Computer Vision II - Homework Assignment 4

Stefan Roth, Xiang Chen, Jochen Gast, Faraz Saeedan, Nikita Araslanov
Visual Inference Lab, TU Darmstadt

July 8, 2019

This homework is due on August 5th, 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.

Problem 1 – Iterative graph cuts for image segmentation 27 points

In this problem we apply graph cuts under a contrast-sensitive Potts model for image segmentation, and extend it for multi-label segmentation via α-expansion similar to the stereo algorithm of the last assignment. The overall approach is based on a simplified version of the GrabCut[1] algorithm for binary foreground-background segmentation.

Binary, interactive GrabCut. Here, we briefly review the original version: GrabCut, an iterative graph cuts algorithm, expects color images as input as well as a bounding box annotation drawn around the object of interest. The algorithm then proceeds to refine a binary labeling for each pixel either classifying it as foreground or background. The unary potentials of the underlying graph cuts algorithm are essentially obtained from learning color statistics of foreground and background pixels, respectively. The pairwise potentials are based on a contrast-sensitive Potts model encouraging labels to respect discontinuities in the color image. Pixels outside the bounding box are fixed to be in the background. With I, Θ and S denoting the image, the parameters of the color distribution and the segmentation, respectively, the segmentation energy for pixels inside the bounding box is given by

$$E(S \mid I, \Theta) = E_d(S \mid I, \Theta) + \lambda \, E_s(S \mid I), \tag{1}$$

where E_d and E_s are the data (unary) term and the smoothness (pairwise) term, respectively, and λ is a trade-off parameter. The data term

$$E_d(S \mid I, \Theta) = \sum_{k=1}^{K} - \log \mathrm{GMM}(I_k \mid \theta_{S_k}) \tag{2}$$

measures how well each pixel fits to the color distribution of foreground or background pixels (depending on the segmentation). Color distributions are represented as Gaussian Mixture Models with K components. The spatial smoothness term favors segmentations that align with the image boundaries by using contrast sensitive Potts potentials on a 4-connected MRF:

$$E_s(S \mid I) = \sum_{(i,j) \in \mathcal{N}} \exp(-\beta \|I_i - I_j\|^2) \cdot \delta_{S_i \neq S_j}. \tag{3}$$

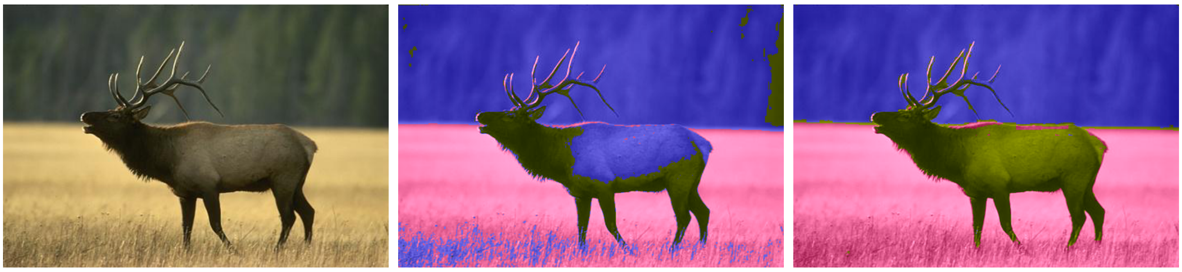
The iterative algorithm then proceeds by initializing the color distributions from user annotation, *e.g.* a bounding box, and then iterating between minimizing the segmentation energy and re-fitting the color distributions.

Multi-label, unsupervised GrabCut: Note that the approach above falls into the category of semi-supervised (interactive) approaches, since it requires bounding boxes as input. Also it allows for binary segmentation only. We will make two simple extensions:

1. First, we want to provide the initial segmentation to the GrabCut algorithm based on an unsupervised initialization. For many input images, a simple clustering algorithm will suffice as initialization.

2. Second, we want to segment an image into more than just two segments. To that end, we can apply α-expansion to the binary problem, in order to find solutions with multiple labels.

[1] C. Rother, V. Kolmogorov, and A. Blake. "GrabCut": Interactive foreground extraction using iterated graph cuts. ACM Transactions on Graphics (SIGGRAPH), August 2004.



(a) Input (b) Initialization (c) Final

Figure 1: Iterative segmentation into three segments. (a) Input image. (b) Initial clustering of color statistics. (c) Final result of iterative graph cuts refinement.

