American Sign Language Recognition using Convolutional Neural Network

Data Set: ASL Dataset

Venkat Giri Anantapatnaikuni
anantapa@usc.edu

Rohit Veeradhi
veeradhi@usc.edu

May 7, 2023

1 Introduction

American Sign Language (ASL) was created to provide a means of communication for individuals who are deaf or hard of hearing and who may not be able to communicate effectively through spoken language. ASL has its own unique grammar, syntax, and vocabulary, and it is a complete language with its own linguistic structure. It allows individuals to express complex ideas and emotions, and it is an important tool for the deaf community to communicate and connect with each other and with the hearing world. In addition to its practical uses, ASL is also important in recognizing and celebrating Deaf culture, which has its own distinct history, traditions, and values. ASL is an integral part of Deaf culture and identity and is recognized as a separate language by many countries and organizations. Overall, the motivation behind the development and use of ASL is to provide a way for the deaf community to communicate and express themselves, recognize and celebrate their unique culture and identity, and promote accessibility and inclusivity for all individuals, regardless of their hearing ability.

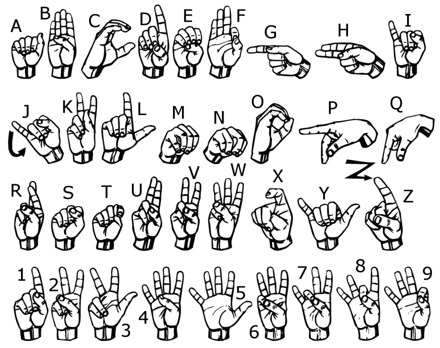


Figure 1: Taking an in-depth look at the American Sign Language (Alphabets and Numbers) [12]

1.1 Objective

This project aims to create a Deep Learning model to translate sign language into a text-based format using the American Sign Language (ASL) alphabet used in fingerspelling. The model will input a single image of a hand sign and accurately classify it according to the ASL alphabet. The project aims to explore different Deep Learning model approaches to determine the most effective one for this task. The selected model will be evaluated for its ability to generalize to new ASL data, ensuring it can accurately classify hand signs it hasn't seen before.

1.2 Theoretical Background

In order to classify input images into their respective output class, a type of artificial neural network called Convolutional Neural Network (CNN), is employed. A typical CNN comprises three major neural layers: convolutional layer, pooling layer, and fully connected neural layer. The convolutional layer is primarily responsible for extracting features from the input images by passing a kernel or filter through them. The pooling layer helps reduce the image's size while maintaining all of its features. The fully connected layer applies weight matrices and bias vectors to the input. The neurons in both the convolutional and fully connected layers can be dropped at a rate specified by the user to remove unproductive neurons from the network. The diagram in Figure 2 depicts the basic CNN (Convolutional Neural Network) structure, which serves as a foundation for the more complex architectures presented in this project.

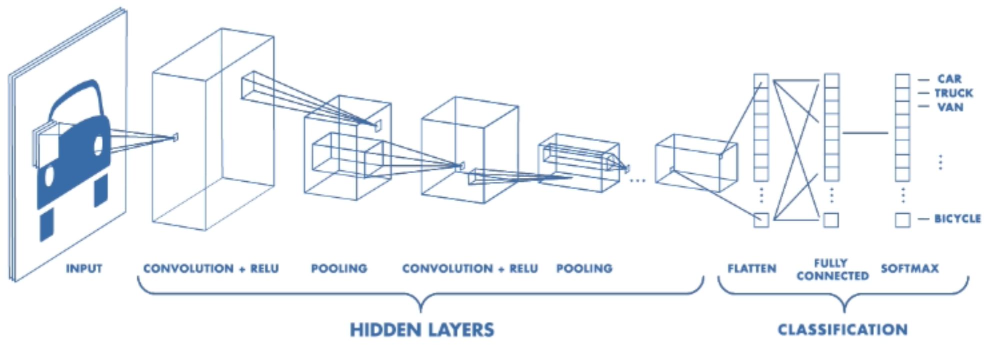


Figure 2: CNN Overall Architecture [10]

The loss function used by models in this work is cross-entropy loss and its equation is given by,

$$C = \sum_{i=1}^{n} y_i ln(a_i)$$

The final layer of the model uses softmax as the activation function which is defined as,

$$a = h(s) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

where $z = (z_1, z_2, ..., z_K)$ and $j = 1, 2, ..., K$

2 Literature Review

Numerous studies have been conducted to explore different methods for recognizing sign language. One study used Principal Component Analysis (PCA) and the K-Nearest Neighbors (KNN) algorithm to recognize American Sign Language. Other studies have used various devices, such as the Microsoft Kinect sensor, an electronic glove, and a Leap Motion controller, to create systems for translating American Sign Language. Another study used a Convex Hull for feature extraction and a KNN for classification in an automated Sign Language Recognition (SLR) system[1][2].

