Edge-based American Sign Language Detection

Dom Dillingham
Ajit Barhate
Julia Ying
W251 Deep Learning in the Cloud and at the Edge
Summer 2021
https://github.com/dom-dillingham/w251-final-project

Introduction

This project aims to create a system deployed on an edge device to translate American Sign Language (ASL) alphabet signed by a user. ASL is a complete natural language expressed with movements of the hands and face, possessing the same linguistic properties as spoken languages. It is the most common language used by North Americans who are deaf or hard of hearing [1]. As ASL uses different grammatical rules and signal systems from English, there is a communication barrier between non-ASL speakers and the deaf community. A machine learning system translating ASL to written English can support the speech and hearing-impaired community by providing a direct means of conveying intended messages without requiring the intended audience to learn ASL.

A common approach in prior research on ASL recognition utilizes Deep Learning architectures to classify hand signs into English letters. A study in 2017 using CNN to process photos of ASL alphabets achieved an accuracy of 82.5% [2]. In 2019, Kasukurthi et al., using a SqueezeNet architecture, achieved a slightly improved accuracy of 83.29% [3]. Other attempts to improve ASL recognition include detecting spelled sequences in videos [4], and using a transformer architecture to accomplish continuous sign language recognition and translation [5].

While also utilizing similar Deep Learning architectures as prior studies, this project proposes an end-to-end pipeline that deploys a model trained in the cloud on an edge device, to provide live translation of ASL. Figure 1 outlines the proposed overarching architecture. A camera feed connected to an edge device captures video of a user signing in ASL. The edge device then feeds still frames from the video stream into a Deep Learning model trained in the cloud and produces an inference, which is displayed to a second user.

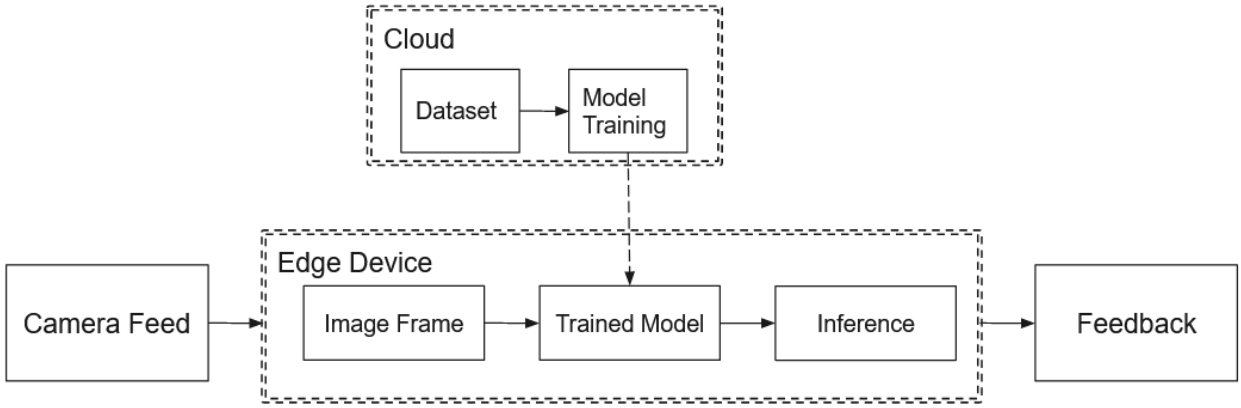

Figure 1. Architecture of proposed end-to-end pipeline for translating ASL live.

As a minimally viable product, we trained a convolutional neural network to predict the letters in the ASL alphabet and deployed it on a Jetson Xavier NX device to run inferences locally. We chose to focus on the alphabet letters only because fingerspelling accounts for a significant

portion of ASL (12 - 35%)[6], and there are readily available image datasets for the alphabets. This proof of concept can be scaled up to incorporate the full ASL vocabulary if training data is available.

Model Architecture

Our inference pipeline utilized a pre-trained EfficientNet-b5 fine-tuned on ASL alphabet images. EfficientNet, proposed by Tan and Le in 2019 [7], outperformed architectures such as ResNet on ImageNet while being orders of magnitude smaller than ResNet through the use of scaling methods.

