Classification Accuracy and Time Complexity for Classification Models on CIFAR-10

STAT 654

Spring 2020

Quentarius Moore

Deep Patel

Pooja Phadke

Rohit Sonje

Adaiyibo Kio

Table of Contents

List of Figures

List of Tables

1. Summary

In this work, we have explored a number of techniques in our efforts to model and understand the classification of images in the CIFAR-10 dataset. Traditional techniques such as, but not limited to, Naïve Bayes, Random Forest, and Voting Classifier were initially implemented to deduce which methods were promising for fitting to the training dataset from Principal Component Analysis (PCA). This method was also implemented again, however; Histogram of Gradient was done before PCA. Lastly, we focused on a more advanced algorithm, Convoluted Neural Networks (CNN), in which the hyperparameters of the CNN model, VGG16, were successfully tuned to achieve an increase in accuracy from 0.7437 to 0.9186.

2. Introduction

Feature engineering is a game-changer in the world of machine learning algorithms. **Feature engineering** is the process of transforming raw data into **features** that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data. Feature engineering turns inputs into things the algorithm can understand.

Considering the Feature engineering for **the unstructured data**, there are numerous techniques like Spatial features, Transform features, Edges and boundaries, Shape features, Moments Texture, etc. for the object identification or image classification. The main principle of improvisation using such techniques lies in **Feature Extraction**.

Feature extraction techniques are applied to get features that will be useful in classifying and recognition of images. Amongst the variety of feature techniques present like Scale Invariant Feature Transform and Speeded-Up Robust Feature, etc., we are going to leverage HOG Feature Extraction technique.

HOG stands for **Histograms of Gradients** in which the complete image is broken down into smaller regions and for each region, the gradients and orientation are calculated. Finally, **Histograms** for each of these regions separately are created using the gradients and orientations of the pixel values. As features define the behavior of an image, they show place in terms of storage taken, efficiency in classification and obviously in time consumption also. In this paper we will not only show the accuracy comparison of ML algorithms but also time complexity comparison performing traditional ML algorithms before and after leveraging HOG Feature extraction technique. At the end we will even compare the best results with the advanced techniques of Convolutional Neural Networks.

3. Dataset

The image dataset being used is CIFAR-10, a dataset consisting of 60000 32x32 color images in 10 classes, with 6000 images per class.

The different classes are airplane, automobile, bird, truck, etc. The classes are disjoint, thus do not overlap with each other such as vehicles in "Truck" are big trucks (18-wheelers, etc.), which are inherently different from the vehicles (sedans, SUVs, etc.) in the "Automobile" class.

There are 50000 images used for training machine learning models and 10000 images are used for testing these models. The dataset is balanced, with 5000 images per class in the training dataset. Figure 1 shows examples of those images with the number above each image indicating the class label. The dataset can be obtained from its University of Toronto repository (found here), from the Python keras library or the Kaggle website. In the course of this work, at various times, the dataset was obtained from all those sources.

Figure 1 Images from CIFAR-10 dataset labeled according to their respective classes

4. Traditional Algorithms on RGB channel values

All work on this data was done using Python and the appropriate machine learning packages. Since this dataset had already been curated, no cleaning of the data was necessary. However, since each image was a 32 x 32 x 3 array, the images were reshaped to be a 1 x 3072 array.

The training data set was split in an 85:15 ratio into a training set (35000 images) and a validation set (15000 images). The training data set was, therefore, 35000 x 3072. Since this would be computationally expensive to implement machine learning algorithms on a dataset with such a large feature space, some dimensionality reduction was called for.

Principal Component Analysis

Principal Component Analysis (PCA) was used to reduce the number of features in the dataset to a number that would be manageable by classical machine learning algorithms. Figure 2 shows that 34 principal components can explain about 80% of the variance in the data. 34 principal components were adjudged to be a good number to balance computational expense and fidelity to the original data.

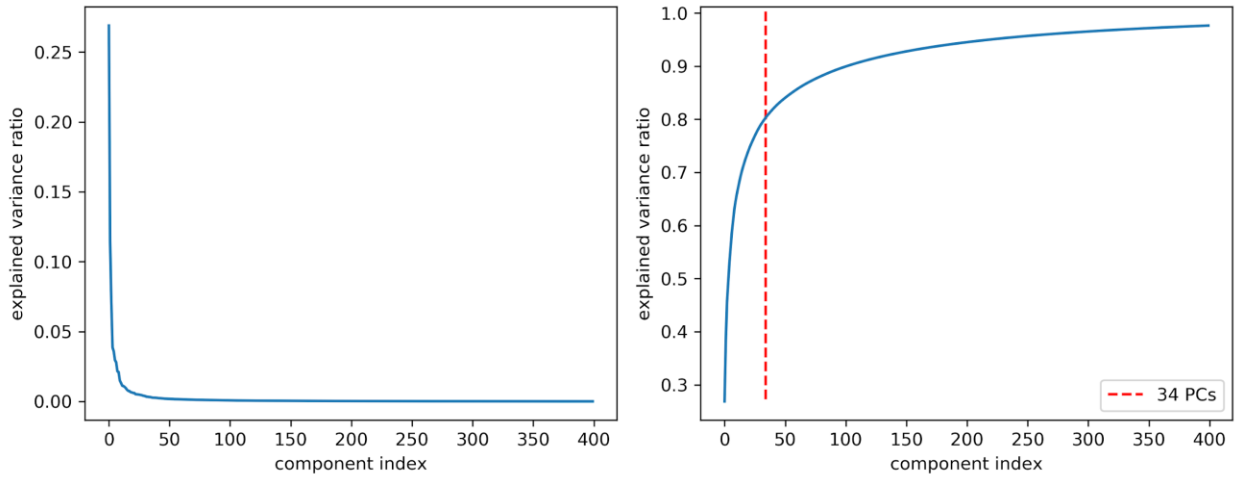

Figure 2 Principal Components Analysis plots

Modeling

The approach taken to the model building for this work is as follows.

(a) Implement traditional machine learning algorithms fitted to the new training dataset obtained from PCA.
(b) Test these models for accuracy using the validation dataset.
(c) Take the models that give the most promising results, i.e. high accuracies and tune them using cross-validation, to prevent overfitting
(d) Test the models again with the validation dataset. By so doing, the best model and parameters are determined.
(e) Use the full training data set (training and validation datasets combined) and build a model with the model parameters determined in (d).

(f) Test the model with the test data and report all metrics.

The following algorithms were implemented.

