Abdul Rahim Ab Ghani
s1982752

Aqil Zainal Azhar
s2119011

Robustness Explorations in Image Classification: Classical Computer Vision vs Deep Learning

Abstract

This report is about an image classification task for about 2,400 images to two classes: cats and dogs. We will explore two classification methods representing a classical and deep learning approach. The classical approach employed is a SIFT-BOW SVM, and the deep learning approach is a pretrained ResNet18 model. The models are trained and tuned through held-out validation sets and put through robustness tests using 9 sets of perturbed images with 10 levels for each perturbation. Both models demonstrate the same robustness against most perturbations except for Gaussian blurring, HSV hue noise, and HSV saturation noise. Interestingly, ResNet also showed some signs of sensitivity to contrast and a larger variance to saturation noise.

1.0 Introduction

The traditional image classification methodology in computer vision usually involves a pipeline of image pre-processing, feature extraction and model training. The models used in these implementations are chosen from shallow machine learning models such as Support Vector Machines (SVM) or K-Nearest Neighbours (KNN). Feature extraction represents an essential element of the process since plugging in raw pixels as features into the supervised models can lead to poor results. Several elements of computer vision are involved in this process, including edge and corner detection. Popular feature detectors/descriptors such as SIFT (Scale Invariant Feature Transform) and SURF (Speeded-Up Robust Features) extract points of interest within the image and provide an appropriate description to be used as input features for the classification model [1]. Other popular feature generation approaches also include using a Bag of Words (BOW) feature representation generated with a K-Means clustering algorithm.

With the introduction of AlexNet [2] in the 2012 ILSVRC competition, much attention has been shifted to deep learning implementations for image classification. Since then, it has been widely accepted that the deep learning models' performance far supersedes classical computer vision, albeit with trade-offs in computational requirements [1]. In contrast to the traditional computer vision pipeline above, these methods employ an end-to-end learning process, which entirely skips the feature generation step. Instead, these features are learned by the deep net during training, where the model discovers the most descriptive and salient features from the dataset provided [1]. The 'deep' aspect of the model allows for multiple stages of feature abstraction from fine pixel-level patterns to more granular lines, shapes, and objects.

Although the recent developments in neural networks for image classification have shown tremendous results, there could exist issues where the features learned during the training process is specific to the dataset at hand. If these are poorly constructed, they will be unable to generalise to images outside the training set. Such effects are compounded when the training set is small, making the model prone to overfit. In contrast, features in the traditional computer vision procedure, such as SIFT descriptors, tend to be more general and not class-specific. Hence, these models might prove to be more robust against misclassifications in more diverse test scenarios. This report will investigate two image classification approaches, a classical computer vision model involving a SIFT-BOW SVM model and a deep neural net model using the ResNet-18 architecture [3]. Performance evaluation will be conducted based on the classification task of cat and dog images. These models will be trained on the same 'clean' training images, and their hyperparameters will be tuned on a held-out validation set. With their optimal hyperparameters, the models will then undergo a final round of testing with a 'perturbed' testing set to measure their robustness towards a more diverse set of images.

2.0 The Dataset

The dataset in this report consists of approximately 2400 RGB images of cats and dogs. The underlying application involves classifying these images into the cat and dog classes. Each class consist of 12 subclasses featuring the species of the cat or dog. There are approximately 100 images for each cat and dog species, making the dataset fairly class balanced. It is important to note that the classification task does not involve classifying each animal (cat or dog) into their respective species. The species' information will be used as part of the stratified split procedure for allocating the hold-out validation set and the folds for the testing. The images are split into three disjoint and equally sized folds (A, B & C), each with stratified classes (cats/dogs) and subclasses (species). Three combinations of training, tuning and testing will be performed on these three folds as follows:

Combinations	Training & tuning (validation)	Testing
1	A + B	C
2	A + C	B
3	B + C	A

The training and tuning fold is further split into a 75:25 ratio, with the larger portion used for training and the smaller for validation. This split is also stratified according to the classes and subclasses. The validation set is used to tune hyperparameters of the associated model trained on the corresponding training data. The models will then be tested on the perturbed versions of the images in the testing set.

3.0 Methodology

3.2 SIFT-BOW SVM

Our first classifier represents an approach from classical computer vision. The feature generation tool of choice is a Dense SIFT Bag of Words algorithm, while the machine learning classifier chosen is a linear Support Vector Machine model. The total dataset is split into its three equal folds, and the training, validation and testing data is established for the first iteration of combinations as described in Section 2.0. This step is performed by the ***trainvalidtest_paths.m*** function with seeds set for each split for reproducibility. All function calls and code are contained within the ***svm_main.m*** script. The following pipeline described below will be repeated for each iteration of the combination, and the results of each are stored for further analysis.

The first step in the SIFT-BOW feature representation is to generate an 'image vocabulary' based on training images in this iteration. This is carried out by ***codebook.m*** function script, which takes the training image paths and the vocabulary size (a hyperparameter) as input. The function first reads all images in the training paths and converts them into grayscale. Each image is subsequently smoothed by a Gaussian kernel filter using the ***vl_imsooth*** function obtained from VLFeat.org. The sigma value set for the filter was 0.5. The steps above represent the pre-processing step before the image is passed through the Dense-SIFT algorithm.

The Dense-SIFT function used in the script, ***vl_dsift,*** is also obtained from VLFeat.org. Dense-SIFT creates SIFT descriptors on a dense grid of locations in the image. The function requires a step size and a bin size parameter to initialise the grids on the images. In our case, these values were set to 30 and 20 pixels, respectively. These ultimately control the number of descriptors extracted from one image. Once these descriptors are extracted from all the training images, the next step would be to construct the vocabulary using a K-Means clustering algorithm. The *k* hyperparameter is set to the vocabulary size input from the ***codebook.m*** function. Again, a prebuilt function for the K-Means algorithm, ***vl_kmeans*** is taken from VLFeat.org. This ultimately produces the vocabulary for the Bag-of-Words algorithm in the form of the K-Means cluster centres. The following step would be to describe the input training images in terms of this vocabulary.

This step is performed by the ***bag_of_words.m*** function script. The script takes the image paths and the vocabulary generated by the previous step. Once again, the images are read and pre-processed in a similar fashion as in ***codebook.m.*** A Dense-SIFT algorithm is also run once again to produce the dense 128-dimensional SIFT descriptors for each image. This time, these descriptors are compared to the vocabulary descriptors, and the pairwise squared distance between each descriptor in the two groups are computed using VLFeat's ***vl_alldist2*** function. Each dense descriptor is described by the corresponding vocabulary descriptor that it is 'closest'. Following this, a histogram count of each vocabulary descriptor's occurrence in the image is performed and normalised. This eventually leads to a lower-dimensional spatially invariant representation of the input image in terms of the Bag-of-Words vocabulary. The ***bag_of_words.m*** function is run for both the training and validation images at this stage. Note that the validation images use the same vocabulary produced from the training images.

