Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mathematical Foundations of Computer Graphics and Vision

EXERCISE 6 - DEEP LEARNING

Handout date: 09.05.2023
Submission deadline: 30.05.2023, 23:59

General Rules

Plagiarism note. Copying code (either from other students or from external sources) is strictly prohibited! We will be using automatic anti-plagiarism tools, and any violation of this rule will lead to expulsion from the class.

Late submissions up to 24h will be accepted with a penalty, except for extensions in case of serious illness or emergency. In that case please notify the assistant and provide a relevant medical certificate.

Software. All exercises of this course use Python. See the exercise session slides and this document for hints or specific functions that could be useful for your implementation.

What to hand in. Upload your solution in a .zip file on Moodle. The file must be called "MATHFOUND23-*-firstname-familyname.zip" (replace * with the assignment number). The .zip file MUST contain a single folder called "MATHFOUND23-*-firstname-familyname" with the following data inside:

- A folder named "code" containing your code
- A PDF README / Report file containing a description of what you've implemented and instructions for running it, as well as explanations/comments on your results.
- Screenshots of all your results with associated descriptions in the README file.

Grading. This homework is 8.3% of your final grade. Your submission will be graded according to the quality of the images produced by your program, the conformance of your program to the expected behaviour of the assignment, and your understanding of the underlying techniques used in the assignment. The submitted code must produce exactly the same images included in your submission (up to randomness).

Introduction

In this exercise you will implement and train a basic fully convolutional neural network for upscaling input images to 2x resolution using `pytorch`. Before giving the details of each of the tasks you will have to implement, this document first starts by giving a few pointers on how to work with the Euler cluster.

Working with the Euler cluster

While it is perfectly fine to use your own machine if you like, you are granted access to the Euler cluster GPU share during the time of this exercise. Here we provide you with some of the commands that you will need to use. It is recommended to have a look at the official documentation[1] for more information and tricks on how to use the cluster.

Connect to the login node. You can connect to the login node via SSH using your `nethz` login

```
ssh {nethz}@euler.ethz.ch
```

We highly recommend you to follow the official guide and setup SSH keys[2].

Load the modules. Once logged in, you need to setup your environment by loading the proper version of `python` and support for the GPU. You can do this using

```
env2lmod  # switching to the new software stack
module load gcc/6.3.0 python_gpu/3.8.5
```

Don't forget to re-load those modules when you connect to the login node and use them.

Schedule a job. To schedule a job on one of the node of the cluster, you can use the `sbatch` command with the following syntax

```
sbatch --time HH::MM::SS -n N -G 1 python path/to/your/script.py
```

where `HH:MM:SS` is the number of hours (`HH`), minutes (`MM`) and seconds (`SS`) during which your code will be running and `N` is the number of cores required by your job. Note that since the cluster is shared by many users, if you request a job which would run for a long time or which requires a lot of cores, this job will likely need a lot of time before it starts.

Working in a virtual environment. We recommend using a virtual environment. If you have never used one, using `virtualenv` is a good choice. In a nutshell, it creates an isolated python environment, that comes in the form of a folder where all your packages are installed, without risking inferring with your system.
To use `virtualenv` on the cluster you need, *only the first time*, to install the package using `pip`. Don't forget to also enable the `python_gpu` module:

[1] `https://scicomp.ethz.ch/wiki/Getting_started_with_clusters`
[2] `https://scicomp.ethz.ch/wiki/Getting_started_with_clusters#SSH_keys`

```
module load python_gpu/3.8.5
pip install --user virtualenv
```

You can then create a virtual environment for the homework. You can do so in your home folder by running

```
virtualenv ~/mathfound23_ex6_env
```

Then, whenever you want to activate it, run

```
source ~/mathfound23_ex6_env/bin/activate
```

Don't forget to (1) load the `python_gpu` module and (2) activate your virtual environment *every time you connect to Euler*.

Install packages. Once you have loaded the `python` module and activated your virtual environment, you can install packages with `pip`. Whenever you schedule a job from the login node, the current software stack will be copied to the node which will run your code.

