Image and Vision Computing: Analysis of Classical and Deep Learning approaches to Image Classification

Abstract

This study investigates the performance of classical machine learning and deep learning models in classifying images of sports balls, from 15 different categories. The dataset comprises over 9,000 images, testing the models' ability to distinguish similar objects under varying conditions. The classical approach combines Scale-Invariant Feature Transform and Histogram of Oriented Gradients features to train Support Vector Machine (SVM) models, while the deep learning approach utilizes a Convolutional Neural Network (CNN) based on the ResNet-18 architecture. The study tries to contrast the models' performance using metrics such as accuracy, and F1 scores, and confusion matrices. We also analyze the robustness of our models by inducing image perturbations such as noise, blur, and occlusion. While the classical approach provides insights into feature importance, the CNN model significantly outperforms the SVM in terms of accuracy and robustness. This research contributes to the broader understanding of image classification techniques, highlighting the advantages of deep learning models in handling complex, real-world data scenarios.

1. Introduction

In the rapidly evolving field of image classification, traditional techniques are increasingly being measured against modern deep-learning methods to assess their respective efficacies. This report addresses this comparison by focusing on a dataset comprising over 9,000 images across 15 distinct sports ball categories, including american football, baseball, basketball, billiard ball, bowling ball, cricket ball, football, golf ball, field hockey ball, hockey puck, rugby ball, shuttlecock, table tennis ball, tennis ball, and volleyball. Despite the creative alterations some images have undergone, this collection offers a solid foundation to evaluate different classification approaches.

A noteworthy aspect of this dataset is its inherent complexity and potential for misclassification, with some balls deliberately painted to resemble others, adding an intriguing layer of difficulty to the classification task. This aspect not only tests the classifiers' ability to generalize across varied representations of objects but also examines their robustness in handling deceptive appearances, a common real-world challenge. The dataset's division into training (7328 images) and test sets (1841 images), with an 80/20 split, provides a structured framework for model training and evaluation, setting the stage for a comprehensive comparison between a classical machine learning approach and a deep learning model.

The primary objective of this project is to assess and compare the sensitivities of the two approaches. On one side, we have a classical method employing feature extraction techniques like Histogram of Oriented Gradients (HOG)(Dalal & Triggs, 2005) and Scale-Invariant Feature Transform (SIFT) (Lowe, 1999) paired with a Support Vector Machine (SVM) classifier. On the other, a deep learning approach utilizes a Convolutional Neural Network (CNN), specifically the ResNet-18 architecture(He et al., 2015). By training these models to perform multi-class classification on the dataset, we aim to explore the subtle differences in model performance, especially in terms of F1 scores and accuracy, and explore their robustness against a series of image perturbations. This work highlights the strengths and limitations of each approach and contributes valuable insights into their applicability and robustness across diverse and complex image classification situations.

In the process of this evaluation, we have considered preliminary results that suggest significant differences in the sensitivities of the two methodologies to various image alterations. This insight leads us to a deeper exploration of how each model processes and interprets different features extracted from the images. For instance, the classical method's reliance on hand-crafted features like HOG and SIFT offers a direct, interpretable mechanism for classification, which exhibits distinct vulnerabilities to specific types of image perturbations. In contrast, the deep learning model, with its ability to autonomously learn features from data, demonstrates a different set of strengths and weaknesses, especially when faced with the same perturbations.

Our in-depth analysis helps us better understand when and why each method works well for complex image classification tasks. We gain valuable insights by examining the detailed relationships between the model design, the extracted features, and how robust the classifications are. These insights can guide the development of more flexible and effective image classification methods that can better handle real-world data challenges. We start the following section with the background of the algorithms we use to develop an understanding of the approach and then move towards the preprocessing, training design we chose and a discussion about the results of our experiments.

2. Methodology

We broadly classify the methodologies we use into two segments i.e., the Classical and the Deep Learning Algorithm.

2.1. Classical Algorithm

The classical approach to image classification combines SIFT features and HOG features to train SVM models for classifying images into different sports ball categories. The details of this training design are discussed in Section 3.3 This method classifies images into different categories of sports balls by extracting features, reducing dimensionality, and performing classification.

2.1.1. Scale-Invariant Feature Transform (SIFT)

SIFT is a feature detection algorithm used to identify and describe local features in images(Lowe, 1999),(Lowe, 2004). It excels at extracting features that remain consistent despite changes in image scale, rotation, and to some extent, illumination and 3D perspective. This resilience makes SIFT exceptionally reliable for tasks requiring recognition across varied conditions. In practice, SIFT identifies keypoints within an image, and generates descriptors for each, encapsulating the local gradient patterns. These descriptors serve as distinctive markers of local shapes and textures, allowing for the accurate recognition of objects, such as sports balls, which may appear in differing sizes or orientations. This attribute of SIFT features is indispensable for recognizing and distinguishing between the various sports balls within the dataset, demonstrating its utility in ensuring robust object recognition.

2.1.2. Histogram of Oriented Gradients (HOG)

HOG is a feature descriptor used for object detection in computer vision and image processing(Dalal & Triggs, 2005). The implementation of HOG feature extraction focuses on calculating image gradients in localized sections to encode the directionality of edges. This process starts by converting images to grayscale to emphasize texture and shape over color, which is essential for distinguishing sports balls by their physical characteristics. These gradients are then compiled into a histogram for each region, encapsulating the edge orientations in compact feature vectors. This technique is particularly effective at capturing the unique shape and appearance of objects, such as the various sports balls in our dataset, by providing a detailed representation of form and structure, making it a powerful tool for object detection in images. In this study, the HOG features were extracted to capture edge and gradient information, which are indicative of shape and texture, providing complementary information to SIFT descriptors.

2.1.3. Support Vector Machines (SVM)

Developed by Vapnik and Chervonenkis in 1960, SVM is a powerful supervised machine learning algorithm widely used for classification and regression tasks(Cortes & Vapnik, 1995). It works by finding the hyperplane that best divides a dataset into classes in the feature space. For non-linearly separable data, SVM uses the kernel trick to transform the input space into a higher-dimensional space where it is possible to find a linear separator. In this methodology, SVM is employed with the Radial Basis Function (RBF) kernel to classify images based on their extracted features, including SIFT and HOG descriptors. We meticulously trained three distinct SVM models to evaluate the effectiveness of our feature extraction techniques:

- **SIFT Model**: This model is trained on histograms derived from SIFT descriptors, effectively using keypoint-based features for recognition.
- **HOG Model**: It utilizes Principal Component Analysis (PCA)-reduced HOG features, focusing on the shape information of objects (Wold et al., 1987).
- **Combined Model**: By concatenating PCA-reduced HOG features with SIFT histograms, this model harnesses both structural and textural information for enhanced classification accuracy.

The entire approach and the training setup is defined in 3.3. Through the strategic use of an RBF kernel, our SVM classifier skillfully manages the nonlinear relationships between image features and their classes. This approach not only leverages the distinct advantages of SIFT and HOG features but also significantly boosts the classifier's ability to distinguish among the varied classes of sports balls within our dataset, showcasing the robustness and adaptability of SVM in handling high-dimensional data for accurate object recognition.

2.2. Deep Learning Algorithm

As discussed in the previous section, the approaches to feature extraction for image classification were primarily dependent on handcrafted features like HOG, SIFT, etc. These features were then fed to shallow classifiers like SVM(Cortes & Vapnik, 1995), random forests(Breiman, 2001), etc. These approaches struggle when the dimensions in the data are extremely high, such as in images with dimensions 512×512, which, when flattened, will have 262144 pixels. Simple Feed-Forward Neural Networks (FFNN) dealt with this high dimensional data much better. However, in the context of image data and the problem statement of Image Classification, even FFNN seemed very inefficient for the following reasons -

- **Lacking Spatial understanding**: FFNN would flatten the image data from 2-D to a long vector. Thus, the spatial relationship between the two pixels gets lost (Egmont-Petersen et al., 2002).
- **Difficulty capturing translation invariant features**: FFNNs are sensitive to variations in the scale, orientation, etc. Flipping an image by 90 degrees can create difficulty in accurately predicting the correct class.

- **Parameter Inefficient**: FFNN does not provide any provision to share parameters. Each neuron in each layer would be connected to every neuron in the next layer, thus giving extremely high learnable parameters, making it very slow.

2.2.1. Overview of Convolutional Neural Network

In order to circumvent the above-mentioned problems, Lecun et al. (LeCun et al., 1998) proposed the CNN with Lenet architecture. Earlier attempts to use CNN with multiple layers did not gain much traction due to limited computing resources and data. However, Krizhevsky et al. in 2012 proposed AlexNet(Krizhevsky et al., 2012), which had five convolution layers and significantly outperformed previous models on ImageNet(Deng et al.). Post this, multiple other models like VGGNet(Simonyan & Zisserman, 2015), GoogLeNet(Szegedy et al., 2014), ResNet(He et al., 2015), etc., all leveraged the CNN architectures with incremental updates to the architecture for solving the problem of image classification.

The individual components of any CNN architecture are as follows -

- **Convolution Layer**

  The primary purpose of the convolution layer is to extract features from the images fed to the model. Convolution is a mathematical operation involving sliding a small kernel or filter over the input image. The parameters of this kernel are kept to be trainable and get updates as the model trains.

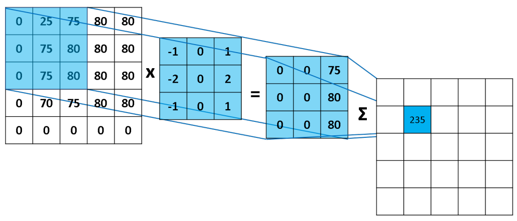


Figure 1. Convolution Operation

  In essence, each time we slide the kernel over the segment of input image, we do an addition of element-wise multiplication between the kernel and the image. This process helps capture the local features present in the data i.e., edges, curves, etc. At every step of convolution, we use multiple such kernel filters stacked on top of one another.

- **Stride**

  A stride is a number that defines the number of pixels by which we move the filter across the input data at any step, i.e., if the stride is set to 2, the kernel filter slides over the input image at a difference of two pixels.

Figure 1 from https://mlnotebook.github.io/post/CNN1/

- **Padding**

  Padding means adding extra rows and columns initiated with some value (usually zero). The reasoning for doing this is often to preserve the spatial information primarily because if the filter size is big at the edges of an image, information can get lost as the pixels at the edges will be used only once with the filter for the element-wise multiplication. Additionally, padding also helps in preserving the input dimensions.

- **Non-linearity**

  If non-linearities are not introduced in a neural network, the deep model would collapse into a combination of linear functions. Thus, we induce non-linearity in the model using activation functions such as Rectified Linear Unit(ReLU) (Agarap, 2019).

- **Pooling Layer**

  These are primarily used to reduce the number of parameters by downsampling the input to the pooling layer. Multiple types of pooling, such as Max Pooling, Average Pooling, etc., exist. Based on the window size we define, pooling would do the operation specified based on the elements of the window. For instance, a max pool would take the maximum element from the window, and an average pool would take the average of elements in the window.

