows examples of the depth image dataset use to train the model.

Another team from TU Munich did work on a neural network architecture that made use of RGB images for gesture classification. They focused on classifying gestures from two gestures datasets - EgoGesture NVIDIA Hand Gesture Dataset. While the NVIDIA dataset was not open access, the EgoGesture dataset was. It consisted primarily of RGB-D videos. While the code for this project was written in PyTorch, the NVIDIA dataset was not open access, and the EgoGesture dataset of 32GB was too large for us to reproduce their results, within our allocated time frame. Moreso, the network architecture used was focused on RGB-D images, which requires a depth sensor to provide information for the depth channel. However, this is less convenient for our application again, since our intended solution is to make use of only RGB information from smartphone cameras in order to classify ASL alphabet and obtain accurate fingerspelling. Moreover, As can be seem from figure 4, the dataset is images

Fig. 3. Depth maps used in the work from UCSD.

the persons own hands.

Our first approaches used a similar architecture as the

Fig. 4. RGB-D images using in training the model from TU-Munich.

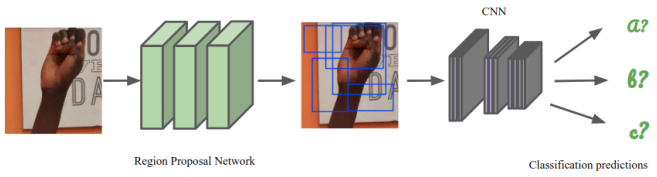

Fig. 5. Diagram showing how RCNN works by having two distinct steps of localisation and classification prediction. This modularity was what gave us the motivation to pursue our work, thinking that we could use a custom CNN for the classification prediction, and use a modified version of the region proposal network from RCNN for the region proposal component

approaches introduced above, however, instead of using a depth images, we solely used RGB images. However, due to numerous drawbacks related to the variability of the size of the image as well as the hands relative to the image, it was unsuccessful. This lead us to explore Neural Networks with region-Proposals.

V. Previous Works and Background on Neural Networks with Region-proposal

The advantage of using Neural Networks with Region-proposals is that they are usually more rigorous to size variability of the object it is detecting, thus in our case, meant that the user could be closeup or further from the camera. An algorithm proposed by University of Science and Technology, Hefei, China [3] made use of the well known Faster R-CNN, with multiscale detection, for hand detection and classification of sign language. This algorithm worked more similarly to our intended algorithm, as it is able to detect and classify hands from a full body RGB image. It focused on classifying hands to distinguish left from right. This modified Faster-RCNN was trained using a proprietary dataset, which prevented us from running the neural network to understand how it works. Moreover, since their dataset format, required region labelling, this meant that we could not use the Kaggle dataset. However, this work gave us an important clue as to how we would pursue our own task. We realised that we could make use of the two part structure in the R-CNN architecture to separate the processes of hand localisation and classification. Figure 5 shows an example of a R-CNN architecture. It is split into two components, one that gives the region proposal, and the second which classifies the hand inside that proposed region. Thus, our plan was to find a dataset, that would enable us to do a region-proposal for a hand, then use the Kaggle Dataset for the classification. This modularity also meant that as long as we have a dataset of pictures of hand gestures from 1 language, we could always implement it into this architecture. However, as we continued our research and implementation, we realized due to the sequential process, the execution time was too long for our objective(over 40 seconds). So, our conclusion was to create our own dataset and implement the Faster-RCNN. Moreover, as we will be making a dataset with region labelling, we extended this to implementing YOLO v3 as well.

VI. Experiments

A. System Specifications

For the training and testing, we use Intel(R) GeForce RTX 2080 with 7979 MiB memory.

