KING MONGKUT'S INSTITUTE OF TECHNOLOGY LATKRABANG
FACULTY OF ENGINEERING
DEPARTMENT OF ROBOTICS AND AI ENGINEERING

01416504 – MACHINE LEARNING IN PRACTICE

American Sign Language Recognition

INSTRUCTED BY: Dr. AKADEJ UDOMCHAIPORN

NAME:
MOHAMED ARFAN MOHAMED NAYAS,
PATTARATORN SORANATHAVORNKUL,
KRITJUHN KAKLAI
STUDENT ID: 62011151, 62011191, 62011268
DATE OF SUB.: 31/05/2022

Introduction

American Sign Language is a natural language that serves as the primary sign language of the deaf communities in America. It has the same linguistic properties as spoken languages, with grammar differing from English. It is expressed by the users' movements of the hands and face. To ease the communication between the deaf people and people who do not understand ASL, we have implemented the use of machine learning.

We use a pre-existing dataset called Sign Language MNIST, which is presented in CSV formats with labels and pixel values in single rows. The ASL letter database of hand features has 24 classes of letters, excluding letters J and Z as they require motion. The first column contains labels from 0-24, representing A-Y, respectively. The second column onwards contains the pixel value: pixel1 up til pixel784, representing a single 28x28 pixel image with grayscale values between 0-255. The training data has 27,455 cases, and the test data has 7172 cases.

Fig. 1. Letters and their ASL representation

Methodology

In our model, we are going to feed it an existing dataset consisting of letters that are already classified to what those letters are:

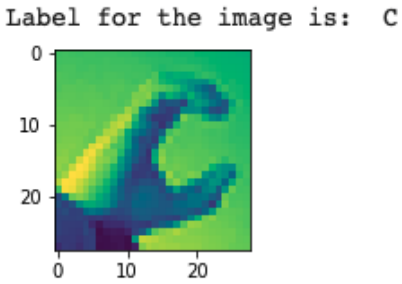


Fig. 2. An image from Sign Language Mnist dataset

On top of fig. 2, we can notice the label for the image being letter C as displayed, therefore, this image is classified as letter C. To further understand the dataset, in Fig. 3, we displayed the distribution of the different labeled letters in the dataset.

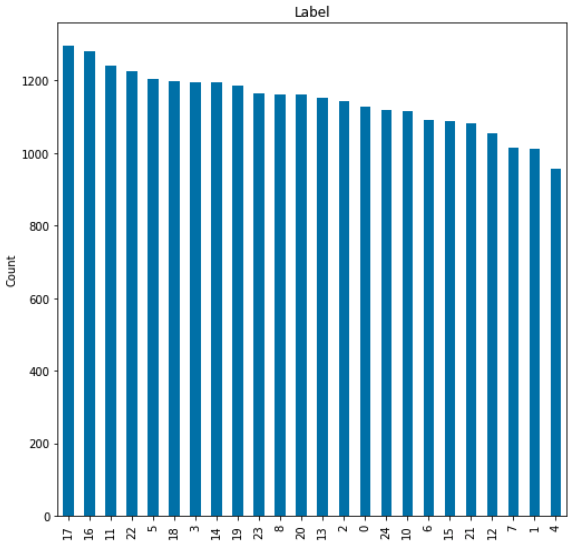


Fig. 3. Bar Chart of Data Distribution

In fig. 3, we can see the number of times each letter appears in the dataset. The y-axis represents the count and the x-axis represents the position number of the letter, ie. 0 represents A and 1 represents B. Since letters J and Z require motion, the numbers representing it (9 and 25) do not appear in the dataset. The data distribution also shows that the data that we use is fairly balanced.

Pre-processing

The Sign Language dataset has around 35,000 data and each of them have (28 x 28 = 784) feature inputs with values of each pixel ranging from 0 to 255. To properly train our model, we need to pre-process the data. We first normalized the image pixel values from [0, 255] to [0, 1]. After normalizing, we then convert class vectors into one-hot encoded format. Then we reshaped the images to (28, 28, 1) for our neural network.

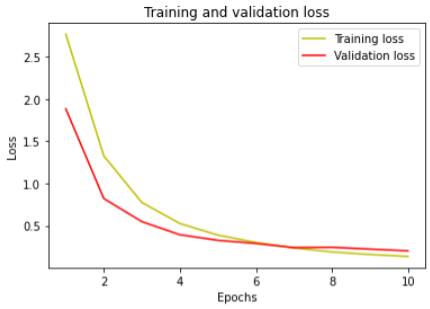
What experiments did you do?