We have provided you with an initial script `problem1` outlining the necessary steps.

Tasks:

- As the first step, we want to find an adequate initialization for the GrabCut algorithm. To that end, please use an algorithm from the `scipy.cluster.vq` package to find `num_segments` initial clusters by means of the color statistics in the input image:

  ```
  cluster_coarse_color_statistics(im, num_segments)
  ```

  Your output should be an integer label mask of the same spatial dimensions as the input image containing labels in $\{0, \texttt{num\_segments} - 1\}$.

  4 points

- Implement a function super-imposing labels on top of a color image, *c.f.* Fig. 1:

  ```
  label2color(im, label)
  ```

  To that end, *(1)* convert the color image into a different color space allowing for separate treatment of color and texture. *(2)* set the texture from the given color image. *(3)* set color channels (*e.g.*, *hue*) by means of the labels. Subsequently, assemble the channels in the alternative color space and convert it to back to RGB to form a color-coded representation. For the color-coding, please discretize an existing color map from `matplotlib.cm` by means of the given labels.

  3 points

- We already found the edges for the 4-connected Potts model for you. However, we still need the contrast sensitive weights to implement Eq. (3). Please compute the weights in:

  ```
  contrast_weight(im, edges)
  ```

  Set the missing parameter β according to

  $$\beta = \left(\frac{1}{|\texttt{Ed}|} \sum_{(i,j) \in \texttt{Ed}} \|I_i - I_j\|_2^2 \right)^{-1} ,$$

  where $|\texttt{Ed}|$ is the number of edges in the Potts model.

  3 points

We will now implement some helper functions:

- For the underlying graph cut algorithm, implement the function

  ```
  make_pairwise(lmbda, edges, cweights, num_sites)
  ```

  which constructs a pairwise capacity matrix from the contrast-sensitive Potts model. Note that this step is similar to the last assignment, however, we apply additional weights extracted from color differences.

  2 points

- For the unary terms we need to fit and evaluate Gaussian mixture models for the color statistics of the distinct segments. To that end, implement

  ```
  negative_logprob_gmm(im, labels, gmm_components, num_segments)
  ```

  which fits a Gaussian mixture model to the pixels assigned to a segment, and subsequently evaluates the corresponding negative log likelihoods of *all* pixels. Here, `gmm_components` is the number of components to use for the mixture model. Note that you need to fit a Gaussian mixture model for every segment, respectively.

  3 points

We now have the ingredients to implement our multi-label segmentation algorithm. Please implement the following two functions, representing the *inner* and *outer* part of the energy minimization:

- Implement a single expansion move, *i.e.* the inner binary update of the algorithm in

  ```
  expand_alpha(alpha, im, label, pairwise, gmm_components, num_segments),
  ```

  where `alpha` is the current label to be expanded, `im` is the input image, `label` is the current assignment of labels, and `pairwise` is the pairwise capacity matrix. Here, you should call `negative_logprob_gmm` to fit and evaluate Gaussian mixture models for all segments, and subsequently construct appropriate unaries from the evaluated (negative) log probabilities. Based on the unaries and pairwise capacities you should update the `label` mask by expanding `alpha` via the graph cuts optimizer provided in `gco.py`.

  5 points

- Implement the outer update of the alpha expansion algorithm in

  ```
  iterated_graphcuts(im, label0, pairwise, gmm_components, num_segments)
  ```

  where `label0` is the initial labeling obtained from unsupervised clustering. Here you should repeatedly call `expand_alpha` to perform updates of the current label map. Please find an appropriate configuration (iterations and order of labels) to yield a satisfying labeling.

  4 points

- Find an appropriate setting of `gmm_components` and `lmbda` to achieve good results for the given elk image *c.f.* Fig. 1. Please write intermediate labelings of your iterations into a binary folder `bin`. When you submit the assignment, please include your initial labeling `init.png`, an intermediate labeling `intermediate.png`, and your final estimate `final.png`. Do not upload your whole `binary` folder!

  3 points

Problem 2 – Fooling CNNs 26 points

Convolutional Neural Networks (CNNs) are a very powerful alternative to classic image segmentation methods due to their ability to learn features from data directly. However, despite their performance, they are easily fooled by applying comparatively simple changes to the inputs. In this problem, we will have a look at a specific model called DeepLabV3, apply it to the Pascal VOC 2007 segmentation dataset, and exploit energy minimization to find simple examples to fool the network.

