Computer Vision 1 - Final Project Report

Jedda Boyle
11398221

Mathijs Mul
6267939

1. INTRODUCTION

In this report we address the two tasks of the final project. The first part is a Bag-of-Words based image classification system, and the second part is a Convolutional Neural Network for image classification. The two parts are discussed in two separate sections. For both parts, we discuss relevant details of the implementation, and results that were obtained.

2. PART 1: BAG-OF-WORDS IMAGE CLASSIFICATION

For this part, a system for image classification is implemented, which should tell if there is an object of a given class in an image. Four-class image classification is performed, for airplanes, motorbikes, faces and cars. The approach is the so-called 'Bag-of-Words' or 'Bag-of-Features' approach. Data are taken from the Caltech4 data set, which has been distributed and divided into separate training and test folders.

2.1 Implementation

The Bag-of-Words image classification system comprises the following steps:

I Feature extraction and description

II Building a visual vocabulary

III Quantize features using visual dictionary

IV Representing images by frequencies of visual words

V Classification

Step I, feature extraction and description, is actually not only chronologically the first step of the algorithm, because at different stages of the process features have to be identified and the corresponding descriptors determined. The function `get_features.m` performs this task throughout the entire process. The function returns the descriptors given some image, SIFT mode and color space.

SIFT is the method by which features are extracted in our implementation, for which we use the VLFeat library. For the identification of feature points, two options are available: dense SIFT or keypoint SIFT. Dense SIFT does not perform any keypoint analysis and simply takes the descriptors of fixed points in the image, e.g. by centering a block around every nth pixel. In our implementation, for dense SIFT the step size is set to 5 pixels, and the blocks are 2×2 pixels in size. Keypoint SIFT does not perform such dense sampling, but instead finds a limited number of salient points first, for which the descriptors are computed. Hence, keypoint SIFT will typically result in fewer descriptors than dense SIFT.

In our implementation, SIFT is available for the following color spaces: gray, RGB, rgb and opponent color space. For grayscale, feature points are determined with respect to a grayscaled version of an image, and the descriptors are computed with respect to these points. For the different color spaces, the following procedure is carried out:

1. Find keypoints in the grayscale version of the image (generated with built-in MATLAB function `rgb2gray`)
2. Convert the image into the alternative color space
3. For each channel in (2), by using the keypoints found in step (1), calculate descriptors separately
4. Concatenate the descriptors to a single larger descriptor

For RGB color space, the channels are just the given color channels of an image. For rgb (normalized RGB), the following transformation is carried out:

$$\begin{bmatrix} O_1 \\ O_2 \\ O_3 \end{bmatrix} = \begin{bmatrix} \frac{R}{R+G+B} \\ \frac{G}{R+G+B} \\ \frac{B}{R+G+B} \end{bmatrix}$$

For opponent color space, this transformation is applied:

$$\begin{bmatrix} O_1 \\ O_2 \\ O_3 \end{bmatrix} = \begin{bmatrix} \frac{R-G}{\sqrt{2}} \\ \frac{R+G-2B}{\sqrt{6}} \\ \frac{R+G+B}{\sqrt{3}} \end{bmatrix}$$

The descriptors themselves are identified with respect to locations in individual channels, and are 128×1-shaped vectors each. For an area of 4×4 pixels, they store the orientation of the directed gradients as normalized histograms with 8 bins, so that $8 \times 16 = 128$ values are registered. Hence, if the descriptors for three color channels are concatenated, the resulting descriptors become vectors with $128 \times 3 = 384$ entries.

From step II onwards, our implementation of the Bag-of-Words classification system is best explained by means of the demo file, `demo.m`, which runs the entire system in a number of different sections. For Step II, we have to build the so-called 'visual vocabulary' (or 'visual dictionary' or

'codebook'). This is a collection of 'visual words' (or 'codewords'), which are obtained by clustering feature descriptors. The visual words are the centroids of these clusters.

Constructing the visual vocabulary takes place in the function `create_dictionary.m`. This is the first function that is called in the demo file. It takes as its first input a vocabulary batch size, which is the number of images from the different training sets that are used to construct the vocabulary. This number is the same for all four image categories, so that the resulting subset of the training images contains an equal number of images from all categories. The images in this set will not be used for training later on. The function also needs the vocabulary size as a parameter. This is the number of visual words that the visual vocabulary must contain. It also has to know the SIFT mode and the color space that is used.

