# W251 Final Project - American Sign Language (ASL) to Audio

Autobot: Curtis Lin, Satya Thiruvallur, Param Viswanathan, Hailey Wu

## Section I: Project Introduction

There are 72 millions people around the globe that have conversational disabilities, some with congenital defects while others from acquired diseases and accidents. Many of them are part of our families and our loved ones. When it comes to communication, sign languages are developed to facilitate the deaf-mute community. 47 countries have acknowledged the sign language as one of the official languages.

American Sign Language (ASL) is the standard natural language that serves as the predominant sign language in US and most English-speaking countries. Each alphabet is represented with one unique hand gesture. Additionally, certain datasets also have signs for special words, such as "nothing," to enhance the language processing on the computer.

Nevertheless, even with the standardization of ASL, it still has many barriers for communication with the deaf-mute community. It first requires both parties (or multiple participants) to understand and use sign languages as the prerequisite. It is a time effort to practice a new language. Moreover, the majority does not have the motivation to learn the ASL. Unless one has an affinity to the disabled group, it will not be the priority to learn gestures. This brings the second barrier where the possibilities for two strangers to communicate in sign language is very minimum. In business context or education, this disadvantage can alienate the deaf-mute community or force them to accommodate the norm with more effort.

The Autobot team wants to overcome challenges for the deaf-mute community, by converting the sign languages directly to audio. When a person shows hand gestures in front of the camera, in real time, these alphabets will be converted into words and then sentences in audio. In this way, it will ameliorate the effort for the disabled to give their opinions.

This project is standing on the shoulders of giants. The following few necessities might not be available a few years back, but now are commodities for researchers. We also use these technologies in this project:

1. Cloud compute power for big data modelling
2. Fast-processing edge device for real-time inference
3. Pre-built image models with transformation for individual scenarios
4. Microservices and reusable models for faster development
5. Strong image recognition communities for ideas

This project has defined the scope of work with considerations of time and cost:

- The project focuses on English. However, with the capability of real-time translation, the tool can accommodate other languages in the future.
- The project only supports one direction of the communication: from ASL to audio. It does not cover the other portion, from spoken English to gestures. There are many potentials of furthering this project to help the community better.
- The project uses Jetson TX2 for inference. However, in real life, it will be more helpful to deploy this app to mobile devices.

In the following sections, we will walk through the process, the challenge, and things that could be enhanced in the future.

## Section II: Dataset

### Data evaluation and selection

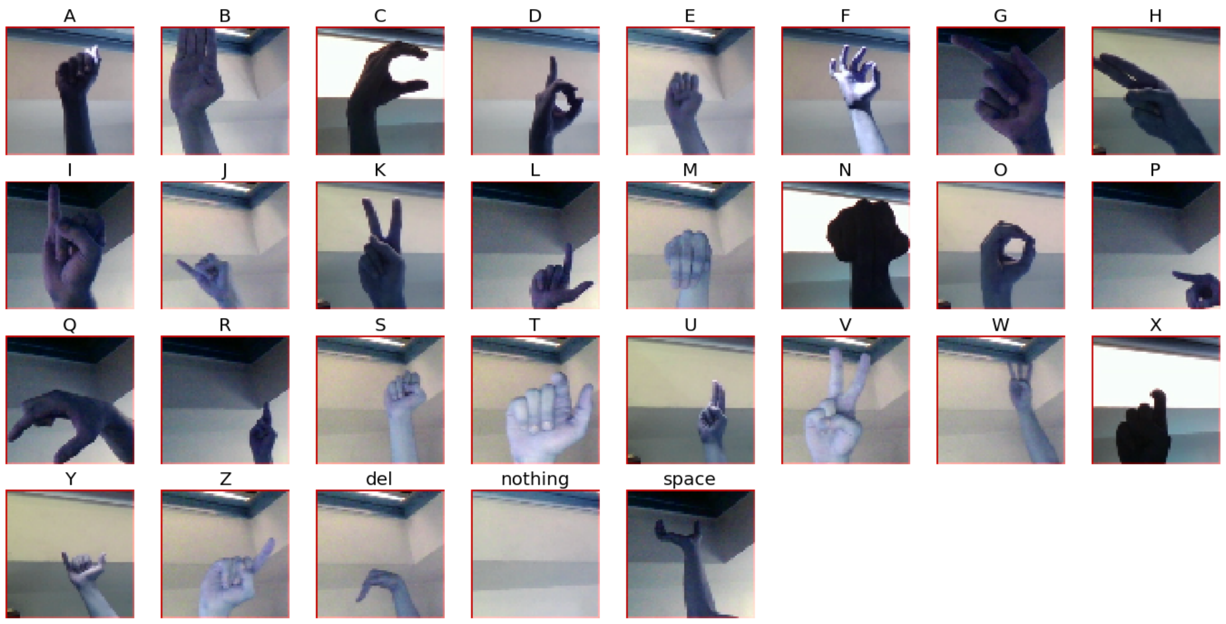
To make autobot feasible, we were considering two major types of sign languages, including word level and letter level. We first evaluated Word-Level American Sign Language (WLASL) video dataset[1]. The WLASL dataset contains 2000 words and 21083 sign videos, performed by 119 signers[2]. However, we faced several challenges with using this dataset: 1) many video urls were unavailable 2) mixed video formats (.mp4 and .swf file ) increased the processing complexity, 3) word level sign languages depend on hand movements, which means frame dependency and body languages need to be considered for deep learning training. Therefore, we decided to use the ASL alphabet dataset for autobot sign language recognition[3].

### EDA of ASL Alphabet dataset[4]

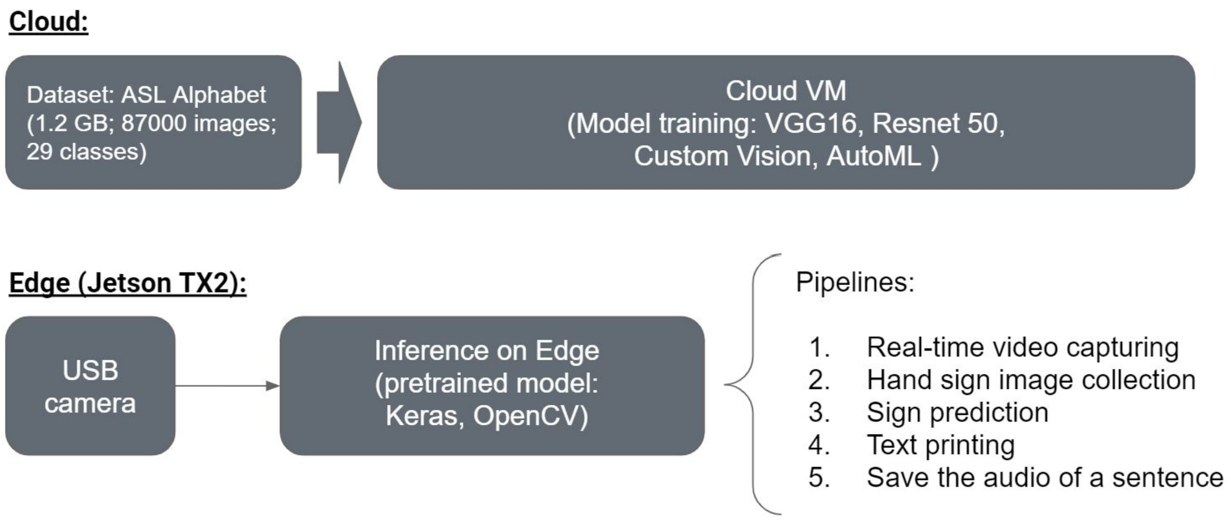
ASL Alphabet dataset including asl_alphabet_train (1.2G) and asl_alphabet_test (368K). The data contains 29 symbols, including ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'del', 'nothing', 'space']

The training dataset contains 29 folders and 87,000 sign images. Each image is 200 in width and 200 in height. The figure below illustrates a sample of symbol images.

