Computer Vision 1: Final Project

Gabriele Bani
11640758
gabriele.bani@student.uva.nl

Andrii Skliar
11636785
andrii.skliar@student.uva.nl

1 Introduction

In this project we have analyzed two main approaches for image classification: Bag-of-Words approach and Convolutional Neural Networks approach.

We have analyzed Bag-of-Words classification approach with different settings, including various features for building visual vocabulary, different approaches to building visual vocabulary, namely K-Means clustering and Gaussian Mixture Models and different approaches for building image representation, such as histograms and Fisher vectors. We have tried various hyperparameters and have observed how they influence the performance of the algorithm. We have included detailed analysis of our results and possible ways of improving the algorithm.

We also analyze CNN architectures and understand their behavior under different hyperparameter settings. We explore how a pretrained network can be used on another dataset in order to achieve good results, without occurring in overfitting or underfitting behaviors, thanks to fine tuning procedures. We evaluate both qualitatively and quantitatively the features from the pre trained model and the fine tuned model, and show how fine tuning constantly produces superior results. Finally, we apply different additional variations to the training procedures and analyze their effects.

2 Bag-of-Words based Image Classication

As mentioned in the task, Bag-of-Words based Image Classication system contains the following steps:

1. Dataset formation
2. Feature extraction and description
3. Building a visual vocabulary
4. Quantize features using visual dictionary (encoding)
5. Representing images by frequencies of visual words
6. Classication

We would like to discuss each of the steps in detail and its implementation in the next section.

2.1 Implementation

2.1.1 Dataset formation

Dataset is formed separately for training and testing phases. Firstly, vocabulary is constructed. To do so, we have used first 250 images per class. Afterwards, out of all the images that are left, depending on the specified size, positive and negative training example are sampled for each class. It is done in a following way:

1. Out of images, which were not used for the vocabulary, first M (given number of positive examples) images are taken. Afterwards, we randomly sample K (given number of negative

examples) images 3 more times (generally, $N-1$, where N is the number of classes) to form negative examples of current class for all the other classes. This gives us $M + 3 * K$ (with M being number of positive examples, K - negative) number of images sampled (note, that they should not be unique) per class.

After repeating this procedure for each class, we get following matrix (here for a better understanding of the data structure being used):

$$\begin{bmatrix} pos\_class\_1 & neg\_for\_class\_2 & neg\_for\_class\_3 & neg\_for\_class\_4 \\ pos\_class\_2 & neg\_for\_class\_1 & neg\_for\_class\_3 & neg\_for\_class\_4 \\ pos\_class\_3 & neg\_for\_class\_1 & neg\_for\_class\_2 & neg\_for\_class\_4 \\ pos\_class\_4 & neg\_for\_class\_1 & neg\_for\_class\_2 & neg\_for\_class\_3 \end{bmatrix}$$

2. We are transforming the matrix in order to have all positive and negative examples for one class in the same row.

This procedure of dataset formation allows us for more robust work of the system as we are not dependent on the number of classes and can use our classifier to train on any new dataset. Note, however, that for this dataset, if we use 250 images for the vocabulary, we can only use maximum 150 images for the training purposes as class with the lowest amount of images consists of 400 images. Also, we didn't try to use different number of images per class in the same experiment as that might lead to class imbalance thus giving us worse results. However, it might be a good idea for a more extensive experimentation.

2.1.2 Feature extraction and description

There are two different options for feature extraction and description: regular SIFT, which extracts descriptors at each keypoint that it has found, and dense SIFT, which extracts descriptors at each k-th pixel. Both approaches are easy to implement using vlfeat functions. However, the main confusion here comes from the fact that SIFT has to be applied to images in different colorspaces. To do this, we have done following:

1. (only for regular SIFT) Convert image to grayscale and find keypoints
2. Calculate descriptors for each of the keypoints. Here, we have used vlfeat function vl_siftdescriptor and have followed an example for calculating gradient provided in the vlfeat documentation page [1].
3. Concatenate descriptors calculated at the same position along the first dimension over all color channels. Thus, in case of 3-channel image for each of the keypoints we will have a vector of size $128 * 3 = 384$.
4. Vectors obtained in the previous step are being concatenated into one matrix and are used as a temporary image representation.

Note, that this operation is done for each image in the formed dataset. Also, one more issue that we have encountered is that some of the images in a dataset are grayscale. For that case, if we have encountered those, we copy single gray channel 3 times in order to form 3-channel image.

2.1.3 Building a visual vocabulary

To form a visual vocabulary, we apply clustering algorithm (in our case, K-means) to the images, which have been selected as a vocabulary set. After running K-means, we get matrix of size $Kx128 * n_channels$, where K is a number of clusters (size of the vocabulary) and $128 * n_channels$ is a dimensionality of each feature. These will serve as visual words.