The next phase would be to train an SVM model with these features. We are again using a prebuilt SVM function ***vl_svmtrain***. To ensure our main script does not get cluttered, we allocate all the necessary training and classification steps on a separate ***svm.m*** function script. This script takes as input the train images and their labels, the test (in this case validation) images and a lambda hyperparameter to control the degree of penalisation for misclassified points. The script contains two instances of the ***vl_svmtrain*** function, with one to train the model and another to predict the validation set labels. The function outputs the predicted label for the validation set and returns the associated weights and bias vectors for the model. These will be used later on during the testing phase. Before testing, the final step is to compute the validation set predictions' accuracy using the ***get_accuracy.m*** function. The accuracy obtained is used to tune the hyperparameters, i.e., the vocabulary size and the lambda penalisation coefficient. The optimal of these are then passed through to the testing phase.

In the testing phase, each image in the test set is perturbed by nine different perturbations at 10 different intensities levels. Further details on these perturbations are described in Section 3.3. This means that the test set is replicated 90 times, with each version corresponding to a specific perturbation type at a specific level of intensity. Each of the 90 versions is passed through the same pipeline described above. Although, this time, using the already established vocabulary from the training images and describing each test image in this vocabulary with the ***bag_of_words.m*** function. This eventually yields 90 different test accuracies for this iteration of train, validation, and test set combinations. The complete procedure above, starting from paragraph 2, is then repeated for the remaining two combinations. At the end of this process, a 3-dimensional matrix of test accuracies for each fold, perturbation type, and perturbation intensity is produced. Further analysis of this data is illustrated in Section 4.0.

3.2 ResNet-18

Our implementation of choice for a deep learning classifier is the Resnet-18 CNN. A pre-trained version of the 18-layer convolutional neural network is available on MATLAB and used for this report's experiments. The model was trained on the ImageNet dataset, a large collection of annotated photographs widely used for computer vision research consisting of various general classes, including animals. We intend to perform a variation of transfer learning for our training purposes, where we freeze the weights of the convolutional layers of the pre-trained model and replace the final fully connected linear layer, which our specific dataset will train.

All code and functions described below will be contained within our ***resnet_main.m*** script. The first step in creating our deep learning classifier is to instantiate MATLAB's pretrained ***ResNet*** model. We then replace the fully connected layer and the output layer with new untrained layers with the appropriate number of classes (in our case 2). We employ a template function from MATLAB ***findLayersToReplace.m,*** which automatically identifies these two layers from the 'lgraph' of the neural network. The new layers are set to have a learning rate factor of 5 times the network's learning rate. This is to speed up training since there are fewer parameters to learn is this pre-trained network. The next step would be to freeze the weights of the convolutional layers. This is achieved by using the template function from MATLAB, ***freezeWeights.m*** and ***createLgraphUsingConnections.m***. The first function freezes the weights specified layers in a lgraph while the second takes in the modified layers object and its connections and creates a new lgraph. At this stage, we have our modified ResNet network ready for training on our image dataset.

The first step, which same as SIFT-BOW implementation, is to perform the fold splitting and train/validation split with the procedure described in section 1. The image paths are stored on MATLAB's 'ImageDatastore' objects. The main reason for doing so is to load the data sequentially in batches to allow for stochastic gradient descent training with the deep network. The ImageDatastore objects can also be augmented before training using MATLAB's ***imageDataAugmenter*** function, which can perform random translations, scaling, reflections, and rotations on the input images to introduce a form of regularisation. Note that these perturbations are not those tested in our robustness experiments. This function is also used to resize the images to 224x224x3 input size as required by the ResNet-18 model. Both the initialisation of the ImageDatastore object and the subsequent augmentations are performed in the ***create_augment_datastores.m*** function. Also, note that the random augmentations are only performed on the training images.

The training options for the modified network is then specified. The number of epochs set for training is three since the pretrained model will take less training time. The initial learning rate and minibatch size are set as a hyperparameter. Standard stochastic gradient descent is chosen as the optimiser, and the network is set to shuffle the training data for every epoch. The training and classification are carried out by MATLAB's ***trainNetwork*** and ***classify*** functions, respectively. The validation accuracy is, as before, used to tune the two hyperparameters mentioned. The optimal model, with the trained weights, is then presented for testing. The perturbed testing set is generated using the different transformation mentioned in Section 3.3. The ***classify*** function is called again to compute the test labels, and subsequently, the test accuracy can be determined for each of the 90 perturbed versions. The entire process mentioned above is repeated using the other two combinations of folds, and the results are collected for analyses.

3.3 Perturbations: Robustness Explorations

To test the classifier's robustness, the test image set is perturbed to 9 ways with ten different levels for each perturbation, resulting in 90 new sets for each test dataset. However, as level 1 is equivalent to the original data set, we have new 80 perturbed datasets for each test dataset. The brief description and how it is produced as below:

1. **Gaussian pixel noise** – Each pixel will be added with Gaussian distributed random number with ten levels of standard deviation from 2 to 18 with the increment of 2. To do that, after the image loaded to MATLAB, we have added a random number of gaussian distribution with a specific standard deviation to each pixel in each channel. Then, we check if any pixel exceeds 255 or less than 0. If yes, we set the pixel to 255 for pixel exceeding 255 and 0 for pixel with a value less than 0 accordingly. The full details of implementation can be found in ***gaussian_pixel_noise.m.***
2. **Gaussian blurring** – Each pixel is convolved with mask of (1/16)*[1,2,1;2,4,2;,1,2,1] by once up to 9 times. This method approximate Gaussian blurring with increasing standard deviation. This method is implemented by creating a filter followed by convolution up to 9 times using for loop. The details can be found in ***gaussian_blurring.m.***
3. **Images Contrast Increase –** The image contrast is increased by a factor of 1.03 to 1.27 with a 0.03 increment. This image is produced by multiplying each pixel in each channel with the contrast increase factor. Each pixel is capped at 255, in which any pixel that exceeds 255 is set to 255. Details can be found in ***image_contrast_increase.m.***
4. **Image Contrast Decrease –** Similarly, the image contrast decreases by a factor of 0.9 up to 0.1 with a 0.1 increment. Each pixel in each channel is multiplied by the contrast factor. No capping is required as pixel value will not exceed 255 or less than 0. Details can be found in ***image_contrast_decrease.m.***
5. **Image Brightness Increase** – The image brightness is increased by adding a constant to each pixel in each channel. The constant is 5 to 45 with an increment of 5. To do that, we simply add each pixel in each channel with the constant and cap the pixel value to 255 if it exceeds 255. The detailed implementation can be found in ***image_brightness_increase.m.***
6. **Image Brightness Decrease**– Similarly, the image brightness is decreased by subtracting a constant to each pixel in each channel. The constant is 5 up to 45 with an increment of 5. To do

that, we simply minus each pixel in each channel with the constant and cap the pixel value to 0 if it is less than 0. The detailed implementation can be found in ***image_brightness_decrease.m.***

