16-720J: Homework 4
Tracking Templates and Control Points

Instructor - Gary Overett, TA - Yang Gao

29 October 2015 - Due Midnight Thursday 12 November 2015

Instructions/Hints

1. Start early: This assignment involves plenty of implementation and cannot be debugged as easily since there are multiple inter-connected components.

2. **Start with small data**: Verify your system on small data before massive processing.

3. To speed up the running time, try to optimize your implementation using vector-based operation and avoid to loop for every pixel.

4. Extra credits are **optional**. Please make sure you have finished all the required questions before starting on the extra credit.

5. **Submission:** Your submission should be a **single zip** file `UniId.zip` that at least contains:

   • A Root Level Folder: `<UniID>` - so that we can unpack your work and uniquely identify it!

   • `<UniId>.pdf`: a pdf containing your answers to all written items in questions marked with a Q, including images or diagrams validating your technique. Your pdf should contain answers for Q1.1, Q1.2, Q2.1.2 and Q2.2.7. We do not accept handwritten scans in this assignment. It must also include code listings for all code you have written. We provide a LaTeXtemplate with examples for this, please use it.

   • `runTrackPooh_LK.m, SDMtrain.m, SDMtrack.m, genPerturbedConfigurations.m, genDisplacementMatrix.m, genFeatureMatrix.m, learnMappingAndUpdateConfigurations.m`: requirements of Q2.

   • `pooh_lk.avi, pooh_sdm.avi`: videos that demonstrates your tracking results (requirement of Q2.1.1 and Q2.2.6 respectively).

   • Any other `.m` files which you wrote for the homework.

   • Remove data and temporary files: in `<UniId>.zip`, make sure you have removed the folder `data/`, `lib/` that we do not ask you to submit. When grading, the functions in `lib/` will be provided to your code, so you do not need to upload them.

   • Remove commented code that is part of your prior work. We will deduct marks for leftover code that we find making your work harder to read.

   • If you have questions, please post them on the BlackBoard first.

Figure 1: **Lucas-Kanade tracking:** The bounding box of the car in the first frame is given, and the Lucas-Kanade tracker will locate the cars position in the rest of the frames.

Overview

This homework consists of two parts. In the first part, you will run the provided Lucas-Kanade (LK) tracker [1] with a fixed template and track a car. You will need to answer some theory questions. In the second part, you will track a "cuter" object, Winnie the Pooh, using the provided LK tracker and the state-of-the-art control point tracker based on Supervised Descent Method (SDM) [3] which you will implement.

1 The Car Tracker: Template Tracking with Lucas-Kanade (10 pts)

In this section, we will track a speeding car in an image sequence (`data/car/carSequence.mat`) by template tracking. See Figure 1 for an illustration on frame 1 (with initialized bounding box), and frames 20, 50 and 100 of the input video.

The first groundbreaking work on template tracking was the **Lucas-Kanade tracker** (see Figure 2 in [1] for a review of the algorithm). It basically assumes that the template undergoes constant motion in a small region. The Lucas-Kanade Tracker works on two frames at a time, and does not assume any statistical motion model throughout the sequence. The algorithm estimates the deformations between two frames under the assumption that the intensity of the objects has not changed significantly between the two frames.

Starting with a rectangle R_t on frame I_t, the Lucas-Kanade Tracker aims to move it by an offset (u, v) to obtain another rectangle R_{t+1} on frame I_{t+1}, so that the pixel squared difference in the two rectangles is minimized:

$$\min_{u,v} J(u,v) = \sum_{(x,y)\in R_t} (I_{t+1}(x+u, y+v) - I_t(x,y))^2$$

Question 1.1 Warmup (5pts)

Starting with an initial guess of (u, v) (usually $(0, 0)$), we can compute the optimal (u^*, v^*) iteratively. In each iteration, the objective function is locally linearized by first order Taylor expansion and optimized by solving a linear system that has the form $A\Delta p = b$, where $\Delta p = (u, v)^T$ is the template offset.

