ASL Recognition using Deep Learning Models

Anuj Doshi, Chinmay Gupta, Surbhi Gupta, Ronit Motiwale

Robert H. Smith School of Business, University of Maryland

BUDT758B: Big Data and Artificial Intelligence for Business

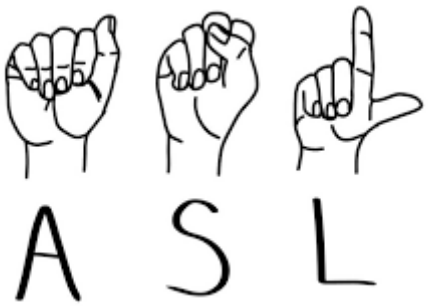
Dr. Kunpeng Zhang

December 18, 2020

Table of Contents

Introduction

American Sign Language (ASL) is the primary language utilized by many hard of hearing, almost deaf and hearing people in North America. It is communicated utilizing signs made with their hands, along with facial motions. It has all the necessary features of a language, with its own principles for word development and word order.



A lot of progress has been made towards developing computer vision systems that translate sign language into spoken language. As per WHO, 466 million individuals experience the ill effects of handicapping hearing loss. They gauge that by 2050, this number will increase to 900 million. A yearly worldwide expense of US $750billion is brought about because of unaddressed hearing loss[8]. 1.1 billion youngsters (matured between 12–35 years) are in danger of hearing misfortune because of presentation to clamor in recreational settings. We need to contribute towards overcoming any barrier between the distinctive client sections inside the populace by building up a model that makes an interpretation of gesture-based communication into English letters in order. Having educated various ideas in profound learning, we concluded it would be an incredible learning experience for us as we actualize our learnings for this reason.

Hence, for our project, we have created a deep learning model by training convolutional neural networks to classify images of American Sign Language letters.

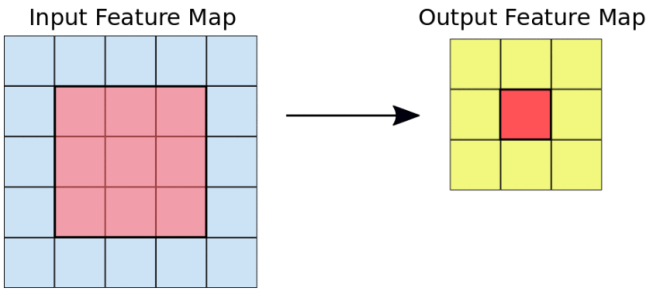
Convolutional Neural Networks for Image Classification

Image classification is the process of identifying images and categorizing them into one of the predefined categories or classes. Neural networks architecture consists of an input layer, many hidden layers and output layer, with the objective to recognize patterns in an image.

Convolutional Neural Network model is one of the most common models used for image classification. Instead of pre-processing the data to derive features, CNN takes the raw input and automatically learns about the features. The image is passed through a series of layers to generate the output. The different layers are:

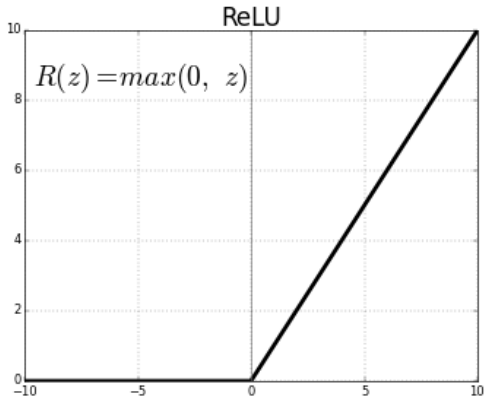
1) Convolution Layer:

A filter produces convolution while moving along the input image matrix.



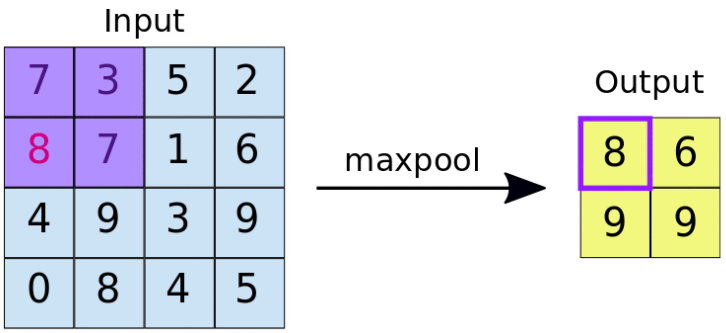
2) Non-Linear Layer:

After each convolution operation, the matrix is passed through an activation function to introduce the nonlinear property in the model. For example, Rectified Linear Unit (ReLU).