[1] https://dxli94.github.io/WLASL/
[2] *WLASL_dataset_exam.ipynb* in project Github
[3] https://www.kaggle.com/grassknoted/asl-alphabet
[4] *EDA_ASL_Alphabet_dataset.ipynb* in project Github



## Section III: Pipelines


Cloud:
Dataset: ASL Alphabet (1.2 GB; 87000 images; 29 classes)
Cloud VM (Model training: VGG16, Resnet 50, Custom Vision, AutoML )
Edge (Jetson TX2):
USB camera
Inference on Edge (pretrained model: Keras, OpenCV)
Pipelines:
1. Real-time video capturing
2. Hand sign image collection
3. Sign prediction
4. Text printing
5. Save the audio of a sentence

## Section VI: Edge

On the Jetson TX devices, we developed the pipeline to utilize the model built in the cloud. The idea is a real-time processing of videos with low latency. While choosing the libraries, we tried to find light-weight packages and simple methods to make our pipeline faster.

The pipeline has five major steps:

1. Read the real-time video input
2. Cut out the hands portion from the frame
3. Conduct inference on gesture
4. Return the text
5. Return the audio of a sentence

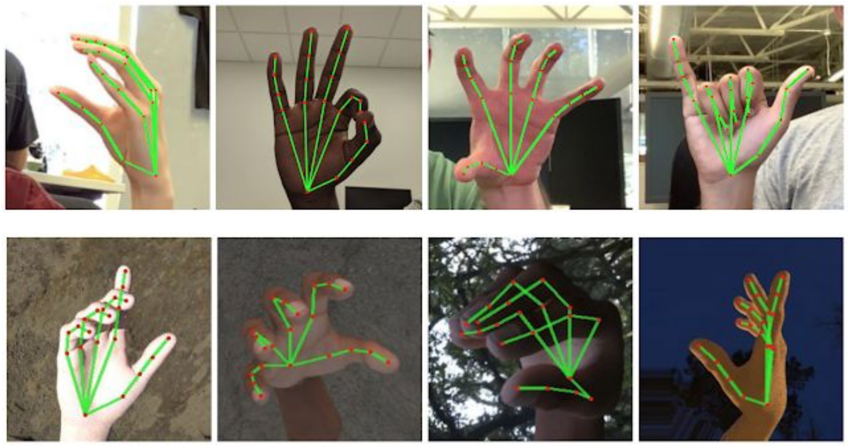
## Read the real-time video input

We continued to use the cv2 library from all previous homeworks and labs. The frame-reading speed matched our need for real-time requirements, and all team members had sufficient exploration of this library, which gave us more time to explore other steps of the pipeline.

```
# Camera
camera = cv2.VideoCapture(1)
camera.set(10, 200)

while camera.isOpened():
    ret, frame = camera.read()
```

## Cut out the hands portion from the frame

We first looked at the Google real time hand detection. It uses the hand landmark model, which can give higher accuracy for gesture detection. With 95.7% accuracy and low latency, the technology is promising for this project. Nevertheless, two concerns arose during our experiment. Using this real time palm detection meant that we needed to train our dataset into the hand landmark model, where each hand has precise keypoint localization of 21 3D hand-knuckle coordinates inside the detected hand regions. However, our original dataset did not have the best quality, and it might need individual photo tuning for good results. While measuring the effort and benefit out of this exercise, we believed this could be an overachiever for our scope of work. This will be an additional layer of inference for the pipeline and will have higher requirements for the device.



Therefore, we moved to cutting a certain window of the frame as the hand detection. It did have more restriction for users on where to put their hands, but on a positive side, it reduced the additional inference for hand detections and consumed very minimum time for the overall pipeline.

```
# crop ROI
img = img[0:int(cap_region_y_end * frame.shape[0]),
      int(cap_region_x_begin * frame.shape[1]):frame.shape[1]]
```

## Conduct inference on gesture

Tensorflow SavedModel gave us much convenience to switch between the few models we built for gestures. Initially, it took some time to understand the difference between Tensorflow 1.x and 2.x. But once we understood how to load the saved model in 2.x, everything worked great with just a few lines of code. There were challenges for the model to understand our gesture, majorly due to the quality issue of the dataset. For example, it will be easier to detect under a dark light and also with the left hand, because this is how the dataset looks. Further data engineering, such as image augmentation and photo tuning, can make the model more general.

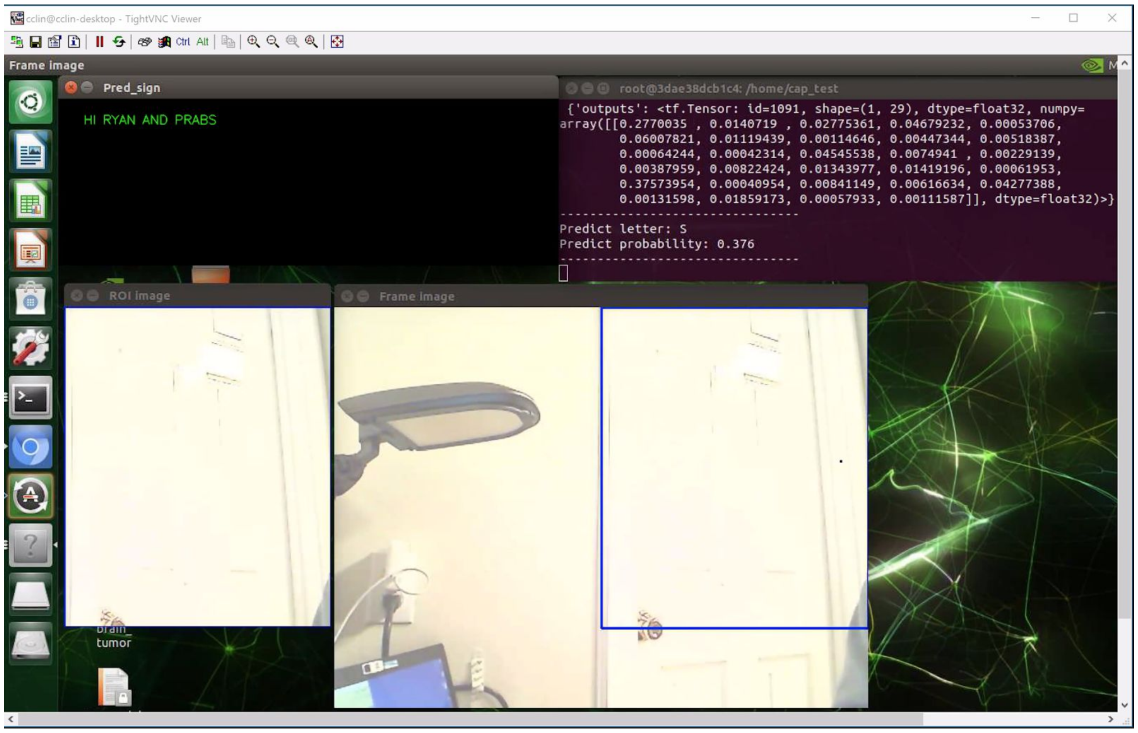
```
# load the pre-trained model (.pb files)
PATH_TO_FROZEN_GRAPH = './saved_model'
detection_graph = tf.saved_model.load(PATH_TO_FROZEN_GRAPH)
infer = detection_graph.signatures["serving_default"]
```