Dataset and DataLoader. To start, you need to make the dataset available in your environment.

- Please download the Pascal VOC 2007 segmentation dataset from `http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtrainval_06-Nov-2007.tar` and extract it to any location on your machine.
- Set an environment variable `VOC2007_HOME` pointing to the `../VOCdevkit/VOC2007` folder. This variable should be accessible in your Python environment.

As usual we provided you with a script `problem2.py` to get you started. We now proceed to implement the dataset and corresponding dataloader:

- Please implement the dataset class

  ```
  VOC2007Dataset(root, train, num_examples)
  ```

  where `root` points to the root folder of the dataset, `train` indicates whether we are looking for the training or validation dataset, and `num_examples` restricts the size of the dataset. The dataset class implements access to an example in `__getitem__(self, index)` and access to the overall number of examples in `__len__(self)`. In the constructor, you will need to read all required image and segmentation filenames; images are located in `root/JPEGImages`, segmentations are located in `root/SegmentationClass`. Depending on the `train` flag you will need to read filenames for the corresponding split from `root/ImageSets/Segmentation/train.txt` or `root/ImageSets/Segmentation/valid.txt`. Here, `num_examples` should essentially limit the number of rows to use from the split file. You will need to find a way to convert the segmentation images to raw label ids (please use the constant list `VOC_LABEL2COLOR` to convert colors to corresponding labels and vice versa).

  Note that we will consider a dataset example to be a Python dictionary with the keys `im` and `gt` for image and ground truth segmentation, respectively. These should be 3D torch tensors (`torch.float32` / `torch.long`) in the `CHW` layout.

  5 points

- Create a corresponding `DataLoader` for your dataset in

  ```
  create_loader(dataset, batch_size, shuffle, num_workers)
  ```

  where `shuffle` indicates whether data is loaded in random order and `num_workers` is the number of CPU workers.

  1 point

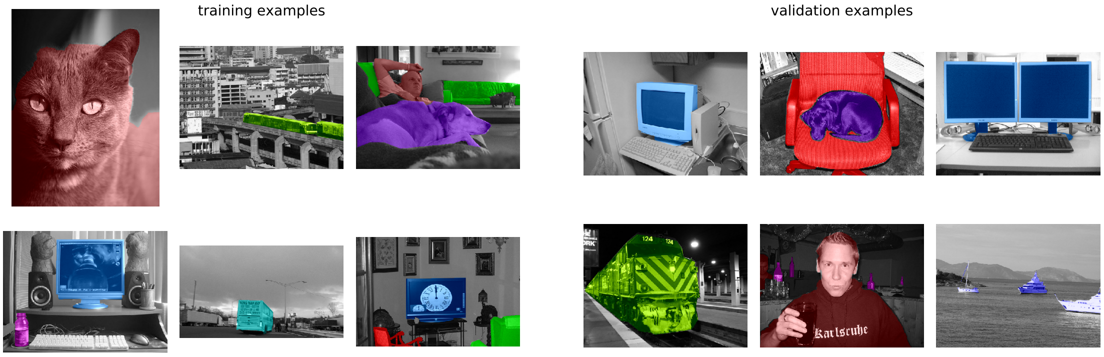


Figure 2: Visualizing dataset examples.

Now we will inspect whether the dataloading pipeline works:

- Similar to the label visualization of `problem1`, please implement

  ```
  voc_label2color(np_image, np_label)
  ```

  to super-impose labels on a given image using the colors defined in `VOC_LABEL2COLOR`.

  2 points

- Subsequently, implement

  ```
  show_dataset_examples(loader, grid_height, grid_width, title)
  ```

  which uses the `loader` to sequentially load (`grid_height`×`grid_width`) examples, and visualizes them in a `grid_height`×`grid_width` grid in a standalone figure with `title`, *c.f*. Fig. 2. Here, you should make use of `voc_label2color(np_image, np_label)`.

  2 points

Inference with DeeplabV3. In the next step, we will use DeepLabV3 to perform inference on the validation images. To that end, we need a couple of helpers:

- As DeepLabV3 expects standardized inputs, please implement

  ```
  standardize_input(input_tensor)
  ```

  which standardizes `NCHW`-Tensors by the image statistics given in `VOC_STATISTICS`.

