Building PyTorch Blocks

Alessio Verardo Arthur Brousse Emna Fendri

Spring 2022

Abstract

The goal of this project is to build a framework for a Noise2Noise network without using existing features from PyTorch, that is we use only the Python standard library.

1 Introduction

We develop a deep learning framework using only the basic tensor operations. In particular, this implies that the `nn` and `autograd` modules from torch cannot be used, hence the implementations of some modules and gradient computations have to be redone from scratch. In particular, the following blocks have been implemented :

- A convolution layer
- A Nearest Neighbor upsampling layer
- An Upsampling layer

In addition, we will also need several classical functions such as :

- ReLU
- Sigmoid
- LeakyReLU
- Mean-squared error as a Loss function
- Stochastic gradient descent (SGD) optimizer
- Adam optimizer
- A container like `torch.nn.Sequential`

We discuss the implementation choices in the following sections.

2 Data and Architecture

The given training data consists of 50'000 pairs of $3 \times 32 \times 32$. Each of the 50'000 pairs provided corresponds to downsampled, pixelated RGB images. We will implement the blocks as explained in the previous section to build our network for denoising the images. The required network follows the architecture shown on fig. 1. It consists of two Convolution layers followed by two Upsampling layers. And as it will be explained in the next sections, the Upsampling consists of a `Nearest Neighbor Upsampling + Convolution`.

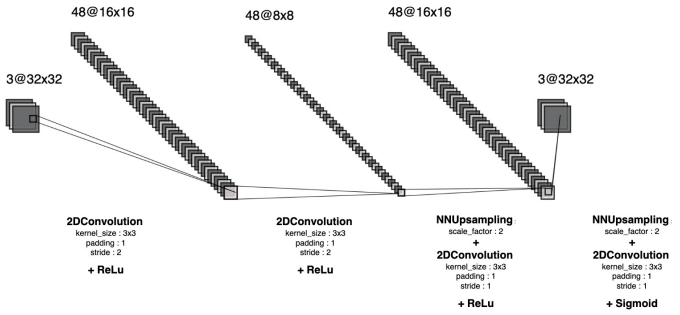

Figure 1: Architecture of the network (takes 1 sample as input)

3 Implementation

3.1 Models

The models contain the building blocks of a neural network that need to be rewritten. They inherit from a superclass **Module**, and all contain the following methods :

- *forward :* defines the forward pass of the module.
- *backward :* defines the backward pass of the module.
- *params :* returns a list of pairs composed of a parameter tensor and its corresponding gradient tensor (with the same size).
- *update :* makes an update step of all the parameters, given a learning rate.
- *zero_grad :* sets the gradient of the model w.r.t. all the parameters to 0.
- *init_parameters :* initialises the parameters of the module.
- *__call__ :* applies the forward pass. This allow us to use a syntactic sugar of the form *module(x)* to apply the forward pass of the module.
- *to:* move the parameters to the specified device.

3.1.1 Sequential :

We define a module that takes as input a list of ordered **Module** : `Sequential()`. It is a container, like `torch.nn.Sequential`, to put together an arbitrary configuration of modules together.

- *__init__(self, *modules)* : Given the modules passed in argument, it will build a sequential layer organizing them in a sequential ordered manner.
- *forward(self, x)*: Iterates over the `forward` function of each module on the network. This way passing the output of the `forward` of the current module as input to the `forward` of the next module. This can be expressed as follows :

$$Y = f(X) = f_K \circ f_{K-1} \circ ... \circ f_2 \circ f_1(X),$$

where K is the number of modules, X and Y are respectively the input and output tensors (N, C, H, W) of the network and $f_i(\ )$ being the *forward* of the i^{th} module.

- *backward(self, *gradwrtoutput)*: Iterates over the *backward* function of each module of the network in a reversed way (starting from the last module). It applies the *backward* function of the current module on the output of the previous module. Which yields to the gradient of the loss $L(\ )$ with respect to the input X that will allow us to compute the gradient of the loss with respect to the parameters. This can be expressed as follows :

$$\frac{\partial L}{\partial X} = b_1 \circ b_2 \circ ... \circ b_{K-1} \circ b_K(\frac{\partial L}{\partial Y}),$$

where $b_i(\ )$ is the `backward` of the i^{th} module.