## Return the text

In this step, the pipeline returns the letter with the largest probability. At the beginning, the device continuously conducted inference estimation, similar to how we processed face detection in homeworks. And the alphabet would be printed on the final dashboard if the video detected an empty frame. In other words, once the user sees the right result, he or she will remove the hands from the camera to let the application return that alphabet.

```python
# extract the letter with the largest probability
max_pred = np.argmax(labeling.get('outputs'))
pred_acc = round(np.amax(labeling.get('outputs')), 3)
pred_sign = gesture_names.get(max_pred)
print("--------------------------------")
print("Predict letter: %s" %str(pred_sign))
print("Predict probability: %s" %str(pred_acc))
print("--------------------------------")
```

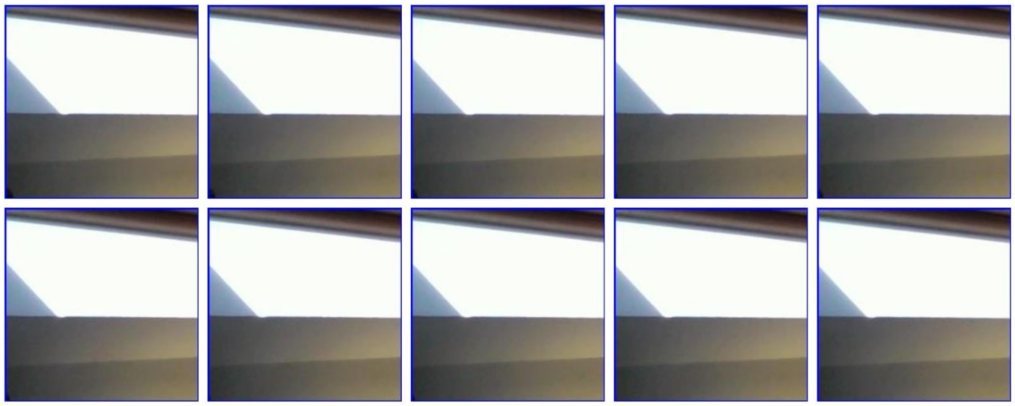
It soon became challenging because inference was made per second, and it was easy to miss the correct one. We then switched to use keyboard interaction. The device would not guess inference unless the user pressed "space." In this way, we saved a lot of compute power, and the user would have enough time to prepare his/her gesture.


cclin@cclin-desktop - TightVNC Viewer
Frame image
Pred_sign
HI RYAN AND PRABS
root@3dae38dcb1c4: /home/cap_test
ROI image
Frame image
brain_
tumor

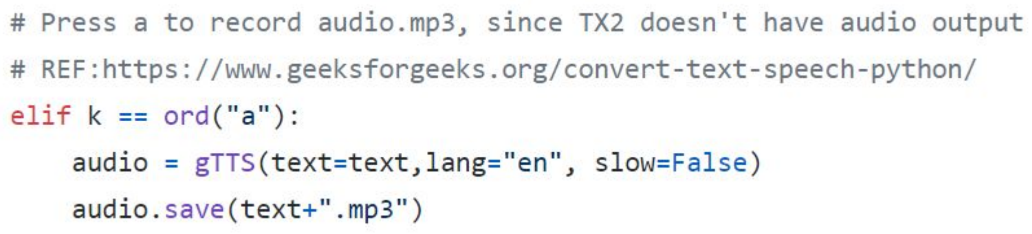
```
 {'outputs': <tf.Tensor: id=1091, shape=(1, 29), dtype=float32, numpy=
array([[0.2770035 , 0.0140719 , 0.02775361, 0.04679232, 0.00053706,
        0.06007821, 0.01119439, 0.00114646, 0.00447344, 0.00518387,
        0.00064244, 0.00042314, 0.04545538, 0.0074941 , 0.00229139,
        0.00387959, 0.00822424, 0.01343977, 0.01419196, 0.00061953,
        0.37573954, 0.00040954, 0.00841149, 0.00616634, 0.04277388,
        0.00131598, 0.01859173, 0.00057933, 0.00111587]], dtype=float32)>}
--------------------------------
Predict letter: S
Predict probability: 0.376
--------------------------------
```

## Return the audio of a sentence

Initially, our goal was to use certain images to tell the computer as a stop and produce the audio. The class “nothing” was a good candidate to be considered as a sentence period. As shown below, nothing is an empty backwall in the dataset.



The challenge we encountered was that it was difficult to decide the seconds waiting before producing the audio from text. Each person has a different pace. Especially when we were unfamiliar with sign languages, it took longer for the team to make gestures. We assumed if it was a concern for us, it would be for few end users as well. Therefore, we made the decision to have keyboard interaction to decide the time for producing the audio. This would give sufficient time for anyone to finish a sentence.



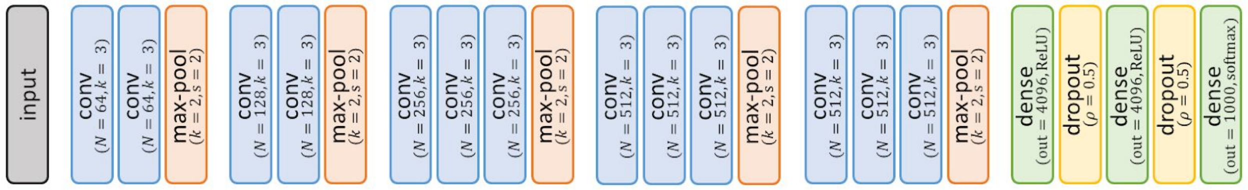
```
# Press a to record audio.mp3, since TX2 doesn't have audio output
# REF:https://www.geeksforgeeks.org/convert-text-speech-python/
elif k == ord("a"):
    audio = gTTS(text=text,lang="en", slow=False)
    audio.save(text+".mp3")
```

As shown above, we used the library gTTS. Surprisingly, the pronunciation and speed are great for real-time scenarios. However, another challenge was concern with audio output for Jetson TX2. Though we try to use both the built-in audio and bluetooth method, the device failed to play the audio. The alternative was to save the mp3. In the future, if the application is moved to a mobile device (e.g. iPhone), then there can be better audio playing for the result.

