Computer Vision II - Homework Assignment 3

Stefan Roth, Xiang Chen, Jochen Gast, Anne Wannenwetch, Junhwa Hur
Visual Inference Lab, TU Darmstadt

June 5, 2019

This homework is due on July 3rd, 2019 at 11:00 am.
Please read the instructions carefully!

General remarks

Your grade does not only depend on the correctness of your answer but also on clear presentation of your results and good writing style. It is your responsibility to find a way to *explain clearly how* you solve the problems. Note that we will assess your complete solution and not exclusively the results you present to us. If you get stuck, try to explain why and describe the problems you encounter – you can get partial credit even if you have not completed the task. Hence, please hand in enough information so that we can understand what you have done, what you have tried, and how your final solution works.

Every group has to submit its own original solution. We encourage interaction about class-related topics both within and outside of class. However, you are not allowed to share solutions with your classmates, and *everything you hand in must be your own work.* Also, you are not allowed to just copy material from the web. You are required to *acknowledge any source of information you use to solve the homework* (*i.e.* books other than the course books, papers, websites, etc). Acknowledgments will *not* affect your grade. Not acknowledging a source you rely on is a clear violation of academic ethics. Note that both the university and the department are very serious about plagiarism. For more details, see the department guidelines about plagiarism at `https://www.informatik.tu-darmstadt.de/studium_fb20/im_studium/studienbuero/plagiarismus/index.de.jsp` and `http://plagiarism.org`.

Programming exercises

For the programming exercises you will be asked to hand in Python code. Please make sure that your code complies with **Python 3.x / PyTorch 1.1**. In order for us to be able to grade the programming assignments properly, stick to the function names and type annotations that we provide in the assignments. Additionally, comment your code in sufficient detail so that it will be easy to understand for us what each part of your code does. Sufficient detail does not mean that you should comment every line of code (that defeats the purpose), nor does it mean that you should comment 20 lines of code using only a single sentence. Your Python code should display your results so that we can judge if your code works from the results alone. Of course, we will still look at the code. If your code displays results in multiple stages, please insert appropriate `sleep` commands between the stages so that we can step through the code. Group plots that semantically belong together in a single figure using subplots and don't forget to put proper titles and other annotations on the plots. Please be sure to name each file according to the naming scheme included with each problem. This also makes it easier for us to grade your submission. And finally, please make sure that you included your name and email in the code.

Files you need

All the data you will need for the problems will be made available in Moodle.

What to hand in

Your hand-in should contain a PDF file (a plain text file is ok, too) with any textual answers that may be required. You must not include images of your results; your code should display these instead. For the programming parts, please hand in all documented `.py` scripts and functions that your solution requires. Make sure your code actually works and that all your results are displayed properly!

Handing in

Please upload your solution files as a single `.zip` or `.tar.gz` file to the corresponding Moodle area at `https://moodle.tu-darmstadt.de/course/view.php?id=15293`. **Please note that we will not accept file formats other than the ones specified!** Your archive should include your write-up (`.pdf` or `.txt`) as well as your code (`.py` scripts). If *and only if* you have problems with your upload, you may send it to `cv2staff@visinf.tu-darmstadt.de`

Late Handins

We will accept late hand-ins, but we will deduct 20% of the total reachable points for every day that you are late. Note that even 15 minutes late will be counted as being one day late! After the exercise has been discussed in class, you can no longer hand in. If you are not able to make the deadline, *e.g.* due to medical reasons, you need to contact us *before* the deadline. We might waive the late penalty in such a case.

Code Interviews

After your submission, we may invite you to give a code interview. In the interview you need to be able to explain your written solution as well as your submitted code to us.

Python Environment

Please follow the instructions in `Readme.txt` to set up your environment.

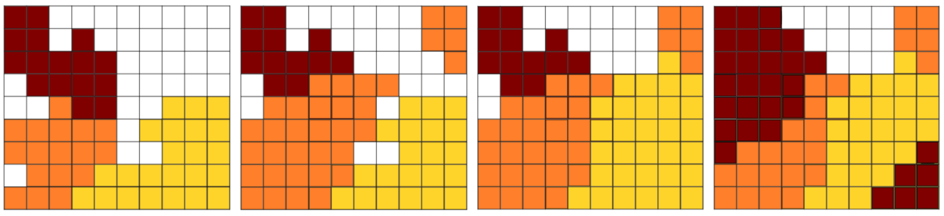

Figure 1: The alpha-expansion algorithm breaks a multi-class labeling problem down into a series of binary sub-problems. At each step, we choose a label α and expand: For each pixel either leave the label as is or replace it with α. Each sub-problem can be solved in a globally optimal way. a) Initial labeling. b) Orange label is expanded: Each label stays the same or becomes orange. c) Yellow label is expanded. d) Red label is expanded.