- What is $A^T A$?
- What conditions must $A^T A$ meet so that the template offset can be calculated reliably?

Question 1.2 Discussion (5pts)

Now you have warmed up, run the provided script `runTrackCar.m`, which runs the template-based Lucas-Kanade tracker on the car sequence. How does it go? In your answer sheet, give a short analysis on possible scenarios the tracker could fail; give possible solution(s) to remedy these problems.

2 The Pooh Tracker: Component-based Tracking (90 pts)

You may have found that in Section 1, the template tracker requires a fixed template and a good initial estimate. For tracking objects with appearance variations (e.g., viewpoint changes), we need to track local components instead of the template as a whole. In this section, you will implement a "cuter" tracker, The Pooh Tracker. We want to track the five components (nose, two eyes and two ears) of a Winnie the Pooh doll as shown in Figures 2 and 3. The hope is that by tracking the local components, the tracker becomes more robust to variations such as viewpoint changes. Moreover, tracking local components provides finer-grained information about the object. Specifically, we will implement The Pooh Tracker with Lucas-Kanade tracker and the state-of-the-art SDM tracker.

Figure 2: The Pooh Tracker using the LK Tracker. We will track five components, including the nose, eyes and ears of Wiinnie the Pooh. The top left figure shows frame 992 with provided rectangles that we will use to initialize the tracker.

2.1 Tracking the Pooh with the LK Tracker

Question 2.1.1 Implementation (8 pts) To track the components on the Pooh (nose, eyes and ears), we will first reuse the LK tracker provided in Section 1. Note that we will now treat each component instead of the whole Pooh as a template, i.e., we will need one LK tracker to track each component. Write a script `runTrackPooh_LK.m` to track Winnie the Pooh using LK tracker provided in `lib/LucasKanade.m`. We will be tracking the test frames starting from `image-0992.jpg`, which are provided in the directory `data/pooh/testing/`. The initial rectangles for each component in frame 992 are provided in `data/pooh/rects_frm992.mat`. Feel free to use your own initial rectangles. Each rectangle contains 4 values `[topLeftX, topLeftY, bottomRightX, bottomRightY]`, indicating the coordinates of the top-left and the

Figure 3: The Pooh Tracker using SDM. Four sampled frames with different viewpoints illustrate the effectiveness of the SDM tracker.

bottom-right corners of the rectangle. Use the five initial rectangles and the provided Lucas-Kanade tracker to track from frame 992 to 3000. Save your tracking results as `pooh_lk.avi`. To generate a output video, you may use the provided function `drawRect.m` to plot the rectangles, and/or modify the provided script `runMakeVideo.m`. Your tracking result should look similar to Figure 2.

Note: Many people have found that when the LK tracking rectangle runs out-of-bounds, `LucasKanade.m` errors. To fix this, one could pad zeros for out-of-bounds portions of the rectangle, but for this homework one quick fix is to translate the rectangle back into the valid region of the image and provide that rectangle to `LucasKanade.m`. Another fix is to ignore the out-of-bounds rectangle and use the previous rectangle that was still in-bounds. Or, if the rectangle has already lost track of what it is supposed to track, then you can just stop your tracker there and save your `pooh_lk.avi`. Make sure you close your `VideoWriter` when ending your tracker, e.g. `close(vidobj)`, otherwise your video will not be saved properly.

Question 2.1.2 Discussions (2 pts) Can it track until frame 3000? If not, which frame does it lose track? By losing track, we mean that the Intersection over Union (remember Homework 3?) is less than 0.5. You do not need to write code for this. Just try to estimate this visually. Please use ≤ 2 sentences to answer when it fails, and the reason you think why it fails. Show the frame where you think the tracker loses track.

Question 2.1.3 Extra Credit (10 pts) Try modifying the LK tracker such that it can track Winnie the Pooh for more frames. Any techniques or modifications are welcomed.