2.1.4 Quantize features using visual dictionary (encoding)

Afterwards, we need to calculate number of occurrences of these visual words in each image. To do so, for each feature extracted in step 2 (section 2.1.2), we find the closest visual word (thus, closest cluster) and represent it as an id of the closest cluster (thus, row number in the matrix of visual words). To do so, we have used very convenient matlab function dsearchn [2], which finds the index of the

[1] `http://www.vlfeat.org/matlab/vl_siftdescriptor.html`

[2] `https://nl.mathworks.com/help/matlab/ref/dsearchn.html`

nearest point in N-dimensional space. This way, we represent image as an unordered list of visual words.

2.1.5 Representing images by frequencies of visual words

To get image representation in a form of frequencies of visual words, we calculate number of occurrences of each visual word in the image and normalize it by the total number of visual words in the image. This can be thought of as a histogram with as many bins as size of the vocabulary (number of clusters). So, now each image will look something like following: $(a_1, a_2, \dots a_K) = (0, 0.01, 0.19, \dots 0)$.

2.1.6 Classication

Before actually doing classification, we need to label data for each class. In our case, we label images of the correct class with 1 and other classes as -1. So, now for each training and testing example, we get following vector: $(a_1, a_2, \dots a_K, label)$. After that, we apply the classification algorithm (One-vs-all SVM in our case) to the data. Generally, because our data is imbalanced (normally, K positive examples and $3 * K$ negative examples), without carefully tuning the parameters, SVM performs pretty bad, simply predicting -1 thus getting an accuracy of 75% for all the classes. To avoid that problem, we have automatically done parameter tuning using built-in tools of LIBSVM [3] ([2]). We are using 3 types of kernels: linear, polynomial and RBF. For all the kernels, we optimize cost parameter (in range $(2^{-5}, 2^{-4} \dots 2^{-3})$), which is responsible for how strong we will penalize our SVM for actually misclassifying data. For polynomials and RBF, we also optimize gamma in the same range as cost. For polynomials we are also optimizing degree, which is the degree of a polynomial function that will be used. All the training is performed in a 10-fold cross-validation mode, which allows for more correct training without actually overfitting to the training data. So, we find the parameters, which give us the highest cross-validation accuracy, retrain it on the whole dataset and use them for testing our classifier. Note, that this is being done for each class and thus classifiers may use different hyperparameters for each class.

2.2 Results and Discussion

Here we would like to discuss the results that we have obtained. Due to the fact that an extensive grid search would take 5490 experiments (*3 for each kernel), we have chosen our baseline model, which we have used for testing different assumptions. As a baseline model for regular SIFT (not dense SIFT), which will be discussed first, we have chosen the one with 150 positive and 150 negative examples per class (so, 600 images in training set for each class), 400 clusters (size of vocabulary), gray-SIFT and linear kernel. For dense SIFT, we use 50 positive and 50 negative examples per class (so, 200 images in training set for each class), SIFT step size of 5 pixels, SIFT block size of 3 pixels and 400 clusters (size of vocabulary). For both options, number of examples used for training the vocabulary is 250.

2.3 Vocabulary size

First, we would like to discuss results obtained for different vocabulary size, presented in table 1 and table 2. As we can see, mean average precision changes quite significantly depending on the vocabulary size. Even more importantly, average precision per class changes very significantly per class. It may be caused by several reasons, which we would like to discuss in more detail.

As mentioned in [3], in general there is no fixed optimal vocabulary size for the images and it highly depends on the data itself. Small visual vocabulary size results into multiple features being represented by one word in a vocabulary thus reducing the discriminative power of the classifier. At the same time, very large visual vocabulary size can add noise and therefore harm the performance of the classification.

However, it is also important to understand that each classifier has been specifically optimized for the provided visual vocabulary, therefore it is difficult to reason about whether performance differs for different classes is due to those classes being more linearly separable in the latent space (and thus can be classified better using linear SVM or due to the size of the visual vocabulary).

[3] `https://github.com/cjlin1/libsvm`

Also, it's important to notice that different vocabulary size might be more suitable for images of one class and less suitable for other classes due to the amount of actual discriminative features common for images of each class.

An interesting thing to see, however, is that larger size of the vocabulary (4000) performs better for dense SIFT and worse for the regular SIFT. This is most probably due to that dense SIFT finds many more descriptors than regular SIFT, so size of the vocabulary should be much bigger in order to account for larger amount of discriminative features.

2.4 SIFT Colorspace

Next, we would like to discuss effect of colorspaces on the final result. We have tried the following colorspaces: Grayscale, RGB, normalized RGB (written as rgb), opponent-RGB, HSV. We have visualized results in table 3 and table 4. Similarly to the different vocabulary size, mean average precision differs significantly for various colorspaces. Again, we can see that depending on the target class, different colorspaces can be more suitable. There is no evident reason for better performance of one colorspace for specific class as the performance highly depends on the dataset itself. However, possible reasons will be discussed further. In our discussion we will mostly be using [6] as it contains a lot of details about effects of different colorspaces on image recognition.

Grayscale colorspace was used as a baseline and it performs pretty well due to the SIFT properties: shift and scale invariance, light intensity change invariance, light intensity shift invariance, light intensity change and shift invariance, light color change invariance, light color change and shift invariance.