(a) Naïve Bayes
(b) K-Nearest Neighbors
(c) Logistic Regression
(d) Random Forest
(e) Support Vector Machines
(f) Extreme Gradient Boosting
(g) Voting Classifier

All algorithms except the Voting Classifier were implemented without any hyperparameter tuning in order to determine which algorithms showed promise enough to warrant further tuning. Table 1 shows the results of testing these models with validation set data. It also shows the time it took to train these models. It is observed that the more accurate models are the most computationally expensive in general. The Support Vector Machine model is the most accurate and most computationally expensive.

Table 1 Metrics for various basic classification models

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**Time (s)**
Naïve Bayes	0.37	0.37	0.37	0.36	0.018
KNN	0.39	0.41	0.39	0.39	0.168
Logistic Regression	0.37	0.37	0.37	0.37	0.836
Random Forest	0.46	0.46	0.46	0.46	26.00
Gradient Boosting	0.48	0.48	0.48	0.48	11.30
SVM	0.52	0.52	0.52	0.52	263.00

It can be observed that Random Forest, Gradient Boosting and Support Vector Machines showed the best promise with accuracies of 0.46, 0.48, and 0.52, respectively. Precision, Recall and F1 Score show a similar trend.

To obtain the best possible model using these three algorithms, hyperparameter tuning was done. Grid Search and Random Search were done over the possible hyperparameter spaces for each of the algorithms in order to determine the one with the best test error using the validation dataset. Cross-validation was used, not just to help find the right hyperparameters, but to also prevent over-fitting the models.

When the best hyperparameters were determined, these models were tested with the test dataset. In addition, a Voting Classifier was also built using a combination of Random Forest, Gradient Boosting and Support Vector Machines. This classifier predicted the class label based on the argmax of the sums of the predicted probabilities of its constituent models. Table 2 shows the metrics for the models as well as the time it took to train them. As can be expected, the Voting Classifier is the most computationally expensive model since it involves building three separate models. For the other models, Gradient Boosting, like the models in Table 1, takes the most time to train.

Figures 3 – 6 also shows the confusion matrices for these models. The ideal confusion matrix should be a diagonal matrix. However, these matrices are not exactly diagonal. The overall picture is that these models misclassify a lot of the images, and better classification models should be sought.

Table 2 Metrics for final classification models

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**Time (min)**
Random Forest	0.49	0.49	0.49	0.49	01:34
Gradient Boosting	0.53	0.53	0.53	0.53	10:03
SVM	0.53	0.53	0.53	0.53	07:16
Voting	0.53	0.55	0.55	0.55	24:52