- **Fully Connected (FC) Layer**

  Fully Connected (FC) layers are usually the last layer of a CNN dealing with the image classification task. At this stage, we flatten the matrix into a vector and feed this vector to a normal FFNN. The number of neurons in the last FC layer equals the number of output classes in our classification problem. We use a Softmax layer to convert the output of this layer to a probability distribution to get the final output class predicted by the model

2.2.2. Optimization and evaluation for CNN

Optimizing the CNN model, i.e., weight updation of the model, is done using the backpropagation algorithm(Rumelhart et al., 1986). Backpropagation allows the gradients to make changes to the weights with the help of optimizers such as stochastic gradient descent (SGD)(Ruder, 2016), RMSProp(Tieleman & Hinton, 2012), Adam(Kingma & Ba, 2014), etc. We use cross-entropy loss as the loss function to train the CNN model for the image classification task. The equation for the loss function is given below -

$$H(p,q) = -\sum_{x \in \text{classes}} p(x) \log q(x) \tag{1}$$

Where p and q are the actual and predicted model distribution. The primary evaluation of the model for image classification is usually accuracy. Some other evaluation metrics, such as the F1 score and confusion matrix, can be much more descriptive than accuracy when we want to study classes individually. Data imbalance can cause accuracy to be an extremely biased metric; hence, the F1-score can be an effective measure to monitor and compare models on some occasions.

2.2.3. ResNet

For our analysis, we use the ResNet architecture first proposed by He et al. (He et al., 2015). In addition to the elements of CNN architecture defined above, ResNet introduced a skip connection, which made it easy to train deep CNN networks.

Reason for using the skip connection

Skip connections are also known as identity mappings because of how they are designed. With deeper models, we often run into problems related to gradients vanishing as the error propagates from deeper to shallower layers. The problem is that the gradients, in a very deep network, on consecutive multiplication lead to diminishing values. Due to this, though the network becomes deep, thus increasing the complexity of the model to learn, the capacity to learn is inhibited by weights getting close to no updates. Skip connections provide an alternate path for the gradient to flow during backpropagation, thus alleviating the dying gradients.

The reason to call skip connection an identity mapping is because the input to a layer is simply added to the output of the layer, thus preserving the original information. Due to this addition, for each layer that is subjected to a skip connection, the network is encouraged to learn a residual mapping that represents the difference between the input and expected output of the layer.

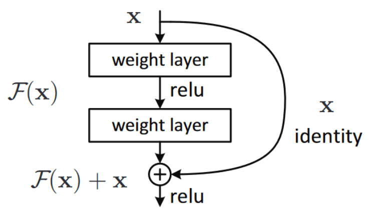

x
weight layer
relu
$\mathcal{F}(\mathbf{x})$
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2. Skip connection implementation

ResNet-18 architecture details

The input for our ResNet model is 512×512. The first convolution layer of the model applies 64 filters of dimension 7×7. A batch normalization layer is used to normalize the input tensors along the dimensions of the batch and is followed by a 2×2 Max Pool layer with stride set as 2. The ResNet-18 has four residual blocks, each consisting of multiple basic blocks (2 for resent-18), each having two convolution layers with 3×3 filters. The number of filters in the convolution layer gradually increases from 64 to 512 in the multiplication of 2 for each residual block as we progress down the residual blocks.

Figure 2 from https://theaisummer.com/skip-connections/

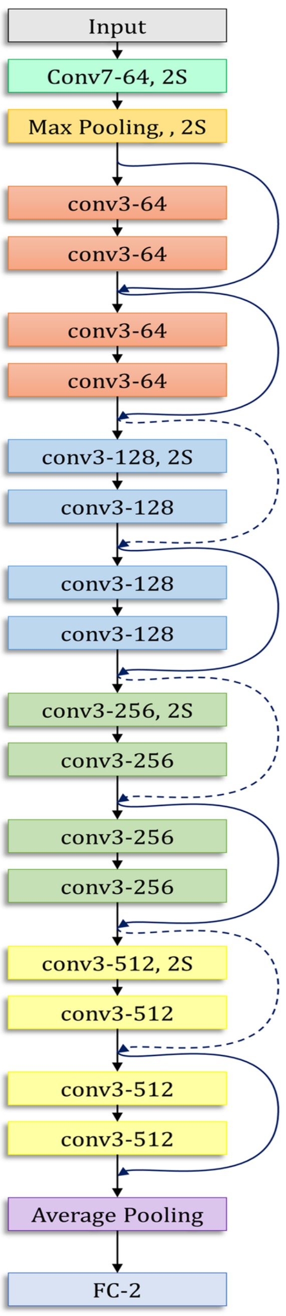


Figure 3. ResNet-18 architecture used(Kundu et al., 2021)

The residual block can be further segregated as follows -

1. Convolution block

The convolution block comprises of two convolutions, but the first convolution layer in the block is used to downsize the feature map as it takes a stride of 2 into account. A batch normalization(Ioffe & Szegedy, 2015) follows each convolution layer, and the first convolution is also followed by an activation function, which in our case is ReLU(Agarap, 2019). The output of the second convolution layer, though, needs to be fused with the input to the convolution block to form a skip connection. However, the shape of the two is different, i.e., the output is downsized by half from the start of the residual block to the end because of the first convolution. Thus, to form a skip connection, we downsample the input to the convolution block using convolution kernels of 1×1 with stride two, and the number of filters is equal to the channels of the output of the second convolution, which makes the dimensions of the two layers the same so we can form the skip connection. These skip connections are represented by the dotted line in Figure 3.

2. Identity block

In the case of identity blocks, the second component of a residual block, we simply perform two convolutions, each followed by a batch normalization using 3×3 filters. The output of the first convolution is passed through a ReLU function. Since we keep the strides to be 1 for both convolutions, we effectively never downsize the feature map, and hence, a skip connection in the identity block is nothing but an addition operation of the input to the identity block and the output after the second batch normalization.

The output of both the convolution and the identity block are put through the ReLU activation to further induce non-linearities in our model. The skip connections discussed above give alternate routes to propagate the gradients back, thus circumventing the vanishing gradient problem.

3. Training design

3.1. Data Preprocessing

Since we are trying to contrast and study the performances of the classical and deep learning models, it was logical to subject both models through similar preprocessing steps. We make use of the original dataset, which has varying image sizes. We process the images to transform all of them to dimensions of 512×512. This brings uniformity to the images being fed to both models. We also do a train-validation split of 0.9 for the deep model.

3.2. Data Augmentation

Data augmentation is an important step in making the classifier model more robust. In augmentation, we introduce variability in our training data with operations such as translation, scaling, rotation, etc. This helps expose the model to a multitude of scenarios during training so that at inference time, it can make better classifications. It can also help deal with the problem of overfitting in deep models.

For our case, we introduce the following augmentation operations to our training data -

1. Random rotation: We introduce a random rotation in our image as an augmentation to the data. We do it by a factor of ± 10 degrees.

2. Random Horizontal and Vertical Flip: Horizontal and vertical flips add robustness to the model during training time. The probability of both the flips is kept at 0.5.

3. Random Translation: This affine transformation controls the random translation of an image along the horizontal and vertical axes. We set this value to be 0.1 on both axes, meaning the image can be translated by up to 10 percent.

4. Scaling: We incorporate the scaling transformation to randomly scale the image between the range of 0.8 to 1.2 so that our model does not take a hit at inference time, even if an image is zoomed in or zoomed out.

3.3. Training setup for classical model

Our image classification approach involved multiple steps for feature extraction, dimensionality reduction, and classification. The first step was addressing the variable-length SIFT descriptors, which are not directly compatible with conventional classifiers due to their varying sizes across images. To resolve this, we applied K-Means clustering(Hartigan & Wong, 1979), which summarized the local features into fixed-size vectors of length 100. This process was essential as it enabled us to standardize the feature set for each image, focusing on the most prevalent patterns and thus allowing our classification models to process the data uniformly.

We applied PCA to our HOG features due to their high dimensionality(Wold et al., 1987). Such high-dimensional data can potentially lead to the curse of dimensionality, where the feature space becomes so vast that the available data becomes sparse, making it difficult for classifiers to learn effectively. PCA helped mitigate this issue by reducing the dimensionality of the HOG features while retaining the most informative components critical for maintaining the model's performance. It also made it more manageable and less prone to overfitting.

For the classification process, we opted for a Support Vector Classifier (SVC) with an RBF kernel. This choice was driven by the SVC's proven effectiveness in handling high-dimensional spaces and its flexibility in capturing complex, non-linear decision boundaries, which are often present in image classification tasks. The regularization parameter (C) should be chosen such that it indicates a moderate emphasis on classifying the training data correctly while still allowing for some generalization. This balance is crucial in complex tasks like this, where the diversity of the images requires a model that can accurately classify seen examples but also generalize well to new, unseen images. To fine-tune our model to achieve optimal performance, we employed Grid Search for hyperparameter optimization. This exhaustive method ensured that we explored all possible combinations of parameters to find the best settings for the SVC, thereby maximizing the model's effectiveness. The best set of hyperparameter comes out to be $C = 10$, $kernel = rbf$.

The strategic integration of SIFT and HOG features, combined with dimensionality reduction through PCA and the refinement provided by K-Means clustering, encapsulated a broad spectrum of information, from local textures and shapes to global patterns. This approach and meticulous hyperparameter tuning via Grid Search resulted in a robust and accurate classification system with improved accuracy and F1 scores, ensuring predictive accuracy and generalizability on unseen data. Refer to Appendix C.2 for implementation.

3.4. Training setup for Deep Learning model

For the training of our deep learning model, we take the augmented data and perform transfer learning using the architecture details specified in Section 2.2.3. We load the pre-trained weights of the ResNet-18 model and modify the number of neurons in the last fully connected layer. The last layer in our model becomes a vector of size 15, which is the total number of classes in our training set. The reason to keep all the weights trainable is because since we already have a good enough estimate of the initial layers since the model has been trained on ImageNet(Deng et al.). We still want the model to learn the intricacies related to our training data further. For instance, curves in our dataset would be of great importance as they would help differentiate multiple classes like rugby ball, tennis ball, shuttle cock, hockey puck, etc.

We use nn.CrossEntropyLoss as our functionality for loss function. The softmax on the output of the last fully connected layer is incorporated into the nn.CrossEntropyLoss in pytorch. We tried to experiment with various learning rates using Adam optimizer to train our model. The initial learning rate was kept at 1e-3, which showed that the learning rate was too high as the loss and validation accuracy never stabilized but kept oscillating. We then gradually varied the learning rate and found the best results at 1e-4 as the learning rate. The training experiments' graphs are shown in Figure 4. As we can see from the figure, the training is much more stable, and the validation performance is significantly better for learning rate 1e-4. The total number of trainable parameters for our ResNet-18 model comes out to be 11184207. The model is trained for ten epochs, after which its performance stops yielding improvements on the validation set. Refer to Appendix C.3 for implementation.

4. Results

The results are discussed for both the SVC and the ResNet model along with the effect of perturbations on the models.

4.1. Results for SVC model

4.1.1. SIFT Model Performance

The SIFT model demonstrated an overall accuracy of 47.31% and a weighted F1 score of 47.01%. The model performed best in classifying baseball images with a precision of 0.74 and an F1-score of 0.72. Conversely, the lowest precision was observed in classifying hockey puck images at 0.31, with an F1-score of 0.35. This indicates a moderate level of effectiveness in distinguishing between different types of sports balls, with specific challenges in identifying certain categories accurately. This variability in performance across categories can be attributed to inherent limitations associated with the model's design. The invariance features of SIFT, particularly its scale and rotation invariance, though beneficial for maintaining consistency across varied image conditions, might inadvertently encapsulate background features. This inclusion of non-discriminatory elements could dilute the distinctiveness of the model's feature descriptors, especially for sports balls with smooth textures or minimal edge detail. Furthermore, SIFT's methodology for generating distinctive descriptions of keypoints within an image may not be as effective for objects that lack pronounced edges or exhibit uniform textures, thus impacting the model's precision in accurately categorizing certain sports balls.