Compared to grayscale-SIFT, both RGB-SIFT and rgb-SIFT lose illumination color invariance, with former also losing illumination intensity invariance, which might result in both better and worse performance depending on the dataset. This might consist of images taken under the same or very different lighting conditions, which will respectively make it better or worse. Notice however, that the main change compared to grayscale-SIFT is that SIFT descriptors are concatenated over all three channels thus resulting in more descriptive and possibly discriminative features.

Though [6] reported opponent-SIFT to be working better than any other SIFT features, possibly due to the fact that it is not fully invariant to changes in light intensity and light color. However, for us it performs bad because of the single grayscale image in the set of Motorbike images. More specifically, because we convert it to 3-channel image by replicating its single channel, when converting to opponent-SIFT, first two layers will be zeroed out as suggested by formulas: $O1 = \frac{(R-G)}{\sqrt{2}}; O2 = \frac{(R+G-2B)}{\sqrt{6}}$, where R, G and B represent corresponding color channels and this possibly results in classifier not being able to handle this image correctly.

Despite the main disadvantage of HSV-SIFT, which is that hue gets unstable under condition of low saturation and therefore gets ignored, HSV-SIFT performed best for us. Note, however, that it is not invariant to light color changes, as only V channel contains information about the intensity of the image and therefore is the only channel that is invariant to light color changes. Apparently, for our dataset the fact that it is only partially invariant was beneficial. However, as already mentioned, this is different from the results provided in [6], where OpponentSIFT performed best.

2.5 Number of training examples

You can see results for various number of training examples in table 5. However, as you can see, Mean Average Precision is pretty much the same (with difference of 1-2%) for any number of examples. Therefore, the results suggest that an extensive search for good balance between positive and negative examples is not worth the improvement we can possibly get. Anyway, the tendency is clear: the more examples we use and the more balanced (where balanced means the same number of examples per class) the training set is - the better. Reasons for such a similar behaviour can be due to the fact that we are optimizing penalty parameter for our SVM, which allows it to avoid issues connected to class imbalance.

2.6 SVM Kernel

Different results for various SVM Kernels, provided in table 6, point us to an interesting fact: both for dense SIFT and regular SIFT, sets of images in latent space are almost linearly separable. This can be derived from the fact that SVM with linear kernel performs as good as SVM with polynomial kernel of degree 10. Also, we can see that SVM with RBF kernel performs much worse, which can lead to a possible conclusion that the geometric shape of the distribution of data is not circular and therefore is not suitable for RBF kernel.

Also, an interesting observation can be done by looking at values of optimal parameters for different kernels, which can lead to the conclusion that in general hard-margin SVM performs better for the given dataset than soft-margin SVM, as the optimal penalty is chosen as 2^15 - the largest value we tried.

Another possibly good idea would be to better optimize gamma for RBF, as it plays the most important role in the form of the kernel, however, the results, acquired by using linear kernel suggest that the improvement in that case might not be worth the computational effort we would need to apply to find more optimal parameters for RBF kernel.

One more possible improvement would be to use more training data, but due to the computational limitations, we were not able to get results in manageable time and decided to use smaller set of training images as we mostly needed it to analyze the performance for different kernels rather than finding the best classifier possible.

2.7 Regular SIFT vs Dense SIFT

As we have shown, dense SIFT gives generally better results even with least amount of training examples. However, it is mostly due to its algorithm, which finds keypoints every n pixels, which results in many more descriptors and is therefore providing more accurate representation of the image. In real-life applications, using dense SIFT might not be feasible as its complexity is much larger than regular SIFT (since the amount of actual keypoints found by regular SIFT is smaller than number of keypoints found by dense SIFT).

2.8 Improvements (Bonus part)

To improve our results, we have decided to use KAZE features ([1]) instead of SIFT features. The main reason for choosing KAZE features was that they preserve all invariances of SIFT features while also achieving faster speed and better results in terms of repeatability, distinctiveness and robustness, which leads to better performance, as reported in the original paper.

Afterwards, we model distribution of the features using Gaussian Mixture Models with specified number of clusters and construct Fisher Vectors from the acquired mixture weights, as suggested in [4]. We have decided to use Fisher Vectors, because they can be interpreted as a generalized version of the BoW approach, that we have been using initially. However, it also allows to preserve the probabilistic nature of the distribution of image features, which might result in better performance. Also, despite the fact that for Fisher Vectors size of visual vocabulary tends to be much smaller, it manages to preserve more information than BoW approach with larger vocabulary size. Note that for faster training, we are initializing GMM with the results, which we get from K-means after running it for 25 iterations, as suggested in VLFeat documentation [4].

These are then fed to a gradient boosting algorithm, namely, RUSBoost ([5]), which classifies the examples. RUSBoost is a boosting algorithm, which uses decision trees for learning and is specifically intended for datasets which have class imbalance. Therefore, it should acquire better results for our training dataset.

