Mini-project for EE-559

Spring 2022

1 Problem statement

The goal of the mini-projects is to implement a Noise2Noise model (Section 7.3 of the course). A Noise2Noise model is an image denoising network trained without a clean reference image. The original paper can be found at https://arxiv.org/abs/1803.04189.

The project has two parts, focusing on two different facets of deep learning. The first one is to build a network that denoises using the PyTorch framework, in particular the `torch.nn` modules and autograd. The second one is to understand and build a framework, its constituent modules, that are the standard building blocks of deep networks without PyTorch's autograd.

1.1 Submission instructions that apply to both projects

We will evaluate your codes with an automated testing suite, so your submission should adhere to the following structure.

```
Proj_SCIPER1_SCIPER2_SCIPER3
├── Miniproject_1
│   ├── __init__.py
│   ├── model.py
│   ├── bestmodel.pth
│   ├── Report_1.pdf
│   └── others
│       ├── otherfile1.py
│       └── otherfile2.py
└── Miniproject_2
    ├── __init__.py
    ├── model.py
    ├── bestmodel.pth
    ├── others
    │   ├── otherfile1.py
    │   └── otherfile2.py
    └── Report_2.pdf
```

Do not submit any IPython notebooks, as they will not be checked. `others` refers to all the codes that you want to include in your submission. You may choose to structure those differently. The immutable ones are the two folders named `Miniproject_1`, `Miniproject_2`, `model.py` in both those folders with the specific structure described above.

A zip file named `Proj_SCIPER1_SCIPER2_SCIPER3.zip` must be uploaded to the Moodle of the course before **Friday May 27th, 2022 23:59PM** . This zip file should uncompress into two directories named `Miniproject_1` and `Miniproject_2`, with the directory structure given above.

When needed, you should add comments to your source codes to facilitate their understanding.

The code must work in the VM provided for the course, in particular, it should not require additional software or libraries.

Exchange of code or report snippets between groups is forbidden. Using code or really anything that is not your original creation without citing it is called plagairism – it is wrong and will be treated very seriously. Every student should have a clear understanding of their group's entire source code and report. This will be checked during the oral presentation.

2 Mini-project 1: Using the standard PyTorch framework

You have, at your disposal, the whole PyTorch framework, with all its powerful modules to implement the denoiser. You are free to explore the various modules available and build any network that you deem necessary. It should be implemented with PyTorch only, in particular, without using other external libraries such as scikit-learn or NumPy.

You should explore various parts of the training pipeline like data augmentation strategies, optimization methods, loss functions, etc.

Training

You have been provided a `train_data.pkl` file with two tensors of the size $50000 \times 3 \times H \times W$. You can load this file with a command like:

```
import torch
noisy_imgs_1, noisy_imgs_2 = torch.load('train_data.pkl')
```

This constitutes the training data and corresponds to 50000 noisy pairs of images. Each of the 50000 pairs provided correspond to downsampled, pixelated images. Train a network that uses these two tensors to denoise i.e., reduce the effects of downsampling on unseen images.

An additional `val_data.pkl` is provided so that you can track your progress. You can load the validation file with a command like:

```
noisy_imgs, clean_imgs = torch.load('val_data.pkl')
```

Your proposed method and network will be evaluated on a different set of images than the ones given here. They will also be of size $3 \times H \times W$, but their noise characteristics may vary.

Evaluation and Submission

The final version of your code should contain a `model.py` that will be imported by the testing pipeline.

This file should contain a class :

```
### For mini-project 1
class Model():
    def __init__(self) -> None:
        ## instantiate model + optimizer + loss function + any other stuff you need
        pass

    def load_pretrained_model(self) -> None:
        ## This loads the parameters saved in bestmodel.pth into the model
        pass

    def train(self, train_input, train_target, num_epochs) -> None:
        #:train_input: tensor of size (N, C, H, W) containing a noisy version of the images.
        #:train_target: tensor of size (N, C, H, W) containing another noisy version of the same images, which only differs from the input by their noise.
        pass

    def predict(self, test_input) -> torch.Tensor:
```

```
            #:test_input: tensor of size (N1, C, H, W) that has to be denoised by the trained
    or the loaded network.
            #: returns a tensor of the size (N1, C, H, W)
            pass
```

The `Model` class will serve as the entry-point to your code for evaluation. Its constructor should not take any inputs. The required interface of the `Model` is shown with comments about what each required function should do.