4.1.2. HOG Model Performance

The HOG model showed an improved performance with an accuracy of 51.17% and a weighted F1 score of 51.09%. This model exhibited its highest precision of 0.70 and F1-score of 0.70 in classifying rugby ball images. The lowest precision was for baseball images at 0.35, with an F1-score of 0.33. This variation in effectiveness across different classes highlights the model's strengths and weaknesses. The HOG descriptor's emphasis on edge and gradient information renders it particularly suitable for objects that exhibit well-defined shapes and pronounced textures, thus explaining its success with rugby balls. Conversely, the model's sensitivity to positional changes poses challenges in accurately classifying images where balls are partially occluded or presented in varied positions, impacting its ability to maintain consistent performance across all sports ball categories. However, the HOG model's performance suggests a better capability in feature representation compared to the SIFT model, particularly for certain classes.

4.1.3. Combined Model Performance

The combined model, integrating both SIFT and HOG features, significantly outperformed the individual models, achieving an accuracy of 61.49% and a weighted F1 score of 61.36%. This model showed a balanced performance across most classes, with the highest precision of 0.74 and F1-score of 0.72 in classifying baseball images as shown in Appendix A. The improvement in the combined model underscores the complementary nature of SIFT and HOG features in capturing the visual essence of the sports balls, leading to more accurate classification results. The combined model's advantages lie in its ability to bring together the distinct strengths of both feature sets—SIFT's invariance to scale and rotation with HOG's emphasis on edge and gradient details—thereby improving the overall texture and shape depiction. This comprehensive feature integration not only boosts the model's discriminability among visually similar classes but also facilitates a more accurate and balanced performance across the diverse array of sports ball categories, underscoring the efficacy of combining multiple feature descriptors for complex classification tasks.

The results showcasing the performance of an SVM classifier, when combined with different feature extraction

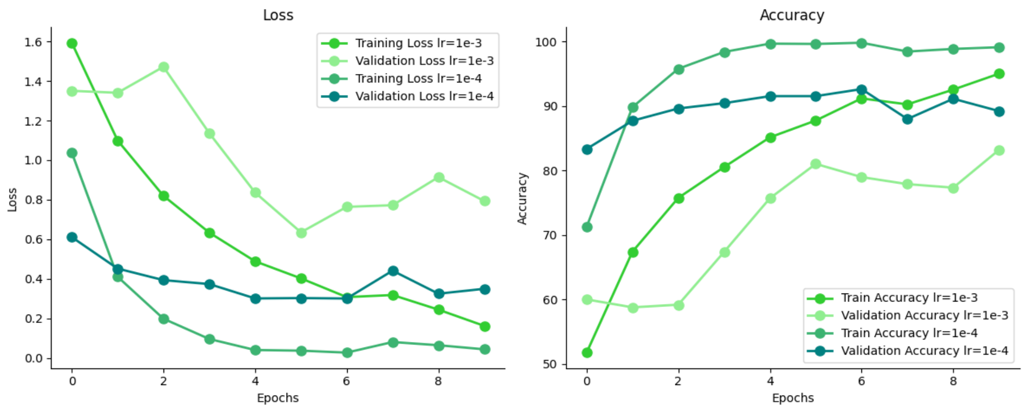


Figure 4. Training and Validation plot of loss and accuracy with learning rate 1e-3 and 1e-4

techniques for classifying images of sports balls, highlight the unique capabilities and contributions of each method. Specifically, the application of the SVM classifier with SIFT, HOG, and a combination of both techniques provides insight into how these approaches impact image classification tasks. While the SIFT model provides a foundational understanding of the image content, the HOG model enhances the detection capabilities by emphasizing edge and gradient information. The combined model leverages the strengths of both SIFT and HOG features, resulting in a superior classification performance.

4.2. Results for ResNet model

The ResNet model with the data augmentation explained in Section 3.4 reaches accuracy bounds of 90.71% and F1-score of 0.91. As mentioned above, we train a ResNet-18 model with a learning rate of 0.001, and the model reaches the best test accuracy of around 79%. On changing and varying learning rates, we encounter the most stable learning with a rate of 0.0001.

The confusion matrix and the entire classification report of this model are mentioned in Appendix B. The F1-score of the classes 'cricket_ball' and 'football' is the highest, reaching 0.94. The worst performing class for the model is 'hockey_ball', reaching an F1-score of 0.83. The class 'hockey_ball' is majorly misclassified with the 'golf_ball', which is understandable since both balls are white and round, and on many occasions, both balls have small depressions on their surface.

Another advantage of using CNN models is we can analyze where the model is trying to focus while making a prediction. This can be done using Gradient-weighted Class Activation Mapping (Grad-CAM)(Selvaraju et al., 2017) visualizations. In Grad-CAM, we feed the input image through our model, compute the gradients of the predicted class score with respect to the feature maps in the convolution layer, and perform a global averaging of those gradients. Once this is done, we take a weighted average using these averages and our feature maps to generate a heatmap that can be overlaid on the input image to see what parts of the image played a role in the classification of the image to that particular class. The Grad-CAM visualization for some of the images is shown in Figure 5. As we can see clearly, in the case of the tennis ball, shuttlecock, and billiards ball, multiple such objects are present in the image, and the model is indeed getting spikes in the heatmap on all of those objects. Additionally, in the case of baseball, it focuses on the stitching of the ball as that is a unique identifier that can help distinguish it from the rest of the balls.

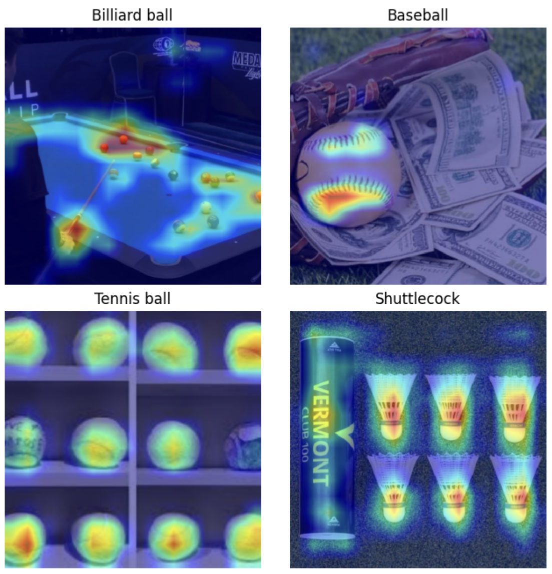


Figure 5. Grad-CAM visualization for sample test images

	Baseline SVC	Implemented SVC	Baseline ResNet	Implemented ResNet
Accuracy	10	61.49	75	**90.71**

Table 1. Accuracy comparisons with baseline

We also experiment with the robustness of the model with various perturbations applied to the test data. The ResNet model fares well in most perturbations because it tries to generate features independently rather than handcrafting them. The details of those are discussed in the following section.

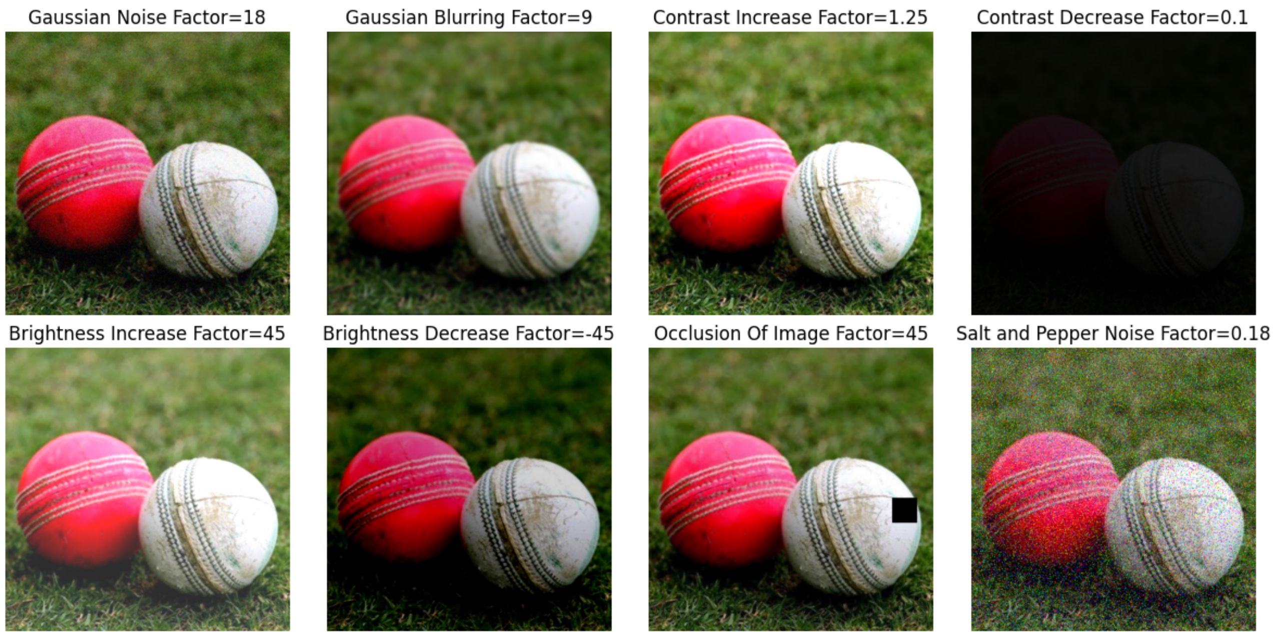

Gaussian Noise Factor=18
Gaussian Blurring Factor=9
Contrast Increase Factor=1.25
Contrast Decrease Factor=0.1
Brightness Increase Factor=45
Brightness Decrease Factor=-45
Occlusion Of Image Factor=45
Salt and Pepper Noise Factor=0.18

Figure 6. Maximum perturbations applied to a sample test image

4.3. Robustness Exploration

In order to check the robustness of our classical and deep learning classification model, we subject our test images to perturbations related to contrast, brightness, noise, and blurring. After all the perturbations we apply, we clip the image back to the range of (0,255) to prevent overflow or underflow. The outcome of the maximum perturbation of all the perturbations we consider is applied on a sample image as shown in Figure 6. The details of how the perturbations are introduced is in Appendix C.4, and the resultant outcome of the models is discussed below -

4.3.1. Gaussian Pixel Noise

We use NumPy to generate random numbers from a normal distribution map of the same size as the input image based on mean zero and increasing standard deviations. This noise is then added to the original image to generate a noisy image, which is then fed to the testing for both models. The effect of this noise becomes very evident in the case of the classical model right as we induce a noise with $\sigma = 2$ as the classical model works using handcrafted features and is dependent on gradients at any pixel. If noise is induced, the model is likely to drop in performance.

The deep learning model, though, learns features on its own from the data and thus is less susceptible to noise of small magnitude. The results only start to take a hit, and the accuracy drops by approximately 10% as we gradually increase the σ.

4.3.2. Gaussian Blur

In order to induce this perturbation, we use a standard Gaussian kernel and repeatedly convolve it over the image using Conv2D operation from a torch. This filter is applied to all the image channels, and we do it sequentially from 0 to 9 times, logging accuracy output at each step.

Gaussian blurring tries to smoothen out the fine details and edges present in the image, which degrades the ability to discriminate the handcrafted features, resulting in a stable accuracy drop from 61.49% to 33.51% for the classical model. The ResNet model, though, is robust to spatial variations as it learns features hierarchically; thus, we barely see a considerable drop in its accuracy. If the Gaussian kernel used to blur the images is increased in dimensions, we might see a drop in the performance of the ResNet model, and the increase in size would cause aggressive image blurring. The solution to this can be to deblur the image before feeding it to prediction using inverse filters, wiener filters, etc.

