ASL (American Sign Language) Alphabet Classification Using Convolutional Neural Network

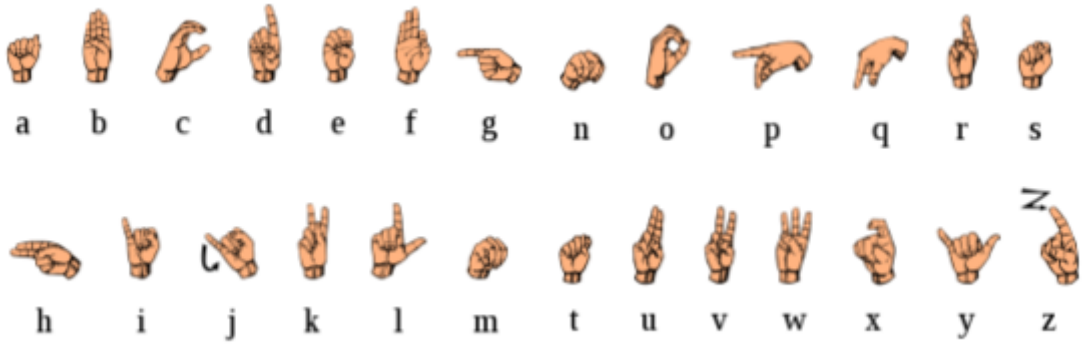
Deep Learning Practical Workshop, Assignment 1

Introduction

This project delves into the realm of American Sign Language (ASL) Alphabet Classification using Convolutional Neural Networks (CNNs). ASL, a vital visual language within the Deaf and hard-of-hearing community, serves as a unique mode of communication. In computer vision and artificial intelligence, recognizing ASL gestures holds great significance.

This endeavor's primary goal is to harness deep learning's capabilities, specifically CNNs, to build a model adept at precisely identifying and classifying the distinctive hand gestures that represent each letter of the English alphabet in ASL. This project demonstrates the prowess of advanced technologies and contributes to fostering communication and accessibility for the Deaf community.

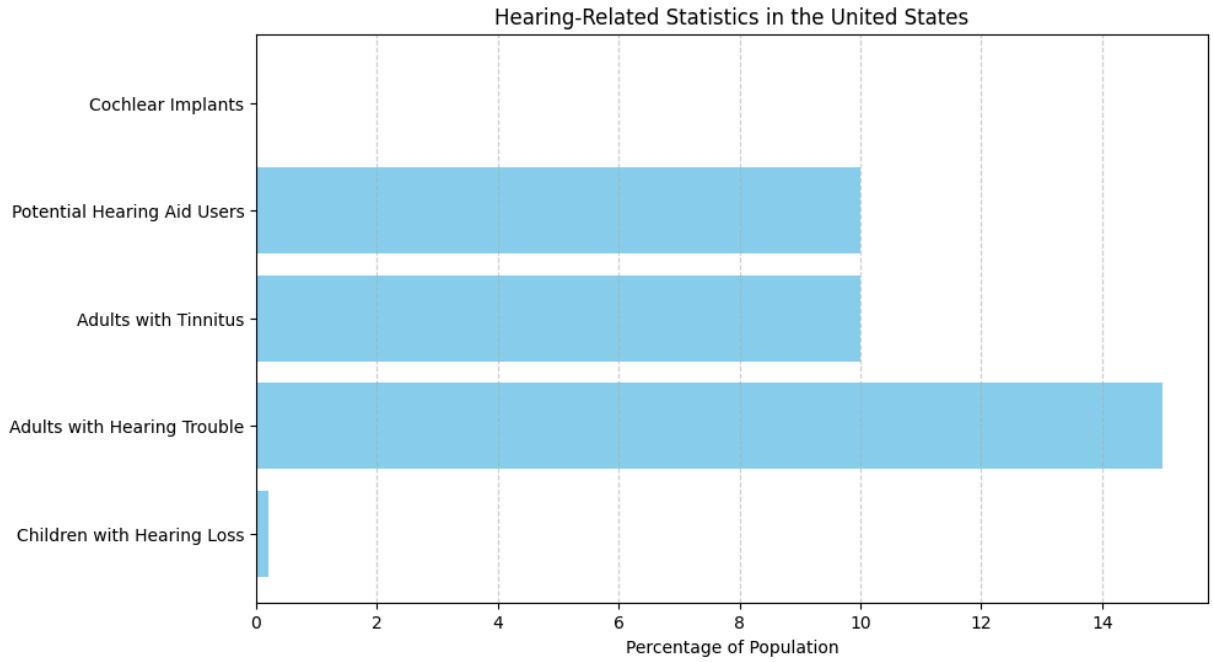
By exploring the complexities of image recognition, our work endeavors to bridge the gap between computer vision and the intricate nature of sign language. We anticipate that this project will positively impact the field of assistive technology, steering us toward a future that is more inclusive and accessible.



Kathy Eva Agafonov	Nadav Cherry	Noam Cohen
206332348	209264241	312213218

Motivation using ASL dataset

The insights derived from hearing-related statistics underscore the importance of developing inclusive ASL recognition models. As we navigate through the prevalence of hearing loss and its diverse impact, the need for robust datasets and representative models becomes paramount. Additionally, our project aligns to address the usage gap in hearing aids, utilizing technology to enhance accessibility and communication for individuals facing hearing-related challenges.



Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Kathy Eva Agafonov	Nadav Cherry	Noam Cohen
206332348	209264241	312213218

Task 1

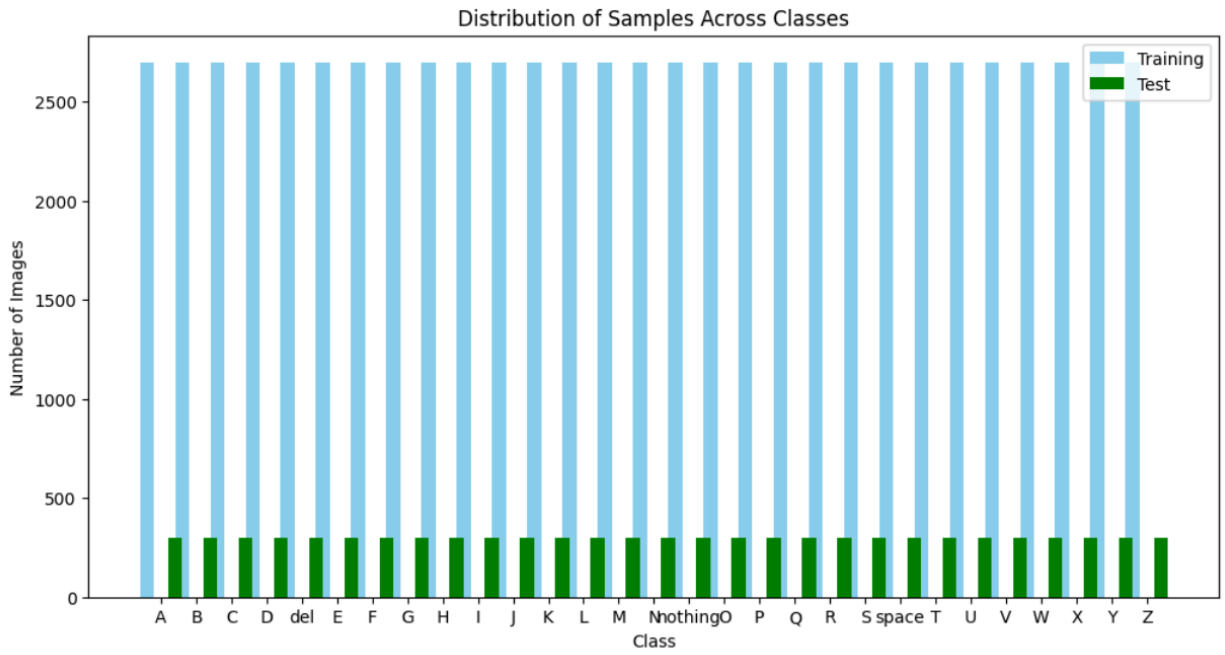
Data Gathering

We used data sets from two different sources.

Source 1: https://www.kaggle.com/datasets/grassknoted/asl-alphabet

Source 2: https://www.kaggle.com/code/sharmayashi/signlangdetection/input

We mixed the data and used an equal number of images and this is the distribution of the data:



Benchmarks

The first benchmark model, trained for 5 epochs, achieved a validation accuracy of approximately 89.5%. The training process used a deep neural network,

and the code was executed on Kaggle Kernels with a GPU.

Link: https://www.kaggle.com/code/dansbecker/running-kaggle-kernels-with-a-gpu

The second benchmark model, trained for 10 epochs, achieved a higher validation accuracy of around 91.2%. This model also utilized a convolutional neural network (CNN) implemented with the Keras framework.

Link: https://www.kaggle.com/code/paultimothymooney/interpret-sign-language-with-deep-learning

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Data Pre-processing

- Our photos were not the same size and we resized them to 200x200**.**
- Our images are colored, with 3 dimensions (RGB) and we have converted to grayscale images to reduce dimensions– 1 dimensionality**.**
- We turned the images into tensors so that we could work with the model.
- We use Normalization.
- Number of Unique Classes in the Dataset: 29.

Data augmentation

The information was very rich, and it seems that some of the images were already augmented beforehand and there was no need for us to do it again.

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Kathy Eva Agafonov	Nadav Cherry	Noam Cohen
206332348	209264241	312213218

Task 2

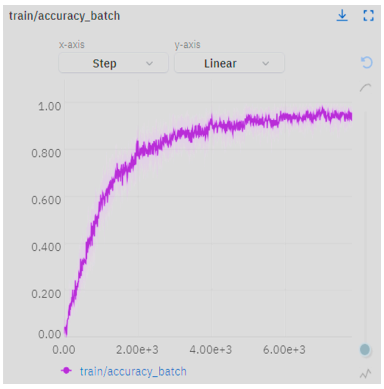
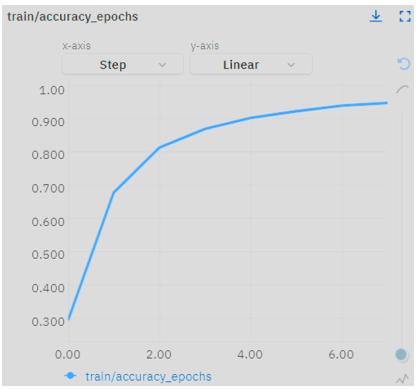
The process of building our model:
First, we created a CNN model, we defined the layers for it according to what was learned in the lecture.
We can see that the number of parameters that this model will learn is 8.1 million parameters:

```
self.conv1 = nn.Conv2d(1,32,3,padding='same')
self.conv2 = nn.Conv2d(32,64,3,padding='same')
self.conv3 = nn.Conv2d(64,32,3,padding='same')
self.conv4 = nn.Conv2d(32,64,3,padding='same')
self.linear1 = nn.Linear(50*50*64,50)
self.linear2 = nn.Linear(50,self.num_classes)
self.mp = nn.MaxPool2d(2,2)
self.relu=nn.ReLU()
```

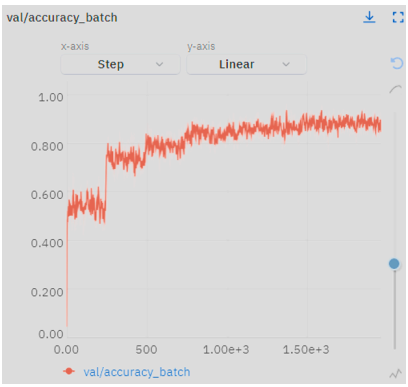
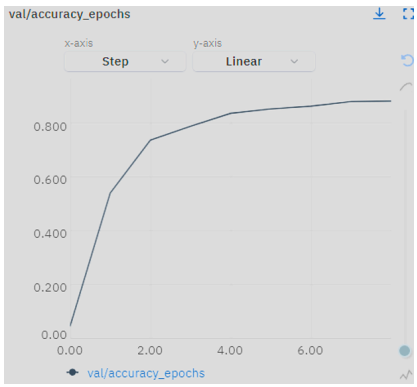
```
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.mp(x)
    x = self.relu(self.conv3(x))
    x = self.relu(self.conv4(x))
    x = self.mp(x)
    x = x.view(-1,50*50*64) ### WHY -1
    x = self.relu(self.linear1(x))
    x = self.linear2(x)
    return F.log_softmax(x,dim=1)
```

```
    | Name           | Type               | Params
-------------------------------------------------------
0   | conv1          | Conv2d             | 320
1   | conv2          | Conv2d             | 18.5 K
2   | conv3          | Conv2d             | 18.5 K
3   | conv4          | Conv2d             | 18.5 K
4   | linear1        | Linear             | 8.0 M
5   | linear2        | Linear             | 1.5 K
6   | mp             | MaxPool2d          | 0
7   | relu           | ReLU               | 0
8   | val_accuracy   | MulticlassAccuracy | 0
9   | test_accuracy  | MulticlassAccuracy | 0
10  | train_accuracy | MulticlassAccuracy | 0
-------------------------------------------------------
8.1 M     Trainable params
0         Non-trainable params
8.1 M     Total params
32.229    Total estimated model params size (MB)
```

