Report: Mini-project 2: Write your own blocks

Castagna Léandre, Epple Pascal, Jaoua Salima
EPFL Lausanne, Switzerland

***Abstract*—The aim of this project is to implement a Noise2Noise model, meaning a model that denoises an image without a clean reference image. In this project, the utilisation of the PyTorch framework is not allowed.**

I. Introduction

Nowadays, it is possible to train convolutional neural networks to denoise images, without using images. Indeed, it has been shown in class that denoising can be achieved without clean samples, if the noise is additive and unbiased. This procedure is called Noise2Noise. The purpose of this paper is to implement Noise2Noise without using the PyTorch framework, except for using PyTorch tensors. Each of these blocks will then be implemented : a convolutional layer, an upsampling layer as a combination of Nearest Neighbor upsampling and a convolution, different activation functions, one optimizer (SGD) and a container that puts together an arbitrary configuration of modules.

II. Code Architecture

Every block (except the SGD optimizer) described in the Introduction inherits the characteristic of a *Module* superclass.

```
class Module (object) :
    def forward (self, *input):
        raise NotImplementedError
    def backward (self, *gradwrtoutput):
        raise NotImplementedError
    def param (self) :
        return []
```

Listing 1. Module class

The method names are straightforward: the *forward* method takes the input as argument and returns the forward pass, the *backward* method computes and stores the gradients needed to update the weights and biases of the Module and back-propagates the gradient with respect to the input, and the *param* method returns the parameters (weight, bias and their respective gradients) of the module (if they exist). The forward pass of the container Module (comparable to *nn.Sequential*) will first repeatedly call the forward passes of the modules it contains, and then repeatedly call their backward passes (in reverse order).
The SGD optimizer is implemented in a separate class which consists of a single *step* method. This method iterates through the parameters of a given model and updates them according to the SGD scheme.

III. Methodology

In this section, we present the different implemented modules that define the building blocks of our own framework.

A. Convolution layer

In a convolutional layer, a filter (also called kernel) slides over the input data, performing element wise multiplication. As a result, the convolution yields an output of different size than the input and thus it allows us to reduce (or even increase, in some cases) the size of an input which is useful when implementing a Noise2Noise model. This layer is mostly used to learn several image features.

1) Forward pass: The convolution operation slides with a filter over the input. This operator can be seen as a linear operator. Then,

$$Y = conv(X) = X \star W = reshape(\tilde{W}\tilde{X}) \quad (1)$$

where X $\in \mathbb{R}^{N_{batch} \times C_{in} \times H \times W}$ is the input,
W $\in \mathbb{R}^{C_{out} \times C_{in} \times k_h \times k_w}$ is the weight of the filter and $\star$ represents the convolution operator.
$\tilde{W} \in \mathbb{R}^{C_{out} \times (C_{in} * k_h * k_w)}$ and $\tilde{X} \in \mathbb{R}^{N_{batch} \times S_1 \times S_2}$ are obtained after some reshaping of W and X respectively.

2) Backward pass: Here, we are interested in $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial W}$. Since the convolution Forward pass can be seen as a linear operator as we saw in 1 with some reshaping. We first consider $\tilde{Y} := \tilde{W}^T\tilde{X}$,$\frac{\partial \tilde{L}}{\partial \tilde{W}}$ and $\frac{\tilde{L}}{\partial \tilde{X}}$. These two derivatives are just given by the back propagation of a linear layer, which are given by:

$$\frac{\partial \tilde{L}}{\partial \tilde{X}} = \tilde{W}^T \frac{\partial \tilde{L}}{\partial \tilde{Y}} \text{ and } \frac{\partial \tilde{L}}{\partial \tilde{W}} = \frac{\partial \tilde{L}}{\partial \tilde{Y}} \tilde{X}^T.$$