In recent years, several studies have focused on using Convolutional Neural Networks (CNNs) for American Sign Language recognition, often in conjunction with Scale-Invariant Feature Transform (SIFT) feature extraction. These studies have explored different deep-learning techniques and provided benchmarks and explanations of open issues. Another study used a new framework called

Three Subunit Sign Modelling for automated sign language recognition, specifically recognizing ASL alphabets using CNNs. Other studies have also explored using CNNs and Support Vector Machines (SVMs) for recognizing American Sign Language. One study used Support Vector Machines (SVMs) to classify gestures in American Sign Language (SASL) and employed parallel Hidden Markov Models. Another study proposed an approach for classifying sign language and achieved a success rate of 86.67% in recognizing ASL alphabet gestures.

Convolutional Neural Networks (CNNs) using Deep Learning techniques are highly effective for recognizing American Sign Language (ASL). Transfer Learning has become a popular approach for this application, which involves using pre-trained models. Lately, there has been a study conducted to enhance the classification of signs using a combination of hand-crafted features and deep learning methods. This involves utilizing the YCbCr segmentation method based on skin color and the local binary pattern to achieve accurate segmentation of shape and texture features or local shape information. To obtain the features, the transfer learning framework VGG-19 is fine-tuned and then combined with the hand-crafted features using a serial-based fusion technique[3]. One more study has been made on GoogLeNet, where Gesture recognition plays a crucial part in various applications. To achieve this, the aim was to extract finger-related features such as their orientation, status (raised or folded), and image position. This requires image processing as a fundamental step. To identify the contact terms, the program employs a sophisticated approach that differentiates various hand movements. The camera captures a dynamic sequence of images contributing to the hand gesture recognition system. To train and detect hand gestures, GoogLeNet is utilized. Transfer learning is utilized to train GoogLeNet, and American Sign Language (ASL) is used to associate the sign with a specific alphabet[9].

3 Dataset Description

The dataset, which includes both the training and testing data, is 1.11 GB in size and consists of 87,000 jpeg images. Each image has dimensions of 200x200 pixels and is in RGB color format. The dataset folder has two sub-folders, one for training and one for testing.

The training dataset has 29 classes, corresponding to the 26 letters of the English alphabet, and 3 classes, namely SPACE, DELETE, and NOTHING. Each class has 3000 training images and is stored in its own sub-folder. The testing dataset has 29 images, one for each class, stored in its own sub-folder. The dataset can be accessed here

The testing dataset was extended to another test set which contains several real-world images. The dimensions of these images were scaled down to 100 x 100. In this extra dataset, all images were stored in their respective sub-folders. The size of the extra dataset is 2.54 GB having 29 classes, with each sub-folder having 30 images. We captured these images with various backgrounds, such as a bright background and a background matching the skin color.

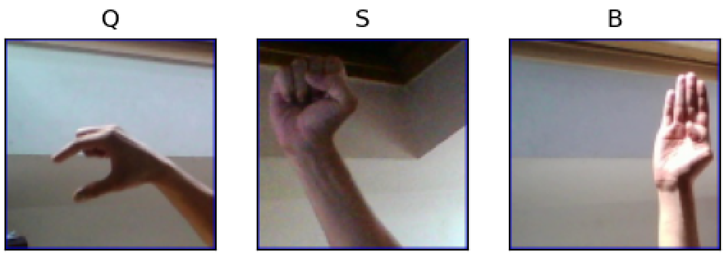


Figure 3: Images from train dataset

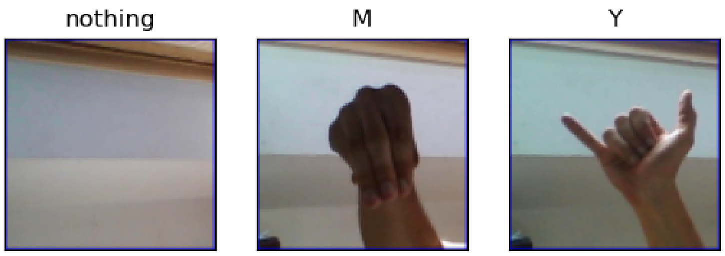


Figure 4: Images from original test dataset

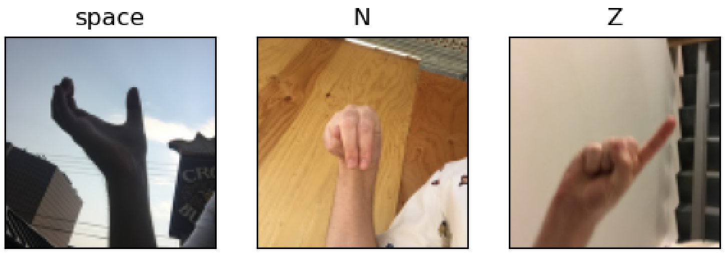


Figure 5: Images from extended test dataset

4 Methodology

This work involves using 4 models outlined in Section 4.1. All models were trained using different hyperparameters only on the Kaggle dataset. The performance was compared, and the best two models were selected. These models were then trained on the entire Kaggle dataset and evaluated using the original test dataset and the extra test dataset. This was done to assess how well these models can generalize real-world images outside the Kaggle dataset.

