**Department of Information Engineering**

# MANIPAL UNIVERSITY JAIPUR

Machine Learning LAB
(DS2231) Lab
File
(JAN – MAY 2022)

**Submitted To:**

Mr. Rahul Saxena
Assistant Professor
IT Department
SCIT, MUJ

**Submitted By:**

Student Name: Pranjal Paira
Registration No.: 209309016
Semester: 4th /Year:2nd
Section: A

Student Name: Aakriti Abhay Singh
Registration No.: 209309087
Semester: 4th /Year:2nd
Section: A

# Problem Statement

## Introduction: -

We present a novel implementation of the infamous Rock-Paper-Scissors (RPS)game interaction with the computer. The framework is Tailored to be Computationally lightweight, as well as entertaining. The application is designed in a way that you can decide go for the best of 5 scores.

The basis of any deep learning model is DATA. Any Machine Learning Enthusiast would agree that in ML the data is far more crucial than the algorithm itself. We need to collect images for the symbol Rock, Paper, Scissor. instead of downloading somebody else's data and training on it, we made our dataset.

We have used the concept of Transfer Learning

Here is a breakdown of our application in steps.

STEP1: Gather Data, for Rock, Paper, and Scissor classes.

STEP2: Visualize the data.

STEP3: Pre-process data and split it.

STEP4: Prepare our model for Transfer Learning

STEP5: Train Our Model.

STEP6: Check for Accuracy, Loss graph and save the model.

STEP7: Test on Live Webcam Feed

STEP8: Create the final application

## Motivation: -

The basis of this project was to experiment with deep learning and image classification to build a simple yet fun iteration of the infamous rock paper scissors game. We have Fine-tuned the NASNETMobile model to recognize the hand signs when it is inside the box, so when the model predicts our hand sign, the AI randomly generates its move, and the winner is decided. Thus, we were able to successfully implement and understand this project.

In practice, for most machine learning problems, we would not design or train a CNN from scratch – we can use the idea of transfer learning to use an existing model that suits our needs and fine-tune it on our data.

# Dataset

## Description: -

The data collected in this project is all manual. The gather data function collects images of our hands. The function is designed in a way that it takes **num_samples** as an argument and then records the many numbers of images for each class.
The images in the ROI box of size *224×224,* are directly saved as this is the size model accepts and image resizing would not be needed.

The data collected by the user is divided into four classes:

1) Nothing
2) Paper
3) Rock
4) Scissors

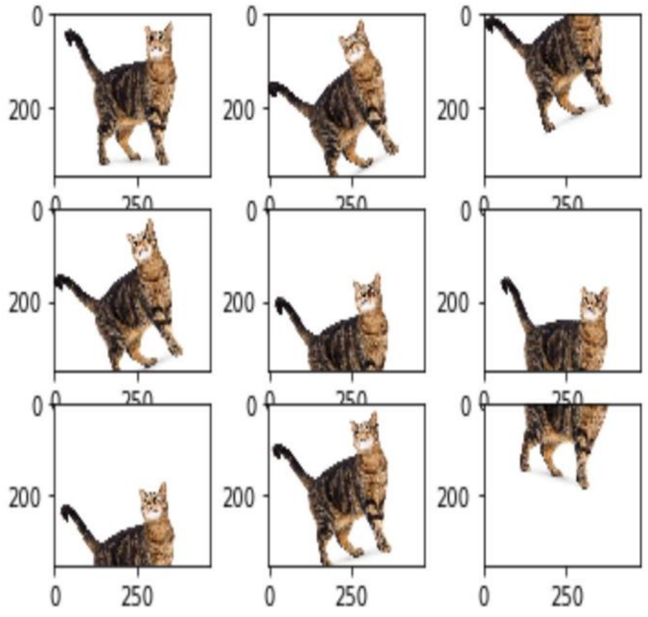
Each class is then labelled with their respective names i.e., Nothing, Rock, Paper and Scissor.

Nothing class is also included here because when the user is not playing a move and the hand is out of the box even then the model would try to predict something so the model is trained on an empty frame so that the model learns that when there is no hand present in the box it is called nothing.

## Summary: -

Summary of the Data: -

1) The number of samples used for each class here is 100, which can be altered based on the computational power the user has.
2) The Total number of images including all the classes is 400, each image is having a label associated with it. These are labelled while recording individual classes, e.g., to record samples of **rock** press ‘r’, to record samples of **nothing** press ‘n’ and so on.
3) Each frame in the ROI is then appended to a list with the selected class name.
4) The images are in the format of a NumPy array i.e., Images are converted into NumPy Array in Height, Width, Channel format.
5) The images are purposefully saved in memory and not on disk. This is because it allows the user to train faster by getting rid of I/O latency caused by loading images on a disk.
6) In the pre-processing phase, the data is normalised by dividing it by 255 so that all the images range between 0-1.
7) The labels are then encoded using Label encoder i.e., nothing = 0, paper=1, rock = 2, scissor = 3 (mapping is done in alphabetical order)
8) Before training the model, the data is augmented. Image augmentation is a technique that is used **to artificially expand the data set**. This is helpful when we are given a data set with very few data samples. In the case of Deep Learning, this situation is bad as the model tends to over-fit when we train it on a limited number of data samples.

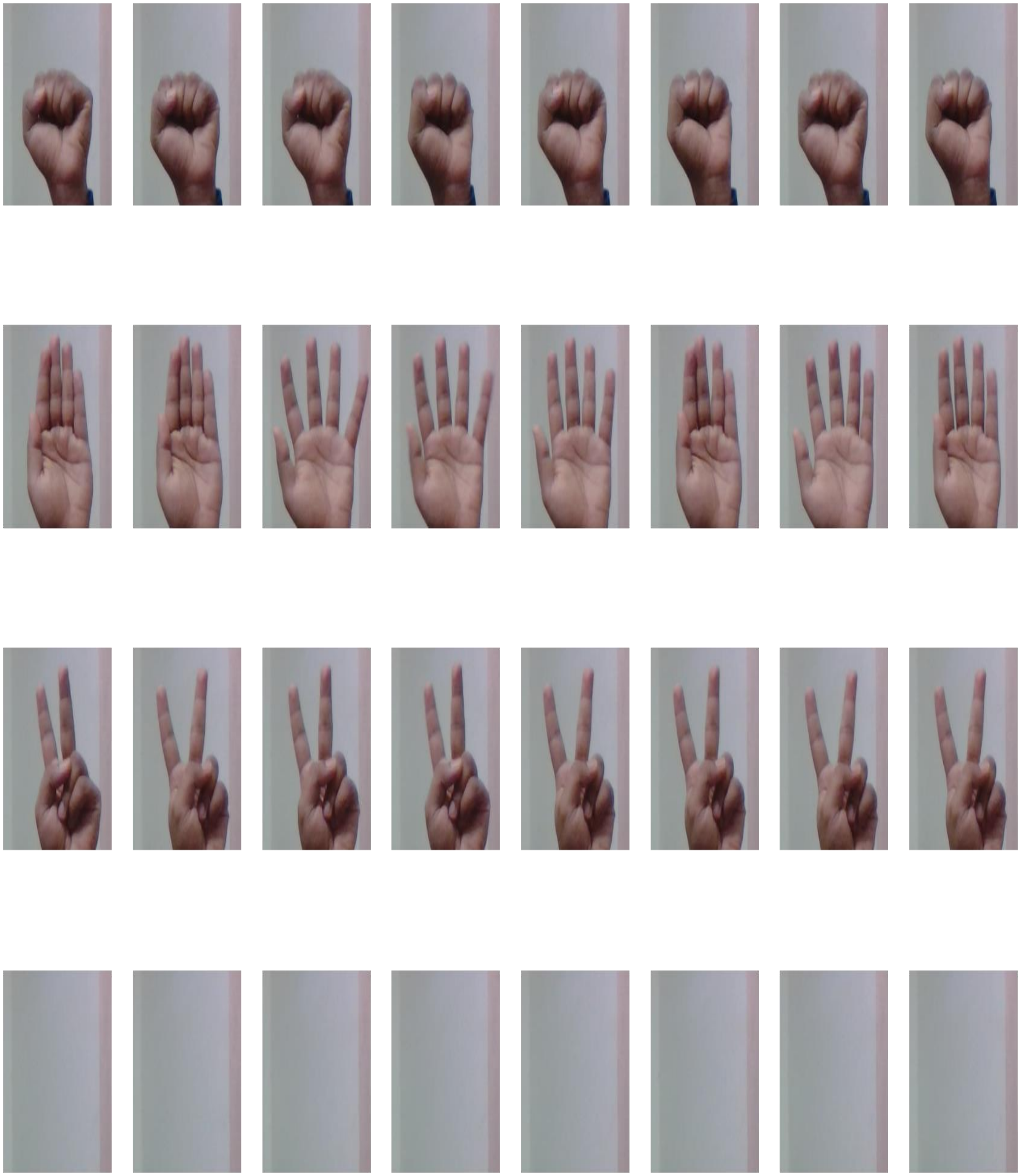


## Class Balance: -

A balanced dataset is a dataset where each output class (or target class) is represented by the same number of input samples.