Using this interface, a model will be trained, and will be evaluated on a test set. Additionally, you should provide your "best" model that you think will perform the best on an unseen test set. You should save this model under "bestmodel.pth" and the model class should provide a function `load_pretrained_model` that loads this model. For that purpose, you can have a look at model serialization methods for PyTorch at https://pytorch.org/docs/stable/notes/serialization.html#saving-and-loading-torch-nn-modules.

Ensure that your code can runs on GPUs, and on CPU only systems, without any changes to the code. The easiest way to make this happen is to define a device as

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

and use this to define the storage location of each of your tensors. For example,

```
    all_ones = torch.ones(2, 3).to(device)
```

will use GPU capabilities if PyTorch detects one, else seamlessly works on the CPU.

Evaluation of the submitted models will be based on the Peak Signal-to-Noise Ratio (PSNR) metric. An extremely simplified implementation of PSNR for a single input image, with pixel values in $[0, 1]$ looks like

```
    def psnr(denoised, ground_truth):
        # Peak Signal to Noise Ratio: denoised and ground_truth have range [0, 1]
        mse = torch.mean((denoised - ground_truth) ** 2)
        return -10 * torch.log10(mse + 10**-8)
```

A network with 8 convolutional layers achieves 24 dB PSNR on the validation data provided (the function above returns PSNR in dB). An extremely simple benchmark achieves 23 dB, so if you are not reaching this you will want to assume that there is a problem with your implementation.

A suggestion: You can consider experimenting with a subset of the training data, and train on the full dataset after you've made your architectural, training choices etc. You can choose services like Google Colab for larger computation power.

We provide a `test.py` that uses a similar interface to the one that we will use for the grading, which trains your model and then evaluates it on a validation set. The training and testing should take at most 10 min on a computer with a small GPU (at most 45 min on a CPU). If your code does not follow the interface or crashes during evaluation, you will not get any points for the code submission.

3 Mini-project 2: Write your own blocks

The goal of the previous section was to exploit PyTorch to build a network. Here you'll build your framework for denoising images without using autograd or `torch.nn` modules. We allow

```
from torch import empty, cat, arange
from torch.nn.functional import fold, unfold
```

and the Python Standard Library[1] *only*. We could consider additional operations if you demonstrate a convincing use-case (please ask on slack or by email to the TAs). Note that you can achieve most elementary tensor operations using the `.foo()` methods or mathematical operators, e.g. instead of `torch.abs(a)` or `torch.sqrt(a)` consider `a.abs()` or `a ** .5`.

[1] https://docs.python.org/3/library/

Your code should work with autograd globally off, which can be achieved with

`torch.set_grad_enabled(False)`

Specifically, you will implement the following blocks that you may have used in the previous problem.

- Convolution layer.
- Transpose convolution layer, or alternatively a combination of Nearest neighbor upsampling + Convolution.
- Upsampling layer, which is usually implemented with transposed convolution, but you can alternatively use a combination of Nearest neighbor upsampling + Convolution for this mini-project.
- ReLU
- Sigmoid
- A container like `torch.nn.Sequential` to put together an arbitrary configuration of modules together.
- Mean Squared Error as a Loss Function
- Stochastic Gradient Descent (SGD) optimizer

With these blocks, build the following network:

```
    Sequential(Conv (stride 2),
               ReLU,
               Conv (stride 2),
               ReLU,
               Upsampling,
               ReLU,
               Upsampling,
               Sigmoid)
```

As specified before, you can implement Upsampling with `Nearest neighbor upsampling + Convolution`. Some suggestions to implement the Convolution layer are given in **??**.