  1 point

- Implement the forward pass of (standardized) inputs in

  ```
  run_forward_pass(normalized, model)
  ```

  Note that you should return both the model's output activations `acts` as well as the final prediction labels `prediction`. Don't forget to put your model into evaluation mode!

  1 point

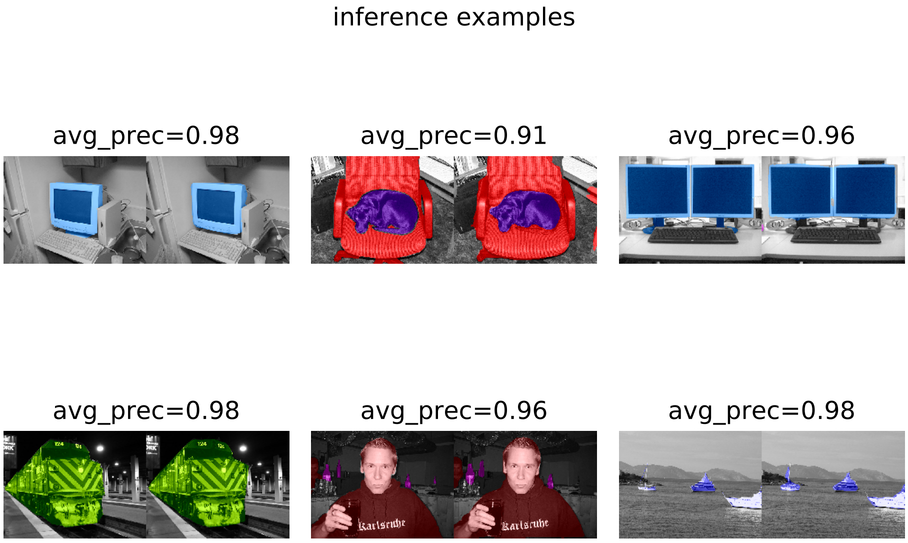


Figure 3: Visualizing inference with DeepLabV3.

Now, we implement the inference using a trained model from the `torchvision` package.

- Implement

  ```
  show_inference_examples(loader, model, grid_height, grid_width, title)
  ```

  which uses the given `model` to perform inference on (`grid_height`×`grid_width`) images obtained from the `loader` and visualizes them in a `grid_height`×`grid_width` grid in a figure with `title`. Here, you should make use of `run_forward_pass` and `voc_label2color`. You can visualize the ground truth and the prediction next to each other, *c.f.* Fig. 3. Feel free to play around with the training loader to make inference for random examples.

  3 points

- For evaluation purposes, please implement

  ```
  average_precision(prediction, gt)
  ```

  which computes the percentage of correct labeled pixels. Please put this performance metric into the figure title of `show_inference_examples`, *c.f.* Fig. 3.

  1 point

Fooling DeepLab3. In order to fool the network, we will look at a specific example image and optimize the input image w.r.t. a false target label. We need a couple of helpers, again:

- To keep things simple, we look for examples consisting of just background and a single other label. Please implement

  ```
  find_unique_example(loader, unique_foreground_label))
  ```

  that sequentially loads examples from `loader` and returns the first found example consisting of just two labels, *i.e.* the background label (`0`) and the given `unique_foreground_label`.

  2 points

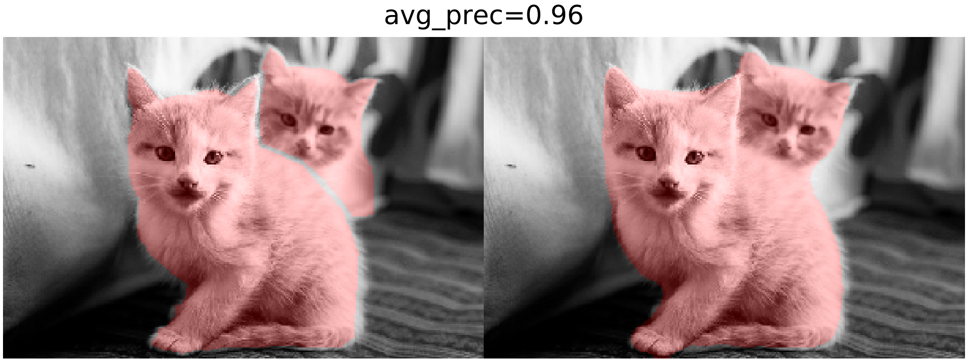


Figure 4: Visualizing unique examples (before fooling).