Here each class has 100 samples.

The Classes in our dataset extend from 0 to 3.

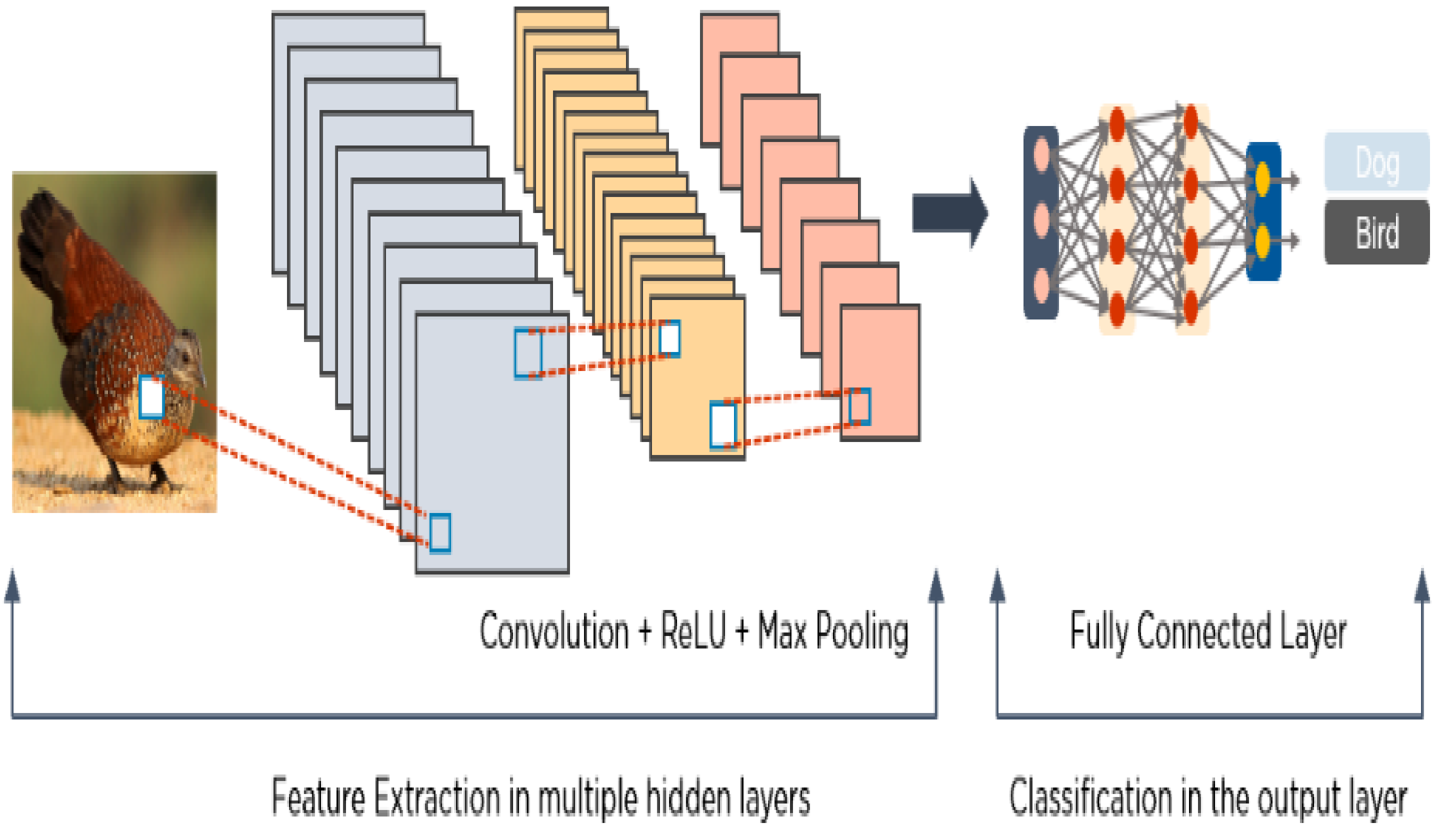


# Algorithms Used

## Neural Networks

### Convolutional Neural Networks

A convolutional neural network (CNN) is a type of artificial neural network used primarily for image recognition and processing, due to its ability to recognize patterns in images. A CNN is a powerful tool but requires millions of labelled data points for training. CNNs must be trained with high-power processors, such as a GPU or an NPU if they are to produce results quickly enough to be useful.


Dog
Bird
Convolution + ReLU + Max Pooling
Fully Connected Layer
Feature Extraction in multiple hidden layers
Classification in the output layer

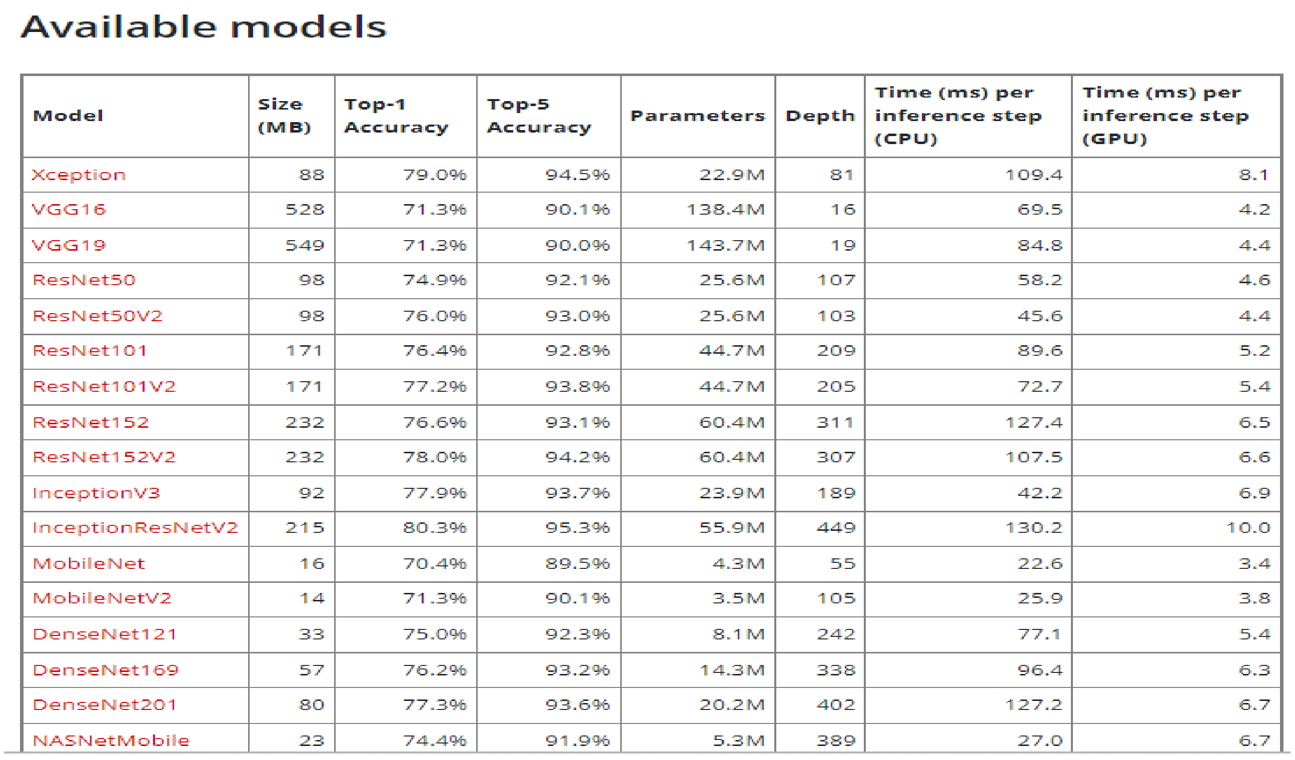
The Model Used in this project: -

## NASNet-Mobile

NASNet-Mobile (**Neural Search Architecture** (NAS) Network) is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224.

