Deep Learning for image restoration

Deep Learning - Miniproject 2

Ali Benabdallah, João Correia, Gil Tinde
EPFL

Spring 2022

1 Introduction

This mini project's goal was to implement a framework to denoise images using the PyTorch library [1], an open source machine learning framework, without using the all mighty autograd or any module from `torch.nn`. We decided not to reinvent the wheel and tried to emulate the "reference" `torch` behavior as much as possible. Central to computer vision tasks, such as image denoising, is the convolution operation. This mini project's main challenge was thus implementing the inner workings of a 2D convolution module.

2 Implementations

For the implementation, we follow a structure similar to the project description. All of our modules, except for the `SGD` Optimizer, inherit from the `Module` class. The `Module` class has the methods `forward()`, `backward()` and `param()` which are consistent with the project description. Additionally, it has the methods `to()`, `state_dict()`, `load_state_dict()`, `set_weight()` and `set_bias()`. A `Module` can be sent to a `torch.device` by passing this to the `to()` method. The `state_dict()` and `load_state_dict()` methods, only to be used on a `Sequential` instance, respectively return and load the provided model state. `set_weight()` and `set_bias()` are self explanatory.

2.1 Sequential Module

The `Sequential` module can be used to construct a network of several `Module` instances in the order that they are passed to its constructor. Calling the `forward()` method will pass the input through the network in a chained fashion with a `forward()` call on each module, before finally returning the output of the last module.

The `backward()` method takes as input the gradient of the loss with respect to the network's output and, in the reverse order of the modules, calls `backward()` on each module in a chained fashion, propagating the gradient of the loss with the respect to each module's output backward using backpropagation [2].

2.2 Linear Layer

We implemented a `Linear` module even though this wasn't asked. The main reason was to test our understanding of backpropagation in the context of matrix multiplications. This was useful because we ultimately compute convolutions with matrix multiplications as suggested in the project description.

2.3 Convolutional Layer

The `Conv2d` module applies a convolution to an input `tensor` through the `forward()` method. This was implemented using the `unfold()` function from `torch.nn.functional`, followed by a matrix multiplication with the `weight` using `matmul()` then finally adding the `bias`. We then compute the output height and width, apply a `view()`, and return the result.

The `backward()` function required a careful handling of the different dimensions such as the height, width and number of channels, both of the gradient passed to `backward()`, and its output. The gradients of the loss with respect to the module's bias and weight are straightforward and updated accordingly. Determining the gradient of the loss with respect to the module's input ultimately boils down to knowing that the weight and input gradient need to be multiplied together, and that `fold()` can be used to produce a result with the correct shape. The source code is documented with comments of the shapes in all the steps.

We'll also mention that the `Conv2d` module's weights are initialized according to the `torch` documentation. In practice, the weights use the Kaiming He initialization[3], but this would require us to use `torch.nn.init`, so we make do with the classic `uniform_()`.

2.4 Transpose Convolutional Layer

For simplicity, the `TransposeConv2d` module uses the `Conv2d` module under the hood. In `forward()` the input is modified before applying the `Conv2d` module's `forward()` method. We achieved this thanks to the `torch` documentation as well as these animations. In addition to having to zero pad the input, the `stride` argument inserts rows and columns of zeros between the rows and columns of the input `tensor` (*upsampling / zero striding*), and `padding` controls the amount of implicit zero padding, effectively reducing the height and width of the `tensor`. The functions `stride_tensor()` and `pad_tensor()` achieve this and are explained in detail along with examples in the source code.

In `backward()`, we first retrieve the output from the `backward()` call on the underlying `Conv2d` module, then apply `unpad_tensor()` and `unstride_tensor()`. We can perform these operations (that discard specific rows/columns) since the zero elements introduced during the forward pass are not functions of the input (always 0), thus they have no influence on the gradients.

2.5 Activation Functions

The two implemented activation functions are `Sigmoid` and `ReLU`. Having no internal parameters to train, `forward()` simply applies the corresponding function and `backward()` computes the corresponding gradient with respect to the original input.

2.6 Loss Function

The `MSE` module computes the mean squared error loss between the input `prediction` and `target` values in `forward()`. In `backward()`, which doesn't take any arguments since it's assumed to be the final output of the network, the gradient of the computed loss with respect to the original input is returned.

2.7 Optimizer

The `SGD` class is the only one that doesn't inherit from `Module`. It performs the stochastic gradient descent procedure through two methods, similarly to how an optimizer from `torch.optim` would be used. It's instantiated with a learning rate and the parameters (and their gradients) to use during training. `step()` performs a SGD step, subtracting from the parameters, and `zero_grad()` resets the gradients to zero.