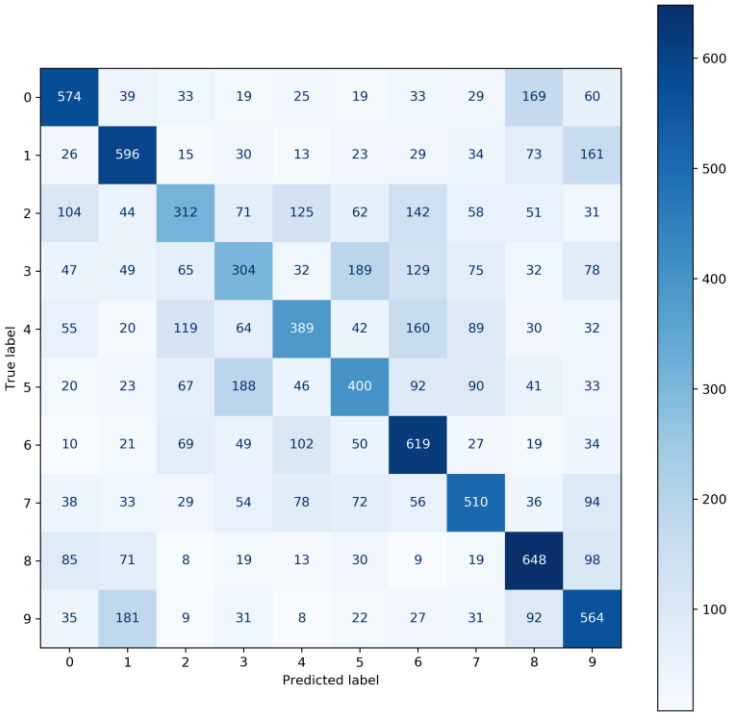

Figure 3 Confusion matrix for Random Forest model

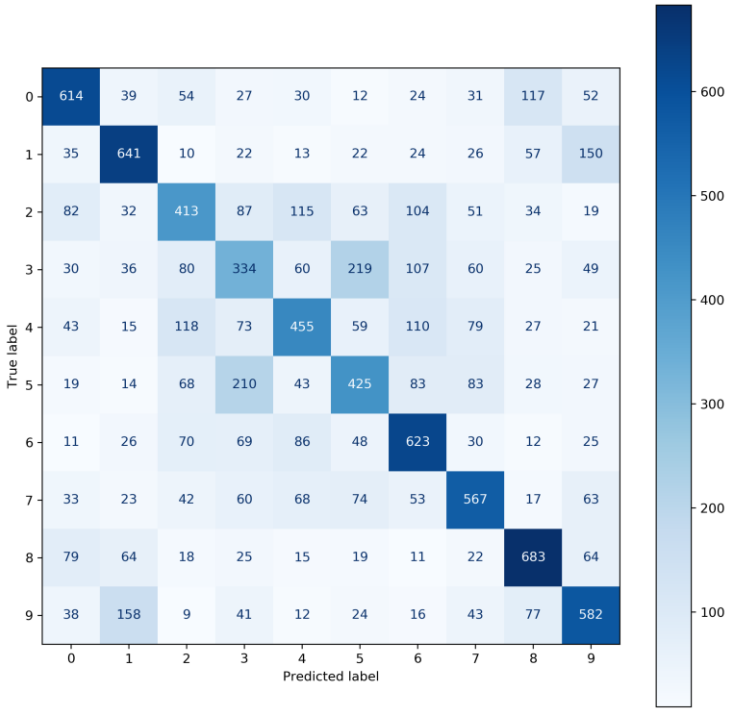

Figure 4 Confusion matrix for Gradient Boosting model

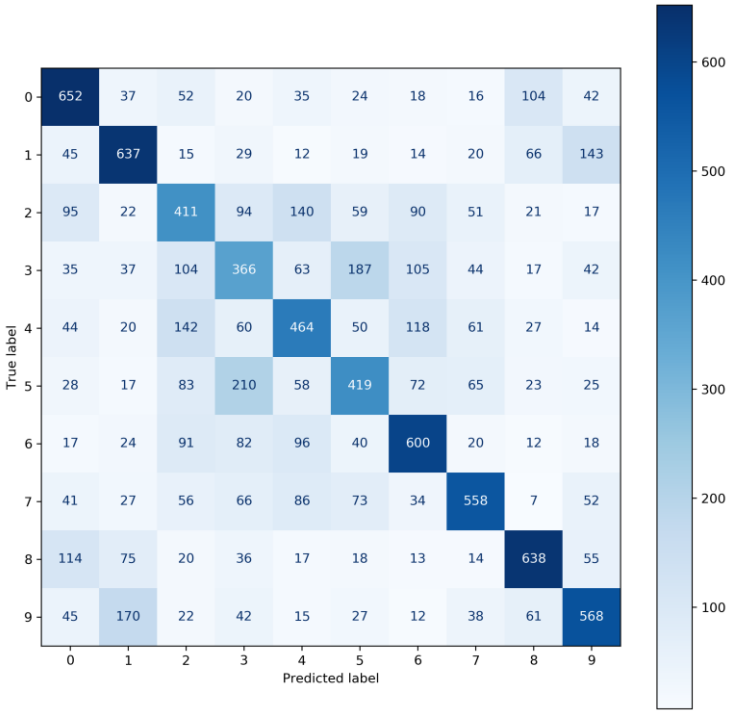

Figure 5 Confusion matrix for Support Vector Machine model

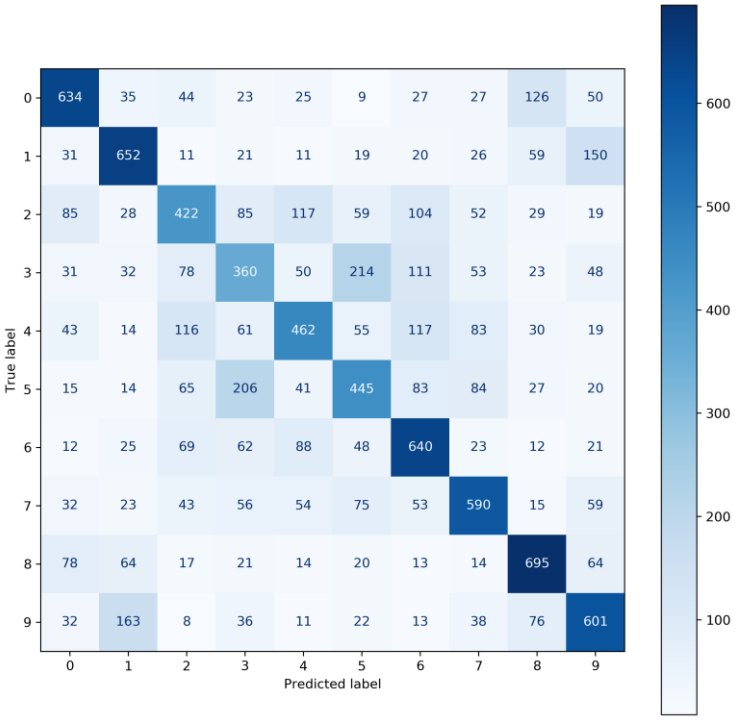

Figure 6 Confusion matrix for Voting Classifier model

5. HOG feature extraction technique

Introduction:

Edge feature extraction only identifies whether a particular pixel is an edge or not. HOG not only identifies whether a particular pixel is an edge or not, but also computes its direction. A complete image is broken down into numerous small regions called 'Localized Portions' and for each portion, the gradients and their orientations are calculated. Finally using these gradients and orientations, histograms are created for each pixel producing the features for the whole image. That is why this is called Histograms of Gradients.

HOG Procedure

Step 1: Preprocess the data:

Preprocessing data is a crucial step in any machine learning project and that's no different when working with images. All the images are required to be converted to width to height ratio of 1:2. In our project, we resized all the images from 32*32 to 64*128.

Step 2: Calculating Gradients for both 'X' and 'Y' directions:

Let the blue pixel at center, shown in figure 7, be our pixel of concern. Its gradients will be calculated as:

Gx (change in X direction) = X2 – X1
Gy (change in X direction) = Y2 – Y1

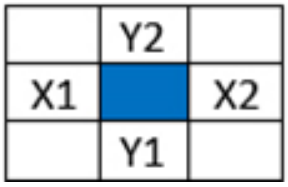

Figure 7 Pixel of an image

The magnitude would be higher when there is a sharp change in intensity, such as around the edges.

Step 3: Magnitude and Orientations:

Using the obtained change in gradients for both the directions, Magnitude and orientations are calculated as follows:

Total gradient magnitude = $\sqrt{(Gx)^2 + (Gy)^2}$

Orientation = $atan\ (\frac{Gy}{Gx})$

Step 4: Histogram of Gradients using 8*8 pixels per cell:

Gradient orientation for any pixel ranges from 0 to 180 degrees. We divide these ranges of 0-180 into 9 bins. Each bin has a bin size of 20 such as 1-20, 21-40, …, 161-180.

We then add the contribution of a pixel's gradient to the bins on either side of the pixel gradient. The higher contribution is given to the bin value which is closer to the orientation as shown in example below:

Let's say, Total gradient magnitude = 13 and Orientation = 36. For this pixel we will add into frequency of histograms as shown in figure 8:

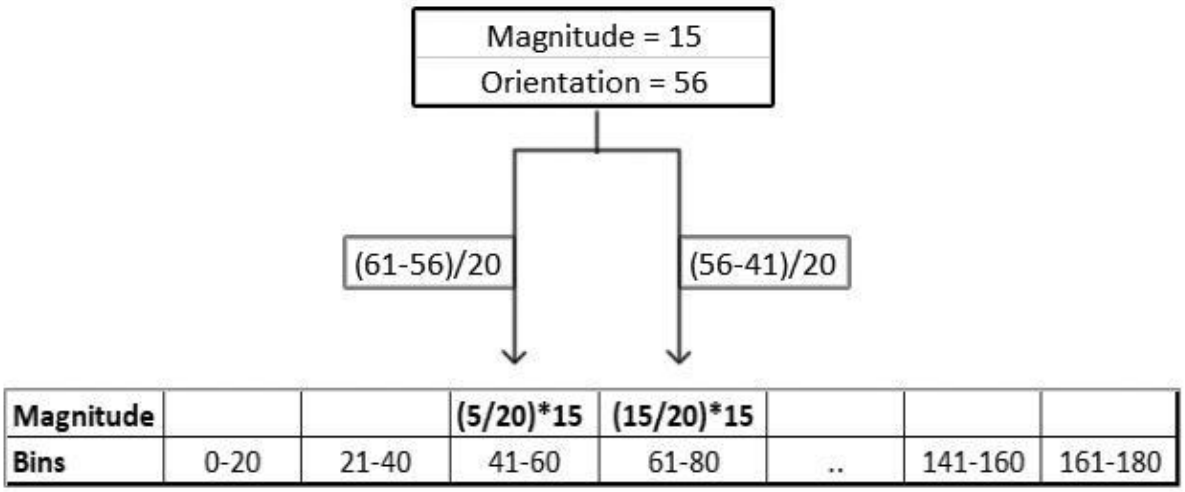

Figure 8 Histogram using Magnitude and Orientation

One histogram is created per each cell made of 8*8 pixels that produces a (9*1) vector of features.