The model requirement here was something with the best balance of Speed and Accuracy and low computational power. Finally, NASNETMobile was chosen.

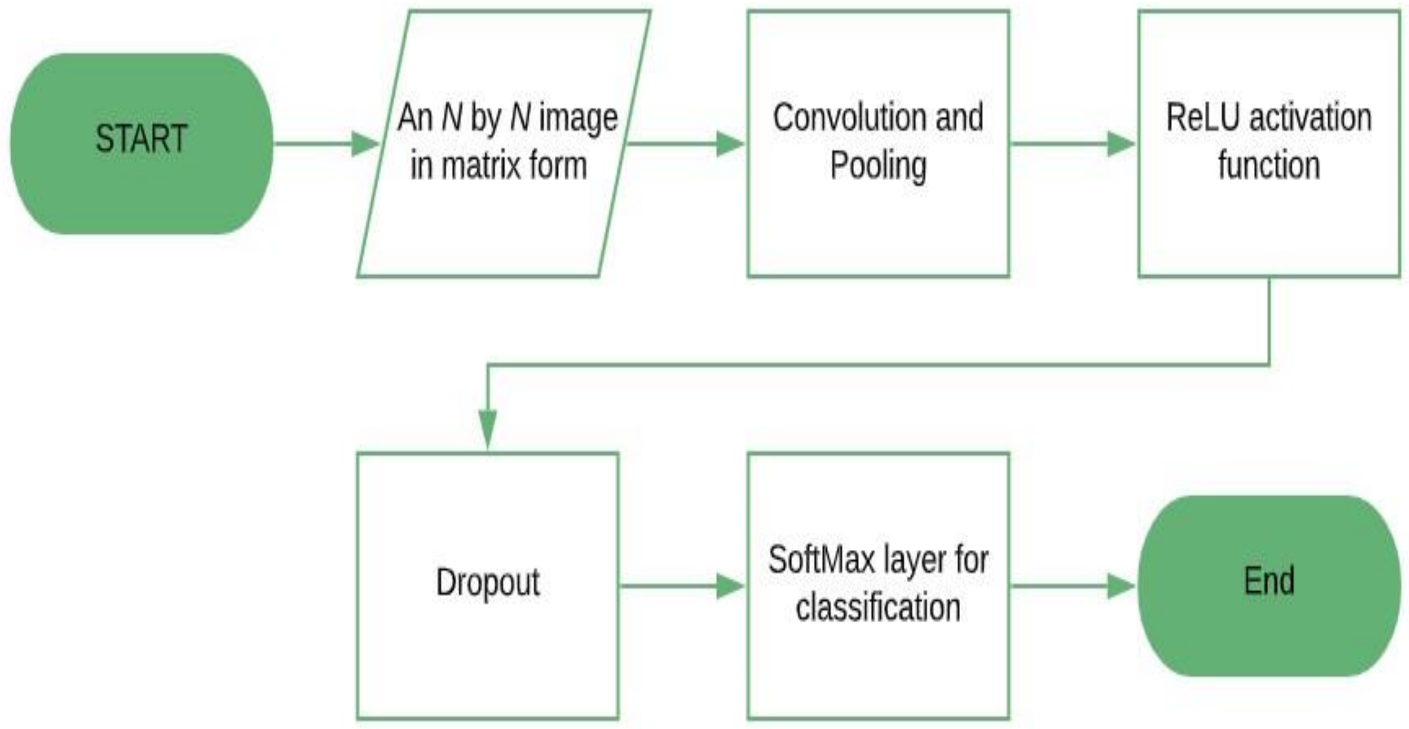
The NASNETMobile is loaded up without the head because it was trained on 1000 ImageNet classes and here, we have to predict just 4 classes, so we don't need the head of the model. This saves time and computational power.



Available models

| Model | Size (MB) | Top-1 Accuracy | Top-5 Accuracy | Parameters | Depth | Time (ms) per inference step (CPU) | Time (ms) per inference step (GPU) |
|---|---|---|---|---|---|---|---|
| Xception | 88 | 79.0% | 94.5% | 22.9M | 81 | 109.4 | 8.1 |
| VGG16 | 528 | 71.3% | 90.1% | 138.4M | 16 | 69.5 | 4.2 |
| VGG19 | 549 | 71.3% | 90.0% | 143.7M | 19 | 84.8 | 4.4 |
| ResNet50 | 98 | 74.9% | 92.1% | 25.6M | 107 | 58.2 | 4.6 |
| ResNet50V2 | 98 | 76.0% | 93.0% | 25.6M | 103 | 45.6 | 4.4 |
| ResNet101 | 171 | 76.4% | 92.8% | 44.7M | 209 | 89.6 | 5.2 |
| ResNet101V2 | 171 | 77.2% | 93.8% | 44.7M | 205 | 72.7 | 5.4 |
| ResNet152 | 232 | 76.6% | 93.1% | 60.4M | 311 | 127.4 | 6.5 |
| ResNet152V2 | 232 | 78.0% | 94.2% | 60.4M | 307 | 107.5 | 6.6 |
| InceptionV3 | 92 | 77.9% | 93.7% | 23.9M | 189 | 42.2 | 6.9 |
| InceptionResNetV2 | 215 | 80.3% | 95.3% | 55.9M | 449 | 130.2 | 10.0 |
| MobileNet | 16 | 70.4% | 89.5% | 4.3M | 55 | 22.6 | 3.4 |
| MobileNetV2 | 14 | 71.3% | 90.1% | 3.5M | 105 | 25.9 | 3.8 |
| DenseNet121 | 33 | 75.0% | 92.3% | 8.1M | 242 | 77.1 | 5.4 |
| DenseNet169 | 57 | 76.2% | 93.2% | 14.3M | 338 | 96.4 | 6.3 |
| DenseNet201 | 80 | 77.3% | 93.6% | 20.2M | 402 | 127.2 | 6.7 |
| NASNetMobile | 23 | 74.4% | 91.9% | 5.3M | 389 | 27.0 | 6.7 |

## The Architecture: -


START
An *N* by *N* image in matrix form
Convolution and Pooling
ReLU activation function
Dropout
SoftMax layer for classification
End

## Number of Layers in the Model: -

1) Convolutional Layer
2) Pooling Layer
3) ReLu (Rectified Linear Unit) Dense Layer
4) Dropout Layer
5) SoftMax Layer

## Transfer learning

It makes use of the knowledge gained while solving one problem and applying it to a different but related problem.

