Mini-project 2
Mini Deep Learning Framework

Marie-Alix Gillyboeuf 299327, Rhita Mamou 286507, Assia Ouanaya 288251
Deep Learning - EE-559, EPFL, Switzerland

Abstract

To deepen our understanding of the behind working of Pytorch, we will focus on implementing from scratch all the building blocks of a deep learning framework. Here we aim to implement the PyTorch modules to further construct a simple image denoiser network.

1 Introduction

Using only Python's standard library, we build different modules with losses and activation functions that apply a `forward` and `backward` propagation. The training and testing are done on the same 50'000 noisy pairs of colored images used in mini-project 1 (Lehtinen et al., 2018). Our aim is to implement a simple denoiser network using only our implemented modules. All of our data and models run on CPUs.

The first section of this report describes the data of interest. The second and third sections present the different classes we implemented. Then we show our results before finally we presenting a summary and a discussion of our work.

2 Modules

All of our modules classes inherit from the mother class `Module` (Figure 1) which contains the following three methods: a `forward`, a `backward` and a `param`. If the module has parameters, its `param` method will returns a list of the parameters of the module (*i.e.* weights and bias) with their gradient.

2.1 Conv2d

`Conv2d` is a module that performs a 2D convolution over an input signal composed of several input planes. It can be used as a layer in a neural network. The constructor instantiates the `Conv2d` layer with different arguments: the number of in channels (C_{in}), out channels (C_{out}), kernel size (k_size) but also several other parameters such as the stride, padding, dilation and a boolean for the bias. The weights (w) and bias (b) are automatically initialised such as the Pytorch `Conv2d` counterpart from the following distribution:

$$w, b \sim \mathcal{U}(-\sqrt{k}, \sqrt{k})$$

$$k = \frac{1}{C_{in} * \prod_{i=0}^{1} k\_size[i]}$$

The `forward` method takes an input of shape $(N, C_{in}, H_{in}, W_{in})$ and performs a convolution between the input and the aforementioned weights and bias if present. Convolution is the mathematical equivalent of multiplication between the weights with a reshaped input. The input is reshaped by unfolding it but also with respect to the padding, kernel size, stride and dilation settings set above. If bias is present, it is simply added to the result with the correct output size. Therefore, convolution can be thought of as a linear function. The output of the method is reshaped with respect to the parameters to the final shape $(N, C_{out}, H_{out}, W_{out})$ where:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times pad - dil \times (k\_size - 1) - 1}{stride} - 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times pad - dil \times (k\_size - 1) - 1}{stride} - 1 \right\rfloor$$

The `backward` method takes as input the gradient of the loss w.r.t. the parameters of the following layer $\frac{\partial L}{\partial s^{(l)}}$ and computes the derivative of the loss w.r.t. the parameters (*i.e.* weights and bias). These two quantities are:

$$(\frac{\partial L}{\partial w^{(l)}}) = \frac{\partial L}{\partial s^{(l)}}(x^{(l-1)})^T$$

$$(\frac{\partial L}{\partial b^{(l)}}) = \frac{\partial L}{\partial s^{(l)}}(x^{(l-1)})^T$$

The final output of the `backwark` of `Conv2d` is the gradient of the loss w.r.t. to the output of the `forward` of the previous layer:

$$(\frac{\partial L}{\partial x^{(l-1)}}) = (w^{(l)})^T \frac{\partial L}{\partial s^{(l)}}$$

The `zero_grad` method sets the gradient of the parameters to zero and the `param` method returns the parameters with their gradient.

