EE-559: Deep learning Miniproject 2 Report

Mahammad Ismayilzada, Batuhan Faik Derinbay, Maciej Styczen

May 27, 2022

Abstract

In this report, we outline architecture and design of our implementation of some of the core components of Pytorch framework and show its effectiveness with a denoising model trained using our framework on the provided dataset of noisy image pairs. Particularly, training and validation loss decreases consistently across epochs and our best model achieves a PSNR[1] score of 23.32 dB at the end.

1 Introduction

Pytorch[PGM+19] is a popular high-performance deep learning library in the scientific community. In this report, we show how one can implement the core components of this library with minimal use of the original Pytorch framework. The interface of our components match their counterparts in Pytorch library (with few keyword arguments omitted) and the underlying implementation is inspired by the Pytorch's design and hence, is highly abstract, modular and extensible. In fact, one can port their Pytorch script to use our framework by simply removing all the `torch.nn` references.

2 Framework Design

2.1 Modules

In this project, we implemented the following torch modules: `Linear`, `ReLU`, `Sigmoid`, `Conv2d`, `Upsampling` (implemented using `TransposeConv2d`), `MaxPool2d`, `MSE` (for a loss function) and `Sequential` (for stacking arbitrary number of layers together). Below we briefly describe some implementation details. Code for all available modules can be found in `modules.py`.

2.1.1 Module

Since all modules share a lot of core functionalities, we implement a base `Module` class similar to `torch.nn.Module` that other classes can inherit from. It is initialized with a unique module name and defines `forward` and `backward` methods for forward and backward propagation respectively which should be implemented by the inheriting specific modules. `forward` method takes an input Tensor and outputs an automatically differentiable Tensor (we explain it in more detail below) containing the result of the forward propagation. In this method, we also keep a reference to our input to make it accessible for back propagation later. `backward` method takes a gradient with respect to the output of the module, accumulates parameter gradients and outputs the gradient with respect to module's inputs (for simplicity, we avoid accumulating input gradients which are not needed for our purposes) `Module` class also provides a pytorch-like interface to manage model parameters (e.g. `module.parameters()`) and submodules (e.g. `module.modules()`). Finally, this class also offers methods to export the state of the module as a dictionary (i.e. `module.state_dict()`), to load a particular state dictionary (i.e. `module.load_state_dict()`) and to cast the module to a torch device (i.e. `module.to(device)`). Code for this class can be found in `module.py`.

2.1.2 Convolution Layers

We implemented 2D Convolution and Transpose Convolution layers which support most of the parameters (e.g. `stride`, `padding` etc.) offered by their Pytorch counterparts. We initialize these layers' parameters similar to Pytorch by randomly sampling from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C\_in * \prod_{i=0}^{1} kernel\_size[i]}$ and $k = \frac{groups}{C\_out * \prod_{i=0}^{1} kernel\_size[i]}$ respectively. Since these 2 layers can be thought as reverse operation of each other, our `forward` and `backward` implementations largely mirror each other's opposite methods and use `fold` and `unfold` functions from Pytorch library under the hood.

[1]https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

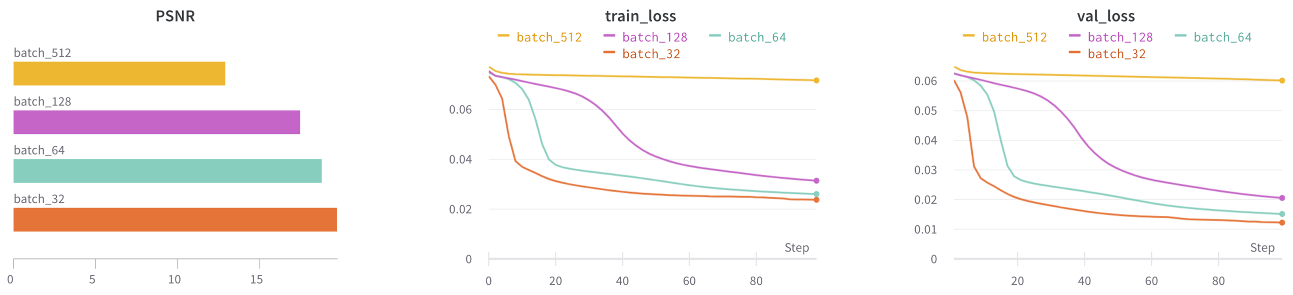

Figure 1: Effect of batch size

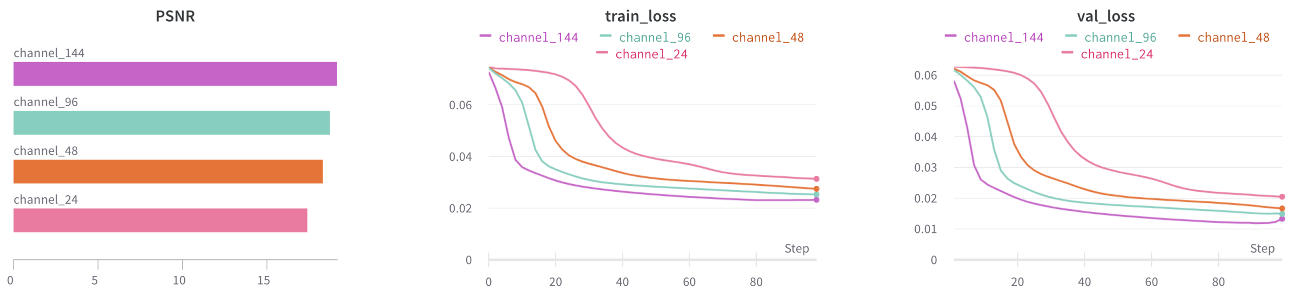

Figure 2: Effect of number of channels

2.2 Optimizers

We defined a base `Optimizer` class that supports an interface similar to Pytorch's. It is decoupled from the modules as in Pytorch and is initialized with a list of parameters to update along with other optimizer specific parameters. As a concrete class, we implemented a simple Stochastic Gradient Descent (SGD) optimizer. Code for this module can be found in `optim.py`.