The key concept behind transfer learning in data science is **deep learning models**. They require A lot of data, which, if your model is also supervised, means that you need a lot of labelled data. As everyone involved in a machine learning project knows, **labelling data samples is very tedious and time-consuming.**

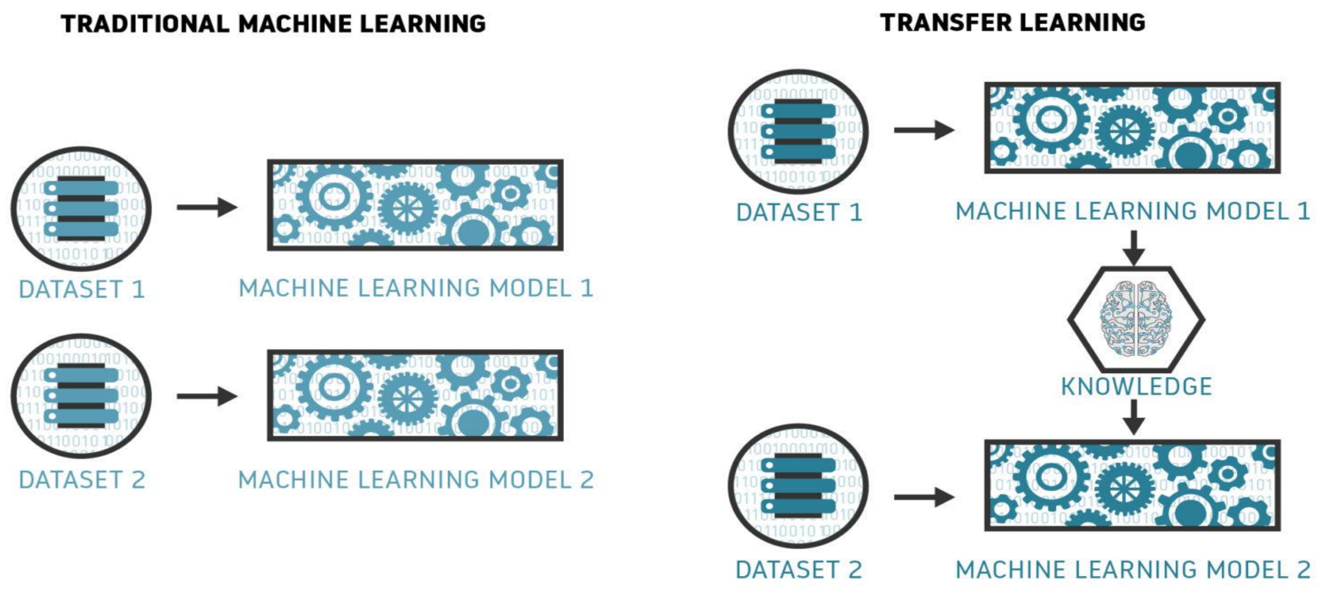
For example, knowledge gained while learning to recognize cars can be used to some extent to recognize trucks.

### Pre-Training

When we train the network on a **large dataset (for example ImageNet)**, we train all the parameters of the neural network and therefore the model is learned. It may take hours on your GPU.

### Fine Tuning

We can give the new dataset to fine-tune the pre-trained CNN. Consider that the new dataset is like the original dataset used for pre-training. Since the new dataset is similar, the same weights can be used for extracting the features from the new dataset.



# Results And Analysis

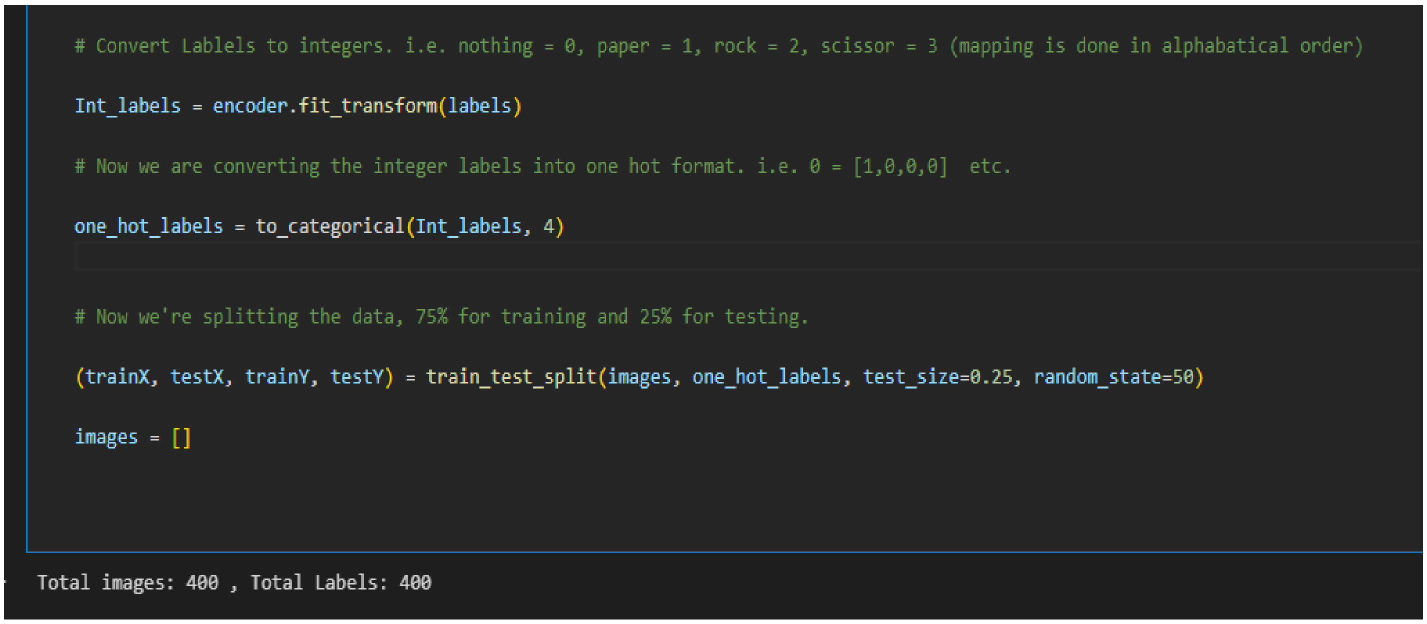
## 1.) Data Pre-processing: -

- **Combining all the classes:** All the four classes are combined in this part. It was not done before because if the user wants to record samples of a single class again, he can do that by just pressing that class's button in the data collection script. If all images were combined initially then a change in a single class would have required recollecting samples of all other classes.

- **Normalizing Data:** Normalize the images by dividing by 255, now our images are in the range 0-1. This ensures, that the numbers will be small, and the computation becomes easier and faster

- **One Hot Encoding of Data:** We are converting the categorical labels to integer and then integer labels into one hot format. i.e., 0 = [1,0,0,0] etc.

**Finally, we have 400 labels and 400 Images**



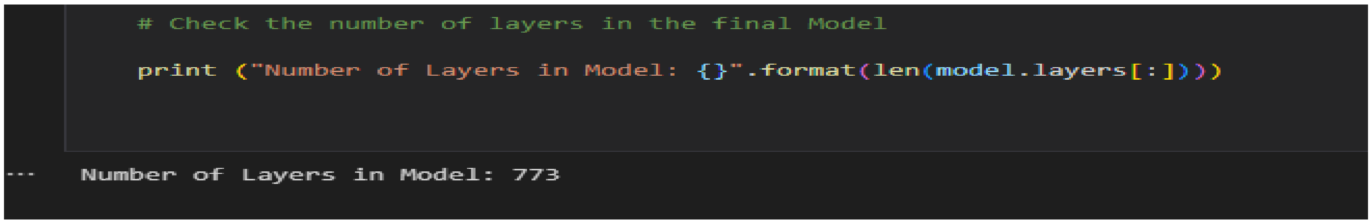
```
# Convert Lablels to integers. i.e. nothing = 0, paper = 1, rock = 2, scissor = 3 (mapping is done in alphabatical order)

Int_labels = encoder.fit_transform(labels)

# Now we are converting the integer labels into one hot format. i.e. 0 = [1,0,0,0]  etc.

one_hot_labels = to_categorical(Int_labels, 4)


# Now we're splitting the data, 75% for training and 25% for testing.

(trainX, testX, trainY, testY) = train_test_split(images, one_hot_labels, test_size=0.25, random_state=50)

images = []
```

```
Total images: 400 , Total Labels: 400
```