Note: Steps 2 to 4 are repeated for every pixel of the image 64*128 pixels.

Step 5: Normalize Gradient using 2*2 cells per block:

Some areas or corners of the images are brighter/darker than the other and the Gradients of the image are very sensitive to such image lighting. Reduction of such lighting variations is done by taking blocks of 16*16 pixels iterated all over the image. One block of 16*16 pixels accommodates 4 cells of 8*8 pixels. Thus, one block has 4 of (9*1) feature i.e. (36*1) vector of features per block as shown below.

Normalization of each block is done by dividing all feature values of a vector by the 'square root of sum of squares' of all 36 values.

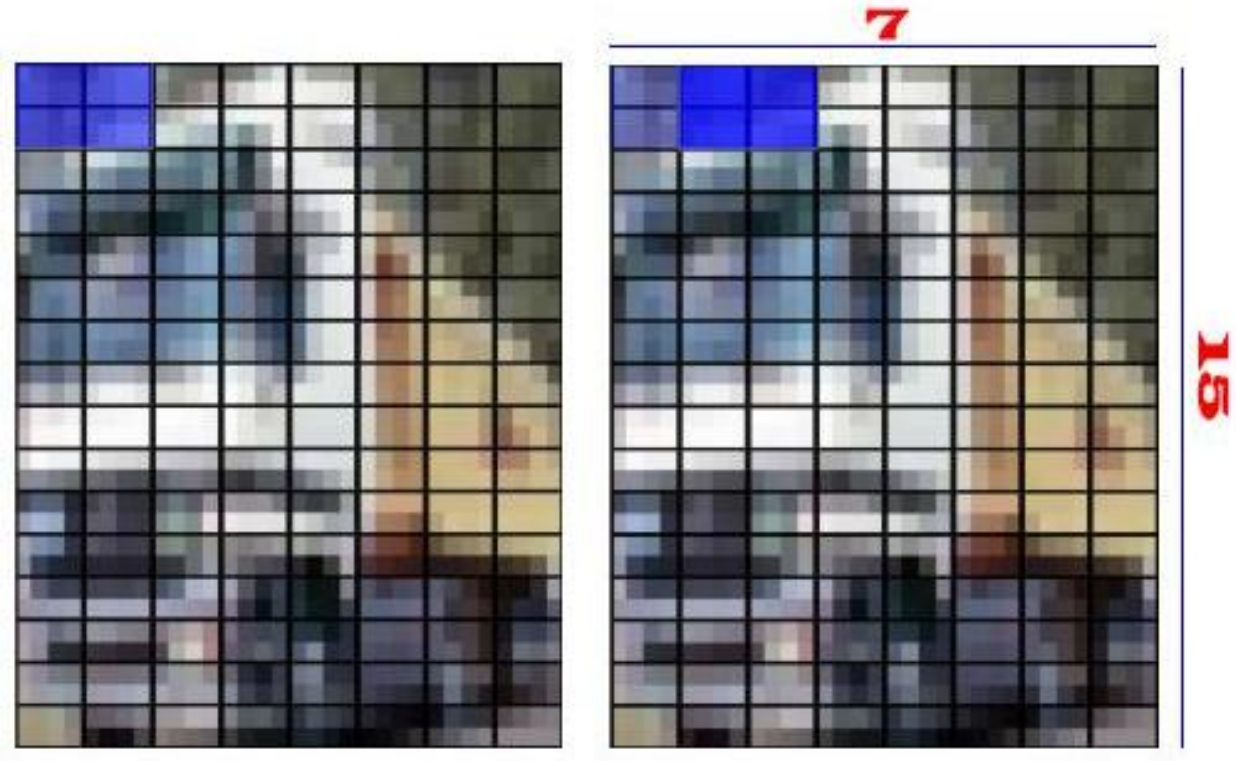

Figure 9 Normalization block covering complete image

Step 6: Feature for the complete image:

We would need 7 blocks of 16*16 pixels to cover 64 pixels width and 15 blocks of 16*16 pixels to cover 128 pixels height. Thus, features for the complete image will be given by 7*15 of (36*1) features which will result in 7*15*36 i.e. (3780*1) vector of features.

We applied HOG feature extraction on all images resulting to train features dataset of shape (50000*3780) and test features dataset of shape (10000*3780). Figure 10 shows examples of images and their HOG features.