## Section V: Cloud Modelling

We pursued the technique of transfer learning to leverage state of the art CNN architectures, such as VGG16, Resnet 50, and automated machine learning platforms, such as Custom Vision and AutoML, to work with our ASL data. We will discuss each approach in detail below.

### VGG-16



The VGG16 architecture is so named for the trainable layers that are stacked together: 13 convolution layers + 3 dense layers.

#### Model Set up

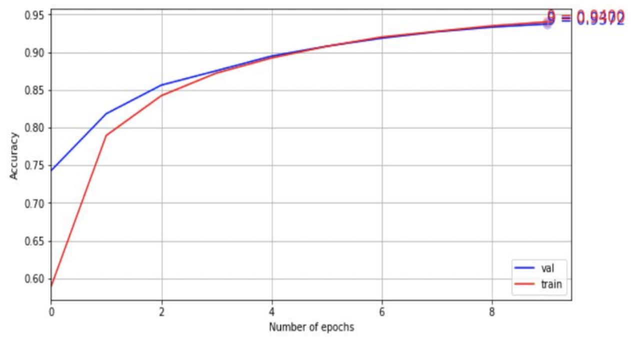
The model was trained with Tensorflow 1.15.2 using Keras. The dataset was resized into 50 X 50 during the pre-processing step and was divided into 80% train and 20% test respectively.

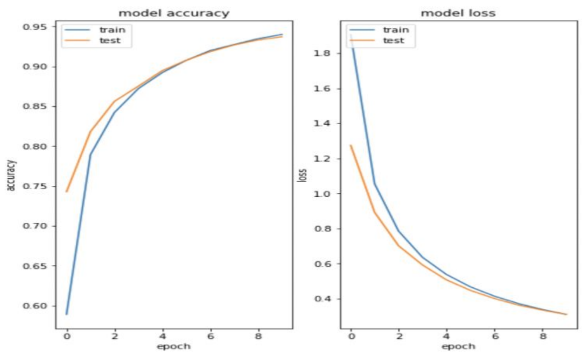
The target labels (y_train,y_test) were one hot encoded into integer values for the 29 class labels. The dataset was shuffled with random_state 13 to ensure consistency across training attempts.

The dimension ordered pre-trained weights for VGG-16 were used without the top layer to train our model. Model was compiled with Adam optimizer and categorical-crossentropy was chosen as the loss function since this was a supervised classification algorithm.

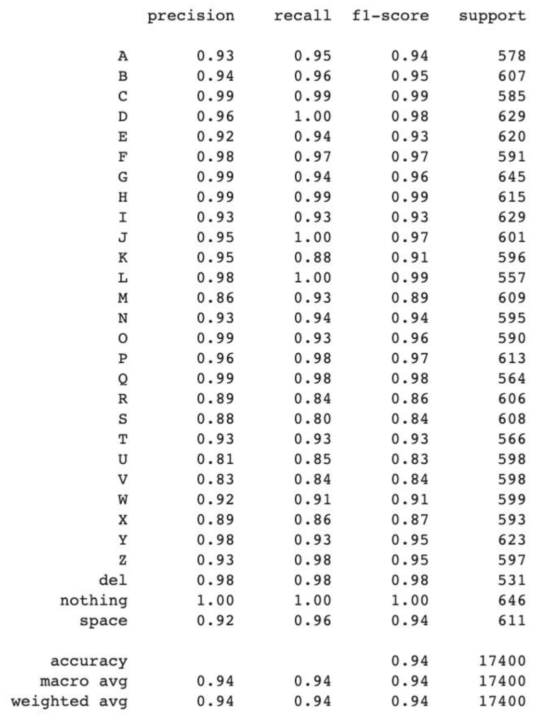
#### Training

Model was trained for 10 epochs  and reached a maximum accuracy of 94% on validation data set. The training process took around 60 minutes on Google Colab using the CPU mode. The best model was converted into pb format ( Protocol buffers) for inference on the edge.