3 Results

We drew inspiration from the network suggested in the project description and tried different values for parameters such as kernel size and stride to find the best network structure given the constraints. We fixed the number of input/output channels of the different (transpose)convolutional layers after determining that they were reasonable given the restriction on training time. Additionally: `nb_epochs` $= 8$, `learning_rate` $= 10^{-4}$, `batch_size` $= 10$. The network that was tested with different parameters has the following structure:

```
Sequential(Conv2d (in_c: 3, out_c: 32), ReLU,
           Conv2d (in_c: 32, out_c: 64), ReLU,
           TransposeConv2d (in_c: 64, out_c: 32), ReLU,
           TransposeConv2d (in_c: 32, out_c: 3), Sigmoid)
```

The results of repeating each experiment 10 times are presented in Table 1. The model that performs best on the PSNR metric is the one with a kernel of size 2, a stride of 1, padding of 1 and default dilation (1). Its architecture is displayed in Figure 1.

Model Performances					
Kernel	Stride	Padding	Dilation	PSNR	PSNR*
2	1	0	1	24.80 ± 0.04	24.00 ± 0.03
2	2	0	1	23.86 ± 0.05	23.12 ± 0.04
2	1	0	2	22.72 ± 0.16	22.06 ± 0.12
2	2	0	2	incompatible	shapes
2	1	1	1	$\mathbf{24.89 \pm 0.08}$	$\mathbf{24.07 \pm 0.06}$
2	2	1	1	incompatible	shapes
3	1	0	1	23.80 ± 0.46	23.21 ± 0.37
3	1	1	1	11.99 ± 2.09	11.42 ± 2.06

Table 1: Comparison of different model performances. PSNR is the PSNR as defined in the test script, while PSNR* is the PSNR as defined in the project description.

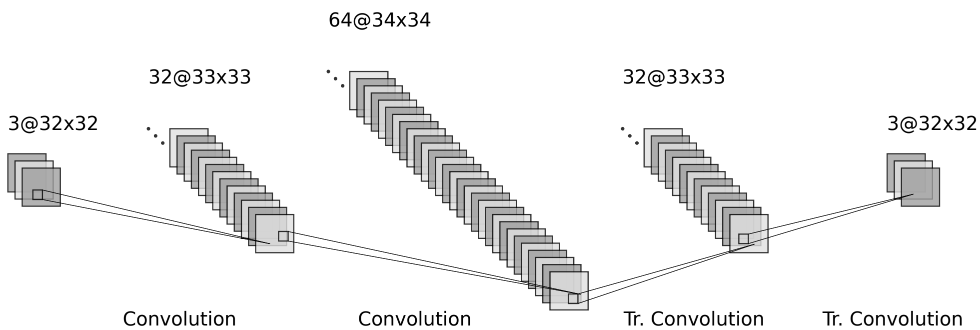
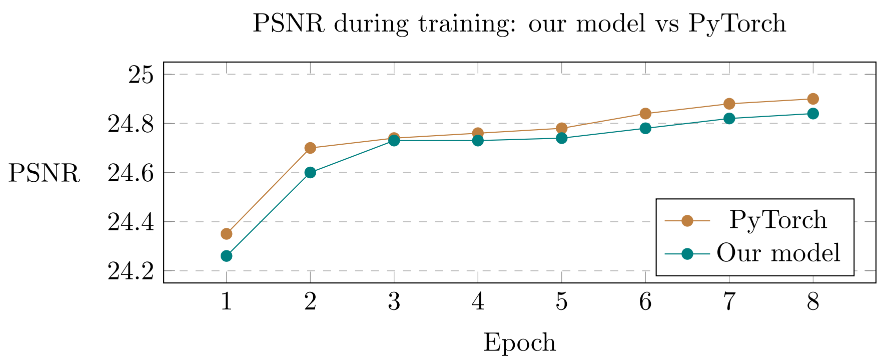


Figure 1: Model Architecture

Training our model on CPU (2018 MacBook Pro i5 8GB) took 37min, vs 9min 30s for the equivalent `torch` model. Training our model on GPU (Tesla P100-16GB) took 2min 10s, vs 58s for the equivalent `torch` model. A comparison of the evolution of the PSNR evaluated on the validation data over the 8 epochs is presented below. Our implementation is able to keep up with PyTorch and one reason for the difference is because of different weight initializations, see Convolutional Layer.



4 Conclusion

Our best performing model performed well on the image denoising task, achieving PSNR values in the desired range while also being comparable to the PSNR values of the equivalent `torch` implementation. Our code is slightly slower than the PyTorch implementation, 4x slower on CPU and 2x slower on GPU, because it has not been optimized in a lower level programming language. Despite this, we are happy with the results obtained and enjoyed the challenge of understanding how convolution is implemented. Possible improvements could be to use different optimizers, such as Adam [4], or try to further optimize the `Conv2d` module.

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[2] H. J. Kelley, "Gradient theory of optimal flight paths," *Ars Journal*, vol. 30, no. 10, pp. 947–954, 1960.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.