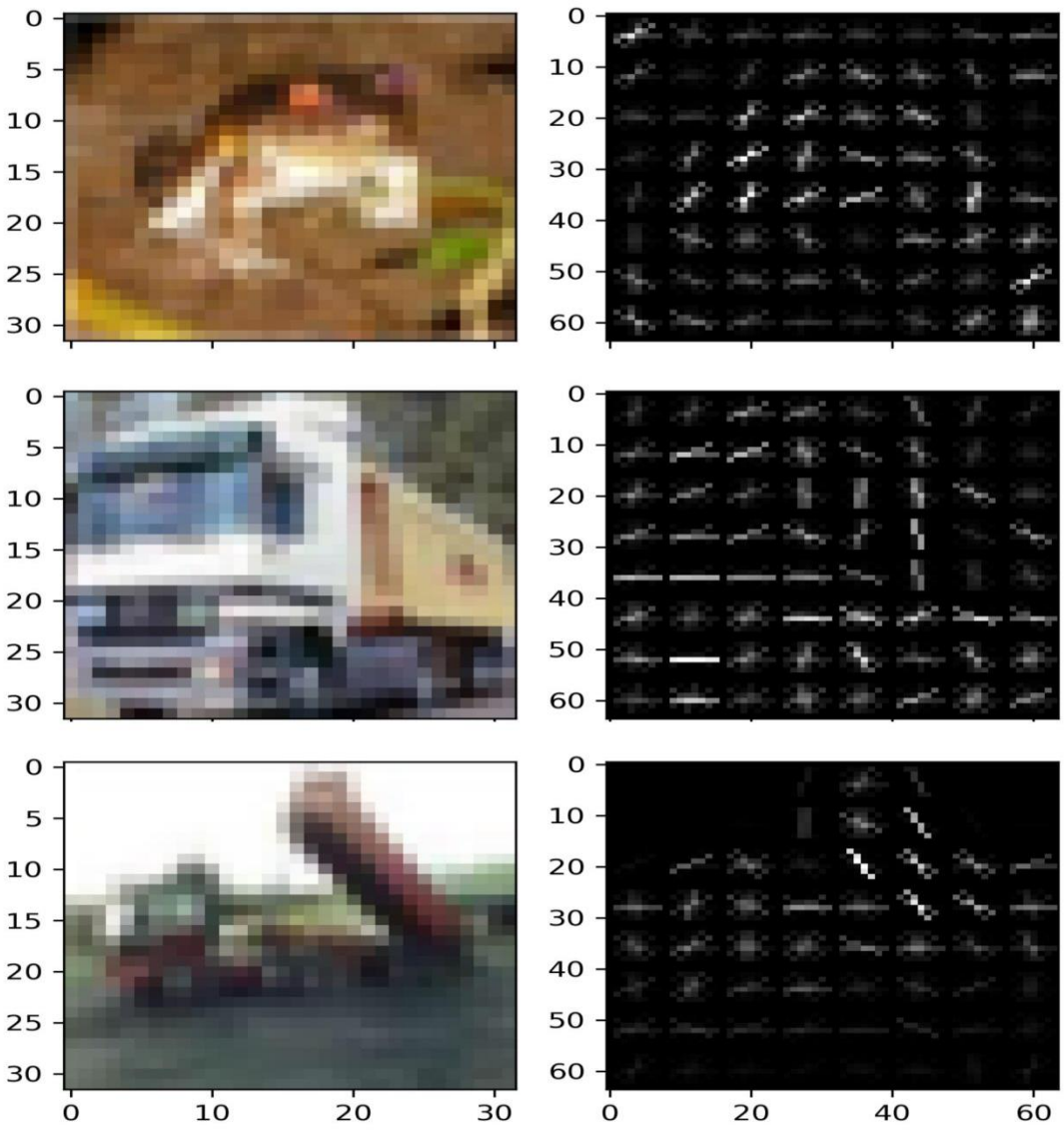

Figure 10 Examples of images and their HOG features

6. Traditional Algorithms on HOG features

The same approach adopted in Section 4 was adopted here. The difference here is that HOG feature extraction was done here before the principal component analysis. Figure 11 shows that 437 principal components can explain about 80% of the variance in the data. 437 is a large number of features and this poses significant challenges for the more computationally expensive algorithms.

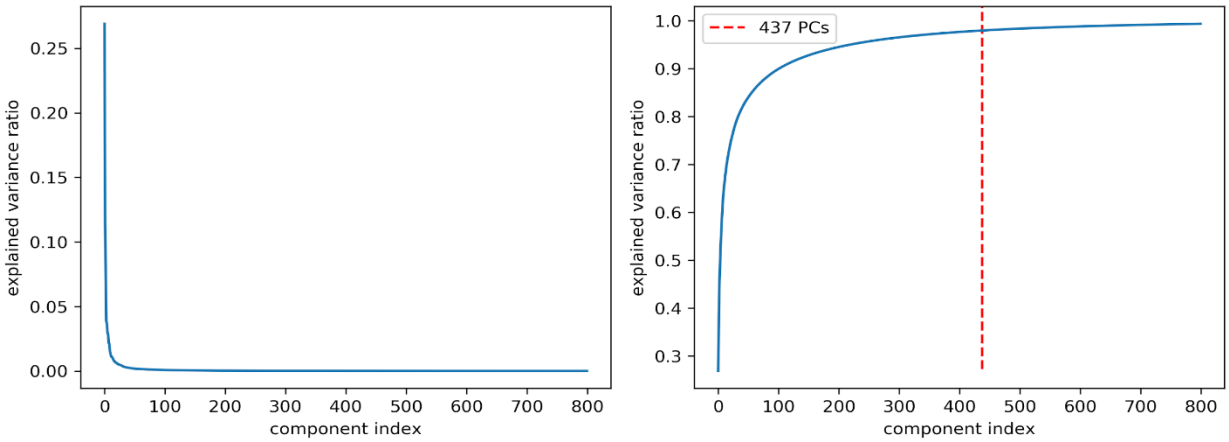

Figure 11 Principal Components Analysis plots

Modeling

Following the approach outlined in the Modeling part of section 4, attempts were made to fit the traditional models using default hyperparameter settings. While Naïve Bayes, K-Nearest Neighbors, Logistic Regression, and Random Forest were able to fit and produce models, the number of features were too much for Support Vector Machines and Gradient Boosting. Table 3 shows the results of testing these models with validation set data, as well as the time it took to train these models.

Table 3 Metrics for various basic classification models

Model	Accuracy	Precision	Recall	F1 Score	Time (s)
Naïve Bayes	0.49	0.48	0.48	0.48	0.298
KNN	0.60	0.16	0.13	0.16	0.168
Logistic Regression	0.54	0.54	0.54	0.54	31.90
Random Forest	0.43	0.43	0.43	0.43	110.00

From Table 3, we can see that the KNN and Random Forest models obtained here are poorer than was obtained without HOG feature extraction. It also must be noted that this comes at a higher computational expense than their counterparts in section 4. Naïve Bayes and Logistic Regression perform like some of the better models in section 4. So, it was decided to tune the hyperparameters for these models to obtain their best possible version.

Table 4 shows the metrics and the training time for the models. The Naïve Bayes shows significantly less performance than what was shown in Table 3. This is likely due to the fact that the previous model was overfit and under cross-validation, when it no longer overfit, it did not do so well. The Logistic Regression model shows good accuracy, but its Precision, Recall and F1-Score are less than what was obtained in section 3. The confusion matrices shown in Figures 12 – 13 also demonstrate this fact, as there are a lot of misclassifications.

