Correlation filter tracking

Peter Horvat

I. Introduction

Correlation filters are a class of classifiers, that are optimized to produce sharp peak in the correlation output in such way that the correlation output for the tracked object is high and the one for the background is low. This filter is then used for accurate localization of an object in a sequence of images. In the following report we will look at the *MOSSE* correlation filter that we implemented and tested on the *VOT2013* dataset. We compared the performance of the MOSSE correlation filter using different parameters, sequences and we also compared the results to two other algorithms, one of which was implemented in the previous assignment.

II. Experiments

For the implementation of all the methods described in the report we used the programming language *Python 3.8* and the testing sequences that were provided in the instructions. The source code for all three methods and all the supporting functions and utilities are available on GitHub. For comparing the speed of the initialization step and the tracking step we added a few lines of code to the *tracker.py* script, which generate a file with speed analysis and robustness calculation. The JSON file is saved in the results directory of the used tracked when using the provided *evaluate_tracker.py* script.

A. Correlation filters

The first step of building a correlation filter is to *initialize* a starting point, which is done in the initialize function of the MOSSE filter class. First we set some parameters, α which represents the update speed, σ which is used in the *create_gauss_peak* function and λ that we use when calculating the filter using the following function:

$$\hat{H}^{\dagger} = \frac{\hat{G} \odot \hat{F^{\dagger}}}{\hat{F} \odot \hat{F^{\dagger}} + \lambda}$$

Then we can set all the other parameters. First we set the starting position of for the tracker and the size of the object we are tracking. Next we calculate the desired correlation response *2D Gaussian* using the predefined σ and *size* parameters. Using the shape of the response $\mathbf{G}$ we extract a feature patch $\mathbf{F}$ from the image at the starting position which we then convert to *Grayscale format*. We also have to consider that both $\mathbf{G}$ and $\mathbf{F}$ are in the 2D Fourier domain. Lastly we calculate the correlation filter using the function we defined earlier and calculate the cosine window which will be used in the tracking step only to improve processing speed.

After we have set all the initial parameters we can start with the iterative part. For each tracking step we calculate $\mathbf{G}$ and $\mathbf{F}$ the same way as in the initialization step. After that we point-vise multiply the cosine window we defined earlier with the feature patch $\mathbf{F}$. Next we calculate the correlation filter H_t for the current image using the same function as before using the current values of $\mathbf{G}$ and $\mathbf{F}$.
The next step is *Localization*, this is done using the following equation:

$$R = f^{-1}\left(\hat{H}^{\dagger} \odot \hat{f}\right)$$

Here we calculate a 2D correlation response which is used to calculate the new location of the object, which we will explain a bit later.
After we have calculated everything we needed, we now can update the correlation filter. Using the current correlation filter H_t, correlation filter $\mathbf{H}$ of the previous image and the α parameter we now can update the $\mathbf{H}$ filter which will again be used as the *previous* $\boldsymbol{H}$. The update is done using the following function:

$$\hat{H_t^{\dagger}} = (1-\alpha)\,\hat{H_{t-1}^{\dagger}} + \alpha\hat{\hat{H}}^{\dagger}$$

At this point we have everything we need to calculate the new position of the object. By taking the coordinates x and y of the maxima of the 2D correlation response $\mathbf{R}$, adding them to the corresponding current position parameters x and y subtracting corresponding *width* and *height* divided by 2 we get the new x and y coordinates of our tracked object.
For the sake of testing our final result will be composed of two more components besides the coordinates, *width* and *height* of the tracked object. This will be than used to determine the tracking window for the object, which will be then used to calculate the average accuracy. We also extracted the robustness value from the *export_utils.py* script

B. Tracking performance evaluation

After doing some testing we came to an conclusion that the best result for the *VOT2013* dataset were received by setting the α to 0.3, λ to 10^{-5} and σ to 2.3. Using these parameters we got the results that are displayed in the table I.
By analyzing the properties of each sequence we concluded that sequences with objects that moves faster are performing much slower than others. For example in the sequence *bolt* the object that we track moves relatively fast in comparison to the sequence *car*. If we compare the results of the these two sequences we see that the faster sequence performs almost three times slower than the sequence with slower movement.
Now if we add the sequence called *face* where there is almost no movement of the object we can observe that the average accuracy of the sequence is much higher. This is because there is less chance for the observing window to move out of focus of the tracked object. We also noticed that the number of failures in the slower sequences was lower or even zero as we can see for the *face* sequence.
At the bottom of the table we can see the average accuracy for all of the sequences, total number of failures during tracking and the average speed for the whole dataset.