4.3.3. Image Contrast change

Contrast change in an image can simply be brought about by multiplying it by a factor specified. We introduce both contrast increase and contrast decrease perturbation, but the function to induce the perturbation remains the same as the operation would just mean multiplying the image by that factor and then clamping it in the range of (0,255). Both our models show robustness when subjected to minor changes in contrast. This is because both SIFT and HOG are designed to extract local features which are invariant to small changes in contrast. The ResNet model is less susceptible to minor variations in contrast as we use batch normalization(Ioffe & Szegedy, 2015), which helps adapt to such changes. While decreasing the contrast, we decrease it drastically to the point that both models struggle to identify the object properly in the image. At contrast factor of 0.1, even a human eye would find it difficult to explain the object in image as shown in Figure 6. However, we can

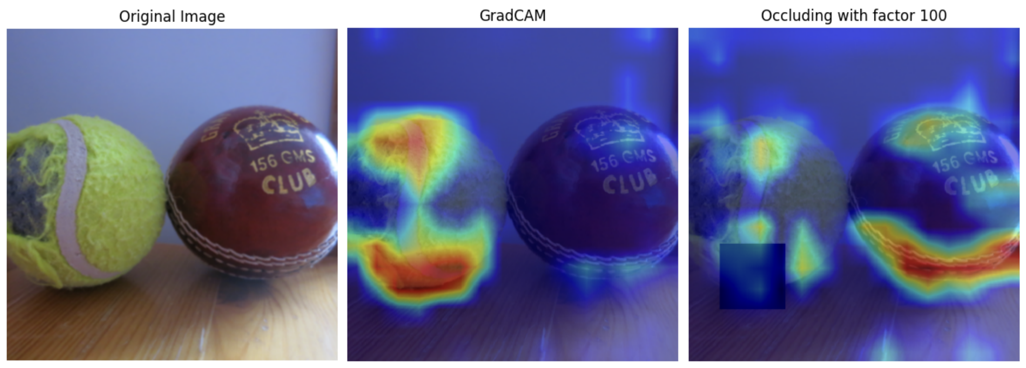


Figure 7. Grad-CAM variation with occlusion increase

observe that even with moderate contrast changes, both models work decently, and only when the contrast change factor is below 0.5 do we see an acute fall in the accuracy of both models.

4.3.4. Image Brightness Change

The change in brightness of an image is nothing but an addition or subtraction of pixel values from all pixel intensities in the image. The brightness change is also done both ways, i.e., increasing the brightness and decreasing it. We observe very similar trends in the accuracy curves for both brightness increase and decrease, as shown in Figure 8(e) and 8(f). SIFT and HOG operate on grayscale images and ignore colour information. The variation in brightness can often change the image's color scale, but since we deal with grayscale images in the classical approach, we see only a minor drop in performance.

Additionally, we use pre-trained weights for the deep model and optimize the weights on our data. The model that we train on and the pre-trained model both have seen multiple images in various lighting scenarios. As a result, both models do a fine job when subjected to variations in brightness. The model in this aspect can be further enhanced if we induce color jitter augmentations.

4.3.5. Occlusion of the image

We randomly select x and y coordinates for placing the occlusion box, and depending on the size of the occlusion, we add it to x and y and replace all the pixels in the region to be 0. Both models do a great job even with the occlusion because the maximum size of occlusion we place is 45×45, which is somewhere around 0.8% of the pixel area of the original image, and the rest of the image remains absolutely unchanged, so the model has an ample amount of context to still derive important features from classifying the image.

Additionally, during the training of the models, we subject the data to translation and rotation augmentations, thus inducing contiguous chunks of all black pixels, and so the model is used to seeing such data, thus resulting in little change in the accuracies. We try to visualize the ResNet model with Grad-CAM (Selvaraju et al., 2017) when subjected to high occlusions as shown in 7. The image has both tennis and cricket balls, but the model focuses on classifying the image as a tennis ball. When we place an occlusion square of size 100px on the region that helps the model classify the image as a tennis ball, the model then finds relevant features in the threading of the cricket ball and classifies the image as a cricket ball. Thus, a perturbation study with a higher occlusion factor for the image of size 512×512 would result in acute drops in accuracy.

4.3.6. Salt and Pepper (SAP) Noise

We utilize the random_noise functionality from skimage.util to introduce SAP noise on the images. This randomly picks pixels in an image to either completely maximize or minimize those selected pixels to a value of 255 or 0, respectively. SAP noise is often detrimental to the performance of classification models as both the classical and deep learning models rely heavily on edges to classify images. Edges, by their very definition, are formed by continuous pixels, and a break even in one of the pixels in the series can result in major misclassifications. This is observed in both the models we train, as shown in Figure 8(h) below. The accuracy of the SVC model drops from 61.49% to 12.33% and that of deep learning model drops all the way from 90.71% to 34.11%.

The way to circumvent this problem can be to add the SAP noise as one of the augmentations in the training data so that the model runs better feature representations to correctly classify the images even with SAP noise. Another approach can be to denoise images before feeding them to the classification model. SAP noise can be well dealt with using approaches like median filtering or a small neural network designed to denoise images (Zhang et al., 2017).

5. Conclusion

Our comprehensive study examined the performance of SVM models trained on SIFT and HOG features against a CNN model leveraging the ResNet-18 architecture. The results clearly demonstrate the superiority of the CNN model, which achieved a significant margin in accuracy and robustness over the classical models. The classical approaches,

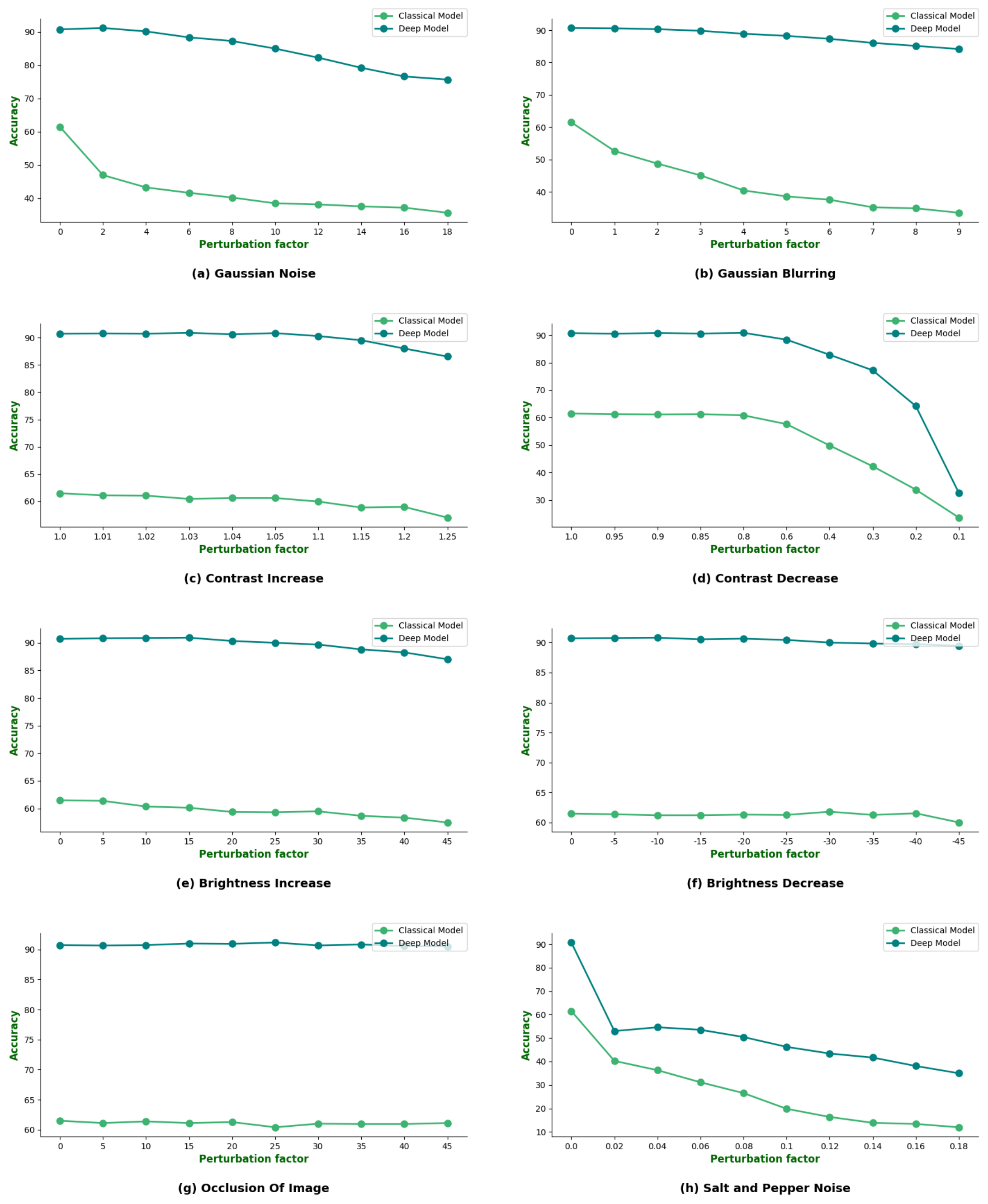


Figure 8. Perturbation results on test data for Classical and Deep Learning model

employing SIFT and HOG features, laid a strong foundation for understanding the importance of feature extraction in image classification. However, their reliance on handcrafted features proved to be a limitation in capturing the comprehensive essence of complex images, as evidenced by their performance under various image perturbations. The combined model, integrating both SIFT and HOG features, demonstrated improved performance, underscoring the potential benefits of feature fusion. Yet, it could not surpass the adaptability and learning capacity of the CNN model.

On the other hand, the deep learning approach, highlighted by the ResNet-18 model, showcased its strength in automatically learning feature representations directly from the data. This ability not only facilitated a higher classification accuracy of 90.71% but also endowed the model with a commendable resilience to image perturbations. Furthermore, techniques like Grad-CAM offered deeper insights into the CNN's decision-making process, revealing its focus areas within images for classification. Further enhancements to this training can be done by adding more augmentations and exploring more advanced deep learning models.

References

Agarap, Abien Fred. Deep learning using rectified linear units (relu), 2019.

Breiman, Leo. Random forests. *Machine learning*, 45: 5–32, 2001.

Cortes, Corinna and Vapnik, Vladimir. Support-vector networks. *Machine learning*, 20:273–297, 1995.

Dalal, Navneet and Triggs, Bill. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pp. 886–893. Ieee, 2005.

Deng, Jia, Dong, Wei, Socher, Richard, Li, Li-Jia, Li, Kai, and Fei-Fei, Li. Imagenet: A large-scale hierarchical image database.

Egmont-Petersen, Michael, de Ridder, Dick, and Handels, Heinz. Image processing with neural networks—a review. *Pattern recognition*, 35(10):2279–2301, 2002.

Hartigan, J. A. and Wong, M. A. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1): 100–108, 1979. ISSN 00359254, 14679876. URL http://www.jstor.org/stable/2346830.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition, 2015.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Kundu, Rohit, Das, Ritacheta, Geem, Zong Woo, Han, Gi-Tae, and Sarkar, Ram. Pneumonia detection in chest x-ray images using an ensemble of deep learning models. *PloS one*, 16(9):e0256630, 2021.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lowe, David G. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pp. 1150–1157. Ieee, 1999.

