Car vs Bike vs Random Images Multi Class Classification Comparison Report Based on Different Metric Using Convolutional Neural Network and Artificial Neural Network in Python Keras Framework

By

Shuvam Sanyal

PRN-15070243015

Semester -III

M.sc Data Science and Spatial Analytics(2019-2021)

1.Introduction

Classifying images is a complex problem in the field of computer vision. The deep learning algorithm is a computerized model simulates the human brain functions and operations. Training the deep learning model is a costly process in machine resources and time. Investigating the performance of the deep learning algorithm is mostly needed. The convolutional neural network (CNN) is most commonly used to build a structure of the deep learning models. Artificial Neural Network comes next to CNN.

In the deep learning algorithm, the object feature extracted engineering is done by the algorithm automatically. Engineering the object feature extraction is a difficult process and timeconsuming. Operations of the feature extraction process required a domain expert operator for design and testing. The deep learning algorithm solves wide problems of classification tasks. The ability to process large clusters of images quickly, state the deep learning algorithm as the most important method for images classification. The ability and flexibility for changeable in the deep learning model with a wide range of data-sets make the deep learning algorithm as the most important technique for the classification task. The most difficult problem in the deep learning algorithm is the cost of hardware and consuming time in training processes. Training the deep learning model may need weeks with costly GPU machine. Performing practical research focused on the performance of the deep learning algorithm is the most important to evaluate the deep learning method. This project points out the success of the advancement of the deep learning algorithms from ANN to gradually CNN in solving of multi-class image classification task problem.Here we have used a set of car and bike images of 25 each and 50 random images and made different multi class deep learning models using Artifical Neural Network and Convolutional Neural Networks(My own CNN Architecture, AlexNet,VGG16 and Lenet-5) to classify these images for car, bike or random category and then compared the multi class classification performances of both the models using different metrics like precision, recall, F1 Score, Confusion Matrix. Our result Shows overall CNN performs much better than ANN in multi-class image classification. Further analysis shows that out of 4 CNN architectures we used, best peformings are Lenet-5 and my own created architecture(CNN-8), while AlexNet and VGG16 perfomances are not so good due to low size of the input image dataset.

2.Methodology:-

2.1 ***Collection of Image Dataset:-***

The image dataset is collected from internet google image. The dataset consists of 25 jpg format images of cars, 25 jpg format images of bikes and 50 random jpg format images of different sizes. The path for the image dataset is given here :- Google Drive Image Link.

2.2 ***Data Pre-Processing: -***

In the data pre-processing phase I first imported all the images from google drive to my google colab notebook. Then I resized those pictures of different sizes to uniform size of 128*128 in RGB format. Then I labelled those images as 0 for car, 1 for bike and 2 for random. The size of our dataset is very small compared to capabilities of deep learning models. Post that we converted those images to NumPy arrays of pixel values between 0 to 255 and re-labelled those images within range [0,1] by using label binaizer class of Python.

2.3 ***Image Data Augmentation:-***

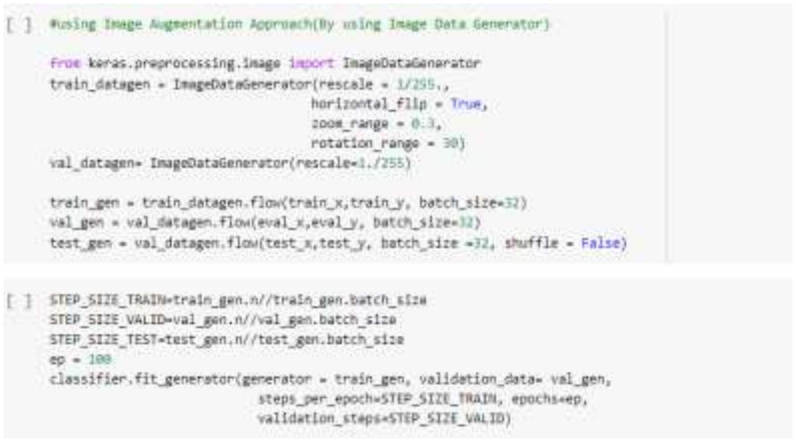
Image data augmentation is perhaps the most well-known type of data augmentation and involves creating transformed versions of images in the training dataset that belong to the same class as the original image. Transforms include a range of operations from the field of image manipulation, such as shifts, flips, zooms, and much more.

The intent is to expand the training dataset with new, plausible examples. This means, variations of the training set image that are likely to be seen by the model. For example, a horizontal flip of a picture of a cat may make sense, because the photo could have been taken from the left or right. A vertical flip of the photo of a cat does not make sense and would probably not be appropriate given that the model is very unlikely to see a photo of an upside-down cat.

Modern deep learning algorithms, such as the convolutional neural network, or CNN, can learn features that are invariant to their location in the image. Nevertheless, augmentation can further aid in this transform invariant approach to learning and can aid the model in learning features that are also invariant to transforms such as left-to-right to top-to-bottom ordering, light levels in photographs, and more.

Image data augmentation is typically only applied to the training dataset, and not to the validation or test dataset. This is different from data preparation such as image resizing and pixel scaling; they must be performed consistently across all datasets that interact with the model. The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the *ImageDataGenerator* class.

We applied data augmentation on our own CNN and ANN architecture.



```
[ ]  #using Image Augmentation Approach(By using Image Data Generator)

     from keras.preprocessing.image import ImageDataGenerator
     train_datagen = ImageDataGenerator(rescale = 1/255.,
                                        horizontal_flip = True,
                                        zoom_range = 0.3,
                                        rotation_range = 30)
     val_datagen= ImageDataGenerator(rescale=1./255)

     train_gen = train_datagen.flow(train_x,train_y, batch_size=32)
     val_gen = val_datagen.flow(eval_x,eval_y, batch_size=32)
     test_gen = val_datagen.flow(test_x,test_y, batch_size =32, shuffle = False)
```

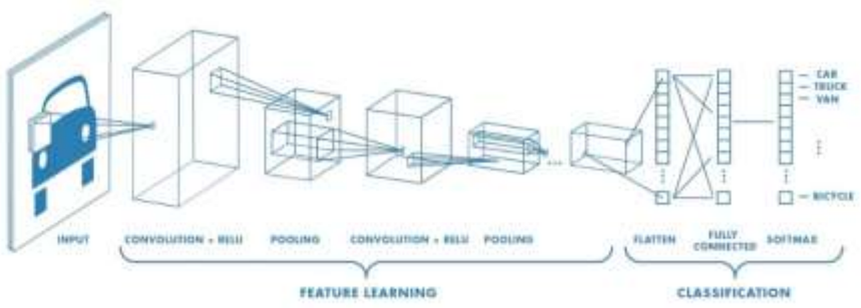
```
[ ]  STEP_SIZE_TRAIN=train_gen.n//train_gen.batch_size
     STEP_SIZE_VALID=val_gen.n//val_gen.batch_size
     STEP_SIZE_TEST=test_gen.n//test_gen.batch_size
     ep = 100
     classifier.fit_generator(generator = train_gen, validation_data= val_gen,
                              steps_per_epoch=STEP_SIZE_TRAIN, epochs=ep,
                              validation_steps=STEP_SIZE_VALID)
```

2.4 <u>Development & Implementation of Deep Learning Models</u>:-

<u>Convolutional Neural Network:-</u>

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

CNN image classifications take an input image, process it and classify it under certain categories (Eg.,car, bike, random images etc). Computers sees an input image as array of pixels and it depends on the image resolution. Based on the image resolution, it will see h x w x d(h = Height, w = Width, d = Dimension). A basic convolutional neural network looks like below:-



Technically, deep learning CNN models to train and test, each input image will pass it through a series of convolution layers with filters (Kernals), Pooling, fully connected layers (FC) and apply Softmax function to classify an object with probabilistic values between 0 and 1.Different building blocks of CNN are:-

<u>Convolution Layer</u>

Convolution is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.