Now, the actual construction of the visual vocabulary takes place. For each of the four image categories, the required number of training images is used.[1] For all of the images that are used in this step, the features are extracted by `get_features.m` according to the above procedure, with SIFT mode and color space as indicated by the parameters of the function `create_dictionary.m`. All the descriptors are concatenated into one big matrix of dimensions $D \times N$, where D is the size of the descriptors (128 for grayscale or 384 for the different color spaces), and N is the number of descriptors that were extracted, i.e. the sum of the numbers of feature points identified in all pictures considered in this step. These descriptors have to be clustered in order to find the visual words. The clustering method that is used in our implementation is the most standard one, namely k-means. This method divides the data into k different clusters, and hence the parameter k equals the vocabulary size. The cluster centroids are the visual words that are stored in the visual vocabulary. For k-means, the `vl_kmeans` function of the VLFeat library is used. Dictionaries are stored in the directory `part1_vocabs`.

The next stage is step III: quantizing features using the visual dictionary constructed in the preceding step. This step is carried out in the function `image2hist.m`. Given some input image, this function first extracts the features of the image according to a given method (depending on SIFT mode and color space), as described for Step I. Next, each descriptor is assigned to the closest visual word from the vocabulary. The closest visual word is understood here as the centroid of the cluster that has the smallest Euclidean distance to the concerned descriptor.

Step IV, representing images by frequencies of visual words, is carried out in the same function. After the closest visual words for an image have been identified, a histogram is constructed that stores the number of times particular words in the vocabulary were found to be closest to a descriptor of the image. Hence, for a vocabulary size k (with k clusters), this histogram will contain k bins with counts. These counts are normalized so that they sum up to 1. The image is now represented in terms of the relative frequencies of the closest visual words. The histogram that expresses this information is a $k \times 1$-vector that can be used for training purposes in the next stage.

What comes next is step V: classification. For this part, a Support Vector Machine (SVM) classifier is trained per object class. This means that we will end up with four different classifiers. For training the classifiers, a set of images (with the size of the training batch) is taken from the training set of each class. As said before, this set must be disjoint from the collection of images that were used to construct the visual dictionary in the earlier step. The function `construct_train.m` does this and constructs the training data in the required format. It iterates over the training sets of the different classes, and per class calls the function `data_folder.m`. This function traverses each individual training directory, starting with the first picture that was not used for vocabulary construction and ending once the train batch size (per class) has been reached. Per image the `image2hist.m` function is used to construct the histogram vector representing the relative frequencies of the codewords, which is the information we will use to train the classifiers.

For the actual training, four classifiers are used. This happens in the training section of the demo file. The training data are available, and do not change per classifier. What we still need are sets of labels, which do change per classifier, because the class of each will determine which training instances serve as positive examples, and which as negative ones. The training labels can easily be constructed, because the training sets are traversed in a fixed order and are segmented into equal-sized subsets.[2] For a train batch size of n, the total set of labels will have size $4n$, with $3n$ negative labels (-1) and n positive labels (+1). The position of the positive and negative labels follows directly from the class of the classifier that is currently being trained.

The LIBLINEAR library is used for the SVM's. The standard configurations are used, namely the ones for a linear SVM with L2-loss function. The library accepts no parameters for changing the kernel.

All of the models that are trained for different parameter settings are stored in the folder `part1_models`. From this directory they can be loaded again in order to assess their performance with respect to the test set. This happens in the testing section of the demo file. First, the images in the test directories are processed by the function `construct_test` in roughly the same way as was previously done by `construct_train`: the test folders are traversed so as to translate the images into the required representation.

For each of the classes, the respective models are tested with these test data. The prediction function of LIBLINEAR outputs the predicted labels of the test images (which for the individual classifiers is a binary classification), the accuracy of the prediction with respect to the actual labels of the test images and the probability estimates. For the purpose of this project, the last quantities are most relevant, because they express the probability with which the test images can be assigned to the class of the classifier in question. As we want to come up with a ranking of the images in the test set, these probabilities are most informative. All results are stored in the folder `part1_results`.

The final section of the demo file constructs the HTML file for the test results according to the required format. The main functionality for this part is the file I/O that goes on in `make_html.m`. In this file, the function `evaluate_scores.m` is called, which determines the predicted ranking by order-

[1] Grayscale images are skipped, because almost all pictures are in full RGB, and hence taking these anomalies into account could corrupt the results.

[2] For training an SVM, no randomization of the training data is required, so the order can be kept constant.

ing the probability estimates from high to low, and which computes the Average Precision $AP(c)$ of the different classifiers. For a single class c, this measure is defined as follows:

$$AP(c) = \frac{1}{m_c} \sum_{i=1}^{n} \frac{f_c(x_i)}{i},$$

where n is the total number of test images ($n = 50 \times 4 = 200$), m_c is the number of images of class c ($m_c = 50$ for all c), x_i is the ith image in the ranked list $X = \{x_1, x_2, \cdots, x_n\}$ and f_c is a function that returns the number of images of class c in the first i images if x_i is of class c, and 0 otherwise. The Mean Average Precision (MAP) of the system is the mean of the AP scores for the different classifiers.