4.1 Models

Two Artificial Neural Networks (ANN) architectures were selected for this work. The first one involves a generic CNN network; the second one is a custom CNN network that aims to improve the generic CNN network. Finally, two pre-trained architectures, GoogLeNet and DenseNet169, were selected to take advantage of transfer learning. Adam optimizer and L2 Regularization are applied in all models.

4.1.1 Baseline CNN

The baseline CNN architecture consists of 3 convolutional layers, each followed by a max pooling layer and a 2D dropout layer. Additionally, the network has three fully connected (linear) layers, followed by two 1D dropout layers. The first convolutional layer uses a kernel size of 5 and a padding of 2, taking an input tensor with 3 channels representing RGB color channels of the input image. It outputs 32 feature maps. The second layer uses a kernel size of 3 and a padding of 1, taking 32 feature maps as input and outputting 32 feature maps. The third convolutional layer uses a kernel size of 3 and a padding of 1, taking 32 feature maps as input and outputting 64 feature maps. To all of these layers, a max pooling layer is added to reduce the spatial dimensions of the output by a factor of 2, and a dropout layer randomly zeros out 20% of the feature maps. The activation function used is ReLU. The learning rate used was 4e-4, the model was trained on 15 epochs, and weight decay was equal to 1e-3.

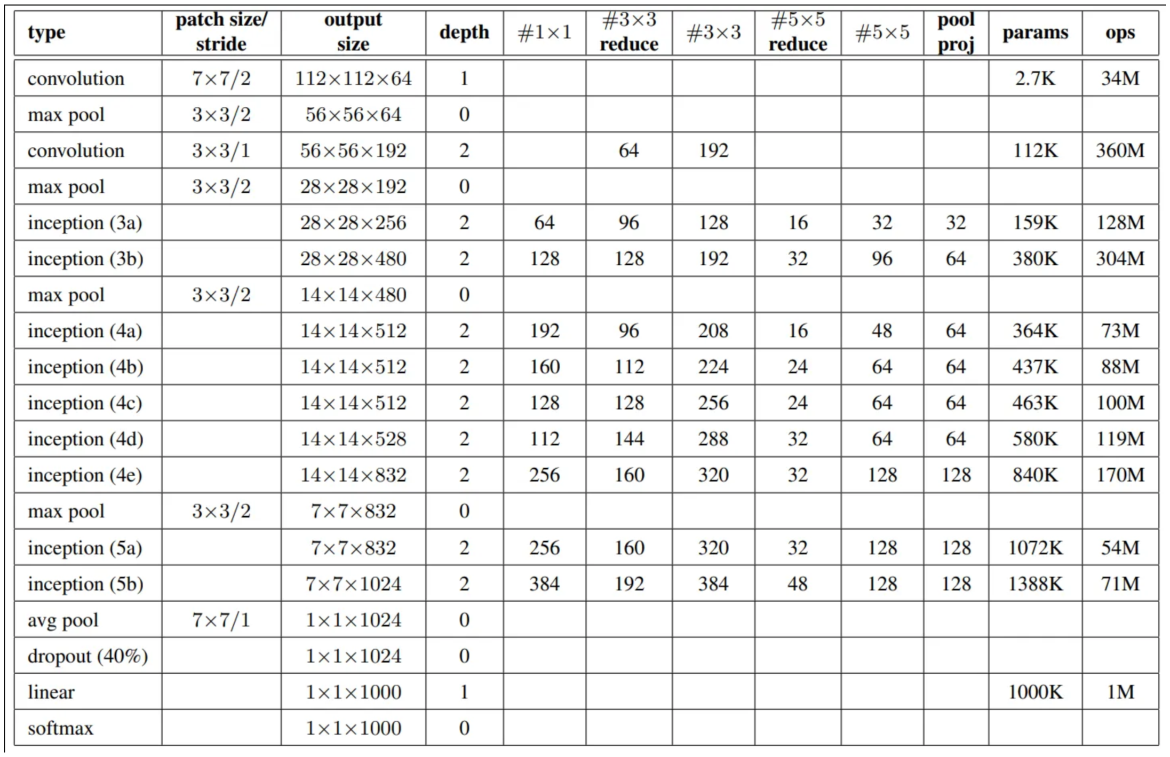
4.1.2 Custom CNN

The baseline CNN architecture consists of 6 convolutional layers, each followed by max-pooling layers and dropout layers. Additionally, the network has three fully connected (linear) layers, followed by two 1D dropout layers. The first convolutional layer has three channels input and 64 output channels with a kernel size of 3 and padding of 1. The second convolutional layer has 64 output channels with a kernel size of 3 and padding of 1. The third and fourth convolutional layers have 128 output channels with a kernel size is 3 and padding of 1. The fifth and sixth convolutional layers have 256 output channels with a kernel size is 3 and padding of 1. To all of these layers, a max pooling layer is added to reduce the spatial dimensions of the output by a factor of 2, and a dropout layer randomly zeros out 20% of the feature maps. The activation function used is ReLU. The learning rate used was 1e-4, the model was trained on 15 epochs, and weight decay was equal to 1e-3.