Lowe, David G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

Ruder, Sebastian. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Selvaraju, Ramprasaath R, Cogswell, Michael, Das, Abhishek, Vedantam, Ramakrishna, Parikh, Devi, and Batra, Dhruv. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pp. 618–626, 2017.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition, 2015.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions, 2014.

Tieleman, Tijmen and Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Wold, Svante, Esbensen, Kim, and Geladi, Paul. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

Zhang, Kai, Zuo, Wangmeng, Chen, Yunjin, Meng, Deyu, and Zhang, Lei. Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. *IEEE transactions on image processing*, 26(7):3142–3155, 2017.

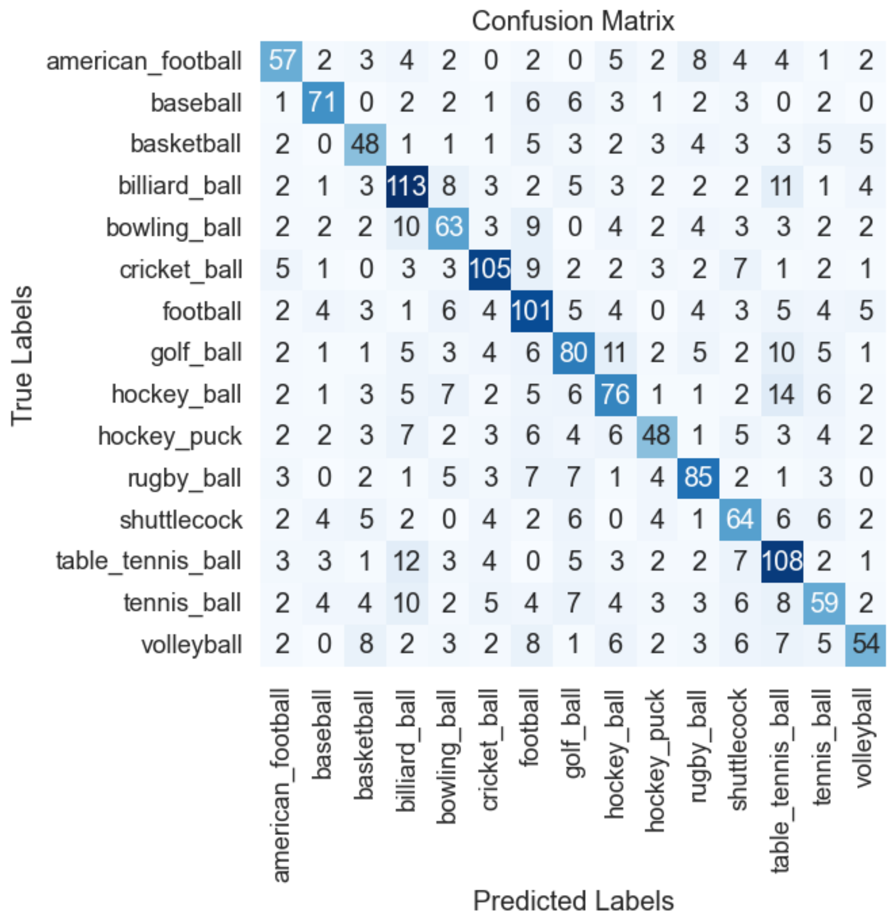
A. Classification report and Confusion Matrix for Classical model

Classification report for the Classical Model

```
                   precision    recall  f1-score   support

 american_football      0.64      0.59      0.62        96
          baseball      0.74      0.71      0.72       100
        basketball      0.56      0.56      0.56        86
     billiard_ball      0.63      0.70      0.66       162
      bowling_ball      0.57      0.57      0.57       111
      cricket_ball      0.73      0.72      0.72       146
          football      0.59      0.67      0.63       151
         golf_ball      0.58      0.58      0.58       138
       hockey_ball      0.58      0.57      0.58       133
       hockey_puck      0.61      0.49      0.54        98
        rugby_ball      0.67      0.69      0.68       124
       shuttlecock      0.54      0.59      0.56       108
 table_tennis_ball      0.59      0.69      0.64       156
       tennis_ball      0.55      0.48      0.51       123
        volleyball      0.65      0.50      0.56       109

          accuracy                          0.61      1841
         macro avg      0.62      0.61      0.61      1841
      weighted avg      0.62      0.61      0.61      1841
```



	american_football	baseball	basketball	billiard_ball	bowling_ball	cricket_ball	football	golf_ball	hockey_ball	hockey_puck	rugby_ball	shuttlecock	table_tennis_ball	tennis_ball	volleyball
american_football	57	2	3	4	2	0	2	0	5	2	8	4	4	1	2
baseball	1	71	0	2	2	1	6	6	3	1	2	3	0	2	0
basketball	2	0	48	1	1	1	5	3	2	3	4	3	3	5	5
billiard_ball	2	1	3	113	8	3	2	5	3	2	2	2	11	1	4
bowling_ball	2	2	2	10	63	3	9	0	4	2	4	3	3	2	2
cricket_ball	5	1	0	3	3	105	9	2	2	3	2	7	1	2	1
football	2	4	3	1	6	4	101	5	4	0	4	3	5	4	5
golf_ball	2	1	1	5	3	4	6	80	11	2	5	2	10	5	1
hockey_ball	2	1	3	5	7	2	5	6	76	1	1	2	14	6	2
hockey_puck	2	2	3	7	2	3	6	4	6	48	1	5	3	4	2
rugby_ball	3	0	2	1	5	3	7	7	1	4	85	2	1	3	0
shuttlecock	2	4	5	2	0	4	2	6	0	4	1	64	6	6	2
table_tennis_ball	3	3	1	12	3	4	0	5	3	2	2	7	108	2	1
tennis_ball	2	4	4	10	2	5	4	7	4	3	3	6	8	59	2
volleyball	2	0	8	2	3	2	8	1	6	2	3	6	7	5	54

Figure 9. Confusion Matrix for the Classical Model

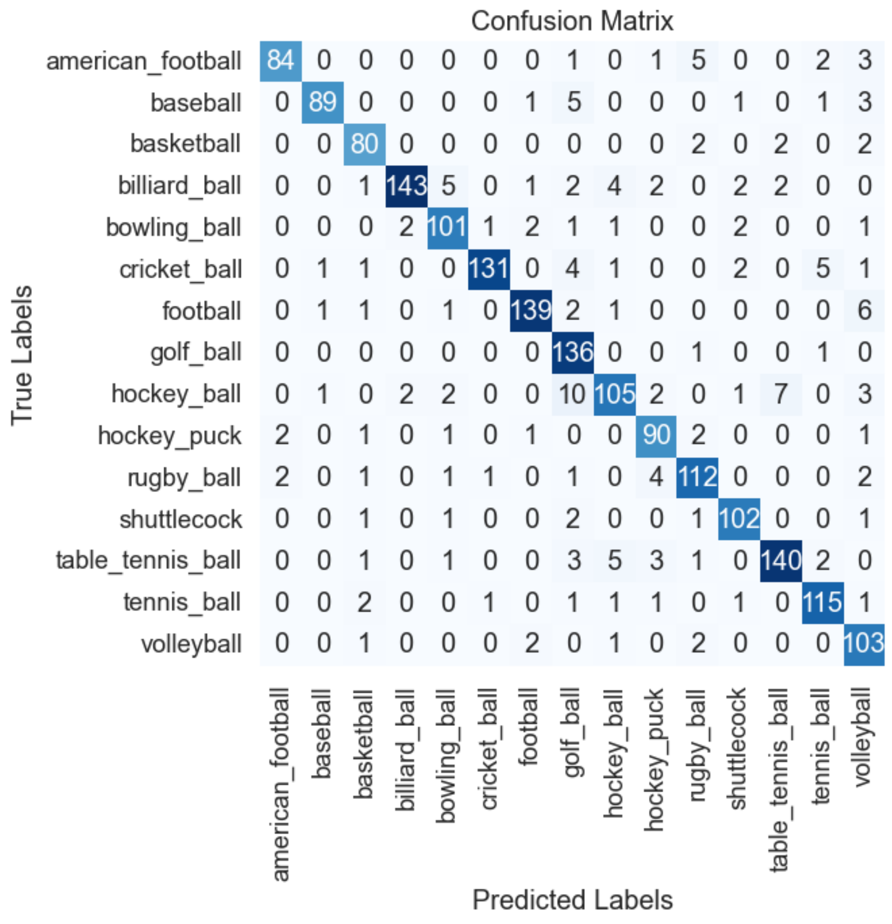
B. Classification report and Confusion Matrix for Deep Learning model

Classification report for the Deep Model

```
                   precision    recall  f1-score   support

 american_football       0.95      0.88      0.91        96
          baseball       0.97      0.89      0.93       100
        basketball       0.89      0.93      0.91        86
     billiard_ball       0.97      0.88      0.93       162
      bowling_ball       0.89      0.91      0.90       111
      cricket_ball       0.98      0.90      0.94       146
          football       0.95      0.92      0.94       151
         golf_ball       0.81      0.99      0.89       138
       hockey_ball       0.88      0.79      0.83       133
       hockey_puck       0.87      0.92      0.90        98
        rugby_ball       0.89      0.90      0.90       124
       shuttlecock       0.92      0.94      0.93       108
 table_tennis_ball       0.93      0.90      0.91       156
       tennis_ball       0.91      0.93      0.92       123
        volleyball       0.81      0.94      0.87       109

          accuracy                           0.91      1841
         macro avg       0.91      0.91      0.91      1841
      weighted avg       0.91      0.91      0.91      1841
```



	american_football	baseball	basketball	billiard_ball	bowling_ball	cricket_ball	football	golf_ball	hockey_ball	hockey_puck	rugby_ball	shuttlecock	table_tennis_ball	tennis_ball	volleyball
american_football	84	0	0	0	0	0	0	1	0	1	5	0	0	2	3
baseball	0	89	0	0	0	0	1	5	0	0	0	1	0	1	3
basketball	0	0	80	0	0	0	0	0	0	0	2	0	2	0	2
billiard_ball	0	0	1	143	5	0	1	2	4	2	0	2	2	0	0
bowling_ball	0	0	0	2	101	1	2	1	1	0	0	2	0	0	1
cricket_ball	0	1	1	0	0	131	0	4	1	0	0	2	0	5	1
football	0	1	1	0	1	0	139	2	1	0	0	0	0	0	6
golf_ball	0	0	0	0	0	0	0	136	0	0	1	0	0	1	0
hockey_ball	0	1	0	2	2	0	0	10	105	2	0	1	7	0	3
hockey_puck	2	0	1	0	1	0	1	0	0	90	2	0	0	0	1
rugby_ball	2	0	1	0	1	1	0	1	0	4	112	0	0	0	2
shuttlecock	0	0	1	0	1	0	0	2	0	0	1	102	0	0	1
table_tennis_ball	0	0	1	0	1	0	0	3	5	3	1	0	140	2	0
tennis_ball	0	0	2	0	0	1	0	1	1	1	0	1	0	115	1
volleyball	0	0	1	0	0	0	2	0	1	0	2	0	0	0	103

Figure 10. Confusion Matrix for the Deep Model

C. Code Files

C.1. Driver file - main.py

The main file which does the training of both models and runs the model through perturbations.