The HTML result files contain the complete ranking for all classifiers, and must be opened in the same directory as the Caltech4 data set, because otherwise the HTML path references do not work anymore. For the sake of organization, however, they are stored in the directory `part1_htmls`.

2.2 Results

Results of the system can be evaluated both qualitatively and quantitatively. Qualitatively, the rankings predicted by the classifiers can be inspected. For a classifier of class c, we would then like to see images of class c high up in the ranking. Quantitatively, we can make use of the notions of AP and MAP defined above.

There are many different parameter settings for which it would be worthwhile to report and compare the results. In particular, we wish to assess the impact of varying settings for the following aspects: keypoint vs dense SIFT sampling, vocabulary size, SIFT color space, number of images used for vocabulary construction and number of training samples.[3]

More specifically, we will study the influence of changing the above-mentioned parameters in the following range:

parameter	*values*
SIFT mode	[**keypoint**, dense]
vocabulary size	[**400**, 800, 1600, 2000, 4000]
color	[**RGB**, gray, rgb, opponent]
number of images used for vocabulary construction (per class)	[**50**, 100, 250]
number of training samples (per class)	[**100**, 200, 300]

Table 1: Parameter settings

Studying all possible combinations would require training 1,440 different models, which is not feasible. Hence, we study the effect of changing the different parameters, while keeping the other parameters constant to a 'default' setting. The default value for each of the parameters is displayed in bold in the above table. The baseline is considered to be the case where all parameters are set to their default values.

[3]As noted before, the recommended LIBLINEAR library does not allow us to change the used kernel, so without changing the library we cannot compare the impact of different kernel choices.

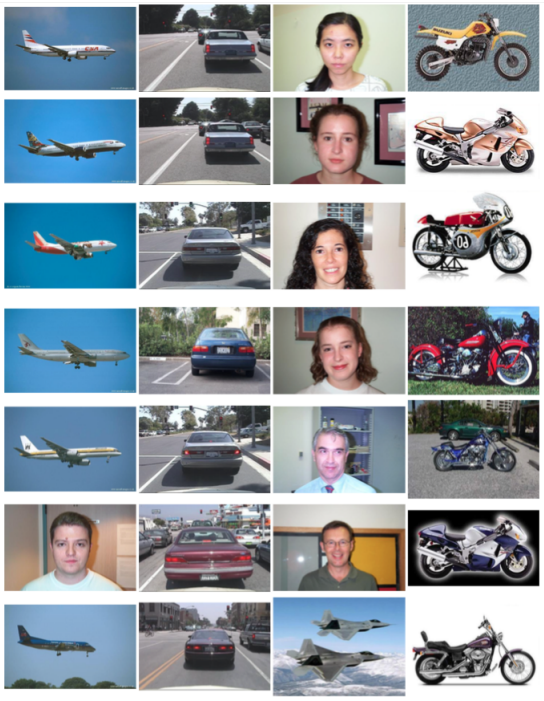

Figure 1: Baseline ranking

Running the entire system for the default settings (keypoint SIFT, 400 visual words, RGB color space, 50 images used for vocabulary construction per class and 100 training instances per class), we obtain the following baseline results:

AP				MAP
airplanes	cars	faces	motorbikes	
0.85686	0.77914	0.67875	0.80088	0.77891

Table 2: Baseline results

The baseline results vary a lot per class: Average Precision is more than 0.85 for airplanes, whereas for faces it is not even 0.7. For cars and motorbikes, Average Precision is in the range 0.8. The MAP score is slightly lower than 0.8. These scores are not shockingly bad, but it seems reasonable to expect that they are not optimal either. At the same time, it is important not to focus only on the quantitative evaluation, but also to consider the qualitative aspect. So we refer to Figure 1, which shows the top results of the rankings predicted by the different classifiers. The left ranking is given by the airplanes classifier, the second one by the cars classifier, the third one by the faces classifier and the right one by the motorbikes classifier. We see that the classifiers make at most one mistake in these top rankings; cars and motorbikes make none. As the system has to be evaluated with respect to these rankings, and most users focus on the top results of a ranking, we see that a qualitative perspective shows that the classifiers may not be as bad as the MAP scores suggest.

Nevertheless, one would still expect the classifiers to be capable of a better performance for other settings. Here, we note that the baseline only takes a rather small vocabulary

size, only few images were used to construct it and no more than 100 training samples per class were used to train the classifiers. With the other parameters kept constant, we will now consider the effects of changing these variables one by one.