7. **HSV Hue Noise Increase** – The perturbed test image set is produced by adding Gaussian random noise to the hue channel of HSV with the increasing number of standard deviations from 0.02 to 0.18 with 0.02 increment. To do that, images are converted to HSV format, and each pixel in the first dimension, which is hue, is added by a random number of Gaussian with a specified standard deviation. If the pixel value is more than 1 or less than 0 after adding a Gaussian random number, it is minus by 1 and set to 0 respectively to keep the number between 0 and 1. The implementation can be found in ***hsv_hue_noise_increase.m.***
8. **HSV Saturation Noise Increase** – Similarly, the perturbed test image set is produced by adding a normal random noise to the saturation channel of HSV with the increasing number of standard deviations from 0.02 to 0.18 with 0.02 increment. To do that, images are converted to HSV format, and each pixel in the second dimension, which is saturation, is added by a random number of gaussian with a specified standard deviation. If the pixel value is more than 1 or less than 0 after adding a Gaussian random number, it is clipped to 1 and 0, respectively. The implementation can be found in ***hsv_saturation_noise_increase.m.***
9. **Occlusion of the Image Increase –** Each image in the test set is occluded with a black pixel (0 value for all channel) with an increasing occlusion size from 5 X 5 occlusion up to 45 X 45 occlusion increment of 5 square lengths. The perturbation is implemented by building up the square using a random number and set the left and right and bottom and top boundary for the occlusion. Then, all the pixels in that occlusion are set to 0. Detailed implementation in ***occlusion_image_increase.m.***

To apply the perturbation, we take the test images path from ***trainvalidtest_paths.m*** and perturb all images. During each iteration, the image is perturbed to all level of one perturbation and saved. The process is repeated for all images and then repeated for all perturbations. The path for each image is saved in a cell array. The cell array then combined struct data type with each row representing one image, each column represents a perturbation level, and we have nine dimensions in struct representing each perturbation type. The detailed implementation is in ***perturbation_main.m.***

4.0 Results & Discussion

4.1 Hyperparameter tuning

Both classifiers have some hyperparameters to tune to determine an optimal model for the data. The primary tool to tune these parameters is the held-out validation set. The training accuracy will not be sufficient for this purpose, as attempting to boost its value will certainly result in overfitting. The result will be detrimental to the model's robustness and its ability to generalise to unseen data. It is also worth pointing out that the accuracy as a metric will be sufficient for our purpose since we have balanced classes.

At this stage of the report, two hyperparameters are identified for the SIFT-BOW SVM classifier. These are the vocabulary size and the penalisation coefficient for the SVM model. The vocabulary size effectively determines the number of cluster centres returned by the K-means algorithm. K-means itself is an unsupervised learning algorithm, and usually, its outputs cannot be 'tuned' since there is no ground truth to ascertain its performance. However, in our implementation, the outputs are subsequently utilised by a supervised classifier, the linear SVM, which can be used to tune a hyperparameter together with the validation set. The lambda penalisation coefficient controls the degree of misclassifications allowed while optimising the tuning parameters (the weights and biases) for the SVM model. A higher penalisation coefficient emphasises correctly assigning all labels to the training data, while a lower coefficient places more weight on obtaining the maximum margin when establishing the decision boundary.

The hyperparameter tuning performed in this report is by no means extensive. Although a finely tuned model will prove to be more robust during testing against perturbed images, the model is already performing at a decently high level of accuracy from the baseline results below. Any hyperparameter tuning is expected to improve validation results marginally. While the outcome may be different for the testing phase, such results cannot be used to tune the hyperparameters since this would not comply with the main reason for testing, which is to obtain a prediction of generalisation performance. The following table illustrates some of the hyperparameters' combinations with the SIFT-BOW SVM model with fold 1.

Table 1 Hyperparameter tuning for SIFT-BOW SVM model (fold combination 1)

Vocabulary Size	Penalisation Coefficient	Validation Accuracy
400	2.20E-05	88.69%
400	2.20E-04	90.70%
400	3.70E-04	90.45%
500	2.20E-05	88.19%
500	2.20E-04	89.95%
500	3.70E-04	90.45%
600	2.20E-05	90.70%
600	2.20E-04	91.21%
600	3.70E-04	89.95%
700	2.20E-05	87.44%
700	2.20E-04	89.95%
700	3.70E-04	91.46%

The general trend with the vocab size is somewhat as expected, where larger numbers lead to better model performance (with lambda adjustments) as more standard features are extracted from the training images to be used in the vocabulary. However, past 600 words, the improvements with larger vocabularies become marginal while the compute time increases. Hence, 600 words was chosen as the optimal size for this dataset. The other hyperparameter evaluated was lambda. A grid search methodology was applied to test out each combination of hyperparameters within a predefined search space. The chosen combination is highlighted in green in the table above. The weights and biases of this SVM model are saved to be used for a testing set later. Similarly, the vocabulary is also saved to test with the corresponding test fold. This process is repeated for the other fold combinations.

Two hyperparameters also needed to be set on the ResNet model, the learning rate and the minibatch size. As mentioned, the number of epochs for the training was set to 3. A baseline run with a learning rate of 1×10^{-4} and minibatch size 40 yielded the accuracy and loss plots during training showed in Figure 1. We can observe that accuracy and loss are plateauing during the third training epoch suggesting that 3 epochs should be sufficient for training this model. With this training period fixed, we performed the same hyperparameter evaluation using the held-out validation set and a grid search implementation as described for the previous model. The final set of tuned hyperparameters used for both models and all fold combinations are showed in Table 2.

