Sign Language Interpreter

Project Report - Group 6
Introduction to AI - Spring 2023

Ayesha Nayyer - 25141 & Hiba Mallick - 24015

Project Objective

While awareness about those hard-of-hearing/speaking has increased in recent times and steps are taken to ensure their participation in hearing communities,such as through the increased use of closed captioning, the gap between those who communicate through sign language and those who do not understand it is still far too great. Our goal with this project was to bridge this distance between deaf individuals who use ASL and the hearing population, most of who are not fluent or familiar with ASL.

We wanted to develop a real-time interpreter for American Sign Language (ASL) that can recognize hand gestures performed by users and convert them into corresponding ASL alphabets. Using ASL alphabets to communicate is known as fingerspelling. This project report outlines the development process, methodology, and evaluation of the AI ASL interpreter.

Data Used

In our approaches, we've used either of two similar datasets that differ in size. Both had 29 classes (26 for the alphabet and "del", "space", and "nothing")

- Dataset 1 was unsplit and comprised of 870 images.
- Dataset 2 was split and comprised of 87000 train images and 29 test images

Methodology

We tried three different methods and various iterations of models for this project.

1. **Traditional Machine Learning**

   Our aim starting out was to use traditional machine learning algorithms trained on a mid-sized dataset (Dataset 1) to create the interpreter.

   We preprocessed and resized and read test and train images into numpy arrays, normalised them by dividing by 255. We then used a feature extraction function to extract features from our images. The features include pixel values in RBG, 12 Gabor filters with varying values and a Sobel feature for edge detection.

   We used a Random Forest (RF) Classifier and varied the number of trees and on 300 trees, our accuracy and RAM usage was optimized at around 48%

   We tried using a combination of Support Vector Machine (SVM) and RF but our accuracy decreased to 30%.

2. **Convolutional Neural Network with Image Classification**

   We decided to use a deep learning approach with CNN for improved results. For this we used D2, a huge dataset with 300 images in different conditions for each symbol.

   We used a 100:10:1 ratio for train, validation and test and used a batch size of 64

   In preprocessing, we tried augmenting and applying different filters to the images and experimenting with what gave the highest accuracy, such as Gaussian Blur, Canny and Sobel and settled on using a Sobel filter in CV2.

   We then built a sequential model with 12 total layers including multiple convolutional layers, dropout regularization, and fully connected layers for classification.

   Then we compiled the model, specifying the optimizer as 'adam', the loss function as 'categorical_crossentropy' (suitable for multi-class classification), and tracking the accuracy metric during training.

```
 model = Sequential()

    model.add(Conv2D(64, kernel_size=4, strides=1,
activation='relu', input_shape=TARGET_DIMS))
    model.add(Conv2D(64, kernel_size=4, strides=2,
activation='relu'))
    model.add(Dropout(0.5))
    model.add(Conv2D(128, kernel_size=4, strides=1,
activation='relu'))
    model.add(Conv2D(128, kernel_size=4, strides=2,
activation='relu'))
    model.add(Dropout(0.5))
    model.add(Conv2D(256, kernel_size=4, strides=1,
activation='relu'))
    model.add(Conv2D(256, kernel_size=4, strides=2,
activation='relu'))
    model.add(Flatten())
    model.add(Dropout(0.5))
    model.add(Dense(512, activation='relu'))
    model.add(Dense(N_CLASSES, activation='softmax'))

    model.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
```

Our model is fitted on 5 epochs and we obtained a 86%accuracy on test images. Despite the good accuracy of the model, it gave poor results on webcam streaming due to changes in lighting, noise and other objects being present in the frame. The results didn't improve despite adding a ROI extractor to the webcam.

3. Convolutional Neural Network with Hand Detection

To further improve our results, we used MediaPipe. The MediaPipe Hand Landmarker task lets you detect the landmarks of the hands in an image. You can use this Task to localize key points of the hands and render visual effects over the hands.

For this, we used MediaPipe's pre-built hand-detection model and trained it using our own dataset. The learning rate is 0.001, it has 25 epochs and the batch side is 128. The dropout rate is set to 0.2, which is the proportion of input units to drop during training to prevent overfitting. We saved the model as a task file and used it with MediaPipe's prebuilt functionality alongside our webcam functionality to predict results on webcam. We used both datasets and on Dataset 1 we obtained 82% on train and 78% on test images. On dataset 2, we obtained 96.4% accuracy

Results

To check our models on real world data, we added webcam functionality. For each of our models, we created a video stream through webcam, captured frames at equal intervals, preprocessed the image and passed that into our model to predict its label.

Our trained model is capable of predicting results from frames captured from a webcam stream. The model prints out a statement showing all the hand landmarks it has identified in the frame and the resulting prediction of the category.

GUI

The GUI consists of a webcam with a bounding box for the user to sign symbols in. The model processes only the image from inside the bounding box and the resulting prediction is printed out with the model's prediction accuracy on top of the box.

The model also saves the previously signed letters and prints them out as a formatted sentence.

Limitations

One of the earliest issues we ran into was the GPU and RAM requirement to run larger models with large datasets. Large datasets allow for a higher accuracy in models but we were unable to do this without a GPU setup.

Another limitation to our models is the confusion between similar looking symbols in ASL.'W' 'V' look extremely similar, 'S' 'T' and 'M' 'N' can also be confused for each other.

Some symbols in ASL have motions attached to them, for example the letters J and Z. Since we are only processing images, we do not have the capability to identify letters with hand-motions which is a big setback.

A overarching limitation is the problem of only identifying fingerspelling. Everyday ASL has gestures for words and concepts, and fingerspelling is not quick enough for fluent, everyday use.

Future Directions

To further improve the accuracy and robustness of this model, we could add another algorithm that prevents similar symbols being confused for each other. It would check if the model recognises one of a group of symbols that look alike, and then run another, stricter model on them to differentiate between those specified groups.

We could also add functionality for letters with motion and then expand our dataset to train our model for ASL words, rather than just fingerspelling.

One of our aims when starting this project was to create a bilingual Sign Language Interpreter by combining datasets for Pakistani Sign Language (PSL) and ASL. This would be possible by grouping common symbols, augmenting labels based on what language the user has chosen to use the app in. However, no dataset is available for PSL yet, and we were not able to create one in time.

We can also pass our predicted sentence through a Python library's autocorrect function (Hunspell or Textblob) so the sentence being produced so errors can be ignored and corrected.

How to Run

There are 2 steps to running the project.

1. Training the Model

1. In the training_task_file.ipynb file, import dependencies, and the datasets from kaggle.
2. Unzip the datasets
3. Go into the files folder and change the name for the 'nothing' classes in both test and train folders to 'none' (The code won't work otherwise)
4. Create and fit the gesture_recognizer model on the data by running the cells
5. Download the gesture_recognizer.task file and save it to your computer.

2. Using the Model

1. In the sign_language.ipynb file , import dependencies
2. Upload the gesture_recognizer task into the runtime (since the model takes time to train, we attached our trained model file with our submission).
3. Run the Colab code snippets for webcam access and all other cells.
4. In the last cell , the webcam will open.
5. In the webcam, start signing letters from ASL and the model will write our the output from the gesture_recognizer.
6. The predicted output for the letter will be in the 'category_name' part of our printed statement.

ASL Alphabets Guide

Here is a guide for ASL alphabet symbols