Report: Object Recognition

Computer Vision - Assignment 5

Ian SCHOLL

ischoll@student.ethz.ch

1. Environment Setup

I used Miniconda on Linux Ubuntu 20.04.3 LTS. My CPU is Core™ i7-1185G7 @ 3.00GHz × 8. Training took me roughly 3 hours for 80 epochs for the CNN-based classifier.

2. Bag-of-words Classifier (60%)

2.1. Local Feature Extraction (20%)

2.1.1 Feature detection - feature points on a grid (5%)

Using the *np.linspace* it is fairly easy to create such a grid of points. The only slightly tricky part was considering the border. As later we will be using local patches of size 16x16 (see 2.1.2) around each grid point (x,y), that patch will cover the area [x-8, x+7]x[y-8, y+7]. So in order to fully use the image I chose to leave a border of 8 pixels on top and on the left, and 7 pixels on the bottom and on the right, as can be seen by my linspaces. Flattening and stacking them then gives the desired shape of [nPointsX*nPointsY, 2].

2.1.2 Feature description - histogram of oriented gradients (15%)

This was definitely the most tricky part of this assignment. There were several choices that one can make depending on what sources (Sources[1], [2], CV lecture) you follow for creating a HoG. For example:

1) use unsigned gradient orientations in the range 0-180° or signed ones in 0-360°
2) use magnitude in weighting the histogram or no
3) split each counted instance proportionally into bins or to just count them fully towards one bin

As described in 2.1.1, I chose to cover patches of the area [x-8, x+7]x[y-8, y+7] around each grid point (x,y), sub-dividing that area into sixteen 4x4 (*cellWidth x cellHeight*) cells. To compute the gradient orientations and magnitudes I used the *cv2.cartToPolar* method, which directly gives the angles in the range [0°, 360°]. All that I had to do for the most basic approach then was assigning each of those to a bin by dividing by the number of bins and taking the floor value.

A simple loop through all the pixels of each cell then allows to compute each single histogram and concatenate to get the full descriptor for each particular grid point.

In the following table I have been experimenting several combinations of the choices 1) - 3) mentioned above, while using values *k=50* & *numiter=300* for the k-means clustering (see my code for how the different combinations were implemented):

Degree range	**Using magnitude**	**Using proportion**	**Average over 5 runs**	
			pos. sample acc.	**neg. sample acc.**
[0°, 360°]	No	No	0.9224	*0.9560*
		Yes	0.8857	0.9160
	Yes	No	0.8776	0.7480
		Yes	0.8776	0.7960
[0°, 180°]	**No**	**No**	***0.9633***	**0.9440**
		Yes	*0.9633*	0.9360
	Yes	No	0.7429	0.6000
		Yes	0.7878	0.7520

As can be seen from this table, for our particular case, introducing magnitude or proportions in the calculation of the histograms makes the result rather worse, especially the magnitude. We receive pretty good results with using just the most basic approach, both using unsigned gradient orientations in the range 0-180° or signed ones in 0-360°. The very best results in this small experiment were achieved with the unsigned gradients (0°-180°) as suggested by Sources [1] & [2], however it has to be mentioned that 5 runs is not a very big sample size. If we wanted to investigate all of this in detail, more runs would be needed and maybe for each of the combinations a new tuning of the values for *k* & *numiter* might be necessary to achieve the best results (see 2.2). Nevertheless, we can confidently state that using the most basic approach as presented in the lecture and proposed by the assignment yields some of the best results and therefore is a good choice in our case.

2.2. Codebook construction (15%)

The implementation of the code for this part was fairly easy, as all that was needed was the correct use of the methods derived in 2.1. However, this part includes the (already pre-implemented) k-means clustering, which gave place to some interesting experimenting (while using the most basic approach of signed gradient orientations in the range 0-360°, no magnitudes and no proportions as investigated in 2.1.2). Namely, while keeping the number of iterations for the clustering at *numiter = 300* (which is the default value of *max_iter* in the KMeans class of sklearn), I have varied the value for the number of clusters k and observed average accuracies over 5 runs each:

k	**numiter**	**Average over 5 runs**	
		pos. sample acc.	**neg. sample acc.**
10	300	*0.9347*	0.8640
50	**300**	**0.9265**	**0.9200**
100	300	0.8286	*0.9720*
1000	300	0.7102	0.9880

Interestingly, varying k biases towards better accuracies for positive samples at lower k, and towards better accuracies for negative samples at higher k. The most balanced and overall highest accuracy is achieved for **k=50**.