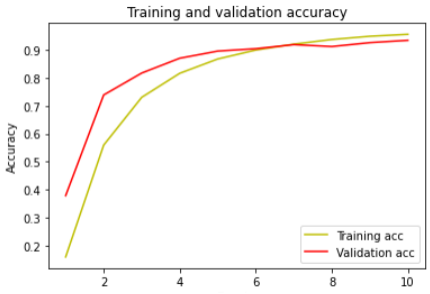
Tuning Process: Change hyperparameters

This is the final model after tuning the parameters.

```
#model.compile(loss ='sparse_categorical_crossentropy', optimizer='adam', metrics =['acc'])
model.compile(loss ='categorical_crossentropy', optimizer = 'Adam', metrics =['acc'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320

max_pooling2d (MaxPooling2D  (None, 13, 13, 32)        0
)

dropout (Dropout)            (None, 13, 13, 32)        0

conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496

max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)          0
2D)

dropout_1 (Dropout)          (None, 5, 5, 64)          0

conv2d_2 (Conv2D)            (None, 3, 3, 128)         73856

max_pooling2d_2 (MaxPooling  (None, 1, 1, 128)         0
2D)

dropout_2 (Dropout)          (None, 1, 1, 128)         0

flatten (Flatten)            (None, 128)               0

dense (Dense)                (None, 128)               16512

dense_1 (Dense)              (None, 25)                3225
```

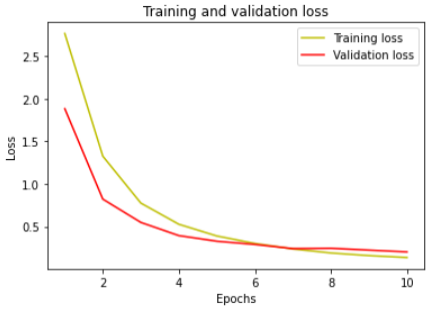


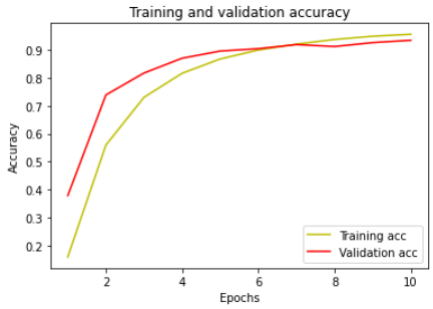


Part of our Tuning process

1) Change the number of iterations(epochs) in training neural network:
   Begin with decreasing the number of the epochs from 10 to 5.

```
#history = model.fit(X_train, y_train, batch_size = 128, epochs = 10, verbose = 1, validation_data = (X_test, y_test))
history = model.fit(X_train, y_train_cat, batch_size = 128, epochs = 5, verbose = 1, validation_data = (X_test, y_test_cat))
```



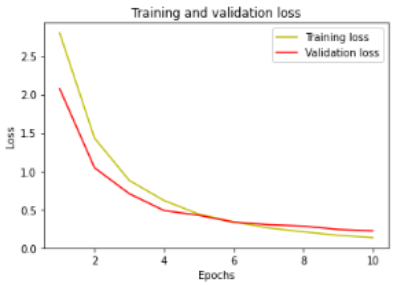


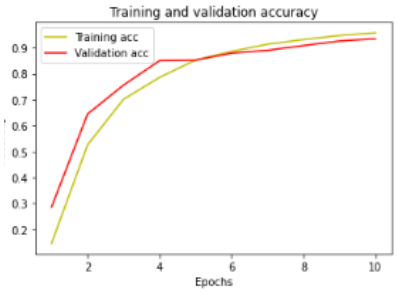

```
accuracy = accuracy_score(y_test, prediction)
print('Accuracy Score = ', accuracy)

# i = random.randint(1,len(predictions))
# plt.imshow(X_test[i,:,:,0])
# print("Predicted Label: ", class_names[int(predictions[i])])
# print("True Label: ", class_names[int(y_test[i])])
```

```
Accuracy Score =  0.9336307863915226
```

2) Change the number of batch sizes by increasing from 128 to 220.

```
[24] #history = model.fit(X_train, y_train, batch_size = 128, epochs = 10, verbose = 1, validation_data = (X_test, y_test))
     history = model.fit(X_train, y_train_cat, batch_size = 220, epochs = 10, verbose = 1, validation_data = (X_test, y_test_cat))
```