2.2 Tracking the Pooh with Supervised Descent Method (SDM) Tracker

You may have already found that the Pooh Tracker with LK is unsatisfactory at some point. In order to track the cute Pooh longer, we will implement the state-of-the-art Supervised Descent Method[1] (SDM) with sparse SIFT descriptors. That is, in contrast to the dense SIFT in Homework 3, we will only extract SIFT descriptors centered at each component.

[1]To gain more details please refer to Sections 3.1 and 3.2 of [3].

Intuition of SDM: We take tracking one component (e.g., nose) for explanatory convenience. Note that the same intuition applies to tracking all 5 components (e.g., nose, eyes, ears) at the same time. Given an estimated location and a true location of a component, we can compute the displacement (difference in x and y) between the two locations. Given the displacement, the current estimated location only needs to move according to the displacement to coincide with the true location, thus perfectly locating/tracking the component. However, in the tracking scenario, the actual location of the component is unknown, and thus making it impossible to compute the displacement this way. Therefore, what SDM does is to predict the displacement required to move the current estimated location of the component to the actual location.

In order to predict the displacement during tracking, SDM requires a training step. During training, SDM establishes a relationship between displacements and feature descriptors (e.g., SIFT descriptors) extracted from the locations of all the components. Utilizing the SIFT descriptors from all the components enables a more robust estimation to the displacement of each component, because the global shape of the tracked target is encoded by the SIFT vectors of each component. To establish the relationship between displacements and feature descriptors, one could find a linear mapping function which maps feature descriptors to displacements. However, as the relationship between displacements and feature descriptors is complex, one linear mapping function may not be enough. Therefore, to approximate the complex relationship between displacements and feature descriptors, SDM iteratively learns a sequence of linear mappings. This sequence of linear mappings is the final output of the SDM training phase.

During the SDM tracking phase, we utilize a sequence of linear mapping functions learned during training to predict the displacement of each component. First, an initial estimate of each component is acquired based on the mean shape and the location of the object in the previous frame. Then the initial estimates are iteratively refined based on the displacements predicted by the sequence of linear mappings learned during training. If the linear mappings are learned appropriately, the component locations will coincide with the actual location of the component. This is how SDM performs tracking.

For simplicity, the above text describes how to perform tracking for a single component. However, we are interested in tracking the location of all components simultaneously. Therefore, during training, the linear mapping simultaneously learns the relationship between the SIFT feature vectors and the displacement for **all components**.

The following paragraphs will present more details of each step.

SDM training phase

In the training phase, SDM will learn a series of mappings between feature descriptors and displacement. In this paragraph, we first describe how to learn one mapping, and then generalize the process to produce a series of mappings. To learn a single mapping, one needs training data. The training data provides the information: given feature descriptors extracted from a configuration, what displacement is required to move the components to the ground truth location? To generate training data, we utilize ground truth annotations of component locations in a video frame. For a single frame with ground truth, we generate many copies of the mean shape configuration and place them at random locations near the ground truth. From now on, we will call each of the randomly placed mean shapes as a perturbed configuration. For each perturbed configuration, we can compute a displacement required to translate it to the ground truth configuration. The feature descriptors for the perturbed configurations are also computed. The computed displacements and the feature descriptors can then be used for training. We repeat this process on all frames with ground truth. Then we aggregate the feature descriptors and displacements from all frames to create a training set. The training

set is used to learn the mapping. Once a mapping is trained, the mapping is predicted back on the training set to get a set of predicted displacements. The predicted displacements are then used to update the location of the perturbed configurations. Given the updated perturbed configurations, we recompute the feature descriptors at the new locations. We also update the displacement between the ground truth. Then, with the updated configurations and data, we learn the next mapping.

The main reason this method is reasonable in generating training data is because the perturbed configurations mimic the tracking scenario. A perturbed configuration is analogous to the initial estimate of a tracked object for the current frame. The ground truth is the actual location of the object in the current frame. The mapping learns how to transform the initial configuration to the actual configuration for the current frame. The mean shape is used to generate the perturbed configurations to achieve robust performance. This is the reason why we generate training data in this manner. Also, to mimic the scale changes of the tracked target during tracking, one should also scale the mean shape at different scales when generating perturbed configurations.