C.1.1. Instructions to run

```
python main.py --load_from_checkpoint_classical True --grid_search True --load_from_checkpoint_deep True
↪ --perturbation_study True
```

load_from_checkpoint_classical - If this argument is set to True, we load the model from checkpoints defined in constants.py file else it retrains the model

grid_search - If this argument is set to True, we perform a grid search else we train the model with rbf kernel and C=10

load_from_checkpoint_deep - If this argument is set to True, we load the model from checkpoints defined in constants.py file else it retrains the model

perturbation_study - If the argument is True, we perform a perturbation study for our model, else we do not do any study for perturbations

C.1.2. Code

```python
from svc_model import load_or_train_svc_model
from resnet_model import load_or_train_resnet_model
from utils import load_datasets, extract_features, make_feature_train_ready
from perturbations import *
from torchvision import transforms
from tqdm import tqdm
from constants import *
from sklearn.metrics import accuracy_score
from plotting import plot_perturbation_graphs
import argparse


device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

class TestLoaderEvaluator:
    def __init__(self, test_loader, values, transformation_function, perturbation_name):
        self.test_loader = test_loader
        self.values = values
        self.transformation_function = transformation_function
        self.perturbation_name = perturbation_name

    def evaluate_deepmodel(self, model):
        """
        The function runs the deep learning model on the test data loader containing some perturbation

        Parameters:
            model (model object) - the ResNet model itself

        Returns
            accuracy (float) - the accuracy of the model on the testloader
        """
        test_labels, predictions = [], []
        model.eval()
        with torch.no_grad():
            for images, labels in tqdm(self.test_loader):
                images, labels = images.to(device), labels.to(device)
                # Run the images through our model
                outputs = model(images)

                # Selecting the best output from the predictions
                _, predicted = outputs.max(1)
                predictions.extend(predicted.cpu().numpy())  # Convert predictions to CPU and append
                test_labels.extend(labels.cpu().numpy())  # Convert labels to CPU and append
        # Calculate accuracy
        accuracy = accuracy_score(test_labels, predictions)*100
        return accuracy

    def evaluate_classicalmodel(self, clustering_model, pca_model, svc_model ):
        """
        The function runs the classical learning model on the test data loader containing some perturbation

        Parameters:
            clustering_model (pickle object) - the kmeans clustering model for sift features
            pca_model (pickle object) - the pca dimension reducing model for hog features
            svc_model (pickle object) - the SVM model which does the classification of the inage

        Returns
            accuracy (float) - the accuracy of the model on the testloader
        """
        # Extract images and labels
        test_images, test_labels = extract_features(self.test_loader)
        # Extract sift and hog features
        sift_test, hog_test = make_feature_train_ready(clustering_model, test_images)
        # Compress hog features
        test_hog_compressed = pca_model.transform(hog_test)
        # Combine the hog and sift features
        test_features = np.concatenate((test_hog_compressed, sift_test), axis=1)
        # Running predictions
        predictions = svc_model.predict(test_features)
        accuracy = accuracy_score(test_labels, predictions)*100
        return accuracy

    def evaluate(self, resnet_model,  clustering_model, pca_model, svc_model ):
        """
        Driver evaluation function which makes calls to both evaluation functions

        Parameters:
            resnet_model (model object) - the ResNet model itself
            clustering_model (pickle object) - the kmeans clustering model for sift features
            pca_model (pickle object) - the pca dimension reducing model for hog features
            svc_model (pickle object) - the SVM model which does the classification of the inage

        Returns
            accuracy_classical (list) - list of all accuracy for classical model for all perturbations
            accuracy_deep (list) - list of all accuracy for deep model for all perturbations

        """
        accuracy_classical, accuracy_deep = [], []
        for value in self.values:
            correct = 0
            total = 0
            self.test_loader.dataset.transform =  transforms.Compose([
                # crop the image to make it 512 * 512
                transforms.Resize(512),
                transforms.CenterCrop(512),
                transforms.ToTensor(),
                transforms.Lambda(lambda x: self.transformation_function(x, value)), # perturbation function
            ])

            # Evaluate the classical model
            test_accuracy_classical = self.evaluate_classicalmodel(clustering_model, pca_model, svc_model )
            accuracy_classical.append(test_accuracy_classical)
            print(f'Test Accuracy with Classical model on {self.perturbation_name} factor {value}:
            ↪  {test_accuracy_classical:.2f}%')

            # Evaluate the deep learning model
            test_accuracy_deep = self.evaluate_deepmodel(resnet_model)
            accuracy_deep.append(test_accuracy_deep)
            print(f'Test Accuracy with deep model on {self.perturbation_name} factor {value}:
            ↪  {test_accuracy_deep:.2f}%')
        return accuracy_classical, accuracy_deep


def main(args):
    train_loader, val_loader, test_loader, class_names = load_datasets(TRAIN_DIR, TEST_DIR, TRAIN_VAL_SPLIT)
    # Load or train a new classical model
    clustering_model, pca_model, svc_model = load_or_train_svc_model(CLASSICAL_MODEL_DIR, CLUSTERING_MODEL_FILE, \
                                            PCA_MODEL_FILE, SVC_MODEL_FILE,
                                            ↪  load_checkpoint=args.load_from_checkpoint_classical, \
                                            grid_search=args.grid_search)

    # Load or train the deep learning model
    resnet_model = load_or_train_resnet_model(DEEP_MODEL_DIR, DEEP_MODEL_FILE,
    ↪  load_checkpoint=args.load_from_checkpoint_deep)
    print("Models Loaded!!")
    if args.perturbation_study:
        accuracy_dict_classical, accuracy_dict_deep = {}, {}
        for key, value in perturbations_dict.items():

            perturbation, modification_function = value[0], value[1]

            # Evaluate the model with different perturbation and perturbation factors
            evaluator = TestLoaderEvaluator(test_loader, perturbation, modification_function, key)
            accuracy_dict_classical[key], accuracy_dict_deep[key] = evaluator.evaluate(resnet_model, \
                                            clustering_model, pca_model, svc_model )
        # Plot the output graph of accuracy with perturbation factor
        plot_perturbation_graphs(accuracy_dict_classical, accuracy_dict_deep, "perturbation_plots.png")


if __name__ == "__main__":
    # Create ArgumentParser object
    parser = argparse.ArgumentParser(description="A script to parse command-line arguments")

    # Add argument for script name
    parser.add_argument('--load_from_checkpoint_classical', help='False for retraining', default=True)
    parser.add_argument('--grid_search', help='Do grid search if retraining', default=True)
    parser.add_argument('--load_from_checkpoint_deep', help='False for retraining', default=True)
    parser.add_argument('--perturbation_study', help='True if we want to retrain classical', default=True)

    # Add argument for additional arguments with identifier
    parser.add_argument('--arguments', nargs='*', help='Additional arguments', dest='arguments')
    # Parse command-line arguments
    args = parser.parse_args()

    # Call main function with parsed arguments
    main(args)
```

C.2. Classical model file - svc_model.py

This file trains the SVC model if load_from_checkpoint_classical is set to False. Also does a grid search if the grid_search parameter is set to True

```
from utils import extract_features, load_datasets, \
    extract_hog_features, extract_sift_features, load_sift_features, \
    make_feature_train_ready, save_sklearn_model, load_sklearn_model, \
    create_directory_if_not_exists
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import numpy as np
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, f1_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from plotting import plot_confusion_matrix
from constants import *

TRAIN_VAL_SPLIT = 0.9999

def svc_trainer(train_features, test_features, train_labels, test_labels, class_names, grid_search):
    """
    The function which does the training for our svc model

    Parameters:
        train_features (array): Training data with features extracted
        test_features (array): Test data with features extracted
        train_labels (array): Correct labels for the training images
        test_labels (array): Correct labels for the test images
        class_names (list): The list of class names i.e. categories of classification

    Return:
        clf (sklearn model object): the classifier model for classification
        accuracy (float): accuracy of the classification model
        f1 (float): f1 score of our classification model
        report (str): the report of the entire classification for each class
    """

    # Grid PARAM SEARCH
    if grid_search:
        pipeline = Pipeline([
            ('scaler', StandardScaler()),
            ('svc', SVC())
        ])

        # Define parameter grid for grid search
        param_grid = {
            'svc__C': [0.1, 1, 10],  # Regularization parameter
            'svc__kernel': ['rbf', 'poly'],  # Kernel type
            'svc__gamma': ['scale', 'auto'],  # Kernel coefficient
        }

        # Perform Grid Search with 5-fold cross-validation
        grid_search = GridSearchCV(estimator=pipeline, param_grid=param_grid,verbose=2, cv=5)
        grid_search.fit(train_features, train_labels)

        # Best parameters found
        best_params = grid_search.best_params_
        print("Best parameters:", best_params)
        clf = grid_search.best_estimator_
    else:
        # Train model with default settings without param search
        clf = make_pipeline(StandardScaler(), SVC(kernel='rbf', gamma='scale', C=10))
        clf.fit(train_features, train_labels)

    predictions = clf.predict(test_features)

    # Convert labels from indexes to actual class names
    test_labels_readable = [class_names[d] for d in test_labels]
    predictions_readable = [class_names[d] for d in predictions]

    # Calculate Metrics
    accuracy = accuracy_score(test_labels_readable, predictions_readable)
    f1 = f1_score(test_labels_readable, predictions_readable, average='weighted')
    report = classification_report(test_labels_readable, predictions_readable)
    plot_confusion_matrix(test_labels_readable, predictions_readable, class_names, "comf_mat_classical.png")

    return clf, accuracy, f1, report

def train_model(clusering_model_dir, pca_model_dir, svc_model_dir, grid_search):
    """
    The model trains the clustering model for sift and the dimensionality reduction model for hog

    Parameters:
        clusering_model_dir (string): The directory where we want to save kmeasns model
        pca_model_dir (string): The directory where we want to save pca model
        svc_model_dir (string): The directory where we want to save svc model
        grid_search (bool): The parameter decides if we want to use default model for svc or do grisearch

    Return:
        kmeans (sklearn model object): the kmeans model for sift
        pca (sklearn model object): the pca model for hog
        best_svc_model (sklearn model object): the best classifier model for classification
    """
    train_loader, val_loader, test_loader, class_names = load_datasets(TRAIN_DIR, TEST_DIR, TRAIN_VAL_SPLIT)
    print("Loading train and test dataset")
    train_images, train_labels = extract_features(train_loader)
    print("Training data loaded!!")
    test_images, test_labels = extract_features(test_loader)
    print("Test data loaded!!")

    print("Extracting SIFT features for kmeans, This may take 5-10 mins time")
    sift_features = load_sift_features(train_images)
    kmeans = KMeans(n_clusters=100)
    kmeans.fit(sift_features)
    save_sklearn_model(kmeans, clusering_model_dir)
    print("CLustering Model Saved!!")
```

```python
    print("Extracting HOG and SIFT features")
    sift_train, hog_train = make_feature_train_ready(kmeans, train_images)
    sift_test, hog_test = make_feature_train_ready(kmeans, test_images)

    print("Compressing HOG features using PCA")
    pca = PCA(n_components=100)  # Set the desired number of components
    pca.fit(hog_train)
    train_hog_compressed = pca.transform(hog_train)
    test_hog_compressed = pca.transform(hog_test)
    save_sklearn_model(pca, pca_model_dir)
    print("PCA Model Saved!!")

    train_combined = np.concatenate((train_hog_compressed, sift_train), axis=1)
    test_combined = np.concatenate((test_hog_compressed, sift_test), axis=1)

    print("Training the SVC model")
    best_svc_model, accuracy_final, f1_final, report_final = svc_trainer(train_combined, \
                        test_combined, train_labels, test_labels, class_names, grid_search)
    print("Accuracy of the SVC model:", accuracy_final)
    print("F1 score of the SVC model:", f1_final)
    print("Model classifcation report")
    print(report_final)
    save_sklearn_model(best_svc_model, svc_model_dir)
    print("Best SVC model saved!!")
    return kmeans, pca, best_svc_model

def load_or_train_svc_model(model_dir, clustering_model_dir, pca_model_dir, \
        svc_model_dir, load_checkpoint=True, grid_search=False):
    """
    The model is driver function to facilitate the entire svc classification

    Parameters:
        model_dir (str): Where the model files are kept
        clustering_model_dir (str): Filename to load or save by for clustering
        pca_model_dir (str): Filename to load or save by for pca
        svc_model_dir (str): Filename to load or save by for classification
        load_checkpoint (bool): Whether to load model or retrain
        grid_search (bool): Whether to do grid search or use default model

    Return:
        clustering_model (sklearn model object): the kmeans model for sift
        pca_model (sklearn model object): the pca model for hog
        svc_model (sklearn model object): the best classifier model for classification
    """
    create_directory_if_not_exists(model_dir)

    # Extracting directories
    clustering_model_dir = model_dir + clustering_model_dir
    pca_model_dir = model_dir + pca_model_dir
    svc_model_dir = model_dir + svc_model_dir

    # check for loading a model or re-training
    if load_checkpoint:
        clustering_model = load_sklearn_model(clustering_model_dir)
        pca_model = load_sklearn_model(pca_model_dir)
        svc_model = load_sklearn_model(svc_model_dir)
    else:
        clustering_model, pca_model, svc_model = train_model(clustering_model_dir, \
                                                    pca_model_dir, svc_model_dir, grid_search)
    return clustering_model, pca_model, svc_model
```