4.1.3 GoogleNet

GoogLeNet is a Convolutional Neural Network (CNN) model with 27 layers, including three pooling layers and nine inception modules. It has several characteristic features, described in the table headings shown in Figure 2. The "type" column lists the different layers of the GoogLeNet model, while the "patch size" column indicates the size of the kernel (filter) used in the convolutional and pooling layers. In GoogLeNet, the height and width of the kernels are the same, and "stride" is used to define the padding value of the kernel.

The "depth" column refers to the number of neurons present in each layer, and the "#1x1", "#3x3", and "#5x5" columns indicate the number of convolution filters used within each inception module. The "pool proj" column refers to the number of 1x1 filters used within the inception module after pooling. The "params" column displays the total weights in each layer, while the "ops" column shows the number of mathematical operations carried out in each layer.



type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

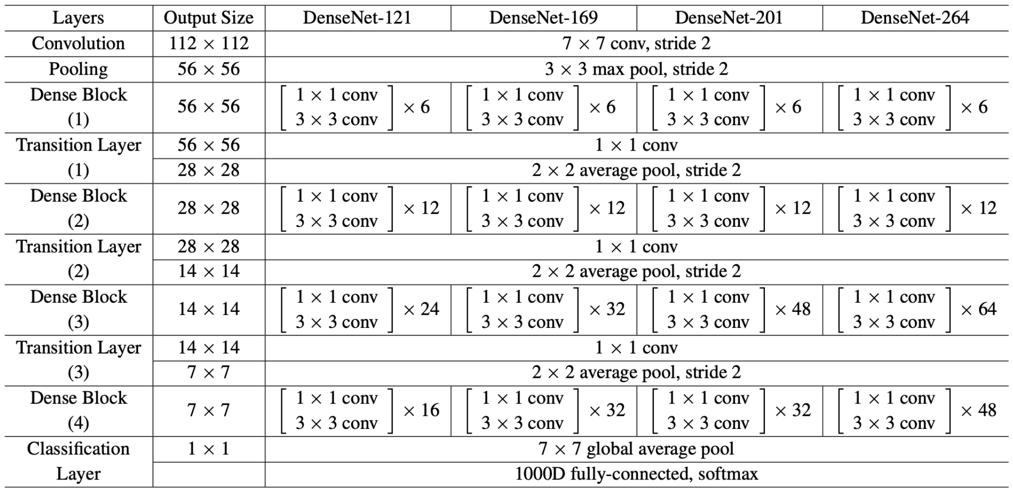
Figure 6: GoogLeNet Configuration Details [11]

In this project, the GoogLeNet model was trained for seven epochs using a learning rate of 5e-4 and a weight decay of 1e-4.

4.1.4 DenseNet

DenseNet is composed of multiple dense blocks, with each block being succeeded by a transition layer that uses pooling and convolution operations to decrease the dimensionality of feature maps. The final layer of the network is a global average pooling layer, which is followed by a fully connected layer that utilizes softmax activation to output the classification probabilities.

The primary concept behind DenseNet involves linking every layer in a feed-forward neural network to each other layer within a dense block. This is accomplished by combining the output feature maps of each layer in the block before feeding them into the next layer. This enables each layer to utilize the feature maps created by all the previous layers in the block, resulting in more comprehensive feature representation and promoting the reuse of features.



Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

Figure 7: DenseNet Configuration Details [5]

In our work, we are using DenseNet169. The model used in this project was trained on five epochs using a learning rate of 5e-4, with a weight decay of 2e-3.

5 Dataset Usage

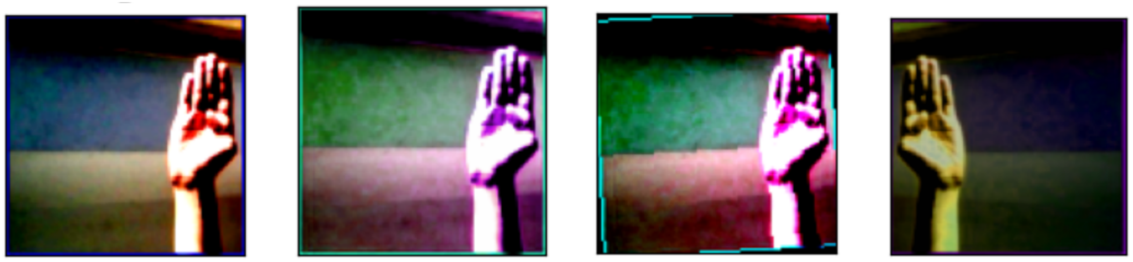
5.1 Pre-Processing

The images in the Kaggle and extra test datasets were captured from different sources. As a result, the image pixel dimensions varied: each image in the Kaggle dataset had a size of 200 x 200, while the extra test dataset had images of pixel dimensions 3024 x 4032. Since these pixel dimensions took a long time to train the models, they were all scaled down to 100 x 100 pixel dimensions. It was also checked that no important features were cropped out from the image after applying this transformation.