Software specifications:

- Ubuntu 18.04
- Nvidia driver 430.40
- CUDA 10.1
- YOLO
  - Python 3.7.4
  - PyTorch 1.2.0
- Faster-RCNN
  - Python 3.6.9
  - PyTorch 1.3.1
- Custom CNN
  - Python 3.6.9
  - TensorFlow 2.0.0
  - Keras 2.2.4-tf

B. Training

We trained our custom CNN, on 87,000 images of sign language letters A-Z, delete, space, and nothing. We used a 80-20 training-testing split. Our custom CNN consists 3 convolution layers, 3 max-pooling layers, and 3 fully connected layers, as shown in figure 6. The network achieved a testing accuracy of 96.64% and the execution time was around 2-4ms. Moreover, looking at the confusion matrix of the result, apart of which is shown in the figure 7, the result was more than satisfactory. However, when testing the model using actual phone images on the application, there were many cases where the result it gave was wrong. This was mainly due to the difference in dimension of the landscape images in the database to the portrait images from the phone. Moreover, since it was a simple CNN, it was unable to classify letters accurately if the person was further than half a meter away. As a result, we were motivated to build our own custom dataset. We fed in images from our custom dataset into YOLOV3 without use of downsampling or resizing. All images were still the original size of 2268 x 4032 pixels. We made use of Adam optimizer,

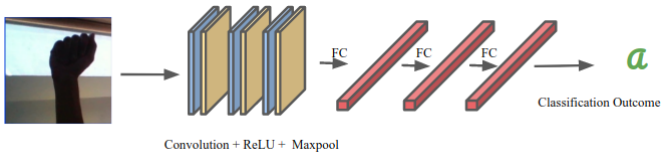

Fig. 6. Custom CNN initially used.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	937	12	0	7	8	5	0	0	0	0	0	0	1
B	4	882	0	2	8	0	0	0	0	0	1	0	0
C	2	0	874	5	0	0	0	0	0	0	0	1	0
D	4	1	0	877	2	0	0	0	2	0	0	0	0
E	9	25	2	16	793	0	0	0	16	0	1	0	0
F	2	0	0	2	2	953	0	0	1	0	0	0	0
G	0	0	0	0	0	0	890	8	1	7	0	0	0
H	0	0	0	0	0	0	9	889	1	4	0	0	0
I	0	0	0	3	0	0	0	0	873	0	9	0	1
J	0	0	0	0	0	0	0	0	4	930	0	0	0
K	0	0	0	0	0	0	1	0	3	0	914	0	0
L	1	0	0	0	0	0	0	0	3	0	8	897	1
M	0	0	0	0	0	0	0	0	5	0	5	0	850

Fig. 7. Part of the Confusion Matrix of The custom CNN model (Letters A to M)

in the default PyTorch settings, with an initial learning rate of 0.002324. We set a batch size of 6, and only needed to train for 70 epochs before obtaining satisfactory classification or convergence. The checkpoint was saved once per epoch. The images in figure 8 show our training progress. Since YOLOV3 is able to detect images at multiple scales, there was no need to make use of region proposals as in the case of RCNN and Faster-RCNN.

All 500 images in our custom dataset were manually labeled using the labelImg open source utility. However, in the process of training our algorithm, we found that bounding boxes were being drawn in the wrong location. The reason is that jpg images always contain metadata which dictate their orientation on the screen. In the case of our images, the labeling was done with the images rotated counterclockwise 90 degrees, while metadata was placed in the images instructing that they be viewed upright by being turned back constraining that they be rotated 90 degrees to clockwise. Since YOLOV3 always process the images in the orientation that they are viewed, all our bounding box locations are taken relative to the top left corner of an images which is in a different orientation from how we labeled them. An example of the incorrect labeling is shown in figure 9.

Despite the incorrect labeling of our initial dataset, our model was able learn the correct hand classification, even thought they learned the wrong bounding box location. An example of this is seen in figure 10. This phenomenon is better understood when take two important facts into accounts. First since YOLO is convolutional, it takes information from all pixels in the image into account before making a classification. Also, as we can see from the equations below, the YOLO loss function is a sum of losses which are calculated relatively independently of one another. We know, from machine learning theory, that for any given function, there is a neural network that can approximate that function. Hence, our neural network learns to approximate a function which can understand the correct classification of an ASL letter, since it sees it via convolution, but systematically identifies it in the wrong location. It achieves this by attempting to minimise the correct classification loss, while minimising the wrong location loss.