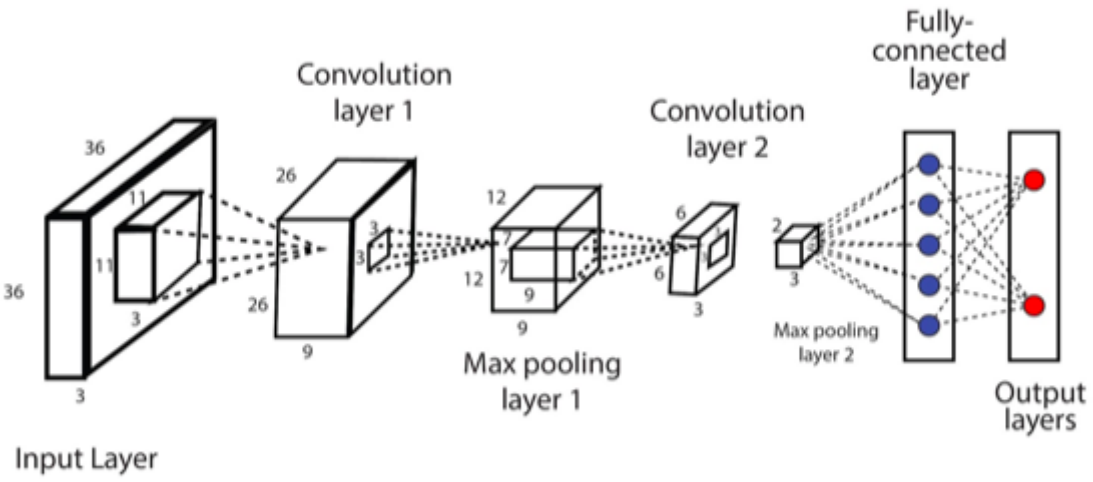
3) Pooling Layer:

Downsampling operation is performed which significantly reduces the number of dimensions on the feature map whilst preserving the critical features. For example, Max Pooling.



4) Fully Connected Layer:

Output from the convolutional network is passed to a fully connected layer which then classifies the images from the predefined classes. The layer consists of an activation function, say softmax which gives us probability distribution from 0 to 1 for each of the labels.



Convolutional Neural Network Architecture

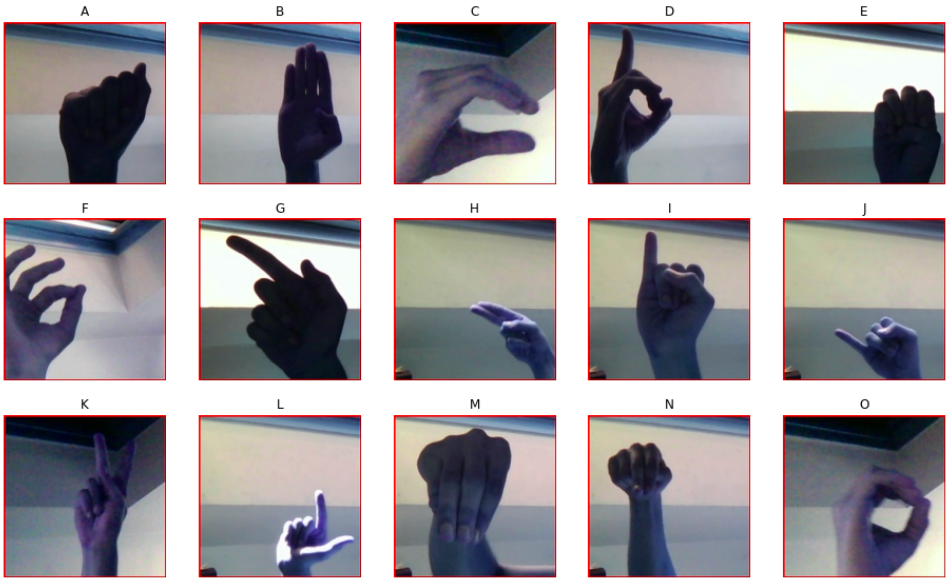
Data Description

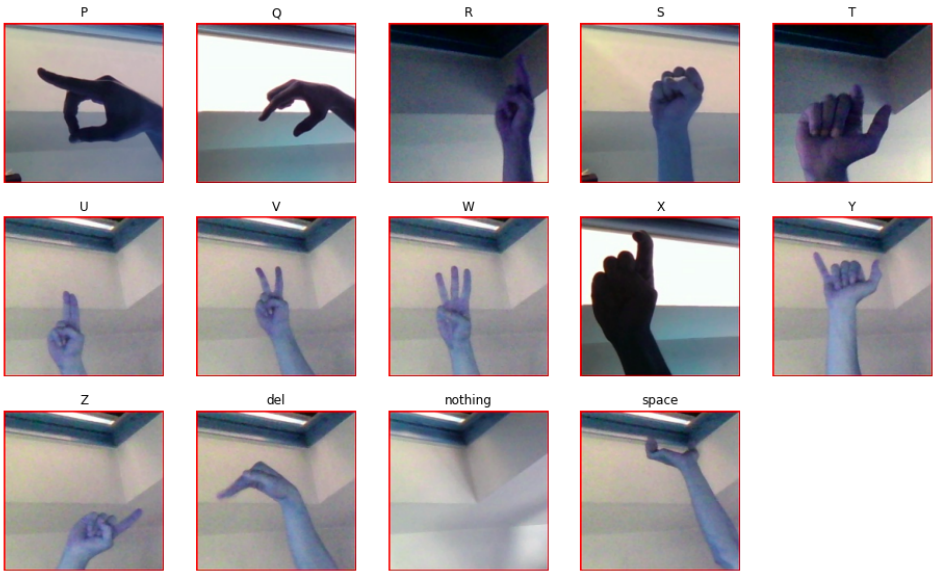
We collected, and analyzed a dataset consisting of 87000 images representing ASL alphabets, delete, nothing and space to validate our deep learning model. We analyzed images with different hands on different backgrounds.The two data sets utilized in our model:

i) ASL Alphabet Train: There are 26 folders representing A-Z English alphabets and 3 separate folders to depict nothing, space and delete. These images were used to train and validate the classifier.

ii) ASL Alphabet Test: The different images were used to test the model created to classify the images. We compared the true labels of the images to the labels predicted by the model.

Sample Data Set:





Data Preprocessing - Image Transformation

1) Resizing of images: All the images were resized from 224x224 to 64x64 for easier computation.
2) Normalization of images: Since all pre-trained models expect normalized input, the images normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
3) Conversion to Torch Tensor: We used PyTorch Framework. Thus, we converted the images to a compatible format.

```
# Validation transformation
# Necessary transforms only
validation_transform = transforms.Compose([transforms Loading... 64), interpolation=2),
                                transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])])
# Test transformation
# Necessary transforms only
test_transform = transforms.Compose([transforms.Resize((64,64), interpolation=2),
                                transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])])
```

Libraries Imported:

Framework**:** PyTorch.

Components used:

```
import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils import data

import torch.utils.data as data_utils

from torch.utils.data import Dataset, DataLoader

import torch.optim as optim
```

```
import torch.optim.lr_scheduler as scheduler

import torchvision

import torchvision.models as models

from torchvision import transforms
```

Utils:

```
import numpy as np

import time

import datetime
```

Visualization:

```
import matplotlib.pyplot as plt
```

Metrics:

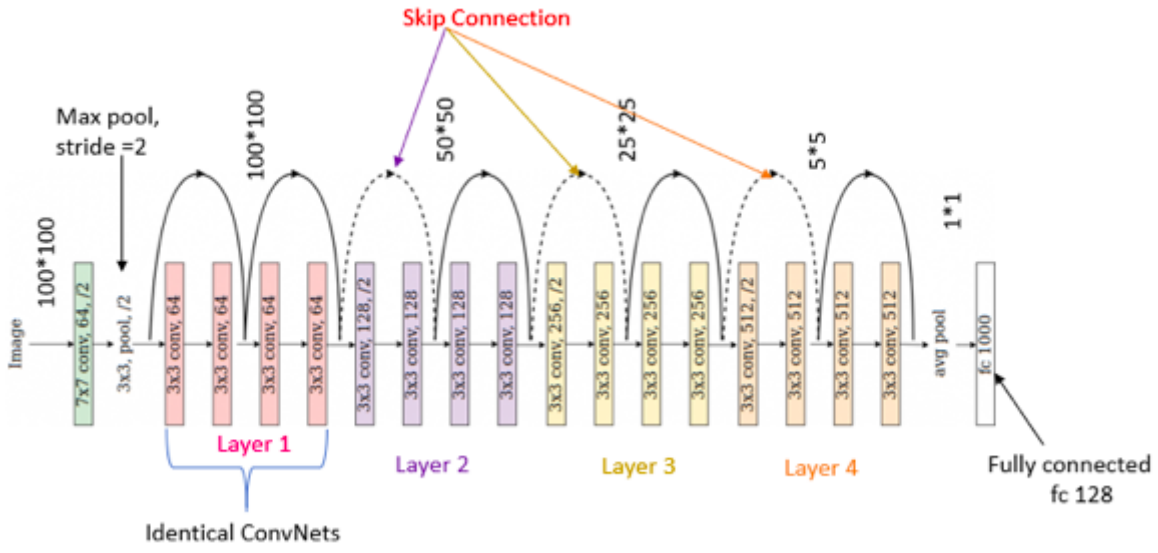
```
from sklearn.metrics import confusion_matrix
```

Existing Models Specifications:

Pre-trained models are models that have been previously trained and have learnt the parameters such as weights and biases. This knowledge can then be used on different data sets for prediction. Different existing models used:

1) ResNet18

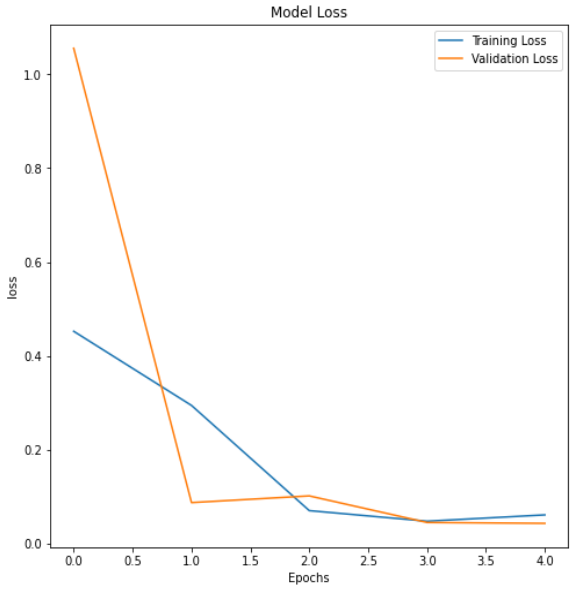
The model is pre-trained on ImageNet dataset [1]. The network is 18 layers deep. There are 11.174M parameters in the ResNet18 architecture[2] and it can help classify images into 1000 object categories. The network has an input size of 224*224.

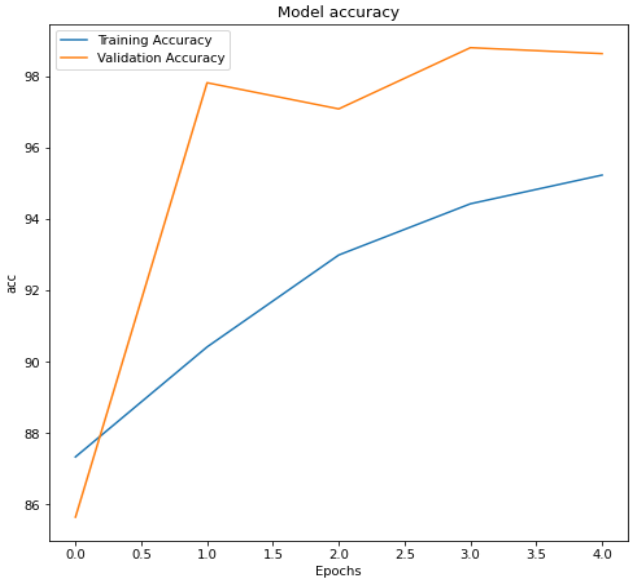


ResNet-18 Architecture

Model Performance:

Accuracy and Loss Metrics:

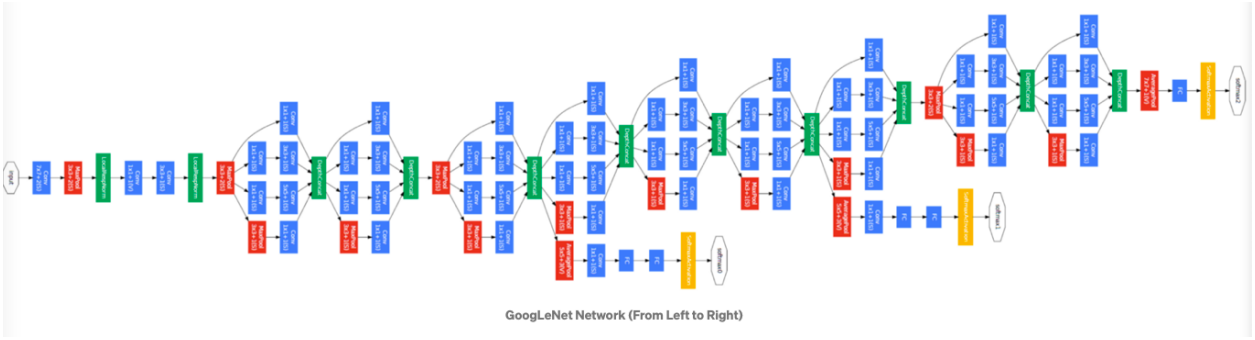




Epoch Number	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Loss	Test Accuracy
0	0.45	87.33%	1.05	85.64%	0.0	93.54%
1	0.29	90.41%	0.08	97.82%		
2	0.07	92.99%	0.10	97.08%		
3	0.04	94.43%	0.04	98.80%		
4	0.06	95.23%	0.04	98.63%		

2) GoogleNet:

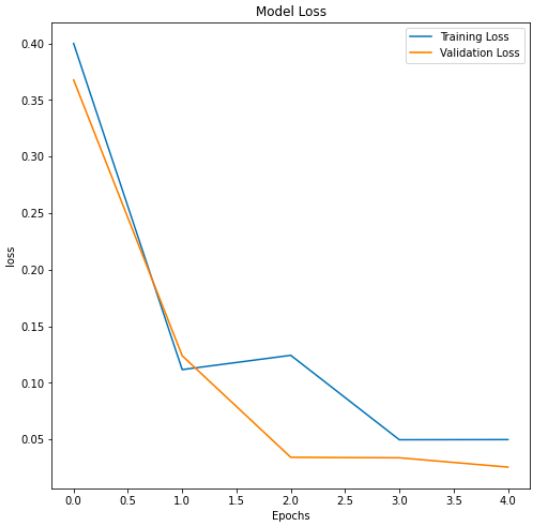
The model which is pre-trained on ImageNet and Place365 dataset.The network is 22 layers deep. The network has an input size of 224*224.[3] The model was the winner of ILSRVRC 2014 with a top-5 error rate of less than 7%.[4]