Training data

Reuse the `.pkl` files from the first project.

Suggested structure to implement your modules

You are free to develop any new ideas you want, and grading will reward originality. The suggested simple structure is to define a class

```
    class Module (object) :
        def forward (self, *input) :
            raise NotImplementedError
        def backward (self, *gradwrtoutput):
            raise NotImplementedError
        def param (self) :
            return []
```

and to implement several modules and losses that inherit from it.

Each such module may have tensor parameters, in which case it should also have for each a similarly sized gradient tensor to accumulate the gradients during the backward-pass, and

- `forward` should get for input and returns, a tensor or a tuple of tensors.

- `backward` should get as input a tensor or a tuple of tensors containing the gradient of the loss with respect to the module's output, accumulate the gradient wrt the parameters, and return a tensor or a tuple of tensors containing the gradient of the loss wrt the module's input.
- `param` should return a list of pairs composed of a parameter tensor and a gradient tensor of the same size. This list should be empty for parameterless modules (such as `ReLU`).

Some modules may require additional methods, and some modules may keep track of information from the forward pass to be used in the backward.

Evaluation and Submission

As in the first mini-project, your submission for the second mini-project will be evaluated on its correctness and performance. Additionally, we will test each module's implementation for correctness, using some kind of automated testing, where each module forward and backward passes will be assessed on custom inputs. Thus you must name each of your modules as `Conv2d`, `TransposeConv2d` or `NearestUpsampling`, `ReLU`, `Sigmoid`, `MSE`, `SGD`, `Sequential`.

Similar to `model.py` from mini-project 1, we expect the submission to have train and test functionality. You can use pickle format to save your model.

```
### For mini-project 2
class Model():
    def __init__(self) -> None:
        ## instantiate model + optimizer + loss function + any other stuff you need
        pass

    def load_pretrained_model(self) -> None:
        ## This loads the parameters saved in bestmodel.pth into the model
        pass

    def train(self, train_input, train_target) -> None:
        #:train_input: tensor of size (N, C, H, W) containing a noisy version of the images
.
        #:train_target: tensor of size (N, C, H, W) containing another noisy version of the
 same images, which only differs from the input by their noise.
        pass

    def predict(self, test_input) -> torch.Tensor:
        #:test_input: tensor of size (N1, C, H, W) that has to be denoised by the trained
or the loaded network.
        #: returns a tensor of the size (N1, C, H, W)
        pass
```

Additionally, each of your implemented modules needs to be accessible from the `model.py` file, so importing the different modules with `from model import Conv2d, TransposeConv2d...` should work. The `test.py` contains additional details and some sample testing methods that indicate how your code is going to be evaluated.

Saving the model For the purposes of saving the trained network's state, you can choose to store each of the modules' states in a `pickle` file. In the function `load_pretrained_model`, you can read each of those states (weight matrices), and assign to your modules of the above specified network.

4 Report and Submission details

Each mini-project should include a report of a maximum of 3 pages. The report will be graded based on clarity and completeness. It should include qualitative and quantitive results of the different approaches, explanations of your choices, and ablations justifying these choices. If you are unsure what constitutes a discussion of results, consult the original Noise2Noise paper linked at the beginning of this document for ideas. Comment on your

failed experimental, architectural, implementation choices. Your code will be evaluated based on quality, clarity of implementation, and correctness.

A Appendix: Implementing Convolution

Much of the difficulty of the second mini-project arises from coding up convolution. This optional appendix gives some hints on how to simplify this computation. If you do implement using the hints given here, you can additionally import `fold` and `unfold` from `torch.nn.functional`.