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \tag{1}$$

$$+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{2}$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \tag{3}$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{4}$$

As we can see above, the YOLO loss function is a sum of losses that handle bounding box location and dimensions, as in (1) and (2), as well as the classification class and confidence as in (3) and (4). This means we can learn the correct classification in the wrong location, since these are considered by YOLO to be distinct entities.

To correct for the problematic bounding box locations, we had to further preprocess the images of our dataset using the exiftran utility in Linux. We manually re-annotated them using the labelImg utility after rotating them 90 degrees clockwise by editing the image metadata.

Once we corrected the orientations of our input jpg images, we were able to achieve a training mAP of 0.979 using YOLOV3. A picture showing our results from YOLOV3 is shown in figure 11.

In the case of Faster-RCNN, our images were also fed in with dimensions of 2268 x 4032 pixels. The learning rate was set to 0.001 and the learning rate was scaled manually using a value of 0.1. Training was for 50 epochs. However given how quickly the model achieved a perfect mean mAP of 1.0, it was clear that the model was overfitting to easily to the training data, which led us to focus on YOLOV3 for our final product application.

In the case of both YOLO and Faster-RCNN, We made use of visdom visualition package to visualise the training, as can be seen in the figure 8 and 12.

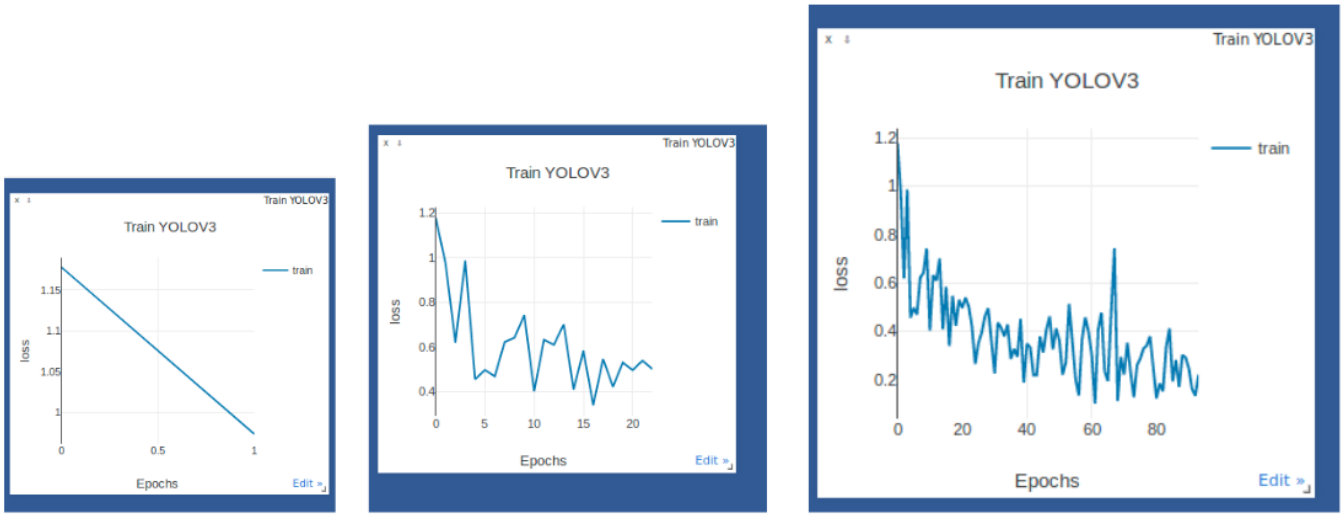

Fig. 8. Images depicting our training progress with YOLOV3, visualised using visdom.