We employed Stratified-KFold, a variant of KFold that preserves the percentage of samples for each class or label. The initial fold delivered the most promising outcomes, as indicated by the attained accuracy. The precision is visually presented across epochs, illustrating the accuracy concerning both training and validation batches.
Notably, the test accuracy reached 88.5%.

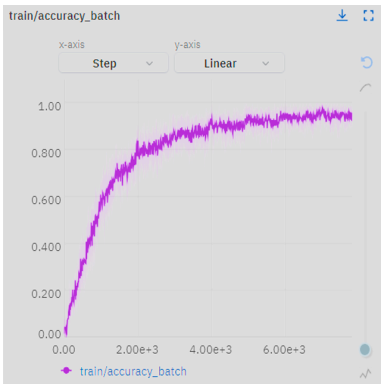
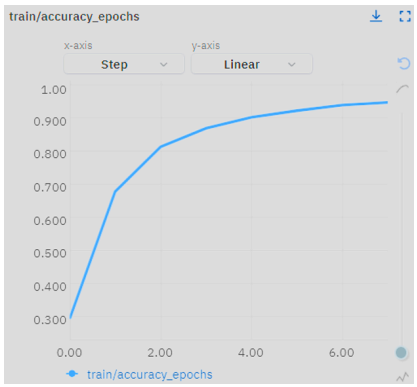
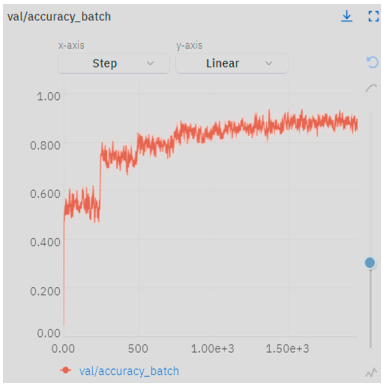
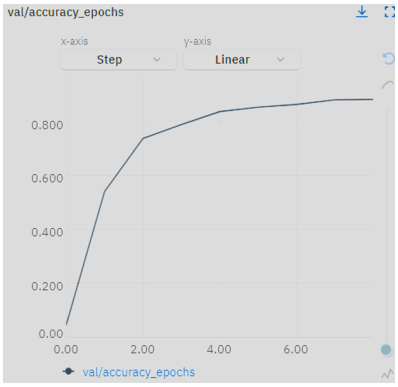
Accuracy	Accuracy for batch	Accuracy for epoch
Train = 0.947		

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Validation=0.884		

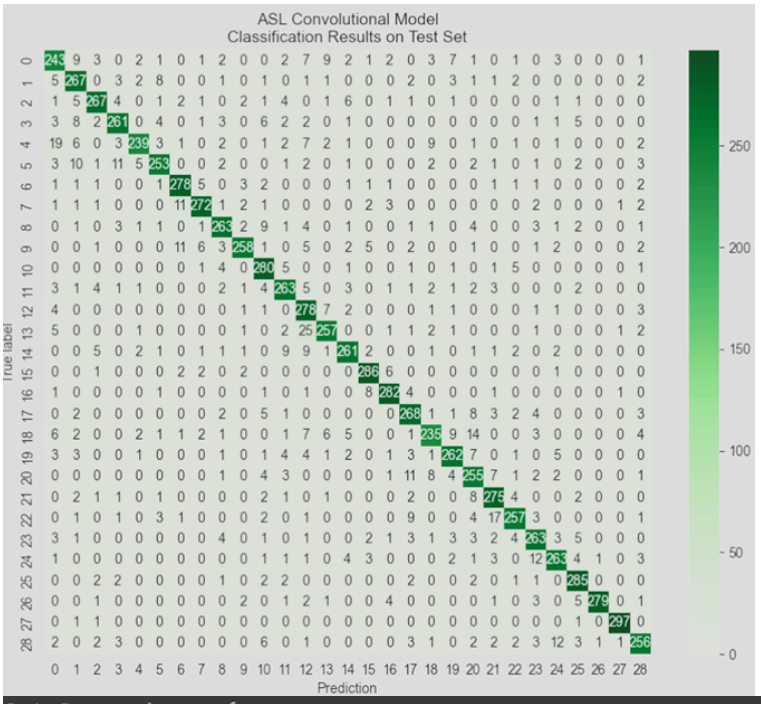
Here, the outcomes of the loss for this fold are presented. The graphs illustrate the loss across epochs and batches for both training and validation data.

Notably, the test loss reached 0.512.

Loss	Accuracy for batch	Accuracy for epoch
Train = 0.947		
Validation=0.884		

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Another measure we used to analyze the results is a confusion matrix, so we could see which images our model is mostly confused about, what it guesses well and what it guesses less.
The y-axis is the real label
The x-axis is the prediction of the model.
Attached is the mapping we did for the letters




```
label_mapping = {
        'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9,
        'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R': 17, 'S': 18,
        'T': 19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25, 'del': 26,
        'nothing': 27, 'space': 28
    # , 'five': 29
    }
```