Next, I have also altered the value of the number of iterations in the K-means clustering, *numiter*. At the same time I was keeping constant the previously found value for k:

k	**numiter**	**Average over 5 runs**	
		pos. sample acc.	**neg. sample acc.**
50	30	0.9143	0.9240
50	**300**	***0.9265***	**0.9200**
50	3000	0.8980	*0.9480*

The number of iterations doesn't seem to greatly affect the result, so staying with the default **numiter = 300** seems to be just fine.

2.3. Bag-of-words Vector Encoding (15%)

2.3.1 Bag-of-Words histogram (10%)

I order for this step to be computationally efficient, I tried to do it all in a vectorized form. Namely, I calculate each feature's distance to all possible cluster centers, and then attribute it to the nearest one by incrementing that index of the histogram by 1. The resulting histogram will show how many times each word of the codebook appears in a given image and our index is now complete.

2.3.2 Processing a directory with training examples (5%)

This part again was fairly simple, just using the methods derived in all the chapters before. As a result, we find the bag of words for all the images given to this method.

2.4. Nearest Neighbor Classification (10%)

For this step I again used a vectorized form, in order for it to be computationally efficient just as in 2.3.1. The image's distance to all positive and negative bag of words histograms gets calculated; if the minimum one is a distance to a negative sample, we assume the sample to also be negative (i.e. label 0, no car). Otherwise, we label it as positive, i.e. 1, car.

2.5. Discussion

This basic approach indeed works fairly well, seeing that we can reliably achieve accuracies of over 90% both for the positive and negative samples. For me this is rather surprising, as we are just 'randomly' choosing a grid of some features to observe and not some logical keypoints. However, as all the pictures are similar in how the cars are placed etc., in this particular case good accuracies can still be achieved. Therefore, for simple cases like this one our approach can be considered adequate.

If the dataset were more complex, I would mainly adapt the part of how the features to observe and form the words are chosen (SIFT etc.), and maybe use unsigned gradient orientations (0°-180°), this then with the rest of the code should still allow for good or even very good results.

3. CNN-based Classifier (40%)

3.1. A Simplified version of VGG Network (20%)

This part again was rather straightforward, following the proposed structure in the assignment. I decided to use *torch.nn.Sequential* instances to form the different convolutional blocks and the classifier. There was 3 small tricky parts: firstly, one had to be careful how to set the paddings, such that the output dimensions of each block ended up to be what was asked for (namely padding=1 for all *Conv2d*). Secondly, the output of the last convolutional block had to be reshaped correctly in the forward call in order for it to have the right dimension for the linear layers in the classifier. Finally, the classifier's middle layer size and dropout probability had to be decided on. For the dropout probability I chose the default value of 0.5, as this seemed to be a fair trade-off between avoiding overfitting and still giving the model a good portion to train on. For the middle layer size, I first chose to go with an intermediate value between 512 and 10, which are the input and output size of that classifier, namely 256. I have also tried a lower value of 32, which intuitively requires less 'downsampling' to the final prediction of the 10 classes in the last linear layer, however the results were slightly worse but still very comparable (see 3.2.1).

3.2. Training and Testing (20%)

3.2.1 Training

The training on my notebook took around 3 hours for 80 epochs (see 1). Apart from the number of epochs, I left all the parameters in the training file at the given value, as they seemed to be producing decent results just the way they were set. The model easily climbed over the 75% baseline during training already after around a third of those epochs for both middle layer sizes, as can be seen in the following figure:

