Assignment4 - Computer Vision

Tushar Nimbhorkar 11394110
Diede Rusticus 10909486

February 2017

1 Bag-of-Words based Image Classification

This part of the assignment focuses on the classification of images based on the bag-of-words (BoW) technique. Images belong to either of the four classes: airplanes, motorbikes, faces and cars. In the training phase, an equal amount of images are used per class to learn a Support Vector Machine (SVM) to correctly predict the classes of the test set images.

The following steps are taking for the BoW classiciation system:

- Feature Extraction and Description
- Building Visual Vocabulary
- Quantize Features Using Visual Vocabulary
- Representing images by frequencies of visual words
- Classification

The following sections describe our implementation strategy in detail. After the BoW classification setup, we have tried to estimate the best set of hyperparameters. Each step in the pipeline is explained, where the name of the function in the matlab code is given for each part.

`feature_extraction`

First, the descriptors are extracted from a set of images of each class, through either keypoint (`vl_sift`) or dense sampling (`vl_dsift` & `vl_sift`). Keypoint sampling only takes descriptors of points in the picture that are edges or corner for example (salient points) and the dense sampler takes (with a step size) from every pixel the descriptor. Naturally, dense sampling contains more information of the image but has higher computational cost. The descriptor consists of colour information and for this purpose a SIFT descriptor must be chosen. RGBSIFT, rgbSIFT, opponentSIFT and graySIFT (`vl_phow`) are implemented for this purpose. In a later stadium, we will do experiments with the different samplers / descriptors and see which combination yields to highest performance in the classification phase. To create the visual dictionary we have used 100 images for each class. Using more images was computationally not feasible for us.

`kmeans`

Now that we have all the descriptors, we have to cluster them in order to make a visual vocabulary. We are doing this by the k-means algorithm. The vocabulary size must be chosen, this will be the amount of clusters it will create. Also, we give it a maximum number of iterations N, to at least have a stopping criterion when the algorithm does not converge in a reasonable time. We have used the build-in function of matlab: `kmeans` to get the clusters. This returns us the cluster centroids locations. Every cluster represents a visual word.

`get_input_features(train)`

For a new set of images, we can now extract their feature descriptors and assign them to the closest visual word from the vocabulary. For every feature descriptor, we used `pdist2` to get the pairwise distances to all visual words, and saved the index of the minimum distance.

We now represent each image by cluster frequencies, by counting each index occurrence in the function `quantize`. It could be the case that one image has far more descriptors than another. Therefore, we are normalizing the cluster frequencies; every frequency divided by the maximum frequency occurring. The frequencies are used as input for the classification model.

`train_svm`

An SVM is trained as the classifier per each object class. So as a result we will have four binary classifiers. In order to train our models, we still need the correct labels. Therefore in the `get_labels` function we create a vector containing all the correct labels (0/1) for the images in the fixed order for the certain model. We use the in-build function of matlab `fitcsvm` to train a binary support vector classifier. The kernel function can be easily specified with this function.

`get_input_features(test)`

Our four models are trained. In order to get the general performance of the models, we are using the models to classify the images in our test set. This means that first, for each of the test images, the feature descriptors are extracted and assigned to the closest visual word. After which all images are represented by their cluster frequencies.

`test_data`

All images in the test set are classified according to the four models. This gives us a ranked list of images according to their classification scores. The higher a classification score, the more likely the image will actually have the label of that class. Because we only need this ranking list, we do not need the image labels. The ranking list will be a matrix where the first column contains the indices of every image. These indices are used to later map them back to the original image. The second column contains the true label of the images. This label is later used to calculate the mean Average Precision of the ranking. The third column contains the scores of every picture, for that class. The list is ordered based on this column; the highest image has the highest probability of being of that class and vice versa.

`evaluate`

In addition to the qualitative analysis we want to measure the performance of our system by calculating the mean average precision. The average precision per class is defined as follows:

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

where n is the number of images, m the number of images of the class, and x_i is the i^{th} image in the ranked list. f returns the number of images in the first i images that are of the class which is tested for. Taking the mean of the four APs, gives us the mean average precision of the system.