Fig. 9. The bounding box location is interpreted different where images are rotated. Hence the need for additional preprocessing.

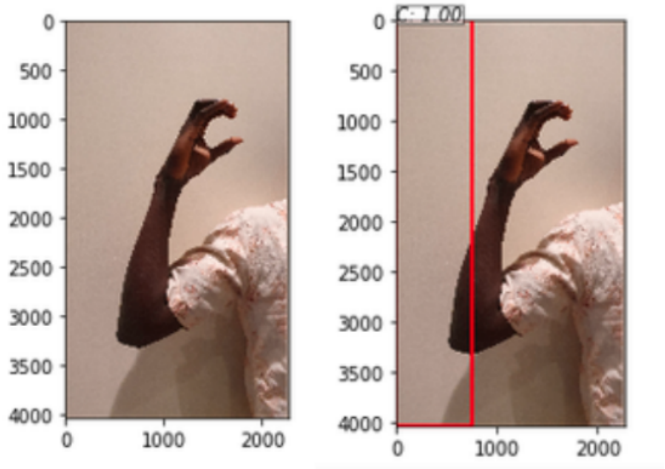

Fig. 10. The effect of the bounding box being predicted in the wrong place due to incorrect metadata. This problem was corrected using exiftran.

VII. APP DEVELOPMENT

We wanted ASL-NET to be implemented as a mobile application, since its application in the context of a smart city would require ease of use. The ideal scenario for would be that the app would run without need for internet connection, making it more accessible in countries around the world. The app would need to run cross platform, at least on the major mobile application platforms. Ideally, its interface would be functional and easy to use. A simple interface would allow users to quickly and efficiently express themselves without any sort of complicated interactions with the phone.

A. Interface

We decided to build the interface with three key buttons. One called “Flip” simply switches the camera from the front camera to the back camera and vice versa. The second button, which is the most important, is the, “Detect” button which captures the image from the phone camera, runs the model and returns the classification. Once the model returns a letter it is appended onto a text box at the middle of the screen. The third button is the, “Reset” button which simply resets the text box at the center of the screen. By the use of these three buttons as shown in the figure 13, the ASL-NET app allows for users to detect American Sign Language letters using the mobile phone camera.

The material design principles were the inspiration for the specific color scheme that was chosen. The mustard-yellow, blue-green and light maroon shades for the button colors were chosen after some research on light colors that