Strides

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so o

Padding

Sometimes filter does not fit perfectly fit the input image. We have two options:

- Pad the picture with zeros (zero-padding) so that it fits(Same Padding).
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.(Valid Padding).

In our model, we have used the same padding mostly.

Non-Linearity (ReLU)

- ReLU stands for Rectified Linear Unit for a non-linear operation. The output is ***f(x) = max(0,x).***

- ReLU's purpose is to introduce non-linearity in our ConvNet. Since, the real-world data would want our ConvNet to learn would be non-negative linear values.

Pooling Layer

Pooling layers section would reduce the number of parameters when the images are too large. Spatial pooling also called subsampling or down sampling which reduces the dimensionality of each map but retains important information. Spatial pooling can be of different types:

- Max Pooling
- Average Pooling
- Sum Pooling

Max pooling takes the largest element from the rectified feature map. Taking the largest element could also take the average pooling. Sum of all elements in the feature map call as sum pooling. In our model, we have used max pooling mostly.

Fully Connected Layer

The layer we call as FC layer, we flattened our matrix into vector and feed it into a fully connected layer like a neural network.

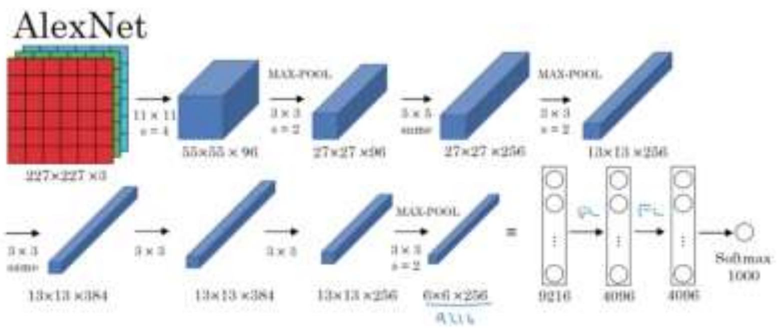
There are various architectures of CNNs available which have been key in building algorithms which power and shall power AI as a whole in the foreseeable future. Some of them have been listed below:

- LeNet
- AlexNet
- VGGNet
- GoogLeNet
- ResNet

We have used four CNN Architectures namely Alexnet,VGG16 and LeNet-5 and my own architecture.

<u>AlexNet</u>

It starts with 224 x 224 x 3 images and the next convolution layer applies 96 of 11 x 11 filter with stride of 4. The output volume reduces its dimension by 55 x 55. Next layer is a pooling layer which applies max pool by 3 x 3 filter along with stride 2. It goes on and finally reaches next two FC layers with 4096 node each. At the end, it uses Softmax function with 1000 output classes. It has 60 million parameters.

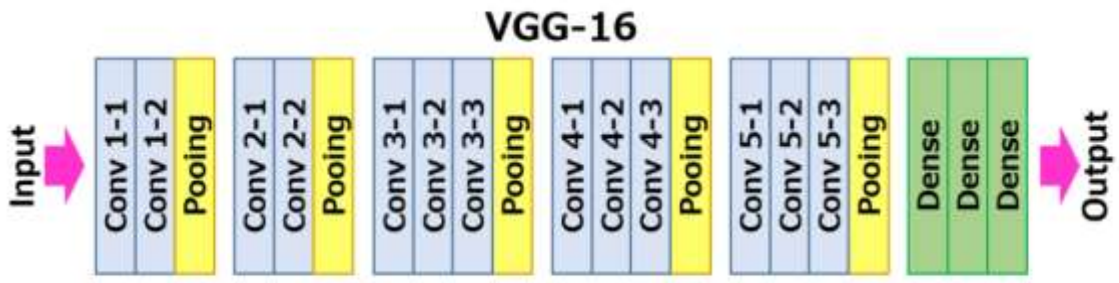


- It uses ReLU activation function instead Sigmoid or Tanh functions.

- It uses "Dropout" instead of regularisation to deal with overfitting. But the training time is doubled with dropout ratio of 0.5

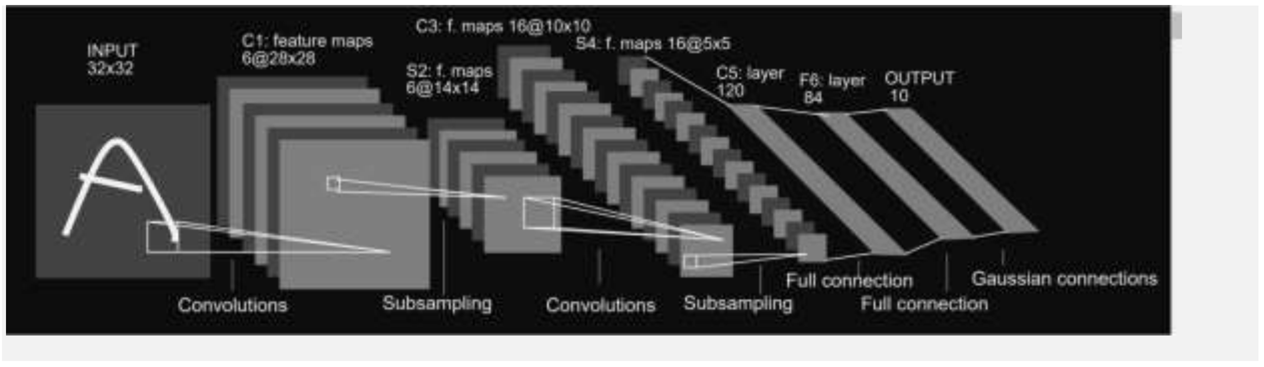
VGG-16

VGG-16 is a simpler architecture model, since it's not using much hyper parameters. It always uses 3 x 3 filters with stride of 1 in convolution layer and uses SAME padding in pooling layers 2 x 2 with stride of 2.