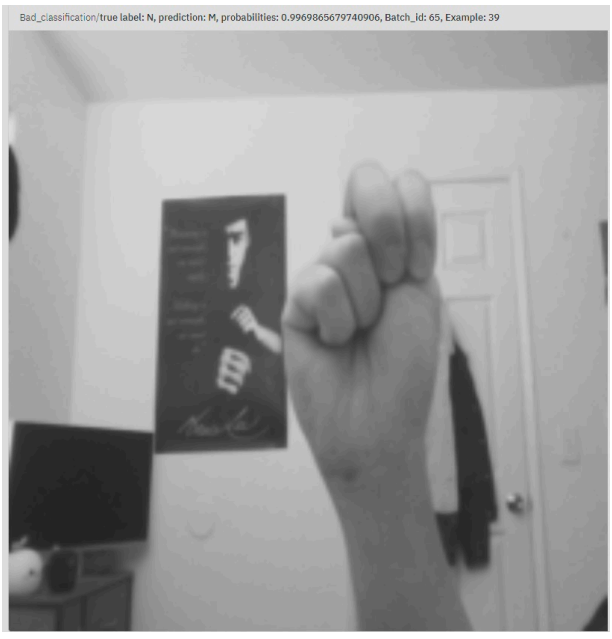
It can be seen in general that the model was able to classify the images well.
Now we can investigate more deeply how the model classified the images and at what confidence levels.

<u>Examples that the model was correct and classified with a high probability</u>
We can see that for the letters 'P' and the classification 'nothing' the model made the fewest mistakes, The example of the letter P was classified as P with a confidence level of 99.99% and the example of the sign 'nothing' was classified with a confidence level of 99.93%, and this is probably due to the fact that the letter ‘P’ has a very characteristic shape to it than the other letter representations, and the same can be said about the images that contain the sign 'nothing'.

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Examples where the model was wrong

We can see in the graph that the model repeated the same error the most times (25 mistakes) when it classified images representing the letter N(13) as an image representing the letter M(12). As we will see in the pictures, it is very easy to get confused between these two signs since the level of imagination between them is very high and it may be difficult for the human eye to distinguish between the letters. The level of confidence that the model predicted the letter N when in reality it was the letter M was 99.81% and the level of confidence that the model predicted the letter M when in reality it was the letter N was 99.69% for the following pictures.



Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Uncertain predictions

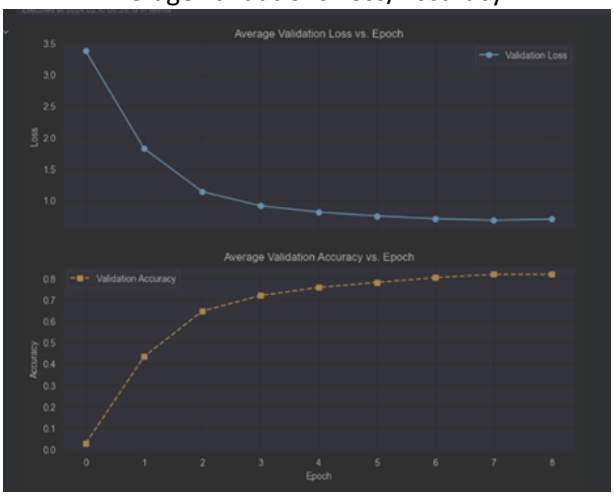
We set the threshold to be 0.4, each correct classification with a probability lower than this, the model put it in ‘the uncertain folder’.

After going through all the images in this folder, we can see that there are 2 symbols that repeat themselves the most times, 4 times for each symbol. These 2 signs are 'space' and the letter 'F'. In addition it was possible to see that all the images in this folder were either very close to looking like another sign, or the image quality was very poor.

The level of confidence that the model predict the letter S was 33.85%,

and the level of confidence that the model predict the letter was 38.36%

Average validations Loss/Accuracy



Average test Loss/Accuracy

11 rows | 11 rows × 3 columns

Fold	Test Accuracy	Test Loss
1	0.885432	0.512107
2	0.870397	0.640237
3	0.836144	0.827486
4	0.807889	0.774458
5	0.856036	0.638033
Average	0.85118	0.678464

The results of all the experiments are summarized in the table in Appendix 1.

Colab notebook:

https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

B.

Based on the graphs, it's evident that the model begins to converge as early as epochs 5-7. Consequently, following thorough research on the subject, our team has opted to enhance the network's complexity. This augmentation in complexity will be apparent in two out of the three improvement suggestions we propose.

1. We decided to add kernels of a larger size to the beginning of the network because some of the images in the data are part of a larger image, for example, a person with a hand or with a lot of background, while some of the images are a hand in the entire image (like the attached image).
   For these cases, we would also like to learn connections between the pixels that are in places further away from each other.
   That's why we added the conv11 and conv7 layers and added more kernels to each layer.

```
   | Name           | Type               | Params
-------------------------------------------------------
0  | conv_11        | Conv2d             | 2.0 K
1  | conv_7         | Conv2d             | 25.1 K
2  | conv1          | Conv2d             | 18.5 K
3  | conv2          | Conv2d             | 73.9 K
4  | conv3          | Conv2d             | 73.8 K
5  | conv4          | Conv2d             | 36.9 K
6  | linear1        | Linear             | 2.0 M
7  | linear2        | Linear             | 1.5 K
8  | mp             | MaxPool2d          | 0
9  | relu           | ReLU               | 0
10 | val_accuracy   | MulticlassAccuracy | 0
11 | test_accuracy  | MulticlassAccuracy | 0
12 | train_accuracy | MulticlassAccuracy | 0
-------------------------------------------------------
2.2 M     Trainable params
0         Non-trainable params
2.2 M     Total params
8.927     Total estimated model params size (MB)
```

```
x = self.relu(self.conv_11(x))
x = self.relu(self.conv_7(x))
x = self.mp(x)
x = self.relu(self.conv1(x))
x = self.relu(self.conv2(x))
x = self.mp(x)
x = self.relu(self.conv3(x))
x = self.relu(self.conv4(x))
x = self.mp(x)
x = x.view(-1,25*25*64)
x = self.relu(self.linear1(x))
x = self.linear2(x)
return F.log_softmax(x,dim=1)
```

```
self.conv_11 = nn.Conv2d(1,16,11, padding='same')
self.conv_7 = nn.Conv2d(16,32,7, padding='same')
self.conv1 = nn.Conv2d(32,64,3,padding='same')
self.conv2 = nn.Conv2d(64,128,3,padding='same')
self.conv3 = nn.Conv2d(128,64,3,padding='same')
self.conv4 = nn.Conv2d(64,64,3,padding='same')
self.linear1 = nn.Linear(25*25*64,50)
self.linear2 = nn.Linear(50,self.num_classes)
self.mp = nn.MaxPool2d(2,2)
self.relu=nn.ReLU()
```