EfficientNet is a family of models created by scaling up the baseline network created by performing a neural architecture search using the AutoML MNAS framework, which optimizes both accuracy and efficiency (FLOPS). The architecture used mobile inverted bottleneck convolution (MBConv), similar to MobileNetV2 and MnasNet [8].

Figure 2 illustrates the basic architecture of EfficientNet-b0. The b5 architecture expands on this base model, adding multiple units composed of global average pooling, rescaling, and 2 Conv2D layers [9]. In total, EfficientNet-b5 has 30 million parameters, a fraction of that of VGG19 (144 million parameters), one of the candidate models we evaluated during our model development process. EfficientNet's increased accuracy and faster inference time make it a suitable candidate for performing inference on the Jetson.

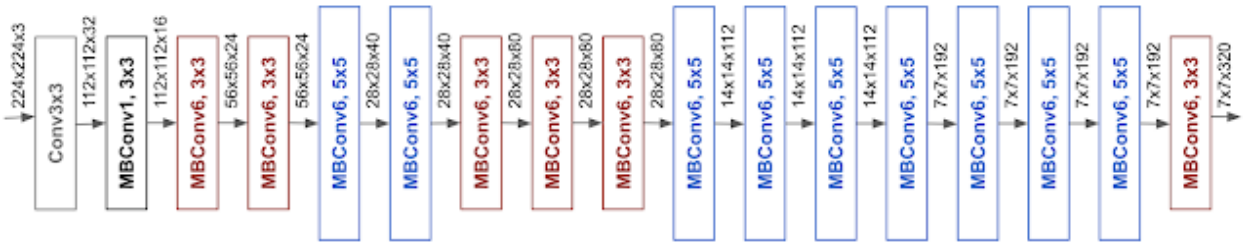

Figure 2: The architecture of baseline network EfficientNet-B0 [8].

Dataset

Six datasets from Kaggle were included in training and testing data [10-15]. We chose to incorporate datasets from multiple sources in hopes to increase variability in examples so our model can generalize well during inference in field deployment.

Out of these six datasets, two datasets [10][13] have large quantities of images, but the images within each class are highly similar. Figure 3 shows a sample of such highly similar images. In our earlier attempts to train a model for ASL translation, we chose one of these datasets for training due to its large number of images. A VGG19 model obtained more than 99% validation accuracy, yet produced very poor performance when making inferences on video inputs

received on the Jetson device. The model was clearly overfitting on a homogeneous set of images and was unable to process any images deviating from the training set. The simple solution to this was to mix multiple datasets to create one large, composite dataset. While individual datasets may lack variations, mixing multiple datasets would introduce divergent features, forcing the model to generalize.

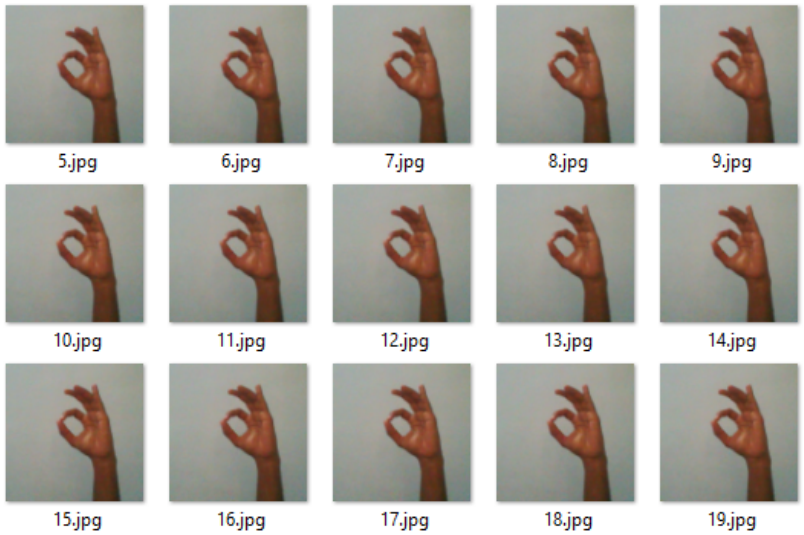

Figure 3: A sample of images from datasets lacking diversity. The images spell the letter F.