We are using built-in Matlab function, namely fitcensemble [5], which allows us to use bult-in Bayesian Hyperparameter optimization with 10-Fold Cross Validation. We are applying optimization to all parameters of RusBooster. However, the main issue with that approach is that it was giving very high accuracy of 96% on test with only 50 examples per class, which outperforms any of the previous classifiers, however, decision values for Boosting were uninterpreterable, therefore, for finding mAP

[4] `http://www.vlfeat.org/overview/gmm.html`

[5] `https://nl.mathworks.com/help/stats/fitcensemble.html`

values, we decided to use SVM classifier as it was still performing much better with suggested features than with previously used ones. You can see final results in table 7. So, the main improvement we get is that classification is much better with less amount of data (due to a better representation) and much faster and more efficient computation (due to faster computation of KAZE features).

Table 1: Results for different vocabulary size with regular SIFT

Sift type	Colorspace	Kernel type	Positive examples	Negative examples	Vocabulary size	Average Precision per Class	Mean Average Precision
Regular	Grayscale	Linear	150	150	400	Airplanes: 0.988342 Cars: 0.871229 Faces: 0.976659 Motorbikes: 0.955168	0.947849
					800	Airplanes: 0.976435 Cars: 0.928243 Faces: 0.983230 Motorbikes: 0.970695	0.964651
					1600	Airplanes: 0. 981340 Cars: 0.966185 Faces: 0.996398 Motorbikes: 0.981280	0.981340
					2000	Airplanes: 0.986271 Cars: 0.986437 Faces: 0.996188 Motorbikes: 0.993889	0.990696
					4000	Airplanes: 0.976052 Cars: 0.995367 Faces: 0.996841 Motorbikes: 0.986896	0.988789

Table 2: Results for different vocabulary size with dense SIFT

Sift type	Colorspace	Kernel type	Positive examples	Negative examples	Vocabulary size	Average Precision per Class	Mean Average Precision
Dense	Grayscale	Linear	150	150	400	Airplanes: 0.963474 Cars: 0.954532 Faces: 0.980028 Motorbikes: 0.983737	0.970443
					800	Airplanes: 0.961537 Cars: 0.978486 Faces: 0.992375 Motorbikes: 0.979649	0.978012
					1600	Airplanes: 0.956262 Cars: 0.951820 Faces: 0.987126 Motorbikes: 0.988056	0.970816
					2000	Airplanes: 0.957658 Cars: 0.990614 Faces: 0.996701 Motorbikes: 0.976580	0.980388
					4000	Airplanes: 0.960409 Cars: 0.992592 Faces: 0.997714 Motorbikes: 0.977598	0.982078

Table 3: Results for different colorspaces with regular SIFT

Sift type	Vocabulary size	Kernel type	Positive examples	Negative examples	Colorspace	Average Precision per Class	Mean Average Precision
Regular	400	Linear	150	150	Grayscale	Airplanes: 0.988342 Cars: 0.871229 Faces: 0.976659 Motorbikes: 0.955168	0.947849
					RGB	Airplanes: 0.968636 Cars: 0.927557 Faces: 0.976546 Motorbikes: 0.954459	0.956799
					normalized-RGB	Airplanes: 0. 963144 Cars: 0.919831 Faces: 0.9477956 Motorbikes: 0.948763	0.944924
					opponent-RGB	Airplanes: 0.964454 Cars: 0.938004 Faces: 0.964781 Motorbikes: 0.862633	0.932468
					HSV	Airplanes: 0.968003 Cars: 0.985938 Faces: 0.984556 Motorbikes: 0.953993	0.973123

Table 4: Results for different colorspaces with dense SIFT

Sift type	Vocabulary size	Kernel type	Positive examples	Negative examples	Colorspace	Average Precision per Class	Mean Average Precision
Dense	400	Linear	150	150	Grayscale	Airplanes: 0.963473 Cars: 0.954532 Faces: 0.980028 Motorbikes: 0.983737	0.970443
					RGB	Airplanes: 0.966848 Cars: 0.981981 Faces: 0.980454 Motorbikes: 0.959074	0.972089
					normalized-RGB	Airplanes: 0. 986919 Cars: 0.907860 Faces: 0.968315 Motorbikes: 0.987028	0.962531
					opponent-RGB	Airplanes: 0.890687 Cars: 0.973661 Faces: 0.991191 Motorbikes: 0.879908	0.933862
					HSV	Airplanes: 0.965373 Cars: 0.969832 Faces: 0.993354 Motorbikes: 0.946563	0.968780

Table 5: Results for different number of training examples with regular SIFT

Sift type	Colorspace	Kernel type	Vocabulary size	Positive examples	Negative examples	Mean Average Precision
Regular	Grayscale	Linear	400	50	50	0.935257
					100	0.927835
					150	0.933353
				100	50	0.942443
					100	0.940208
					150	0.948553
				150	50	0.949839
					100	0.940246
					150	0.947849