2. The second enhancement, prompted by the swift convergence of the network, involved transitioning from treating the images solely as black and white, as initially done, to considering them in color. This entailed adjusting the input depth from 1 to 3, reflecting the red, green, and blue color channels. The rationale behind this modification is to leverage color information within the images, specifically to identify variations in skin tone and the color of the hand.
   The changes in the code:
   The first layer should get 3

```
self.conv1 = nn.Conv2d(3,32,3,padding='same')
```

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

The transformation before the change

```
self.transform = transforms.Compose(
    [
        transforms.Resize((200, 200)),
        transforms.Grayscale(num_output_channels=1),
        transforms.ToTensor(),

    ]
)
```

The transformation after the change and

```
self.transform = transforms.Compose(
    [
        transforms.Resize((200, 200)),

        transforms.ToTensor(),

    ]
```

3. The third enhancement we proposed involves employing ensemble learning techniques, wherein all k-fold models are aggregated, and each model's contribution is weighted based on its achieved accuracy level. This approach aims to harness the collective predictive power of multiple models, assigning greater influence to those that demonstrate higher accuracy. By amalgamating the insights from diverse models, we aim to enhance the overall performance and robustness of the system.

C

Following the first improvement, which involved adding kernels, the average accuracy for Kfold=5 increased from 85% in the unimproved model to 90.8% post-enhancement. Subsequent to this improvement, we suggest several further enhancements based on our observations:

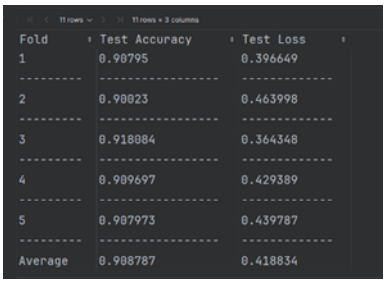
1. Addressing Overfitting: Analysis of loss and accuracy graphs indicates slight overfitting. To mitigate this, we recommend introducing a dropout layer to the model architecture.
2. Adjusting the learning rate schedule, such as using learning rate decay or adaptive learning rate methods like Adam or RMSprop, can help the model converge faster and more effectively.
3. Data Cleaning: Notably, some misclassifications occurred due to unclear or low-quality images in the dataset. We propose a data cleaning step to filter out such images, thereby improving the model's robustness and performance.

Regarding the second improvement, transitioning from grayscale to color images resulted in decreased accuracy, dropping to 72.1%. To address this, we suggest:

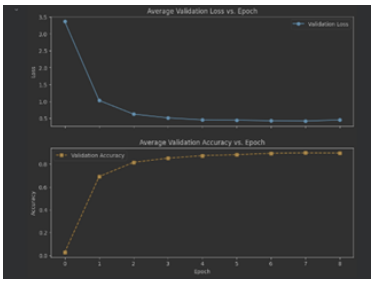
1. Normalization: Incorporating a normalization step in the transformation process for color images can enhance model performance. Despite not being initially specified, we implemented this change and observed significant improvement. With normalization, the average accuracy for all folds reached 90.8%.
2. Image Dimension Increase: Considering that some images in the dataset are larger in size, resizing them to a larger dimension could prevent information loss and potentially enhance model performance.
3. Hyperparameter Tuning: During the k-fold stage, exploring various hyperparameters can significantly impact model outcomes. Fine-tuning these parameters can lead to improved accuracy and generalization.

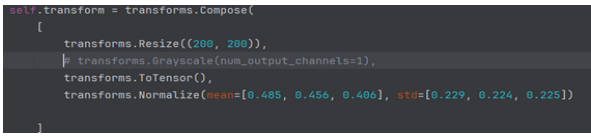
Incorporating these suggestions can further refine the model's performance and ensure its efficacy across diverse scenarios.

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing



Fold	Test Accuracy	Test Loss
1	0.90795	0.396649
2	0.90823	0.463998
3	0.918084	0.364348
4	0.909697	0.429389
5	0.907973	0.439787
Average	0.908787	0.418834





```
self.transform = transforms.Compose(
    [
        transforms.Resize((200, 200)),
        # transforms.Grayscale(num_output_channels=1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]
```

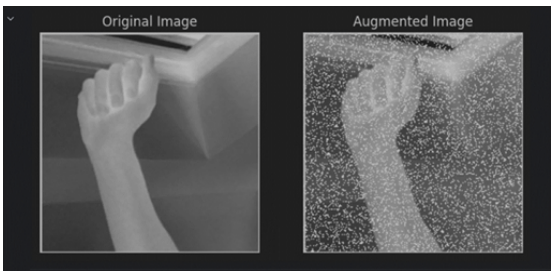
D

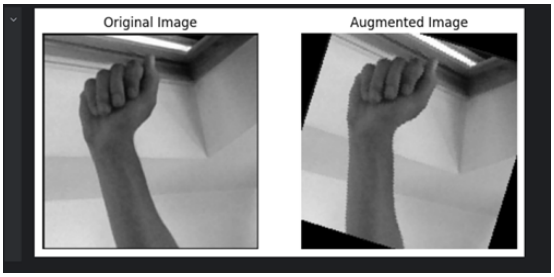
It appears that the attempts at inference-time augmentation did not yield the desired improvement in model accuracy. Despite efforts to augment the image data for training purposes, the accuracy of the model did not improve and, in fact, decreased to 55%.

The first augmentation involved creating new images by randomly turning off pixels with a probability of 20% of the pixels in the original image. The intention was to introduce variability and expand the training dataset.

The second augmentation entailed adding images by rotating the original images within a range of 20 degrees in each direction. This augmentation aimed to further diversify the dataset and enhance the model's ability to generalize across different orientations.

However, despite these augmentation techniques, the model's accuracy did not improve and experienced a decrease instead. This outcome suggests that the chosen augmentation strategies may not have been effective for this particular task or dataset. Further experimentation with different augmentation methods or adjustments to the augmentation parameters may be necessary to achieve improved results. Additionally, thorough analysis and understanding of the impact of each augmentation on the model's performance could provide insights for future refinement of the augmentation strategy.