However, we are interested in $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial W}$. To be able to transform $\frac{\partial \tilde{L}}{\partial \tilde{W}}$ to $\frac{\partial L}{\partial W}$ we just need to make the inverse operation that we made to W. As $\tilde{X}$ was just obtained by reshaping X using the function *view* from python, the inverse operation is just the function *view* in the dimension of X. Thus, $\frac{\partial L}{\partial W}$ is obtained by just reshaping $\frac{\partial L}{\partial \tilde{W}}$ using *view*. It is a similar argument to go from $\frac{\partial \tilde{L}}{\partial \tilde{X}}$ to $\frac{\partial L}{\partial X}$. Since the transformation $\tilde{X}$ was obtained by using the function *unfold* in python on X. We use the function *fold* on $\frac{\partial \tilde{L}}{\partial \tilde{X}}$ in order to get $\frac{\partial L}{\partial X}$.

B. Nearest neighbor upsampling

The nearest neighbor upsampling layer is a simple layer with no weights that will augment the dimension of the input according to some predefined scale factor. Upsampling combined with a convolutional layer can be seen as a sort of transposed convolution operation.

1) Forward pass:

Mathematically, the nearest neighbor upsampling with a scale factor s on an input X can be stated as:

$Y_{ij} = X_{kl}$ for $(k-1)s < i \leq ks$ and $(l-1)s < j \leq ls$

2) Backward pass:
There are no weights to update during the backward pass of the nearest neighbor upsampling layer. In other words:

$$\frac{\partial Y}{\partial W} = 0$$

The backpropagated gradient, on a 2D input and with a scale factor of s, is the following:

$$(\frac{\partial Y}{\partial X})_{ij} = \sum_{k=1}^{s}\sum_{l=1}^{s} Y_{((i-1)s+k)((j-1)s+l)}$$

C. Activation functions

- *ReLU* : The Rectified Linear Unit activation function is one of the most used activation function in machine learning.
  1) Forward Pass : The forward pass applies the ReLU to the 2D input X:

  $$Y_{ij} = ReLU(X_{ij}) = \begin{cases} X_{ij}, & \text{if } X_{i,j} \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

  2) Backward Propagation : The gradient of the output with respect to the input is :

  $$(\frac{\partial Y}{\partial X})_{ij} = \begin{cases} 1, & \text{if } X_{ij} \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

- *Leaky ReLU*: The leaky ReLU is based on ReLU, but it has a small slope for negative values. The slope is determined by a non-negative parameter α. The leaky ReLU is supposed to better handle sparse gradients.
  1) Forward propagation: The forward pass performs the following operation on a 2D input:

  $$LeakyReLU(X_{ij}) = \begin{cases} X_{ij}, & \text{if } X_{ij} \geq 0, \\ \alpha X_{ij} & \text{otherwise} \end{cases}$$

  2) Backward propagation: The gradient of the output with respect to the input is :

  $$(\frac{\partial Y}{\partial X})_{ij} = \begin{cases} 1, & \text{if } X_{i,j} \geq 0, \\ \alpha & \text{otherwise.} \end{cases}$$

- *Sigmoid* : Sigmoid is an activation function that is bounded between (0,1), defined and differentiable for all real.
  1) Forward Propagation : During the forward pass, the sigmoid function, often denoted as σ, is applied to every element of the input:

  $$Y_{ij} = \sigma(X_{ij}) = \frac{1}{1+e^{X_{ij}}}.$$

  2) Backward Propagation : The gradient of the output with respect to the input is

  $$(\frac{\partial Y}{\partial X})_{ij} = \frac{\partial \sigma(X)}{\partial X}_{ij} = \frac{e^{-X_{ij}}}{(1+e^{-X_{ij}})^2}.$$

D. Optimizer

The only optimizer implemented in this framework is the Stochastic Gradient Descent (SGD) algorithm. The module is initialized by passing the learning rate lr, which must be positive, and the model whose weights need to be updated.The method step (at step i)updates the parameters of the model using the typical Stochastic Gradient Descent algorithm:

$$W_i = W_{i-1} - lr \times \frac{\partial L}{\partial W_{i-1}}$$

E. Loss function

In our framework, just one loss function has been implemented : MSE Loss. The MSE loss function that computes the l_2-norm of the difference of two inputs.