Table 4 Metrics for various basic classification models

Model	Accuracy	Precision	Recall	F1 Score	Time (s)
Naïve Bayes	0.35	0.33	0.32	0.33	1.640
Logistic Regression	0.53	0.33	0.32	0.33	171.0

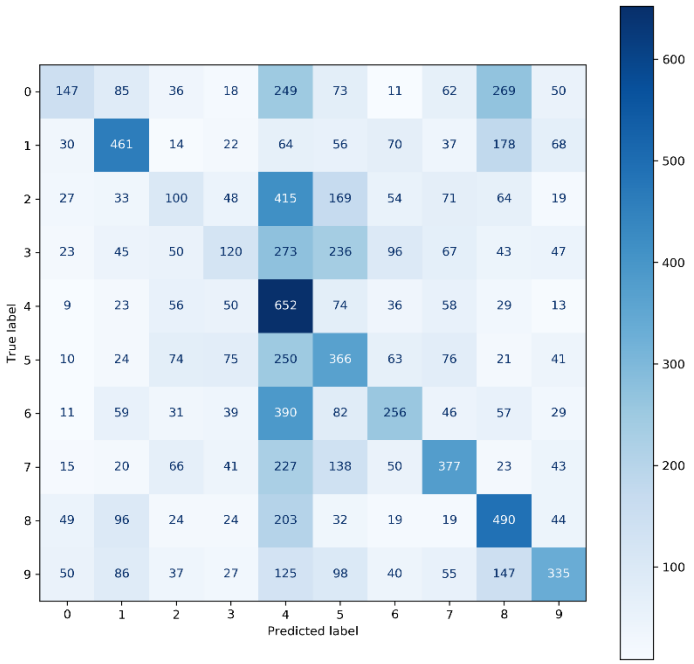

Figure 12 Confusion matrix for Naïve Bayes model

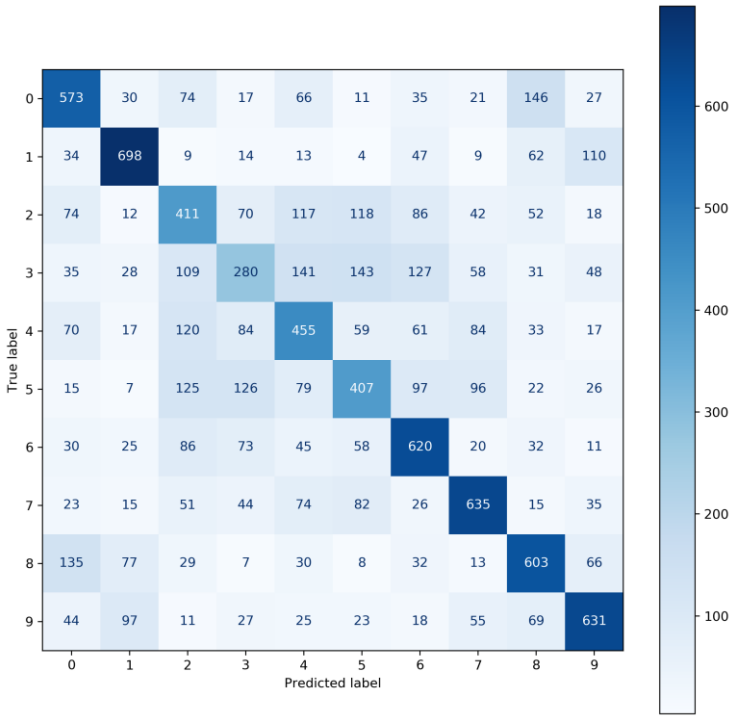

Figure 13 Confusion matrix for Logistic Regression model

7. Advanced Algorithms

We used advanced algorithms such as CNN which are proven to be very good at Image analysis.

I. CNN

We used the following 3-layer Convolutional Neural Network:

- Layer 1 – 32 filters of 3*3, same padding, relu activation, max pooling and 25% dropout
- Layer 2 – 64 filters of 3*3, same padding, relu activation, max pooling and 25% dropout
- Layer 3 – 128 filters of 3*3, same padding, relu activation, max pooling and 25% dropout
- Layer 4 – 512 dense, fully connected flattening layer with 50% dropout and softmax activation with 10 classes.

We initially trained the model on 20 epochs but as can be seen in figure 14, the validation accuracy plateaued after 12 epochs while training accuracy continued increasing indicating overfitting.

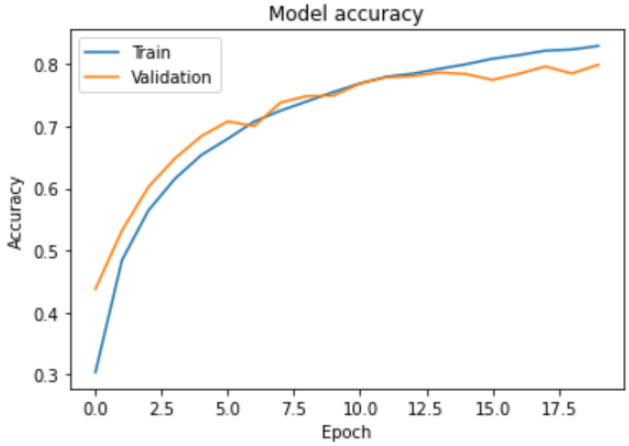

Figure 14 Learning curve of CNN trained on 20 epochs, indicates overfitting

Training the model on only 12 epochs and batch size 128 resulted in the learning curves shown in Figure 15.