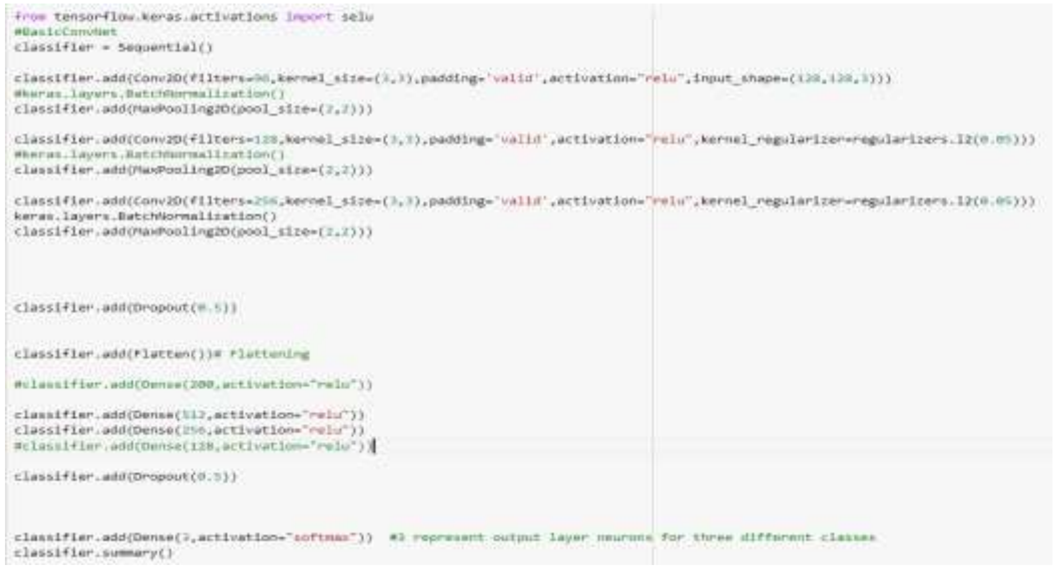
LeNet-5

LeNet-5 CNN architecture is made up of 7 layers. The layer composition consists of 3 convolutional layers, 2 subsampling layers and 2 fully connected layers.



My own architecture(CNN-8):-

In my own CNN architecture , I have used total 8 layers. The layer composition consists of 3 convolutional layers, 3 sub sampling layers and 2 fully connected layers.



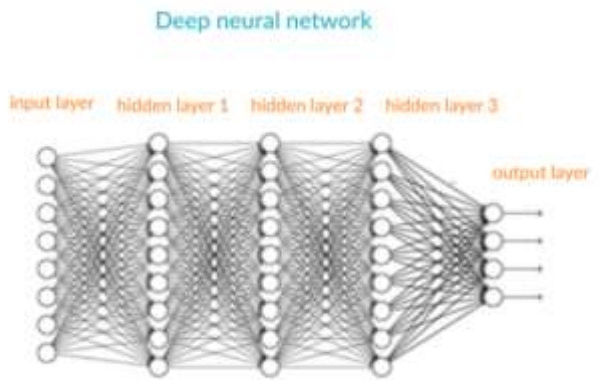
```
from tensorflow.keras.activations import selu
#BasicConvNet
classifier = Sequential()

classifier.add(Conv2D(filters=96,kernel_size=(3,3),padding='valid',activation="relu",input_shape=(128,128,3)))
#keras.layers.BatchNormalization()
classifier.add(MaxPooling2D(pool_size=(2,2)))

classifier.add(Conv2D(filters=128,kernel_size=(3,3),padding='valid',activation="relu",kernel_regularizer=regularizers.l2(0.05)))
#keras.layers.BatchNormalization()
classifier.add(MaxPooling2D(pool_size=(2,2)))

classifier.add(Conv2D(filters=256,kernel_size=(3,3),padding='valid',activation="relu",kernel_regularizer=regularizers.l2(0.05)))
keras.layers.BatchNormalization()
classifier.add(MaxPooling2D(pool_size=(2,2)))



classifier.add(Dropout(0.5))


classifier.add(Flatten())# Flattening

#classifier.add(Dense(200,activation="relu"))

classifier.add(Dense(512,activation="relu"))
classifier.add(Dense(256,activation="relu"))
#classifier.add(Dense(128,activation="relu"))

classifier.add(Dropout(0.5))


classifier.add(Dense(3,activation="softmax"))  #3 represent output layer neurons for three different classes
classifier.summary()
```

Artificial Neural Network:-

Artificial Neural Networks (ANN) is a supervised learning system built of a large number of simple elements, called neurons or perceptrons. Each neuron can make simple decisions, and feeds those decisions to other neurons, organized in interconnected layers. Together, the neural network can emulate almost any function, and answer practically any question, given enough training samples and computing power. A “shallow” neural network has only three layers of neurons:

- **An input layer** that accepts the independent variables or inputs of the model
- **One hidden layer**
- **An output layer** that generates predictions

A Deep Neural Network (DNN) has a similar structure, but it has two or more “hidden layers” of neurons that process inputs. In order to discover the optimal weights for the neurons, we perform a backward pass, moving back from the network’s prediction to the neurons that generated that prediction. This is called backpropagation.



***<u>Batch Size</u>*:-**

Batch size is a term used in machine learning and refers to the number of training examples utilized in one iteration. Here we have mostly got better results with 16 and 32 as our batch sizes after doing tuning using multiple batch sizes.

<u>Epochs:-</u>

An epoch is a term used in machine learning and indicates the number of passes of the entire training dataset the machine learning algorithm has completed. .If the batch size is the whole training dataset then the number of epochs is the number of iterations. Here we used 70 epochs for ANN and mostly 100 epochs for CNN models.

<u>Optimizers:-</u>

optimizers shape and mold the deep leanring model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction. Here mostly we used Adam, Adamax,SGD,RMSProp while doing the optimzer hyper-parameter tuning of our models and found Adam to be wokring best with Adamax being the second best.

***<u>Test Train Split</u>*: -**

After the data preprocessing, we have splitted our image dataset into training data and testing data and validation data in 0.2 and 0.5 ratios respectively.

***<u>Oversampling of Imbalanced Dataset</u>*:-

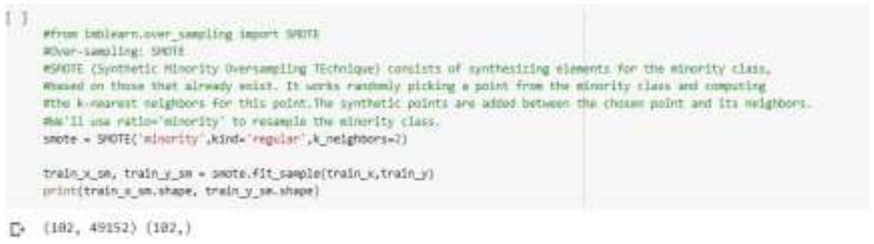
Since our image dataset was a little imbalanced one, so we tried applying the oversampling technique. To handle this problem, we used over sampling of the dataset using SMOTE Sampling. We ignored under sampling as it reduced the number of datapoints in training dataset, resulting serious adverse effects on accuracy.

SMOTE (Synthetic Minority Over-sampling Technique) is famous over sampling technique which we used to create non duplicated synthetic samples of minority class, so that minority class becomes equal to majority class. It selects similar records and then it randomly alters it in one column at a time around the difference of the nearest neighbors.

The SMOTE algorithm works in 4 simple steps: -

- First, we select the minority class and then input vector associated with it.
- Then we find the K nearest neighbours of each observation in imbalanced class in dataset.
- After that, we choose any of these neighbours randomly and put a synthetic point in any place on the line, which joins the synthetic point and the point we selected in one of the nearest neighbours.
- We keep repeating these steps until the dataset or particular class of the dataset in balanced.