1. Implement a Super-Resolution Network

As stated above, in this homework you will implement and train a basic neural network for upscaling input images to 2x resolution using `pytorch`.

Our objective is to solve the super-resolution problem where given a low resolution image I_l (Fig. 1(b)), we would like to estimate the corresponding high resolution image I (Fig. 1(a)), such that:

$$I_l = (I * k)_{\downarrow s} \ ,$$

where k is a degradation (blur) kernel. The down-sampling operation $\downarrow_s$ depends on the considered scaling factor s.

We will use a neural network model to predict an estimate I^* (Fig. 1(c)) of the high resolution image from the low resolution image I_l

$$I^* = \mathcal{F}_g(I_l|\lambda_g).$$

where λ_g are the parameters of the model. Training the generator$\mathcal{F}_g$ can be formally expressed as

$$\lambda_g^* = \arg\min_{\lambda_g} \ \mathbb{E}_{I\sim p_I} \ \Big[\mathcal{L}\big(I, \mathcal{F}_g(I_l, |\lambda_g)\big)\Big] \ .$$

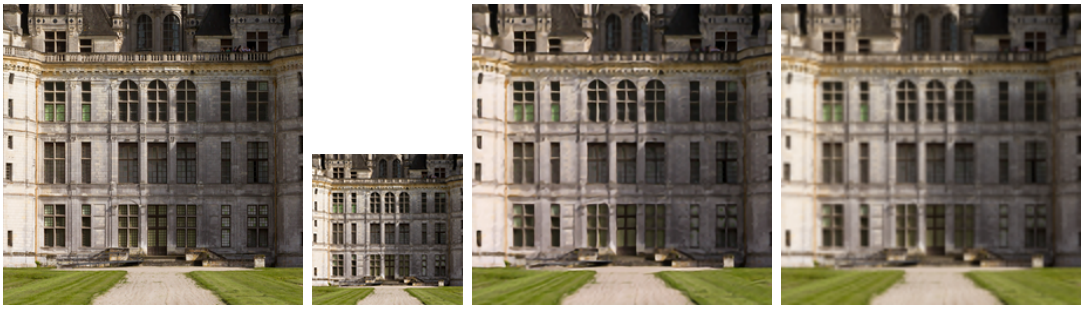
where p_I is the distribution of natural image that we will approximate using the provided dataset and $\mathcal{L}$ is a loss function (for example $L1$ or MSE).

1.1. Task 1 - Datasets, Preprocessing and Data loading.

For this problem you have received a dataset that contains 2 folders: `train` and `eval`. The first will be used for training the model while the second is used for evaluation.

You might have noticed that the dataset only contains images of one size. This is because the down sampled version will be created on the fly. More on this below.

When training models using modern deep learning framework it is often the case that dataloading is handled as separate threads that prepare the inputs into min-batches to be transferred to the



(a) (b) (c) (d)

FIGURE 1. (a) One of the sample image from the evaluation dataset which we want to recover (b) The downscaled image, used as input by the network (c) The output of the network (d) Bicubic interpolation result. Note how the result from the network is sharper.

GPU when needed. In `pytorch` this data preparation process is largely simplified as one simply needs to define a `dataset` class that can then serve as input to a `dataloader` that will handle all the multi-threading aspects and mini-batch creation. The first task of the homework is to create such a class and make sure the `dataloader` can create mini-batches.

We recommend you to have a look at the official introduction tutorial[3] if you have never used `pytorch` before.

As stated above, your first task is to write the `dataset` class:

- Create a `SRDataset` class that can read original resolution images and that can produce their low-resolution counterparts.
- Your class should inherit from `torch.utils.data.Dataset`
- The constructor should take the path of a folder as argument and create a list of all image filenames within that folder
- You should redefine `__len__(self)` method to return the size of the dataset, i.e. the length of the list of filenames
- You should redefine `__getitem__(self, index)` to return the training sample correspond to the requested index. See below for more information on this method.