To train the SDM tracker, we provide $m(m = 10)$ training images in `data/pooh/training/`. Each training image comes with annotations (5 points of nose, eyes and ears) in `data/pooh/ann`. Run the provided script `runPoohInfo.m` to view the training images and annotations. You will need about 5 mappings to produce reasonable tracking results. In summary, the main training steps include:

1. given the true locations (annotations) of the 5 components in each training image, generate a set of n perturbed configurations for each annotation. You will get mn perturbed configurations.

2. given n perturbed configurations from all m training images, generate a mn-by-10 displacement matrix D. Each row of the displacement matrix stores the displacement in x and y for all 5 components for a single configuration, thus resulting in 10 dimensions.

3. for all m training images, extract SIFT descriptors centered at each perturbed configuration. For each perturbed configuration, concatenate the 5 SIFT vectors to get a $5 \times 128 = 640$ dimensional array. Combine the features from all perturbed configurations to get a mn-by-640 feature matrix F.

4. given the feature matrix F and displacement matrix D, learn a linear mapping $W \in \mathbb{R}^{640\times 10}$ with the least squares criterion.

5. use W to compute the displacement of F, and then use the displacement to update the location of each component in each perturbed configuraton.

6. repeat steps 1-4 until convergence

Figure 4 illustrates the flow chart for the training phase. Below we will implement the SDM training phase step-by-step.

Question 2.2.1 Training step 1: perturbation (10 pts) We will implement training step 1 to generate a set of perturbed annotations, which corresponds to step (b) in Figure 4. The goal of the training phase is to learn how to transform an initial shape to a given annotation. To synthesize such transformation, we will slightly perturb an initial shape by translation and scaling to mimic the tracking scenario, i.e., the object will move a few pixels in the next frame. Implement `genPerturbedConfigurations` with the following steps:

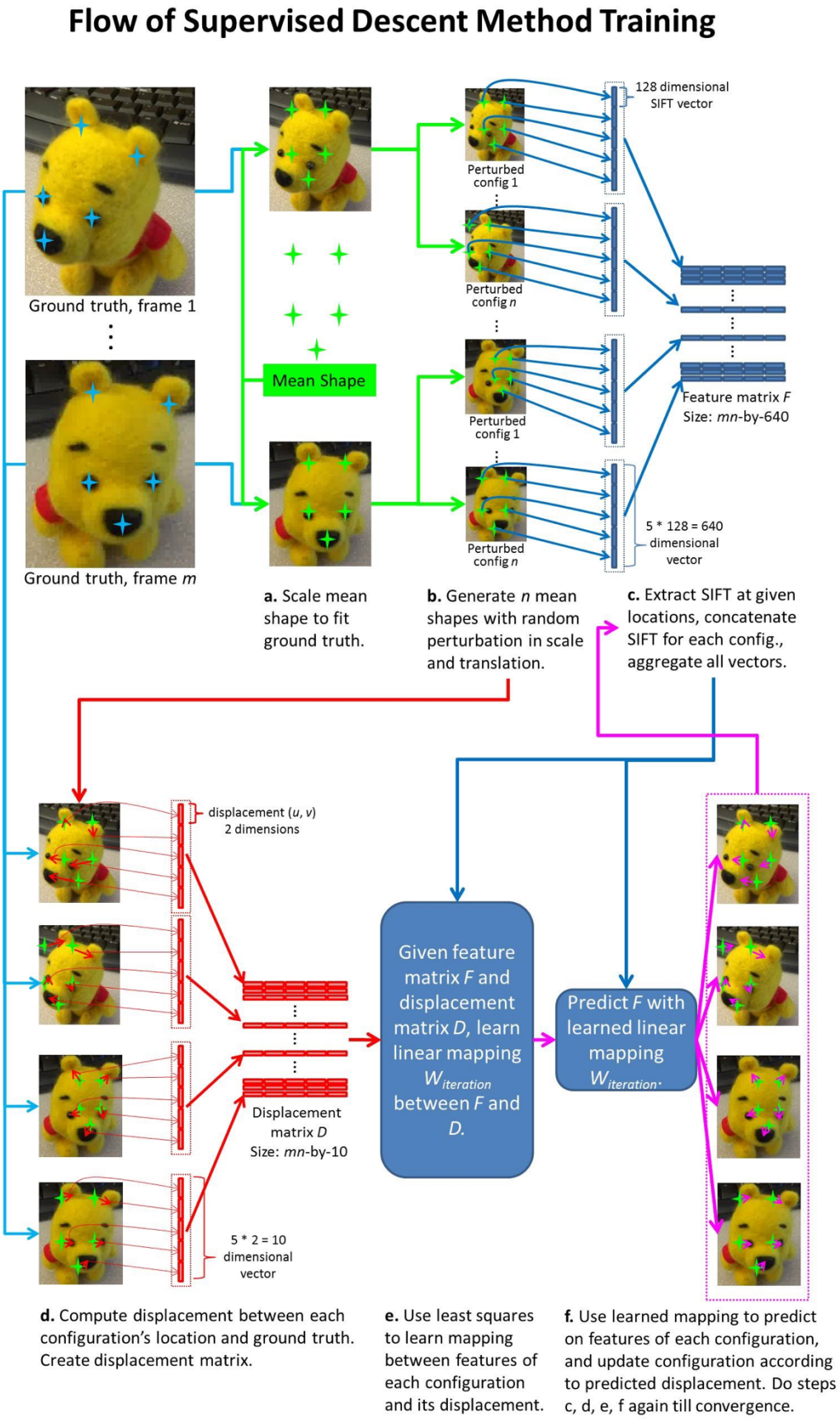

Figure 4: Flow for SDM training

1. Initialize the mean shape (a 5×2 matrix provided in `data/pooh/mean_shape.mat`) at the center of each frames annotation. Each row indicates the coordinates of each components. Also, scale the mean shape according to the size of the frames annotation accordingly with the `lib/findscale.m` provided. (see step (a) of Figure 4)

2. Perturb the initial mean shape by translation and scaling for n times. Note that you should apply the same perturbation to all 5 components and not a different translation and scale for each component. Each set of perturbed location contains 5 points corresponding to the location of each component. You have to be careful in making sure that your perturbed examples actually matches the tracking scenario during testing, because we are trying to mimic the tracking scenario here (e.g., in general Winnie moves less than 10 pixels per frame in our sequence). You might find `randn` useful. (see step (b) of Figure 4)

Please implement the function:

```
perturbedConfigurations = genPerturbedConfigurations(singleFrameAnnotation,
                          meanShape, n, scalesToPerturb)
```

where `singleFrameAnnotation` is a 5-by-2 matrix storing the ground truth (x, y) locations of the 5 components for a given frame. `meanShape` is a 5-by-2 matrix storing the mean shape of the 5 components. n is a scalar which specifies how many perturbed configurations to be sampled (100 is a good starting point). `scalesToPerturb` is a vector which specifies how many different scales to scale the mean shape ($[0.8, 1.0, 1.2]$ is a good choice, but when you are debugging, it is recommended to use just 1.0 for simplicity). The output `perturbedConfigurations` is a 4-by-$(n \times 5)$ matrix which is the format `lib/siftwrapper.m` expects, and has the following format:

$$\begin{bmatrix} x_{\text{nose 1}} & x_{\text{left eye 1}} & x_{\text{right eye 1}} & x_{\text{right ear 1}} & x_{\text{left ear 1}} & x_{\text{nose 2}} & x_{\text{left eye 2}} & \dots \\ y_{\text{nose 1}} & y_{\text{left eye 1}} & y_{\text{right eye 1}} & y_{\text{right ear 1}} & y_{\text{left ear 1}} & y_{\text{nose 2}} & y_{\text{left eye 2}} & \dots \\ s_{\text{nose 1}} & s_{\text{left eye 1}} & s_{\text{right eye 1}} & s_{\text{right ear 1}} & s_{\text{left ear 1}} & s_{\text{nose 2}} & s_{\text{left eye 2}} & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \end{bmatrix} \tag{1}$$