Table 6: Results for different SVM kernels

SIFT type	Colorspace	Vocabulary size	Positive examples	Negative examples	Kernel type	Optimal parameters	Mean Average Precision
Dense	Grayscale	400	50	50	Linear	Best C: 32768 Best Gamma: 32678	0.970443
					Polynomial	Best C: 32768 Best Gamma: 32678 Best Degree: 10	0.970385
					RBF	Best C: 32768 Best Gamma: Airplanes: 1024, Cars: 8192, Faces: 1024, Motorbikes: 2048	0.944942
Regular	Grayscale	400	50	50	Linear	Best C: 32768 Best Gamma: 32678	0.935257
					Polynomial	Best C: 32768 Best Gamma: 32678 Best Degree: 10	0.943228
					RBF	Best C: 32768 Best Gamma: 512	0.867488

Table 7: Results for Bonus Part

Airplanes	Cars	Faces	Motorbikes	Total
1	0.994422	0.998084	0.999208	0.997928

3 Convolutional Neural Networks for Image Classification

3.1 Understanding the Network Architecture

The network can be logically divided in 5 blocks, and clearly shows a pattern. Every block contains a convolutional layer, and apart the last block, they also all have a pooling layer. It would make no sense in fact to apply a further pooling at this point since the size of the data is already 1 in the first two dimensions. Also, apart from the first and last block, we have a ReLU non-linearity after the convolution in the same block. In the last layer, it is reasonable to not include the non linearity since we want to output probabilities, and thus apply a softmax. So, the most frequent pattern seen in this network is (conv, relu*, pool), repeated many times in the network, where we put the star on relu because there is one exception in which it is not present.

Pooling layers and non-linearity layers have no parameters. Since parameters are shared, convolutional layers have a number of parameters equal to $d \times d \times F_i \times F_o$, where d is the size of the filter, F_i is the number of input filters (from the previous layer), and F_o is the number of output filters of the conv layer. So, by looking at our architecture, the layer with most parameters is layer 10 (the second-to last conv layer), with $4 \times 4 \times 64 \times 64 = 65536$ weights. To compute the memory consumption, this value has to be multiplied by 4, since every weight is represented by a 32 bit floating point number, and we obtain the same value reported in fig. 1, 256 KB (in the row "param mem").

The memory needed for a convolutional layer is given by $I \times I \times F_o$, where I is the dimension of the data and f_o is the number of output filters of the layer. We can calculate the sizes of the filters and see that, as seen in fig. 1, the first conv layer occupies the most memory, 128 KB per image.

Figure 1: Pretrained network information

layer	0	1	2	3	4	5	6	7	8	9	10	11	12	13
type	input	conv	mpool	relu	conv	relu	apool	conv	relu	apool	conv	relu	conv	softmxl
name	n/a	layer1	layer2	layer3	layer4	layer5	layer6	layer7	layer8	layer9	layer10	layer11	layer12	layer13
support	n/a	5	3	1	5	1	3	5	1	3	4	1	1	1
filt dim	n/a	3	n/a	n/a	32	n/a	n/a	32	n/a	n/a	64	n/a	64	n/a
filt dilat	n/a	1	n/a	n/a	1	n/a	n/a	1	n/a	n/a	1	n/a	1	n/a
num filts	n/a	32	n/a	n/a	32	n/a	n/a	64	n/a	n/a	64	n/a	4	n/a
stride	n/a	1	2	1	1	1	2	1	1	2	1	1	1	1
pad	n/a	2	0x1x0x1	0	2	0	0x1x0x1	2	0	0x1x0x1	0	0	0	0
rf size	n/a	5	7	7	15	15	19	35	35	43	67	67	67	67
rf offset	n/a	1	2	2	2	2	4	4	4	8	20	20	20	20
rf stride	n/a	1	2	2	2	2	4	4	4	8	8	8	8	8
data size	32	32	16	16	16	16	8	8	8	4	1	1	1	1
data depth	3	32	32	32	32	32	32	64	64	64	64	64	4	1
data num	1	1	1	1	1	1	1	1	1	1	1	1	1	1
data mem	12KB	128KB	32KB	32KB	32KB	32KB	8KB	16KB	16KB	4KB	256B	256B	16B	4B
param mem	n/a	10KB	0B	0B	100KB	0B	0B	200KB	0B	0B	256KB	0B	1KB	0B

3.2 Preparing the Input Data

We implemented the function getCaltechIMDB as described by the requirements. Regarding the $4D$ matrix imdb.images.data, we scale every input image to 32 by 32 (with 3 color channels), since otherwise the images would have different width and height. The rest of the data structure is created as required.

3.3 Updating the Network Architecture

Since Block-5 of the network outputs 64 filters, NEW_INPUT_SIZE has to be 64. We set NEW_OUTPUT_SIZE to 4 since we want to predict one probability for each of the four classes that we are considering.

