

# **Embedded Deep Learning: Final Project Report**

## **American Sign Language Fingerspelling Interpreter**

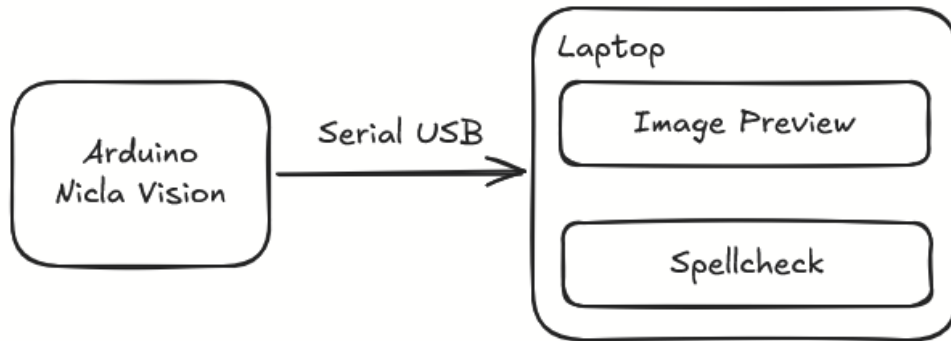
Rosina Ananth, Nathan Chow, and Edward Lu

### **Abstract**

American Sign Language (ASL) is the preferred language for many amongst the deaf and hard of hearing communities. Fingerspelling is a part of ASL which assigns a sign to each letter of the English alphabet and is used for proper names or when signs for words are not known. Since those born deaf may not be able to communicate vocally, complications may arise when trying to communicate with the non-deaf. Even for those who are familiar with ASL, it can still be difficult to translate fingerspelled words due to a high speed of signing. For our project, we aimed to create a fingerspelling interpreter that runs on an embedded device with a camera. We first trained a fingerspelling classifier to classify signs to letters, and then quantized and deployed this model on an Arduino Niela Vision board. We used TensorFlow Lite and Edge Impulse to train the fingerspelling classifier and deploy it on the MCU. We also have a frontend Python script that reads the images from the Niela Vision and displays them for a better user experience. Lastly, we used a library to spell check signed words and print them to the terminal. Future work could add interpretation of non fingerspelled signs for full ASL to speech capability and incorporate a better understanding of ASL grammar to provide a higher quality translation.

### **High Level Block Diagram**

This figure below illustrates a block diagram of our fingerspelling interpreter. The board captures images of signed words, which is transmitted over a serial USB connection to a laptop for visualization. On the laptop, a Python script receives the image data, and displays a live preview of the recognized signs. The script also simultaneously checks the spelling of the interpreted words. We use the TextBlob library to perform the spell checking.



## Dataset Source

We used the MNIST American Sign Language (ASL) fingerspelling image dataset for training. The dataset features thousands of labeled grayscale images, each representing a single letter of the ASL alphabet.

See the dataset here: <https://www.kaggle.com/datasets/datamunge/sign-language-mnist/data>

## Feature Extraction

We use Convolutional Neural Networks (CNNs) to do feature extraction. The CNN's we use in our model (see below for the model architecture) identify key spatial patterns in the images. We believe the first CNN (16 filters, 3x3 kernel size) implicitly extracts the edges and/or shapes of the hands in the input images. The second CNN (32 filters, 3x3 kernel size) extracts more complex features of the images that are related to the letters/labels. We use 16 then 32 filters because we thought that the first CNN would extract "easy"/simple features and then the more complex second CNN would extract "hard"/more nuanced features.

## Classifier Architecture

We tried to make the architecture simple and robust. To this end, we use a two phase pipeline for our classifier. The model first augments the data set, applying gaussian noise, brightness and contrast adjustments, horizontal flipping, rotations, translations, and zooming to expand the training set and reduce overfitting when the model is deployed. Then, we normalize the input image from 0-1. After that, we have two CNN's each followed by a max pooling layer. Then, we flatten the features learned by the CNN's and apply a dense layer. Finally, we have a final dense layer that outputs the predictions of the letters.