3.1.2 NNUpsampling :

We define a module to perform a Nearest Neighbor Upsampling of a given scale factor.

- *forward(self, input)*: Given a scale factor s (necessarily an integer) and a tensor input X of shape (N, C, H, W), this function outputs the upsampled version of X of shape $(N, C, s \times H, s \times W)$. The *repeat_interleave* function is used here to perform the upsampling.
- *backward(self, x)*: This function returns the gradient with respect to the input of the module. In this case it will be a (N, C, H, W) tensor. The method first puts the correct outputs from later summation, then sums over all outputs generated from one input, and finally reshapes to the original shape.

3.1.3 Upsampling :

We define the Upsampling operation as a module that inherits from `Sequential`, as it is a combination of a Nearest Neighbour Upsampling followed by a Convolution. The following method is redefined :

- *__init__* builds an upsampling layer, taking as input *in_channels* (the number of channel as input of the convolution layer), *out_channels* (the number of channel as output of the convolution layer), *kernel_size* (the kernel size of the convolution), *scale_factor* (the scale factor for the upsampling layer), and *padding* (the padding used in the convolution layer). It then builds an upsampling layer by combining a NNUpsampling layer with a convolution one.

3.1.4 Conv2d :

We defined the convolution layer as a **Module**, taking given dimensions as input and output. It also is given a kernel size, a padding and a stride parameter. The **Module** are defined as :

- The *__init__* method initialises each convolution layer with the passed parameters, transforms the parameters to tuples if they are ints, and stores the parameters for further use in *forward* and *backward*. The gradients with respect to weights and biases are also set to 0.
- The *forward(self, x)* pass first defines the dimensionality of the output of the convolution, then computes the convolution between the input x and the weights of the current model using a matrix multiplication. If a bias term is included, it is added at that point.
- *init_parameters* initialises the parameters of the module using the He initialization [[1]] for the weights, and the value 0.01 for the biases (according to this). Bluntly, the He initialization uses zero-centered Gaussian random variables with standard deviation of $\sqrt{\frac{2}{n}}$ for the weights initialization where n is the number of parameters in the convolution kernel, i.e. $n = N_{in} \cdot N_{out} \cdot K_1 \cdot K_2$ where (K_1, K_2) is the kernel size..
- The *backward(self, x)* pass returns the gradient with respect to the input of the module. Bluntly, it does so by first transforming the gradient of the further layers into the right shape, then by multiplying it with the weights of the input using matrix multiplications. The implementation was inspired from this article.
- *param* returns the list of parameters, i.e. the weights and gradients of the convolution layer.

3.2 Activation functions :

We defined the **Sigmoid**, **ReLU** and **LeakyReLU** activation functions in our framework. Each of these functions redefines the *forward* and *backward* from the mother class **Module** as :

- **Sigmoid** :
  - $forward(self, x) = \dfrac{1}{1 + e^{-x}} = sigmoid(x)$
  - $$\frac{\partial \mathcal{L}}{\partial x_i} = backward\left(self, \frac{\partial \mathcal{L}}{\partial \boldsymbol{O}}\right) = sigmoid(x_i)(1 - sigmoid(x_i))\frac{\partial \mathcal{L}}{\partial O_i}$$
- **ReLU**
  - $forward(self, x) = max(x, 0)$
  - $\frac{\partial \mathcal{L}}{\partial x_i} = backward\left(self, \frac{\partial \mathcal{L}}{\partial \boldsymbol{O}}\right) = \begin{cases} \frac{\partial \mathcal{L}}{\partial O_i} & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$ where x_i is the i-th input and o_i is the ith output of the layer.

- **Leaky ReLU**
  - $forward(self, x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$
  - $\frac{\partial \mathcal{L}}{\partial x_i} = backward\left(self, \frac{\partial \mathcal{L}}{\partial \boldsymbol{O}}\right) = \begin{cases} \frac{\partial \mathcal{L}}{\partial O_i} & \text{if } x_i > 0 \\ \alpha \frac{\partial \mathcal{L}}{\partial O_i} & \text{otherwise} \end{cases}$
    where x_i is the i-th input and o_i is the ith output of the layer.