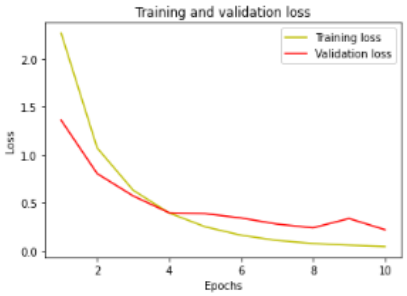


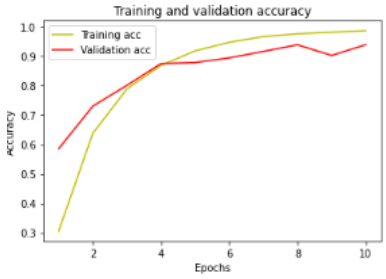



```
[28] accuracy = accuracy_score(y_test, prediction)
     print('Accuracy Score = ', accuracy)

     # i = random.randint(1,len(predictions))
     # plt.imshow(X_test[i,:,:,0])
     # print("Predicted Label: ", class_names[int(predictions[i])])
     # print("True Label: ", class_names[int(y_test[i])])

     Accuracy Score =  0.9602621305075293
```

3) Change the learning rate in optimization algorithms. From the previous model we use the optimizer = 'Adam' to optimizer = 'RMSprop' for maintaining a moving average of the square gradients and dividing the gradient by the root of average. So they use it to estimate the variance.

```
#If your targets are one-hot encoded, use categorical_crossentropy. Examples of one-hot encodings:
# If your targets are integers, use sparse_categorical_crossentropy.

#model.compile(loss ='sparse_categorical_crossentropy', optimizer='adam', metrics =['acc'])
model.compile(loss ='categorical_crossentropy', optimizer = 'RMSprop', metrics =['acc'])
model.summary()
```



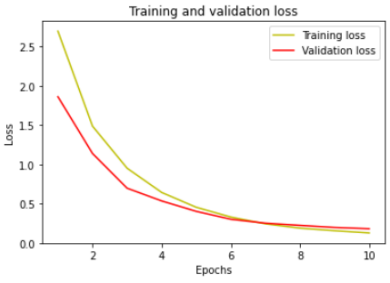


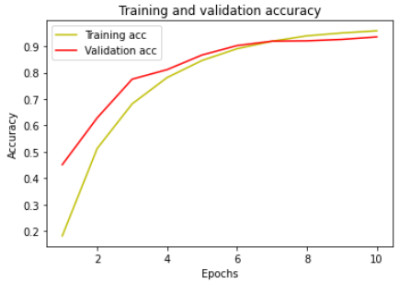

```
[45] accuracy = accuracy_score(y_test, prediction)
     print('Accuracy Score = ', accuracy)

     # i = random.randint(1,len(predictions))
     # plt.imshow(X_test[i,:,:,0])
     # print("Predicted Label: ", class_names[int(predictions[i])])
     # print("True Label: ", class_names[int(y_test[i])])

     Accuracy Score =  0.9637479085331846
```

4) Change the dropout of the model :

model.add(Dropout(0.2)) to model.add(Dropout(0.3))

```
[11] ########################################################

     #Model

     model = Sequential()

     model.add(Conv2D(32, (3, 3), input_shape = (28,28,1), activation='ReLU'))
     model.add(MaxPooling2D(pool_size = (2, 2)))
     model.add(Dropout(0.3))

     model.add(Conv2D(64, (3, 3), activation='ReLU'))
     model.add(MaxPooling2D(pool_size = (2, 2)))
     model.add(Dropout(0.3))

     model.add(Conv2D(128, (3, 3), activation='ReLU'))
     model.add(MaxPooling2D(pool_size = (2, 2)))
     model.add(Dropout(0.3))

     model.add(Flatten())

     model.add(Dense(128, activation = 'ReLU'))
     model.add(Dense(25, activation = 'softmax'))
```






```
accuracy = accuracy_score(y_test, prediction)
print('Accuracy Score = ', accuracy)

# i = random.randint(1,len(predictions))
# plt.imshow(X_test[i,:,:,0])
# print("Predicted Label: ", class_names[int(predictions[i])])
# print("True Label: ", class_names[int(y_test[i])])
```