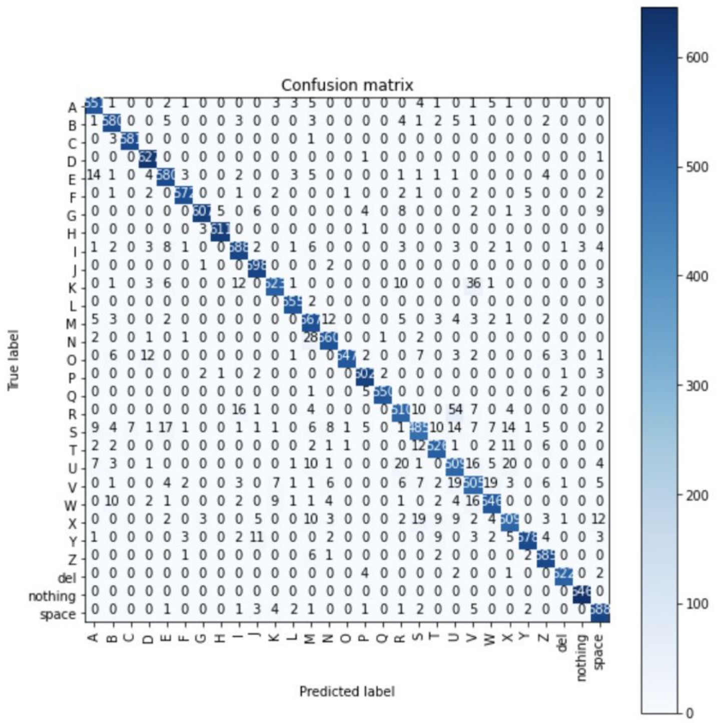




## Evaluation



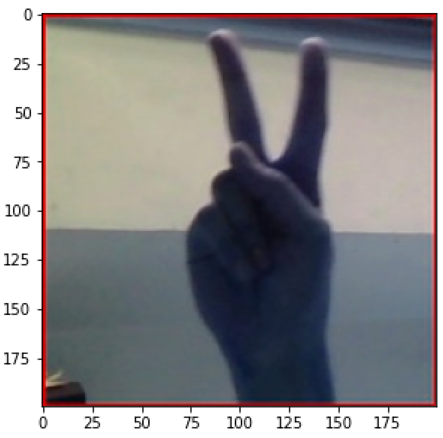
```
              precision    recall  f1-score   support

           A       0.93      0.95      0.94       578
           B       0.94      0.96      0.95       607
           C       0.99      0.99      0.99       585
           D       0.96      1.00      0.98       629
           E       0.92      0.94      0.93       620
           F       0.98      0.97      0.97       591
           G       0.99      0.94      0.96       645
           H       0.99      0.99      0.99       615
           I       0.93      0.93      0.93       629
           J       0.95      1.00      0.97       601
           K       0.95      0.88      0.91       596
           L       0.98      1.00      0.99       557
           M       0.86      0.93      0.89       609
           N       0.93      0.94      0.94       595
           O       0.99      0.93      0.96       590
           P       0.96      0.98      0.97       613
           Q       0.99      0.98      0.98       564
           R       0.89      0.84      0.86       606
           S       0.88      0.80      0.84       608
           T       0.93      0.93      0.93       566
           U       0.81      0.85      0.83       598
           V       0.83      0.84      0.84       598
           W       0.92      0.91      0.91       599
           X       0.89      0.86      0.87       593
           Y       0.98      0.93      0.95       623
           Z       0.93      0.98      0.95       597
         del       0.98      0.98      0.98       531
     nothing       1.00      1.00      1.00       646
       space       0.92      0.96      0.94       611

    accuracy                           0.94     17400
   macro avg       0.94      0.94      0.94     17400
weighted avg       0.94      0.94      0.94     17400
```



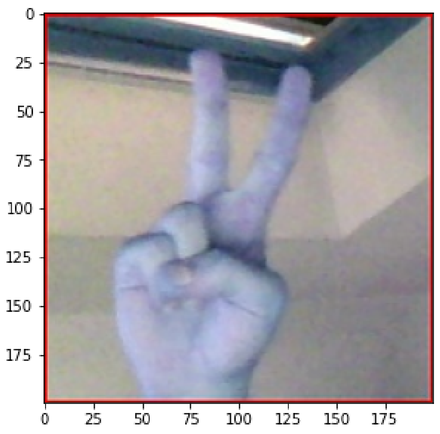
*Picture 1: Listing the class-level precision, recall and F-1 score, overall accuracy; Picture 2: Confusion matrix*

The table in **Picture 1** presents the overall metrics and per character class metrics. While the overall accuracy achieved was 94%, we see that characters ‘C’, ‘G’ , ‘H’ and ‘Q’ performed the best while the characters ‘U’ and ‘V’ had the lowest scores.

This information is also corroborated in the Confusion matrix in **Picture 2**. We see characters ‘K’ and ‘V’ are often mixed up , similarly for characters ‘R’ and ‘U’.  Listing the pictures below for ‘K’ and ‘V’ characters to show that these signs are indeed quite similar resulting in the confusion.