`experiment` The whole pipeline is included in this function. All parameters which we need to tune are arguments for this function, such as:

`experiment_nr`
: Keeps track of the specific experiment with the accompanying parameter setting.

`sampler`
: Either keypoints or dense sampler.

`descr_type`
: Implemented the four colourspaces: RGB-SIFT, rgb-SIFT, Opponent-SIFT, and Gray-SIFT.

`vocab_size`
: The amount of cluster centers is pre-fixed. We are testing for [400, 800, 1600, 2000, 4000].

`nr_feature_images`
: The images which are used to extract the features and descriptors for the visual vocabulary can not be the same as the images used for training the models. We always set this number to 100 per class.

`nr_train_images`
: Expected is that when you increase the number of images used for the training phase, the more accurate your model. This is something we will investigate through experiments. We are testing for [50, 150, 250] images per class.

`nr_test_images`
: All images in the test set are used. This boils down to 50 images per class.

`kernel`
: The linear kernel function is used as the default. However, also the Radial-basis function is used, just like the Polynomial kernel function. Their performances are compared

`N`
: This is the maximum iterations our k-means algorithm can do.

Results

The results of all experiments are shown in Table 1. We see that the dense sampler always outperforms the keypoint sampler. This is because more information is gained when having more descriptors. The dense sampler is extracting information of (almost) every pixel, and therefore, also background information is taken into account. When changing the vocabulary size, the system performance did not increase much. We did not see a significant improvement when increasing the vocabulary size to a number higer than 400, so for the remaining experiments we kept it at this size. For the descriptors, RGB-SIFT and rgb-SIFT both yield a high MAP. There is no significant difference between the two. We tried different numbers of training images, to see how much our system would be improved when adding more data. We saw a slight improvement when using more images, so for the data available the best setting would be to use 250 images per class at least, in your training phase. Of the three kernel functions, the linear kernel showed the best scores. Therefore, we kept this kernel as the default kernel function for our experiments.

Table 1: The results of the hyperparameters tuning

Keypoint vs. Dense Sampler						
#	Voc Size	Sampler	SIFT	Nr. Train Images	Kernel	MAP
1	400	Keypoints	RGB	100	Linear	0.869
2	400	Dense	RGB	100	Linear	0.976
3	400	Keypoints	rgb	100	Linear	0.859
4	400	Dense	rgb	100	Linear	0.916
5	400	Keypoints	Gray	100	Linear	0.771
6	400	Dense	Gray	100	Linear	0.921
7	400	Keypoints	Opponent	100	Linear	0.938
8	400	Dense	Opponent	100	Linear	0.946
Best tested parameter setting						
#	Voc Size	Sampler	SIFT	Nr. Train Images	Kernel	MAP
9	400	Dense	RGB	250	Linear	0.977

2 Convolutional Neural Networks for Image Classification

Since modern Convolutional Neural Networks (CNNs) take 2-3 weeks to train across multiple GPUs on ImageNet, it is best to use a pre-trained network than train it from scratch. As training from scratch not only requires lots of data, it has also a high computational cost. The process of fine-tuning a pre-trained CNN is as follows:

Understanding the network

1. Patterns in the architecture:
   There is a pooling layer in between successive convolutional layers. The pooling layer is used to reduce the data dimensionality. Relu layer applies an elementwise activation function.
2. Layer 10 seems to have the most parameters because the parameter memory size is the biggest. Layer 10 also seems to be the biggest because it has the biggest overall size as well. This is nicely been visualized by vl simplenn display