C.3. Deep model file - resnet_model.py

This file trains the ResNet model if load_from_checkpoint_deep is set to False.

```
import torch
from utils import load_datasets, create_directory_if_not_exists
from torchvision.models import resnet18
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, f1_score
from plotting import plot_confusion_matrix, plot_training_metrics
from constants import *


device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

def model_backbone(model_path, class_names, load_model=False):
    """
    Loads the model architecture and weights if needed

    Parameters:
        model_path (str): path at which the model is kept if needs to be loaded
        class_names (list):  The list of class names i.e. categories of classification
        load_model (bool): True if you want to load existing weights

    Return:
        model: torch model object for resnet
    """
    model = resnet18(pretrained=True)
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, len(class_names))  # Modify the output layer to match the number of classes
    model = model.to(device)
    if load_model:
        model.load_state_dict(torch.load(model_path))
    return model

def evaluate_on_test(model_path, test_loader, class_names, load_model):
    """
    The function helps run evaluations on test files

    Parameters:
        model_path (str): path at which the model is kept if needs to be loaded
        test_loader (Torch DataLoader): dataloader which contains the data on which the model needs to be evaluated
        class_names (list):  The list of class names i.e. categories of classification
        load_model (bool): True if you want to load existing weights

    Return:
        model: torch model object for resnet
    """
    model = model_backbone(model_path, class_names, load_model=load_model)
    model.eval()  # Set the model to evaluation mode
    test_labels, predictions = [], []

    with torch.no_grad():
        for images, labels in tqdm(test_loader):
            images, labels = images.to(device), labels.to(device)
            # Running the images through model
            outputs = model(images)
            # Extracting the top prediction
            _, predicted = outputs.max(1)
            predictions.extend(predicted.cpu().numpy())  # Convert predictions to CPU and append
            test_labels.extend(labels.cpu().numpy())  # Convert labels to CPU and append

    # Generate classification report
    test_labels_readable = [class_names[d] for d in test_labels]
    predictions_readable = [class_names[d] for d in predictions]

    # Calculate accuracy
    accuracy = accuracy_score(test_labels_readable, predictions_readable)
    f1 = f1_score(test_labels_readable, predictions_readable, average='weighted')
    report = classification_report(test_labels_readable, predictions_readable)
    plot_confusion_matrix(test_labels_readable, predictions_readable, class_names, "comf_mat_deep.png")
    print("Accuracy of the ResNet model:", accuracy)
    print("F1 score of the ResNet model:", f1)
    print("Model classifcation report")
    print(report)
    return model

def train_model(train_loader, val_loader, test_loader, model_path, class_names, num_epochs=10):
    """
    The function helps train the resnet model

    Parameters:
        train_loader (Torch DataLoader): dataloader which contains the data on which the model needs to be trained
        val_loader (Torch DataLoader): dataloader which contains the data on which the model needs to be validated
        test_loader (Torch DataLoader): dataloader which contains the data on which the model needs to be evaluated
        model_path (str): path at which the model is kept if needs to be loaded
        class_names (list):  The list of class names i.e. categories of classification
        num_epochs (int): The number of epochs to train

    Return:
        model: torch model object for resnet
    """
    model = model_backbone(model_path, class_names)
    # Define loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)
    loss_list, acc_list = [],[]
    best_val_loss, best_model_state = float('inf'), None
    for epoch in range(num_epochs):
        model.train()  # Set the model to training mode
        running_loss = 0.0
        correct = 0
        total = 0
        for images, labels in tqdm(train_loader):
            images, labels = images.to(device), labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass and optimize
            loss.backward()
            optimizer.step()

            # Calculate training accuracy
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        train_accuracy = 100 * correct / total
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Train Accuracy:
        ↪  {train_accuracy:.2f}%')

        model.eval()  # Set the model to evaluation mode
        val_loss = 0.0
        val_correct = 0
        val_total = 0
        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                # Calculate validation accuracy
                _, predicted = outputs.max(1)
                val_total += labels.size(0)
                val_correct += predicted.eq(labels).sum().item()

        val_loss /= len(val_loader)
        val_accuracy = 100 * val_correct / val_total
        loss_list.append([train_loss, val_loss])
        acc_list.append([train_accuracy, val_accuracy])
        print(f'Epoch [{epoch+1}/{num_epochs}], Validation Loss: {val_loss:.4f}, Validation Accuracy:
        ↪  {val_accuracy:.2f}%')

        # Check if current validation loss is better than the best seen so far
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict()
            # Save the best model checkpoint
            print("Saving the new best model")
            torch.save(best_model_state, model_path)

    print("Saving training graph!!")
    plot_training_metrics(loss_list, acc_list, num_epochs, "training_graph.png")
    model = evaluate_on_test(model_path, test_loader, class_names,load_model=True)
    return model

def load_or_train_resnet_model(model_dir, model_file, load_checkpoint=True):
    """
    The driver function to facilitate resnet modules
    Parameters:
        model_dir (str): the directory at which the model is kept
        model_file (str): the name of the model file either to be loaded or to be saved by
        load_checkpoint (bool): the variable which defines whether to laod or retrain model

    Returns:
        resnet_model: torch model object for resnet
    """
    create_directory_if_not_exists(model_dir)
    train_loader, val_loader, test_loader, class_names = load_datasets(TRAIN_DIR, TEST_DIR, TRAIN_VAL_SPLIT)

    model_path = model_dir + model_file
    if load_checkpoint:
        resnet_model = model_backbone(model_path, class_names, load_checkpoint)
    else:
        resnet_model = train_model(train_loader, val_loader, test_loader, \
                                model_path, class_names)
    return resnet_model
```

C.4. Perturbations file - perturbations.py

This file contains all the perturbation functions used to induce perturbations on the image.

```
import torch
import numpy as np
from PIL import Image
import torch.nn.functional as F
from skimage.util import random_noise
import random


torch.manual_seed(42)

def scale_image(image):
    # Bring values in the range of 0 to 255
    scaled_image = torch.clamp(image, 0, 255)

    # Convert to integer tensor
    scaled_image = scaled_image.round()/255
    return scaled_image

def add_gaussian_noise(image_tensor, std_dev):
    # Converting image to range (0,255)
    image_tensor = (image_tensor * 255)
    # Convert the tensor to numpy array
    image_np = image_tensor.numpy()
    # Generate Gaussian noise with the given standard deviation
    noise = np.random.normal(loc=0, scale=std_dev, size=image_np.shape)
    # Add the noise to the image
    noisy_image = image_np + noise
    noisy_image = torch.from_numpy(noisy_image).type(torch.float32)
    # Ensure pixel values are integers in the range 0..255
    return scale_image(noisy_image)


def gaussian_blur(image_tensor, num_iterations):
    # Converting image to range (0,255)
    image_tensor = (image_tensor * 255)
    # Assuming your input image is of shape [batch_size, channels, height, width]
    input_image = image_tensor.unsqueeze(0)  # Example shape [batch_size, 3 channels, 32x32]
    # Assuming your kernel is of shape [out_channels, in_channels, kernel_height, kernel_width]
    kernel_channel = torch.tensor([[1, 2, 1],
                                   [2, 4, 2],
                                   [1, 2, 1]], dtype=torch.float32) / 16

    # Apply the convolution operation separately to each channel
    for iteration in range(num_iterations):
        conv_outputs = []
        for i in range(input_image.shape[1]):  # Iterate over each input channel
            input_channel = input_image[:, i:i+1, :, :].to(torch.float32)  # Select the current input channel
            conv_output = F.conv2d(input_channel, kernel_channel.unsqueeze(0).unsqueeze(0), padding='same')  #
            ↪  Apply convolution
            conv_outputs.append(conv_output)
        # Concatenate the outputs along the channel dimension
        input_image = torch.cat(conv_outputs, dim=1).to(torch.float32)

    # Now output_image will have shape [batch_size, channels, height, width]
    return scale_image(input_image.squeeze(0))

def contrast_change(image_tensor, factor):
    # Converting image to range (0,255)
    image_tensor = (image_tensor * 255)
    # Contrast simply means multiplication by factor
    contrast_image = image_tensor * factor
    return scale_image(contrast_image)

def brightness_change(image_tensor, factor):
    # Converting image to range (0,255)
    image_tensor = (image_tensor * 255)
    # Brightness means adding or subtracting factor from all pixels
    brightness_image = image_tensor + factor
    return scale_image(brightness_image)

def place_square_occlusion(image_tensor, size):
    # Converting image to range (0,255)
    image_tensor = (image_tensor * 255)
    # Randomly select the top-left corner of the square region
    x = random.randint(0, image_tensor.shape[1] - size - 1)
    y = random.randint(0, image_tensor.shape[2] - size - 1)
    # Set the pixels in the square region to zero
    image_tensor[:, x:x+size, y:y+size] = 0
    return scale_image(image_tensor)

def add_salt_and_pepper_noise(image_tensor, noise_strength):
    # Converting image to range (0,255)
    image_tensor = (image_tensor * 255)
    # Convert image to numpy array and normalize to [0, 1]
    image_np = image_tensor.float().permute(1, 2, 0).numpy() / 255.0
    # Add salt and pepper noise
    noisy_image_np = random_noise(image_np, mode='s&p', amount=noise_strength)
    # Convert back to torch tensor and rescale to [0, 255]
    noisy_image = torch.tensor(noisy_image_np * 255, dtype=torch.uint8).permute(2, 0, 1)
    return scale_image(noisy_image)
```