Each column indicates a location on which you will extract a SIFT descriptor, and has the format $[x, y, s, r]^T$, where x and y are the coordinates, s is the scale of the SIFT point (how large area the SIFT descriptor should cover), and r is the orientation (fixed to 0). The TAs have found that a suitable SIFT scale for the nose, left eye, right eye, right ear and left ear on the mean shape is $[7, 4, 4, 10, 10]$ respectively. However, as the size of the object changes, the SIFT scale should also change. Therefore, you should adjust the SIFT scales for perturbed configurations accordingly You may use the provide function `lib/findscale.m` to get an estimated scaling factor. Each perturbed configuration corresponds to 5 consecutive columns in the `perturbedConfigurations` matrix.

Question 2.2.2 Training steps 2&3: prepare D and F (10 pts) The goal here is to prepare the displacement matrix D and the feature matrix F for learning a mapping.

Given m training images, generate a $mn \times 10$ displacement matrix D, where each row contains the displacements between an initial shape and a true annotation of 5 components. The displacement should be computed for the (x, y) for the 5 components, so you have 10 displacement values for each configuration (i.e., locations of 5 components). See step (d) in Figure 4 for an illustration. For the assignment, please implement the following function in `genDisplacementMatrix.m`:

```
D = genDisplacementMatrix(...)
```

You can decide the inputs to the function, but the output is the displacement matrix D.

To generate the feature matrix, you will first pass the `perturbedConfigurations` matrix to the provided `lib/siftwrapper.m` to extract the SIFT descriptors. Then, create a (5×128)-dimensional feature vector by concatenating the SIFT descriptors of the 5 components for a single perturbed configuration. Stack all feature vectors from all perturbed configurations in all frames to generate the $mn \times 640$ feature matrix F. You need to make sure that the rows in D and F are ordered in the same way, i.e., the i-th row in F corresponds to the displacement of the i-th row in D. See step (c) in Figure 4 for an illustration. For the assignment, please implement the following function in `genFeatureMatrix.m`:

```
F = genFeatureMatrix(...)
```

You can decide the inputs to the function, but the output is the feature matrix F.

Question 2.2.3 Training steps 4&5: linear mapping and update configuration (10 pts) Now we have the displacement matrix $D \in R^{mn \times 10}$ and the feature matrix $F \in R^{mn \times 640}$. The goal here is to learn a linear mapping $W \in R^{640 \times 10}$ by solving the linear least square problem $W = \arg\min_W ||FW - D||_F^2$. Recall this problem can be solved using pseudo-inverse (which was also used in the LK tracker). See step (e) in Figure 4 for an illustration. To assist you in solving the linear least square problem, we provide a function:

```
W = learnLS(F,D)
```

in `lib/learnLS.m` where D is the displacement matrix, F is the feature matrix, and W is the linear mapping.

Given a feature vector f and the learned linear mapping W, you can now predict a 1×10 displacement vector using fW. Use the learned linear mapping W to update the current location of the 5 components (see step (e) and (f) of Figure 4). In the homework, please implement the following function which learns the mapping and updates the locations of the perturbed configuration in `learnMappingAndUpdateConfigurations.m`.

```
[...]  = learnMappingAndUpdateConfigurations(...)
```

You can decide the inputs and outputs to this function.

Question 2.2.4 Sequentially learn multiple mappings (10 pts) Until now you have implemented the training phase for learning one mapping that predicts a displacement vector from a feature vector. However, updating the configuration (locations of 5 components) only once is generally not enough to precisely converge to the ground truth annotation. We will need more mappings for predicting a more precise configuration.

Specifically, based on the updated configuration in step 5, repeat steps 2-5 until you have 5 mapping matrices W. Each mapping progressively moves your initial configuration toward the ground truth annotation. One thing you should self-check is whether your loss function is decreasing, i.e. whether the Euclidean distances between your current estimate and the actual ground truth is decreasing. If it did not decrease, then something is wrong.