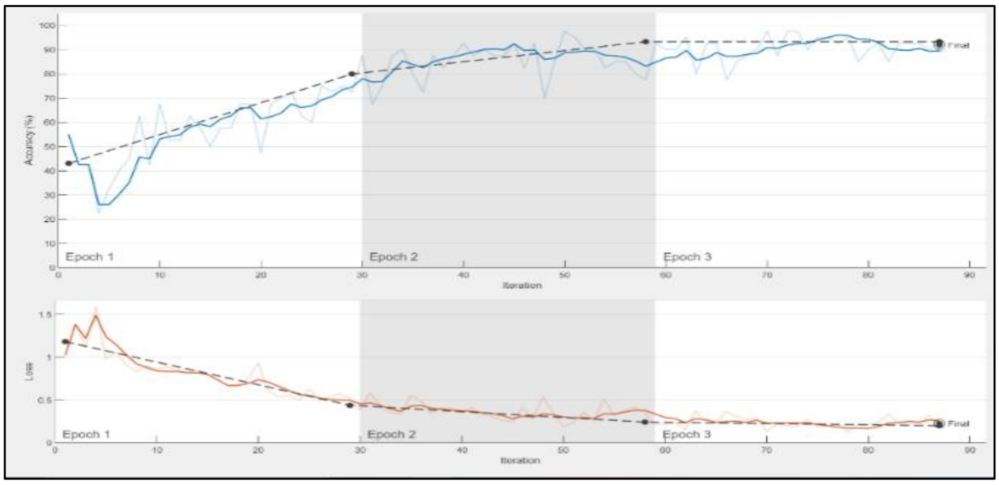

Figure 1 Accuracy and loss plots during training

Table 2 Tuned hyperparameters for each model and fold

Fold	**SIFT-BOW SVM**		**ResNet**	
	Vocabulary size	**Lambda**	**Minibatch size**	**Learning rate**
1	600	2.20E-04	40	8.00E-04
2	600	3.70E-04	40	8.00E-04
3	600	2.20E-04	40	1.00E-03

4.2 Robustness Exploration

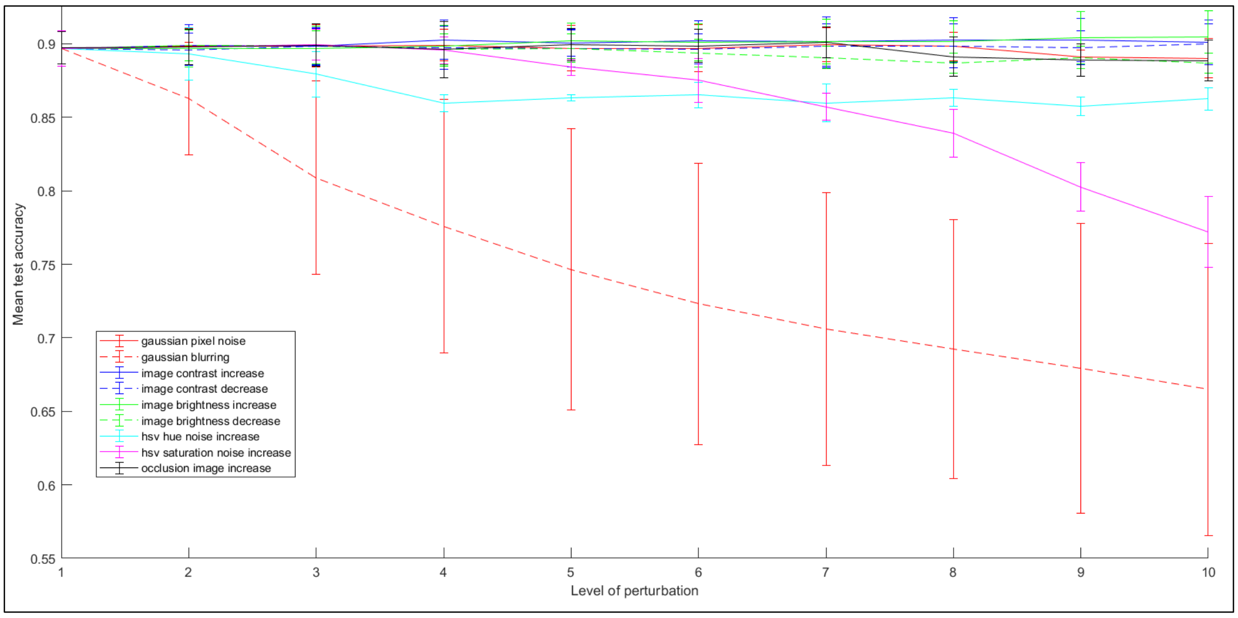

Figure 2 SIFT-BOW SVM Robustness exploration results

Figure 2 demonstrates the relationship between the test accuracy and the perturbations level applied for the SIFT-BOW SVM model for each of the 9 different perturbations tested. Since the test was conducted on 3 disjoint folds, we plot the results' average and include error bars representing the standard deviation. The immediate observation is that only three perturbations drastically deteriorate performance relative to the rest.

These are the Gaussian blurring, HSV hue increase and HSV saturation noise increase. The Gaussian blurring impact appears to have a large variance between different folds, as indicated by the error bars. The blurring and saturation noise continuously deteriorates performance with higher levels, while the hue noise impact's plateau after the fourth level. Meanwhile, the other 6 perturbations had a relatively minimal impact on the test accuracy at all 10 levels. A visual comparison can be made on the impact of some of the perturbations at their strongest level tested.

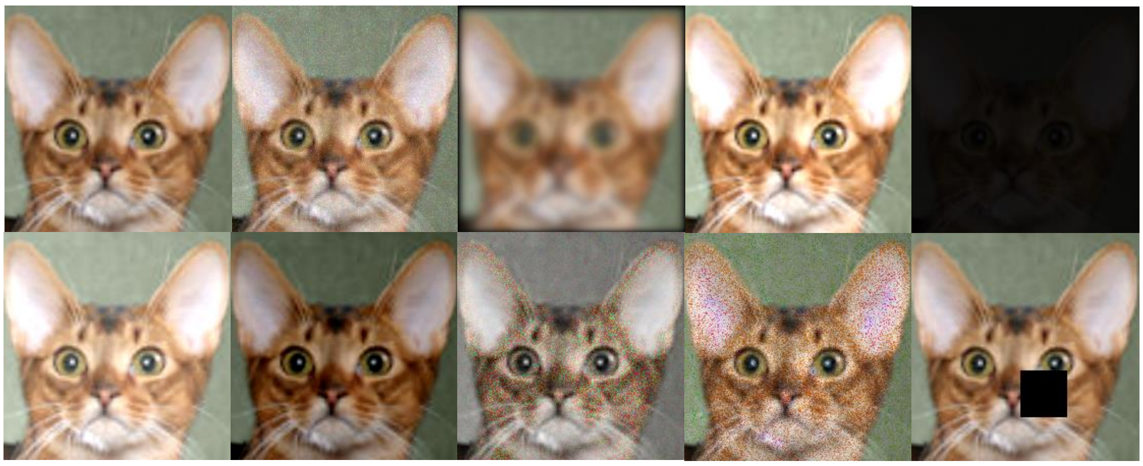

Figure 3 Original image with level 10 perturbations. From top left to bottom right: Original, Gaussian Pixel Noise, Gaussian Blurring, Contrast Increase, Contrast Decrease, Brightness Increase, Brightness Decrease, HSV Hue Noise, HSV Saturation Noise, Occlusion

From Figure 3, we can observe the drastic visual change to the original image when Gaussian blurring is applied to the 10th perturbation level. The change causes the objects' overall shapes and textures to be heavily distorted, dissolving the finer edges and corners. Since the descriptor employed in the SIFT-BOW algorithm depends strongly on these textures and their orientations, blurring them could eliminate the important class distinctive features identified in the BOW's kNN algorithm. If we consider the nose or whiskers of a cat to be one of the important vocabulary terms for the cat class, then from the example above, these features are almost entirely indiscernible from the distorted image. Hence, the BOW algorithm will struggle to describe these distorted images in terms of the training vocabulary. The same argument can also be made for the Gaussian blurring distortion, which similarly induced poor robustness.