2.3 Autograd

Since we are not allowed to use the `autograd` library from Pytorch, we implemented our own little autograd module that provides minimum necessary capabilities to be able to do gradient based learning. Although initially we tried to implement our own tensor by subclassing pytorch Tensor, we eventually found it to be a little cumbersome and error-prone, we ended up simplifying our approach by modifying the Pytorch Tensor attributes directly. More specifically, in each forward pass we build a computation graph of tensors by storing required objects inside the tensor which we can use later for back propagation. Similar to original `requires_grad` and `grad` attritubes of tensors, we set `grad_required` and `gradient` attributes for our autograd purposes. Note that setting the original attributes also work and we initially did that, but realized that when torch autograd is not set off, then we run into the issue where Pytorch also builds a computation graph alongside ours and this results in a out-of-memory crashes when models are slightly bigger. Hence, we avoided using the same names which also ensures that we are in no way using any original autograd capabilities. We also free up computation graph references once the `backward` method is finished and skip building the graph if the model is set to be in `eval` mode for memory optimization. Code for this module can be found in `autograd.py`.

3 Experiments

3.1 Unit Tests

In order to efficiently and continuously test our implementation, we wrote several unit tests covering wide array of edge cases. These tests can be run using regular python `unittest` or `pytest` packages. Tests can be found under `tests` directory.

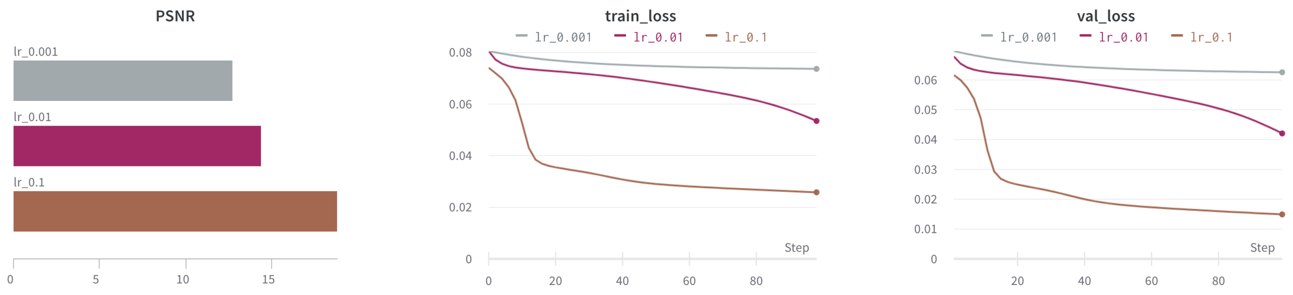

Figure 3: Effect of learning rate

3.2 Model Training

Next we train various denoising models using our framework with the model architecture suggested in the project description. Our training dataset consists of $50,000$ and validation dataset $1,000$ pairs of noisy images. Each image has 3 channels and is of size `32x32`. Each pixel in the image is represented with a value between 0 and 255. For stability purposes, we normalize the data to a range between 0 and 1 by dividing each pixel value by 255. When we predict, we cast the model output value to a range between 0 and 255 by multiplying and clamping the bigger values. We use `MSE` as a loss function and `SGD` as an optimization method. In order to see the effects of different hyper-parameters, we train 3 sets of models. In each set, we fix all the other parameters except the one we are interested in measuring the effect of and vary this parameter in a reasonably wide interval. Particularly, we look at effects of batch size, learning rate and number of input/output channels (we define only one parameter called `hidden_dim` for both channels for simplicity). For batch size, number of channels and learning rate, we experiment with the following values respectively: `[32, 64, 128, 512]`, `[24, 48, 96, 144]`, `[0.001, 0.01, 0.1]`. More details for the model can be found in `model.py`. We also provide `train.py` which we used to train our models and to perform the mentioned hyper-parameter search. We log the results of our experiments to popular *wandb.ai*[2] platform and they can be accessed here. Our best model is trained for 100 epochs with a slightly different model design (with a padding of 1 in addition to the stride) and with the following hyperparameters: `batch_size=16, learning_rate=0.1, channels=144`.

3.3 Results

For each sets of models, we report the final PSNR score and training and validation loss over epochs for various scenarious in Figure 1, Figure 2 and Figure 3. First of all, we can see that both training and validation loss consistently decreases in all settings which shows that our framework is correctly implemented. We can also see that models with smaller batch size, more number of channels and a higher learning rate perform better which matches our expectations. Our best model achieves a PSNR score of 23.32 dB.

3.4 Notes

While we believe our implementation of `Conv2d` is correct, it does seem to produce slightly different result (difference being less than 1e-6) using a different Pytorch version than ours (`1.11.0`), hence, we had to adjust the `atol` value in `test.py` to pass the test.

References

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[2]https://wandb.ai