## 2.) Transfer Learning: -

- **The NASNETMobile is loaded up without the head:** It is because it was trained on 1000 ImageNet classes and here, we have to predict just 4 classes, so we don't need the head of the model. This saves time and computational power.
- **The image is passed from Convolutional Layer:** A convolutional layer is the main building block of a CNN. It contains a set of filters (or kernels), parameters of which are to be learned throughout the training. The size of the filters is usually smaller than the actual image. Each filter convolves with the image and creates an activation map.
- **The next is the Pooling Layer:** Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn, and the amount of computation performed in the network. The pooling layer summarises the features present in a region of the feature map generated by a convolution layer.
- **The next Layer is the ReLU dense Layer:** We use a combination of random weights and rectified linear unit (ReLU) activation function to add a ReLU dense (ReDense) layer to the trained neural network such that it can achieve a lower training loss. It ranges from 0 to infinity.
- **The next Layer is the Dropout Layer:** The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting.
- **The final Layer is our SoftMax Layer:** SoftMax is used as the activation for the last layer of a classification network because the result could be interpreted as a probability distribution. It ranges from 0 to 1. The final layer will contain the number of nodes equal to the number of classes which in our case is 4.

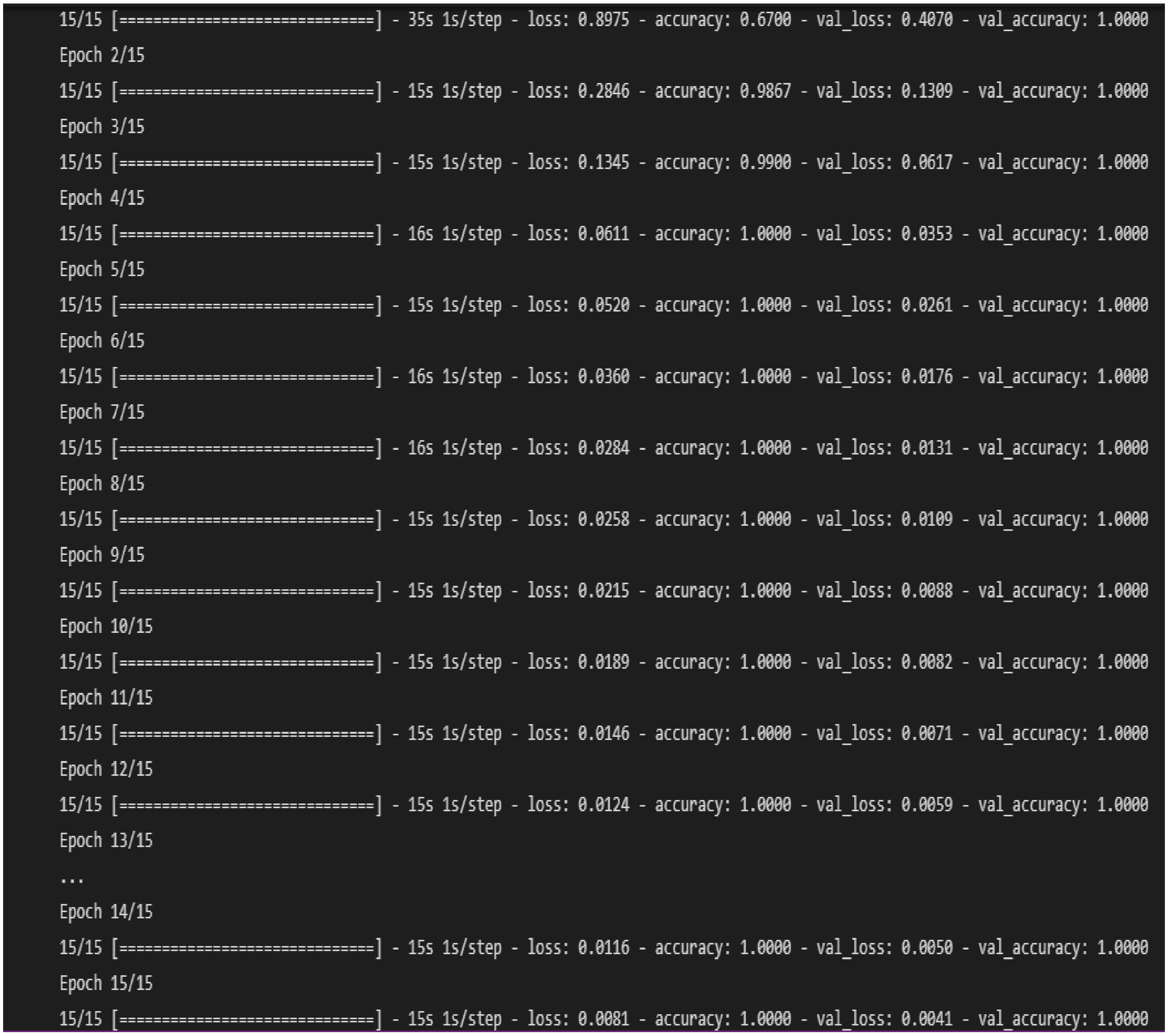
**Finally, we have 773 Layers in our model.**



```
# Check the number of layers in the final Model

print ("Number of Layers in Model: {}".format(len(model.layers[:])))

...  Number of Layers in Model: 773
```