To avoid training loss being skewed by the two large datasets resulting in the model overfitting on a small range of hand gesture and positioning, for these two datasets, only 1000 images per class were randomly sampled and included in the training dataset. For the remaining datasets, additional manual processing removed erroneous images.

The composite dataset includes 201,486 images. With the exception of J and Z, each letter had approximately 8,000 instances of examples. The signs for J and Z include motion, and as such, some of the datasets omitted images for these letters. J had 2,061 examples and Z had 5,031 examples. Combined across the six datasets, the images varied in size, resolution, lighting, zoom, angle, and context. Figure 4 shows a sample of images in the final dataset showcasing a broad diversity of examples for the same letter. 80% of the images were used in training and 20% were used for validation.

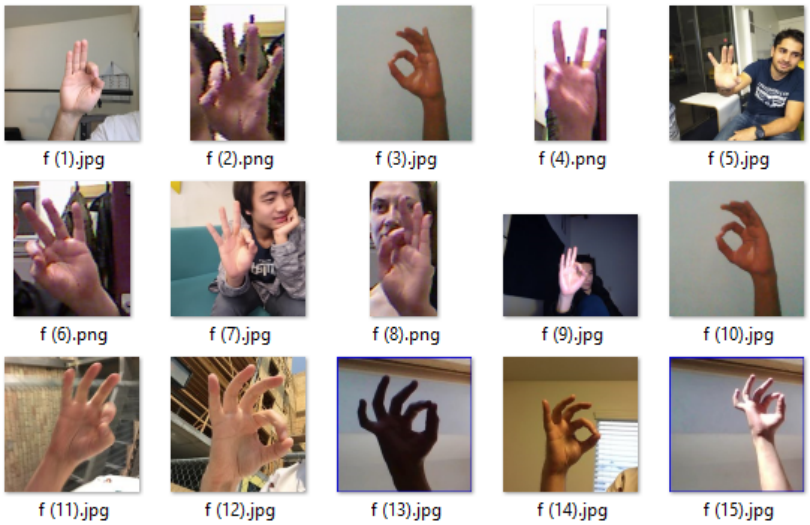

Figure 4: A sample of the composite dataset containing diverse features. The images spell the letter F.

Model Training

We used an AWS g4dn.xlarge instance with a NVIDIA T4 GPU to fine-tune the EfficientNet model on the composite ASL alphabet dataset. The model, built in Pytorch, was trained for 5 epochs with a batch size of 16 and 5-fold cross validation. Validation accuracy was calculated at the end of every epoch.

To ensure the model can generalize beyond images in the training dataset, we utilized image augmentation methods included in the TorchVision library. The training images underwent random transformations including random resized crop, random horizontal flip, random affine and random rotation.

We also used label smoothing of training samples and mix-up of training samples and labels. Cosine annealing scheduler was used for the learning rate with a fixed starting learning rate of 0.0001.

To compare the various variants of EfficientNet, we trained EfficientNet-b0 and b2 through b7 in parallel on multiple replicates of the aforementioned instance.

Model training results

As a baseline and also the smallest model, EfficientNet-b0 achieved a validation accuracy of 84.88%. Intuitively, having a larger network could help improving accuracy, and this trend was observed for EfficientNet-b2 through b6, each having better validation accuracy than its smaller counterparts. Contrary to expectation, EfficientNet-b7 only had a moderate accuracy, comparable to Efficientnet-b2. It is possible that the b7 model is too large for the size of our

data, or the large model may require additional adjustments in training procedures for better results. Table 1 contains the training results for the 7 EfficientNet models.

Model	# Params	Val accuracy
efficientnet-b0	5.3M	84.88 %
efficientnet-b2	9.2M	86.20 %
efficientnet-b3	12M	87.35 %
efficientnet-b4	19M	88.79 %
efficientnet-b5	30M	89.66 %
efficientnet-b6	43M	89.93 %
efficientnet-b7	66M	86.94 %

Table 1: Training results of various versions of EfficientNet models

Our final choice of model for the inference pipe is EfficientNet-b5. While b6 had the best accuracy, b5 is only marginally worse in terms of validation accuracy (89.66% versus 89.93%), but is about 25% smaller. Given the model size will have a tangible impact on the edge device's speed performance, b5 is the better model for deployment.