Question 2.2.5 Integration (15 pts) Please integrate all the code you have written and update the provided `SDMtrain.m`.

SDM Prediction

We now elaborate on the prediction and tracking using SDMs. During training, we learned the mapping between concatenated SIFT vectors and displacement to a ground truth location of each component. However, in prediction time, we do not have the ground truth location, so we would like to utilize the mapping learned during training to predict the location of each component. The location of each component can be estimated by updating an initial estimate of the objects location with the displacement predicted by the mapping. The initial estimate is the mean shape. As one mapping function is usually not enough, we iterate this update process multiple times. The number of iterations depend on the iterations used during training.

During implementation, we have to ensure that our prediction procedure is coherent with the training procedure so that we can utilize the mapping learned in training. Below are the detailed steps of SDM prediction, which is also visualized in Figure /refflowpred.

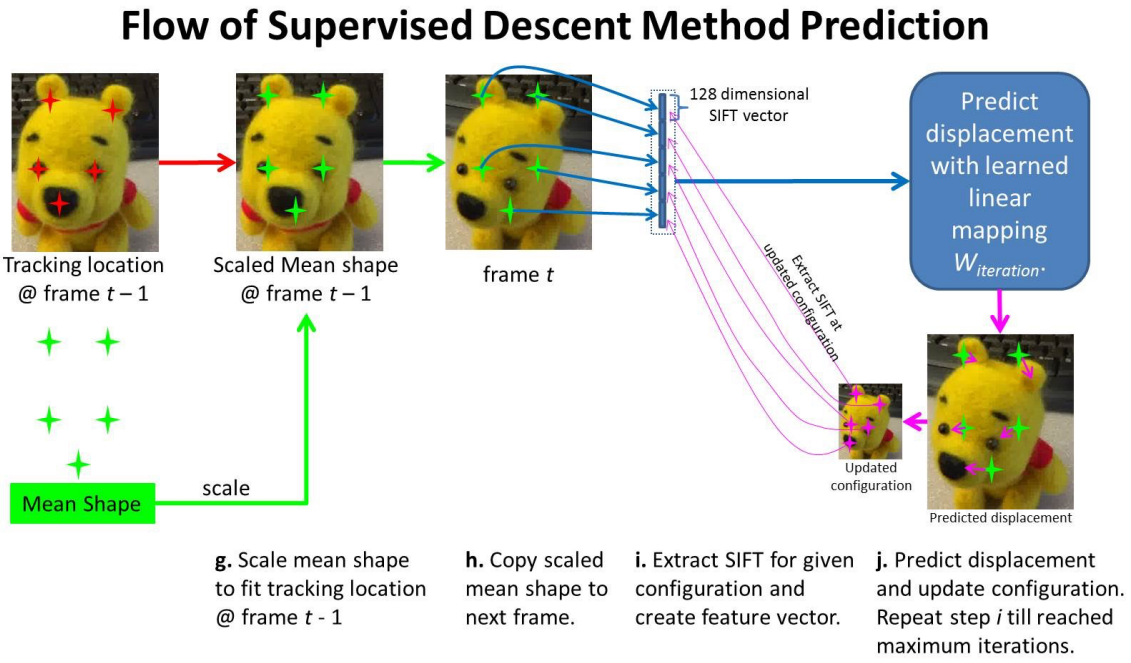

Figure 5: Flow for SDM prediction/tracking

1. Given the tracking location of each component in the previous frame, translate and scale the mean shape so that it "fits" the location of the previous frame. For translation, the mean shapes center should match the objects center in the previous frames. For scale, you can use `lib/findscale.m`. The scaled mean shape serves as the starting point for SDM on the current frame. This step corresponds to step (g) and (h) in Figure 5.

2. Extract SIFT features for the current configuration estimate and create the concatenated feature vector. Just like what you did in training, the scale of the SIFT features should also be rescaled according to the scale of the mean shape. This step corresponds to step (i) in Figure 5.