2.2 Nearest Neighbor Upsampling

`NearestUpsampling` takes a multi-channel data and outputs another multi-channel data with the last dimensions scaled by a factor. The `forward` method performs the operation upsampling by transforming

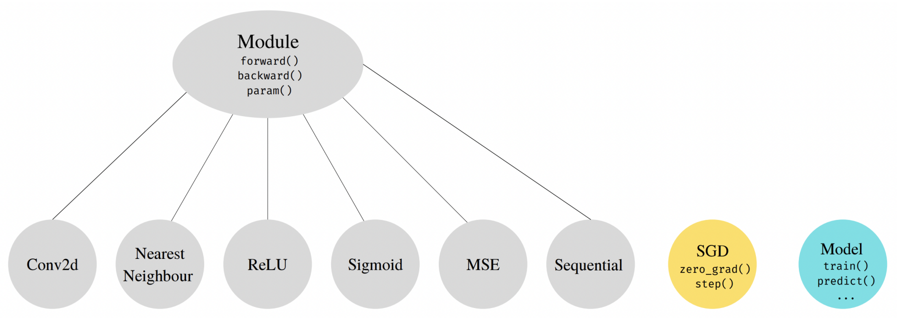

Figure 1: Relationship of inheritance between the implemented classes.

the input of shape $(N, C_{in}, H_{in}, W_{in})$ to an output shape of $(N, C_{in}, H_{in} \times s, W_{in} \times s)$ where s stands for the scale factor.

The `backward` method downsizes the Tensor input. More explicitly, it takes as input the gradient of the loss w.r.t. the following layer $\frac{\partial L}{\partial s^{(l)}}$, that has a shape $(N, C_{in}, H_{in}, W_{in})$ and it outputs $(N, C_{in}, H_{in}/s, W_{in}/s)$. That is because the returned Tensor will be then the input for the backward of the next layer. To do this, we used the unfold method, in order to do an operation over each block. Then, we sum and reshape according the desired shape.

We used the combination of `NearestUpsampling` and `Conv2d` modules to perform the equivalent of Transposed Convolution.

2.3 Sequential

We designed a version of the `Sequential` based on the module of PyTorch. The Sequential module combines layers in order to build a network. It is initialized with a list of modules that constitute the different layers.

Its `forward` method takes any input of the correct size and calls the `forward` one of each module in the order they are put in the list, such that the output of the `forward` function of a module is the input for the `forward` function of the following module.

Its `backward` method takes as input the gradient of the loss with respect to the output. It calls the `backward` of each modules of the network in reversed order, such that the output of the `backward` function of a module is the input of the `backward` function of the previous module.

The `param` method returns the concatenated list of each module parameters. The `zero_grad` method sets the gradient of the parameters to zero by calling the `zero_grad` function of each module.

2.4 Activation function

2.4.1 ReLU

The `forward` method takes as parameter the input x and returns:

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The `backward` method takes as input $\frac{\partial L}{\partial s^{(l)}}$ and returns:

$$\frac{\partial L}{\partial s^{(l)}} * ReLU'(x)$$

with

$$ReLU'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

2.4.2 Sigmoid

The `forward` method takes as parameter an input x and returns:

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

The `backward` method takes as input $\frac{\partial L}{\partial s^{(l)}}$ and returns:

$$\frac{\partial L}{\partial s^{(l)}} * \sigma'(x)$$

with

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

2.5 MSE

Given a prediction output $\hat{y}$ and a target y, the mean squared error (MSE) is defined as

$$L(\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2$$

and its gradient is

$$\nabla L(\hat{y}) = \frac{2}{n}(y_i - \hat{y}_i)$$

3 Additional classes

3.1 SGD

As an optimizer, we implemented a Stochastic Gradient Descent class. It contains a `step` function that performs one update of the parameters of the modules as follows:

$$w_i = w_{i-1} - \eta \frac{\partial L}{\partial w_{i-1}},$$

$$b_i = b_{i-1} - \eta \frac{\partial L}{\partial b_{i-1}}$$

where η is the learning rate (set to 0.1 by default)

3.2 Model

The `Model` class is initialized with a sequential object that contains the modules which constitute the network we want to train, an optimizer (`SGD` object), and a loss criterion (`MSE` object).