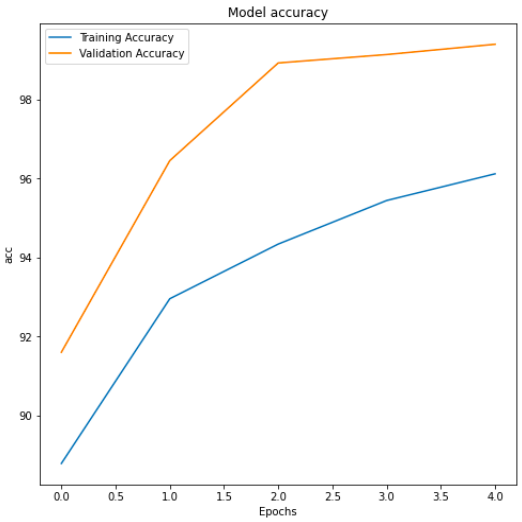


GoogleNet Architecture

Model Performance:

Accuracy and Loss Metrics:

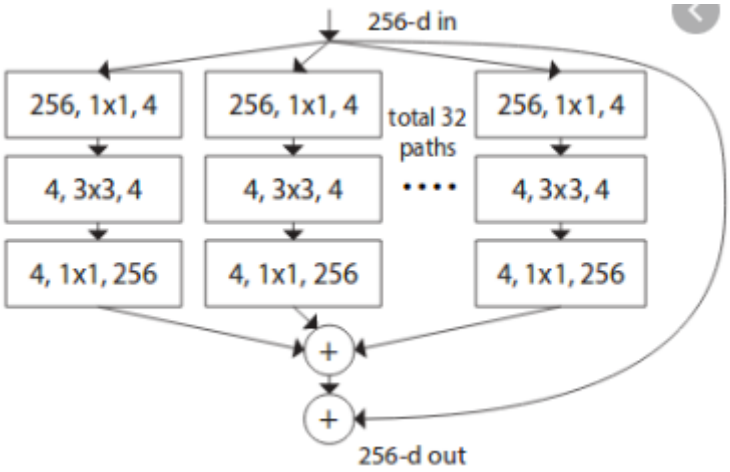




Epoch Number	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Testing Loss	Testing Accuracy
0	0.41	88.78%	0.37	91.59%	0.0	96.77%
1	0.11	92.95%	0.12	96.45%		
2	0.12	94.34%	0.03	98.92%		
3	0.05	95.44%	0.03	99.14%		
4	0.05	96.12%	0.03	99.39%		

3) Resnext50_32x4d:

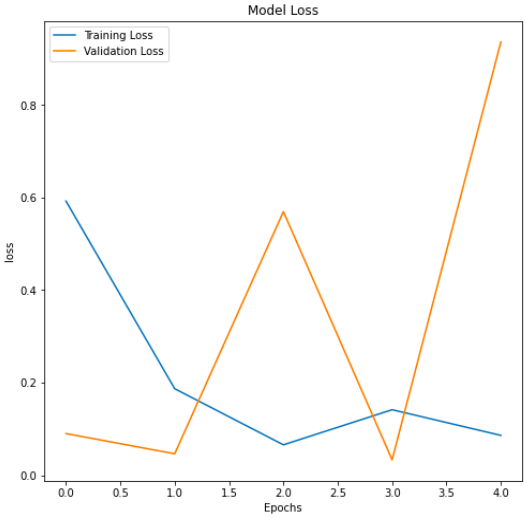
The model is pre-trained on ImageNet dataset. The model is 50 layers deep with a top-5 error rate of 6.30% [1].

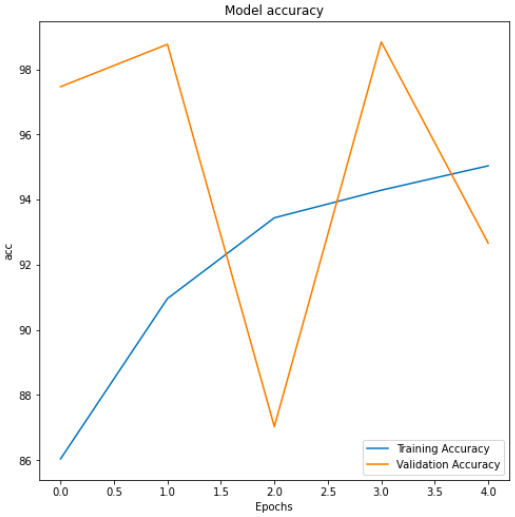


ResNext50 with 32 cardinality Architecture

Model Performance:

Accuracy and Loss Metrics:



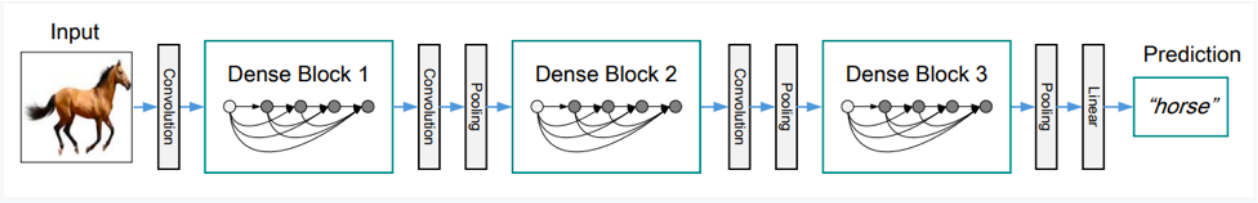


Epoch Number	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Loss	Test Accuracy
0	0.59	86.04%	0.09	97.47%		
1	0.18	90.96%	0.05	98.77%		
2	0.07	93.44%	0.57	87.03%	0.0	90.32%

3	0.14	94.29%	0.03	98.84%		
4	0.87	95.03%	0.94	92.66%		

4) DenseNet:

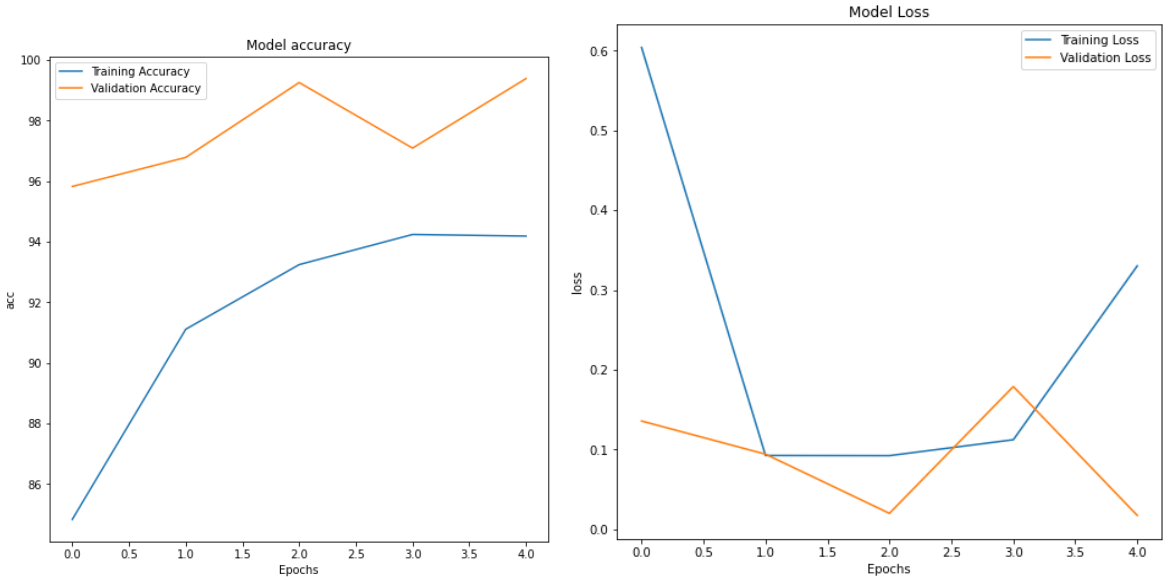
The model has been pre-trained on ImageNet image dataset with a top 1 error rate of 22.2% [5]. DenseNet layers add a small set of feature maps. Each layer has a direct access to the gradients from the loss function and the original input image [6].



DenseNet Architecture

Model Performance:

Accuracy and Loss Metrics:



Epoch Number	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Testing Loss	Testing Accuracy
0	0.53	86.78%	0.40	93.22%		
1	0.14	91.67%	0.07	97.89%		

2	0.27	92.55%	0.65	87.48%	0.0	96.77%
3	0.05	94.08%	0.01	99.76%		
4	0.01	95.20%	0.01	99.78%		

Our Proposed Model:

Network Structure:

Input Layer: (3 ,64, 64)

Number of convolution Layers: 5

Number of Pooling Layers: 3

Dropout Layers: 5

Activation Functions: ReLU

Linear Layer: (512, 29)

```
CNN(
  (layer): Sequential(
    (0): Conv2d(3, 32, kernel_size=(7, 7), stride=(2, 2), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Dropout2d(p=0.1, inplace=False)
    (4): BasicBlock(
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (5): BasicBlock(
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (6): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): Dropout2d(p=0.1, inplace=False)
    (10): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
```