The figure below shows the model architecture as Python Tensorflow code:

```
aug_layers = tf.keras.models.Sequential([
    Input(IMG_SHAPE),
    tf.keras.layers.GaussianNoise(0.1),
    tf.keras.layers.RandomBrightness(0.1),
    tf.keras.layers.RandomContrast(0.1),
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomTranslation(0.1,0.1),
    tf.keras.layers.RandomZoom(0.05, 0.05)
])

model = tf.keras.models.Sequential([
    Input(shape=IMG_SHAPE),
    Rescaling(1./255),

    Conv2D(16,(3,3),activation= 'relu'),
    MaxPooling2D(),
    Conv2D(32,(3,3), activation='relu'),
    MaxPooling2D(),

    Flatten(),

    Dense(128,activation= 'relu'),
    Dense(27, activation='softmax')
])

model_with_aug = tf.keras.models.Sequential([
    aug_layers,
    model
])

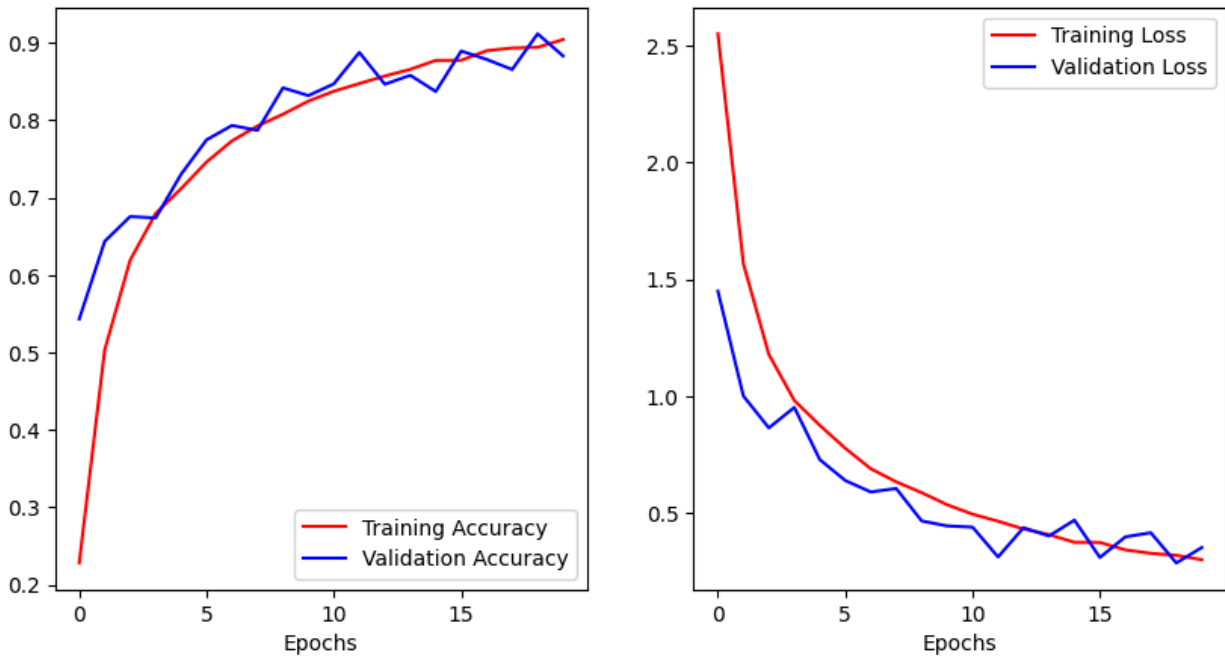
model_with_aug.compile('adam', loss=tf.keras.losses.categorical_crossentropy, metrics= ['accuracy'])
```

After training the model, we apply post training quantization using `TFLiteConverter` to `uint8` so that the model can be run on an embedded device. This process was similar to what we did in one of the homeworks.

## Confusion Matrix and Overall Accuracy Metrics

Below are the accuracy and losses for the validation and training datasets for the model described. These are the results for the unquantized model during training and testing. We got around 88% validation accuracy and 0.35 validation loss.