Since the Kaggle dataset images were captured in the same environments, there is no variation in brightness, skin color, and background. To change this, we added a random color jitter to generalize the image to real-world images. Specifically, the brightness, contrast, saturation, and hue were altered by a random magnitude. Each image is normalized to ensure that the network parameters have a similar distribution. In general, normalization improves the generalization of an image and also reduces computation time. Since pixels have positive values, the resulting negative normalization values are cut off at zero and contribute to black spaces in the images.

The next step involves rotating the image by a random degree. This ensures that the model learns hand signs that are rotationally invariant, as real-world signs are not likely to be consistently oriented.

The last step involves flipping the images horizontally. All these processes are shown in Figure 8.



(a) Normalized (b) Color Jitter) (c) Rotation (d) Flip

Figure 8: Pre-Processing on Training Dataset

5.2 Data Handling

The Kaggle dataset was split into the train, validation, and test sets. This was done because the Kaggle dataset has a high degree of similarity between many images, and the models that we were implementing were capable of attaining high accuracy with the dataset, and this would make it difficult to compare the models. Therefore, a split of 70/15/15 was used to highlight the models with the best generalization capabilities. While splitting, a random stratified split was used to ensure that all three datasets split had a balanced sample of classes. The hyperparameters for each model are tuned on training and validation sets. Once the best hyperparameters set is finalized, the test set split is used to evaluate each model and select the best two models.

Once the best models are selected, the entire Kaggle dataset is used to train the models until convergence. In these models, the Kaggle and extra test data are used to evaluate the performance.

The initial phase of selecting the best models is done only using normalization and image resize pre-processing step. The final evaluation of the best models is done by augmenting the train data with color jitter, horizontal flip, and rotation pre-processing step as well.

5.3 Evaluation Metrics

During the initial experimental stage, four evaluation metrics were employed as the primary method for evaluating the performance of each model for comparison purposes. Furthermore, these same metrics were also utilized to assess the performance of the final models.

1. Accuracy - The formula provided below is used to calculate the model's accuracy.

$$\frac{TruePositive + TrueNegative}{TruePositive + FalseNegative + TrueNegative + FalsePositive}$$

2. F1-Score - The F1-Score can be defined as the harmonic average of precision and recall.

$$\frac{2 * Recall * Precision}{Recall + Precision}$$

3. Recall - Recall can be defined as the fraction of true positives out of the combined total of true positives and false negatives.

$$\frac{TruePositive}{TruePositive + FalseNegative}$$

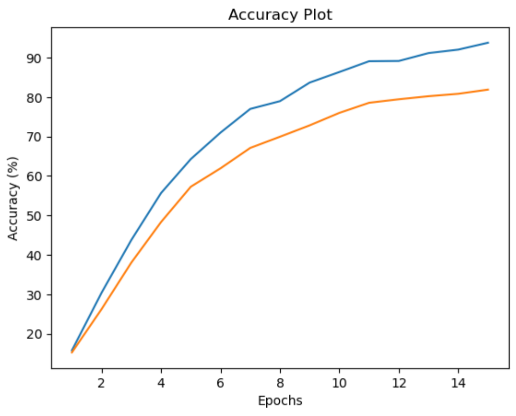
4. Precision - Precision can be defined as the proportion of accurate positive predictions made by the model relative to the total number of positive predictions, including both true and false positives.

$$\frac{TruePositive}{TruePositive + FalsePositive}$$

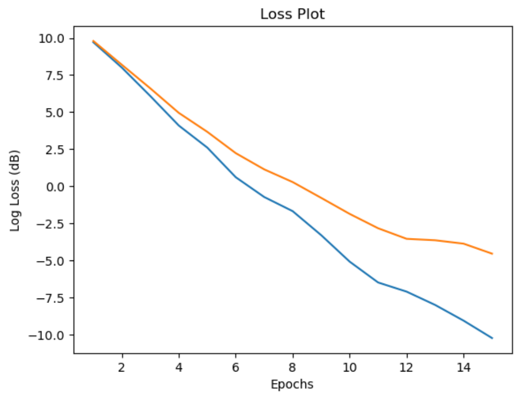
6 Results

All four models are trained iteratively, and their hyperparameters are adjusted accordingly. The training of each model was halted once it converged and before the model was overfitted. This section shows the accuracy and loss curves for the split mentioned in Section 5.2.

The first model is the baseline CNN model, which took 15 epochs to a good generalization of the train and validation sets. The final train accuracy was 93.77%, whereas the validation accuracy was 89.89% with approximately 10% generalization error. This is shown in Figure 9.



