# Introduction

Image classification falls under the supervised approach of assigning labels (or probability of labels) from a predefined group of classes to input images. Its current applications extend across various domains, including autonomous driving vehicles, medical imaging, and remote sensing [1]. The traditional methodology of image classification in computer vision would usually involve 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 Neigbours (KNN). Feature extraction represents an important element of the process since merely 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 [2]. Other popular approaches in feature generation also include using a Bag of Words (BOW) feature representation that is generated with a K-Means clustering algorithm.

With the introduction of AlexNet [3] 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 performance of the deep learning models far supersedes that of classical computer vision, albeit with trade-offs in computational requirements [2]. In contrast to the traditional computer vision pipeline above, these methods employ an end-to-end learning process, which entirely skips the engineering step of feature generation. In this sense, the model accepts as input, the full image with its corresponding label annotation and performs the training process without requiring any handcrafted features. 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 [2]. 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. Specifically, the development of Convolutional Neural Networks (CNN) has greatly influenced the field of image classification, making use of its kernels or filters to detect these learned features throughout the image.

Although the recent developments in neural networks for image classification has shown tremendous results, there is an intrinsic flaw where the features learned during the end-to-end training process is specific to the dataset at hand. If these are poorly constructed, then 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.

In this report, we will investigate two approaches to image classification, a classical computer vision model involving on a SIFT-BOW feature representation trained with a linear SVM model and deep neural net model based off the ResNet-18 CNN architecture [4]. 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. The models, with their optimal hyperparameters will then undergo a final round of testing with a ‘perturbed’ testing set as a measure of their robustness towards a more diverse set of images. Further details about the dataset and robustness explorations will be given in the following sections.

# 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 of the cat and dog species, making the dataset fairly class balanced. Each image is labelled with their respective class and subclass. For instance, cat\_1 refers to the image of a cat belonging to species class 1. Its important to note, that the classification task does not involve classifying each animal (cat or dog) into their respective species. The information on the species 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 3 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 3 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.

# Methodology

## SIFT-BOW

Our first classifier represents an approach from the classical computer vision. Our feature generation tool of choice is a Dense SIFT Bag of Words algorithm. The standard machine learning classifier chosen is a soft linear Support Vector Machine model. Before proceeding to this stage, the total dataset is split into its 3 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 combination, and the results of each are stored for further analysis.

The first step in the SIFT-BOW feature representation is to generate a ‘image vocabulary’ based on training images in this iteration. This is carried out by ***codebook.m*** function script which takes as input the training image paths and the vocabulary size. The vocabulary size is treated as a hyperparameter in this report, although theoretically its effect on performance is fairly straightforward in that the model’s performance is a monotonic function of this vocabulary size. That said, increasing this parameter will affect the training time and so a balance is required between model performance and computational load. 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 in 2 inputs, 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’ to. Following this, a histogram count of occurrence of each vocabulary descriptor in the image is performed and normalized. This eventually leads to 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 the both the training and validation images at this stage. Note that the validation images use the same vocabulary produced from the training images.

Finally, the input data for the linear SVM classifier is ready and training can commence. We are again using a prebuilt SVM function from VLFeat.org, ***vl\_svmtrain***. This function can perform both the training and inference phase of the SVM model. To ensure our main script does not get cluttered, we allocate a contain all the necessary steps for training and classification on separate ***svm.m*** function script. This script takes as input the train images and its labels, the test (in this case validation) images and a lambda hyperparameter to control 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 labels for the validation set. 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. The final step before testing is to compute the accuracy of the validation set predictions. This is computed using the simple ***get\_accuracy.m*** function script, which was created to minimise clutter in the main script.

The accuracy obtained above is used to tune the hyperparameters mentioned before i.e., the vocabulary size and the lambda penalisation coefficient. Each step starting from paragraph 2 in this section is repeated with different sets of hyperparameters, with the final validation accuracy achieved stored for comparison. The hyperparameter tuning was kept brief, with only a handful of combinations explored. The optimal of out these is then passed through to the testing phase.

In the testing phase, each image in the test set is perturbed by 9 different types of perturbations each at 10 different levels of intensities. 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 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 2 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 on this data is illustrated in Section 4.