After SMOTE we get training dataset like this :-



```
[ ]
    #from imblearn.over_sampling import SMOTE
    #Over-sampling: SMOTE
    #SMOTE (Synthetic Minority Oversampling TEchnique) consists of synthesizing elements for the minority class,
    #based on those that already exist. It works randomly picking a point from the minority class and computing
    #the k-nearest neighbors for this point.The synthetic points are added between the chosen point and its neighbors.
    #We'll use ratio='minority' to resample the minority class.
    smote = SMOTE('minority',kind='regular',k_neighbors=2)

    train_x_sm, train_y_sm = smote.fit_sample(train_x,train_y)
    print(train_x_sm.shape, train_y_sm.shape)

    (102, 49152) (102,)
```

2.5 <u>Evaluation of Model and Validation for Multi Class Classification:</u>

***<u>Precision, Recall, F Score</u>*: -** Precision refers to the ratio percentage of relevant prediction of the correct class belongingness of a particular new test dataset relative to the total number of positive predictions being made. Whereas the recall is the ratio percentage of sum of all positive predictions which are actually positive to the total number of all relevant predictions. our intuition tells us that we should maximize the positive classes, known in statistics as recall, or the ability of a model to find all the relevant cases within a dataset. F score comes into picture to deal with different machine learning models when there are imbalance class in our dataset. The closer the F score to 1, the better the model performance on new unseen test data set will be. In order to have a classification model with the desired balance of recall and precision, we maximize the F1 score as much as possible.

Confusion Matrix: - Confusion matrix is a performance management matrix for classifying two or more classes. four different combinations of predicted as well as actual values creates the confusion matrix. The rows represent predicted values & the columns represent actual values. Unlike binary classification, there are no positive or negative classes in multi class classification. At first, it might be a little difficult to find TP, TN, FP and FN since there are no positive or negative classes, but it's actually pretty easy. What we have to do here is to find TP, TN, FP and FN for each individual class. The diagonal elements in the multi class classification confusion matrix are the actual true positives, whereas the sum of the all other elements in the 3*3 confusion matrix for three class are the false positive for that class. Here we fix one class as positive and take all other classes as negative and start analyzing the confusion matrix.

- True Positive (TP) refers to the value that is predicted as true and actually those are true.
- False Positives (FP-Type I Error) are those cases which have been predicted as yes but actually no.
- True Negative (TN) refers to those cases where the value is predicted as no and actually those are no.
- False Negatives (FN-Type II Error) are those cases where the value is predicted as no, but those are actually yes.

Based on these four parameters the performance of algorithms can be adjudged by calculating the following ratios: -

$$Accuracy(\%) = \frac{TP+TN}{TP+FP+TN+FN}$$

$$TPR(\%) = \frac{TP}{TP+FN}$$

$$FPR(\%) = \frac{FP}{FP+TN}$$

$$precision(\%) = \frac{TP}{TP+FP}$$

$$recall = \frac{TP}{TP+FN}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

2.6 CNN and ANN Deep Learning Image Classification Overfitting/Underfitting Preventive Techniques used

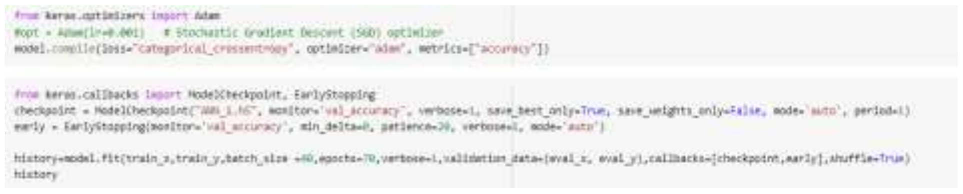
Adding neuron layers or inputs—adding neuron layers, or increasing the number of inputs and neurons in each layer, can generate more complex predictions and improve the fit of the model.

Retraining neural networks—running the same model on the same training set but with different initial weights, and selecting the network with the best performance.

Decreasing regularization parameter—regularization can be overdone. By using a regularization performance parameter, you can learn the optimal degree of regularization, which can help the model better fit the data.

Dropout:- **Dropout** is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. Here for ANN we have used a drop out of 0.2 and 0.5 for all the CNN architectures. It randomly "kill" a certain percentage of neurons in every training iteration. This ensures some information learned is randomly removed, reducing the risk of overfitting.

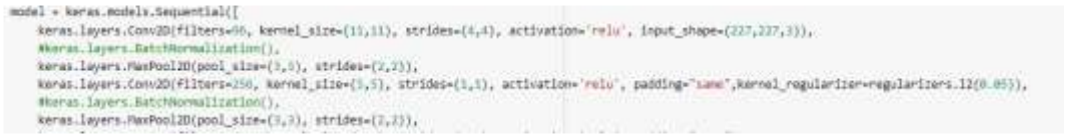
Early Stopping:- **early stopping** is a form of **regularization** used to avoid overfitting when training a learner with an iterative method, such as gradient descent. Such methods update the learner so as to make it better fit the training data with each iteration.We have used the early stopping while fitting the model.



```
from keras.optimizers import Adam
#opt = Adam(lr=0.001)   # Stochastic Gradient Descent (SGD) optimizer
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

```
from keras.callbacks import ModelCheckpoint, EarlyStopping
checkpoint = ModelCheckpoint("ANN_1.h5", monitor='val_accuracy', verbose=1, save_best_only=True, save_weights_only=False, mode='auto', period=1)
early = EarlyStopping(monitor='val_accuracy', min_delta=0, patience=20, verbose=1, mode='auto')

history=model.fit(train_x,train_y,batch_size =40,epochs=70,verbose=1,validation_data=(eval_x, eval_y),callbacks=[checkpoint,early],shuffle=True)
history
```

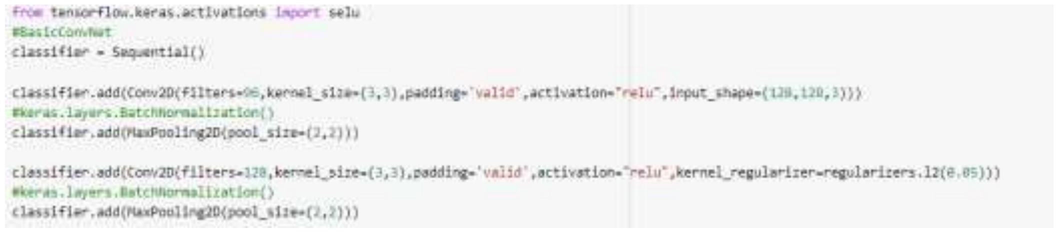
L2 Regularisation:- **Regularization** in machine learning is the process of regularizing the parameters that constrain, regularizes, or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model, **avoiding** the risk of **Overfitting**. **L2 regression** adds "squared magnitude" of coefficient as penalty term to the loss function. Here, we ended up using the L2 regularisation in CNN and ANN in first layer after input layer only with final value of 0.05 for CNN and 0.001 for ANN.