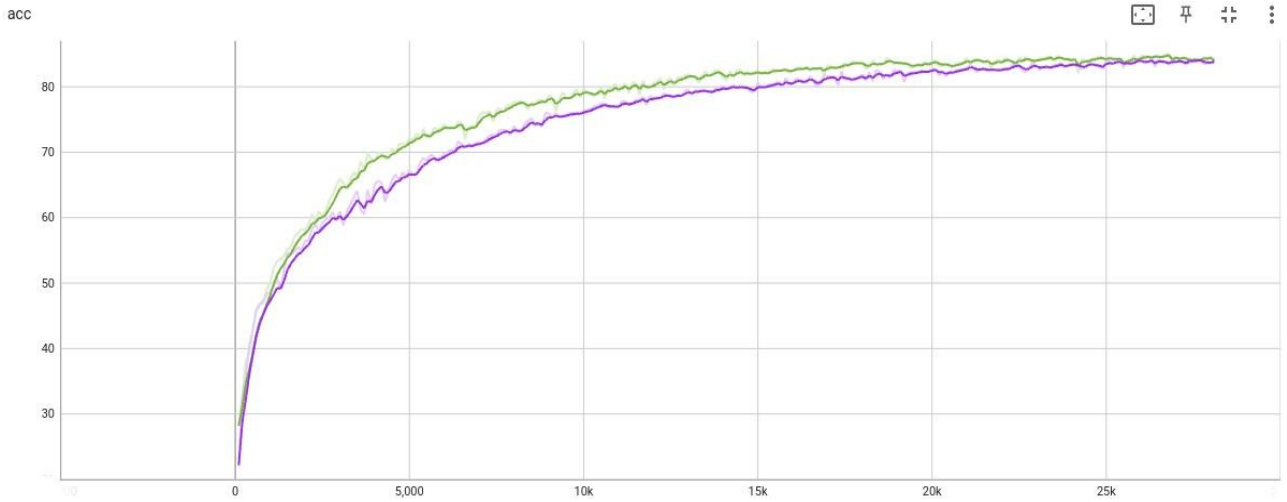

Figure 1: Accuracy over the whole training process for run 19118 (green, middle layer size 256), and run 58144 (purple, middle layer size 32)

A final accuracy of above 82 % in both cases is a pretty good result indeed. We can see that with a middle layer of size 256 in the classifier the accuracy tends to rise a bit quicker, but the saturation value to which the accuracy converges is very similar in both cases. The way the curve is flattening at around 82-84%, indicates that probably even with far more epochs we would not be able to achieve much higher accuracies. To do so, we would rather have to tweak the network.
The loss also consistently drops from 2 to finally under 0.5 respectively 0.2, as can be seen here:

Figure 2: Loss over the whole training process for run 19118 (green, middle layer size 256), and run 58144 (purple, middle layer size 32)

So the loss also decreases faster and more with a middle layer of size 256 in the classifier, however the resulting difference on accuracies is marginal as we were able to observe in Figure 1.
We can conclude that testing is both successful and bringing the expected result. The final accuracy certainly is a good result, also considering that we have only been implementing a simplified version of the VGG Network here. This shows that our model should be able to predict the class for most images right, unless it has been overfitted to the training data. This will be investigated in the next section 3.2.2.

3.2.2 Testing

Setting the right path to my model in *test_cifar10_vgg.py* and executing it results in the following output when using the model of my best run 19118 with classifier middle layer size 256:

```
[INFO] test set loaded, 10000 samples in total.
79it [00:05, 14.09it/s]
test accuracy: 82.6
```

Figure 3: Output for testing the model of run 19118 (middle layer size 256)

We can see that indeed the model was not overfitted to the training dataset and shows very similar results in accuracy after testing as well. Therefore the model consistently outperforms the 75% baseline and achieves even more than the expected results.

4. Hand-in

See together with this report:

- All my code (excluding the *data* folder)
- the *runs* folder generated by *train_cifar10 vgg.py* including training logs, trained model, the parameter setup and the tensorboard log file of my best run 19118 (see 3.2.1)
- Some images of the results in the folder *images*
- A file *CV_Assignment5_Experiments.ods* with the results of the investigation on the effect of the *k* & *numiter* values and the degree range, magnitude and proportion (see 2.1.2 & 2.2)

5. Sources

[1] Satya Mallick. *Histogram of Oriented Gradients explained using OpenCV.* LearnOpenCV, https://learnopencv.com/histogram-of-oriented-gradients/ (last consulted on 12.12.2021)

[2] Mrinal Tyagi. *HOG (Histogram of Oriented Gradients): An Overview.* Towards data science, https://towardsdatascience.com/hog-histogram-of-oriented-gradients-67ecd887675f (last consulted on 12.12.2021)