3. Apply the mapping learned during training to each component. The mapping corresponding to the current iteration should be used. This step corresponds to step (j) in Figure 5.

4. Update the location of your current configuration according to the predicted displacement computed in the previous step and go back to step 2. Iterate until you reached the number of iterations used in training. This step corresponds to step (j) in Figure 5.

Question 2.2.6 Implementation (20 pts) Please integrate all the code you have written and update the provided `SDMtrack.m`. `SDMtrack.m` will generate an output video `pooh_sdm.avi`. Please include it in your submission. The testing code starts from frame 992 and ends at frame 3000. If your tracker cannot track for at least 200 frames, then something is wrong. If your tracker survives till frame 1400, you get full marks.

The TAs have provided a master script `runTrackPooh_SDM.m` which runs `SDMtrain.m` and `SDMtrack.m`. Please make sure your tracker runs without errors when it is called from `runTrackPoohSDM.m`. This will be used for grading. Do not modify `runTrackPooh_SDM.m`.

Question 2.2.7 Discussions (5 pts) You tried two different trackers on the same sequence. Which one performs better, and why? List two advantages and disadvantages each for the LK and SDM trackers respectively on this part-based tracking task.

Question 2.2.8 Extra credits (3+3+3 pts) Try overcoming these challenges. The first challenge starts at frame 1410 where there is significant scale change (3 pts). If you tracker loses track, try adjusting the `scalesToPerturb` to suit your needs. The second challenge is around frame 2232 where Winnie becomes super small (3 pts). The final challenge is at frame 2452 where there is slightly bigger pose change (3 pts). Even the TAs tracker cannot perfectly deal with the final two challenges.

Question 2.2.9 Extra credits (10 pts) Start tracking from frame 3511 in `data/pooh/QX2.2.9_extra_credit`. Winnie rotates by 180 degrees in the next few frames. Try adding rotation perturbation in your `genPerturbedConfigurations` code and track the rotated Winnie. You can add extra positive examples, but do not add extra positive examples between frames 3511 and 4100. Please submit a video called `pooh_sdm_rotated.avi` if you succeed.

3 Tips for Writing Tracking Code

Trackers in general do not require many lines of Matlab, but a small error can cause a complete failure in tracking. Therefore, we strongly advise you to check each step of your code. When facing a bug, avoid running the whole pipeline to debug. Try to find or manually create the simplest possible examples and see if the program does what you expect. Usually, there is some logical error, or there is a plus sign which should be a minus, or a multiplication instead of a division. Here are some more tips to help you when you run into bugs. Good luck.

1. Remember to read `runTestSIFT.m` and `runPoohInfo.m` carefully. It visualizes the ground truth and also tells you how to run SIFT from vlfeat [2] properly. It also gives you a hint for what SIFT scales to select for each of the components.

2. If you normalize the SIFT vectors so that they are unit vectors, performance might improve. Dont forget to do it for both training and testing.

3. Make sure what you are doing during training and testing time is EXACTLY the same: same normalization, same scaling, etc. If not, machine learning assumptions fail and your tracker will not perform well.

4. If the tracker does random things, try training on a frame, and then predict directly on the same frame with the same initial value as during training. If the prediction is not what was provided in training, then something is very wrong.

5. Non-ideal perturbation by translation and scale may cause your tracker to fail. Try gradually adding different perturbations and avoid adding all of them at once.

6. There are multiple "scales" in play. The first "scale" is the difference in size between the tracked object and the mean shape. The second "scale" is the scale of the SIFT feature points, which should change according to the first "scale".

7. In general, plotting and visualizing your output is a great way to debug.

References

[1] S. Baker and I. Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal of Computer Vision*, 56(3):221–255, 2004.

[2] A. Vedaldi and B. Fulkerson. Vlfeat: An open and portable library of computer vision algorithms. In *Proceedings of the international conference on Multimedia*, pages 1469–1472. ACM, 2010.

[3] X. Xiong and F. De la Torre. Supervised descent method and its applications to face alignment. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 532–539. IEEE, 2013.