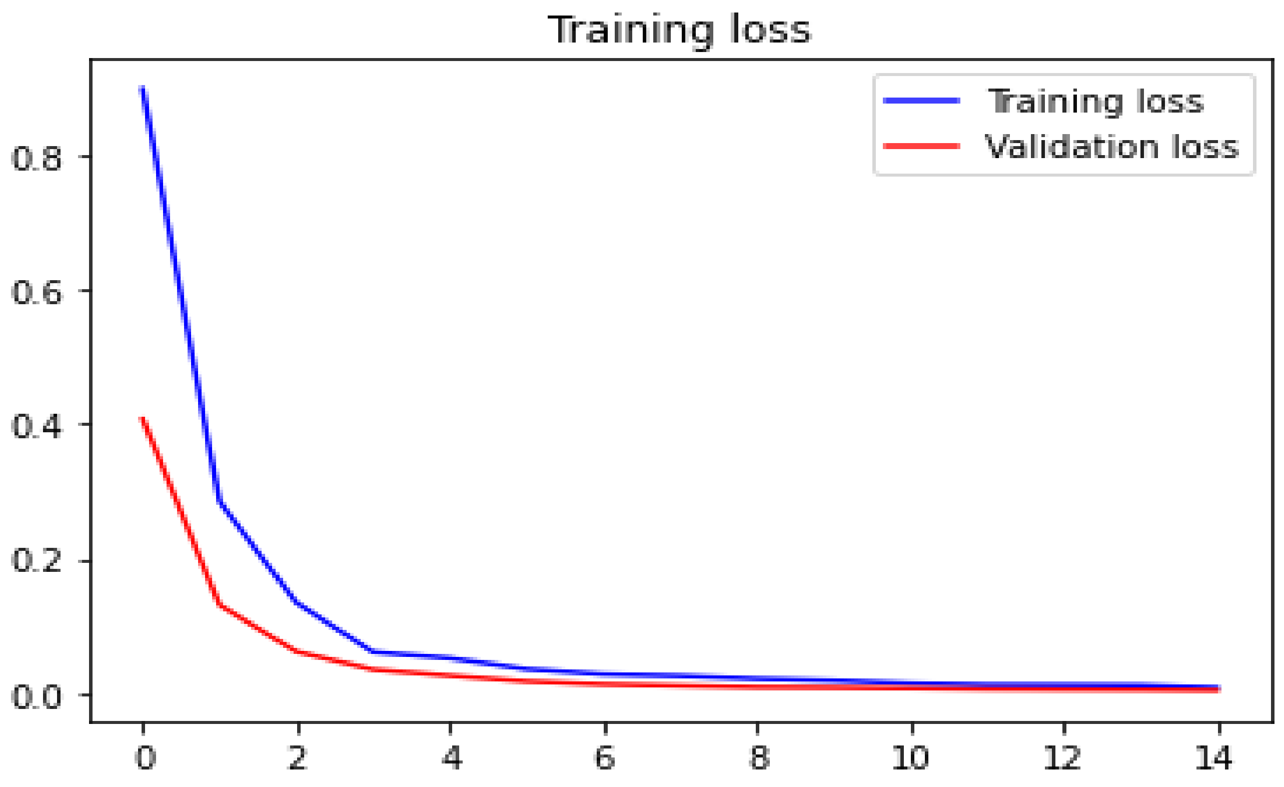
## 3.) Training Our Model: -

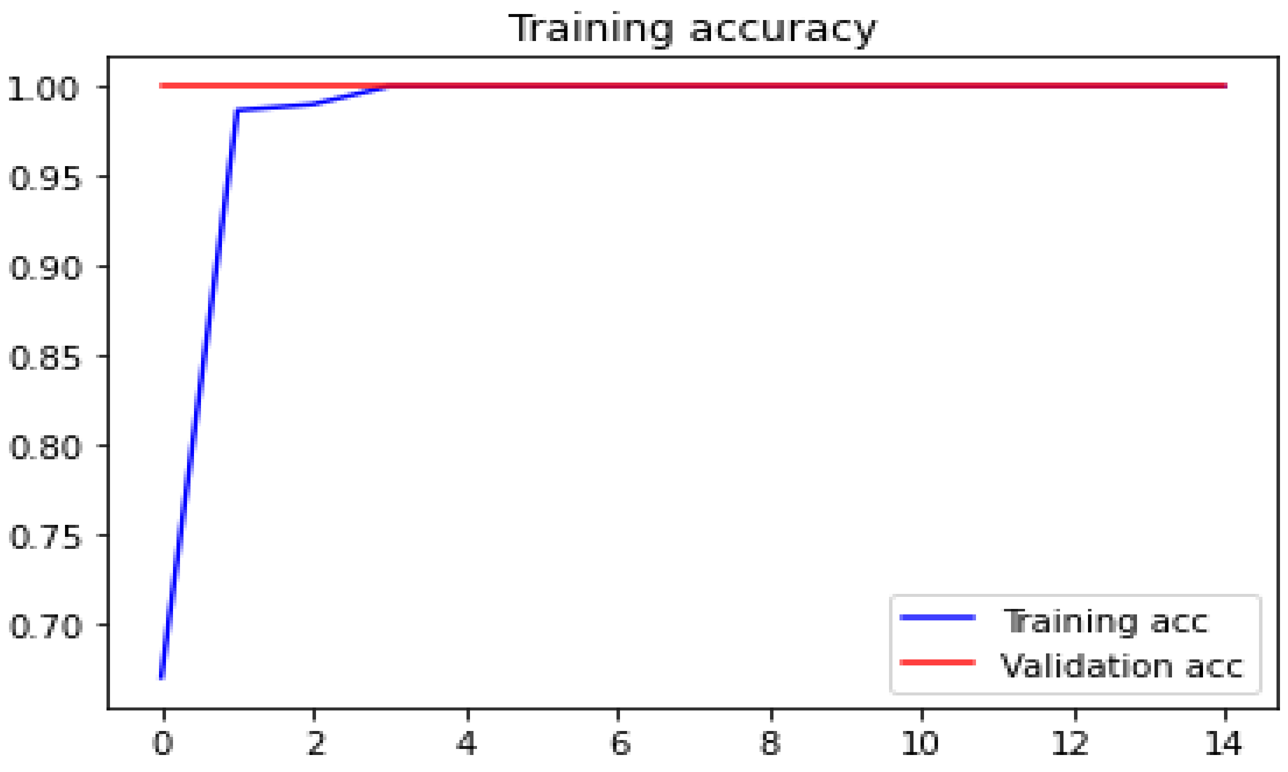
With Epoch=15 and batch size=20 we start training our model.



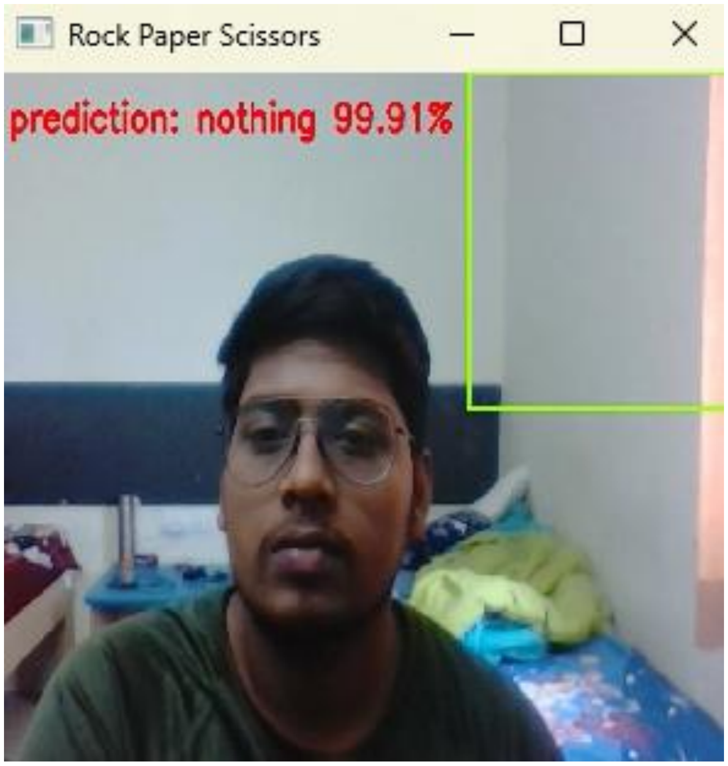
```
15/15 [==============================] - 35s 1s/step - loss: 0.8975 - accuracy: 0.6700 - val_loss: 0.4070 - val_accuracy: 1.0000
Epoch 2/15
15/15 [==============================] - 15s 1s/step - loss: 0.2846 - accuracy: 0.9867 - val_loss: 0.1309 - val_accuracy: 1.0000
Epoch 3/15
15/15 [==============================] - 15s 1s/step - loss: 0.1345 - accuracy: 0.9900 - val_loss: 0.0617 - val_accuracy: 1.0000
Epoch 4/15
15/15 [==============================] - 16s 1s/step - loss: 0.0611 - accuracy: 1.0000 - val_loss: 0.0353 - val_accuracy: 1.0000
Epoch 5/15
15/15 [==============================] - 15s 1s/step - loss: 0.0520 - accuracy: 1.0000 - val_loss: 0.0261 - val_accuracy: 1.0000
Epoch 6/15
15/15 [==============================] - 16s 1s/step - loss: 0.0360 - accuracy: 1.0000 - val_loss: 0.0176 - val_accuracy: 1.0000
Epoch 7/15
15/15 [==============================] - 16s 1s/step - loss: 0.0284 - accuracy: 1.0000 - val_loss: 0.0131 - val_accuracy: 1.0000
Epoch 8/15
15/15 [==============================] - 15s 1s/step - loss: 0.0258 - accuracy: 1.0000 - val_loss: 0.0109 - val_accuracy: 1.0000
Epoch 9/15
15/15 [==============================] - 15s 1s/step - loss: 0.0215 - accuracy: 1.0000 - val_loss: 0.0088 - val_accuracy: 1.0000
Epoch 10/15
15/15 [==============================] - 15s 1s/step - loss: 0.0189 - accuracy: 1.0000 - val_loss: 0.0082 - val_accuracy: 1.0000
Epoch 11/15
15/15 [==============================] - 15s 1s/step - loss: 0.0146 - accuracy: 1.0000 - val_loss: 0.0071 - val_accuracy: 1.0000
Epoch 12/15
15/15 [==============================] - 15s 1s/step - loss: 0.0124 - accuracy: 1.0000 - val_loss: 0.0059 - val_accuracy: 1.0000
Epoch 13/15
...
Epoch 14/15
15/15 [==============================] - 15s 1s/step - loss: 0.0116 - accuracy: 1.0000 - val_loss: 0.0050 - val_accuracy: 1.0000
Epoch 15/15
15/15 [==============================] - 15s 1s/step - loss: 0.0081 - accuracy: 1.0000 - val_loss: 0.0041 - val_accuracy: 1.0000
```