Problem 1 – Graphcuts and α-expansion using Numpy 17 points

Recall the α-expansion algorithm presented in class. It provides a way of doing inference on a Markov random field with multiple labels by breaking the solution down into a series of binary problems, each of which can be solved exactly. The main idea is to iterate through each label, where in each iteration all nodes are given the choice to either keep their current label or to switch to the new label, *c.f.* Fig. 1. Here, we will implement an approximate version of the α-expansion algorithm, which will be performed step-wise in `problem1.py`. As stereo likelihood we assume a Laplacian, *i.e.*

$$p(\mathbf{I}^0 \,|\, \mathbf{I}^1, \mathbf{d}) = \prod_{i,j} \frac{1}{2s} \exp\Big(-\frac{|I^0_{i,j} - I^1_{i,j-d_{i,j}}|}{s}\Big). \tag{1}$$

Tasks:

- Implemented the function `edges4connected(height, width)` that takes the dimensions of a MRF and creates edges for a four-connected grid graph, *i.e.* a graph where each node is connected to its four neighbors. The resulting dimensions should be $|E| \times 2$. Each row indicates the linear index of the two nodes which are connected by that edge. Please be aware that Numpy uses a row-major ordering as opposed to Matlab's or Julia's column major ordering. Hence, we expect your result to follow a row-major layout.

  2 points

- Implemented the function `negative_log_laplacian(x, s)` which evaluates an element-wise negative log Laplacian corresponding to one factor in Eq. (1). Here, `x` corresponds to the inner difference of the images, *i.e.* $x \sim I^0_{i,j} - I^1_{i,j-d_{i,j}}$.

  1 point

- Implement the function `negative_stereo_loglikelihood(i0, i1, d, s, invalid_penalty)` to compute the elementwise negative stereo log likelihood (Eq. (1)). In contrast to Assignment 2, no warping is required due to the discrete disparities. Since we can run into the problem that given disparities yield positions outside the valid image range, please evaluate these invalid regions to the given penalty. You should use `negative_log_laplacian` with parameter `s` to evaluate the elementwise Laplacian (negative log) values.

  4 points

- Implement the α-expansion algorithm in

  `alpha_expansion(i0, i1, edges, d0, candidate_disparities, s, lmbda)`

  where `i0`, `i1` are the given stereo images, `edges` encode the MRF neighborhood, `d0` are the initial disparities, `candidate_disparities` contain a list of disparities to be considered in the optimization, `s` is the shape parameter of the likelihood, and `lmbda` is the regularization parameter for the MRF. As a result the final disparity estimate should be returned.

  For this algorithm, you need to perform a binary graph cut at each iteration. To that end, make use of the provided `graphcuts` function which performs a binary s-t cut given unary and pairwise potentials. To compute the unary potentials you should use the negative stereo log likelihood from above. Use a standard Potts model (with parameter `lmbda`) for your pairwise potentials, where a constant penalty is imposed on `edges` with different labels:

$$E_{pq}(\alpha, \beta) = \begin{cases} 0, & \text{if } \alpha = \beta \\ \lambda > 0, & \text{otherwise.} \end{cases} \tag{2}$$

  To make the call to `graphcuts` efficient, your pairwise term should be stored in a sparse matrix in upper triangular form as explained in `gco.py`.

  6 points

- Visualize your stereo results in `show_stereo(d, gt)`. For this visualization, please only display disparities in the valid range of the ground truth disparities (gt > 0).

  1 point

- Evaluate your results by computing the percentage of falsely labeled pixels w.r.t. the ground truth. Implement this evaluation in `evaluate_stereo(d, gt)`. Again, only take pixels into account with a valid ground truth disparity (gt > 0). Where does the graph cuts algorithm have problems in particular?