Error Analysis

We analyzed EfficientNet-b5's predictions on the validation set. Figure 5 is the confusion matrix on the 26 letters in the alphabet.

The most common confusions are: M predicted as N, K predicted as V, O predicted as M. Figure 6 shows images of the three most commonly confused pairs. As seen in the photos, confusing the letter "M" for "N" is understandable given the signs for the two letters are reasonably similar. A similar argument can be made for confusing K and V, observing the minor differences between the two, and the sign for O has a similar handshape as M but from a different angle.

Surprisingly, the model does not confuse these letters the other way around. For example, N is not often predicted as M. The asymmetrical confusion pattern may suggest that the convolutional network is sensitive to only certain features.

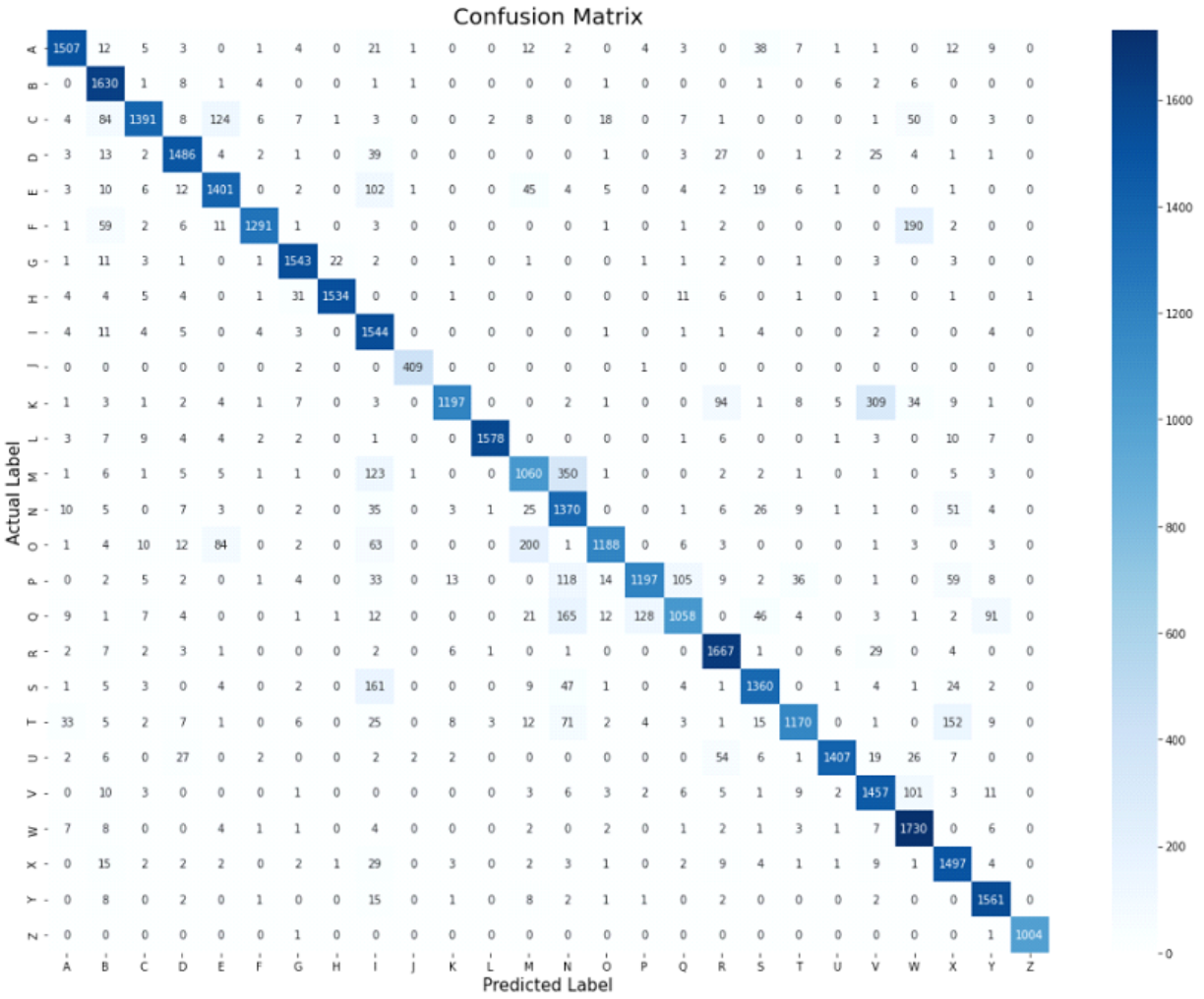

Figure 5: confusion matrix of the validation dataset predictions by Efficientnet-b5

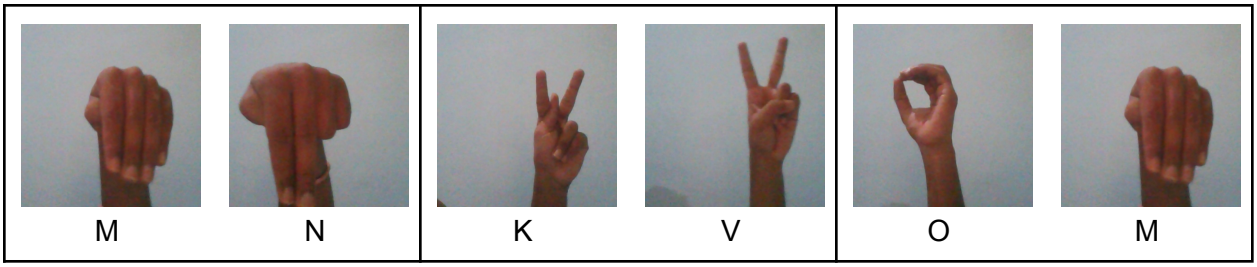

Figure 6: The top three most commonly confused images.

Implementation and Deployment

Inference Architecture