First, we adjust the baseline settings by using dense sampling instead of keypoint sampling. Descriptors are now extracted for each 5th pixel, with 2×2-pixel SIFT blocks. The following results are obtained:

AP				MAP
airplanes	cars	faces	motorbikes	
0.90191	0.71314	0.97208	0.79111	0.84456

Table 3: Dense sampling

We observe a marginal increase in the AP for airplanes (0.04), and a significant improvement for faces: almost 0.2. For motorbikes performance drops marginally (0.01), and for cars it decreases by 0.07. It is striking that the results differ so much per class. In particular, at first sight it is surprising that the AP decreases for cars and motorbikes. Dense sampling provides so much more information that it seems natural to think that the classifiers should learn more from it. This turns out to be false. Apparently, very representative keypoints are selected for cars and motorbikes, and supplying more information only 'distracts' the classifiers. For airplanes and faces it is the other way around. It seems that the keypoint selection method used for the baseline does not manage to find all points of interest, so that dense sampling actually provides information that improves the performance of the models for these classes. In particular the classifier for faces realizes an extreme improvement with dense sampling. Hypothetically, this could have something to do with the background of the images. The training and test images with faces are the only ones that tend to be taken indoors, while the other pictures are usually situated outdoors. As dense sampling provides information about the entire picture, differentiation with respect to the background may have somehow influenced the results.

Next, we look at the impact of changing the vocabulary size k. The quantitative results are reported in Table 4.

	AP				MAP
k	airplanes	cars	faces	motorbikes	
400	0.85686	0.77914	0.67875	0.80088	0.77891
800	0.86099	0.78502	0.68196	0.80432	0.78307
1600	0.82914	0.79891	0.70562	0.80056	0.78356
2000	0.84722	0.77478	0.74564	0.82263	0.79757
4000	0.79841	0.80266	0.75502	0.80155	0.78941

Table 4: Changing vocabulary size with other parameters kept constant to default setting

The first row of this table ($k = 400$) corresponds with the baseline case. Doubling the vocabulary size yields a slightly improved performance for all classes. Doubling it again, to $k = 1600$, increases the Average Precision for all classes except airplanes and motorbikes. Increasing k to 2000 improves the AP for all classes except cars, and doubling it to 4000 increase the AP for cars and faces while decreasing the score for airplanes and motorbikes. Looking at the MAP, we see a (marginal) increase up until $k = 2000$. For $k = 4000$, the MAP drops slightly.

For most classes, the difference in scores caused by changing the vocabulary size is marginal. A few things stand out, though. We note that one result of increasing the vocabulary size is that the variance between the AP scores for the different classes decreases (6×10^{-3} for $k = 400$ against 5×10^{-4} for $k = 4000$). This is largely due to the significant increase for the faces-classifier, and the decrease for the airplanes-classifier. For cars and motorbikes, very little difference is observed. In fact, for airplanes the performance only deteriorates for $k = 4000$, which indicates that a large vocabulary size is not good for all classifiers. As the MAP also decreases when $k = 4000$, it seems that this number of visual words is just too high for the average classifier. The optimum in our case lies around $k = 2000$. Intuitively, it makes sense that increasing the vocabulary size improves the performance up to a certain point, because the ability to distinguish between more codewords allows the classifiers to learn more fine-grained distinctions. However, if the number of visual words exceeds some critical value, it seems that the distinctions may become too fine-grained. In the case of the airplanes this could mean that detectors that used to be assigned to the same codeword are now scattered among different words, which creates a divergence between the histograms of pictures belonging to the same class. Clearly, for faces this critical limit has not yet been reached at $k = 4000$, which suggests that classification of faces profits from a more diverse vocabulary, perhaps because faces form a less homogeneous collection than airplanes or motorbikes.

Now we change the color space c. Results are displayed in the below table, where 'opp' denotes opponent color space:

	AP				MAP
c	airplanes	cars	faces	motorbikes	
RGB	0.85686	0.77914	0.67875	0.80088	0.77891
gray	0.79732	0.75077	0.6461	0.81757	0.75294
rgb	0.84601	0.67758	0.59858	0.7845	0.72667
opp	0.93426	0.79333	0.80993	0.84993	0.84686

Table 5: Changing color space with other parameters kept constant to default setting

Again, the first row of the table just contains the baseline results. Changing from RGB-SIFT to grayscale SIFT, we see a decreased performance for all classifiers, except for motorbikes, where the model performs marginally better. In order to perform grayscale SIFT, a color image is first converted to grayscale, which means that 2/3 of the information stored in the different color channels is 'lost'. Hence, one would expect a decreased performance for all classifiers when only grayscale color is used. Perhaps for the class of motorbikes, color is somehow less telling than it is for the other classes, so that collapsing the three color channels into one actually improves the AP score.