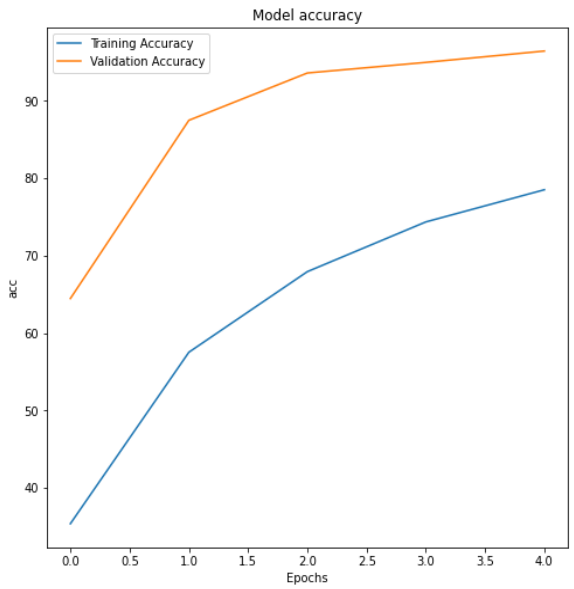
```
(11): BasicBlock(
  (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(12): AvgPool2d(kernel_size=2, stride=2, padding=0)
(13): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), bias=False)
(14): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(15): ReLU(inplace=True)
(16): Dropout2d(p=0.1, inplace=False)
(17): BasicBlock(
  (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(18): BasicBlock(
  (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(19): Conv2d(128, 256, kernel_size=(2, 2), stride=(1, 1), bias=False)
(20): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(21): ReLU(inplace=True)
(22): Dropout2d(p=0.1, inplace=False)
(23): BasicBlock(
  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)


   (24): BasicBlock(
     (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
     (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), bias=False)
     (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
   (25): AvgPool2d(kernel_size=2, stride=2, padding=0)
   (26): Conv2d(256, 512, kernel_size=(2, 2), stride=(1, 1), bias=False)
   (27): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   (28): ReLU(inplace=True)
   (29): Dropout2d(p=0.1, inplace=False)
   (30): BasicBlock(
     (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), bias=False)
     (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
   (31): BasicBlock(
     (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (conv3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), bias=False)
     (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
   (32): AvgPool2d(kernel_size=2, stride=2, padding=0)
 )
 (linear): Sequential(
   (0): Linear(in_features=512, out_features=29, bias=True)
 )
)
```

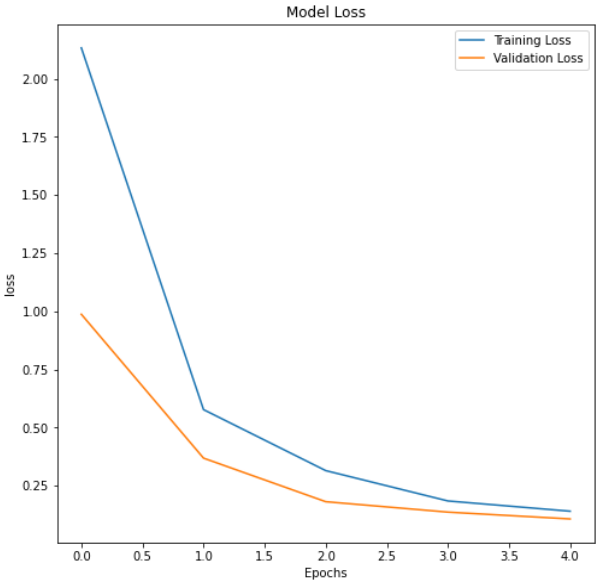
We have used multiple Basic Blocks in our custom CNN model to tackle the gradient vanishing problem and ensure gradient flows to the first layer while backpropagating. This is similar to a Resnet model. Basic Blocks are blocks which perform only 2 basic operations. They are also called as shortcut connections.

To obtain high accuracy in comparison to pretrained models, we fine tuned our parameters by hit and trial method.

Model Performance:

Accuracy and Loss Metrics:





Epoch Number	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Loss	Testing Accuracy
0	2.13	35.34	0.98	64.45	0.18	96.77%
1	0.57	57.51	0.36	87.50		
2	0.31	67.93	0.18	93.62		
3	0.18	74.36	0.13	94.99		
4	0.13	78.52	0.10	96.46		

Challenges Faced:

1) We used Google Collab to run our various image classification models. Initially, we had a training set of approximately 87,000 images. Due to the shortage of RAM on the Google Colab GPU, we were unable to run our models for the large data set. As a result, we had to trim our training dataset and use 27,000 images to train our classification models and validate it on a set of approximately 9,000 images.

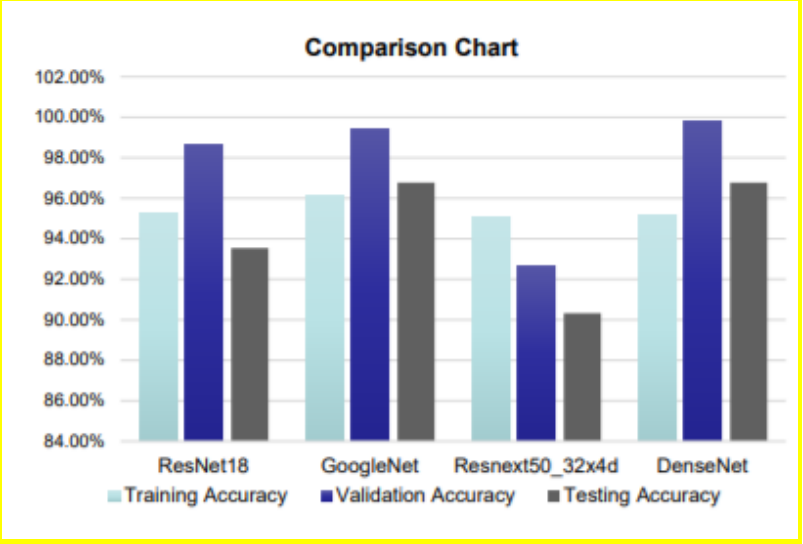
2) Additionally, to tackle the issue of out of memory on Google Colab, we had to change the input image size from (224, 224) to (64, 64).