The architecture of our end-to-end pipeline deployed on the NVIDIA Jetson Xavier NX is depicted in Figure 7. The system takes a live video stream from a USB camera feed, the frames of which are fed to a Docker container running on the Jetson device. When the user begins the prediction process, OpenCV CSRT object tracking is used to follow the movement of the hand. The image of the hand is cropped and inputted into a fine-tuned EfficientNet model, which classifies the images and produces a letter prediction to be displayed on the OpenCV display. The user monitoring the display may suggest a different prediction, in which case the image and user suggestion is sent an S3 bucket where additional training samples may be housed.

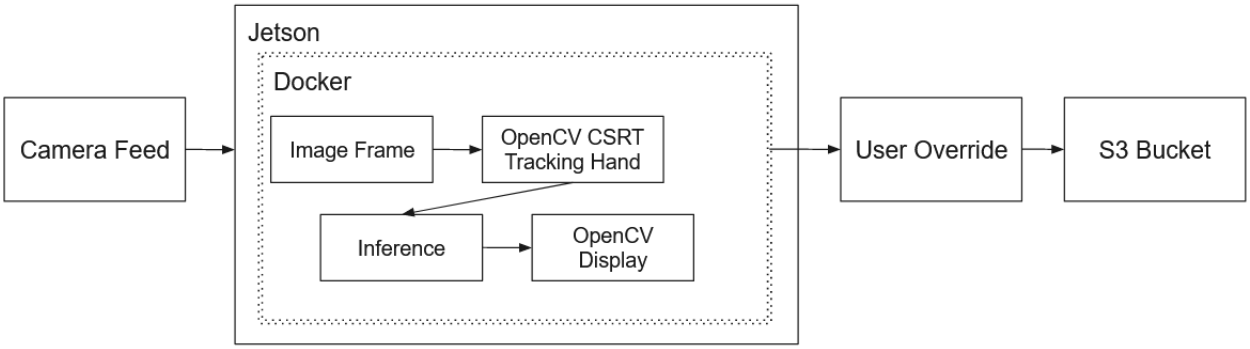

Figure 7: Architecture of the deployed pipeline

Model Quantization

We ran static quantization on the final trained model to improve runtime inferencing on the Jetson device. Running the pipeline using a model with and without quantization showed a marked difference in performance. Compared to the unquantized model, making inferences using the quantized model doubled the frame rate in the output video.