(a) Accuracy Plot



(b) Loss Plot

Figure 9: Accuracy and Loss Curves for Baseline CNN

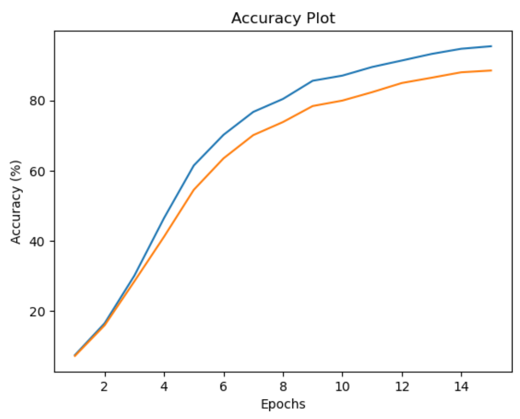
The second model is the custom CNN, which took 15 epochs to converge to a good generalization of the train and validation sets. The final train accuracy was 95.40%, whereas the validation accuracy was 88.50%. This performed relatively better than the baseline CNN and had less generalization error. The respective plots shown in Figure 10

Both the above models performed almost the same but the second model does a better job at generalization. Pre-trained models are used in the next part for the purpose of transfer learning.

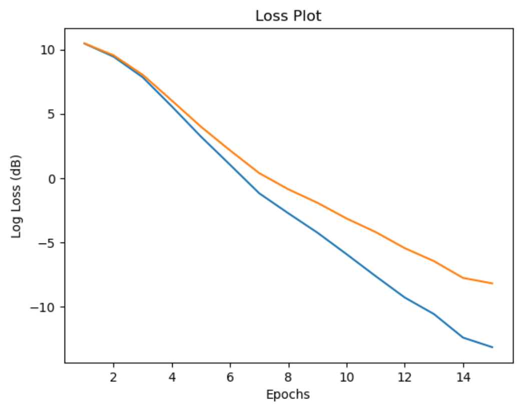
The first pre-trained model used is GoogLeNet. Pre-trained weights are used as the weight initialization, and the curves are plotted as shown in Figure 11. This model took only seven epochs to converge as compared to the previous models. The final train accuracy is 99.31%, and the validation accuracy is 98.44%.

The second pre-trained model used is DenseNet169. Pre-trained weights are used as the weight initialization, and the curves are plotted as shown in Figure 12. This model took only five epochs to converge compared to the previous models. The final train accuracy is 99.37%, and the validation accuracy is 99.39%.

The pre-trained models exhibited the fastest convergence time and attained over 95% validation accuracy after just one epoch and reached close to 100%. The transfer learning models have the least generalization errors in order of 1%. This is far superior to the CNN architectures used. The