Accuracy and loss in training and validation data



Below are the resulting metrics on the test dataset for all classes in Edge Impulse. Edge Impulse picked a random set of the input dataset for testing. The results are for the quantized model which we uploaded to Edge Impulse. It seems like Edge Impulse did not generate a confusion matrix for our data possibly because there are a large number of classes. Overall the accuracy was around 82% for the random test dataset on Edge Impulse, which is lower than the result when testing on the training dataset after training in Python. This could possibly be due to the quantized model performing worse or perhaps Edge Impulse picked different images for their testing dataset and these random images happen to give a worse accuracy.

 ACCURACY  
**82.03%**

### Metrics for Pretrained learn

METRIC	VALUE
Weighted average Precision 	0.88
Weighted average Recall 	0.87
Weighted average F1 score 	0.86

### Confusion matrix

	F1-SCORE	PRECISION	RECALL
A	0.86	0.83	0.89
B	0.93	1.00	0.87
C	1.00	1.00	1.00
D	0.46	0.76	0.33
E	0.83	0.96	0.73
F	0.92	0.89	0.96
G	0.81	0.98	0.69
H	0.81	0.90	0.75
I	0.86	0.75	0.99
J	0.00	0.00	0.00
K	0.99	1.00	0.98
L	0.91	0.99	0.84
M	0.80	0.79	0.81
N	0.93	0.90	0.97
O	0.85	1.00	0.73
P	0.88	0.92	0.84
Q	0.98	0.96	1.00
R	0.83	0.71	1.00
S	0.58	0.88	0.43
T	0.79	1.00	0.66
U	0.80	0.93	0.70
V	0.92	0.91	0.94
W	0.96	0.92	1.00
X	0.87	0.83	0.91
Y	0.87	0.95	0.80
Z	0.00	0.00	0.00
	0.00	0.00	0.00

Note that classes J, Z, and None, have a score of 0 across the board because there are no samples in the dataset for those classes. J and Z are absent from the dataset because they require dynamic motion of the hand.

## Deployment Methods

As stated above, we used Tensorflow to train and test and TensorflowLite to create a model to run on an embedded device. We deployed our model on the Arduino Nicla Vision microcontroller. We also have a frontend that visualized the images captured by the on-board camera of the Nicla Vision which were streamed to a laptop through USB serial. The model inference ran at about 120-150 ms per frame. The frontend visualization of the hands and results ran at about 5 FPS. We display the top three most confident letters on the visualization (see Images and Videos Section). The visualization was written using Python and OpenCV.

## **Significant Challenges and Lessons Learned**

A big challenge we encountered after our first deployment was that the model expects the background of the image to be fully white. This is because the training data all contain white or fairly white backgrounds. So, we could not just place our hand in front of the camera because the contours of objects in the background could blend with the hand when the image is converted to grayscale and the model would not be able to know where the hand is. This means we had to sign the letters in front of a whiteboard so that the background would always be white. After doing this, the model performed much better!

Another one of the challenges we encountered was aiming the camera accurately, as embedded systems typically lack a built-in screen for real-time feedback. As stated above (and shown below), we had to use a laptop to display the images captured by the Nicla Vision using a display on a laptop.

We also ran into an issue where the model deployed on the Nicla Vision was not performing well. After some debugging, we realized it was because the camera images were in full color, but our model wanted grayscale images. So, we manually convert the input image to the model to grayscale by averaging the color channels and cropping the center to be 28x28 (the same size as the images in the MNIST dataset) and the result seemed to work much better!

In deployment, the model seemed to be confused about similar looking signs/letters. For instance, when signing “A,” “M,” and, “N” the captured grayscale images all looked very similar and the model struggled to distinguish between the letters. Even to a human it was difficult to know if the images were signing A, M, or N! We think this might have been because of the low resolution of the input images. If we were to do this project again, we would use a different dataset, perhaps one in higher resolution and in color.

These challenges made us realize that embedded systems solutions rarely work out of the box and often require significant tweaking to accommodate different hardware configurations. Adapting the Arduino code to send data to external displays and perform image transformations

showed us that embedded machine learning projects are actually very iterative and solutions need to be constantly refined.

## Links to Images and Videos

Below is a link to images and videos we took during testing and during the demo:  
<https://photos.app.goo.gl/mbdpxByLnfrTnmU8>