User Interface and Interactions

The user interface utilizes OpenCV's built-in methods. Videostream from the webcam is horizontally mirrored and displayed live. A stationary 300x300 pixel yellow square is outlined in the center of the image feed. A user can initiate hand tracking and inference by placing one hand in the yellow square and press 1. The yellow box is then replaced by a moving square bounding box outlined in blue tracking the user's hand, and the model's inference is displayed on the upper left hand corner of the screen, updating every 0.5 seconds. The user can press 2 at any time to stop the hand tracking and inference process. With the current configuration, our system runs at roughly 4 frames per second during inference. Figure 8 shows the interface while the inference process is off and on, respectively.

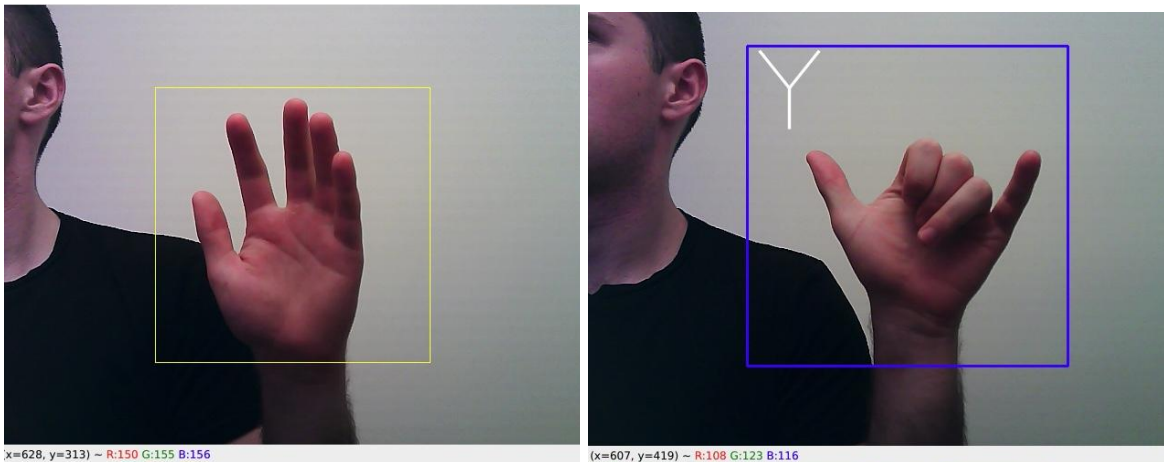

Figure 8: Interface when hand tracking and inference is not active (left) and active (right)

During the tracking process, the user can press any key from A to Z to indicate the letter currently being signaled. If the key does not match the model's inference, the interface captures a screenshot of the current frame, crops out the area within the bounding box, and uploads to an Amazon S3 bucket using the boto3 library. The filename is the timestamp prefixed with the user inputted letter. These images can be used in future model training and fine tuning to help the model recognize the signs it had initially mistaken.

Hand Tracking

To reduce noise in the image input for the model during inference, we opted to use a bounding box to crop out only the hand. This posed a distinct technical challenge. Unlike frontal face detection, hand detection cannot utilize a cascading classifier because the shape, size, and angle of the hand change depending on the gesture. For instance, a single cascading classifier cannot recognize both an open hand and closed fist.

Among alternatives to a cascading classifier, the current industry leading solution to hand detection is the MediaPipe library [16], which can detect landmark points in a moving hand in real time without a GPU. However, the prebuilt version of Mediapipe is not compatible with ARM processors and thus cannot be implemented on the Jetson device.

We had further explored other possibilities for hand tracking and bounding boxes. Available pretrained DNNs such as the one provided by Openpose [17] can achieve a similar result as MediaPipe. However, in our test of the implementation, OpenCV's DNN module could only make inference at approximately one frame every two seconds, an unacceptable speed for live ASL sign translation. Other solutions such as YOLOv5 and Jetson Inference library were also considered and dismissed due to difficulty integrating them into our pipeline.