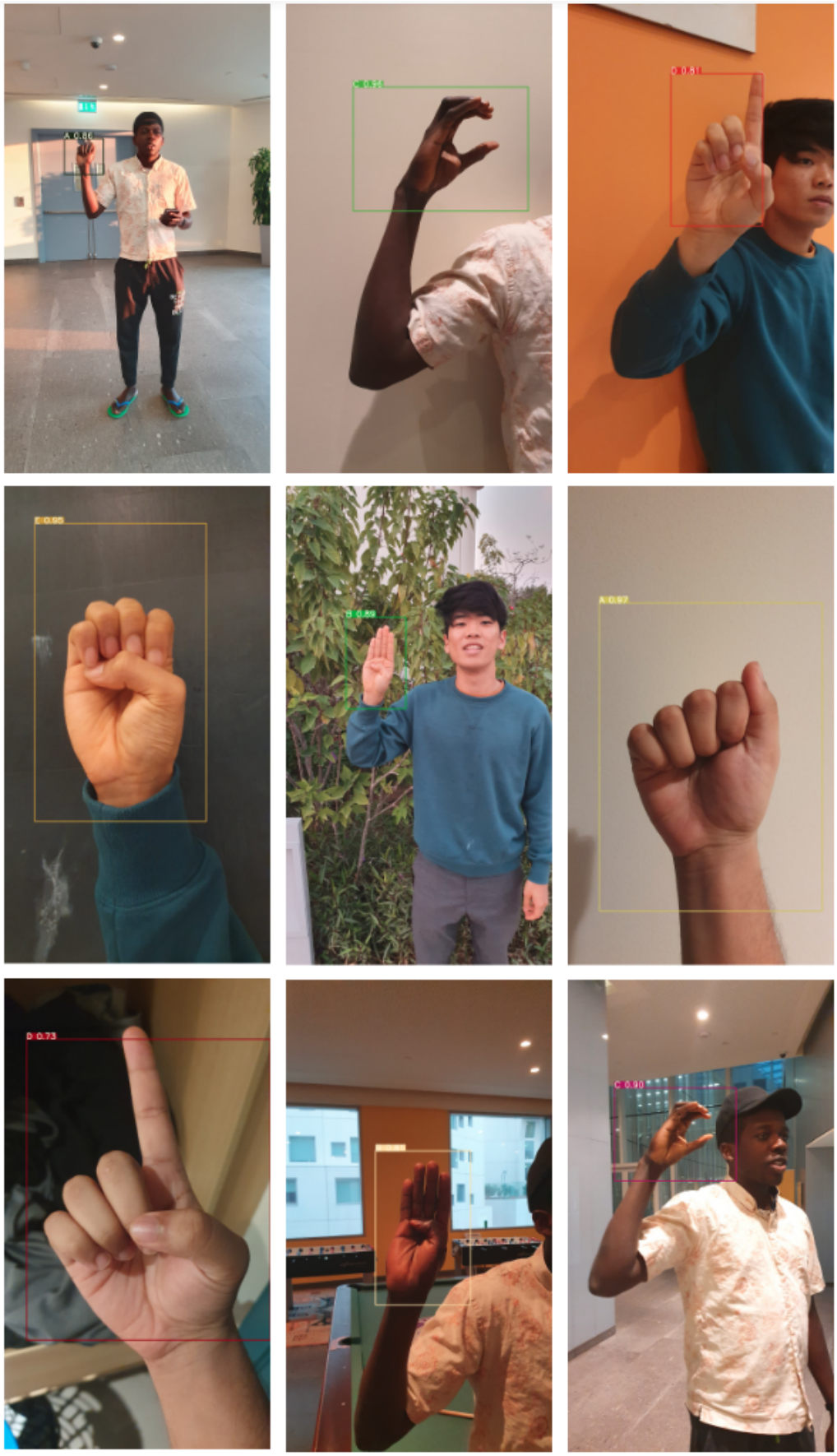

Fig. 11. Images depicting our results from training YOLOV3 on our dataset.