```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(227,227,3)),
    #keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same",kernel_regularizer=regularizers.l2(0.05)),
    #keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
```

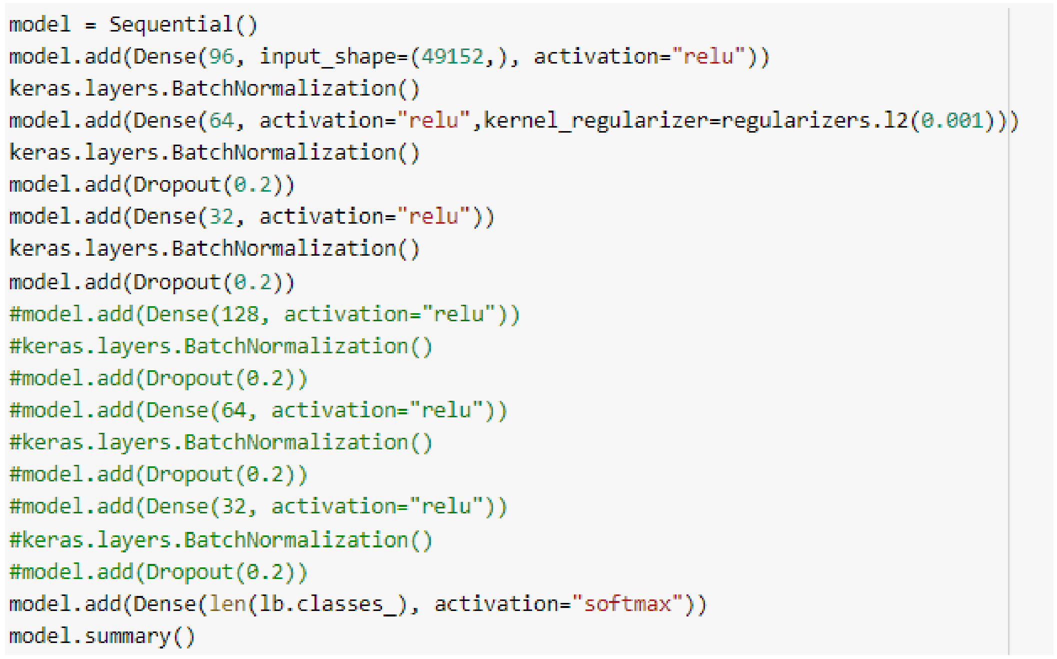
<u>Batch Normalisation:-</u> **Batch normalization** is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-**batch**. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks. Here we added batch normalisation after each input, convolutional, fully connected layers in CNN but we were getting overfitted result mostly with low test accuracy. So we decided not to use batch normalisation for CNN. Whereas in ANN, after using batch normalisation after each of the layers we got much better loss and accuracy result than what we were getting before.

<u>CNN-Batch Normalisation Usage</u>



```
from tensorflow.keras.activations import selu
#BasicConvNet
classifier = Sequential()

classifier.add(Conv2D(filters=96,kernel_size=(3,3),padding='valid',activation="relu",input_shape=(128,128,3)))
#keras.layers.BatchNormalization()
classifier.add(MaxPooling2D(pool_size=(2,2)))

classifier.add(Conv2D(filters=128,kernel_size=(3,3),padding='valid',activation="relu",kernel_regularizer=regularizers.l2(0.05)))
#keras.layers.BatchNormalization()
classifier.add(MaxPooling2D(pool_size=(2,2)))
```

<u>ANN-Batch Normalisation Usage</u>



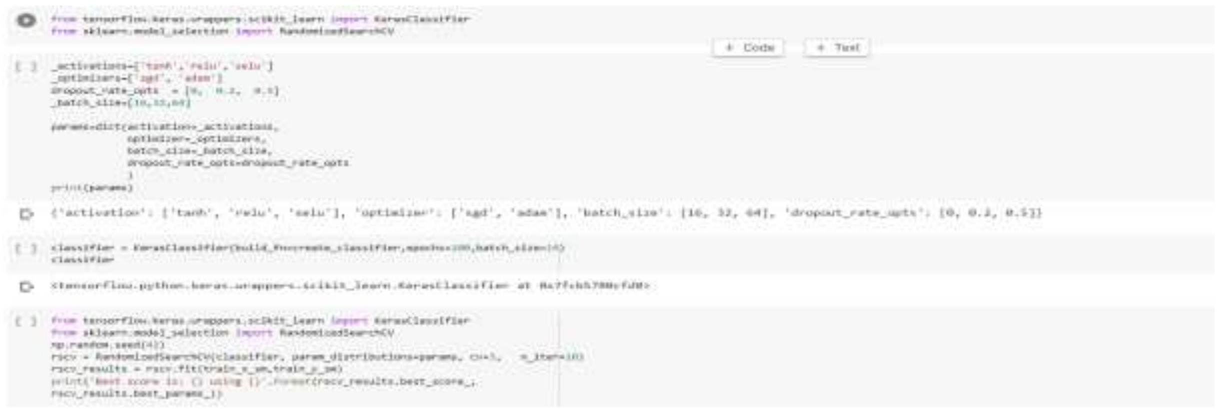
```
model = Sequential()
model.add(Dense(96, input_shape=(49152,), activation="relu"))
keras.layers.BatchNormalization()
model.add(Dense(64, activation="relu",kernel_regularizer=regularizers.l2(0.001)))
keras.layers.BatchNormalization()
model.add(Dropout(0.2))
model.add(Dense(32, activation="relu"))
keras.layers.BatchNormalization()
model.add(Dropout(0.2))
#model.add(Dense(128, activation="relu"))
#keras.layers.BatchNormalization()
#model.add(Dropout(0.2))
#model.add(Dense(64, activation="relu"))
#keras.layers.BatchNormalization()
#model.add(Dropout(0.2))
#model.add(Dense(32, activation="relu"))
#keras.layers.BatchNormalization()
#model.add(Dropout(0.2))
model.add(Dense(len(lb.classes_), activation="softmax"))
model.summary()
```

Hyper Parameter Optimisation Techniques used:-

1. ***Manual***: select hyperparameters based on intuition/experience/guessing, train the model with the hyperparameters, and score on the validation data. Repeat process until you run out of patience or are satisfied with the results.
2. ***Grid Search***: set up a grid of hyperparameter values and for each combination, train a model and score on the validation data. In this approach, every single combination of hyperparameters values is tried which can be very inefficient!
3. ***Random search***: set up a grid of hyperparameter values and select *random* combinations to train the model and score. The number of search iterations is set based on time/resources.

We used random search, grid search and manual search for our CNN and ANN model hyper parameter tuning to compare all three and finally to arrive with the best hyper-parameters for the model.

Random Search Code Screen Shot



```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import RandomizedSearchCV
```

```
_activations=['tanh','relu','selu']
_optimizers=['sgd', 'adam']
dropout_rate_opts  = [0,  0.2,  0.5]
_batch_size=[16,32,64]

params=dict(activation=_activations,
            optimizer=_optimizers,
            batch_size=_batch_size,
            dropout_rate_opts=dropout_rate_opts
            )
print(params)
```

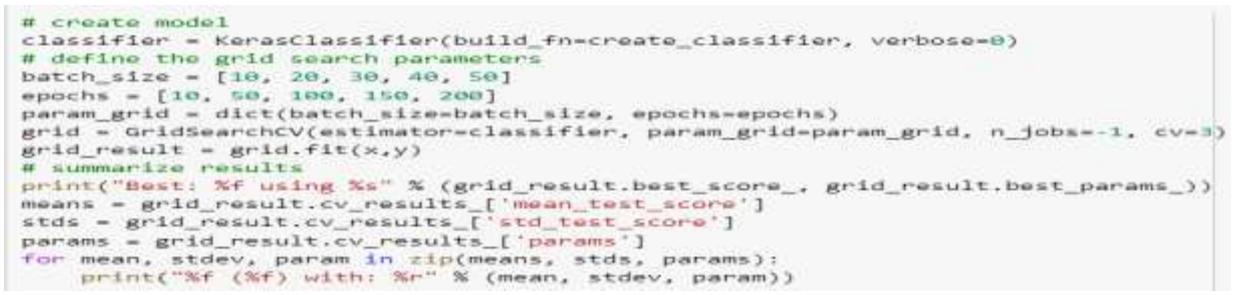
```
{'activation': ['tanh', 'relu', 'selu'], 'optimizer': ['sgd', 'adam'], 'batch_size': [16, 32, 64], 'dropout_rate_opts': [0, 0.2, 0.5]}
```