A.1 Convolutions can be seen as linear layers

If $f : \mathbb{R}^{c \times h \times w} \to \mathbb{R}^{c' \times h' \times w'}$ represents the action of a bias-less convolution then it is straightforwardly seen[2] that f satisfies the properties: (1) for any scalar a, $f(ax) = af(x)$, and (2) $f(x + y) = f(x) + f(y)$. For example:

```
import torch

if __name__ == "__main__":
    kernel_size = (2, 2)

    x = torch.randn((1, 3, 32, 32))
    y = torch.randn((1, 3, 32, 32))
    a = torch.randn((1,))

    out_channels = 4

    conv = torch.nn.Conv2d(in_channels=x.shape[1],
                           out_channels=out_channels,
                           kernel_size=kernel_size,
                           bias=False)

    torch.testing.assert_allclose(a * conv(x), conv(a * x))
    torch.testing.assert_allclose(conv(x + y), conv(x) + conv(y))
```

Mathematically, this means that convolution can be thought of as multiplication by a $c'h'w' \times chw$ matrix after some reshaping. Thus, when developing a deep learning framework, if we already know how to apply a linear layer, we can implement a convolution layer by writing the convolution as a linear function, applying our logic for linear operations, and reinterpreting the output as a convolution. This is true of both the forward and backward operations.

Thus, most of the complications in implementing convolutions are due to the bookkeeping of treating convolution as a linear layer. The dimensionality of the output given the input and the settings (kernel size, etc.) of the convolutions can be computed using the formulae given in the `shape` section of https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.

Note that we have not discussed the bias term, this would likewise need to be reshaped and treated correctly. The idea is that if f is a convolution without bias, and $\text{flatten}(f(x)) = W\text{flatten}(x)$ for some $W \in \mathbb{R}^{c'h'w' \times chw}$, then after flattening, a convolution with bias can be written as $W\text{flatten}(x) + b$ for some $b \in \mathbb{R}^{c'h'w'}$.

A.2 Special operations : `fold` and `unfold`

Pytorch offers two useful primitives to work with convolution as matrix operations, `fold` and `unfold`. `unfold` extracts *patches* (proximate pixel values) into columns. This operation is sometimes also called "im2col". For the specific case of a 2×2 kernel, if `test` is

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
```

[2] Conceptualize convolution as a fancy average.

```
             [ 8.,  9., 10., 11.],
             [12., 13., 14., 15.]]])
```

then `torch.nn.Unfold(kernel_size=2)(test)` is

```
tensor([[[ 0.,  1.,  2.,  4.,  5.,  6.,  8.,  9., 10.],
         [ 1.,  2.,  3.,  5.,  6.,  7.,  9., 10., 11.],
         [ 4.,  5.,  6.,  8.,  9., 10., 12., 13., 14.],
         [ 5.,  6.,  7.,  9., 10., 11., 13., 14., 15.]]])
```

where each column contains the values of a 2 x 2 patch. E.g., the first column contains the values 0, 1, 4, and 5 since these are the four values in the upper left of `test`. Not all columns are colored, but you can see how the sliding and reshaping work. This can be used to implement convolution as matrix multiplication as in this example:

```python
import torch

in_channels = 3
out_channels = 4
kernel_size = (2, 3)

conv = torch.nn.Conv2d(in_channels, out_channels, kernel_size)

x = torch.randn((1, in_channels, 32, 32))

# Output of PyTorch convolution
expected = conv(x)

# Output of convolution as a matrix product
unfolded = torch.nn.functional.unfold(x, kernel_size=kernel_size)
wxb = conv.weight.view(out_channels, -1) @ unfolded + conv.bias.view(1, -1, 1)
actual = wxb.view(1, out_channels, x.shape[2] - kernel_size[0] + 1, x.shape[3] - kernel_size[1]
    + 1)

torch.testing.assert_allclose(actual, expected)
```

Note that this code assumes batch_size=1, but your code should work with arbitrary batch sizes. This shows to evaluate the forward pass of a convolution as a linear operation, evaluating the backwards pass of a convolution as the backwards pass of a linear operation follows by analogy.

`fold` is essentially the inverse of `unfold` that contains an additional term related to how many values were included in a patch. For more on the precise relationship, consult the PyTorch documentation at https://pytorch.org/docs/stable/generated/torch.nn.Fold.html.