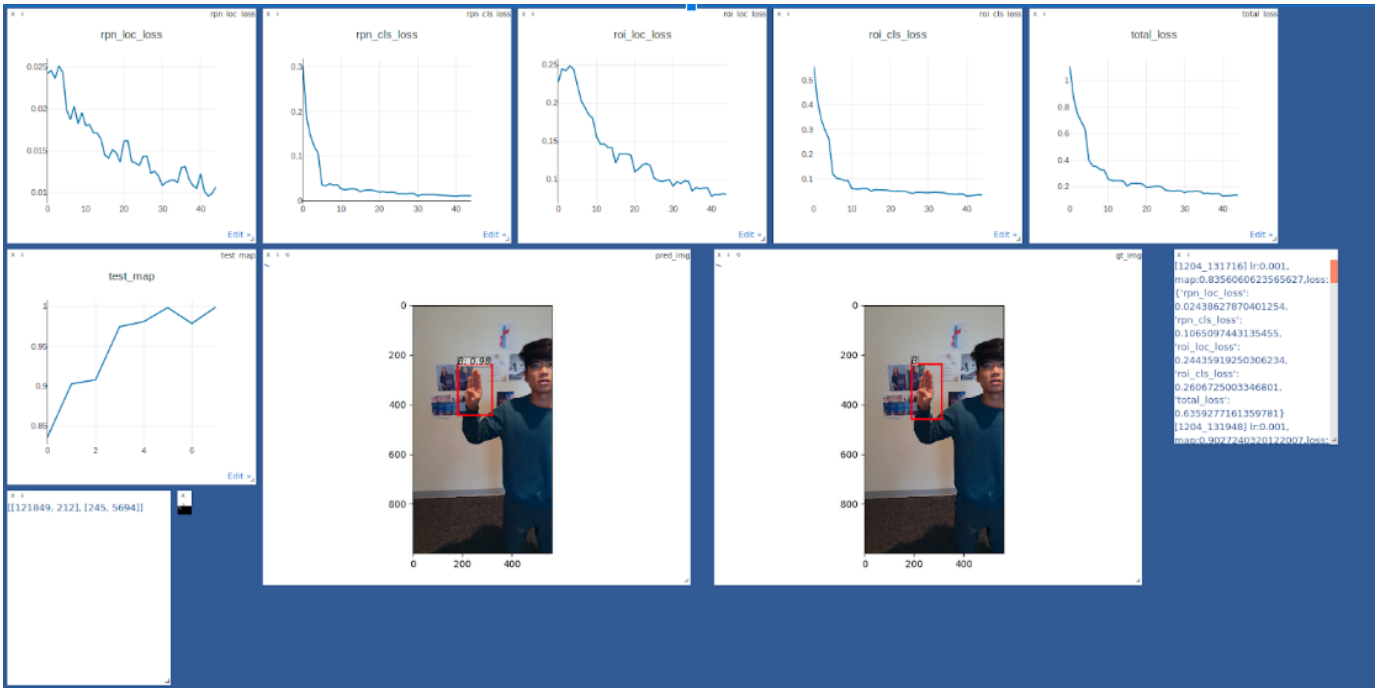

Fig. 12. Images depicting our training progress on Faster-RCNN, visualised using visdom.

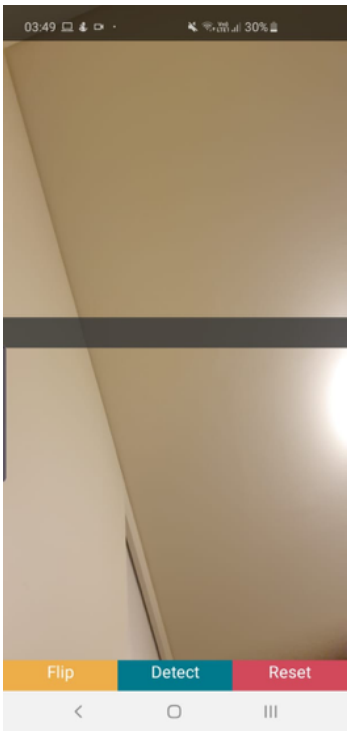

Fig. 13. The image shows the app interface with the three buttons: ”Flip”, ”Detect” and ”Reset”.

represent clarity and positivity.

B. Choice of technology

In order to build the app, certain choices needed to be made in terms of the technologies used to build the app. In order to build native apps for Android, the android studio software is usually used with the Java/Kotlin programming language. Additionally for IOS, to build native applications requires the XCode software and the objective C programming language. However, another technology called React Native exists which basically acts a wrapper that is written in the Javascript programming language and then can be compiled to run on both Android and Javascript. The slogan for React Native being, “Learn once, write anywhere.” and this was appealing to the ASL Net app as it allowed the focus to be on the model rather than the implementation of the app on each platform as different programming languages and software’s would need to be learned. The expo framework, which allows React Native projects to be initiated from the terminal and demo apps to be run directly on Android and iPhones was used. The expo online platform also simplified the process of generating the signed APK file in order to be submitted to the Google Play Store. Therefore because of the focus on a single programming language (Javascript) React Native was chosen as the technology to develop the ASL-NET app. The build cycle can be seen on figure 14

C. Technical difficulties

The initial plan for the app infrastructure involved the native mobile app being able to run the model in the phone itself.

Fig. 14. The react native build cycle.