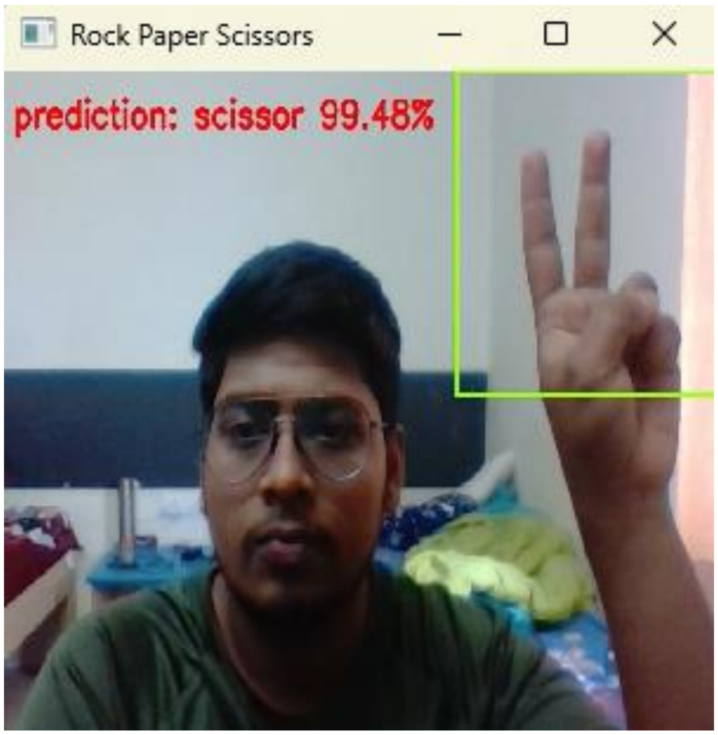
**The Results After 15 Epochs of Training Our Model: -**