3) Initially, our model kept on generating the following error:
"CUDA error 59: Device-side assert triggered."
To tackle this, we included (CUDA_LAUNCH_BLOCKING="1") at the very beginning of the code to make sure we get a useful stack trace.[7]

4) We faced an issue in detecting ASL alphabets from a video stream as this process was computationally challenging for our machines. Therefore, we had to reduce the scope of our project to design a significant model to detect individual images, instead of a video stream.

Result:

We were able to successfully achieve similar results as that of the pre-trained models. Our model used lesser inference time due to lesser parameters used. We fine tuned our parameters such as kernel size, stride, padding, number of neurons in each layer using hit and trial method to achieve high accuracy.



Model Name	Training Accuracy	Validation Accuracy	Testing Accuracy
Resnet18	95.23%	98.63%	93.54%
Googlenet	96.12%	99.39%	96.77%
Resnext50_32x4d	95.03%	92.66%	90.32%
Densenet	95.20%	99.78%	96.77%
Custom Model	78.52%	96.46%	96.77%

Conclusion:

In our project, we were successful in planning, executing, and evaluating Deep Learning Networks to enable interpretation of ASL gesture-based communication into the English language. Our custom CNN model was able to perform at par with the existing pre-trained models to accomplish test accuracy of 96.77% and test loss of 0.18%. Given the promising execution, we believe that our custom CNN model makes a critical contribution towards interpreting ASL. Going forward, we would like to further work on extending our custom model to enable translating ASL videos to English texts.

References:

[1]: *Resnet-18-pytorch - OpenVINO™ toolkit*. (n.d.).

https://docs.openvinotoolkit.org/latest/omz_models_public_resnet_18_pytorch_resnet_18_pytorch.html

[2]: Ruiz, P. (2019, April 23). *Understanding and visualizing ResNets*. Medium.

https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

[3]: *GoogLeNet convolutional neural network - MATLAB googlenet*. (n.d.). MathWorks - Makers of MATLAB and Simulink - MATLAB & Simulink

https://www.mathworks.com/help/deeplearning/ref/googlenet.html

[4]: *Convolutional neural network architecture: Forging pathways to the future*. (n.d.). MissingLink.ai.

https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-architecture-forging-pathways-future/

[5]: *Liuzhuang13/DenseNet*. (n.d.). GitHub. https://github.com/liuzhuang13/DenseNet

[6]: Ruiz, P. (2018, October 18). *Understanding and visualizing DenseNets*. Medium.

https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a

[7]: Ogayo, P. (2019, November 14). *CUDA error: Device-side assert triggered*. Medium

https://towardsdatascience.com/cuda-error-device-side-assert-triggered-c6ae1c8fa4c3

[8]: WHO | World Health Organization

https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss#:~:text=Over%205%25%20of%20the%20world's,will%20have%20disabling%20hearing%20los

Appendix

README File

We used google colab with GPU as Hardware Accelerator for running all our models.
Our code can be divided into 2 different parts:

1. Predefined models
2. Custom model

Part 1: Predefined Model

For the first part, each of us ran different predefined models on our system to save time and proceed quickly. There are 2 changes that need to be made for each predefined model.

Step 1. Enable GPU Accelerator on Colab
Step 2. Uncomment the model that you want to use.

```
## Pre-trained models


#resnet18 = models.resnet18(pretrained=True)
#resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
# densenet = models.densenet161(pretrained=True)
googlenet = models.googlenet(pretrained=True)
```

Step 3. Select the model and pass it.

```
# Defining a model (pre-trained)
model = CNN(googlenet) #make change here
```

Part 2: Custom Model

Step 1. Enable GPU Accelerator on Google Colab
Step 2. Click on Run All

Colab Links for Code:

Part 1: Predefined Models

Resnet18:https://colab.research.google.com/drive/1RM2jPf008Pr7Uxy3QcM10xYbQ9pd4aL_?usp=sharing

Googlenet:https://colab.research.google.com/drive/1EAIBoCTsIc7nMXpcvu5bnjLPie8Yvs-l?usp=sharing

Dense Net:
https://colab.research.google.com/drive/1crSTZ-AYvs3oLz03xHXv7OqCTnTZQi3X?usp=sharing

Resnext50_32x4d:https://colab.research.google.com/drive/1708pFWYnlJEddd8AenFmyPn5y6hbiGRV?usp=sharing

Part 2: Custom Model

https://colab.research.google.com/drive/1qmHTKO1oJNDJ43E0V8ElLe_3CJq0dN75?usp=sharing