To read an image given its path, you can use `torchvision.io.read_image(image_path)`. This will directly convert it to a tensor. Convert this image to be `float` between 0 and 1.

As the images are relatively large, a common trick is to select a random crop from the current sample (i.e. the image which will get returned). Use `torchvision.transforms.RandomCrop` to randomly crop a square 64 by 64 patch from the image. This naturally *augments* the dataset by creating an "infinite" number of variations of the input images.

[3] `https://pytorch.org/tutorials/beginner/basics/intro.html`

Another very common augmentation that is performed is to slightly alter the colors of the image before it is returned. Read the documentation of `torchvision.transforms.ColorJitter` and use it to modify the colors of the sample before it is returned. A good starting value for each parameter is 0.2, but feel free to pick any value you think is reasonable.

Read the documentation of `torchvision.transforms.Compose` and use it to compose the above transformations in the right order.

Finally, to create the low resolution version of this cropped patch with altered colors, use `torchvision.transforms.Resize` to *bilinearly downscale* it by a factor of 2.

Once the dataset class is created it is straightforward to use it with a dataloader that handles multiple workes, shuffling and putting the samples into a batch. The code below is a simple way to test your dataset and make sure it is doing the right thing

```
# You create an object from the dataset class:
# For example: train_dataset = SRDataset(data_path)
# Assuming the train_dataset object is properly initialized
# and that the __getitem__ returns a pair of images

train_dataloader = torch.utils.data.dataloader.DataLoader(
    train_dataset,
    batch_size=4,
    shuffle=True,
    num_workers=2,
    drop_last=True,
    pin_memory=True,
)
print(f" * Dataset contains {len(train_dataset)} image(s).")

for _, batch in enumerate(train_dataloader, 0):
    lr_image, hr_image = batch
    torchvision.io.write_png(lr_image[0, ...].mul(255).byte(), "lr_image.png")
    torchvision.io.write_png(hr_image[0, ...].mul(255).byte(), "hr_image.png")
    break # we deliberately break after one batch as this is just a test
```

1.2. **Task 2 - Create the Model.** The following task is to implement the prediction function $\mathcal{F}_g(I_l|\lambda_g)$ as a simple fully convolutional neural network. The architecture is illustrated in Figure 2. The first step is a bilinear upsampling of the low resolution image according to the scale factor (in our case $\times 2$). All convolution layers have kernel size of 3×3. Implement the proposed architecture as a `pytorch` module (inherit from `torch.nn.Module`, see the documentation for which methods you need to implement) named `BasicSRModel`.

- Use `nn.Conv2d`, `nn.LeakyReLU`, and `nn.Sequential` as the building blocks
- Initialize the different layers in the constructor of your model class, a good practice is to set the number of blocks and other hyperparameters using the arguments of the constructor.

- The number of channels for each block should be as follows
  - *first* convolution: input 3 - output 64
  - *i-th* block's convolution: input 64 - output 64
  - *Last* convolution: input 64 - output 3

Once you have created your module class, you can create an instance of that class and check the number of parameters of your model using

```
model = BasicSRModel(...)
num_params = 0
for param in model.parameters():
    num_params += param.numel()
print(num_params)
```

with this model you should have 372803 parameters.

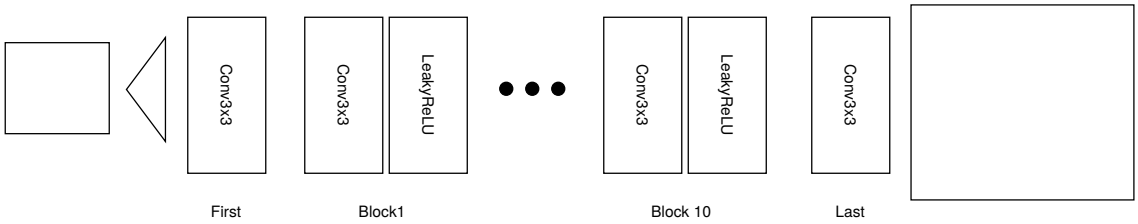


FIGURE 2. Simple Super-resolution model. See Task 2 for details about the different components.