*Picture 3: Character 'K'*



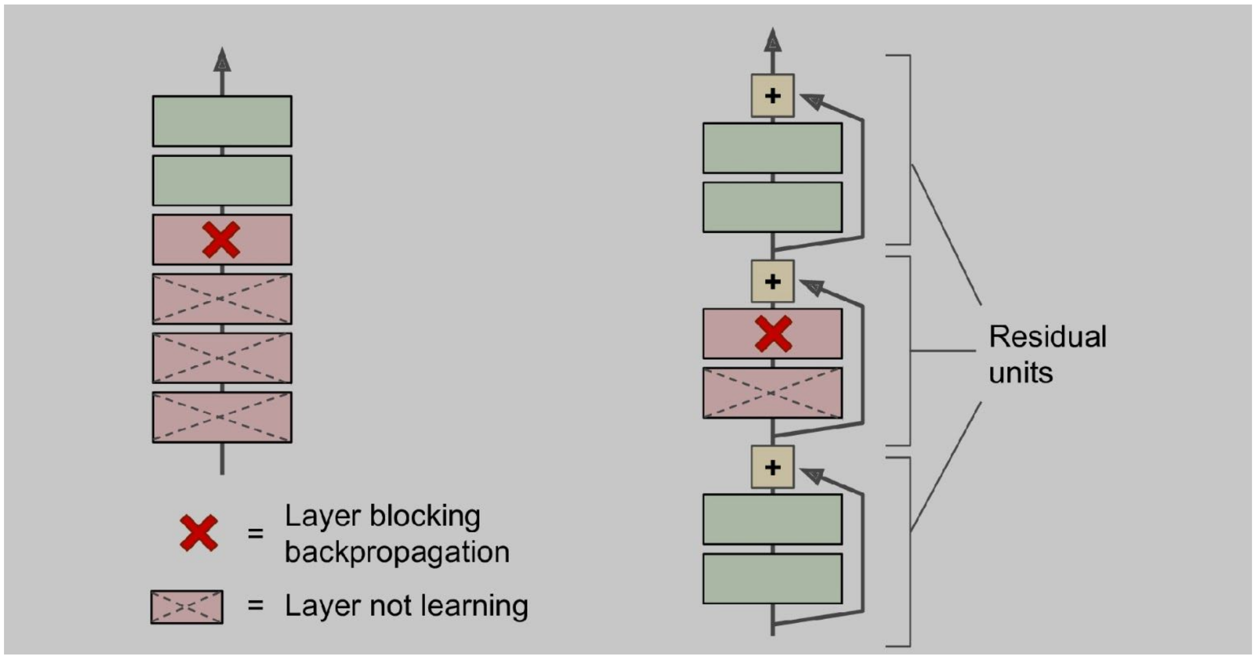
*Picture 4: Character 'V'*

## ResNet50

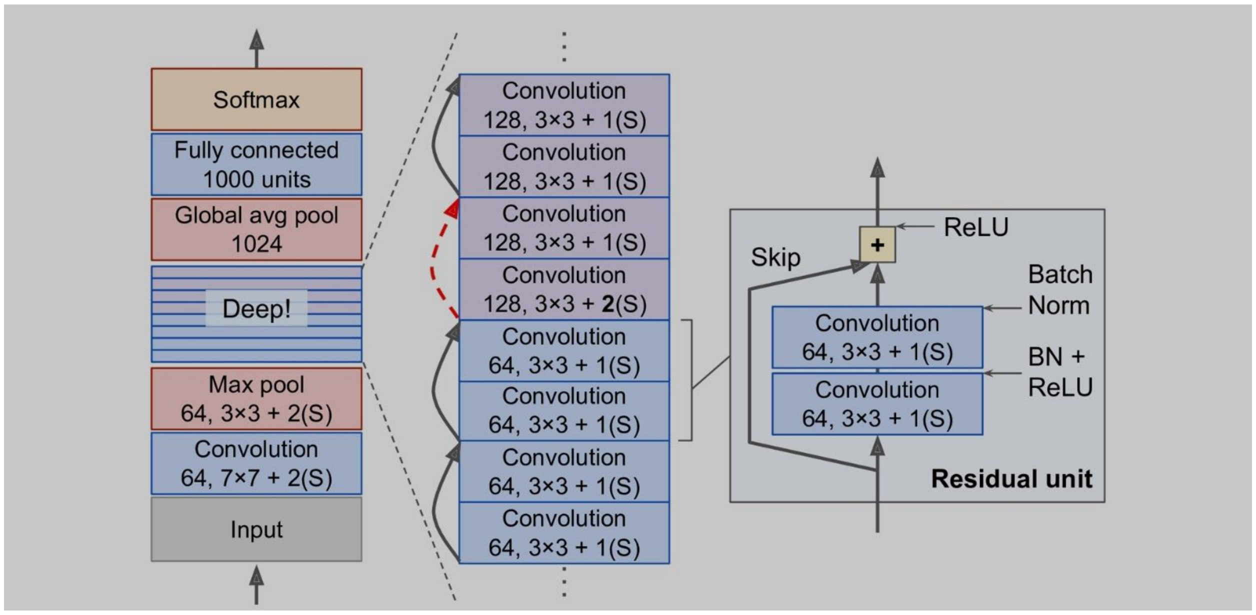
### Architecture

The ResNet50 or the Residual Network architecture has 50 layers, hence the name. The key to being able to train such a deep network is to use skip connections (shortcut connections) where the signal feeding into a layer is also added to the output of a layer, located a bit higher up the stack. When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network outputs a copy of its inputs, in other words, it just models the identity function.

If you add many skip connections, the network can start making progress even if several layers have not started learning yet, as shown in the figure below. The deep residual network can be seen as a stack of residual units (RUs), where each residual unit is a small neural network with a skip connection.



The ResNet's architecture is simple, with a deep stack of residual units. Each residual unit is composed of two convolutional layers (and no pooling layer), with Batch Normalization (BN) and ReLU activation. It uses 3x3 kernels and preserves spatial dimensions (stride 1, "same" padding).


Softmax
Fully connected 1000 units
Global avg pool 1024
Deep!
Max pool 64, 3×3 + 2(S)
Convolution 64, 7×7 + 2(S)
Input
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 2(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
ReLU
Skip
Batch Norm
BN + ReLU
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Residual unit

**Pretrained Model - Transfer Learning**

- We were using the ResNet-50 model pretrained on ImageNet and imported it from the Keras applications. This created the ResNet-50 model and downloaded the weights pretrained on the ImageNet dataset.
- ResNet-50 expects the images to be 224 x 224 pixels in size so we used the tf.image.resize() function to resize our images.
- We used the ImageDataGenerator to load the images and augment them in various ways.

**Training Parameters**

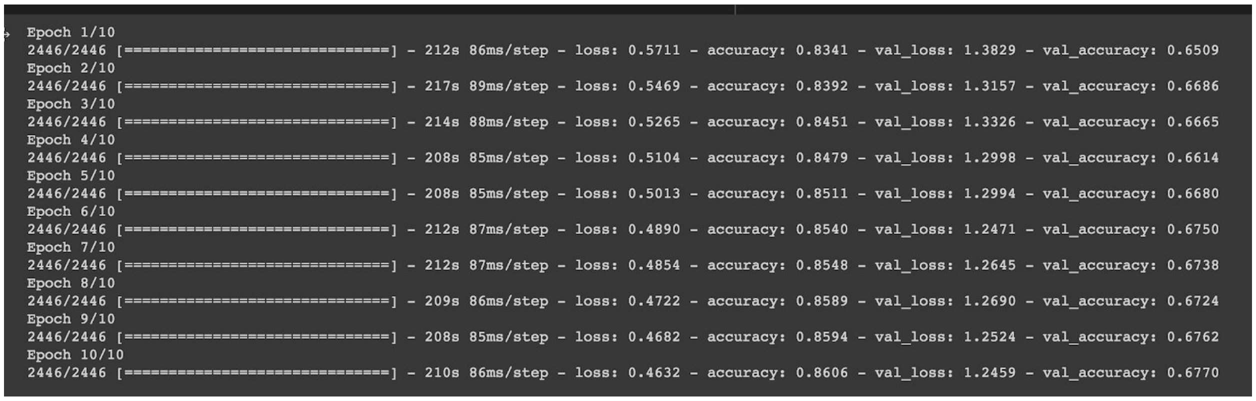
Two iterations each with 10 Epochs were run to create the final model. The first run was executed with all the pretrained layers frozen and the second one was run after unfreezing the pretrained layers.

- The following parameters were used for the first run of epochs
  - Epochs = 10
  - Batch Size = 32
  - Number of steps = # of samples / batch size
    - 78300 / 32 = 2446 for train set
    - 8700 / 32 = 271 for the validation set
  - Optimizer parameters

    -
      ```
      keras.optimizers.SGD(lr = 0.2, momentum=0.9, decay=0.01)
      ```
  - Compilation parameters

    -
      ```
      model.compile(loss = "categorical_crossentropy",
      optimizer=optimizer, metrics=["accuracy"])
      ```
  - All the input layers from pre-trained model was frozen by setting
    - Layer.trainable = False

After training the model for 10 epochs the validation accuracy was close to 68% and then stopped making any progress. This indicated that the top layers were well trained.



```
Epoch 1/10
2446/2446 [==============================] - 212s 86ms/step - loss: 0.5711 - accuracy: 0.8341 - val_loss: 1.3829 - val_accuracy: 0.6509
Epoch 2/10
2446/2446 [==============================] - 217s 89ms/step - loss: 0.5469 - accuracy: 0.8392 - val_loss: 1.3157 - val_accuracy: 0.6686
Epoch 3/10
2446/2446 [==============================] - 214s 88ms/step - loss: 0.5265 - accuracy: 0.8451 - val_loss: 1.3326 - val_accuracy: 0.6665
Epoch 4/10
2446/2446 [==============================] - 208s 85ms/step - loss: 0.5104 - accuracy: 0.8479 - val_loss: 1.2998 - val_accuracy: 0.6614
Epoch 5/10
2446/2446 [==============================] - 208s 85ms/step - loss: 0.5013 - accuracy: 0.8511 - val_loss: 1.2994 - val_accuracy: 0.6680
Epoch 6/10
2446/2446 [==============================] - 212s 87ms/step - loss: 0.4890 - accuracy: 0.8540 - val_loss: 1.2471 - val_accuracy: 0.6750
Epoch 7/10
2446/2446 [==============================] - 212s 87ms/step - loss: 0.4854 - accuracy: 0.8548 - val_loss: 1.2645 - val_accuracy: 0.6738
Epoch 8/10
2446/2446 [==============================] - 209s 86ms/step - loss: 0.4722 - accuracy: 0.8589 - val_loss: 1.2690 - val_accuracy: 0.6724
Epoch 9/10
2446/2446 [==============================] - 208s 85ms/step - loss: 0.4682 - accuracy: 0.8594 - val_loss: 1.2524 - val_accuracy: 0.6762
Epoch 10/10
2446/2446 [==============================] - 210s 86ms/step - loss: 0.4632 - accuracy: 0.8606 - val_loss: 1.2459 - val_accuracy: 0.6770
```