For normalized RGB (rgb, or 'nrgb' in the code), performance deteriorates for all classifiers. This may be explained by the fact that, in a different way than upon conversion to grayscale, normalized RGB also takes away much of the information that is stored in the original color channels. The normalization transformation is not injective, meaning that many different $\langle R, G, B\rangle$ tuples are mapped to the same normalized $\langle r, g, b\rangle$ tuple. For let $\langle R', G', B'\rangle$ be a fixed combination of RGB-values from the range $[0, 255]$, which is normalized to $\langle r', g', b'\rangle$. Then for *any* constant a, the

color $\langle aR', aG', aB' \rangle$ is also mapped to $\langle r', g', b' \rangle$, because $\frac{aC}{aR+aG+aB} = \frac{C}{R+G+B}$, for $c \in \{R, G, B\}$. This implies that much of the color variation in RGB space is lost upon normalization, which could explain why the classifiers yield worse results.

It is interesting to see that for all classifiers, converting the color space to opponent colors at least marginally improves the performance. For RGB-SIFT the MAP is 0.78; for opponent-SIFT it is 0.85. Unlike the other conversions, the transformation to opponent color space does not take away information. Instead, it transforms the RGB channels by emphasizing the opposition and contrast between different colors. Probably, this improves the performance of all classifiers because contrasts within a picture are emphasized, which causes descriptors to be more distinctive, and thereby more informative.

Now we change the number of images that is used to construct the vocabulary. These images cannot be used for training later on, so increasing the size of this set implies that fewer training instances are available. However, for the default setting of 100 training instances per class, this does not matter. We let m denote the number of images that is used from each class to generate the visual dictionary (so the total number of images used is $4m$). The results are as follows:

	AP				MAP
m	airplanes	cars	faces	motorbikes	
50	0.85686	0.77914	0.67875	0.80088	0.77891
100	0.86581	0.80217	0.64888	0.80612	0.78075
250	0.87739	0.77215	0.67104	0.80502	0.7814

Table 6: Changing the number of images used for vocabulary construction per class with other parameters kept constant to default setting

The results are as we would expect: with all other parameter settings kept constant, increasing the number of images used to create the visual words improves the performance, even if just slightly. The only exception is the classifier for cars, which obtains a higher AP for $m = 100$ than for $m = 250$, but this anomaly need not be significant. It makes sense that a larger basis for the vocabulary should lead to better scores, because the more pictures are used to create the visual words, the more likely the algorithm is to cluster the words according to the features that are really essential to particular categories. However, we do note that the scores tend to only improve marginally. Apparently, using only 50 images per class for vocabulary construction provides an acceptable basis, and increasing this number does not yield spectacular improvement. It is not possible to investigate how performance could be improved even more by increasing m much further, because then not enough pictures would remain available for training. The training set with faces, for instance, only contains 400 images, and perhaps some of them have to be skipped because they are in grayscale.

Finally, we assess the impact of changing the number of training instances n per class (so the total number of training samples is $4n$ for each classifier, with $3n$ negative samples and n positive ones). See Table 7.

As should be expected, the performance of the classifiers continues to improve as more training samples are used. For all classifiers the AP scores increase steadily upon expanding the training set, suggesting that the over-all performance of the system could be boosted even further by training on more than $n = 300$ samples per class. However, not enough images are available to test this hypothesis if we wish to use an equal number of training samples per class. Contrary to the other parameter settings we explored, the number of training instances appears to have an identical effect on all classifiers. This makes sense, as the training aspect relies on the SVM architecture which is the same for all classifiers. All parameters used before were in fact no model parameters, but variables governing the process preceding the classification.

	AP				MAP
n	airplanes	cars	faces	motorbikes	
100	0.85686	0.77914	0.67875	0.80088	0.77891
200	0.89856	0.82818	0.71724	0.83687	0.82021
300	0.91427	0.84376	0.76023	0.84905	0.84183

Table 7: Changing the number of training instances per class with other parameters kept constant to default setting

3. PART 2: CONVOLUTIONAL NEURAL NETWORK IMAGE CLASSIFICATION

In this section we experiment with an end-to-end image classification model using Convolutional Neural Networks (CNNs). Using CNNs we can incorporate learning the image features into the training process. This differs from the Bag-of-Words approach that uses hand-crafted features. Determining the image features is a crucial aspect of image classification so incorporating it into the learning phase will improve the accuracy of the image classification.

The CNN outputs a feature representation of the image. The output features of the network can be used for classification or as an input for another classification algorithm, such as an SVM.

As was the case in the previous section we are building an image classification model that will perform four class image classification on the Caltech4 image set.

3.1 Implementation

An implementation of an CNN based image classification model consists of three major parts.

I Defining the network architecture.

II Data preprocessing.

III Training the network.

Training the network is a computationally costly operation. We also known that some layers of a CNN learn universal features. This is particularly true of the shallow layers that learn features such as corners, edges and blobs. Therefore, it makes practical sense to use a technique called transfer learning to leverage a pretrained CNN which was trained on another data set. In this project we base our model on a CNN that was pretrained on the 10 class CIFAR-10 data set.