Fig. 16. Web API for our application

Initially, the YOLO model was imported into tensorflow.js by the use of Onnx. Onnx is a framework built by Microsoft that allows different models to be ported from each other. For example, through Onnx our pytorch model that was in a .pt format was converted into a tensorflow format (.json) Once the model was imported into tensorflow.js and run on the mobile, a myriad of bugs were discovered. Due to technical difficulties in getting tensorflow.js to run on the phone through react native, another alternative solution was needed. This is when the team decided to build the ASL API which is an API endpoint that takes in an image and runs the model on a server and returns the output as the letter detected, space, delete or none respectively. This would mean after the, "Detect" button was clicked, the image from the mobile camera would be captured, the image would be uploaded to the server where the pre trained model would run and the result displayed on the app screen. This process would result in a latency time that depends on the speed of the internet connection and the speed that the model runs on the server. This route was decided based on time constraints to build a fully functional app.

D. ASL-NET API

An API is a system that allows applications to communicate with each other and usually involves some sort of data transfer either in, out or both. For example, the facebook API would allow access to data from users such as friends and liked pages (with the user's permission of course) In order to build an API, a server endpoint is required. The figure 15 shows how websites, android and ios apps can make use of APIs.

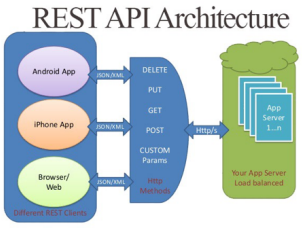

Fig. 15. The REST API architecture and the different type of calls possible.

The team had to decide the technology to be used for the API and there were two primary choices: Node.js or Flask. Node.js is a server environment written in JavaScript whereas Flask is a server environment written in Python. When writing in Node.js the tensorflow.js library would have to be used as it is the only stable machine learning library that can run on a node.js server. Node.js was first chosen as the server to be used and the pytorch model was converted into a tensorflow.js model which is a .json file using Microsoft's onnx platform. Once this was done it was done it was found to be a tedious task to keep converting from pytorch to tensorflow.js format when any updates were made to the model and this is when flask seemed to be a better choice for implementation. Because Flask is written in python and our primary models were trained on pytorch which is written on python, there would be almost native support of pytorch models on Flask with minimal code changes. This made development of the model to run on an API endpoint to be very efficient and easily deployed.

In order to first test the API the team built a web app interface that allows users to upload an image of a ASL letter sign and it would return the result after running the model on the image. This simple interface was built by Flask's template engine that allows serving of HTML pages. This simple interface can be seen in figure 16.

One of the advantages of building an API for the ASL detection is that it enables the API to be used to deploy on almost any platform as long as it can call the API. The exact steps that goes on in the backend are listed below:

1) Image is sent to API server endpoint through a POST upload method.
2) Server receives image and saves it using a temporary BLOB file format on the server.
3) The image is rescaled on the server
4) The .pt pytorch model is loaded
5) The model predict function is then run on the image
6) The model classification results in an array of 29 items where each has a precision value
7) This array is passed into the function the team built that takes the array as an argument
8) The function finds the max value in the array and returns the index of that item in the array
9) Then the function looks at another array predefines that has the values: ['a' ,'b' ,'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'space', 'delete', 'none']
10) Using the index of the max value the function then simply returns the index of this array above which would correspond to the letter / character detected. Note how when there is no letter detected, "none" is returned, and special characters such as "space" and "delete" are also included as they are important functionalities needed for

users to type and correct errors.

Another important decision made was to not save the image onto the server as it would cause both privacy concerns and exploit memory limitations. Initially the server had an image called predict.jpg and every time a post request it made to the API with the predict.jpg file is replaced with the image that was uploaded. The limitations to this method was that if multiple requests were to come at the same time it would definitely cause interference as the same file path is being called and may result in different classifications going to different users.