	Accuracy	Failures	Speed (FPS)
bicycle	0.43	12	1280.9
bolt	0.35	9	657.1
car	0.20	4	1663.8
cup	0.40	2	751.8
david	0.50	2	649.9
diving	0.31	0	1095.4
face	0.56	0	617.7
gymnastics	0.47	1	886.8
hand	0.27	12	507.0
iceskater	0.41	0	201.2
juice	0.32	31	1041.6
jump	0.24	1	478.5
singer	0.34	4	283.6
sunshade	0.36	10	1063.3
torus	0.29	9	1266.4
woman	0.42	8	716.2
Average/Total	0.37	105	822.6
Robustness		0.1575	

Table I: General performance data for *VOT2013* dataset

C. Speed analysis

In the following table II we can see that the initialization part of the tracker is mostly faster than the tracking part. There are more calculation steps needed in the tracking part which is probably the reason why in average the tracking part process less frames per second than the initialization part.

	Initialization (FPS)	Tracking (FPS)
bicycle	1623.9	1013.0
bolt	908.5	746.8
car	1666.1	1611.3
cup	748.4	663.3
david	643.1	556.5
diving	1000.1	1069.1
face	814.0	493.9
gymnastics	1000.1	953.4
hand	1002.0	685.1
iceskater	250.1	126.7
juice	1228.7	964.7
jump	666.5	408.8
singer	238.1	222.4
sunshade	1569.7	990.9
torus	2478.0	1247.1
woman	899.9	649.3
Average	**1100.2**	**790.7**

Table II: Speed comparison for *VOT2013* dataset

D. Comparing algorithms

To further test the performance of the algorithm we implemented, we compared it to the *NCC* and *Mean-Shift* tracker. From the data in table III we can see that the tracker we implemented is not as good as the NCC tracker, but the results are close enough to call it relatively efficient. The biggest difference we can see is the difference in the average overlap and the speed of the initialization step, which is more than ten times faster for the *NCC tracker*. On the other hand the algorithm we implemented in the previous assignment has more than double the number of failures and the robustness is more than ten times lower. The only advantage to the other algorithms is the average tracking step speed and the general tracking speed of the algorithm. This does not directly reflect to its efficiency, but it is a interesting observation.

	MOSSE	NCC	MS
Accuracy	0.37	0.62	0.31
Robustness	0.1575	0.1813	0.0134
Failures	105	97	245
Initialization (FPS)	1072.6	10813.6	1969.9
Track (FPS)	791.6	860.8	1694.6
Speed (FPS)	799.11	877.84	1701.89

Table III: Algorithm comparison

E. Connection between parameter change and performance

In the table IV we see the results of our testing at different parameters. At the first glance we can see that the overlap, robustness and number of failures is not affected at all. On the other hand if we observe the average speed at different α values, we can see that larger the value the faster the algorithm is. But there are still some anomalies, for which we could not find the reason of its occurrence.

	Test1	Test2	Test3	Test4	Test5
α - *alpha*	0.09	0.1	0.2	0.3	0.16
σ - *sigma*	3.3	1	2	2.3	3.8
Accuracy	0.37	0.37	0.37	0.37	0.37
Robustness	0.16	0.16	0.16	0.16	0.16
Failures	105	105	105	105	105
Initialize (FPS)	1061.2	1005.9	994.9	1002.0	1034.7
Track (FPS)	813.7	749.2	777.1	777.1	784.2
Speed (FPS)	787.9	796.4	821.6	827.1	792.3

Table IV: Effect of different parameters on the performance

III. Conclusion

There is a lot of room for improvement by refining the algorithm. By looking at the results we came to an conclusion that this implementation of the algorithm is a good starting point as the number of failures is not so terrible, but the average overlap value is below 50% which could still be improved, same goes for the number of failures that occur during the tracking of an object.