3.2 Defining the Network Architecture.

The network architecture is a key design decision of a CNN which can dramatically impact the performance of the network. The fact that we are using transfer learning means that the structure of our network is dependant on the structure of the pretrained network.

The pretrained network consists of 14 layers. This includes an output softmax layer and an input layer which accepts inputs of size 32x32x3. The 12 intervening layers are divided into 4 segments. Each segment consists of a convolutional layer, a max-pooling layer and a layer of rectified linear units. The convolutional layer filters the images. The pooling layer aggregates a set of adjacent activation units to downsample the data and provide invariance to local translations. The rectified linear units perform an element-wise activation function which adds non-linearity into the model.

The data size decreases and the data depth increases as the layers get deeper. All of the parameters that need to be learnt occur in the convolutional layers. The deeper convolutional layers have more parameters.

3.3 Data Preprocessing.

The data preprocessing step prepares the images for input into the network.

As mentioned in the previous section the network accepts input images of size 32x32x3. So all the images in our data set are re-sized appropriately.

The fact that a CNN requires a fixed size input can be limitation. In some scenarios resizing an image into a predefined shape could dramatically alter the appearance of the object within the image. This effect is minimized by the fact that the network is trained on images that have undergone the same transformation. Also reducing the size of an image to 32x32x3 results in a loss of information, but this is offset by a reduction in the size the parameter space which means it is easier to find an optimal solution.

In the image preprocessing step we also subtract the mean from each image. The helps remove variance in the images that could make the training process less stable and reduce the effectiveness of weight sharing.

The data preprocessing step could include data augmentation. Data augmentation is a technique that alters the training data in an attempt to make it more representative of the possible input space. In computer vision this could be operations like constrained random transformations of the images such as random flipping.

3.4 Training the Network.

The training process takes labelled examples of the images and uses them to update the weights of the pretrained network to better fit the new data set. Training is done by passing the images forward through the network, measuring the prediction error and using the error to update the weights. This process is dependant on a number of hyper-parameters. In this section we will experiment with the learning rates, weight decay, batch size and number of epochs, to train a fine tuned network.

The batch size specifies the number of images that are fed forward through the network for each update of the weights. The weight updating is done in batches because it requires less memory and is more computationally efficient. The number of epochs determines the number of times that all the data gets passed through the network.

The training process has four learning rates. Both the previous and new layers of the network have a bias and weight learning rate. These determine the how much an update step can alter the current weights. Weight decay, which is a related parameter, determines the rate at which the learning rates converge towards zero.

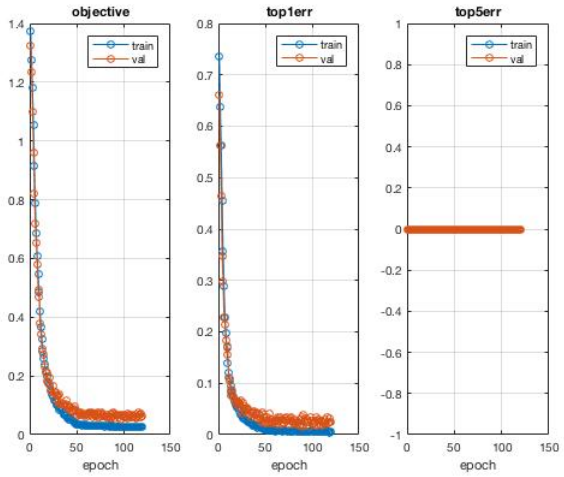

Figure 2: Results from training with default parameters and 120 epochs.

Due to the number of hyper-parameters and the time it takes to train the network it is important to have a search strategy. We set all the hyper-parameters to a default value, then altered only one at a time to determine an effective value. This is not an ideal search scheme because it does not account for interaction between altering hyper-parameters but it's was only option that the computational resources available to us allowed.

The default parameters used were:

Weight Decay = 0.001.

Batch Size = 100.

Number of epochs = 50

Bias learning rate of new layers = 1

Bias learning rate of previous layers = 0.2

Weight learning rate of new layers = 4

Weight learning rate of previous layers = 2

The results of of training the network with 120 epochs can be seen in figure 2. We can see from figure 2 that the there are diminishing returns as the number of epochs increases. Epochs beyond the 60th seem to result in very small improvements. At 40 epochs the top 1 error is 0.09, at 80 epochs the top 1 error is 0.038 and at 120 epochs the top 1 error is 0.025. So the gain is relatively small but not irrelevant. Whether this additional accuracy is worth the additional computation depends on the required accuracy and the available computing resources. In this scenario, since all the training times are less than an hour it seems like there is no reason not to use 120 epochs and gain as much accuracy