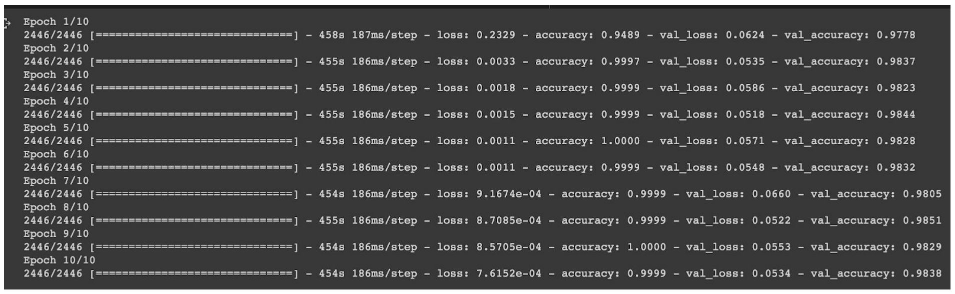
We then continued to train the model but this time by unfreezing all the layers. We used a much lower learning rate to avoid damaging the pretrained weights.

- The following parameters were used for the second run of epochs
    - Epochs = 10
    - Batch Size = 32
    - Number of steps = # of samples / batch size
        - 78300 / 32 = 2446 for train set
        - 8700 / 32 = 271 for the validation set
    - Optimizer parameters

        -
          ```
          keras.optimizers.SGD(lr = 0.01, momentum=0.9,
          decay=0.001)
          ```
    - Compilation parameters

        -
          ```
          model.compile(loss = "categorical_crossentropy", o
          optimizer=optimizer, metrics=["accuracy"])
          ```
    - All the input layers from pre-trained model was enabled by setting
        - Layer.trainable = True

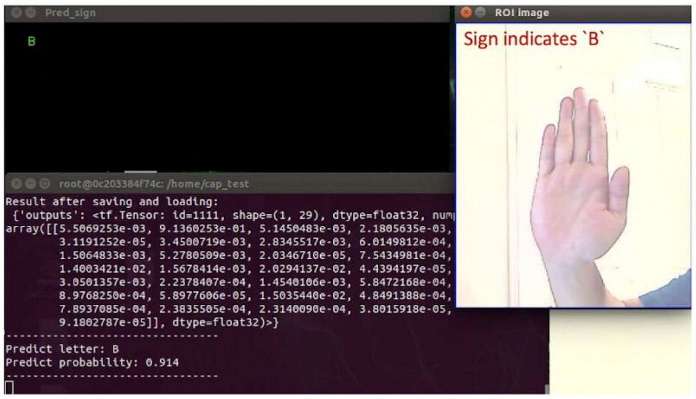
After training the model for 10 epochs the validation accuracy was close to 98% for the validation data set as shown below.



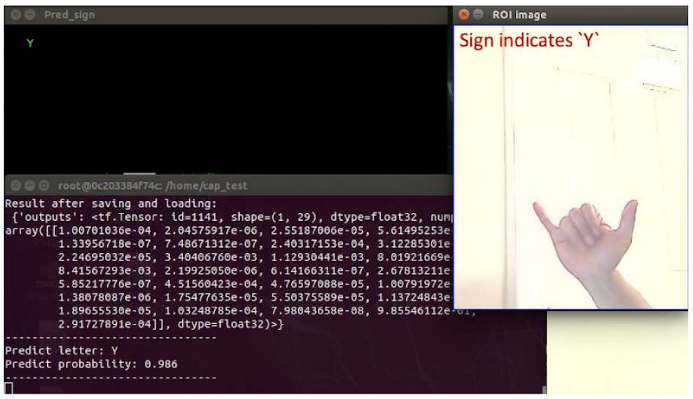
```
Epoch 1/10
2446/2446 [==============================] - 458s 187ms/step - loss: 0.2329 - accuracy: 0.9489 - val_loss: 0.0624 - val_accuracy: 0.9778
Epoch 2/10
2446/2446 [==============================] - 455s 186ms/step - loss: 0.0033 - accuracy: 0.9997 - val_loss: 0.0535 - val_accuracy: 0.9837
Epoch 3/10
2446/2446 [==============================] - 455s 186ms/step - loss: 0.0018 - accuracy: 0.9999 - val_loss: 0.0586 - val_accuracy: 0.9823
Epoch 4/10
2446/2446 [==============================] - 455s 186ms/step - loss: 0.0015 - accuracy: 0.9999 - val_loss: 0.0518 - val_accuracy: 0.9844
Epoch 5/10
2446/2446 [==============================] - 455s 186ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.0571 - val_accuracy: 0.9828
Epoch 6/10
2446/2446 [==============================] - 455s 186ms/step - loss: 0.0011 - accuracy: 0.9999 - val_loss: 0.0548 - val_accuracy: 0.9832
Epoch 7/10
2446/2446 [==============================] - 454s 186ms/step - loss: 9.1674e-04 - accuracy: 0.9999 - val_loss: 0.0660 - val_accuracy: 0.9805
Epoch 8/10
2446/2446 [==============================] - 455s 186ms/step - loss: 8.7085e-04 - accuracy: 0.9999 - val_loss: 0.0522 - val_accuracy: 0.9851
Epoch 9/10
2446/2446 [==============================] - 454s 186ms/step - loss: 8.5705e-04 - accuracy: 1.0000 - val_loss: 0.0553 - val_accuracy: 0.9829
Epoch 10/10
2446/2446 [==============================] - 454s 186ms/step - loss: 7.6152e-04 - accuracy: 0.9999 - val_loss: 0.0534 - val_accuracy: 0.9838
```

## Section VI: Functional Testing

From the models that we have been trained, most training models had good accuracy on both training and evaluating data (> 0.9). When performing sign recognition in edge devices, we found some signs had great prediction accuracy, such as ‘B’ and ‘Y’. In the figure, the probability of ‘B’ prediction is 0.914 and the probability of ‘Y’ prediction is 0.968.