## ResNet-18

Our implementation of choice for a deep learning classifier is the Resnet-18 CNN. A pretrained version of the 18-layer convolutional neural network is available on MATLAB and will be used for the experiments in this report. 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 [5]. For our training purposes we intend to perform a variation of transfer learning, where we freeze the weights of the convolutional layers of the pretrained model and replace the final fully connected linear layer, which will be trained by our specific dataset.

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 ***resnet18*** model. We then proceed to 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 2 layers from the ‘lgraph’ of the neural network. The new layers are set to have a learning rate factor of 5 times the learning rate specified by the network. This is to speed up training since there are less parameters to learn is this pretrained network. The graph of the modified network is shown in Figure 1. Note the ‘skipped connection’ architecture that is characteristic to ResNets.

Table

Description automatically generated with medium confidence

Figure 1 Modified ResNet18 Architecture

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. By freezing the weights in a layer, the learning rate factor applied to that layer is 0. At this stage we have our modified ResNet network ready for training on our image dataset.

The first step as in the previous SIFT-BOW implementation is to perform the fold splitting and train/validation split with the procedure described in section 2. However, these images are never read into memory, but instead their paths (and corresponding labels) are stored in MATLAB’s ‘ImageDatastore’ objects. The main reason for doing so is the ability 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 prior to 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. 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. Note that the random augmentations are only performed on the training images.

Following this, the input for the model is ready and training options for the modified network is specified. The number of epochs set for training is 3, since the pretrained model will take less training time. The initial learning rate and minibatch size is set as a hyperparameter. A 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. As with the validation set, the testing set will be initialised using an ImageDatastore object and resized appropriately. The ***classify*** function is called again to compute the test labels and subsequently the test accuracy can be determined for each of 90 perturbed versions. The entire process mentioned above is repeated by using the other 2 combinations of folds and the results are collected for analyses.

## 3.3 Perturbations: Robustness Explorations

# Results & Discussion

## Hyperparameter tuning

Both classifiers have some hyperparameters to tune to determine an optimal model for the data. The main 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 of which will be detrimental to 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 assurance of the balance in 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 together with the validation set can be used to tune a hyperparameter. The lambda penalisation coefficient controls the degree of misclassifications allowed while optimizing the tuning parameters (the weights and biases) for the SVM model. A higher penalisation coefficient emphasises the preference in 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.

As mentioned previously, 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, from the baseline results below, the model is already performing at a decently high level of accuracy. Any hyperparameter tuning is expected to marginally improve validation results. 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 of testing which is to obtain a prediction of generalisation performance. The following table illustrates some of the combinations of hyperparameters tested with the SIFT-BOW SVM model with fold 1.

|  |  |  |
| --- | --- | --- |
| **Vocabulary Size** | **Penalisation Coefficient** | **Validation Accuracy** |
| 100 | 2.20E-04 | 83.42% |
| 200 | 2.20E-04 | 86.18% |
| 300 | 2.20E-04 | 89.70% |
| 400 | 2.20E-04 | 87.69% |
| 500 | 2.20E-04 | 90.45% |
| 600 | 1.20E-04 | 89.95% |
| 600 | 1.80E-04 | 90.20% |
| 600 | 2.20E-04 | 91.46% |
| 600 | 2.70E-04 | 89.95% |
| 600 | 3.00E-04 | 89.70% |
| 700 | 2.20E-04 | 91.71% |
| 800 | 2.20E-04 | 91.46% |

Table 1 Hyperparameter tuning for SIFT-BOW SVM model

The general trend with the vocab size is as expected where larger numbers do lead to better model performance as more and more common features are extracted from the training images to be used in the vocabulary. There is one exception where the 300-word configuration delivered better performance than the larger 400. However, past 600 words the improvements with larger vocabularies become marginal while the compute time continues to increase. Hence, 600 words was chosen as the optimal size for this dataset. Using the 600-words configuration the other hyperparameter was, lambda, was manipulated. The parameter yielded the optimal result at with a validation accuracy of 91.46%. The weights and biases of this SVM model are saved to be used for testing set later. Similarly, the vocabulary for fold 1 is also saved.

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 and minibatch size 40 yielded the accuracy and loss plots during training showed in Figure 2.

Graphical user interface, application

Description automatically generated

Figure 2 Accuracy and loss plots during training

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 as before. This was repeated for each one of the 3 fold combinations. The final set of optimal hyperparameters used for both models and all fold combinations are as follows.

## 4.2 Robustness Exploration

# 5.0 Conclusion