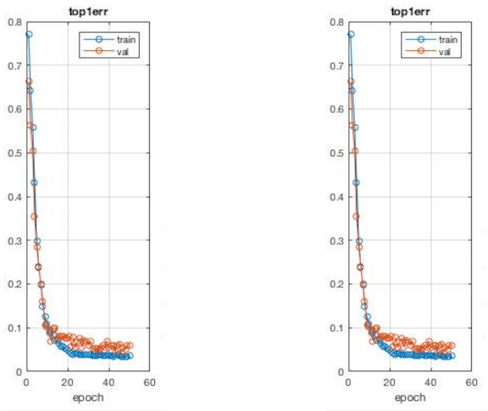

Figure 3: Result from training network with different batch sizes. Batch size 50 on the left and 100 on the right.

as possible. Especially, since there does not seem to be any indication of over-fitting in the validation set.

The results of our experimentation with different batch sizes of 50 and 100 can be seen in figure 3.4. From figure we can observe that the batch size of 50 converges on a more optimal solution, converges more quickly and results in less variation in the error of the validation set. The smaller batches are performing better because they result in more weight updates per epoch. However, if the data is noisy smaller batches can result in erroneous updates. But these experiments show that this is not the case in this situation.

The next parameter we are going optimise is the learning rate. The learning rate determines the speed of convergence to an optimal weight. If the learning rate is too large we will step over the optimal solution and if the learning rate is too small we will converge too slowly. We found that relatively small learning rates, less than 0.5 performed best. This makes sense if you consider that the pretrained network is already quite accurate. This indicates that the training starts from a near optimal solution. Therefore, the accuracy of the convergence given by a smaller learning rate is more beneficial than increased learning speeds. However, if the learning rate is too small then the new CNN will be too similar the pretrained network. Figure 4 shows how training with a learning rate of the new layers fixed to 0.005 doesn't make much of an impact on the resulting features.

We also know that the initial layers of the pretrained network are going to be most relevant to the new network we are trying to train. This is because these layers detect features that are more universal to all image classification tasks. So we required that the learning rates for the previous layers were less than the learning rates for the new layers. We experimented with setting the learning rates of the previous layers to 0 and 0.1. In both experiments all the other parameters were equal but the achieved error for the non-zero previous layer learning rates was 30% more accurate. This indicates that although the previous layers do generalise to our model they aren't optimal. So it's important to do some retraining of these previous layers.

We limited our search of possible learning rates to between 0 and 1. Our search of this parameter space was loosely based upon binary search. We began with 0.5 and then tried the value of the two parameters in the the middle of the

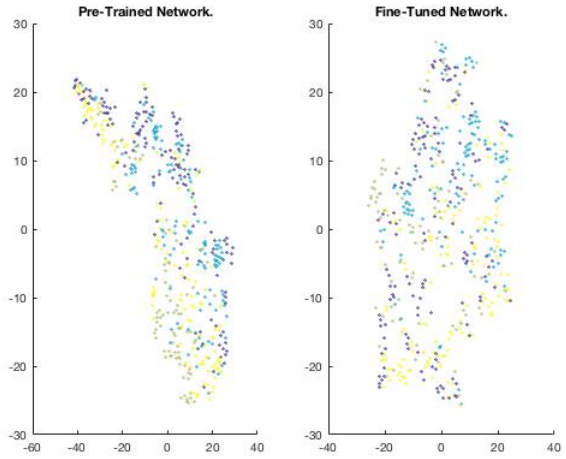

Figure 4: CNN feature of 500 images coloured according to image class.

range below and above the current parameter. We continued this until the center parameter resulted in better accuracy.

Using this technique we determined that the following learning rates are effective.

Bias learning rate of new layers = 0.09

Bias learning rate of previous layers = 0.04

Weight learning rate of new layers = 0.3

Weight learning rate of previous layers = 0.2

We can visualise how these learning rates alter the pretrained network by comparing the feature space of the pretrained network with the network trained using these learning rates. This is shown in figure 5. As figure 5 shows the features of the pretrained network are less descriptive of the features of the different image classes. This is observable by the fact that the features of the different image classes are not clearly separated in the feature space of the pretrained network.

3.5 Results

In this section we aim to compare four different models that were developed in this project which are; the pretrained CNN, the CNN trained with default parameters, the fine-tuned CNN and the Bag-of-Words model.

The pretrained CNN, which was trained on a different image set, produced feature representations of the images which an SVM could use to obtain an accuracy of 89.28%. This accuracy is high, especially if you consider that the Bag-of-Words model obtained less than 80% accuracy and that the CNN wasn't even trained on the same image set. Both models use SVMs to classify the image features. The difference is that the CNN's features were learnt rather than being hand-crafted. The fact that the pretrained CNN is more accurate than the Bag-of-Words model indicates two important observations. Firstly, learning the feature representation of images can improve the accuracy of a model.