  3 points

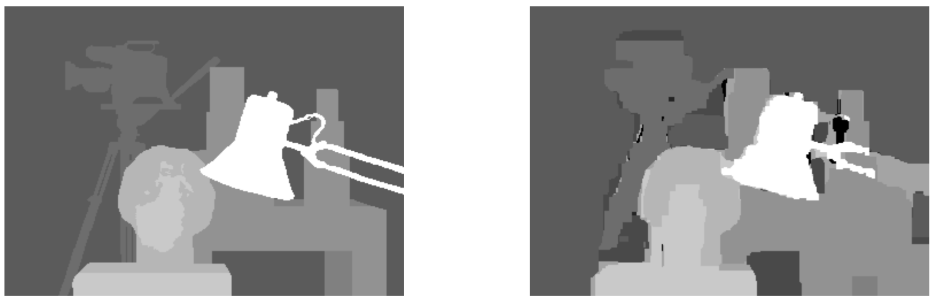

Figure 2: Qualitative results. Left: Ground truth. Right: Results using graph cuts stereo algorithm.

Problem 2 - Horn-Schunck optical flow using PyTorch 27 points

In this problem, you will implement a global optical flow algorithm. To simplify gradient-based optimization, we implement this exercise in PyTorch using its *autograd* package.

The universal datatype to store numerical values in PyTorch is the `torch.Tensor` class. *Tensors* are N-dimensional arrays not unlike Numpy's `ndarray`. While they can have arbitrary dimensions, they are typically constructed with the four dimensions

$$(\texttt{batch\_size} \times \texttt{channels} \times \texttt{height} \times \texttt{width}),$$

which is sometimes referred to as `NCHW`-Memory-Layout. The reason for this lies in PyTorch's common use case for deep learning applications: Here, the first dimension corresponds to the number of data samples, *e.g.* the amount of images that are processed simultaneously. The second dimension represents the number of data channels. For instance, a flow field has two components and, hence, `channels` $= 2$. The dimensions `height` and `width` correspond to the size of the data in vertical and horizontal directions, *i.e.* rows and columns of an image.

Note: Numpy and its associated packages typically assume an `HWC` format; so you should pay particular attention to the dimensions of your constructed arrays/tensors. If not otherwise specified, make sure that implemented functions take 4D tensors as inputs and equally return 4D results. This also holds for scalar outputs which should have shape ($\texttt{batch\_size}, 1, 1, 1$).

Helper functions: In `problem2.py` we already imported the helper functions `flow2rgb`, `rgb2gray`, `read_flo`, and `read_image`. Feel free to make use of these functions.

The test data for this assignment is taken from the Middlebury optical flow database. In a first step, you implement several basic functions that will be needed for the optical flow estimation:

- Implement `numpy2torch(array)` and `torch2numpy(tensor)` that convert 3D Numpy arrays (`HWC`-format) to 3D PyTorch tensors (`CHW`-format) and vice versa.

  2 points

- Implement `load_data()` which loads `frame10.png` and `frame11.png` and converts the images to grayscale. Additionally, load the ground truth optical flow from `flow10.flo` using the provided function `read_flo`. Convert all loaded data to 4D PyTorch tensors.

  2 points

For a quantitative evaluation of optical flow, we use the average endpoint error (EPE) metric. The EPE for a single pixel (i, j) is defined as

$$\mathrm{EPE}(\boldsymbol{u}_{i,j}, \boldsymbol{u}_{i,j}^{\mathrm{GT}}) = \sqrt{\left(u_{i,j} - u_{i,j}^{\mathrm{GT}}\right)^2 + \left(v_{i,j} - v_{i,j}^{\mathrm{GT}}\right)^2}, \tag{3}$$

where $\boldsymbol{u}_{i,j} = (u_{i,j},\, v_{i,j})^T$ is an estimated flow (with horizontal and vertical components u/v) and $\boldsymbol{u}^{\mathrm{GT}} = (u_{i,j}^{\mathrm{GT}},\, v_{i,j}^{\mathrm{GT}})^T$ represents the corresponding ground truth flow.

- Implement `evaluate_flow(flow,flow_gt)` to compute the average endpoint error (AEPE) between the estimated flow and the ground truth flow by averaging the EPE over all pixels.

  1 point

- The ground truth data uses values greater than $1e9$ for invalid pixels. Make sure to not include these pixels in your AEPE calculation.