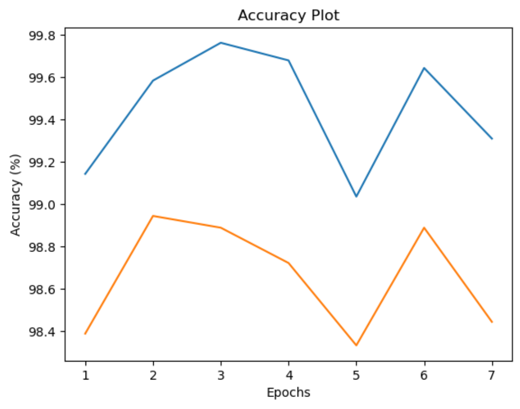


(a) Accuracy Plot

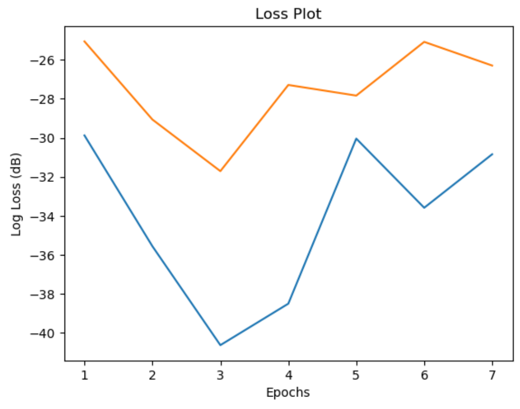


(b) Loss Plot

Figure 10: Accuracy and Loss Curves for Custom CNN

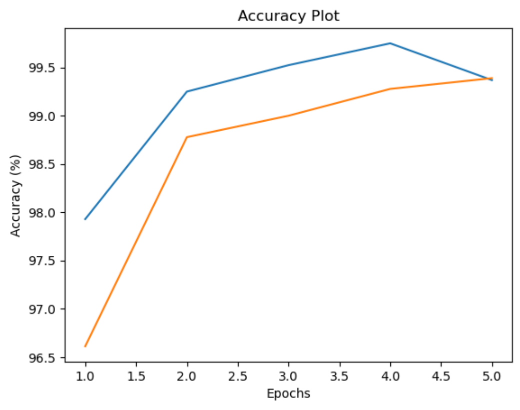


(a) Accuracy Plot

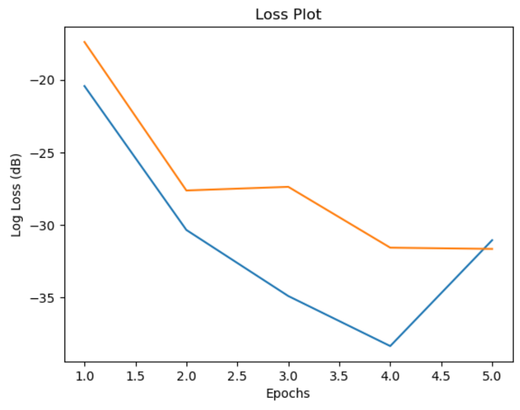


(b) Loss Plot

Figure 11: Accuracy and Loss Curves for GoogLeNet



(a) Accuracy Plot



(b) Loss Plot

Figure 12: Accuracy and Loss Curves for DenseNet169

time taken to train the models and the average epoch time is noted for each model. Also, the evaluation metrics mentioned in Section 5.3 are noted for each model. This is shown in Table 1.

Model	**Average Epoch Time (s)**	**Test Accuracy (%)**	**F1-Score**	**Recall**	**Precision**
Baseline CNN	35.48	83.16	0.8305	0.8316	0.8375
Custom CNN	36.17	87.77	0.8775	0.8778	0.8837
GoogLeNet	41.32	98.44	0.9843	0.9844	0.9854
DenseNet169	59.30	98.55	0.9856	0.9856	0.9861

Table 1: Evaluation metrics for each model on ASL Kaggle Dataset

Model	**Train Time (s)**	**Test Accuracy (%)**	**F1-Score**	**Recall**	**Precision**
GoogLeNet	2172.58	45.56	0.4158	0.4356	0.4839
DenseNet169	2374.26	53.33	0.5214	0.5333	0.5959

Table 2: Evaluation metrics for each model on Extra Dataset

Note that the test accuracy mentioned in the table corresponds to the split of the train dataset and not the actual test dataset. Overall, the average epoch time for all models was almost the same except for DenseNet, which had an average time of 59.3 seconds. The final test accuracies for GoogLeNet and DenseNet were 98.44% and 98.55%, respectively. The additional metrics correspond closely to the accuracy metric and do not need any further explanation.

As mentioned, the best two models are the transfer learning models: GoogLeNet and DenseNet169. These models are selected based on the fact that they reached faster convergence and had higher generalization accuracy. To these models, the Kaggle and extra test datasets are used to evaluate the final test accuracies. In addition, the entire Kaggle dataset is given for the training phase, and all the pre-processing steps mentioned are performed. The following results are shown in Tables 3 and 2

7 Conclusion & Future Scope

This work provided a good generalization on the ASL Kaggle dataset using the mentioned models. The best models were found to be the pre-trained models GoogLeNet and DenseNet169. The use of transfer learning significantly improved the rate of convergence and the accuracy of the model. While these models can generalize on ASL Kaggle dataset, they performed poorly on the extra test set, which contains real-world images of different variations. Pre-processing helped the model's performance using the ASL Kaggle dataset significantly, whereas it improved it to some extent for the extra test set.

The ASL Kaggle dataset and the extra test dataset contain images of sign language shown using only the right hand. The work can be extended to include a person's left hand in the datasets, as many prefer showing sign language using their preferred hand. This requirement makes it a necessary extension to real-world applicability. Also, we have used images, but the work can be improved

Model	**Train Time (s)**	**Test Accuracy (%)**	**F1-Score**	**Recall**	**Precision**
GoogLeNet	2172.58	100	1	1	1
DenseNet169	2374.26	100	1	1	1

Table 3: Evaluation metrics for each model on ASL Kaggle Test Dataset

to detect movement to fully determine their meaning as well. Deep learning models trained on ASL datasets can be integrated into various devices, such as smartphones and tablets, to provide a more accessible interface. Models like Recurrent Neural Networks (RNN) should be explored for application to real-time video feeds.

8 Discussion

8.1 General

The first phase of this project involves finding the best model to classify sign languages based on images. A baseline CNN model architecture was created, which had 3 CNN layers. This performed relatively well but had a slightly high generalization error rate. The custom CNN, aimed at improving the baseline CNN, had more depth in networks, specifically 6 CNN layers. This caused the model to perform much better than the baseline model by showing less generalization error. In addition, both CNN models used the L2 regularization technique and dropout. This was expected as the CNN structure can extract the locally clustered features within an image far more effectively than simpler models or architectures like SVM and KNN.

The next phase was to make use of pre-trained models for transfer learning. The selected models, i.e., GoogLeNet and DenseNet, were far superior to the CNN structures. This was expected as both these models have a far deeper neural network architecture. It is also because these models were pre-trained on the ImageNet dataset, which caused faster convergence and better generalization with high accuracy. These models can pick up similar features, such as edges and curves, from the ASL Kaggle dataset, and only the final layer of the network needs to be retrained.