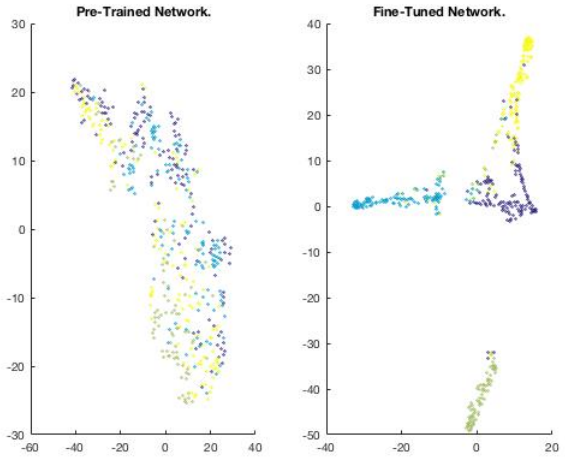

Figure 5: CNN feature of 500 images coloured according to image class.

Secondly, CNNs used for image classification generalise well, which indicates that there are similarities in the features that identify different kinds of objects.

Without the SVM the accuracy of the pretrained CNN is much lower. If we look at figure 6 we can see that the features outputted by the CNN are not clearly clustered. This suggests that the additional classification of the SVM is an important aspect of the 89.28% accuracy.

The next step is to compare the pretrained network accuracy with the new network trained using the default parameters. The accuracy of the new CNN using the default parameters was 97.25%. So retraining the network resulted in a large increase in accuracy.

Finally, we wanted to compare the results of the network using default parameters with the fine tuned parameters. Given that the default parameters result in 97.25% accuracy, it will be difficult to improve this much, given that every percent of accuracy closer to 100 is harder to obtain. The fine-tuned network got 98.75% accuracy.

One of the clear differences between the performance of the pretrained network and the fine-tuned network is the role of the SVM. The features produced by the fine-tuned network clearly separate into classes. This difference can be seen in figure 7. As a result of this we can directly use the network to classify images because the additional SVM doesn't add much accuracy. If we compare 6 and 7 we can see how the features generated by the optimised CNN result in a better separation of the features of each of the classes.

In our experiments we found that the increase of accuracy resulting from the SVM could vary a lot. In one of our experiments the accuracy of the CNN alone was 45%.3 and adding the SVM increased the accuracy to 86.28%. However, as the accuracy of the CNN increases the gain resulting from the SVM diminished. This makes sense if you consider that the increasing accuracy of the CNN means that the features are more descriptive of the image classes. We can visualise this in 7 which shows how the more accurate CNN creates features that are quite different for the different image classes.

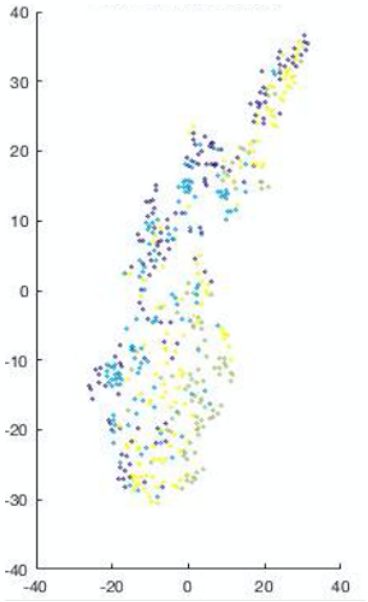

Figure 6: Features generated by the pretrained CNN.

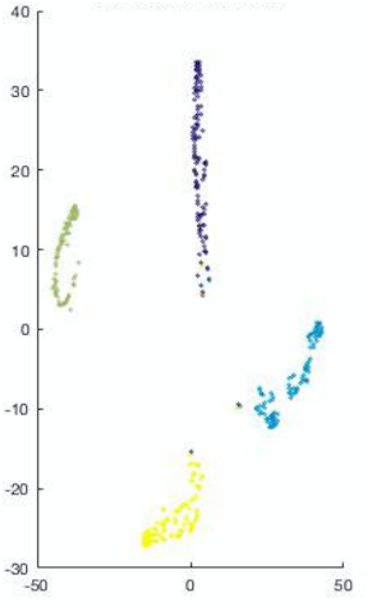

Figure 7: Features generated by the fine-tuned CNN.

The results indicate a few important conclusions. Firstly, incorporating image feature representation into the learning phase improves the accuracy of a model. Also, CNN's can generalise well to other image sets but there is a gain in accuracy to be gotten from retraining the network. Lastly, using an SVM on the output features of a network increases the accuracy but this gain diminishes as the network learns to separate the features of the different image classes better.