Computer Vision II - Homework Assignment 3

Stefan Roth, Samin Hamidi, Krishnakant Singh
Visual Inference Lab, TU Darmstadt

June 07, 2023

This homework is due on June 21, 2023 at 23:59.
Please read the instructions carefully!

General remarks

Your grade not only depends on the correctness of your answer, but also on clear presentation of your results and a 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. You can get partial credit even if you have not completed the task, but addressed some of its challenges. Hence, please hand in enough information demonstrating your effort: 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* (*e.g.* books other than the course books, papers, websites). Acknowledgements will *not* affect your grade. However, not acknowledging a source you used is a clear violation of academic ethics. Note that both the university and the department take any cases of plagiarism very seriously. For more details, see the department guidelines about plagiarism 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 runs with **Python 3.6 or higher** (and **PyTorch 1.8** or higher in later assignments). In order for us to be able to grade the programming assignments properly, insert your code in the provided function definitions. Feel free to experiment with your code in the main() function. However, make sure that your final submission can execute the code originally provided in the main() function. Additionally, comment your code in sufficient detail, so that it is clear 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. In your final submission, please do not use pop-up windows for visualising intermediate results or ask for user input, unless instructed in the assignment task. Of course, you are welcome, in fact even encouraged, to use visualisation while developing your solution. Please be sure to carefully read the comments in the provided code skeleton, as these provide important details on the function's semantics (*e.g.* what arguments it expects and what results it should return), as well as useful tips on implementation. 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 assignments will be made available over Moodle.

What to hand in

Your hand-in should contain a PDF file for any non-programming ("pen & paper") tasks in the assignment. You are encouraged to typeset your solution, but we will also accept scans or photos of your handwritten solution as long as the image quality does not inhibit its readability. For the programming parts, you must not submit any images of your results; your code should be able to generate these instead. Please hand in your completed version of `.py` scripts provided with the assignment. Make sure your code actually executes and that all functions follow the provided definitions, *i.e.* accept the specified input format and return the instructed output with correct types and dimensions.

Handing in

Please upload your solution files as a single `.zip` or `.tgz` file to the corresponding assignment on Moodle. **Please note that we will not accept file formats other than the ones specified!** Your archive should include your write-up (`.pdf`) as well as your code (`.py`). 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.

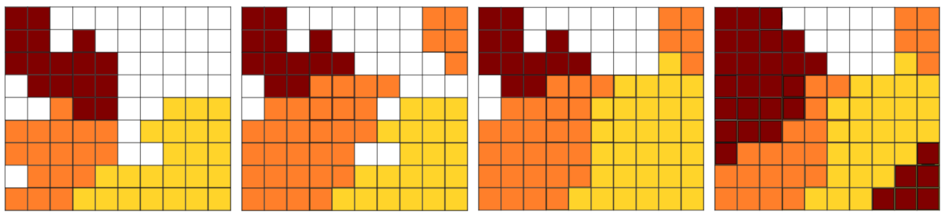

Figure 1: The alpha-expansion algorithm breaks a multi-class labeling problem down into a series of binary sub-problems. At each step, we choose a label α and expand: For each pixel either leave the label as is or replace it with α. Each sub-problem can be solved in a globally optimal way. a) Initial 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 for Image Denoising 12 points

Recall the α-expansion algorithm presented in class. It provides a way of doing inference on a Markov random field (MRF) 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, as shown in Fig. 1. Here, we will implement an approximate version of the α-expansion algorithm, which will be performed step-wise in `problem1.py`. We will assume that images are corrupted by pixelwise independent Gaussian noise with standard deviation σ_N. Hence, for an image of size $H \times W$ we can write an appropriate likelihood model (abandoning some of the constants) as

$$p(\mathbf{y}|\mathbf{x}) \propto \prod_{i,j} \exp\left\{-\frac{1}{2\sigma_N^2}(x_{i,j} - y_{i,j})^2\right\}, \tag{1}$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{H\times W}$ are the original (denoised) and the corrupted image, respectively.

Tasks:

- Implement the function `mrf_denoising_nllh(x, y, sigma_noise)` to compute the elementwise negative denoising *log* likelihood corresponding to Eq. (1).

  2 points

- Implement the function `edges4connected(height, width)` that takes the dimensions of an 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. Hence, we expect your result to follow a row-major layout.