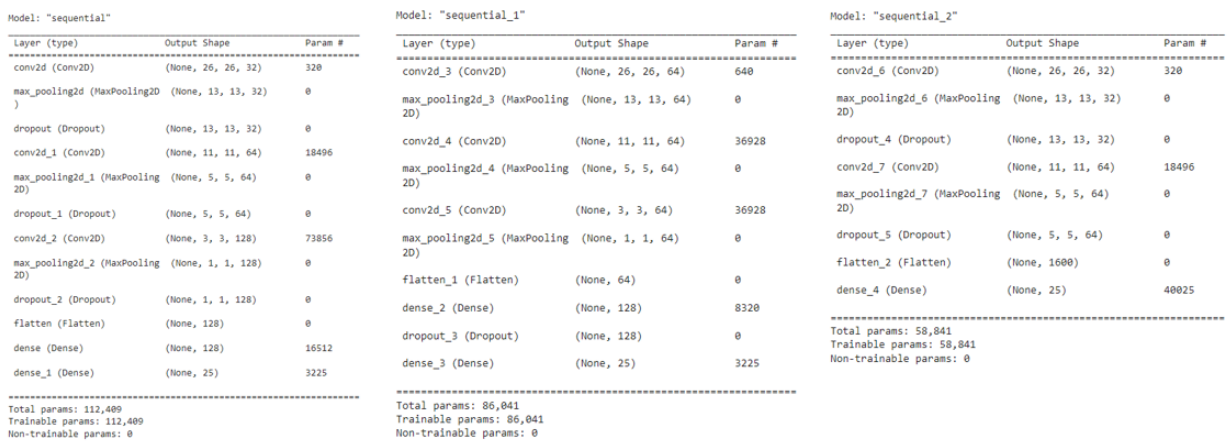
```
Accuracy Score =  0.9350250976017848
```

Ensemble model

We obtain 93.05 % accuracy from training one CNN model. We use the Ensemble technique, which combines several models to improve the accuracy of algorithms. The pre-processing steps are the same as the original model: normalize image pixel values, convert class vectors into the one-hot encoded format, and resize the image. We use Categorical Cross Entropy as the loss function with the Adam optimizer.

We use three different CNN models for assembling. Each model is trained separately and saved as model 1, model 2, and model 3. By comparing the training and validation accuracy together with the graph of each model, we find out that model 1 is the best. There are some overfitting in model 2 and 3.

We get 93.94 % accuracy from model 1, 91.83% accuracy from model 2, and 91.77% accuracy from model 3. Regarding the accuracy of each model, the Grid search will find the best combination of the weight of each model. The weight is divided to be 0.4 for model 1, 0.1 for model 2 and 0.2 for model 3. Then we use Tensordot to sum the product of all elements over specified axes. The final accuracy from the Ensemble model is 96.28%.



Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
dropout (Dropout)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_1 (Dropout)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 128)	0
dropout_2 (Dropout)	(None, 1, 1, 128)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 128)	16512
dense_1 (Dense)	(None, 25)	3225

Total params: 112,409
Trainable params: 112,409
Non-trainable params: 0

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 26, 26, 64)	640
max_pooling2d_3 (MaxPooling2D)	(None, 13, 13, 64)	0
conv2d_4 (Conv2D)	(None, 11, 11, 64)	36928
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_5 (Conv2D)	(None, 3, 3, 64)	36928
max_pooling2d_5 (MaxPooling2D)	(None, 1, 1, 64)	0
flatten_1 (Flatten)	(None, 64)	0
dense_2 (Dense)	(None, 128)	8320
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 25)	3225

Total params: 86,041
Trainable params: 86,041
Non-trainable params: 0

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_6 (MaxPooling2D)	(None, 13, 13, 32)	0
dropout_4 (Dropout)	(None, 13, 13, 32)	0
conv2d_7 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_7 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_5 (Dropout)	(None, 5, 5, 64)	0
flatten_2 (Flatten)	(None, 1600)	0
dense_4 (Dense)	(None, 25)	40025

Total params: 58,841
Trainable params: 58,841
Non-trainable params: 0

Fig. 4. Models using in Ensemble (Model1,2,3 respectively)

What insight or knowledge do you get from the experiments?

There are numerous knowledges we gained from the experiments such as:

- CNN model
- Ensemble technique
- How to deal with overfitting
- Tuning hyperparameters of the model by using different activation functions, loss function, epoch and batch size

During the experiment, we faced the problem of overfitting that can be observed from the accuracy-loss graph and the difference between training and validation accuracy. The problem can be solved by removing layers, applying dropout, and early stopping. At first, we used one CNN model and got 93.94 % accuracy. We try to improve the accuracy by applying ensemble technique to combine different models.