Its `train` method performs the training of the sequential model with input given as its arguments. The training is done as follows: first, the `forward` function of the sequential model is called and returns an output of the same size as the input, then the loss is computed by calling the `forward` method of the criterion. After that, we set the gradients of the parameters to zero by calling the `zero_grad` method before performing the backpropagation. Finally, we update the parameters of the model using the step function of the optimizer. We do so for a finite number of epochs set in the parameters of the function.

The `Model` class also contains a `save` and a `load` function which we use to save and load our best model in a `.pth` form.

4 Results

To assess correctly the performance of the implemented framework we trained the network on the test data in function of the number of epochs. Here we used a batch size of 100, number of epochs of 10 and a learning rate of 1.

Layer	C_{out}	image size	Function
input	3	32x32	
c0	48	16x16	Convolution 3x3
c1	64	8x8	Convolution 3x3
ct1	48	16x16	Transpose conv. 3x3
ct0	32	32x32	Transpose conv. 3x3
output	3	32x32	

Table 1: Network architecture of our final model. C_{out} denotes the number of output channels for each layer. Transpose convolution is implemented as an nearest-neighbor upsampling followed by a 2D-convolution. Except for the last layer which is followed by Sigmoid activation function, all the layers are followed by ReLU activation function.

We then plotted the images to compare the results between the clean image, and the predicted one (after the training). For this we used a $\eta = 1$, a batch size = 100, number of epochs = 10 and we got a PSNR = 21.91.

5 Discussion and Conclusion

With this project, we have shown that it is possible to implement from scratch Pytorch modules using only python's basic library. With our modules, we were able to create a simple model that tries to recover the signal of corrupted images, without seeing clean images and without any explicit characterization of the noise.

Through this model, we achieved a performance score of 21.91 dB, which is a bit lower than the first mini-project PSNR (22.96 dB). This is understandable as our model is more simple and shallow, otherwise, its training becomes computationally expensive. It succeeds though in denoising the input images as seen above. Optimizing the speed of our modules would be key to unlocking higher computational power. This will allow us to handle deeper networks, use higher number of epochs and therefore achieve a higher PSNR value and a better prediction.

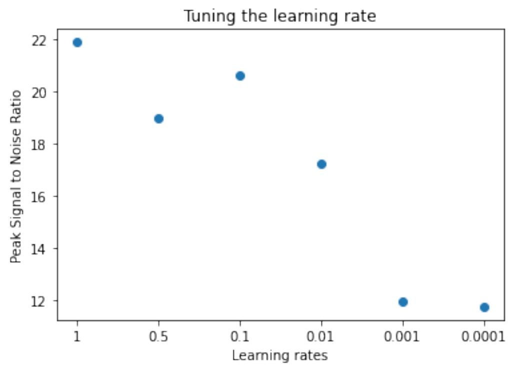

Figure 2: PSNR as a function of the learning rate of the ADAM optimizer.

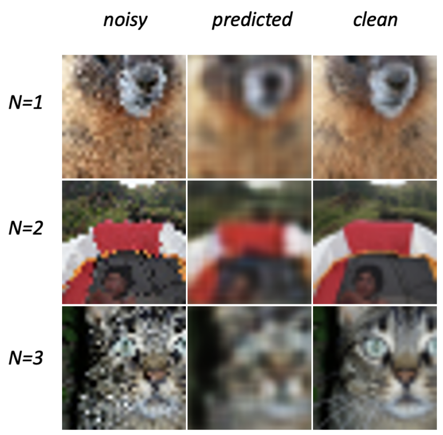

Figure 3: Results images. Noisy images were the input given to train our model, predicted images correspond to the output of our trained network and clean images were the target we tried to reach with our network.

References

Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. 2018. Noise2Noise: Learning Image Restoration without Clean Data. *arXiv:1803.04189 [cs, stat]*, October. arXiv: 1803.04189.