Our eventual solution was to use OpenCV's object tracking algorithm. This requires an added user interaction step that places the hand into a pre-designated area and initiates tracking. The built-in CSRT tracker achieved a decent performance at an acceptable speed. One notable

disadvantage of using an object tracker instead of a specifically trained hand detector is that when the hand is moving, the CSRT tracker occasionally fixes onto a background object and loses tracking of the hand. To mediate this, we applied a skin filter [18] to mask colors outside the typical range of human skin tone in order to eliminate background distractions for the tracker. Figure 9 shows a frame after the skin filter is applied. Note that the masked image is only used for the tracker to produce a bounding box. The unfiltered image is used as input for the EfficientNet model.

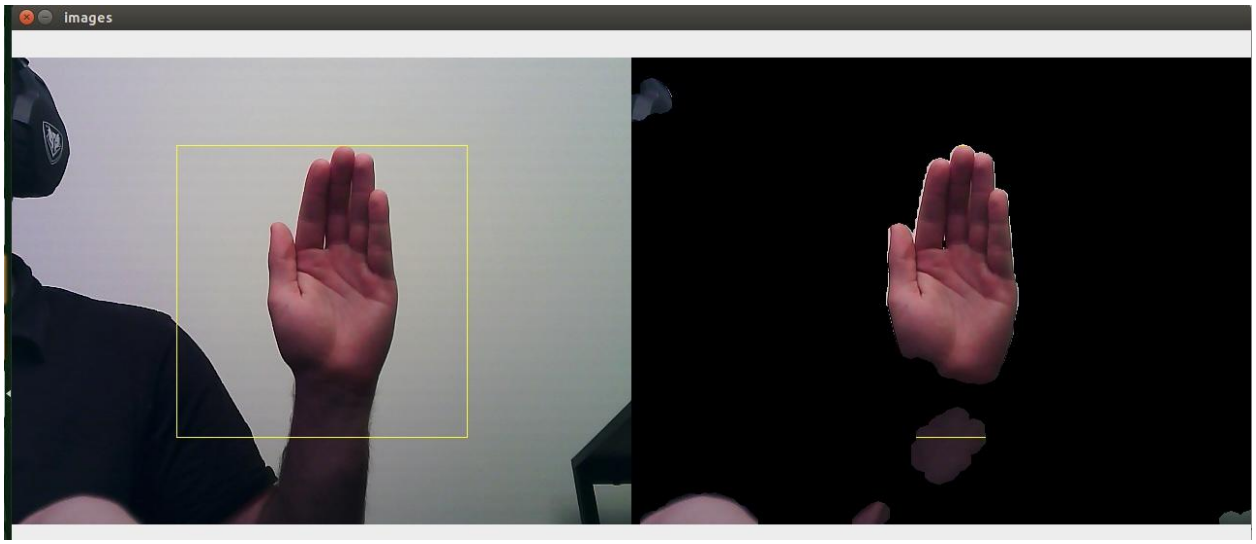

Figure 9: A frame of the live feed received from the camera (left) and the same frame after the skin filter is applied (right).

Input Image Processing

Prior to inference, the bounded object tracking image is transformed to the training input size of 224x224. A tensor of the resized image is then sent to Jetson's GPU for inference on the EfficientNet model. The prediction from the model is then returned to the CPU and displayed on the OpenCV display that is displaying the input frames.

Docker Container

The container on the Jetson is based on the NVIDIA L4T PyTorch image using version r32.4.3-pth1.6-py3 for the JetPack 4.4. The final container image is hosted on DockerHub as ddil2149/final-jetson-local. When running the Docker container, an appropriate AWS iam secret key and value must be provided to ensure that images can be uploaded to an appropriate S3 bucket.

Conclusions

In this project, we deployed an end-to-end pipeline on an Edge device that translates ASL alphabet in near real time using a convolutional neural network trained in the cloud. We note that diversity in training data vastly improved prediction accuracy.

We have shown using this minimal viable product that deep learning models can be deployed on Edge devices to assist in translating ASL and facilitating communication with the deaf community. The choice of EfficientNet's relatively compact architecture, coupled with model quantization, produced acceptable performance on the edge, though it still needs improvement to achieve true real-time inferences.