- Also visualize the found example in

```
show_unique_example(example_dict, model)
```

  showing prediction and average precision for the example (before being fooled), *c.f.* Fig. 4.

2 points

Fooling Optimization. We finally proceed to fool the network. To that end, we consider the input image (instead of the model parameters) as the parameters we want to optimize. The process is as follows:

1. Given a given example we convert all pixels with a `src_label` to a `target_label`. We will refer to resulting label mask as `fake_gt`. For instance, we may convert the cat label in the ground truth mask of Fig. 4 to a dog label.

2. We activate gradient tracking for the input cat image `input_tensor` by enabling its `requires_grad` flag.

3. We run the standard pipeline consisting of standardization and model forward pass (in evaluation mode) to obtain output `activations`.

4. We then apply a `cross_entropy` to compute the loss of the `activations` w.r.t. the `fake_gt` mask.

5. The resulting gradient is given in `input_tensor.grad`; here we set pixels corresponding to background (in the original ground truth) to zero as we only want the foreground label to change.

6. We apply subsequent updates to the input image to minimize the loss w.r.t. the input image. Eventually, the input image should be changed, such that the network will change its prediction from the original label to the target label. Results can be improved by using a second-order optimizer such as LBFGS.

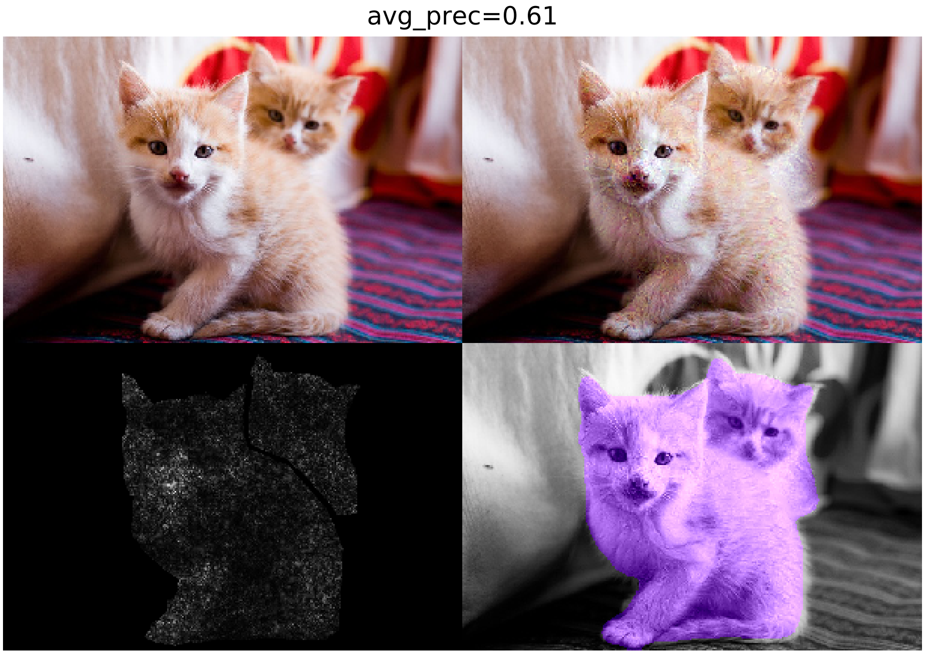


Figure 5: Visualizing the example (after being fooled). Top left: Original input. Top Right: Updated input that fools the network into thinking the cat is a dog. Bottom left: Absolute differences. Bottom right: New prediction. Note how the required changes to the input are perceptually rather small.

Please implement the fooling attack in

```
show_attack(example_dict,model,src_label,target_label,learning_rate,iterations)
```

where `src_label` is the original foreground label, `target_label` is the new target label, and `learning_rate`, and `iteration` are parameters for the optimizer. Please use `LBFGS` as the optimizer. Show your results in a Figure depicting the input before fooling, the input after fooling, the difference between both inputs (*e.g.* L2-Norm between colors), and the new prediction by the DeepLabV3 model, *c.f.* Fig. 5. If you have implemented the algorithm correctly, the new prediction for the foreground should be the target label and the overall average precision should have dropped significantly.

6 points

.