EE559-Deep Learning Project 2
Using a non-standard PyTorch framework

Malena Aguiriano Calvo - Henry Papadatos - Aitana Waelbroeck

***Abstract*—In this report, we explain how we implemented a deep-learning framework, using PyTorch tensors and operations and the standard python libraries, without using autograd or torch.nn modules. We then compared its performance to a PyTorch framework.**

I. Introduction

The aim of this project is to create a deep learning framework, using PyTorch tensors and operations to implement a deep learning network that can denoise images without a clean reference. The blocks that will be implemented are:

1) Convolution layer
2) Upsampling layer
3) ReLU
4) Sigmoid
5) A container like torch.nn.Sequential
6) Mean Squared Error as a Loss Function
7) Stochastic Gradient Descent optimizer

For the first part of the report, we will explain how we implemented these blocks, while in the second part we will analyze its performance and compare it to that of Pytorch's.

II. Implementation of the deep learning framework

A. Module Structure

The modules will have the following structure:

class Module (object)
    def forward (self,*input)
    def backward (self,*gradwrtoutput)
    def param (self)

Forward implements a forward pass of neural network. Backward implements a backward pass on the gradients of the loss with respect to the output. Param returns a list of tuples of parameter tensors or an empty list for functions without parameters to train (eg. Sigmoid).

B. Convolution Layer

Our convolution layer follows what is implemented in torch nn.conv2d. The weights (shape: [out_channels, in_channels, kernel_size[0], kernel_size[1]]) and bias (shape: [out_channels]) are initialized by sampling them from a uniform distribution $U(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{in} \prod_{i=0}^{1} \text{kernel\_size[i]}}$. While our weight gradient and bias gradient are initialised at 0. Our convolution module has as parameters, besides the standard ones, stride, padding and dilation. Our forward convolution, allows us to perform a linear operation by unfolding the input and reshaping the weights and bias. We then reshape it so that the output is the correct size.

The back-propagation is also done by transforming the backward pass into a linear operation. Once this transformation is done, we can apply a regular linear chain rule:

$$\frac{\partial L}{\partial W} = X^T * \frac{\partial L}{\partial Y}$$
$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} * W^T$$
$$\frac{\partial L}{\partial b} = \sum_{h=i,w=1}^{n} (\frac{\partial L}{\partial y})_{h,w}$$

Where w is the width of the image and h is the height of the image. The mapping back from linear operation to the original dimensions is done using the torch.nn.fold function, taking into account the kernel size, the padding and the stride. The convolution layer supports batch operation, and thus we have to sum the gradient contribution of each data point of the batch. The updated parameters are a list of two tuples: [(weight, gradweight), (bias, gradbias)].

C. Upsampling Layer

The upsampling layer was implemented as a combination of Nearest neighbor upsampling and a convolution. For the forward pass of the Nearest neighbor upsampling, two matrices were generated, A, for the width dimension transformation and B, for the height dimension transformation. For instance in the case of a 2 by 2 input and a scale factor of 2 we would have the following matrix for A and B: $\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$
This is the general configuration of the matrix A and B which is adjusted depending on the scale factor and the input sizes. The matrix operation applied was the following:
Output $= (((\text{Input} * A)^T) * B)^T$
Followed by a forward pass convolution.
For the backward pass we first apply a backward pass convolution. We then use the following matrix operation:
Output $= (\text{Input}^T * B)^T * A$
The backwards upsampling nearest neighbor sums up the gradients of the same region in the input matrix (shown in figure 1, eg. red region gradients summed together). The regions are of size scale_factor by scale_factor. It results in a matrix of size height divided by scale_factor and width divided by scale_factor,

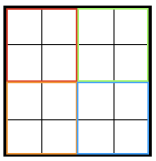

Fig. 1. Input matrix of backwards upsampling showing the regions where the gradients get summed together

The transpose operations for the forward and backward are done to transpose the last two dimensions of the tensors.
Param returns the parameters obtained from the convolution.

D. ReLU Module

This module implements the Rectified Linear Unit activation function.

$$f(x) = max(0; x)$$

It's derivative for the backward pass is defined as:

$$f'(x) = 1_{x>0}$$

There are no parameters to train so it returns an empty list in $param$.

E. Sigmoid Module

This module implements the Sigmoid activation function.

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

It's derivative for the backward pass is defined as:

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

There are no parameters to train so it returns an empty list in $param$.

F. Sequential Module

The sequential module takes as input an arbitrary sequence of modules and applies the forward pass of each module to the input, following the order given in the instructor. The backward pass of each module is done by applying the back-propagation of each module in reverse order to the one given.