The first server hosting was done locally by hosting it on a team members laptop itself. This was done for the ease and fast debugging ability. The flask API system was then moved to the Heroku hosting service with the 1 Dyno plan as it allows relatively good GPU and CPU capabilities that would allow for the model to run. Once several tests were done on the Heroku hosting service it was noticed that the classification takes a significantly more time. It was deduced that Heroku would need more Dyno's which means it needs more GPU power to run the YOLO model more faster. This is when we started looking for alternative methods to run the model on an API server endpoint without having to spend too much money on resources. The team decided to work on a campus server that had a powerful GPU server. The campus server was set up to host the Flask server indefinitely until it is shut down and therefore when an image is uploaded using the POST method to the ip address of the server we were able to get the model to run slightly faster using the campus server.

E. ASL-NET APP

In order to connect the API to the app, the expo camera library needed to be imported as the image needed to be converted to a consistent jpg format as different android phones and iphones export in varying formats. The steps that take place in a normal use-case of the app in the backend are as follows:

1) User points mobile phone camera at the hand that is making the ASL letter and clicks the, "Detect" button
2) Phone captures image and saves it as a temporary blob file
3) This blob file is converted to a jpg format
4) Using React Native's inbuilt FormData() function, the data being the image is parsed into a form data format and then is posted directly to the server endpoint. This server endpoint changed from the local machine to the heroku server to finally the ip address of the campus server.
5) The model is run on the server and returns a result as one of the 26 classification letters, space, delete or none.
6) The result is then appended in the textbox at the center of the app. However when space or delete is returned this is parsed as a literal space character and the delete keyword is parsed as a delete function that deletes the last character in the textbox. The none character would be parsed as a simple dash, '-', to indicate that there was no classification made as no letter or character was detected.

Fig. 17. The image shows the app being used in real time to detect the letter, "A". The app was tested under different light conditions and background environments to see how it well it would work and it was satisfactory.

In order to make the images consistent as the data that the model was trained on was taken by the, "portrait", vertical mode in the mobile phone camera, it was decided to not allow the app to be horizontal but to stick to the vertical orientation alone. The final appearance of the application can be seen in figure 17.

F. Limitations

One of the biggest limitations that the implemented app has is speed. The model requires at an average from the click of the, "Detect" button to the result appended to the text box at the center of the screen to be around 7 seconds. This amount of time is not feasible as the communication of the user would not be seamless.

It was also quickly evident that the size of trained pytorch YOLO v3 model was astonishingly large, being at around 450mb plus which makes it harder to package the mobile app with the model itself and run the model on the phone. This large file size would first make it harder for users with a non broadband internet connection to download the app and also make the user less inclined to download the app due to its file size and the space it would take up on their phone.

G. Improvements

Numerous improvements can be made to improve the user experience as a whole for users of the ASL-NET app. Firstly a server with a more powerful set of GPU's would reduce the amount of time taken to run the model. The API as a whole could even be discarded and the model could be run on the phone, the only problem being the size of the

pretrained .pt file. The model would have to be converted into the tensorflow.js, ".json" format in order for it to work on react native but once it is debugged efficiently it is possible for it to run on the phone and this would increase the speed of classification dramatically. We are also considering to use a lighter model such as YOLO-Lite which can run on a mobile device without the need for GPU. [2]

REFERENCES

[1] B. Kang, S. Tripathi, and T. Q. Nguyen. Real-time sign language fingerspelling recognition using convolutional neural networks from depth map. *CoRR*, abs/1509.03001, 2015.

[2] J. Pedoeem and R. Huang. YOLO-LITE: A real-time object detection algorithm optimized for non-gpu computers. *CoRR*, abs/1811.05588, 2018.

[3] J. Wang and Z. Ye. An improved faster R-CNN approach for robust hand detection and classification in sign language. In X. Jiang and J.-N. Hwang, editors, *Tenth International Conference on Digital Image Processing (ICDIP 2018)*, volume 10806, pages 352 – 357. International Society for Optics and Photonics, SPIE, 2018.