Computer Vision II - Homework Assignment 4

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

June 21, 2023

This homework is due on July 5th, 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.

Python Environment

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

Note: `asgn4.py` provides the code skeleton for this and the next problem set.

Problem 1 – Semantic Segmentation using CNNs — 16 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 Fully Convolutional Network (FCN) and apply it to the Pascal VOC 2007 segmentation dataset.

Dataset and DataLoader. To begin, you need to set up the dataset in your environment.

- Download Pascal VOC 2007[1] segmentation dataset and extract it to any location on your machine.
- Set an environment variable `VOC2007_HOME` pointing to folder

  `<YOUR_PATH>/VOCtrainval_06-Nov-2007/VOCdevkit/VOC2007`

  of the dataset. This variable should be accessible in your Python environment.

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.

  Note: Use `VOC_LABEL2COLOR`, provided in `utils.py`, to convert colors to corresponding labels and vice versa. The color (224, 224, 192) indicates an ambiguous label (*i.e.* neither an object nor the background). Pixels with this label should be excluded in the score calculation (see below).

  Make sure that accessing by index into the instances of `VOC2007Dataset` returns a Python dictionary with keys `im` and `gt` for image and ground truth segmentation, respectively. These should be 3D torch tensors (`torch.float32` for `im` and `torch.long` or `torch.int64` for `gt`) in the `CHW` layout.

  5 points

[1] Use course credentials for access.

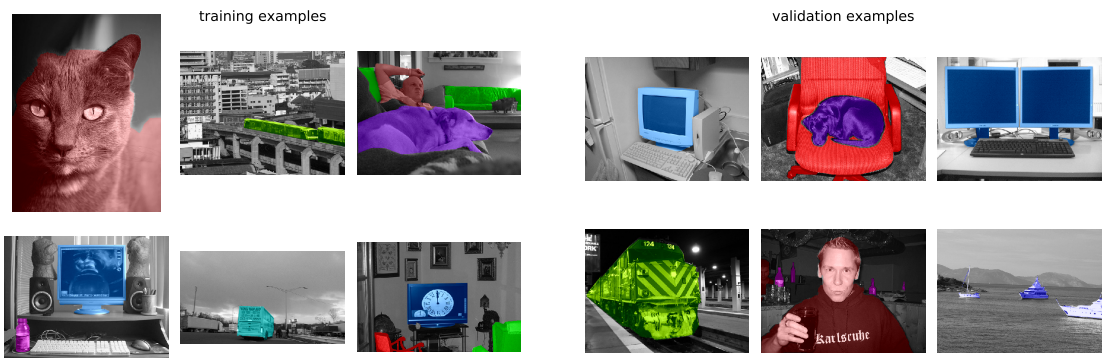

Figure 1: Visualizing dataset examples.

- 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

Now we will inspect whether the dataloading pipeline works:

- Similar to the label visualization of Fig. 1, please implement

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

  to super-impose labels on a given image using the colors defined in `VOC_LABEL2COLOR`. To that end: (1) Convert the color image into a different color space allowing for separate treatment of color and texture (e.g. YCbCr). (2) Set the texture from the given color image. (3) Set color channels (e.g., hue) depending on the labels. Subsequently, assemble the channels in the alternative color space and convert it to back to RGB to form a color-coded representation.

  Hint: You may find `skimage.color` useful for this task.

  2 points

- Next, 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. 1. Here, you should make use of `voc_label2color(np_image, np_label)`.

  2 points

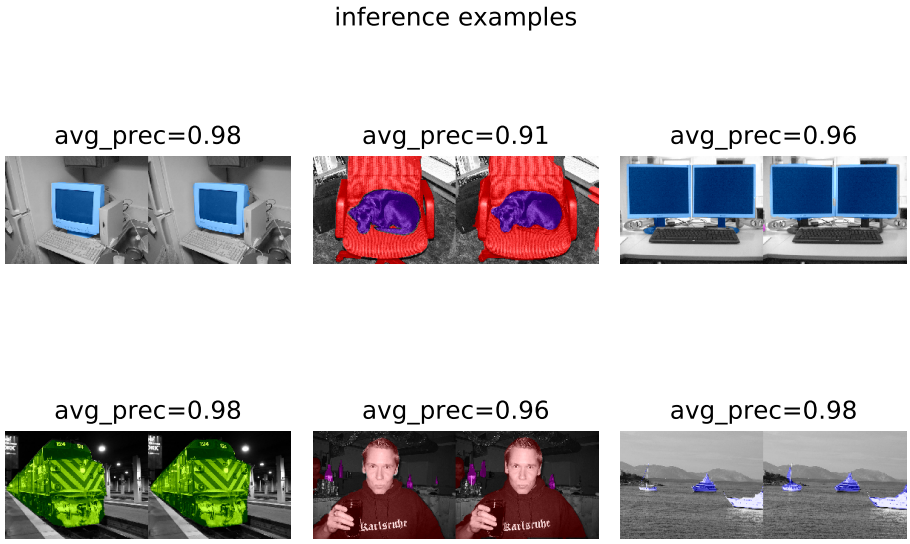

Figure 2: Visualizing inference with FCN.

Inference with FCN. In the next step, we will use FCN to perform inference on the validation images.