Interestingly, the model is very robust again occlusion perturbations. This behaviour is consistent throughout the 3 folds, as shown by the narrow error bar, indicating that the occlusion boxes' random placement does not impact the model's performance. This provides us with confidence that the vocabulary features are fairly spread out in the image. Such results agree with existing literature with similar models in that moderate occlusion; is not detrimental to performance [4].

Figure 4 shows the effect of the perturbation on ResNet18 results. Without perturbation, ResNet achieve more than 95% mean test accuracy. With perturbations, ResNet performance is only significantly affected by blurring, noise, and contrast decrease. Other perturbations do not significantly affect performance as the mean test accuracy is above 90% for level 10 perturbations.

The finding is consistent, as in [5]. As stated in the paper, the blurring perturbations may cause the earlier layers' filter responses to change slightly, but the changes are propagating throughout the layer to create larger changes in the last layer. Meanwhile, the effect of noise is more apparent in the early layers and not so apparent in the last layer. Nevertheless, both perturbed images show much different responses in the last layer compared to the original image.

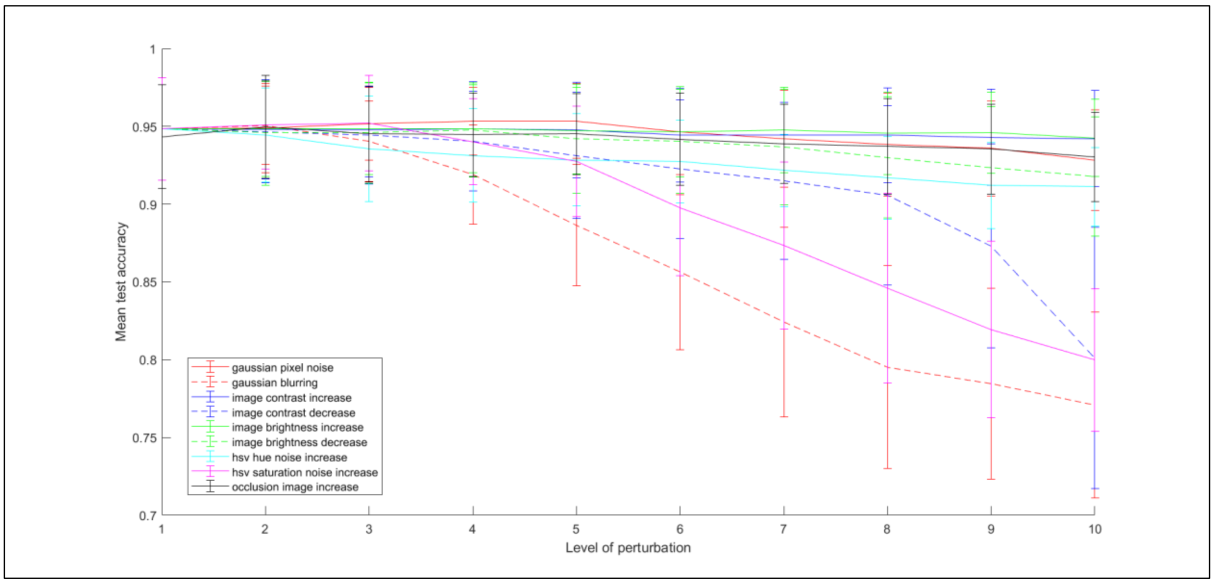

Figure 4 ResNet18 Robustness Exploration Result

The effect of noise is less for Gaussian Pixel noise applied to all RGB components than a single HSV component. We believe the smaller effect is due to noise applied to all components making the features are still captured by ResNet while applying to a single component, making it less captured by the model. HSV saturation noise yields the worst accuracy than HSV hue noise. This implies that saturation, which represents the intensity of the image, carries more significant weight in features representation while hue representing colour carries less weight as the model performs with a mean test accuracy of more than 90% for level 10 perturbation.

ResNet is also affected by contrast decrease. The mean test accuracy went down to approximately 85% for level 10 contrast decrease, which is the multiplication of 0.1 to all RGB components. As shown in Figure 3, the image with contrast decrease level 10 will not be easily recognisable by a human. Thus, we can consider that ResNet performance is good.

In comparison with the SIFT-BOW SVM model, both networks demonstrate the same poor robustness towards Gaussian blurring, HSV hue noise and HSV saturations noise perturbations. However, as mentioned, the ResNet also suffers from image contrast changes, more so with a decrease than an increase. Surprisingly, this behaviour was not exhibited by the SVM model even with the drastic image distortion shown in Figure 3. In this sense, the SVM model can be regarded as more robust against such perturbations. The SVM also showed more consistent and slightly better results with the saturation noise as indicated by it's a narrower error bounds and a less steeper descent than that of ResNet's. Besides these specific perturbations, both models are very robust towards the other perturbations, even at their strongest level in our testing. It is also important to note that by augmenting the dataset during training, we provide the ResNet with extra layer protection against overfitting and make it more robust against images that vary slightly from the training set. Another possible extension could be to compare these models without such augmentations to observe their impact. Also, since a pretrained ResNet model was employed, a direct comparison might be unfair to the SVM model since it was effectively trained on a larger training set. As mentioned, the ResNet was pretrained on the ImageNet dataset, containing much more images that include cats and dogs. However, this experiment could not have utilised a fresh ResNet model since the small size of the training set will risk overfitting. Future work could probably investigate these same models with a much larger dataset to allow an un-pretrained ResNet model to compete with the SVM model.

5.0 Conclusion

In this report, we implement and compare two image classification models against a series of perturbed images to evaluate their robustness. The two models are chosen from two different approaches in image classification. The SIFT-BOW SVM model represents the classical computer vision pipeline, while the ResNet- model represents a deep learning approach. Both models were tasked with the classification of cat and dog images, trained on clean training datasets and tuned to give the best validation performance. Then, both models were evaluated on a testing dataset that was perturbed in nine different ways, each at 10 different levels. As expected, the ResNet model achieved superior performance during training. Both models exhibit the same robustness (defined as the absence of a decrease in accuracy) against most perturbations except for Gaussian blurring, saturation noise and hue noise (to a smaller degree).