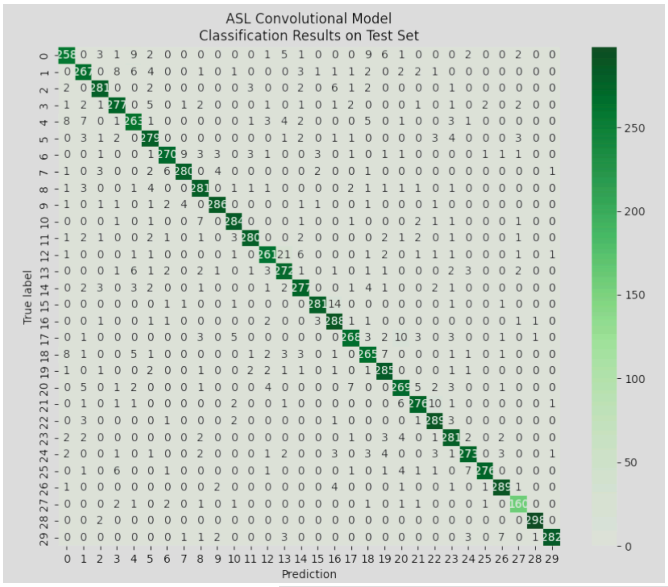


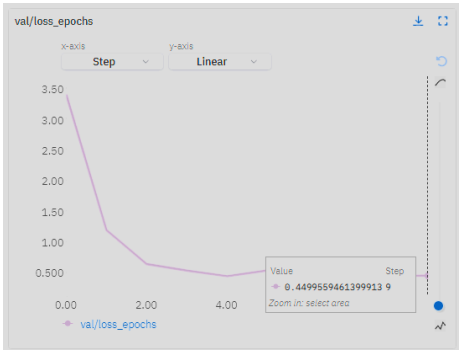


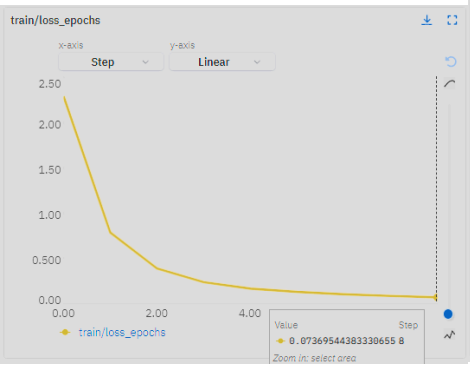
Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

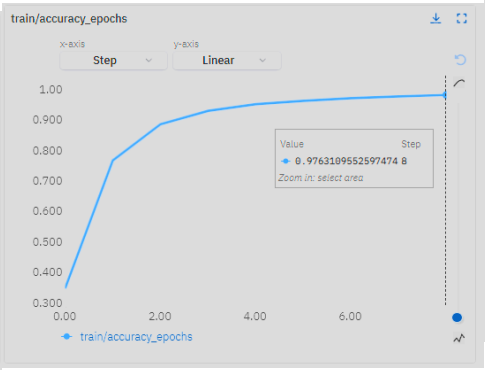
E. new category

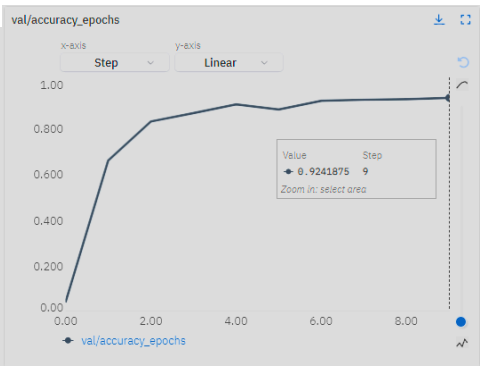
The new category we added to the model is called '5' and is represented by an open palm.











The results of all the experiments are summarized in the table in Appendix 1.

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Task 3

Transfer learning model

The results of all the experiments are summarized in the table in Appendix 1.

In this section, we applied transfer learning using pre-built and pre-trained models with ImageNet weights. The primary objective of transfer learning is to leverage knowledge gained from a large dataset and apply it to a smaller one. We froze the early convolutional layers of the network, training only the last layers responsible for predictions.

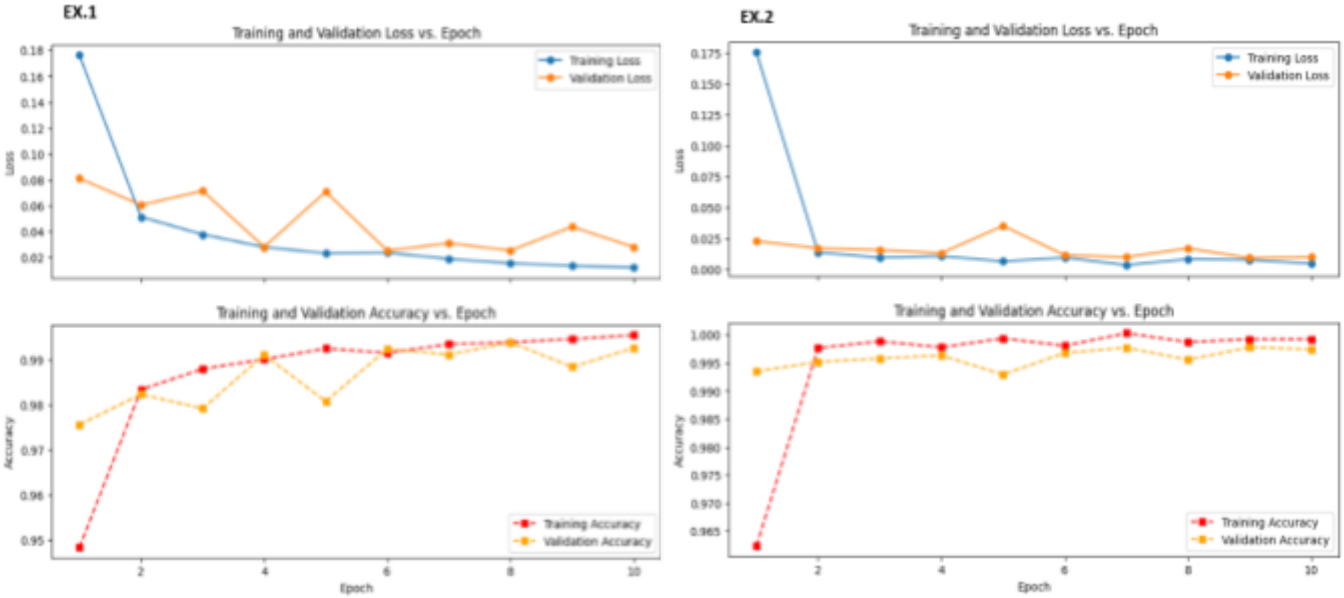
The concept is that the initial convolutional layers capture general, low-level features applicable across various images, such as edges, patterns, and gradients. The subsequent layers specialize in identifying specific features within individual images.