G. Mean Squared Error Module

This module implements the Mean Squared Error loss. It computes the mean squared error between each element in the predictions and targets y, for n = N*C*H*W:

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

It's derivative for the backward pass is defined as:

$$L'(\hat{y}, y) = \frac{2}{n}(\hat{y} - y)$$

There are no parameters to train so it returns an empty list in param.

H. Stochastic Gradient Descent Module

This module implements the Stochastic Gradient Descent optimizer. In this case the structure of the module is slightly modified. Since it doesn't need a backward and forward, it only has a step function. $Step$ calculates the new weights w and biases b using the following equation where L is the loss and lr is the learning rate:

$$w = w - lr * \frac{\partial L}{\partial w}$$
$$b = b - lr * \frac{\partial L}{\partial b}$$

We return the parameters w, $\frac{\partial L}{\partial w}$, b and $\frac{\partial L}{\partial b}$ in the function param.

III. Results

A. Network Architecture and Hyperparameters

The network's architecture was given in the project description and is as follows:

Sequential (Conv2D(input_channel = 3,output_channel = 32,kernel_size = 3, padding = 1, stride = 2) ,
ReLU() ,
Conv2D (input_channel = 32,output_channel = 32,kernel_size = 3, padding = 1, stride = 2),
ReLU() ,
Upsampling (input_channel = 32,output_channel = 32,kernel_size = 3, scale= 2, padding = 1) ,
ReLU() ,
Upsampling (self.channel,3,kernel_size = 3, scale= 2, padding = 1) ,
Sigmoid())

We performed the data normalization by dividing the data by the maximal value of a pixel (255).
We used the following hyperparameters to train our neural network:

Hyperparameter	Value
Learning rate	2.5
Batch size	40
Epochs	100
Number of images used	50000

When we were tuning the hyper-parameters, we realised that the learning rate needed to be big (2,5) for the model to train. This is due to the fact that $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ had really small values, and following the formula of the SGD II-H, the learning rate had to compensate for such small derivatives.

We then implemented exactly this architecture using PyTorch models and autograd. We implemented the Upsampling by applying a torch.nn.UpsamplingNearest2d followed by a torch.nn.Conv2d. The other modules can be replaced by their PyTorch equivalent. The following graph show the comparison between the PyTorch and our from scratch implementation:

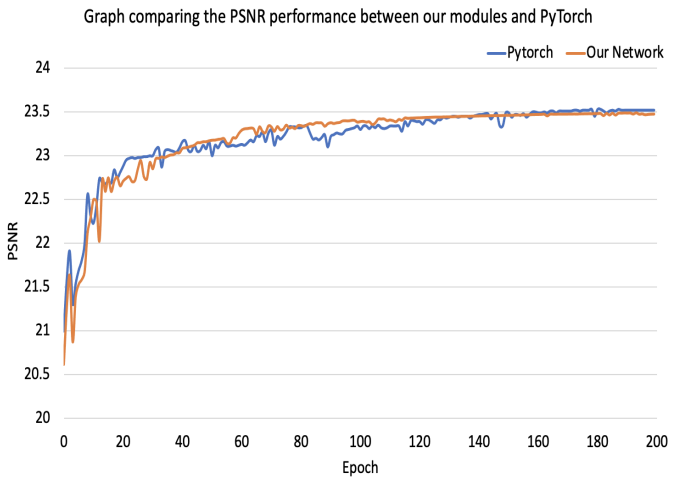

Fig. 2. Our Network's performance vs PyTorch's. Note that the performances are not exactly the same because the initialisation of the weights are randomly sampled from a uniform distribution, and are therefore different for the two models plotted.

B. Your analysis

The graph shows that our model build from scratch performs as well as the equivalent Pytorch implementation, reaching around 23,5 dB.This is true for the same architecture and hyper parameters. Therefore, our implementation performed well.

IV. Conclusion

We were able to implement a framework without using pre-existing neural network toolboxes, that performed similarly to that of PyTorch. However, many improvements could be made in order to improve the performance. For instance we implemented a very simple SGD. Momentum, dampening and weight decay could be added to see how it would affect the performance.
This project showed that simple deep learning structures can be built by ourselves and perform well compared to already existing frameworks.

V. Appendix

The Pytorch version used to plot Fig.1 can be found in the folder "others" of "Miniproject_2" under the name: "Miniproject2_pytorch_version". The folder "others" also contains a python file named "debug_model". This is the file we used to debug, since it measures the psnr and shows the evolution of the training images by plotting the images after each epoch.