3.4 Setting up Hyperparameters

We run the experiments using the default hyperparameters specified in the source files and the suggested batch sizes (50, 100) and epochs number (40, 80, 120). We report in table 8 the results at the end of the training, and in fig. 2 the respective training curves. These results are all very

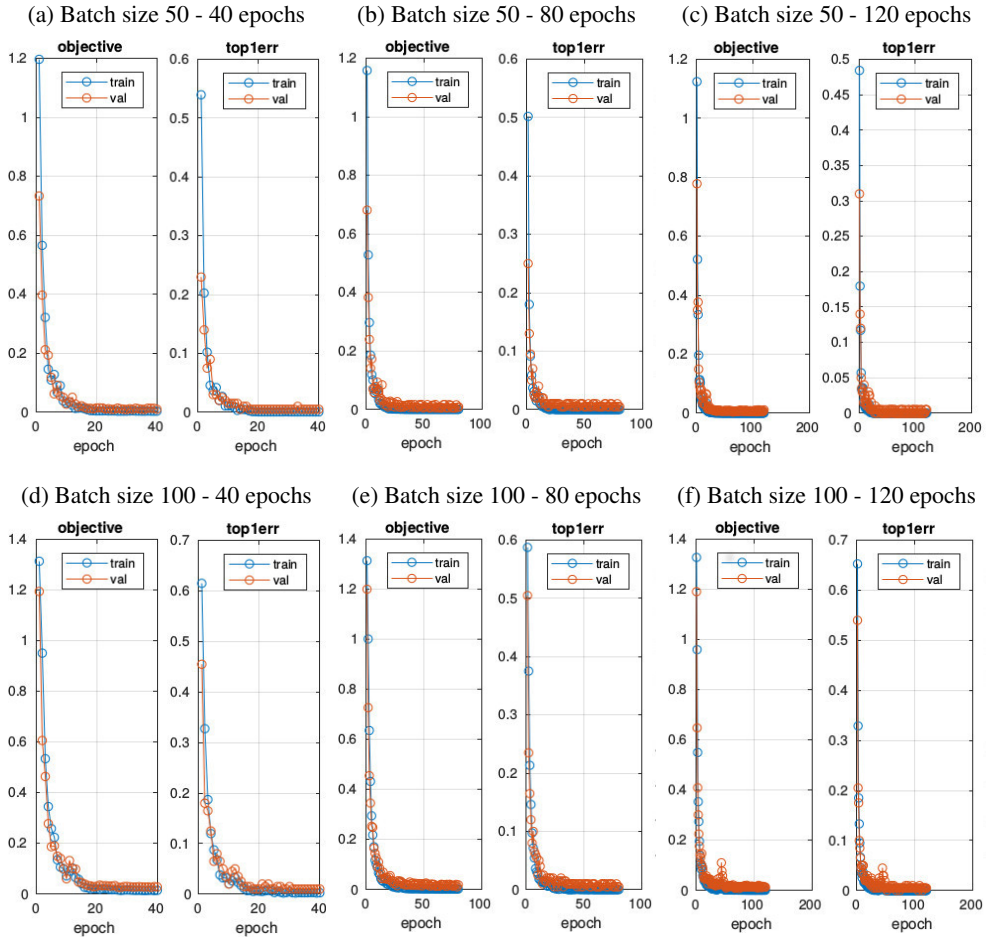
	Train loss			Train err			Val loss			Val err		
	40 ep	**80 ep**	**120 ep**	**40 ep**	**80 ep**	**120 ep**	**40 ep**	**80 ep**	**120 ep**	**40 ep**	**80 ep**	**120 ep**
Batch size 50	0.35	0.056	0.037	0.054	0	0	1.03	0.93	1.10	0.50	0.50	1.00
Batch size 100	1.49	0.26	0.15	0.27	0	0	2.83	1.94	1.36	1.00	0.50	0.50

Table 8: Results under different settings - all the values are divided by 100 for better visualization

impressive, with the lowest validation errors being of just one percent. Batch size seems to have an impact on the training, making the losses decrease more slowly. Even though this might affects the evaluation metric on other situations, in our case the training and validation errors are so low that we cannot make reliable conclusions about them. The number of epochs affects mainly the value of the loss function, since the training accuracies are so high that no meaningful deductions can be made from them.

Finally, we kept the model with highest accuracy on the validation set. Since multiple models have the same validation accuracy, we kept the first one found, which in our case was with batch size 50 for 40 epochs. Even though it is clearly not the best in term of loss, we kept it as best model in order to keep consistency in the code. Results in the bonus part will be based on this model.