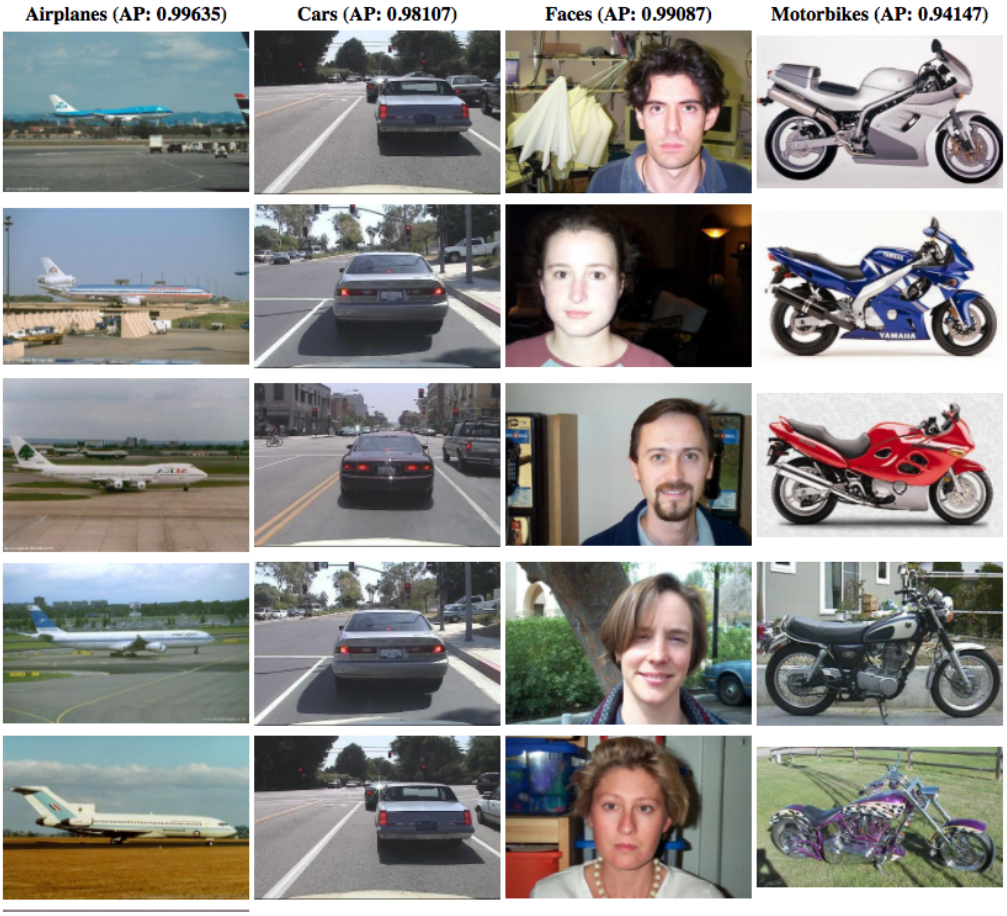
Preparing the input data

The task is to implement the getCaltechIMDB function under finetunecnn.m script. Let imdb denote the input struct to be given to MatConvNet. This struct is implemented such that Imdb will have the following data members: imdb.images (data, labels, sets) and imdb.meta. The getCaltechIMDB function reads images, their labels, and their splits (either training or validation) from the data directory and creates the imdb struct. Because the pre-trained CNN only accepts input data of size 32x32x3, we cannot use the grayscaled images. Also, we need to resize the images because they all have different sizes. We create the other two data members according to the proper split set and class label.

Updating the network architecture

We are not training the network all over again. Instead we want to keep as much information as necessary from the training class. We do this by transfer learning / fine-tuning. We take a CNN which is pre-trained on some dataset that knows ten classes. Our dataset only knows four classes so we replace the fully-connected output layer of this CNN by a new output layer with only four nodes. We re-train the network according to the new dataset by backpropagating the errors and updating the weights. This results in weight updates all over the network. We create a new layer which has 64 input nodes, because the layer before the

Figure 1: Five highest ranked results of the best tested parameter setting: vocabulary size: 400, sampler: dense, SIFT: RGB, number of train examples: 250, kernel function: linear.