Interestingly, the ResNet model also showed some signs of sensitivity to a contrast decrease, a behaviour that was not observed in the SVM model. ResNet's saturation noise performance was also more unstable in terms of variance between different folds. Besides these, there was no other significant evidence that one model was more robust to perturbations over another. Future experiments can involve ablation studies relating to removing the data augmentations in the ResNet model and removing the pretrained element to observe its impact and testing a few recent methods in dealing with distorted images.

6.0 References

1. O’Mahony, N., Campbell, S., Carvalho, A., Harapanahalli, S., Hernandez, G.V., Krpalkova, L., Riordan, D. and Walsh, J., 2019, April. Deep learning vs. traditional computer vision. In Science and Information Conference (pp. 128-144). Springer, Cham.
2. Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25, pp.1097-1105.
3. He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
4. Morales, A., Ferrer, M.A., Diaz-Cabrera, M. and Gonzalez, E., 2013, October. Analysis of local descriptors features and its robustness applied to ear recognition. In 2013 47th International Carnahan Conference on Security Technology (ICCST) (pp. 1-5). IEEE.
5. Dodge, S. and Karam, L., 2016, June. Understanding how image quality affects deep neural networks. In *2016 eighth international conference on quality of multimedia experience (QoMEX)* (pp. 1-6). IEEE.

1.0 Appendix

svm_main.m

```
%% Preparation
clear
clc
warning('off','MATLAB:imagesci:png:libraryWarning')
run('vlfeat/toolbox/vl_setup')
data_path = '../catdog/';
test_accuracy = zeros(9,10,3);

%% Extracting fold and train/val split
fold = 1;
valid_ratio = 0.25;
fprintf('Getting paths and labels for all train, val & test data\n')
[train_paths, val_paths, test_paths, train_labels,...
    val_labels, pred_labels] = trainvalidtest_paths(data_path,
valid_ratio, fold);
validation_logs = [];

%% Hold-out hyperparameter tuning
% Hyperparameters
space_numwords = [400,500,600,700];
space_lambda = [2.2e-5,1.8e-4,2.2e-4,3.7e-4,2.2e-3];
for i = 1:size(space_numwords, 2)
    for j = 1:size(space_lambda, 2)
        num_words = space_numwords(i);
        LAMBDA = space_lambda(j);

        %-----------------------------------------------------------
        fprintf('Fold: %d \n',fold);

        % Step 1: Represent each image with bag of words
        fprintf('Using Bag of words representation for images\n')
        vocab = codebook(train_paths, num_words);

        train_image = bags_of_words(train_paths,vocab);
        val_image = bags_of_words(val_paths,vocab);

        % Step 2: Classify each test image by training and using the
appropriate classifier
        fprintf('Using SVM classifier to predict test set
categories\n');
        [predicted_val_labels, W, B] = svm(train_image,
train_labels, val_image, LAMBDA);

        % validation result
        val_accuracy = get_accuracy(predicted_val_labels,
val_labels);
        validation_logs = [validation_logs; num_words, LAMBDA,
val_accuracy];
    end
end
```

```
%% Evaluate perturbation & testing

% REMEMBER TO SET OPTIMAL W & B

fprintf('Fold: %d\n', fold)

%Performing Perturbations

% Evaluating validation accuracy
ptypes = fieldnames(perturbations);
% Loop through perturbations
for i = 1:numel(ptypes)
    % Loop through levels
    for j = 1:10
        perturbation = perturbations.(ptypes{i});
        pert_paths = perturbation(:,j);
        pert_image = bags_of_words(pert_paths,vocab);
        label_emtpy = zeros(1,size(pert_image,1));
        [~,~,~, scores] = vl_svmtrain(...
                            pert_image', label_emtpy, 0, 'model', W,
'bias', B, 'solver', 'none');

        pred_labels = cell(size(test_image,1),1);
        for k = 1:size(pert_image,1)
            if scores(k) > 0
                pred_labels{k} = 'cat';
            else
                pred_labels{k} = 'dog';
            end
        end
        test_accuracy(i,j,fold) = get_accuracy(predicted_val_labels,
val_labels);
    end
end

%% Test result analysis
av_test_accuracy = mean(test_accuracy,3);
std_test_accuracy = std(test_accuracy,0,3);

plot_test_results(av_test_accuracy, std_test_accuracy, ptypes);
```

trainvalidtest_paths.m

```matlab
function [train_image_paths, val_image_paths, test_image_paths, ...
    train_labels, val_labels, test_labels] =
trainvalidtest_paths(data_path, val_ratio, fold)


cat_dir = dir(fullfile(data_path, 'CATS/*.png'));
dog_dir = dir(fullfile(data_path, 'DOGS/*.png'));
sample_paths = cell(size(cat_dir,1) + size(dog_dir,1), 3);


idx = 1;
for i = 1:size(cat_dir,1)
    file = fullfile(data_path, 'CATS', cat_dir(i).name);
    sample_paths{idx,1} = file;
    species = regexp(file, 'cat_\d+', 'match');
    sample_paths{idx,2} = species{:};
    sample_paths{idx,3} = 'cat';
    idx = idx + 1;
end
 for i = 1:size(dog_dir,1)
    file = fullfile(data_path, 'DOGS', dog_dir(i).name);
    sample_paths{idx,1} = file;
    species = regexp(file, 'dog_\d+', 'match');
    sample_paths{idx,2} = species{:};
    sample_paths{idx,3} = 'dog';
    idx = idx + 1;
 end


group = sample_paths(:,2);
rng(1000);
cv = cvpartition(group, 'KFold', 3, 'Stratify', true);

train_group = sample_paths(cv.training(fold), :);
rng(1000);
hpartition = cvpartition(train_group(:,2), 'holdout', val_ratio,
'Stratify', true);
train_idx = training(hpartition);
val_idx = test(hpartition);

train_image_paths = train_group(train_idx, 1);
train_labels = train_group(train_idx, 3);
val_image_paths = train_group(val_idx, 1);
val_labels = train_group(val_idx, 3);
test_image_paths = sample_paths(cv.test(fold), 1);
test_labels = sample_paths(cv.test(fold), 3);
```

codebook.m

```
function dictionary = codebook(image_paths,num_of_words)
fprintf('\nvocabulary\n');
num = size(image_paths,1);
container = [];
step_p = 30;
binSize = 20;
for i = 1:num
    img = single(im2gray(imread(image_paths{i})));
    input_img = vl_imsmooth(img, 0.5);
    [~, sift_features] = vl_dsift(input_img,'Step',step_p,'size',
binSize,'fast');
    % NOTE THE TRANSPOSE
    container =[container;(single(sift_features'))];
end
fprintf('\nstart to building vocaulary\n')
% ONLY TAKE THE FIRST OUTPUT OF V1_KMEANS
dictionary = vl_kmeans(container',num_of_words);
fprintf('\nfinish building vocaulary\n')
```