  2 points

- To measure the quality of your denoised image, implement function *peak signal-to-noise ratio* (PSNR) `p = psnr(x_gt, x)` which is defined as

$$\text{PSNR} = 10 * \log_{10}\left(\frac{v_{\text{Max}}^2}{\text{MSE}}\right), \tag{2}$$

where v_{Max} is the maximally possible pixel intensity value, and

$$\text{MSE} = \frac{1}{HW} \sum_{i,j} \left(X_{gt}(i,j) - X(i,j) \right)^2 \tag{3}$$

is the mean squared error between the original (`x_gt`) and the denoised image (`x`).

2 points

- Implement the α-expansion algorithm in

  `alpha_expansion(y, x0, edges, candidate_pixel_values,` σ_N`,` λ`)`

  where `y` is the given noisy image, `edges` encode the MRF neighborhood, `x0` is the initial estimated denoised image, `candidate_pixel_values` contain a list of pixel values $\{0, ..., 254\}$ to be considered in the optimization, σ_N is the assumed noise level for the image, and λ is the regularization parameter for the MRF. The function should return an estimate of the original (denoised) image `x`.

  In this assignment, we will work with the noisy image `la-noisy.png` as `y`. You will need to experimentally determine the values of λ and σ_N that you find yield a good denoising result (in terms of PSNR) and return these values in functions `my_lmbda` and `my_sigma`, respectively. You may start with $\lambda = 5$ and $\sigma_N = 5$.

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

$$E_{pq}(\alpha, \beta) = \begin{cases} 0, & \text{if } \alpha = \beta \\ \lambda > 0, & \text{otherwise.} \end{cases} \tag{4}$$

  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

Remarks:

- You will need `maxflow` package for `graphcut` to work. Install it with

  ```
  pip install PyMaxFlow.
  ```

- We strongly encourage to study the contents of `gco.py`, especially the provided example usage of `graphcut`.
- The candidate pixel values (*i.e.* "labels") are integer values.
- We will evaluate `alpha_expansion` based on the solution quality in terms of PSNR. As a sanity check, make sure that the PSNR of your denoised image w.r.t. the ground truth is *at least higher* than that of the noisy image `y`.

Problem 2 - Horn-Schunck optical flow using PyTorch **20 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. For detailed instructions on PyTorch, you may refer to the PyTorch intro provided in Moodle.

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.

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(im1_filename, im2_filename, flo_filename)`

  which loads `im1_filename` and `im2_filename` and converts the images to grayscale. Additionally, load the ground truth optical flow from `flo_filename` 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

$$\mathrm{EPE}(\boldsymbol{u}_{i,j}, \boldsymbol{u}_{i,j}^{\mathrm{GT}}) = \sqrt{\left(u_{i,j} - u_{i,j}^{\mathrm{GT}}\right)^2 + \left(v_{i,j} - v_{i,j}^{\mathrm{GT}}\right)^2}, \tag{5}$$

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

2 points

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 the 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

$$E(u, v) = \int \Big(\big(I(x + u(x, y), y + v(x, y), t + 1) - I(x, y, t) \big)^2 + \lambda_{hs} \cdot \big(\|\nabla u(x, y)\|^2 + \|\nabla v(x, y)\|^2 \big) \Big) \mathrm{d}x \mathrm{d}y \tag{6}$$

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. (6). 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 $\lambda_{hs} = 0.002$. For 500 iterations, update the flow estimate with a simple gradient descent step using a learning rate of 18. Here, you should make use of *autograd* to backpropagate the gradient into the variables of interest. Initialize each entry of the estimated flow with zero. In the same function, display the AEPE of your estimate before and after optimization on the command line and create a Figure which visualizes your estimated flow using `flow2rgb()`.

4 points

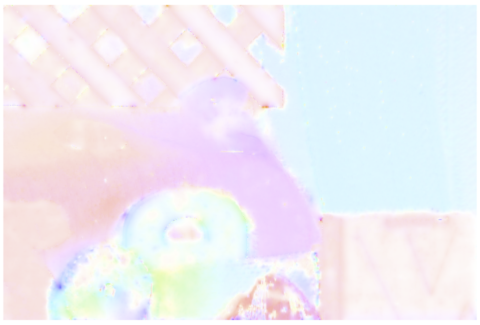

Figure 2: Qualitative results. Results to be expected from a plain gradient descent optimizer.