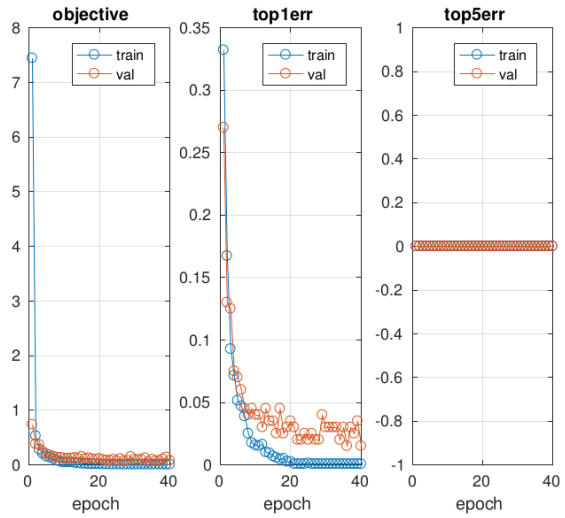


original output layer consisted of 64. These 64 are the number of features, which we do not want to loose. The new output size will be changed 4 according to the number of classes.

Setting up the hyperparameters

We are tuning the hyperparameters by a couple of experiments. The results are shown in Table 2. Increasing the size of the training set really improved the accuracies. Smaller dataset used for fine-tuning are prone for overfitting. This is later showed in the results. Using the full dataset for fine-tuning, we will have a more general model. This is to be expected as the model is trained on a more general data distribution. Increasing the number of epochs did not change much, so 40 epoch suffices. Changing the batch size from 100 to 50 also did not change the performance significantly. We played a bit with the learning rates, and this give us some improvements on the performances. Its practical to use smaller learning rates for fine-tuning a CNN, because the weight updates must be small. Otherwise we will loose to much information. This is because we expect that the CNN weights are relatively good, so want to avoid distorting them. The learning rate for the updated layer is bigger because this is a randomly initialized layer that needs much improvement still.

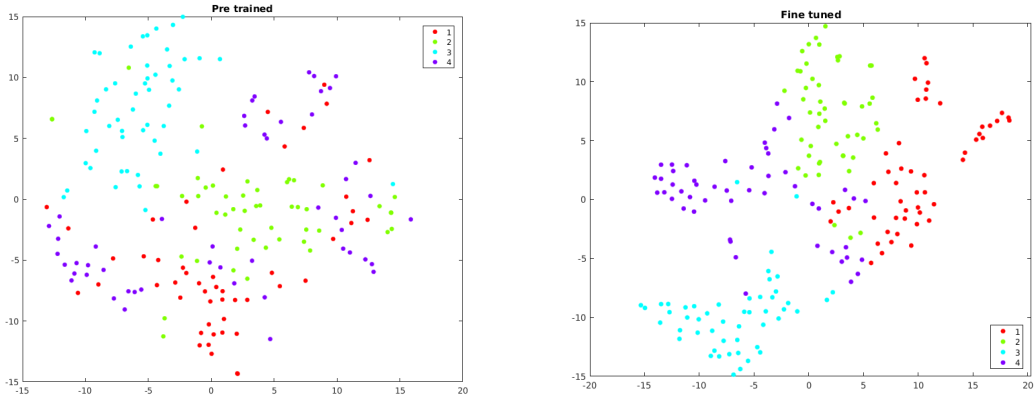
Figure 2: This figure shows the objective and error functions for every epoch during training.



The final setting will be:

1. lr-prev-layer = 0.01, 0.02
2. lr-upd-layer = 0.05, 1
3. weight-decay = 0.0001
4. batch-size = 100
5. epochs = 40
6. nr. train images = 400
7. nr. test images = 50