1.3. **Task 3 - Implement the Training Loop.** Now that the model is implemented and the dataset is ready to be used, the remaining piece is the training loop. A typical `pytorch` training loop can be organized as follows:

```
for epoch in range(number_of_epochs):
    for _, batch in enumerate(train_dataloader):

        low_res, high_res = batch

        # reset the gradient
        optimizer.zero_grad()

        # forward pass through the model
        high_res_prediction = model(low_res)

        # compute the loss
        loss = loss_function(high_res_prediction, high_res)
```

```
# backpropagation
loss.backward()

# update the model parameters
optimizer.step()

# log the metrics, images, etc
...
```

Where `train_dataloader` and `model` are defined as in tasks 1 and 2 respectively.

You will also need to specify an `optimizer` and a loss function. We recommend you to use *Adam*

```
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),
                             lr=learning_rate)
```

with a learning rate of `1e-4` (you will need to evaluate the influence of the learning rate later on) Finally, you can setup the loss function using `torch.nn.L1Loss`.

Following the above will create a model and train it on the `CPU`. While this will works, it will be very slow to train. `pytorch` makes it easy to switch to `GPU` training by using the `.to(device)` syntax. Whenever you have a model, tensor, loss function, you can move it to the `GPU` using this. A good practice is to use the following snippet to make your model run on `GPU` only if available.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('Using {} device'.format(device))
```

Do not forget to move the model, the mini-batch (low resolution and high resolution images) and the loss function to the same device when training your model.

Finally, we ask you to

- add logging options to see how the loss evolves during training (`tensorboard`[4] is commonly used for that).
- regularly save the parameters of your model with `torch.save` and the method `.state_dict()`
- provide in your report a plot for the evolution of the loss during training.

For the moment we are only interested in making sure everything works properly. A reasonable training time would be 1 hour for this model and task.

1.4. **Task 4 - Model Evaluation.** Once the parameters of a trained model are saved in a file, we can evaluate how good this model is performing on a test set. To do so we, will write a test functionality which does the following :

- Run the evaluation on the provided *eval* dataset. To set a baseline, do the evaluation with traditional bilinear and bicubic upscaling methods. Do not augment the evaluation images.
- Load a pre-trained model from a file using `torch.load` and the method `.load_state_dict`, run the evaluation with this model.

[4] `https://pytorch.org/docs/stable/tensorboard.html`

- Use PSNR and SSIM as metrics (you can use the implementation available in the package `pytorch-msssim`).

Provide a table in your report showing the average of the entire evaluation dataset for the both metrics.

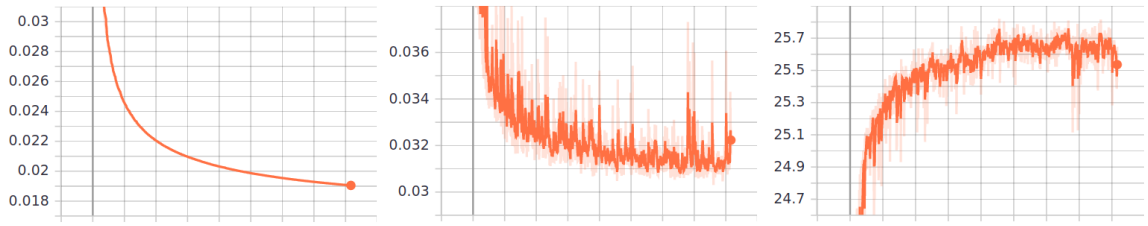
For an image with pixel values ranging from 0 to 1, the PSNR is defined as

$$\mathrm{PSNR}(I^*, I) = -10 \cdot \log_{10}\left(\mathrm{MSE}(I^*, I)\right)$$

You can implement it using `torch.nn.MSELoss()` and `torch.log10`.

Often a validation loss can be tracked during training. This loss is not used by the optimizer but only used to see the progress of the training on a different dataset. Add this functionality to your training loop:

- Add a different dataset/dataloader
- Disable gradient computation using `with torch.no_grad():`
- Log the evolution of the validation loss. In the report, provide the plot with both training (L1) and validation loss for all metrics (L1, PSNR and SSIM) (see Figure 3).



(a) (b) (c)