```
classifier = KerasClassifier(build_fn=create_classifier,epochs=100,batch_size=16)
classifier
```

```
<tensorflow.python.keras.wrappers.scikit_learn.KerasClassifier at 0x7fcb5780cfd0>
```

```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import RandomizedSearchCV
np.random.seed(42)
rscv = RandomizedSearchCV(classifier, param_distributions=params, cv=3,   n_iter=10)
rscv_results = rscv.fit(train_x_sm,train_y_sm)
print('Best score is: {} using {}'.format(rscv_results.best_score_,
rscv_results.best_params_))
```

Grid Search Code Screen Shot



```
# create model
classifier = KerasClassifier(build_fn=create_classifier, verbose=0)
# define the grid search parameters
batch_size = [10, 20, 30, 40, 50]
epochs = [10, 50, 100, 150, 200]
param_grid = dict(batch_size=batch_size, epochs=epochs)
grid = GridSearchCV(estimator=classifier, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(x,y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

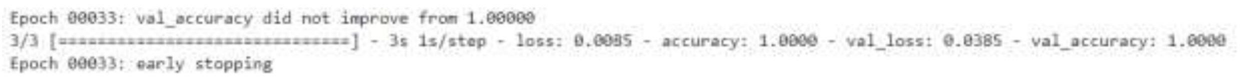
3.Accuracy Result Comparison and Discussion:-

CNN Results Comparison on Different Metrics:-

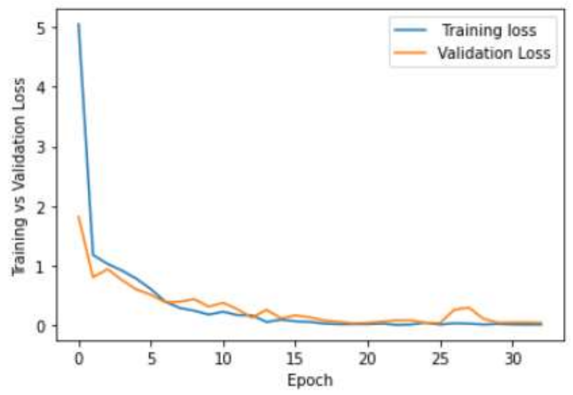
Best Performing CNN Architecture Results

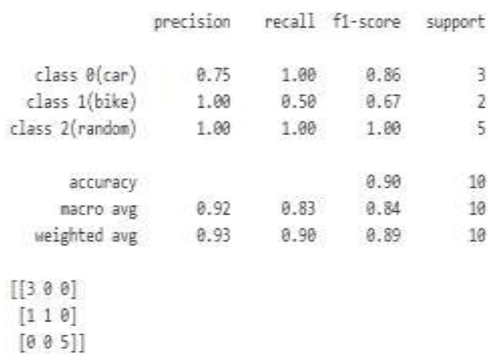
Lenet-5 Result:-

Train and Validation Loss and Accuracy,Precision,Recall,F1 Score & Confusion Matrix:-

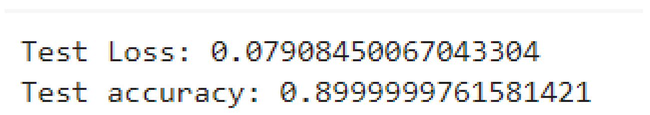


```
Epoch 00033: val_accuracy did not improve from 1.00000
3/3 [==============================] - 3s 1s/step - loss: 0.0085 - accuracy: 1.0000 - val_loss: 0.0385 - val_accuracy: 1.0000
Epoch 00033: early stopping
```



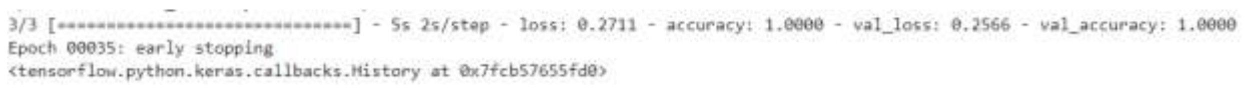


```
                  precision    recall  f1-score   support

    class 0(car)       0.75      1.00      0.86         3
   class 1(bike)       1.00      0.50      0.67         2
 class 2(random)       1.00      1.00      1.00         5

        accuracy                           0.90        10
       macro avg       0.92      0.83      0.84        10
    weighted avg       0.93      0.90      0.89        10

[[3 0 0]
 [1 1 0]
 [0 0 5]]
```



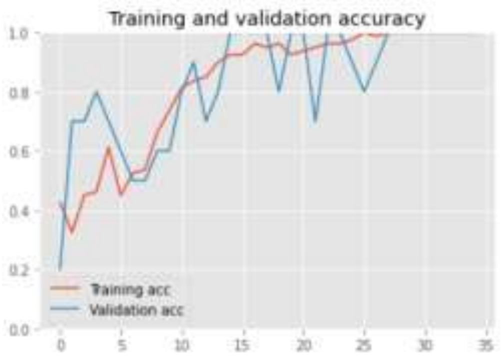
```
Test Loss: 0.07908450067043304
Test accuracy: 0.8999999761581421
```

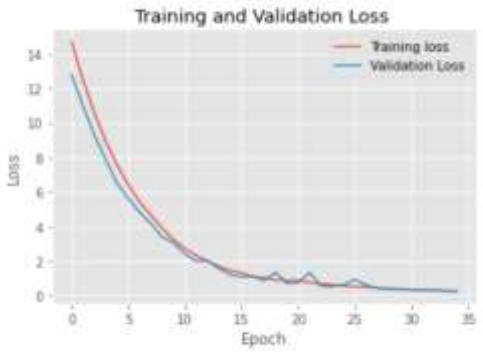
My Own CNN Architecture(CNN-8) Results:-



```
3/3 [==============================] - 5s 2s/step - loss: 0.2711 - accuracy: 1.0000 - val_loss: 0.2566 - val_accuracy: 1.0000
Epoch 00035: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fcb57655fd0>
```

Epoch Vs Loss/ Accuracy





```
                  precision    recall  f1-score   support

   class 0(car)       1.00      1.00      1.00         3
  class 1(bike)       1.00      1.00      1.00         2
class 2(random)       1.00      1.00      1.00         5

       accuracy                           1.00        10
      macro avg       1.00      1.00      1.00        10
   weighted avg       1.00      1.00      1.00        10

[[3 0 0]
 [0 2 0]
 [0 0 5]]
```