- As FCN 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`. `acts` contains the models output (not the auxiliary output). `prediction` contains the class that is assigned to every pixel based on `acts`. Do not forget to put your model into evaluation mode![2]

  1 point

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, as illustrated

[2] See https://stackoverflow.com/questions/60018578/what-does-model-eval-do-in-pytorch if you hear about it for the first time.

by Fig. 2. Here, the numbers need not be replicated exactly and serve only for illustration purposes. 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 correctly labeled pixels. Please put this performance metric into the figure title of `show_inference_examples`, *c.f.* Fig. 2.

  1 point

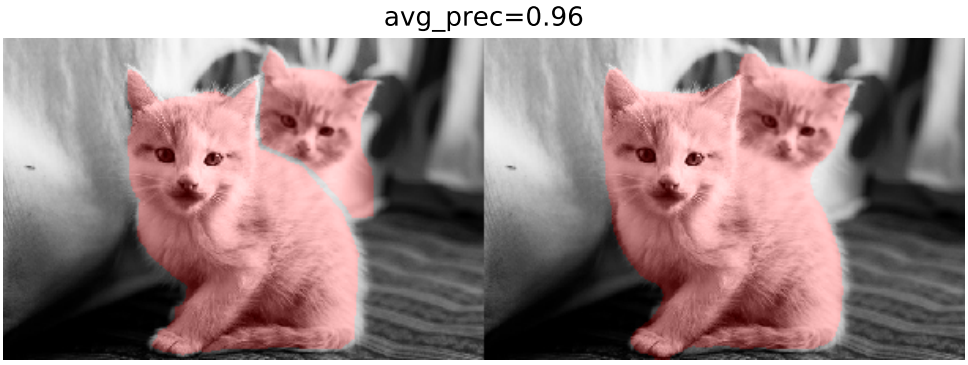

Figure 3: Visualizing unique examples (before fooling).

Problem 2 (Bonus) – Fooling CNNs — 10 points

In this bonus assignment we would like to take the model from the past section and try to exploit energy minimization to find simple examples to fool the network.

Fooling FCN. 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

- 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. 3.

  2 points

Fooling Optimization. We finally proceed to fool the network and will implement

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

The main idea is to consider the *pixel values* as the parameters we want to optimize, not the *network* parameters. The process is as follows:

1. Given an image example we convert all pixels with a `src_label` to a `target_label`. We will refer to resulting label mask as `fake_gt`. For example, in Fig. 3 we convert the cat label in the ground truth mask to a dog label.

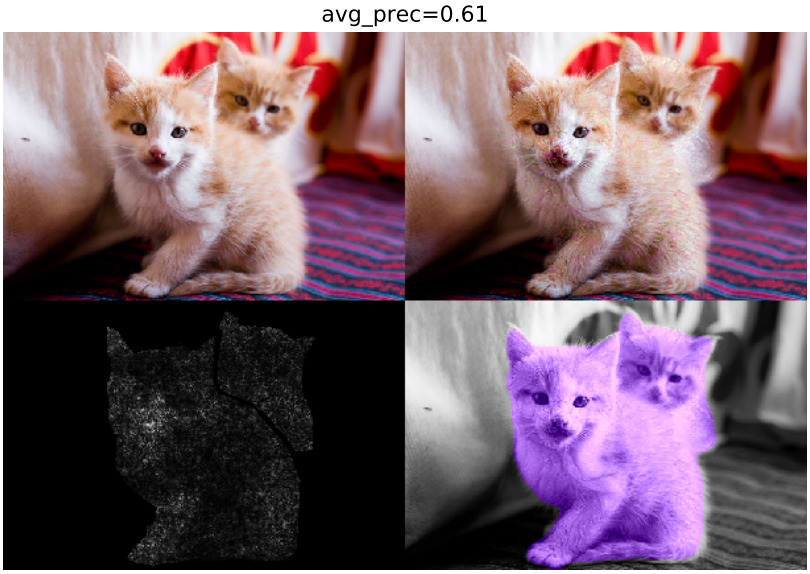

Figure 4: Visualizing the example (after being fooled). Top left: Original input. Top Right: Updated input that fools the network into thinking the cat is a dog. Bottom left: Absolute differences. Bottom right: New prediction. Note how the required changes to the input are perceptually rather small.

2. We enable the gradient tracking for an input image (prior to the forward pass) using `requires_grad` flag.
3. We run the standard pipeline consisting of standardization and model forward pass (in evaluation mode) to obtain the predictions.
4. We then apply a `cross_entropy` to compute the loss of the predictions w.r.t. the `fake_gt` mask.
5. The resulting gradient is accessible through `.grad` property of the input tensor. Here we will set pixels corresponding to background (in the original ground truth) to zero as we only want the foreground label to change.
6. We update the input image to minimize the loss. Eventually, the input image will change such that the network will alter its prediction from the original label to the target label. Please use `L-BFGS` as the optimizer (available in `torch.optim`). The learning rate and the number of optimiser iterations specify the arguments `learning_rate` and `iteration`, respectively.

Visualise your results by depicting (1) the input before fooling, (2) the input after fooling, (3) the difference between both inputs (*e.g.* L2-Norm between colors), and (4) the new prediction by the FCN model, as demonstrated in Fig. 4. If you implemented the algorithm correctly, the new prediction for the foreground would be the target label, and the average precision would drop significantly as a result.

6 points