bag_of_words.m

```
function image_feats = bags_of_words(image_paths,codebook)
vocab_size = size(codebook, 2);
fprintf('\nbags of words\n')

num = size(image_paths,1);
step_p = 30;
binSize = 20;
image_feats = zeros(num,vocab_size);
for i = 1:num
    img = single(im2gray(imread(image_paths{i})));
    input_img = vl_imsmooth(img,0.5);
    [~, sift_features] = vl_dsift(input_img,'Step',step_p,'size',
binSize,'fast');
    sift_features = single(sift_features);
    dist = vl_alldist2(sift_features,codebook);

    [~,index]=min(dist, [], 2);
    hist_v =histc(index,[1:1:vocab_size]);
    image_feats(i,:) = do_normalize(hist_v);
end
```

get_accuracy.m

```
function [accuracy] = get_accuracy(predicted,ground_truth)
correct = 0;
for i = 1:numel(predicted)
    correct = correct + strcmp(predicted(i), ground_truth(i));
end
accuracy = correct/numel(predicted);
end
```

resnet_main.m

```
%% Initialise ResNet & Replace Learnable/Output layer
clear
clc

net = ResNet;
inputSize = net.Layers(1).InputSize;
lgraph = layerGraph(net);
[learnableLayer,classLayer] = findLayersToReplace(lgraph);

newLearnableLayer = fullyConnectedLayer(2, ...
        'Name','new_fc', ...
        'WeightLearnRateFactor',5, ...
        'BiasLearnRateFactor',5);

lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);

figure('Units','normalized','Position',[0.3 0.3 0.4 0.4]);
plot(lgraph)
ylim([0,10])

layers = lgraph.Layers;
connections = lgraph.Connections;
layers(1:68) = freezeWeights(layers(1:68));
mod_lgraph = createLgraphUsingConnections(layers,connections);

data_path = '../catdog/';
test_accuracy = zeros(9,10,3);

%% Extracting fold and train/val split

fold = 1;
valid_ratio = 0.25;
fprintf('Getting paths and labels for all train, val & test data\n')
[train_paths, val_paths, test_paths, train_labels,...
    val_labels, pred_labels] = trainvalidtest_paths(data_path,
valid_ratio, fold);

validation_logs = [];

%% Perform Hold-out Hyperparameter Tuning
% Hyperparameters
space_mbs = [20,40,70];
space_lr = [1e-4, 8e-4,1e-3,8e-3];
for i = 1:size(space_mbs, 2)
    for j = 1:size(space_lr, 2)
    % Set Hyperparameters
    miniBatchSize = space_mbs(i);
    initial_lr = space_lr(j);
            % Initialising Optimiser
            options = trainingOptions('sgdm', ...
                'MiniBatchSize',miniBatchSize, ...
                'MaxEpochs',3, ...
                'InitialLearnRate',initial_lr, ...
                'Shuffle','every-epoch', ...
                'Plots','training-progress');

            % Choose to validate while training optional (for
illustration at cost of performance)
            validate_while_train = false;

        %------------------------------------------------------------
        % Train & evaluate network
        fprintf('Fold: %d\n', fold);

        % Augmenting data
        fprintf('Augmenting training data\n')
        augtrain_imds = create_augment_datastores(train_paths,
train_labels, true, inputSize(1:2));
        augval_imds = create_augment_datastores(val_paths,
val_labels, false, inputSize(1:2));

        % Input validation data
        if (validate_while_train)
            options.ValidationData = augval_imds;
            valFrequency =
floor(numel(augtrain_imds.Files)/miniBatchSize);
            options.ValidationFrequency = valFrequency;
        end

        % Training network
        mod_net = trainNetwork(augtrain_imds,mod_lgraph,options);

        % Evaluating validation accuracy
        fprintf('Evaluating network on validation data\n')
        [YPred,probs] = classify(mod_net,augval_imds);
        val_accuracy = mean(YPred == val_labels);
        validation_logs = [validation_logs; miniBatchSize,
initial_lr, val_accuracy];
    end
end

%% Evaluating Fold Test Results

% REMEMBER TO USE OPTIMAL MOD_NET

fprintf('Fold: %d\n', fold)

%Performing Perturbations

% Evaluating validation accuracy
ptypes = fieldnames(perturbations);
% Loop through perturbations
for i = 1:numel(ptypes)
    % Loop through levels
    for j = 1:10
        perturbation = perturbations.(ptypes{i});
        pert_paths = perturbation(:,j);
        pert_imds = create_augment_datastores(pert_paths, ...
            test_labels, false, inputSize(1:2));
        [YPred,~] = classify(mod_net,pert_imds,'Acceleration','none'
);
        test_accuracy(i,j,fold) = mean(YPred == test_labels);
    end
end


%% Test result analysis

av_test_accuracy = mean(test_accuracy,3);
std_test_accuracy = std(test_accuracy,0,3);

plot_test_results(av_test_accuracy , std_test_accuracy, ptypes);
```

create_augment_datastores.m

```
function [aug_imds] = create_augment_datastores(paths, labels,
train, dims)
imds = imageDatastore(paths,'Labels', categorical(labels));

pixelRange = [-30 30];
scaleRange = [0.9 1.1];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange, ...
    'RandXScale',scaleRange, ...
    'RandYScale',scaleRange, ...
    'RandRotation',[0 45]);

if (train)
    aug_imds = augmentedImageDatastore(dims,imds,
'DataAugmentation',imageAugmenter,...
        'ColorPreprocessing', 'gray2rgb');
else
    aug_imds = augmentedImageDatastore(dims,imds,
'ColorPreprocessing', 'gray2rgb');
end

end
```

perturbation_main.m

```
function [perturb] = perturbation_main(test_paths)
%PERTURBATION_MAIN Run all perturbations for all level for one set
of
%dataset
%   The function take nx1 cell array containing all file path for
the
%   dataset and return struct n X level X #of perturbations.
perturb = struct;

perturb.gaussian_pixel_noise = gaussian_pixel_noise(test_paths);
perturb.gaussian_blurring = gaussian_blurring(test_paths);
perturb.image_contrast_increase =
image_contrast_increase(test_paths);
perturb.image_contrast_decrease =
image_contrast_decrease(test_paths);
perturb.image_brightness_increase =
image_brightness_increase(test_paths);
perturb.image_brightness_decrease =
image_brightness_decrease(test_paths);
perturb.hsv_hue_noise_increase = hsv_hue_noise_increase(test_paths);
perturb.hsv_saturation_noise_increase =
hsv_saturation_noise_increase(test_paths);
perturb.occlusion_image_increase =
occlusion_image_increase(test_paths);
end
```

gaussian_pixel_noise.,m

```
function [perturbed_path] = gaussian_pixel_noise(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'gaussian_pixel_noise';
perturbed_path = {};
stdev = 0:2:18; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
```

```
        %apply perturbation
        num = uint8(stdev(level).*randn(size(img)));
        imgNew = img + num;
         %cap value between 255 and 0
        imgNew(imgNew > 255) = 255;
        imgNew(imgNew < 0) = 0;

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

gaussian_blurring.m