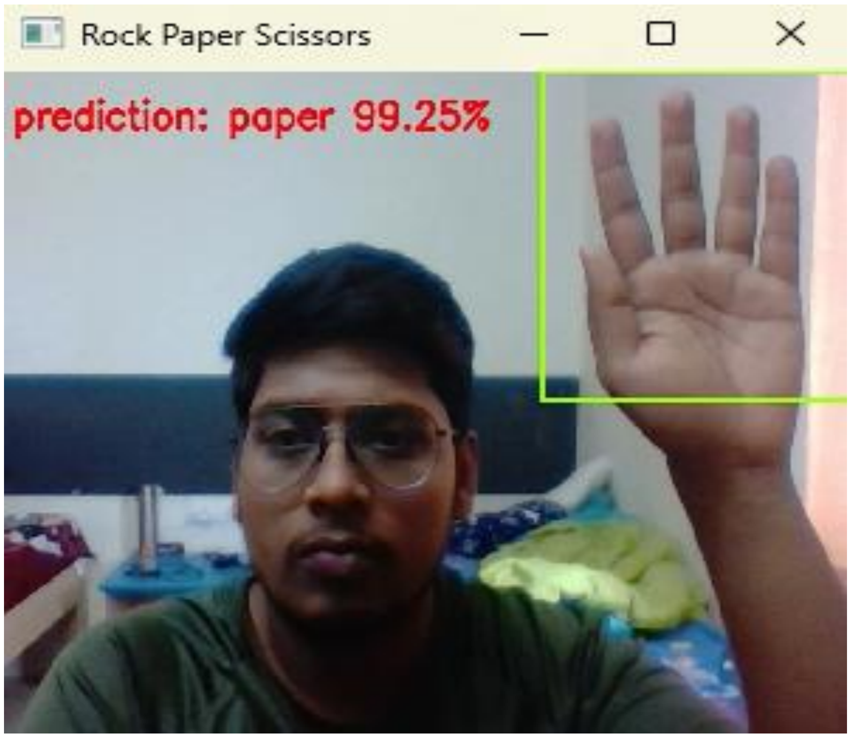


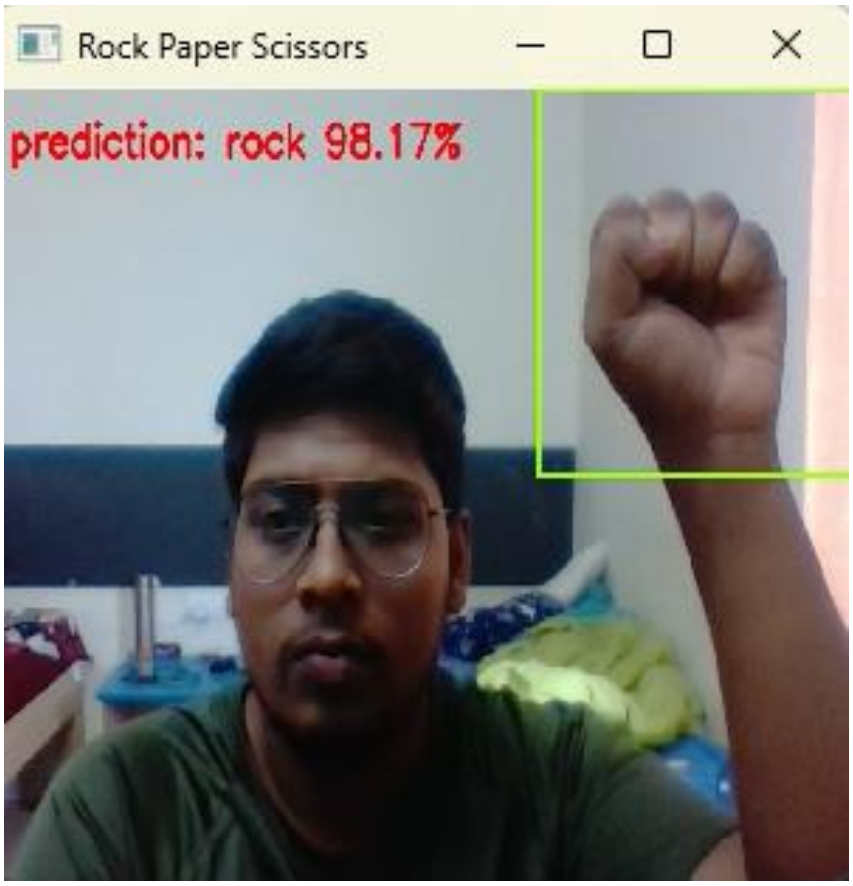


**Live Feed Test Results: -** Few test results for the live webcam feed.

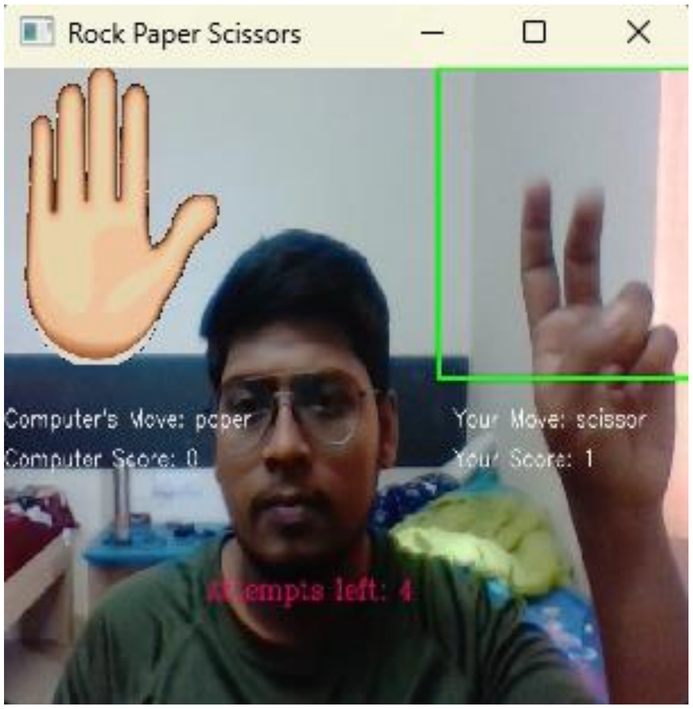




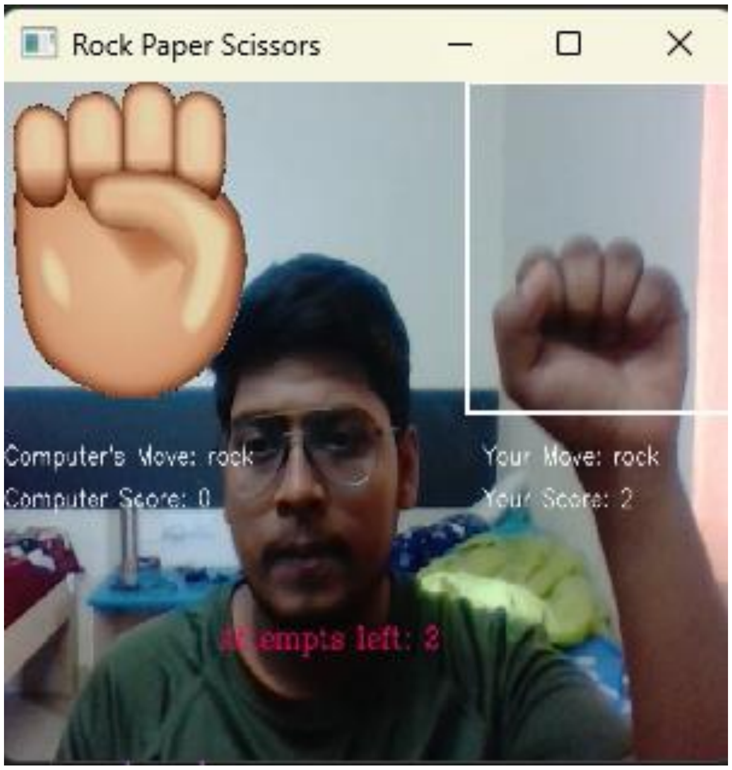

Rock Paper Scissors
prediction: paper 99.25%


Rock Paper Scissors
prediction: rock 98.17%

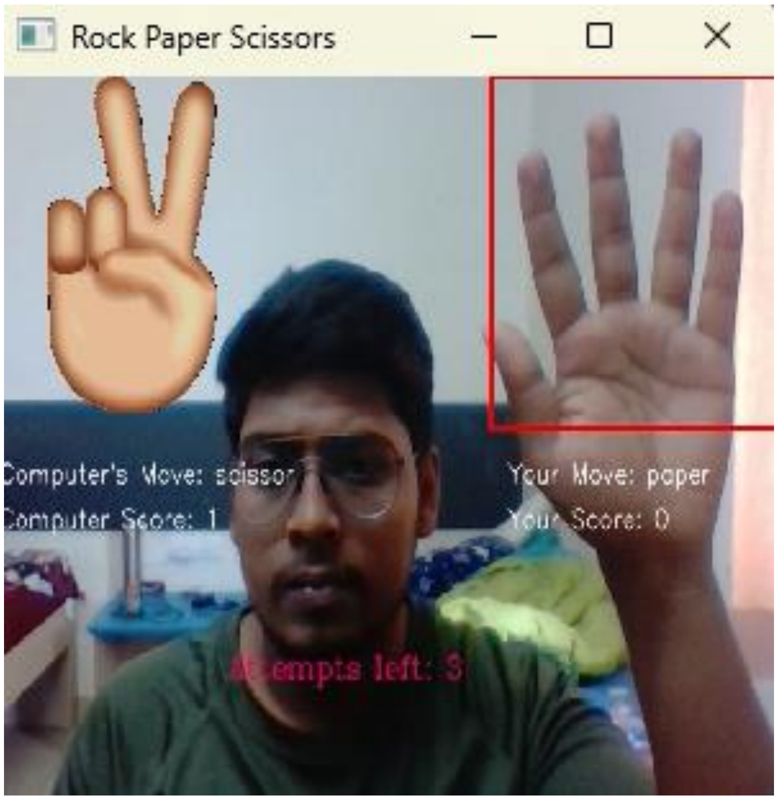
**Some stills from the Final Application: -**



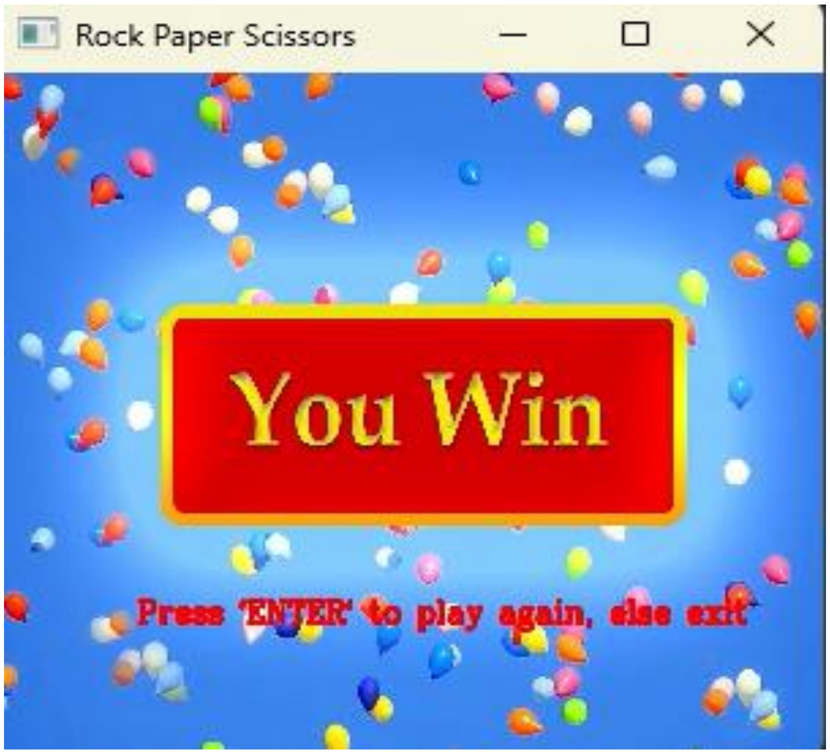
Computer's Move: paper
Computer Score: 0
Your Move: scissor
Your Score: 1
Attempts left: 4



Computer's Move: rock
Computer Score: 0
Your Move: rock
Your Score: 2
Attempts left: 2



Computer's Move: scissor
Computer Score: 1
Your Move: paper
Your Score: 0
Attempts left: 3



You Win

Press 'ENTER' to play again, else exit

## Conclusion

To conclude we look at a quick summary of how we built R-P-S Game.
It is crucial to understand that RPS has no random Component.
The outcome is entirely determined by the decision of the players. It is worth noting that we’re purposefully saving the images in memory and not on disk. This is because it will allow us to train faster by getting rid of I/O latency caused by loading images on disk.
From here we identified and converted the labels into vector using one hot format and normalized the data. Later we implemented Transfer learning and Data Augmentation to train the model. We incorporated NASNET mobile to provide better accuracy and computational power. Finally, the model was examined and tested with live test feed for good results. Lastly, no doubt Deep learning will give us the best results for this game.
Thus, our RPS game using CNN is complete.

## References

- For TensorFlow:
  https://www.tensorflow.org/

- For Keras API:
  https://keras.io/api/applications/

- Towards Data Science:
  https://towardsdatascience.com

- For NumPy functions:
  https://numpy.org/doc/stable/numpy-user.pdf

For Open-Cv:
https://opencv.org