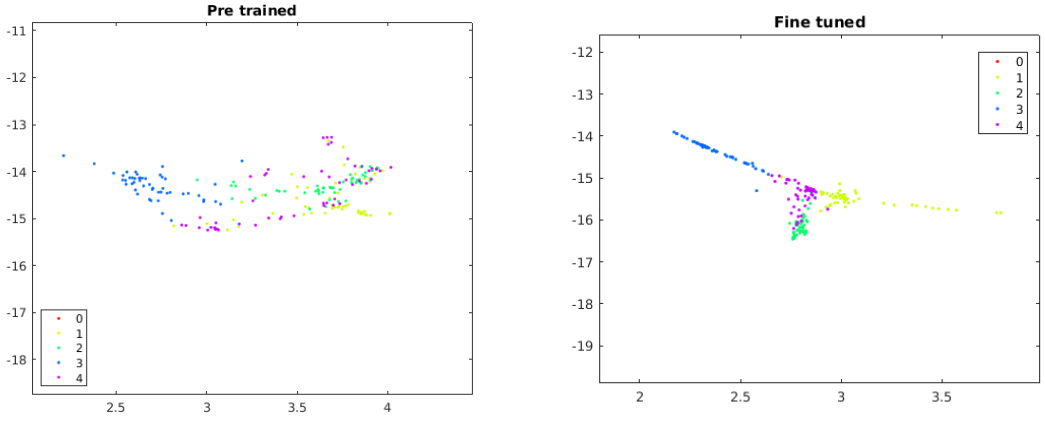
Experiments We want to minimize the objective function, as well as the error function. For more epochs, it is shown in Figure 2 that the error and objective function both go down. The validation error is a bit higher than the training error because training a model always biases the training set. It is not overfitting, because the validation set is not going up again. Figure 3 shows the features of the pre-trained and fine-tuned network on the provided dataset with the best set of parameters. Table 2 shows some parameter settings and the accompanying accuracies. Figure 4 shows the features when we have smaller learning rates for the updated layer, namely: 0.01 and 0.02 instead of 0.05 and 1. It shows that classification is performing way less. If we want to compare the results of the BoW model and the CNN model, we need to calculate the accuracies for the BoW model. We do this by assigning the class with the highest score for the SVM models for each image, like a softmax. This way, we not only have a ranked list, but we also have a predicted class for each image. The benefit of using this technique is that every image gets exactly one class assigned. If we only assign it to a class when the model actually tells you it is, then we would have many images not being assigned to any class. This is not a possible outcome, so we chose to use the softmax technique. We calculated the accuracies of the BoW classified pictures and they are shown in Table 3. The BoW performance is very similar to the fine-tuned CNN performance under certain parameter settings. However, the pre-trained SVM and the fine-tuned SVM both perform some lower on the classifiction compared to the BoW technique.



(a) Pre-trained net

(b) Fine-tuned net

Figure 3: This figure shows features of the pre-trained and fine-tuned network on the provided dataset. Every class in its own colour. The fine-tuned model clearly classifies the points.



(a) Pre-trained net

(b) Fine-tuned net

Figure 4: This figure also shows the features of the pre-trained and fine-tuned network on the provided dataset, however, different parameters are used. Namely: Learning rates for the previous and updated layer: [0.01,0.02]

Table 2: Hyper parameter tuning

#	1	2	3	4	5	6	7	8	9
lr. prev.	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
lr. up.	0.05	0.05	0.01	0.01	0.05	0.05	0.05	0.05	0.05
	1	0.05	0.02	0.02	0.02	1	1	1	1
weight decay	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.001
batch siz	100	100	100	100	100	100	50	50	50
epochs	40	40	40	40	40	40	80	120	40
nr. train im	100	100	100	200	400	400	400	400	400
nr. test im	20	20	20	30	50	50	50	50	50
f-t (CNN)	0.91	0.84	0.88	0.93	0.94	0.97	0.97	0.95	0.96
p-t (SVM)	5.01	4.94	4.94	11.03	94.5	94.5	94.5	94.5	94.5
f-t (SVM)	5.44	5.15	5.44	11.43	96	95.5	95.5	96.5	95.5

Table 3: Accuracies class models BoW

Airplanes	0.99
Motorbikes	0.98
Faces	0.96
Cars	0.96
Mean	0.97