The final phase involved evaluating the best model's performance on real-world images. This corresponds to the test dataset and the extra test dataset being used. The ASL Kaggle test dataset used had images with very little or no feature difference compared to the train dataset. As expected, the trained models performed well, reaching an accuracy score of 100%. On the other hand, the extra test set had images corresponding to different features such as background color, lighting, skin color, etc. In fact, the hand was very hard to recognize in some images as the image's background was also similar. This test set was deliberately chosen to imitate real-world images. To our surprise, the models performed poorly, reaching an accuracy score of 50%. The model was trained using the Kagle dataset, which had no distinct features, and most images were similar.

8.2 Challenges

Several engineering difficulties arose during this project. The first challenge involved evaluating the four models. Since the images in the Kaggle dataset were very similar, it was hard to differentiate the models on their performance as they were nearly identical when many train samples were used. To show a clear distinction, the train sample size was reduced from 87,000 images to 12,000 images. Doing this reduced computational complexity and provided a clear distinction between the models.

As mentioned before, due to the similarity of the Kaggle dataset, it would be very difficult to generalize on a real-world dataset with the hands of different people in different environments with different lighting conditions. The use of pre-processing techniques helped overcome these issues. In the end, the performance on the extra test set was poor, as some images were not that easy to distinguish. Nonetheless, the model performance was better than expected on such a dataset as the preliminary experiment was performed using minimal pre-processing steps, and the accuracy was worse. Another challenge was coding; multiple individuals contributed to the code, presenting logistical issues. However, these were solved using appropriate code versions and refracting the code to be modular. Another challenge was the hardware capabilities, as some models, like VGG16, required a higher VRAM memory for the training part.

8.3 Questions Answered

This work answered several questions, both anticipated and unanticipated. It was expected that pre-trained models for transfer learning would work well in generalization due to its CNN architecture. The work also answered how pre-processing could affect the accuracies. It showed the benefit of using horizontal flip and color jitter augmentation on the training set to generalize a low-diversity dataset. The extra test set also performs better than the model trained using minimal pre-processing methods. The overall result of the extra test set was unexpected even after data augmentation was performed. A solution could be to use segmentation to augment the training images with different backgrounds to improve the model's generalization. Overall, this work answered how well a model would perform on an unseen real-world ASL dataset.

References

[1] P. Das, T. Ahmed and M. F. Ali, "Static Hand Gesture Recognition for American Sign Language using Deep Convolutional Neural Network," 2020 IEEE Region 10 Symposium (TENSYMP), Dhaka, Bangladesh, 2020, pp. 1762-1765, doi: 10.1109/TENSYMP50017.2020.9230772.

[2] P. Pruthvi and J. Geetha, "Convolution Neural Network for Predicting Alphabet Sign Language and Comparative Performance Analysis of CNN, KNN, and SVM Algorithms," 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2022, pp. 1-6, doi: 10.1109/GCAT55367.2022.9972174.

[3] R. G. Rajan and M. Judith Leo, "American Sign Language Alphabets Recognition using Hand Crafted and Deep Learning Features," 2020 International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 2020, pp. 430-434, doi: 10.1109/ICICT48043.2020.9112481.

[4] Aleksejs Zorins and Peter Grabusts, "Review of Data Preprocessing Methods for Sign Language Recognition Systems based on Artificial Neural Networks", 2016 Information Technology and Management Science Volume 19, doi: 10.1515/itms-2016-0018.

[5] Gao Huang, Zhuang Liu, Laurens van der Maaten and Kilian Weinberger, "Densely Connected Convolutional Networks", 2017, doi: 10.1109/CVPR.2017.243.

[6] S. Thakar, S. Shah, B. Shah and A. V. Nimkar, "Sign Language to Text Conversion in Real Time using Transfer Learning," 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2022, pp. 1-5, doi: 10.1109/GCAT55367.2022.9971953.

[7] A. Chandrasekhar, L. Singh, S. Tripathi and N. Malik, "Sign Language Recognition System Using Deep Learning," 2023 13th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2023, pp. 67-71, doi: 10.1109/Confluence56041.2023.10048804.

[8] D. Kumari and R. S. Anand, "Static Alphabet American Sign Language Recognition using Convolutional Neural Networks," 2022 International Conference on Advances in Computing, Communication and Materials (ICACCM), Dehradun, India, 2022, pp. 1-6, doi: 10.1109/ICACCM56405.2022.10009175.

[9] S. Jadhav, B. Chougula, G. Rudrappa, N. Vijapur and A. Tigadi, "GoogLeNet Application towards Gesture Recognition for ASL Character Identification," 2022 IEEE International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), Ballari, India, 2022, pp. 1-5, doi: 10.1109/ICDCECE53908.2022.9793165.

[10] Blog post: Understanding of Convolutional Neural Network (CNN) — Deep Learning

[11] Blog post: Deep Learning: GoogLeNet Explained

[12] Google images: Sign Language