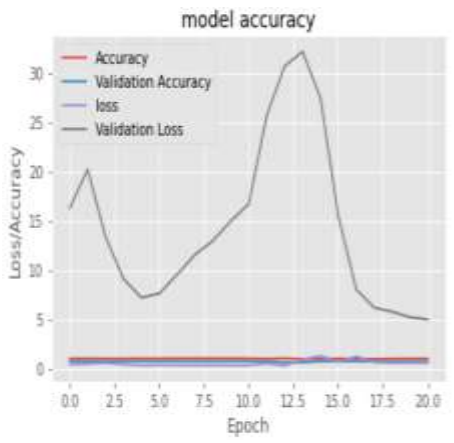
```
Test Loss: 0.26072797179222107
Test accuracy: 1.0
```

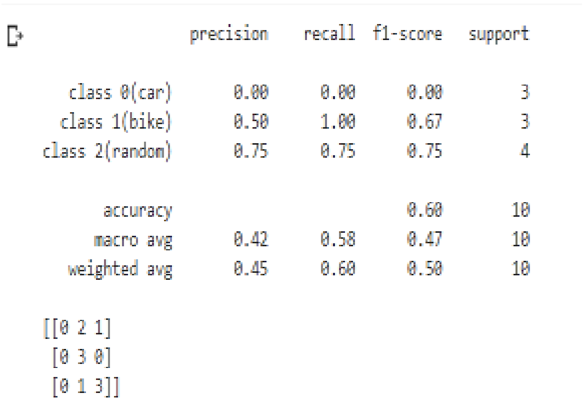
<u>Worst Performing CNN Architectures:-</u>

<u>Alexnet Result:-</u>

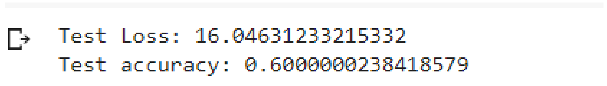
<u>Train and Validation Loss and Accuracy,Precision,Recall,F1 Score & Confusion Matrix:-</u>

```
Epoch 00022: val_accuracy did not improve from 0.70000
2/2 [==============================] - 1s 305ms/step - loss: 3.6772 - accuracy: 0.9500 - val_loss: 5.7944 - val_accuracy: 0.7000
Epoch 00022: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fcb363f0a58>
```





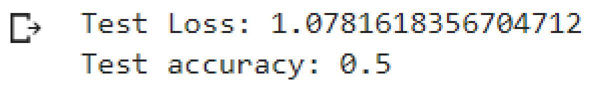
```
              precision    recall  f1-score   support

  class 0(car)       0.00      0.00      0.00         3
 class 1(bike)       0.50      1.00      0.67         3
class 2(random)      0.75      0.75      0.75         4

      accuracy                           0.60        10
     macro avg       0.42      0.58      0.47        10
  weighted avg       0.45      0.60      0.50        10

[[0 2 1]
 [0 3 0]
 [0 1 3]]
```



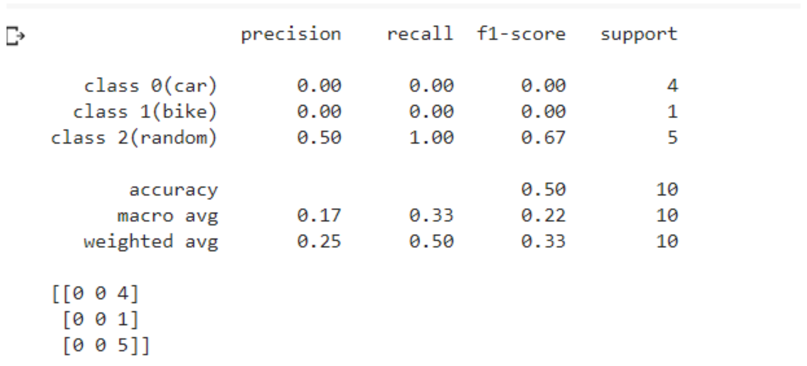
```
Test Loss: 16.04631233215332
Test accuracy: 0.6000000238418579
```

VGG 16 Result:-

Train and Validation Loss and Accuracy,Precision,Recall,F1 Score & Confusion Matrix:-

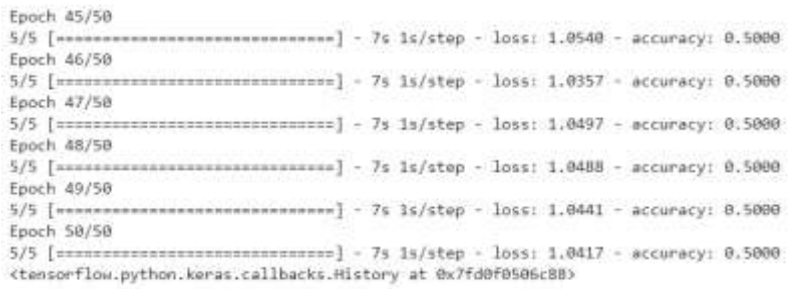


```
Test Loss: 1.0781618356704712
Test accuracy: 0.5
```



```
                 precision    recall  f1-score   support

   class 0(car)       0.00      0.00      0.00         4
  class 1(bike)       0.00      0.00      0.00         1
class 2(random)       0.50      1.00      0.67         5

       accuracy                           0.50        10
      macro avg       0.17      0.33      0.22        10
   weighted avg       0.25      0.50      0.33        10

[[0 0 4]
 [0 0 1]
 [0 0 5]]
```

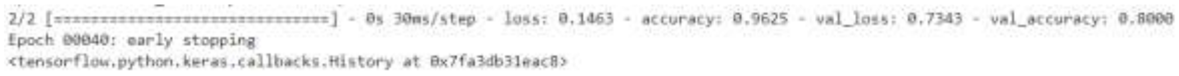
Using Image Data Augmentation on this small dataset result:-



```
Epoch 45/50
5/5 [==============================] - 7s 1s/step - loss: 1.0540 - accuracy: 0.5000
Epoch 46/50
5/5 [==============================] - 7s 1s/step - loss: 1.0357 - accuracy: 0.5000
Epoch 47/50
5/5 [==============================] - 7s 1s/step - loss: 1.0497 - accuracy: 0.5000
Epoch 48/50
5/5 [==============================] - 7s 1s/step - loss: 1.0488 - accuracy: 0.5000
Epoch 49/50
5/5 [==============================] - 7s 1s/step - loss: 1.0441 - accuracy: 0.5000
Epoch 50/50
5/5 [==============================] - 7s 1s/step - loss: 1.0417 - accuracy: 0.5000
<tensorflow.python.keras.callbacks.History at 0x7fd0f0506c88>
```

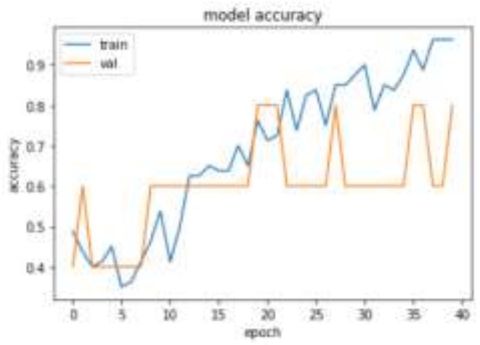
<u>ANN Results comapared to best CNN Models on Different Metrics:-</u>