Our choice of pre-trained models includes ResNet18, AlexNet, DenseNet, and VGG16, with an image size of 224x224, a batch size of 64/128, 10 epochs, and no data augmentation.

Experiments on the ResNet18 model without layer freezing:

Initially, we ran two ResNet18 models (with varying learning rates), without freezing layers and achieving exceptional results with over 99% test accuracy.

The only difference between experiment 1 and experiment 2 is in the learning rate, in experiment 1 the learning rate = 0.001, and in experiment 2 the learning rate = 0.0001.



In analyzing the graph from Experiment 1, it is apparent that the validation curves (loss and accuracy) do not strictly follow a monotonic trend; however, the fluctuations are minimal.

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

On the other hand, examining the graph from Experiment 2 reveals that if we had employed the early stopping method, as implemented in Task 2, the computational time would have been reduced. This is because the graph converged within a few epochs, rendering the full ten epochs unnecessary.

Notably, Experiment 2, with a learning rate of 0.0001, exhibited better performance compared to Experiment 1.

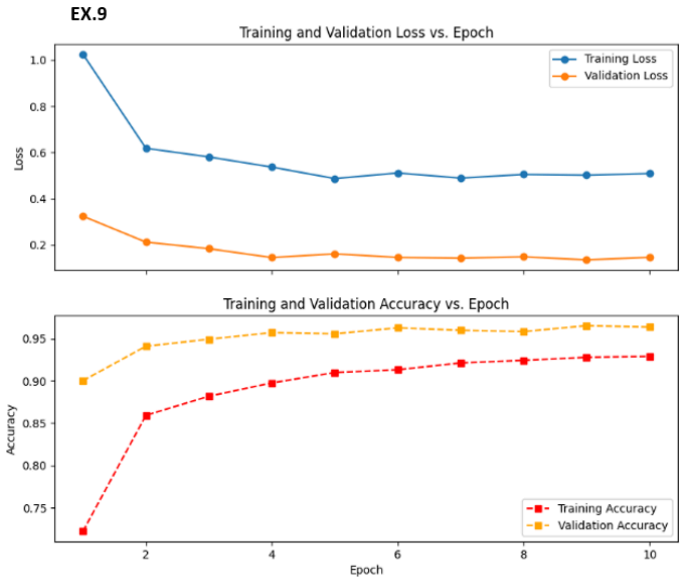
Experiments on the ResNet18 model with layer freezing:

After the experiments without freezing the layers, we conducted experiments by freezing layers except for the last ones. The ResNet18 model with freezing exhibited a slight drop to 84.49% in test accuracy. We explored hyperparameter adjustments, such as modifying the learning rate, batch sizes, and normalization with different std and mean values. However, an experiment with adjusted normalization resulted in overfitting.

Experiments on additional models:

Continuing the exploration, we ran AlexNet, DenseNet, and VGG16 models with a learning rate of 0.001, obtaining good results overall. However, these results did not surpass those of the experiments conducted without freezing layers (experiment 1 and 2).

In this series of experiments, we got a high result of 96.62% in test accuracy in experiment 9 with the VGG16 model.



Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Summary of experiments:

Despite the initial intention to avoid overfitting through freezing layers, the results without freezing layers consistently outperformed the experiments with frozen layers. This phenomenon suggests that the transfer learning models, particularly ResNet18 in the mentioned experiments, benefited more from fine-tuning across the entire network rather than isolating the last layers.

While the initial assumption was that freezing layers might prevent overfitting by leveraging pre-trained knowledge, the models exhibited superior generalization and performance when allowed to adapt across all layers.

Additionally, Freezing layers in a neural network during training typically saves runtime because the frozen layers don't require gradient computations, and their parameters are not updated during backpropagation. This can lead to faster training times since a significant portion of the model is essentially fixed.

However, in our experiments, there were no noticeable changes in the running times with and without freezing the layers.

The extent of the runtime improvement depends on several factors: the size of the Model, Computational Resources, dataset Size, and more.

Classic ML model using Feature extraction

We chose to use experiment number 2 of the previous section (Q3c.Ex10 at the Excel file), the experiment involved utilizing a pre-trained ResNet18 model for image classification. The model was fine-tuned on the dataset, achieving impressive accuracy rates of 99.98% on the training set and 99.86% on the test set. This experiment served as a baseline for comparison.

For this task, the last fully connected layer of the ResNet18 model was removed, and the preceding layer's output was used as features. These features were then fed into a linear regression model for classification. Surprisingly, the linear regression model achieved similar performance to the fine-tuned deep learning model, with training and test accuracies of 99.99% and 99.86%, respectively.

This approach potentially offers advantages such as reduced computational complexity and faster runtime, as it bypasses the need for end-to-end training of a deep learning model. The results suggest that leveraging the learned representations from a pre-trained deep learning model can be as effective as fine-tuning the model directly, demonstrating the versatility and power of transfer learning techniques in practical applications.

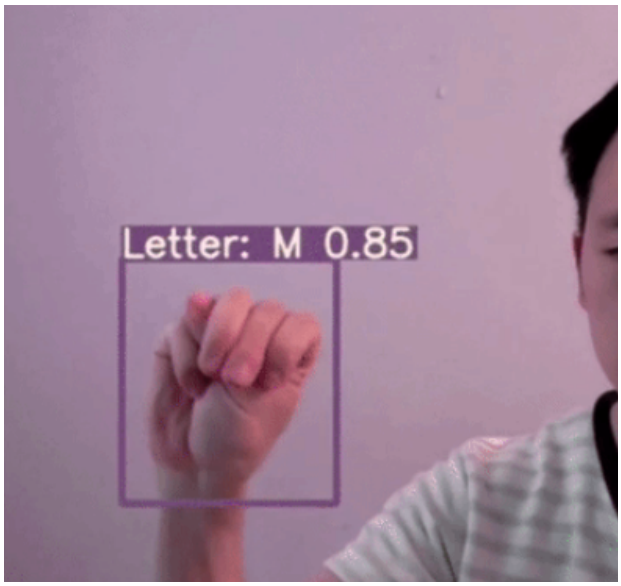
Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Conclusion and future research

The results we got at work are high and at a good level of accuracy.