Limitations

While successfully deployed, the final pipeline produced in this project does have limitations. Foremost, the pipeline is constrained to just the ASL alphabet. There currently exists a lack of comprehensive training data that encompases the full ASL language on which to develop a corresponding pipeline. As an extension of the above, certain letters and many words require motion to sign. The current pipeline is only designed to handle static frames and, as much, may struggle when the complexity of the language to translate increases.

With regards to the existing pipeline, the speed and accuracy of the model provide some limitations. At three frames per second, the inference speed of the pipeline is far slower than human comprehension. Increased efficiencies to model architecture and faster tracking object tracking would assist in increasing inference speed. The accuracy of the model, as previously described, could also be improved as some letters tend to be confused with similar ones. Potentially increasing the sample size of these confused letters may increase accuracy. Finally, the skin filter applied during object tracking may be limiting the pipeline in many scenarios. The skin filter requires appropriate lighting to track the hand and was designed to work with the hands of the project team which may not scale to the general population of lighting scenarios and skin tones.

Future Work

One potential followup to this project is to expand the inference capabilities to signs for full words, as well as letters involving motion. This would also prompt a revised UI to allow for the user to give more complex inputs.

Given that our current interface already captures erroneously predicted images along with user feedback, we should implement a way for periodically updating the models, training on the user-corrected images.

While EfficientNet is a highly accurate and faster model, other state of the art models should also be considered to find an optimal model with the best balance in terms of speed of performance and prediction accuracy. Additional data augmentation techniques should also be explored to enable the model to capture the diversity in sign language that would manifest in practical use during inference.

References

[1] U.S. Department of Health and Human Services. (n.d.). *American sign language*. National Institute of Deafness and Other Communication Disorders. https://www.nidcd.nih.gov/health/american-sign-language.

[2] Vivek Bheda and Dianna Radpour (2017). Using Deep Convolutional Networks for Gesture Recognition in American Sign Language*. CoRR, abs/1710.06836.*

[3] Nikhil Kasukurthi, Brij Rokad, Shiv Bidani and Aju Dennisan (2019). American Sign Language Alphabet Recognition using Deep Learning*. CoRR, abs/1905.05487.*

[4] Bowen Shi, Diane Brentari and Greg Shakhnarovich and Karen Livescu (2021). Fingerspelling Detection in American Sign Language*. CoRR, abs/2104.01291.*

[5] Necati Cihan Camgoz, Oscar Koller, Simon Hadfield, & Richard Bowden. (2020). Sign Language Transformers: Joint End-to-end Sign Language Recognition and Translation.

[6] Carol A. Padden and Darline C. Gunsals. How the alphabet came to be used in a sign language. Sign Language Studies, 4(1):10–13, 2003. 1

[7] Mingxing Tan and Quoc V. Le (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. CoRR, abs/1905.11946.*

[8] *EfficientNet: Improving accuracy and efficiency THROUGH AutoML and model scaling*. Google AI Blog. (2019, May 29). https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html.

[9] Agarwal, V. (2020, May 31). *Complete architectural details of all efficientnet models*. Medium. https://towardsdatascience.com/complete-architectural-details-of-all-efficientnet-models-5fd5b736142.

[10] https://www.kaggle.com/grassknoted/asl-alphabet

[11] https://www.kaggle.com/danrasband/asl-alphabet-test

[12] https://www.kaggle.com/mrgeislinger/asl-rgb-depth-fingerspelling-spelling-it-out

[13] https://www.kaggle.com/kapillondhe/american-sign-language?select=ASL_Dataset

[14] https://www.kaggle.com/kuzivakwashe/significant-asl-sign-language-alphabet-dataset

[15] https://www.kaggle.com/signnteam/asl-sign-language-pictures-minus-j-z

[16] https://google.github.io/mediapipe/

[17] https://github.com/CMU-Perceptual-Computing-Lab/openpose

[18] Adrian Rosebrock. (2021, April 17). *Tutorial: Skin detection example using python and opencv*. PyImageSearch. https://www.pyimagesearch.com/2014/08/18/skin-detection-step-step-example-using-python-opencv/.