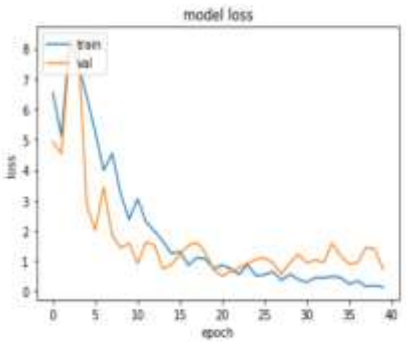
<u>Train and Validation Loss and Accuracy,Precision,Recall,F1 Score & Confusion Matrix:-</u>

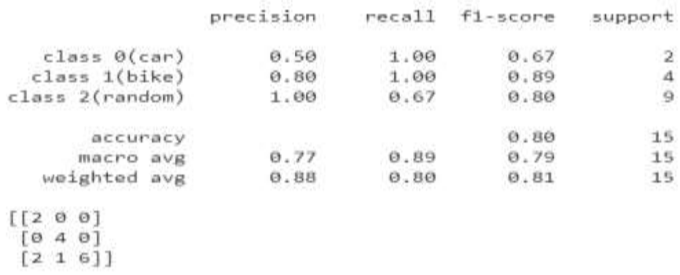


```
2/2 [==============================] - 0s 30ms/step - loss: 0.1463 - accuracy: 0.9625 - val_loss: 0.7343 - val_accuracy: 0.8000
Epoch 00040: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa3db31eac8>
```

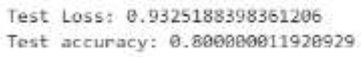
Epoch Vs Loss/ Accuracy







```
                   precision    recall  f1-score   support

     class 0(car)       0.50      1.00      0.67         2
    class 1(bike)       0.80      1.00      0.89         4
  class 2(random)       1.00      0.67      0.80         9

         accuracy                           0.80        15
        macro avg       0.77      0.89      0.79        15
     weighted avg       0.88      0.80      0.81        15

[[2 0 0]
 [0 4 0]
 [2 1 6]]
```



```
Test Loss: 0.9325188398361206
Test accuracy: 0.800000011920929
```

So if we compare all of our CNN , ANN models we used on different metrics we can see at first CNN architectures are performing much better than ANN architectures due to the presense of convolutional layers in CNN.

The reason why **Convolutional Neural Networks** (CNNs) do so much better than artificial neural networks on images is that the convolutional layers take advantage of **inherent properties** of images.

<u>Convolutions</u>

- Simple feedforward neural networks **don't see any order in their inputs**. If you shuffled all your images in the same way, the neural network would have the very same performance it has when trained on not shuffled images.

- CNN, in opposition, take advantage of **local spatial coherence** of images. This means that they are able to reduce dramatically the number of operations needed to process an image by using convolution on **patches of adjacent pixels,** because adjacent pixels together are meaningful. We also call that local connectivity. Each map is then filled with the result of the convolution of a small patch of pixels, slid with a window over the whole image.

<u>Pooling layers</u>

- There are also the **pooling layers,** which downscale the image. This is possible because we retain throughout the network, features that are organized spatially like an image, and thus downscaling them makes sense as reducing the size of the image. **On classic inputs you cannot downscale a vector**, as there is **no coherence** between an input and the one next to it.

ANN on other hand uses one perceptron for each input (e.g. pixel in an image) and the amount of weights rapidly becomes unmanageable for large images. It includes too many parameters because it is fully connected. Each node is connected to every other node in next and the previous layer, forming a very dense web — resulting in redundancy and inefficiency. As a result, difficulties arise whilst training and *overfitting* can occur which makes it lose the ability to generalize.

Another common problem is that ANNs react differently to an input (images) and its shifted version — they are not translation invariant. For example, if a picture of a car appears in the top left of the image in one picture and the bottom right of another picture, the ANN will try to correct itself and assume that a car will always appear in this section of the image.

Hence, ANNs are not the best idea to use for image processing. One of the main problems is that spatial information is lost when the image is flattened(matrix to vector) into an ANN.

Also, as far as data imbalance problem is concerned, the dataset is so small and we applied the SMOTE to handle this problem. But this procedure did not benefit us much in getting better averaged accuracy. So, we proceeded with shuffled dataset in order to keep the variance and uncertainty alive in our dataset. On the other hand, image data augmentation, the popular feature engineering technique for image dataset , when the set is so small, does not seem to work well and gives poor accuracy of 50% on validation dataset as well as underfits the dataset.

Apart from ANN, we did not use any batch normalisation during building the neural network architecture in CNN models as it was overfitting the model.

We ran random search and grid search to achieve optimal combination of hyper parameter tuning. In ANN, random search gave 61% accuracy with optimizer adam, dropout rate 0.2 and batch size of 16. But due to very less amount of dataset, the hyper parameter tuning was overfitting itself. So, we additionally did manual tuning for ANN achieved a much better result with train and validation loss of 95% and 80% while train and validation error or 0.2 and 0.7. This models overfits a little and hence we applied much advanced deep learning models like CNN for further this multi class car vs bike vs random image classificatiom. ANN gives 80% accuracy on test dataset with 0.9 as loss, which seems to be decent but of course performance can be improved with far better models like CNN.

In CNN, my own architecture , that I created, CNN-8 along with Lenet-5 gave the best result with training and validation accuracy close to 100% and at the same time train and validation accuracy almost close to 0.The loss vs accuracy graphs mentioned above in the result supports this claim. CNN-8 gave test dataset accuracy of 1 with test loss of 0.26 whereas the Lenet-5 gave test dataset accuracy of 89% with a test loss of 0.079.

Alex Net, due to its work with 60 million parameters and we having very less dataset, is overfitting the model with train and validation accuracy of 95% and 70% with high train and validation loss as 3.6 and 5.7. It performs poorly on test dataset with only 50% accuracy.

As far as confusion matrices, precision, recall and F score are concerned since we are dealing with an imbalance class problem in a small dataset, so we will mostly focus on weighted average F Score which is 1 and 0.9 for my own CNN-8 and the Lenet-5 CNN model. Also, for both these models each class precession and recall are high as well. With only 1 class has been misclassified in Lenet.

With hyper-parameter tuning, the best parameters came out to be dropout rate at 0.5 with optimizer "Adam" or "Adamax", batch size of 32 with epoch of 100. In L2 regularisation, 0.05 or 0.001 od learning rate proved to be best options available for this dataset in CNN and ANN. Stochastic Gradient Descent does not perform well as optimizer as Adam or Adamax. For better performance , in future we would like to use some automated hyper parameter optimization techniques like Bayesian Optimization, Optuna or Tree based Pozran method as random search , being searched for best hyper parameter at random, depends on luck and introduces high bias and variance sometimes. Also gird search , being run as a combination of each of the different hyper parameters, take too much time and computationally expensive.

Few of the further suggestions to improve the accuracy and minimizing the losses across the training and validation and testing dataset for all the deep learning model used above are:-

- Increase the size of the image dataset.

- Keep manipulating the dropout rate, regularisation ,early stopping, batch size, optimizer, number of epochs using different hyper parameter tuning options to tackle overfitting /under fitting, if necessary.Also use batch normalisation after each layer if it is helping.

- Increase the number of convolutional or fully connected layers in case of CNN and hidden layers in terms of ANN to increase the accuracy. Also keep changing and decreasing the number of neurons in each of these hidden layers to achieve optimisation.