FIGURE 3. The 3 plots show the metric value (y axis) at each iteration (x axis) for (a) L1 on the training set (b) L1 on the evaluation set (c) PSNR on the evaluation set.

1.5. **Task 5 - Exploration.** Now you should have a properly set up training pipeline and test function. We will take this as a starting point for further investigations and we will explore a few important elements when using and training deep learning models.

Setting our Baseline. Now that everything is set:

- Train the model for a sufficiently long time to consider it converged
- This will create your main training plot
- Evaluate the model

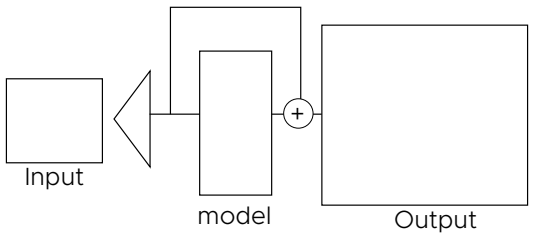


FIGURE 4. Model with Residual Connection

A Different Model. During the lecture you have seen that residual connections are used in many architectures. We propose to use them here as well. Your implementation should follow the model illustration in Figure 4. Essentially your prediction should be added to the bilinearly upsampled image.

- Implement and train the model
- Compare the training plots of this new model with the previous one
- Run a full evaluation and add this new model to your previous evaluation table
- Be careful to run the training for the same number of steps to obtain comparable curves and results

Effect of the Learning Rate. Experiments with the learning rate and its effect on the training

- Using the original model, run the training with the following learning rate values: `1e-2, 1e-3, 1e-5, 1e-6`
- Plot the training curves and comment the results

Different Downscaling During Inference. To better understand what our model is doing, we will consider a different task and apply the trained model to images obtained with a different downsampling method.

- Run the evaluation in the case where the downscaling is using nearest-neighbors and with bicubic interpolation.
- Comment the results

1.6. **Task 6 - Stretched Objective - Adversarial training (optional bonus exercise).** *Bonus points will be added to Exercise 6, maximum grade cannot exceed 100%*

This last bonus task is to implement adversarial training for super-resolution. We invite you to check existing works such as:

- ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks[5] by Wang et al.

[5] `https://openaccess.thecvf.com/content_eccv_2018_workshops/w25/html/Wang_ESRGAN_Enhanced_Super-Resolution_Generative_Adversarial_Networks_ECCVW_2018_paper.html`

- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network[6] by Ledig et al.

Feel free to use the existing implementations of the models (generator and discriminator). You should however implement your own GAN training loop.

- Plot the training curves in this new setting
- Check the results visually and include images in your report
- Run the quantitative evaluation. Comment on the relation between the metrics and your visual evaluation.

Required deliverables

You are required to provide the following deliverables

A. Code for `SRDataset`, `BasicSRModel`, residual model, training and evaluating the models.

B. Training curves for the different learning rate values.

C. Training curves for the residual model.

D. Numerical and visual results for all experiments (traditional upscaling methods, different learning rates, model, inference with different downscaling methods).

E. Optionally, adversarial training code, quantitative and visual results.

Feedback

To help improve this course and the homework, we invite you to provide any comments about the homework, including clarity issues, a rough estimate of the time needed for this exercise, given your previous experience with deep learning tasks. This feedback is optional, voluntary, and doesn't play any role in grading the homework. Feel also free to suggest any interesting application you would have liked to see explored.

[6] `https://openaccess.thecvf.com/content_cvpr_2017/html/Ledig_Photo-Realistic_Single_Image_CVPR_2017_paper.html`