3.3 Loss functions

A mother class **Loss** is implemented, with methods *forward*, *compute_gradient*, and *__call__*. These methods are then defined in each subclass corresponding to different losses. We only implemented the mean-squared error loss, as it was the only one needed for the purposes of this project. For this loss, the *forward* and *backward* passes are implemented as follows :

- $forward(self, y\_pred, y\_true) = \frac{\sum_i (y\_pred_i - y\_true_i)^2}{||y\_true||}$
  where y_true and y_pred are $N \times C \times H \times W$ tensors.
- $= backward(self, x) = \frac{2(y\_pred - y\_true)}{||y\_true||}$

3.4 Optimiser

A base class **Optimiser** for all optimisers is defined, which contains an *__init__* method, which initializes a list of parameters to optimise, and a learning rate. The second method is the *step* method, which performs one step of the optimisation process. We implemented the stochastic gradient descent optimizer and the Adam optimizer (since it has shown that is really better than the SGD in this task (for mini-project 1)).

- *SGD Optimizer :* The parameters are updated using the following rule : $p_t = p_{t-1} - lr \times \nabla p$, with lr as the learning rate;
- *Adam Optimizer :* The parameters are updated using the following rule : $p_t = p_{t-1} - lr \times \frac{f_t}{\sqrt{s_t} + \epsilon}$, with lr as the learning rate, f_t and s_t as the first and second estimated moments from the previous gradients (see [[2]] for more details), and ϵ as a parameter of the optimizer.

3.5 Model

The class **Model** contains the definition of the parameters we chose for the training of our model. The parameters were chosen in order to respect the constraints given in the context of the mini-project (limited time per epoch, etc.). It also contains the methods used for training and prediction.

4 Results

We test our framework on the same dataset as the first mini-project, i.e. we train our network on a dataset made of 50'000 pairs of $3 \times 32 \times 32$ noisy RGB images. Given the simple network in the statement of the project, we manage to achieve a PSNR of 22.1108, using the Adam optimizer. The original network was trained using an SGD optimizer, which was not giving any tangible result (maximal PSNR achieved was 16.0656) both with the same training parameters, batch_size = 100, learning rate of 10^{-4} and 18 epochs. The comparison between the two optimizers is shown in fig. 2 and fig. 3. Clearly, the Adam optimiser gives a much more convincing output, as we can distinguish the shape of the image, whereas the SGD optimiser simply outputted a uniformly grey image. The image remains blurry, as the network used is still not very sophisticated. Interestingly enough, by simply removing the sigmoid at the end of the network, the PSNR goes up to 22.6. However, we stick with the original and required architecture.

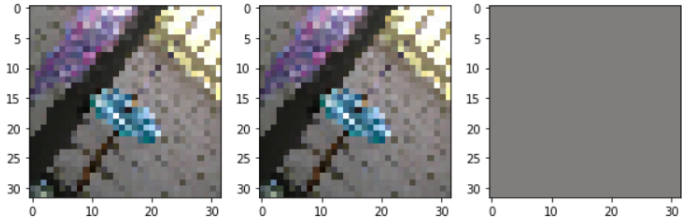

Figure 2: Denoising using a SGD optimiser

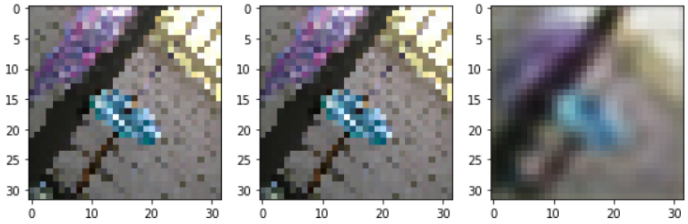

Figure 3: Denoising using an Adam optimiser

5 Conclusion

In this project, we built a small deep learning framework, in order to implement a denoiser. In order to do that, we re-implemented some of the basic functionalities available in the PyTorch framework. Our final model uses a from-scratch version of a convolution module, a tuned-version of the Adam optimizer, a learning rate of $1e^{-4}$ and a batch size of 100, and achieves a PSNR of 22.1108.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. DOI: 10.48550/ARXIV.1512.03385. [Online]. Available: https://arxiv.org/abs/1512.03385.

[2] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980.