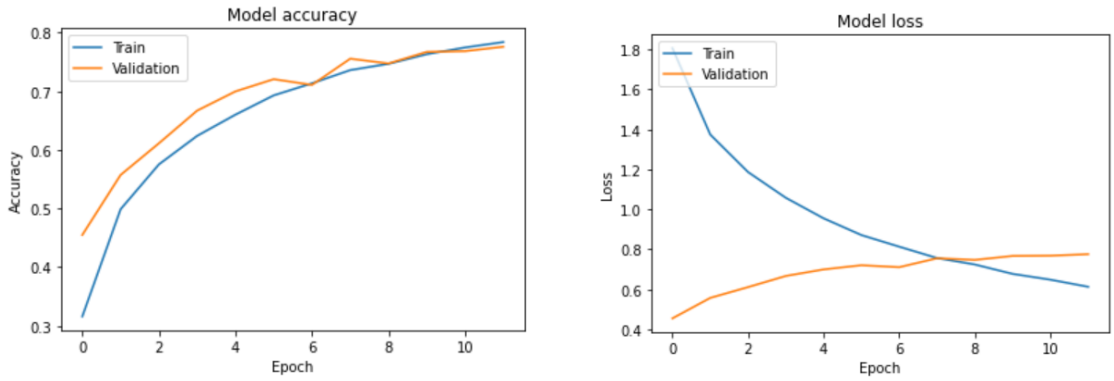

Figure 15 Learning and loss curve of CNN trained on 12 epochs

Summary of the model trained, parameters learned and untrainable parameters is given in Figure 16.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 32)	896
activation_1 (Activation)	(None, 32, 32, 32)	0
conv2d_2 (Conv2D)	(None, 30, 30, 32)	9248
activation_2 (Activation)	(None, 30, 30, 32)	0
max_pooling2d_1 (MaxPooling2	(None, 15, 15, 32)	0
dropout_1 (Dropout)	(None, 15, 15, 32)	0
conv2d_3 (Conv2D)	(None, 15, 15, 64)	18496
activation_3 (Activation)	(None, 15, 15, 64)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	36928
activation_4 (Activation)	(None, 13, 13, 64)	0
max_pooling2d_2 (MaxPooling2	(None, 6, 6, 64)	0
dropout_2 (Dropout)	(None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 6, 6, 128)	73856
activation_5 (Activation)	(None, 6, 6, 128)	0
conv2d_6 (Conv2D)	(None, 4, 4, 128)	147584
activation_6 (Activation)	(None, 4, 4, 128)	0
max_pooling2d_3 (MaxPooling2	(None, 2, 2, 128)	0
dropout_3 (Dropout)	(None, 2, 2, 128)	0
flatten_1 (Flatten)	(None, 512)	0
dense_1 (Dense)	(None, 512)	262656
activation_7 (Activation)	(None, 512)	0
dropout_4 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 10)	5130
activation_8 (Activation)	(None, 10)	0

Total params: 554,794
Trainable params: 554,794
Non-trainable params: 0

Figure 16 CNN model summary

Table 5 Metrics for CNN

Precision	Accuracy	Recall	F1 score
0.7824	0.7804	0.7844	0.7806

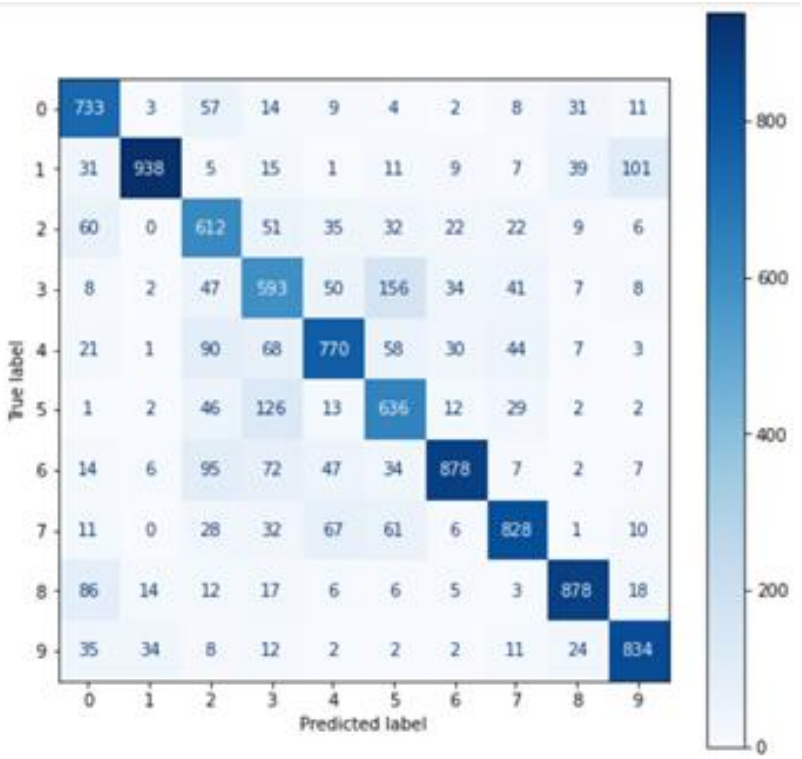

Figure 17 CNN Confusion Matrix

II. VGG16

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition". It consists of 16 convolution layers combined in 5 Conv-blocks. It has approx. 14.7M trainable parameters. Figure 17 shows the architecture of VGG16. It shows the 5 Conv-blocks.

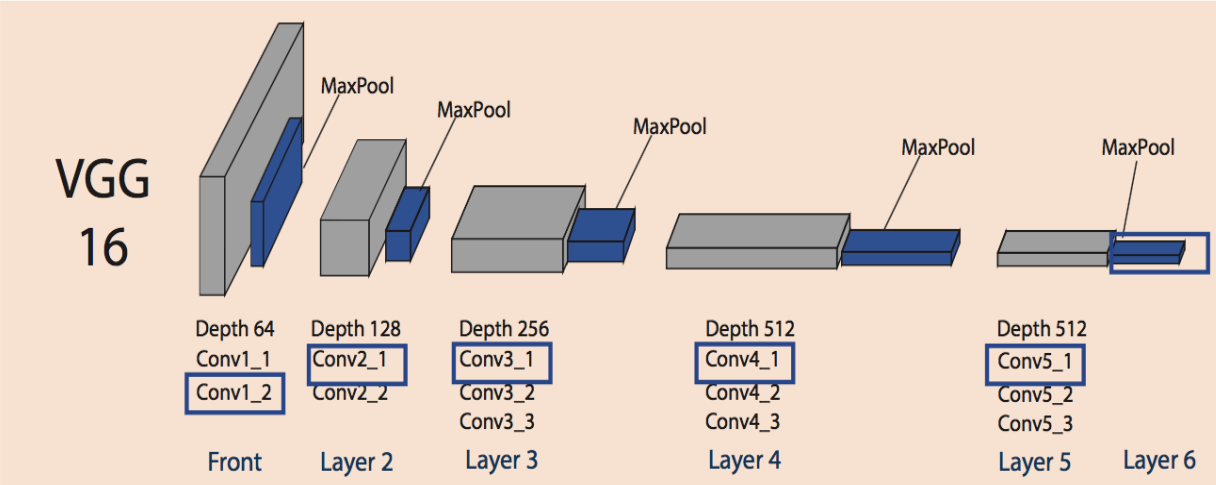

Figure 18 Architecture of VGG16

Methodology :

- Train the VGG16 with the existing weights (Pre-trained) on the data.
- Check for accuracy, recall, precision and f1-Score on the validation data.
- Take the trained model and check on Test data.
- Fine-Tuned the VGG16 by training the model from Block4_Conv1.
- Check for accuracy, recall, precision and f1-Score on the validation data.
- Take the Tuned trained model and check on Test data.