As a follow-up study, we would like to connect the model to the live camera.

As we can see in this project:



https://blog.roboflow.com/computer-vision-american-sign-language/

References

- Early epoch stopping- https://lightning.ai/docs/pytorch/stable/common/early_stopping.html
- Transforming and augmenting images - https://pytorch.org/vision/stable/transforms.html
- K-fold with pytorch - https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-pytorch.md
- Layers freezing - https://stackoverflow.com/questions/62523912/freeze-certain-layers-of-an-existing-model-in-pytorch
- Classic ML model using Feature extraction - https://ai.plainenglish.io/4-classical-feature-extraction-techniques-you-must-try-7ea0969e9c23

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing

Appendix number 1

Link to the file:

https://docs.google.com/spreadsheets/d/1gh8KEMbEMfPbqFBcAM4vGatWReXFZGhB/edit?usp=sharing&ouid=106898911457943927491&rtpof=true&sd=true

		description	train loss	train acc	val loss	val acc	test loss	test acc	augmantation	image size	epochs	batch size	learning rate	freezing layers	Unique correct samples	unique err	runtime (h)
Q2.a	EX.1	K-fold, fold_1	0.163912	94.80%	0.553302	88.47%	0.512107	88.54%	FALASE	200X200	7	64	0.0002	FALASE			0:18:51
Q2.a	EX.2	K-fold, fold_2	0.187663	93.89%	0.644928	87.12%	0.640237	87.04%	FALASE	200X200	14	64	0.0002	FALASE			0:35:48
Q2.a	EX.3	K-fold, fold_3	0.265142	91.42%	0.826987	83.80%	0.827486	83.61%	FALASE	200X200	17	64	0.0002	FALASE			0:42:18
Q2.a	EX.4	K-fold, fold_4	0.37379	87.95%	0.749666	81.27%	0.774458	80.79%	FALASE	200X200	11	64	0.0002	FALASE			0:29:04
Q2.a	EX.5	K-fold, fold_5	0.266715	91.31%	0.650156	85.02%	0.638033	85.60%	FALASE	200X200	10	64	0.0002	FALASE			0:26:41
Q2.c	EX.6	Add layers	0.10271	96.74%	0.465807	90.85%	0.460563	90.99%	FALASE	200X200	6	64	0.0002	FALASE			0:18:26
Q2.c	EX.7	Colors without normalization	0.662101	78.12%	0.982542	72.27%	0.991513	72.18%	FALASE	200X200	11	64	0.0002	FALASE			0:30:25
Q2.c	EX.8	Colors with normalization	0.219595	93.05%	0.522804	88.33%	0.531416	87.92%	FALASE	200X200	13	64	0.0002	FALASE			0:41:16
Adding few images of a new category																	
Q2.e	EX.9	K-fold, fold_1	0.0868562	97.36%	0.477845	91.30%	0.442121	91.40%	FALASE	200X200	6	64	0.0002	FALASE			0:20:12
Q2.e	EX.10	Add layers	0.0736954	97.63%	0.449956	92.42%	0.448235	92.40%	FALASE	200X200	8	64	0.0002	FALASE			0:24:58
Q2.e	EX.11	Colors with normalization	0.133857	95.65%	0.393095	90.86%	0.396649	90.80%	FALASE	200X200	8	64	0.0002	FALASE			0:25:44
Classinc machine learning modle																	
Q.3.c	EX.12	ResNet18	0.0125	99.55%	0.02816	99.26%	0.0184	99.43%	FALASE	224X224	10	64	0.001	FALASE	29	18	1:02:42
Q.3.c	EX.13	ResNet18	0.0015	99.98%	0.0093	99.77%	0.00778	99.86%	FALASE	224X224	10	64	0.0001	FALASE	29	8	1:07:03
Q.3.c	EX.14	ResNet18	0.7501	83.34%	0.67248	84.13%	0.671035	84.49%	FALASE	224X224	10	64	0.0001	TRUE	29	29	1:08:00
Q.3.c	EX.15	ResNet18	0.3551	90.54%	0.3217	91.00%	0.3251	91.14%	FALASE	224X224	10	64	0.001	TRUE	29	29	1:09:52
Q.3.c	EX.16	ResNet18	0.4002	89.85%	0.3819	89.94%	0.38365	90.08%	FALASE	224X224	10	64	0.0005	TRUE	29	29	1:05:31
Q.3.c	EX.17	ResNet18	0.3875	97.74%	0.3354	90.77%	0.33742	90.67%	FALASE	224X224	10	64	0.001	TRUE	29	29	1:07:14
Q.3.c	EX.18	AlexNet	0.4011	87.51%	0.13785	95.84%	0.13506	95.92%	FALASE	224X224	10	64	0.001	TRUE	29	29	1:04:25
Q.3.c	EX.19	DenseNet121	0.2984	92.75%	0.25496	92.75%	0.259734	92.73%	FALASE	224X224	10	64	0.001	TRUE	29	28	1:11:15
Q.3.c	EX.20	VGG16	0.5587	90.12%	0.14708	96.36%	0.1398713	96.62%	FALASE	224X224	10	64	0.001	TRUE	29	27	1:26:02
Q.3.c	EX.21	AlexNet	0.41237	85.91%	0.15347	95.42%	0.15636	95.32%	FALASE	224X224	10	128	0.001	TRUE	29	28	0:49:39
Classic ML model using Feature extraction																	
Q.3.d	EX.22	ResNet18		99.99%				99.99%	FALASE	224X224	10	64	0.001	TRUE			

Colab notebook:
https://colab.research.google.com/drive/1izmt-ZoBJmSfDgd7pFmqh9jcEFuahzlK?usp=sharing