How can you measure the quality of the experiments?

There are numerous ways we can measure the quality of our experiments, such as:

- Measuring the accuracy
- Summarizing the history of accuracy and loss in the graph
- Using confusion matrix
- Fractions of incorrect predictions

The quality of the experiments can be measured by the accuracy, the graph of accuracy and loss, and the confusion matrix. In fig. 5, we can see the results from our accuracy and loss graphs.

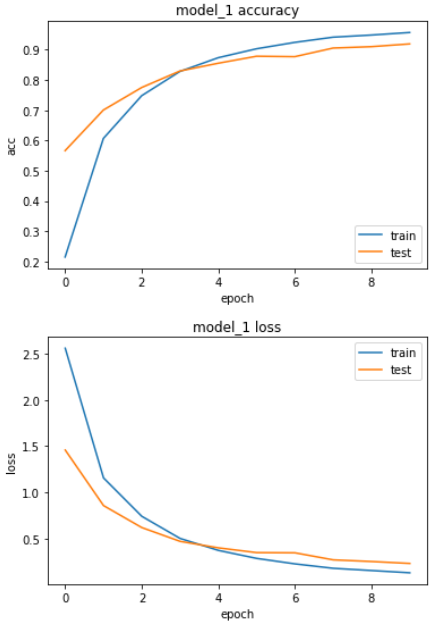


Fig. 5. Accuracy and Loss Graph

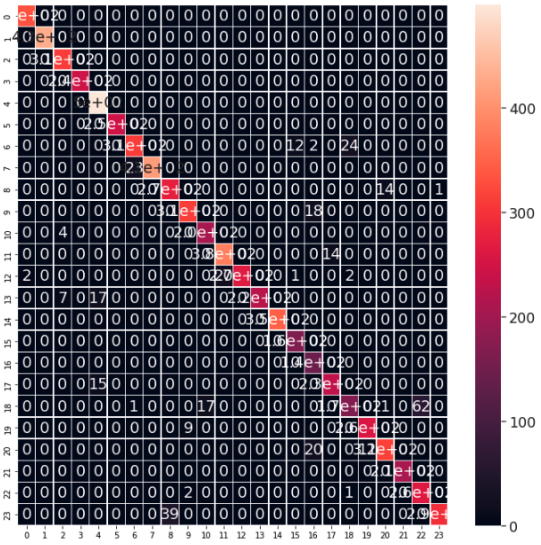


Fig. 6. Confusion Matrix

In fig. 6, we can understand more about the quality of the experiment through the confusion matrix. This helps us understand the validity of our model.

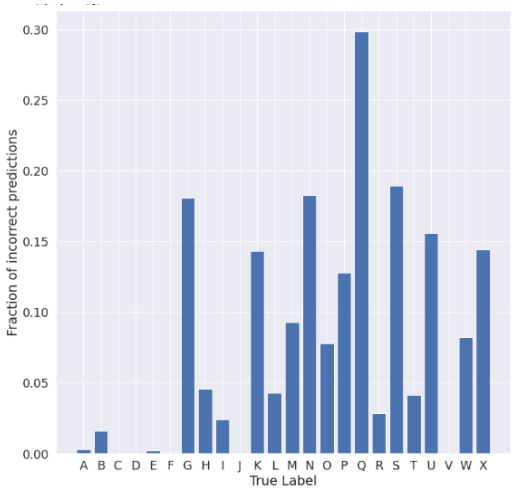


Fig. 7. Fractions of incorrect predictions before applying ensemble ensemble technique

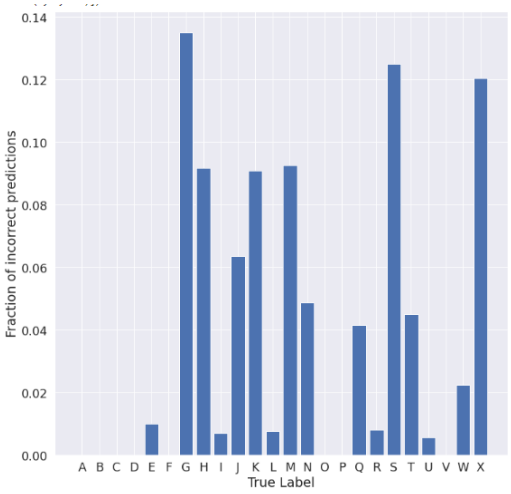


Fig. 8. Fractions of incorrect predictions after applying ensemble ensemble technique

In fig. 8, we can visualize the fractions of incorrect predictions made by our model in the form of a bar chart. As we can see, letters G, S, and X are the top three fraction of incorrect predictions and many letters do not have any. This might be due to the fact that letter S in ASL has a very close representation to other letters such as N, M, A.