C.5. Utility file - utils.py

This file is the primary utility file being used to generate datasets, load models, save models, extracting features, etc.

```python
from skimage import color, io, feature
from skimage import img_as_float
import cv2
from torchvision.datasets import ImageFolder
from constants import *
from torch.utils.data import DataLoader, random_split
import torch
from PIL import Image
import numpy as np
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
import pickle
import tqdm
import os

torch.manual_seed(42)
sift = cv2.SIFT_create()
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

def create_directory_if_not_exists(directory):
    """
    Check if a directory exists, if not, create it.
    """
    if not os.path.exists(directory):
        os.makedirs(directory)
        print(f"Directory '{directory}' created successfully.")
    else:
        print(f"Directory '{directory}' already exists.")


def load_datasets(train_dir, test_dir, train_val_split, batch_size=32):
    # Load the training dataset
    dataset = ImageFolder(root=train_dir)
    class_names = dataset.classes
    # get train size
    train_size = int(train_val_split * len(dataset))
    val_size = len(dataset) - train_size
    train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
    # Apply transformations
    train_dataset.dataset.transform = train_transform
    val_dataset.dataset.transform = val_transform
    # Make data loader for train and val
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
    # Load the testing dataset
    test_dataset = ImageFolder(test_dir, transform=val_transform)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
    return train_loader, val_loader, test_loader, class_names

def extract_features(data_loader):
    features = []
    labels = []
    for batch in data_loader:
        batch_data, batch_labels = batch
        # Extract features from each image in the batch separately
        for image in batch_data:
            np_image = image.permute(1, 2, 0).numpy()
            pil_image = Image.fromarray((np_image * 255).astype(np.uint8))
            features.append(pil_image)
        labels.extend(batch_labels.tolist())
    return features, torch.tensor(labels).numpy()

#Extract HoG features from an image(512X512)
def extract_hog_features(image, pixels_per_cell=(32, 32), cells_per_block=(2, 2)):
    image = img_as_float(image)
    if image.ndim == 3:
        image = color.rgb2gray(image)  #grayscale conversion
    hog_features = feature.hog(image, pixels_per_cell=pixels_per_cell, cells_per_block=cells_per_block,
    ↪  feature_vector=True)
    return hog_features

def extract_sift_features(image):
    numpy_image = np.array(image)
    # Convert the color space to grayscale using OpenCV
    grayscale_image = cv2.cvtColor(numpy_image, cv2.COLOR_RGB2GRAY)
    # Extract sift features
    _, des = sift.detectAndCompute(grayscale_image, None)
    if des is not None:
        return des
    return None

def load_sift_features(images):
    sift_features = []
    for image in images:
        descriptor = extract_sift_features(image)
        if descriptor is not None:
            sift_features.extend(descriptor)

    return np.array(sift_features)

def make_feature_train_ready(kmeans_model, images):
    sift_histograms, hog_features = [], []
    for i, image in tqdm.tqdm(enumerate(images)):
        des = extract_sift_features(image)
        if des is not None:
            # perform kmeans on sift features
            visual_words = kmeans_model.predict(des)
            histogram, _ = np.histogram(visual_words, bins=range(101))
            sift_histograms.append(histogram)
        else:
            sift_histograms.append(np.zeros(100))
        # get all hog features
        hog_feature = extract_hog_features(image)
        hog_features.append(hog_feature)
    return sift_histograms, hog_features

# Save model files
def save_sklearn_model(model, key):
    with open(key, "wb") as f:
        pickle.dump(model, f)

# Load model files
def load_sklearn_model(key):
    with open(key, "rb") as f:
        model = pickle.load(f)
    return model
```

C.6. Plotting file - plotting.py

This file helps plot the perturbation plot of accuracy with respect to the factor of perturbation. Also helps in plotting the training metric graphs and the confusion matrices

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import math
from constants import *

def plot_confusion_matrix(true_labels_numeric, predicted_labels_numeric, classes, filename):
    conf_matrix = confusion_matrix(true_labels_numeric, predicted_labels_numeric)

    # Plot confusion matrix
    sns.set(font_scale=1.2)  # Adjust font scale as needed
    plt.figure(figsize=(8, 6))  # Adjust figure size as needed
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap="Blues", cbar=False, square=True,
                xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.savefig(filename, bbox_inches='tight')
    plt.close()


def plot_training_metrics(loss, acc, epochs_num,filename):
    epochs = list(range(10))
    loss_train = [sublist[0] for sublist in loss]
    loss_val = [sublist[1] for sublist in loss]
    accuracy_train = [sublist[0] for sublist in acc]
    accuracy_val = [sublist[1] for sublist in acc]
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Plot loss in the first subplot
    ax1.plot(epochs, loss_train, marker='o', linestyle='-', color='mediumseagreen', linewidth=2, markersize=8,
    ↪  label='Training Loss')
    ax1.plot(epochs, loss_val, marker='o', linestyle='-', color='teal', linewidth=2, markersize=8,
    ↪  label="Validation Loss")

    ax1.set_title('Loss')
    ax1.set_xlabel('Epochs')
    ax1.set_ylabel('Loss')
    ax1.grid(False)
    ax1.legend()

    # Plot accuracy in the second subplot
    ax2.plot(epochs, accuracy_train, marker='o', linestyle='-', color='mediumseagreen', linewidth=2, markersize=8,
    ↪  label="Train Accuracy")
    ax2.plot(epochs, accuracy_val, marker='o', linestyle='-', color='teal', linewidth=2, markersize=8,
    ↪  label='Validation Accuracy')
    ax2.set_title('Accuracy')
    ax2.set_xlabel('Epochs')
    ax2.set_ylabel('Accuracy')
```

```python
    ax2.grid(False)
    ax2.legend()
    ax1.spines['top'].set_visible(False)
    ax1.spines['right'].set_visible(False)
    ax2.spines['top'].set_visible(False)
    ax2.spines['right'].set_visible(False)
    # Adjust layout to prevent overlap
    plt.tight_layout()

    # Save the plot
    plt.savefig(filename)

    # Don't forget to close the plot to release resources
    plt.close()


def plot_perturbation_graphs(accuracy_dict_classical, accuracy_dict_deep, filename):

    perturbation_names = list(perturbations_dict.keys())
    # Determine the number of models
    num_models = len(perturbation_names)
    # Calculate the number of rows and columns for the grid layout
    num_rows = math.ceil(num_models / 2)
    num_cols = 2
    fig_labels_name = ['(a)','(b)','(c)','(d)','(e)','(f)','(g)','(h)']
    # Create subplots with better color choices
    fig, axs = plt.subplots(num_rows, num_cols, figsize=(20, 6*num_rows))
    # Create subplots

    # Iterate over the dictionary entries and plot the accuracies on each subplot

    for i, perturbation_name in enumerate(perturbation_names):
        row = i // num_cols
        col = i % num_cols
        x_values = perturbations_dict[perturbation_name][0]
        accuracies_classical = accuracy_dict_classical[perturbation_name]
        accuracies_deep = accuracy_dict_deep[perturbation_name]
        epoch_labels = [str(x) for x in x_values]
        axs[row, col].plot(epoch_labels, accuracies_classical, marker='o', linestyle='-', color='mediumseagreen',
        ↪  linewidth=2, markersize=8)
        axs[row, col].plot(epoch_labels, accuracies_deep, marker='o', linestyle='-', color='teal', linewidth=2,
        ↪  markersize=8)
        axs[row, col].set_title(fig_labels_name[i] + " "+ perturbation_name, fontsize=14, fontweight='bold',
        ↪  color='black', y=-0.3)
        axs[row, col].set_xlabel('Perturbation factor', fontsize=12, fontweight='bold', color='darkgreen')
        axs[row, col].set_ylabel('Accuracy', fontsize=12, fontweight='bold', color='darkgreen')
        axs[row, col].tick_params(axis='both', which='major', labelsize=10)
        axs[row, col].spines['top'].set_visible(False)
        axs[row, col].spines['right'].set_visible(False)
        # axs[row, col].legend()
        axs[row, col].legend(['Classical Model', 'Deep Model'], loc='upper right', bbox_to_anchor=(1.0, 1.05),
        ↪  borderaxespad=0)

    plt.subplots_adjust(hspace=0.5)
    # Add a title for the entire figure
    plt.grid(False)
    # Adjust layout
    plt.savefig(filename)
    # Don't forget to close the plot to release resources
    plt.close()
```

C.7. Grad-CAM visualization file - gradcam.py

This file helps in generating a Grad-CAM image.

```
from resnet_model import model_backbone
from utils import load_datasets
import json
import torch
from torchvision import models, transforms
from PIL import Image as PilImage
from perturbations import *
from omnixai.data.image import Image
from omnixai.explainers.vision.specific.gradcam.pytorch.gradcam import GradCAM
from constants import *

# Load the pre-trained ResNet model
train_loader, val_loader, test_loader, class_names = load_datasets(TRAIN_DIR, TEST_DIR, TRAIN_VAL_SPLIT)

model = model_backbone(DEEP_MODEL_DIR + DEEP_MODEL_FILE, class_names, load_model=True)
model.eval()  # Set the model to evaluation mode

print("Model loaded")
image_path = 'tennis.jpg'  # Replace 'your_image.jpg' with the path to your image
img = Image(PilImage.open(image_path).convert('RGB'), )
preprocess = lambda ims: torch.stack([transform(im.to_pil()) for im in ims])


transform =  transforms.Compose([
                transforms.Resize(512),
                transforms.CenterCrop(512),
                transforms.ToTensor(),
                # transforms.Lambda(lambda x: place_square_occlusion(x,100)), # replace with perturbation you want
            ])# Load and preprocess the image
explainer = GradCAM(
    model=model,
    target_layer=model.layer4[0].conv2,
    preprocess_function=preprocess
)

# Explain the top label
explanations = explainer.explain(img)
print("Predicted class", class_names[explanations.get_explanations()[0]['target_label']])
explanations.ipython_plot(index=0, class_names=class_names)
```

C.8. Constants file - constant.py

This file has all the constants we use predefined

```
import torchvision.transforms as transforms
from perturbations import *

train_transform = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomHorizontalFlip(),
```

```python
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomVerticalFlip(),
    transforms.RandomResizedCrop(512, scale=(0.8, 1)),
    transforms.ToTensor(),
])

# Define transform for test/validation (without augmentation)
val_transform = transforms.Compose([
    transforms.Resize(512),
    transforms.CenterCrop(512),
    transforms.ToTensor(),
])

# Model contants
TRAIN_DIR = 'dataset/train'
TEST_DIR = 'dataset/test'
TRAIN_VAL_SPLIT = 0.9
CLASSICAL_MODEL_DIR = "models/classical/"
DEEP_MODEL_DIR = "models/deepmodel/"
CLUSTERING_MODEL_FILE = "clustering_model_scale.pkl"
SVC_MODEL_FILE = "svc_model_scale.pkl"
PCA_MODEL_FILE = "pca_model_scale.pkl"
DEEP_MODEL_FILE = "resnet_18_scale.pth"

# perturbation values
gaussian_noise = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18 ]
gaussian_blurring = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
contrast_inc = [1.0, 1.01, 1.02, 1.03, 1.04, 1.05, 1.1, 1.15, 1.20, 1.25 ]
contrast_dec = [1.0, 0.95, 0.90, 0.85, 0.80, 0.60, 0.40, 0.30, 0.20, 0.10]
brightness_inc = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
brightness_dec = [-x for x in brightness_inc]
occlusion_size = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
sap_noise = [0.00, 0.02, 0.04, 0.06, 0.08, 0.10, 0.12, 0.14, 0.16, 0.18]


# perturbation dictionary with values and functions
perturbations_dict = {"Gaussian Noise": (gaussian_noise, add_gaussian_noise), \
                "Gaussian Blurring": (gaussian_blurring, gaussian_blur), \
                "Contrast Increase": (contrast_inc, contrast_change), \
                "Contrast Decrease": (contrast_dec, contrast_change), \
                "Brightness Increase": (brightness_inc, brightness_change), \
                "Brightness Decrease": (brightness_dec, brightness_change), \
                "Occlusion Of Image": (occlusion_size, place_square_occlusion), \
                "Salt and Pepper Noise": (sap_noise, add_salt_and_pepper_noise)}
```