```
function [perturbed_path] = gaussian_blurring(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'gaussian_blurring';
perturbed_path = {};
mask_level = 0:1:9; %range of perturbation level
filter = 1/16*[1,2,1;2,4,2;1,2,1];

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %initialize imgNew for 0 masking
    imgNew = img;

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
```

```
        %apply perturbation
        for x = 1:1:mask_level(level)
            imgNew = imfilter(img,filter,"conv");
            img = imgNew;
        end

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

image_contrast_increase.m

```
function [perturbed_path] = image_contrast_increase(img_paths)
%This function take file path image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'image_contrast_increase';
perturbed_path = {};
contrast_level = 1.0:0.03:1.27; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        imgNew = img.*contrast_level(level);
        imgNew(imgNew > 255) = 255;

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

image_contrast_decrease.m

```
function [perturbed_path] = image_contrast_decrease(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'image_contrast_decrease';
perturbed_path = {};
contrast_level = 1.0:-0.1:0.1; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        imgNew = img.*contrast_level(level);
        imgNew(imgNew > 255) = 255;
        imgNew(imgNew < 0) = 0;

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);
        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

image_brightness_increase.m

```
function [perturbed_path] = image_brightness_increase(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.
%%Set parameter
pert = 'image_brightness_increase';
perturbed_path = {};
brightness_level = 0:5:45; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        imgNew = img + brightness_level(level);
        imgNew(imgNew > 255) = 255;

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end
    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

image_brightness_decrease.m

```
function [perturbed_path] = image_brightness_decrease(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'image_brightness_decrease';
perturbed_path = {};
brightness_level = 0:5:45; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        imgNew = img - brightness_level(level);
        imgNew(imgNew < 0) = 0;

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

hsv_hue_noise_increase.m

```
function [perturbed_path] = hsv_hue_noise_increase(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'hsv_hue_noise_increase';
perturbed_path = {};
noise_level = 0.00:0.02:0.18; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    imgrgb = (imread(img_path));
    imghsv = rgb2hsv(imgrgb);
    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        [x,y,~] = size (imghsv);
        num = noise_level(level).*randn(x,y);
        imghsv(:,:,1) = imghsv(:,:,1) + num;
        imgNew = imghsv;
        %Set hsv value between 0 and 1
        imgNew(imgNew > 1) = imgNew(imgNew > 1) - 1;
        imgNew (imgNew <0) = 0;
        %convert hsv to rgb
        imgNew = hsv2rgb(imgNew)

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

hsv_saturation_noise_increase.m

```
function [perturbed_path] = hsv_saturation_noise_increase(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'hsv_saturation_noise_increase';
perturbed_path = {};
noise_level = 0.00:0.02:0.18; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    imgrgb = (imread(img_path));
    imghsv = rgb2hsv(imgrgb);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        [x,y,~] = size (imghsv);
        num = noise_level(level).*randn(x,y);
        imghsv(:,:,2) = imghsv(:,:,2) + num;
        imgNew = imghsv;
        imgNew(imgNew > 1) = 1;
        imgNew (imgNew <0) = 0;
        imgNew = hsv2rgb(imgNew);

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

occlusion_image_increase.m

```
function [perturbed_path] = occlusion_image_increase(img_paths)
%This function take file path for image and returned path of
perturbed
%image
%   The function will load one image, apply perturbation for all
level and
%   save img path in one row. Repeat for all image.

%%Set parameter
pert = 'occlusion_image_increase';
perturbed_path = {};
square_length= 0:5:45; %range of perturbation level

display_progress(pert);

for img_post = 1:1:size(img_paths,1)
    %read image file
    img_path = img_paths{img_post};
    img = imread(img_path);

    %create empty cell array for perturbed image path for ONE image
    perturbed_path_level = {};

    %Apply perturbation at all level for one image and save image
and path
    for level = 1:1:10
        %apply perturbation
        [x,y,~] = size (img);
        left_x = randi(x - square_length(level));
        right_x = left_x + square_length(level);
        top_y = randi (y - square_length(level));
        bottom_y = top_y + square_length(level);
        imgNew = img;
        imgNew (left_x:right_x,top_y:bottom_y,:) = 0;

        %Create filename to save new image and path
        img_name = create_img_name(img_path);
        [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level);

        %save file using new file name
        imwrite(imgNew,img_perturbed_path);

        %save path of new image for all level for ONE image
        perturbed_path_level=
[perturbed_path_level,img_perturbed_path];
    end

    %save path of all perturbed image to overall cell
    perturbed_path = [perturbed_path;perturbed_path_level];
end
```

create_img_name.m

```
function [img_name] = create_img_name(img_path)
%CREATE_IMG_NAME Convert image path to image name
%   Create image name from image path by taking out the location
folder and
%   file typ in the path
img_name = replace(img_path(16:end),'.png','');
end
```

create_img_path.m

```
function [fullFilename,img_perturbed_path] =
create_img_path(img_name,pert,level)
%CREATE_IMG_PATH Create fullFilename to save file and path
%   Create full file name by including pertubation type and level
in.
%   Create full path by including folder location in file.
baseFilename = img_name+"_" + pert+"_"+ "level"+level + ".png";
fullFilename = fullfile(pert,baseFilename);
img_perturbed_path =
char(fullfile('..','perturbation',fullFilename));
end
```

plot_test_results.m

```
function plot_test_results(av_accuracy,std_accuracy, ptypes)
%PLOT_TEST_RESULTS Summary of this function goes here
%   Detailed explanation goes here

figure
hold on;
linespec = {'r', '--r', 'b', '--b', 'g', '--g', 'c', 'm', 'k'};
for i = 1:size(av_accuracy,1)
    x = 1:10;
    y = av_accuracy(i,:);
    err = std_accuracy(i,:);
    errorbar(x,y,err, linespec{i})

end
xlabel('Level of perturbation');
ylabel('Mean test accuracy');
legends = ptypes;
for i = 1:numel(ptypes)
    legends{i} = strrep(ptypes{i}, '_', ' ');
end
legend(legends)
end
```