Testing on our own images

To further make sure our model is valid enough to be deployed, we need to use it to predict with our own images. In fig. 8a, we can see our own image we used to test the model and in fig. 8b we can see the class it labeled the image as, which it accurately predicted as letter F. The images need to be resized to 28 x 28 as we need 784 feature inputs.

Fig. 9a

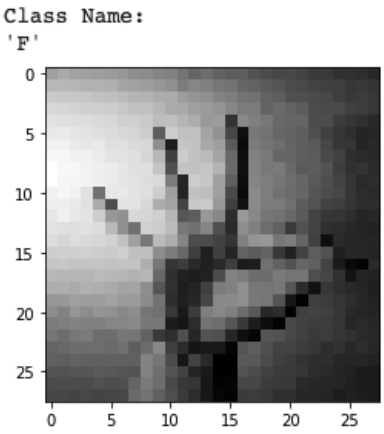


Fig. 9b

We also tested this model with multiple images which can be captured from our webcam through google colab. So far our model has been accurately predicting most of the images with some wrongly classified because the training data and our own data is pretty different. This problem can be tackled by creating our own dataset, with images we capture ourselves and label them.

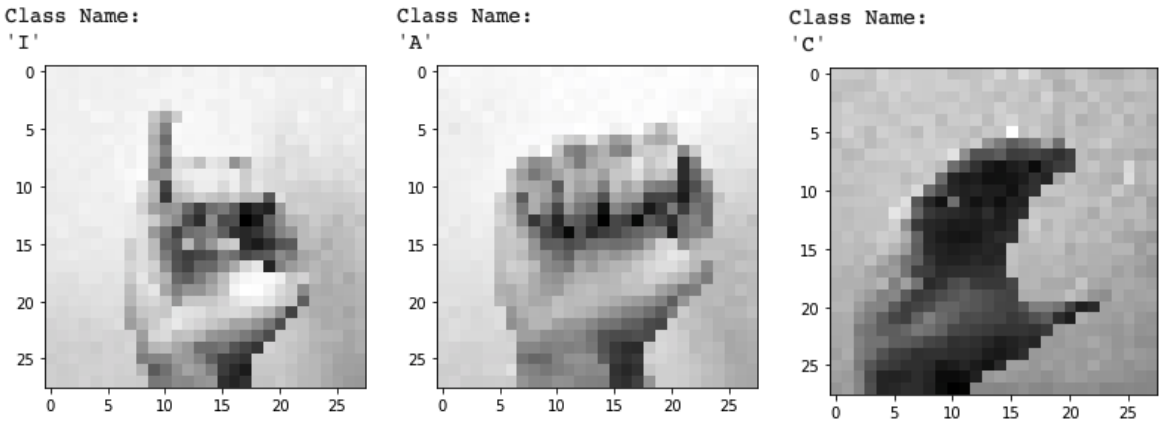


Fig. 10. Other examples of our own data

Conclusion

In this report, we discussed the working of our ASL recognition model and how it accurately predicts the input images we feed. The images need to be preprocessed to have 784 feature inputs with pixel values ranging from 0-255. We also tackled the problem of overfitting and to optimize the validity of our model, we used ensemble technique. The final accuracy of our model exceeds that of models 1,2 and 3. After applying ensembles, the overall accuracy is increased and the fraction of incorrect predictions is decreased. To understand the quality of our images, we used confusion matrix, accuracy and loss graphs, and fractions of incorrect predictions. Lastly, we put the model to real life test using our own images which it predicts very well.

References

Bhattiprolu, Dr. Sreenivas. "Sign Language MNIST." 14 April 2021. <https://github.com/bnsreenu/python_for_microscopists/blob/master/213-ensemble_sign_language.py>.

"Sign Language Recognition Using Python and OpenCV." n.d. *Data Flarir.* 15 May 2022. <https://data-flair.training/blogs/sign-language-recognition-python-ml-opencv/>.

"Sign Language MNIST." n.d. *Kaggle.* 15 May 2022. <https://www.kaggle.com/datasets/datamunge/sign-language-mnist>.

B., Sreenivas. "212 - Classification of mnist sign language alphabets using deep learning." 7 April 2021. *Youtube.* <https://youtu.be/3hjsdfTVWRQ>.