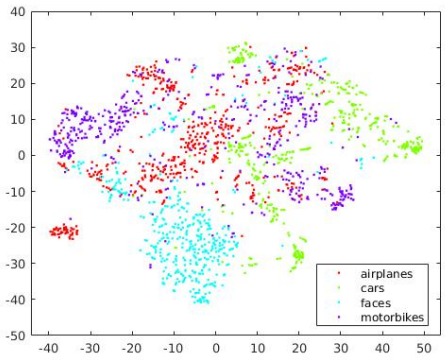
Figure 2: Training curves with different batch sizes and epochs



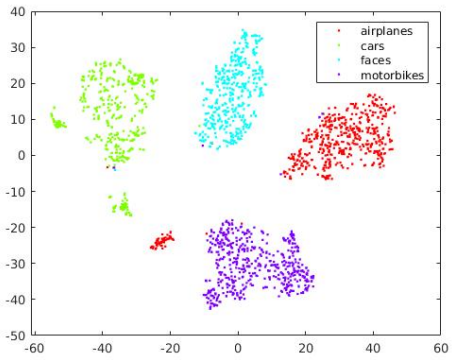
3.5 Feature space visualization

Figure 3: Tsne low dimensional visualization

(a) Pretrained features visualization



(b) Finetuned features visualization



We extract features from the two networks before the last layer, skipping the relu activation as done in the script train_svm.m. In general, features are considered as the neuron activations, which means, that they include the nonlinearity. To keep consistency with the given code, we however ignore the relu. Anyways, in this simple setting the results are qualitatively very similar in either case.

From fig. 3 we can see a clear difference in the visualization of the feature spaces in the two cases. In the pretrained model, the data points are spread in the feature space without clearly forming separable clusters. Without label coloring, it would be difficult even for a human to separate the data clearly in clusters. We can see however that most of the faces are close to each other, while all the other classes are much more spread out and overlapping. On the other side, we clearly see a completely opposite situation in the feature space visualization of the finetuned features. The data is clearly divided in 4 major clusters, even though minor clusters are still present. A human in this case could clearly identify the four major clusters with no problem. The problem of overlapping is almost completely removed (only a few outliers are found in other classes clusters). This means that given a datapoint, we can be almost sure that the neighboring datapoints are of the same class, which is a very good indicator that the network understands the main characteristics of the data.

3.6 Evaluating Accuracy

We measure the accuracies using the three configurations, and obtain 99.50 $\%$ for the finetuned model, 89.50 $\%$ for the SVM using the pretrained model features and 99.50 $\%$ using SVM with the finetuned model features. We can see from these results that the pretrained model was not able to model well enough the new data, while the finetuned model could. In fact, it could not only achieve high accuracy by itself, but also allowed the SVM to achieve almost the same accuracy when using its features. This could mean that the pretrained network did not generalize well enough, or that the datasets used is significantly different from the one used to pretrain it. We suggest the second option, since pretrained accuracy is still pretty high, almost 90%.

Comparing with the results obtained in the first part of the homework, where we used hand crafted features with the bag of words approach, we obtain much better results, with much less effort. In the previous part, many parameters had to be set empirically, and some of those had much more relevance than others (vocabulary size, color space, SIFT type). These regard the kind of features that will be used, and are typically dataset dependent. So, whenever we want to analyze a new dataset, we have to search again for good configuration of parameters, knowing that even one wrong choice could lead to poor results (see RBF kernel in our case). Although training a neural network from scratch might be also hard, nowdays there are plenty of pretrained models available, especially in computer vision. When fine tuning, the advantages of using neural networks become overwhelming. The number of parameters needed for fine tuning is much lower, since the architecture of the network is already decided. Also, transfer learning can be used even for different tasks easily, while other

	CNN classification	SVM on finetuned features
Original	99.50	99.50
Only noise	98.50	97.50
Only rotation	99.00	98.50
Noise + rotation	95.00	97.00
Freezing layers	97.00	95.00

Table 9: Accuracies under different situations

approaches typically do not allow this. Although the main advantage of hand crafted features and the bag of words approach is that they can work well with small amount of data, we have shown here that by fine tuning a pretrained model, we can achieve even better results even with small datasets. To conclude, neural networks provide a consistent way to extract knowledge from data and allow to achieve the best results in many different situations, and using hand crafted features typically does not pay off in both easy of use and results obtained.

3.7 Improvements (Bonus part)

3.7.1 Data augmentation

For this part, we execute two different experiments. First, we introduce gaussian and salt and pepper noise. We do not create a new dataset, but randomly (and independently) apply them with probability 0.5 for every batch. The goal of this kind of data augmentation is to make the network more robust to noise, since during trained it tries to learn to classify even when the images are noisy. We set the sigma parameter for gaussian to 0.01, and it can be interpreted as the width of the filter. We set salt and pepper parameter to 0.05, which means that on average, 5% of the pixels will be modified.

We also apply rotation as data augmentation. Since the imrotate function can change dramatically the image with black pixels, we restrict the rotation angle between -20 and 20 degrees. As before, the rotation happens with probability 0.5, while the rotation angle is chosen uniformly between -20 and 20.