Data Pre-processing:

I started with reshaping the images from (32,32,3) to (96,96,3) of the training dataset. After that, I normalized all the images by 1/255. Then I split the training dataset in train and valid data. I used an 85-15 train-valid splitting ratio. Furthermore, I used data augmentation techniques on the train data. Figure 18 shows the summary of VGG16.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 96, 96, 3)         0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 96, 96, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 96, 96, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 48, 48, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 48, 48, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 48, 48, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 24, 24, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 24, 24, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 24, 24, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 24, 24, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 12, 12, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 12, 12, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 6, 6, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 6, 6, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 6, 6, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 6, 6, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 3, 3, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Figure 19 VGG16 Summary

Model Building and Analysis:

We downloaded the VGG16 model. The VGG16 model is pretrained. We wanted to check how well this pretrained model works on the CIFAR-10 data. So, we created a function to train this model on the data. We used a batch size of 128, 30 epochs to train the model. Adam optimizer and categorical cross-entropy were used as optimizer and loss function resp. The results after training are shown in Table 6 and Figure 20 shows the learning and loss curve for training & validation data of the pre-trained VGG16 model.

Table 6 Metrics for Pre-Trained VGG16

Data	Precision	Recall	Accuracy	F1-Score
Train	0.7860	0.5637	0.6979	0.6565
Validation	0.7863	0.5645	0.7527	0.6571
Test	0.7877	0.5677	0.7437	0.6598

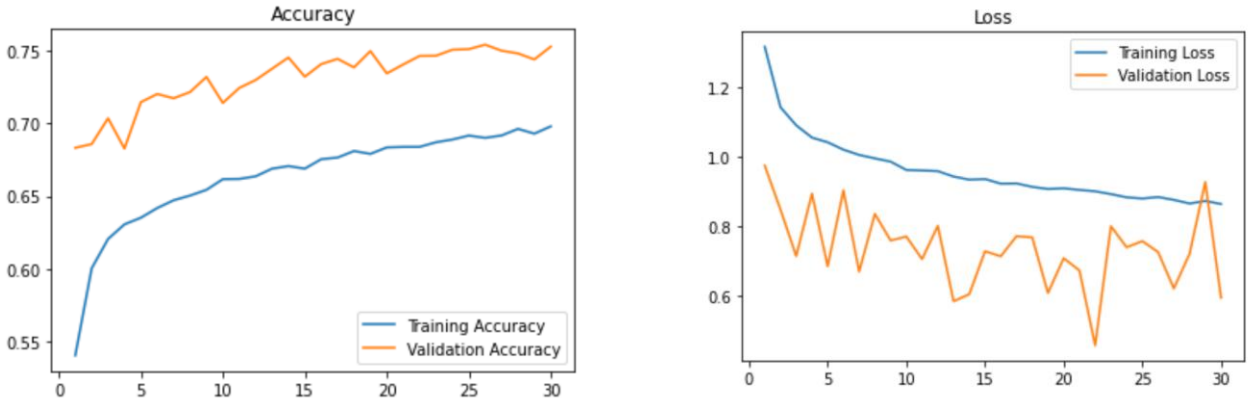

Figure 20 Learning and Loss curves of Pre-Trained VGG16

After we decided to fine tune the VGG16 since the accuracy on test data wasn't good enough. So, re-train the VGG16 model weights from Block4_Conv1. We used a batch size of 128, 50 epochs to train the model. Adam optimizer and categorical cross-entropy were used as optimizer and loss function resp. The results obtained after training are shown in Table 7.

Table 7 Metrics for Fine-Tuned VGG16

Data	Precision	Recall	Accuracy	F1-Score
Train	0.9231	0.8700	0.9663	0.8957
Validation	0.9235	0.8709	0.9181	0.8964
Test	0.9235	0.8710	0.9186	0.8964

Figure 21 shows the learning and loss curves for training & validation data of Fine Tuned VGG16 model, and Table 8 shows the metrics comparison between the Pre-trained and Fine Tuned VGG16.

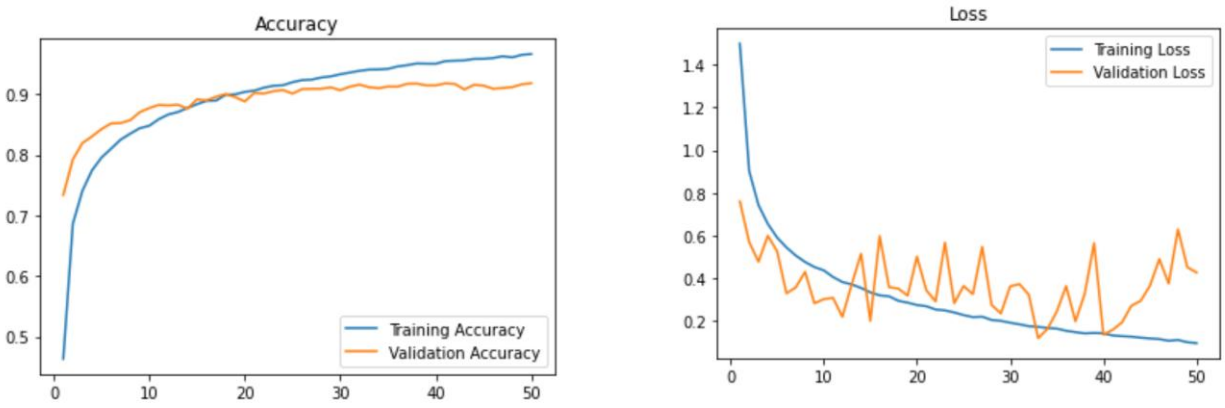

Figure 21 Learning and Loss curves of Fine-Tuned VGG16

Table 8 Metrics comparison for VGG16 on Test data

VGG16	Precision	Recall	Accuracy	F1-Score	Time(s)
Pre-Trained	0.7877	0.5677	0.7437	0.6598	4750
Fine Tuned	0.9235	0.8710	0.9186	0.8964	5000

8. Conclusion

Traditional machine learning methods did not perform well on the CIFAR-10 image data. We implemented HOG to check if this will improve the performance of traditional classification methods. But, based on this work, the HOG did not improve the performance of the traditional machine learning methods. However, that problem may be solved by tweaking the settings and parameters if one has the time. Also, its use is severely hampered by the fact that it requires a lot of features in order to be accurate. That makes it computationally expensive when it works, and impossible to use in some cases.

The Fine Tuned VGG-16 works the best on CIFAR-10 data. While, It gives the best result among all methods implemented, the time complexity of VGG-16 is higher than most of the methods.

We can certainly improve the performance of the VGG16 by retraining the whole VGG16 with more epochs and tweaking the hyperparameters such as learning rate, batch size.

9. References

- We referenced this VGG-16 implementation on Kaggle notebook of Ahmet Yasir Akbal on Kaggle.
- Jason Brownlee's tutorial on How to Develop a CNN from Scratch. https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/
- feature-engineering-images-introduction-hog-feature-descriptor