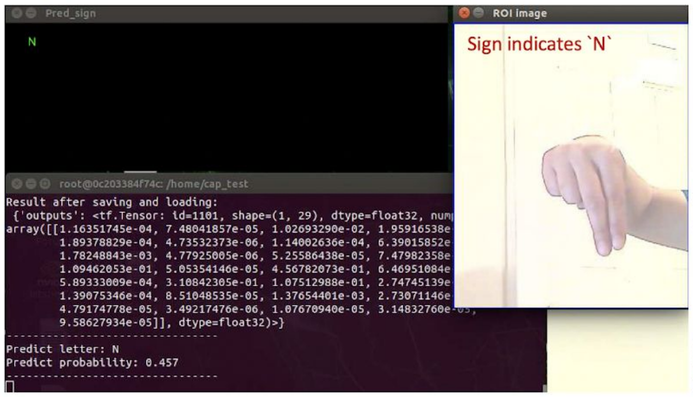


```
Result after saving and loading:
 {'outputs': <tf.Tensor: id=1111, shape=(1, 29), dtype=float32, num
array([[5.5069253e-03, 9.1360253e-01, 5.1450483e-03, 2.1805635e-03,
        3.1191252e-05, 3.4500719e-03, 2.8345517e-03, 6.0149812e-04,
        1.5064833e-03, 5.2780509e-03, 2.0346710e-05, 7.5434981e-04,
        1.4003421e-02, 1.5678414e-03, 2.0294137e-02, 4.4394197e-05,
        3.0501357e-03, 2.2378407e-04, 1.4540106e-03, 5.8472168e-04,
        8.9768250e-04, 5.8977606e-05, 1.5035440e-02, 4.8491388e-04,
        7.8937085e-04, 2.3835505e-04, 2.3140090e-04, 3.8015918e-05,
        9.1802787e-05]], dtype=float32)>}
-------------------------------
Predict letter: B
Predict probability: 0.914
-------------------------------
```

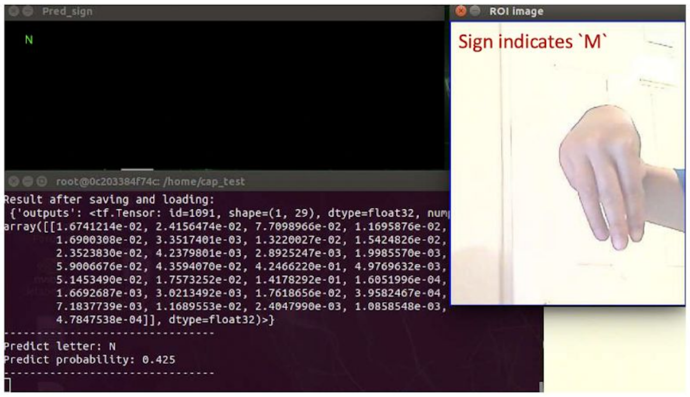


```
Result after saving and loading:
 {'outputs': <tf.Tensor: id=1141, shape=(1, 29), dtype=float32, num
array([[1.00701036e-04, 2.04575917e-06, 2.55187006e-05, 5.61495253e
        1.33956718e-07, 7.48671312e-07, 2.40317153e-04, 3.12285301e
        2.24695032e-05, 3.40406760e-03, 1.12930441e-03, 8.01921669e
        8.41567293e-03, 2.19925050e-06, 6.14166311e-07, 2.67813211e
        5.85217776e-07, 4.51560423e-04, 4.76597088e-05, 1.00791972e
        1.38078087e-06, 1.75477635e-05, 5.50375589e-05, 1.13724843e
        1.89655530e-05, 1.03248785e-04, 7.98043658e-08, 9.85546112e-01,
        2.91727891e-04]], dtype=float32)>}
-------------------------------
Predict letter: Y
Predict probability: 0.986
-------------------------------
```

However, when signs had similar hand gestures, the prediction became error-prone, such as ‘M’ and ‘N’. In the first example shown below, the prediction accuracy of ‘N’ was only 0.457. The second example showed that ‘M’ was misclassified as ‘N’.



```
Result after saving and loading:
 {'outputs': <tf.Tensor: id=1101, shape=(1, 29), dtype=float32, num
array([[1.16351745e-04, 7.48041857e-05, 1.02693290e-02, 1.95916538e
        1.89378829e-04, 4.73532373e-06, 1.14002636e-04, 6.39015852e
        1.78248843e-03, 4.77925005e-06, 5.25586438e-05, 7.47982358e
        1.09462053e-01, 5.05354146e-05, 4.56782073e-01, 6.46951084e
        5.89333009e-04, 3.10842305e-01, 1.07512988e-01, 2.74745139e
        1.39075346e-04, 8.51048535e-05, 1.37654401e-03, 2.73071146e
        4.79174778e-05, 3.49217476e-06, 1.07670940e-05, 3.14832760e-05,
        9.58627934e-05]], dtype=float32)>}
-------------------------------
Predict letter: N
Predict probability: 0.457
-------------------------------
```



```
Result after saving and loading:
 {'outputs': <tf.Tensor: id=1091, shape=(1, 29), dtype=float32, num
array([[1.6741214e-02, 2.4156474e-02, 7.7098966e-02, 1.1695876e-02,
        1.6900308e-02, 3.3517401e-03, 1.3220027e-02, 1.5424826e-02,
        2.3523830e-02, 4.2379801e-03, 2.8925247e-03, 1.9985570e-03,
        5.9006676e-02, 4.3594070e-02, 4.2466220e-01, 4.9769632e-03,
        5.1453490e-02, 1.7573252e-02, 1.4178292e-01, 1.6051996e-04,
        1.6692687e-03, 3.0213492e-03, 1.7618656e-02, 3.9582467e-04,
        7.1837739e-03, 1.1689553e-02, 2.4047990e-03, 1.0858548e-03,
        4.7847538e-04]], dtype=float32)>}
-------------------------------
Predict letter: N
Predict probability: 0.425
-------------------------------
```

In addition, the brightness of background and size of hand also resulted in the variabilities of prediction. Therefore, we will discuss how to further improve the sign recognition through edge devices in the next section.

## Section VII: Next Step

- The accuracy for real-time detection might be enhanced with the palm detection. This will require to process the training dataset with the hand landmark model, and will need additional inference to detect the palm.
- The current dataset includes frames from the same individuals under the same circumstance (light, position etc). The model can be more general if the dataset has many other performers with more real-world sign language images.
- The word-based analysis can have more efficient conversation with less time consumed positioning for each alphabet. It includes analysis of frame dependency as well as the overall body languages, which can be much more challenging for the current model to understand.
- As mentioned in the first section, there are 72 millions people around the globe that have conversational disabilities. It is important to make ASL-to-Audio easily accessible so these people can be benefited. Therefore, incorporating small devices or creating mobile apps will be necessary in the future.

## Section VIII: Project Github Link

https://github.com/sthiruvallur/w251-final-project/tree/master

## Section VIII: Reference

- https://arxiv.org/abs/1910.11006
- https://arxiv.org/abs/1409.1556
- https://www.geeksforgeeks.org/convert-text-speech-python/
- https://github.com/athena15/project_kojak/blob/master/real_time_gesture_detection.py
- https://ai.googleblog.com/2019/08/on-device-real-time-hand-tracking-with.html
- https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html
- https://blog.metaflow.fr/tensorflow-how-to-freeze-a-model-and-serve-it-with-a-python-api-d4f3596b3adc
- https://www.tensorflow.org/guide/saved_model
- https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a
- https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
- https://medium.com/@prasadpal107/saving-freezing-optimizing-for-inference-restoring-of-tensorflow-models-b4146deb21b5
- https://github.com/victordibia/handtracking
- https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html
- https://leimao.github.io/blog/Save-Load-Inference-From-TF-Frozen-Graph/
- Releases · fchollet/deep-learning-models
- https://github.com/pipidog/keras_to_tensorflow#load-a-pb-file-by-tensorflow