1 point

To evaluate the brightness constancy for a given flow $(u, v)^T$, it is necessary to warp the second image I_2. Here, a backward warping should be performed, *i.e.* for each pixel (i, j) the corresponding brightness value is obtained by looking it up at position $(i + u(i, j), j + v(i, j))$.

- Write a function `warp_image(im,flow)` that warps an image by the estimated flow. You can use the PyTorch functional `grid_sample` to implement the warping.

4 points

- In function `visualize_warping_practice(im1, im2, flow_gt)`, warp the second image by the ground truth flow to obtain I_2^w. Subsequently, display I_1, I_2^w and the difference between both images in separate subplots within a Figure. What do you observe?

2 points

Now, you implement a simplified form of dense Horn-Schunck optical flow without the coarse-to-fine estimation.

- Implement an optical flow energy function which assumes brightness constancy between corresponding pixels, a pairwise MRF prior and quadratic penalties for both terms. In particular, write the function `energy_hs(im1, im2, flow, lambda_hs)` to compute the energy function as

$$E(u, v) = \int \Big(\big(I(x + u(x, y), y + v(x, y), t + 1) - I(x, y, t)\big)^2 + \lambda_{hs} \cdot \left(\|\nabla u(x, y)\|^2 + \|\nabla(x, y)\|^2\right)\Big) \mathrm{d}x\mathrm{d}y \tag{4}$$

as defined in Lecture 7 (Global Models for Optical Flow) on Slide 54. Note: We do not linearize the brightness constancy assumption here as we will just use *autograd* to minimize Eq. (4). The parameter λ_{hs} should scale the prior term and allows for a trade-off between the two components of the posterior.

4 points

- Implement a flow estimation algorithm `estimate_flow(im1,im2,flow_gt, lambda_hs, learning_rate, num_iter)` performing an iterative minimization of the energy function with parameter $\lambda_{hs} = 0.0015$. For 400 iterations, update the flow estimate with a simple gradient descent step using a learning rate of 20. Here, you should make use of *autograd* to backpropagate the gradient into the variables of interest. In the same function, display the AEPE of your estimate on the command line and create a Figure which visualizes your estimated flow using `flow2rgb`.

4 points

In the next step, we aim to accelerate the optimization and therefore use the second-order optimizer LBFGS.

- Implement `estimate_flow_LBFGS(im1, im2, flow_gt, lambda_hs, learning_rate, num_iter)` that uses the LBFGS function provided in `torch.optim` instead of your manually implemented gradient descent approach. Again, display your flow result and its AEPE.

2 points

As a last step, we aim to apply a coarse-to-fine-strategy. We thus estimate the flow on downscaled versions of the image, upscale the estimated flow and use it as an initialization for the estimation on larger images.

- Using parameter $\lambda_{hs} = 0.0015$, implement `num_level` levels of coarse-to-fine optical flow estimation with an LBFGS optimizer in function `estimate_flow_coarse_to_fine(im1, im2, flow_gt, lambda_hs, learning_rate, num_iter, num_level)`. For each level, evaluate the AEPE and visualize the results upscaled to full resolution.

4 points

- What do you observe if you change the number of levels starting from four levels?

1 points

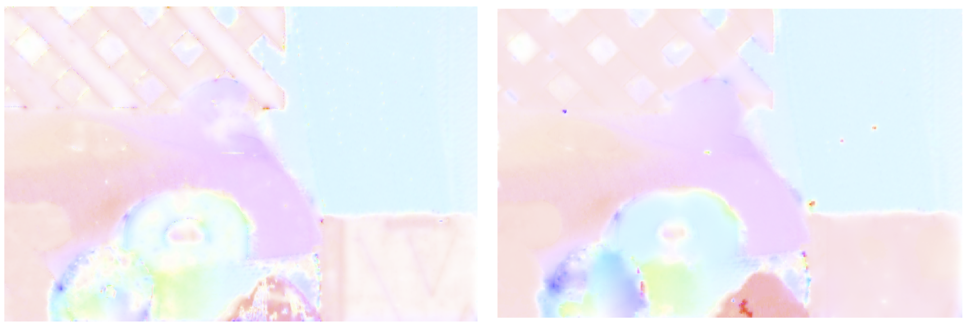

Figure 3: Qualitative results. Left: Results to be expected from a plain gradient descent optimizer. Right: Results using coarse to fine estimation (using LBFGS on each level).