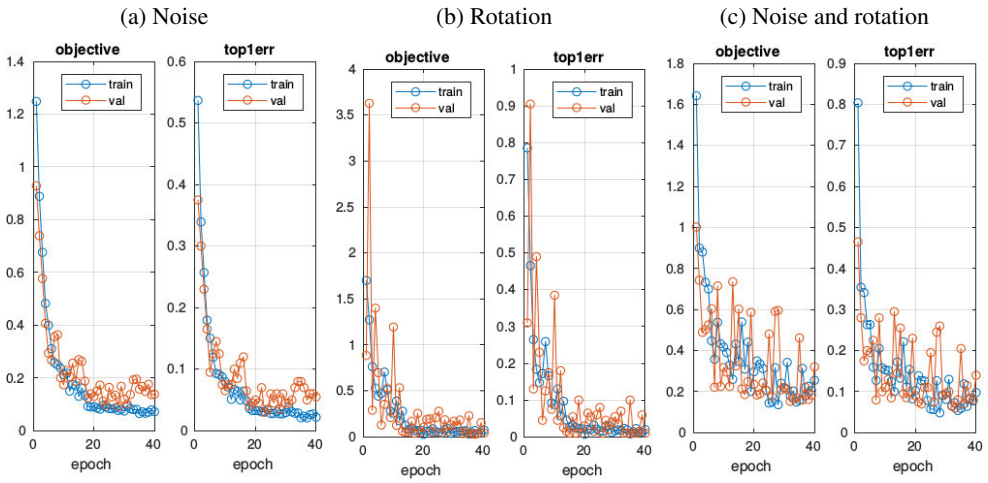
finally, we apply noise and rotation. Since this made the training very unstable, we lowered the probability of rotation to 0.2. All the results can be seen in table 9, while the training curves can be seen in fig. 5

The main interesting point to analyze is the behavior of the training curves, which is much different from what seen in the previous experiments. We see in fact that all our data augmentation make the learning process more difficult, with the training and even more validation curves showing wiggling behaviors. We realize that the most extreme wiggles happen when rotation is present when the highest angles are chosen. Overall, data augmentation lead to slightly worse test accuracies, in particolar when combined together. However, the models produced are now capable of better dealing with noise and rotation, which could be a desirable features in many situations.

3.7.2 Freezing the layers

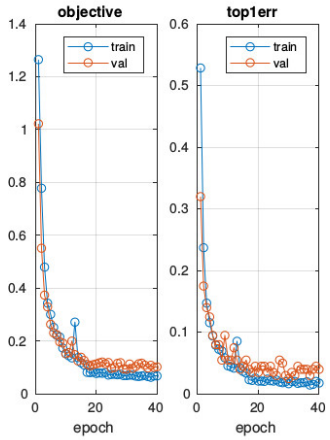
Freezing all the layers of the network in our case leads to poor validation results (0.35 validation error) since the network cannot produce features that explain well the new dataset, so we choose to use a less invasive freezing for this task. We freeze the first two conv layers by setting their learning rate to zero, and set the next two conv layers with one tenth of the default learning rate. For the last layer we stick to to the default learning rate. The reason for this is that we assume the early layers to capture general low level features, while allow the middle layers to slowly adjust to the characteristics of the new dataset. The last conv layers still keeps the default learning rate because it is where we want most of the learning to happen, at high level.

Figure 4: Training curves with different data augmentation



We run one experiment, with batch size 50 for 40 epochs, and the results can be seen in table 9. This performance drop respect to the fully finetuned model suggests us that the features caputred in the first layers are not general enough and cannot represent at best general features of images.

Figure 5: Training curves when freezing layers



4 Conclusion

In this project we have explored behaviour of BoW approach and CNNs aproach for image classification under different settings.

For BoW approach we have shown that the behaviour is highly dependent on the dataset itself as there are a lot of parameters, which need to be optimized for a specific set of training data. That has been supported by various results for different hyperparameters. That also means, that this algorithm generalizes worse than CNNs. However, we have also shown the ways to optimize hyperparameters in order to achieve better performance.

We have explored how CNNs behave when used for transfer learning, and argued about their advantages over traditional feature extractors, providing empirical results. We have implemented and experimented different variations of the training proecdure, and shown how fine tuning can be harder when using data augmentation, and how freezing layers might lead to lower performance if the features extracted by the pretrained network are not general enough.

In general, after analyzing how two main approaches for image classification work, we can conclude that there are various improvements that can be done to make algorithms perform better and more efficient, and there is still a lot to innovate in the field of Computer Vision.

References

[1] Pablo Fernández Alcantarilla, Adrien Bartoli, and Andrew J. Davison. Kaze features. In Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid, editors, *Computer Vision – ECCV 2012*, pages 214–227, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[3] W. Liu, H. Cui, J. Hou, and J. Kang. On visual vocabulary size in svm classification. In *2016 IEEE International Conference on Industrial Technology (ICIT)*, pages 962–967, March 2016.

[4] Jorge Sanchez, Florent Perronnin, Thomas Mensink, and Jakob Verbeek. Image Classification with the Fisher Vector: Theory and Practice. *International Journal of Computer Vision*, 105(3):222–245, December 2013.

[5] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano. Rusboost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 40(1):185–197, Jan 2010.

[6] K. van de Sande, T. Gevers, and C. Snoek. Evaluating color descriptors for object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1582–1596, Sept 2010.