1) Forward Propagation: The forward function takes two inputs : an input $X \in \mathbb{R}^{n\times c\times h\times w}$ (computed by the model) and a target T. On n samples, the forward is given by:

$$y = \frac{1}{n\times c\times h\times w}\sum_{i=1}^{n}(X_i - T_i)^2$$

2) Backward Propagation : The gradient of the output with respect to the input X, if $X \in \mathbb{R}^{n\times c\times h\times w}$, on n samples, is given by:

$$\frac{\partial Y}{\partial X} = \frac{2}{n\times c\times h\times w}\sum_{i=1}^{n}(X_i - T_i) \qquad (2)$$

F. Sequential

This model is used in order to be able to build a neural network. Indeed, it corresponds to a container that puts together an arbitrary configuration of modules. They are added to it in the order they are passed in the constructor. To initialize the container, we give in arguments a list of modules that is append in the order of the input. Let's present the different methods that are implemented in Sequential:

1) Forward: Forward implement the forward pass of the container and takes in argument an input. For each module in our list, we call the function forward of this module on the variable named input. However, in each step, the input changes and becomes the output of the forward pass.
2) Backward: Implements the backward of the container by taking in input the gradient of the loss with respect to the input. And for each module, we call the function backward of the module of the variable gradient which represent the output of the step before.
3) param : This function returns for each module, the parameters and the corresponding gradient. Meaning, for each module, it returns the weight following by the gradient of the loss with respect to the weight and the bias following by the gradient of the loss with respect to the bias
4) zero_grad : Set all the gradients to 0.
5) add_modules :This function allows us to add modules into the container after the initialisation.

IV. Testing

To test our framework, we first built the basic model given in the project description. In this basic model, we first increase

```
Sequential(Conv (stride 2),
           ReLU,
           Conv (stride 2),
           ReLU,
           Upsampling,
           ReLU,
           Upsampling,
           Sigmoid)
```

Fig. 1. Network used to test our framework

the number of channel from 3 to $Channel_{ini}$ in the first convolution layer. In the next convolution, we increase the number of channels by a factor 2. We then decrease the number of channels to $Channel_{ini}$. Finally, we decrease it to get back three channels. In order to optimize this model, we want to visualize the way the model performs when we increase the hyper parameter $Channel_{ini}$. In the figure 2, we can see how the model behaves when we increase $Channel_{ini}$. By looking on it, we have therefore chosen to take $Channel_{ini} = 32$. In fact, the maximal number of channels is chosen to be 32 otherwise the model will then be too complex, and so will take a lot of time to run.

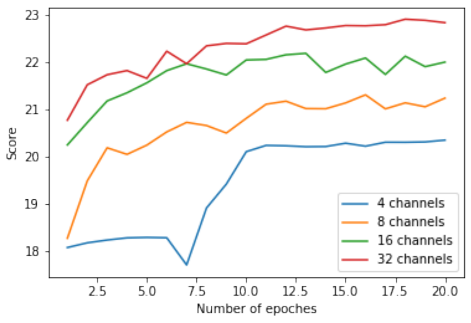

Fig. 2. Network used to test our framework

As in the Project 1, we compared our trained model by visualizing the different images in order to have an idea on how our model work. There is the noise image, the image returned by our model and the clean image as we can see in the figure 3. The results are still impressive by analogy to the final model constructed using PyTorch in project 1. One can also notice that there are no major differences on how well the model performs between the model of the project 1 and the model of the project 2. That's why we chose to plot the same pictures. However, there is a quite huge different in psnr score 25.68 and 23.8 respectively.

Regarding tech specifications, 30 epochs of the implemented model run in about 150 seconds on the online platform *Kaggle*, whose GPUs are *NVidia K80*s. On a CPU, 1 epoch takes around 6 minutes (MacBook Pro, 2.3 GHz Intel Core i5 dual core).

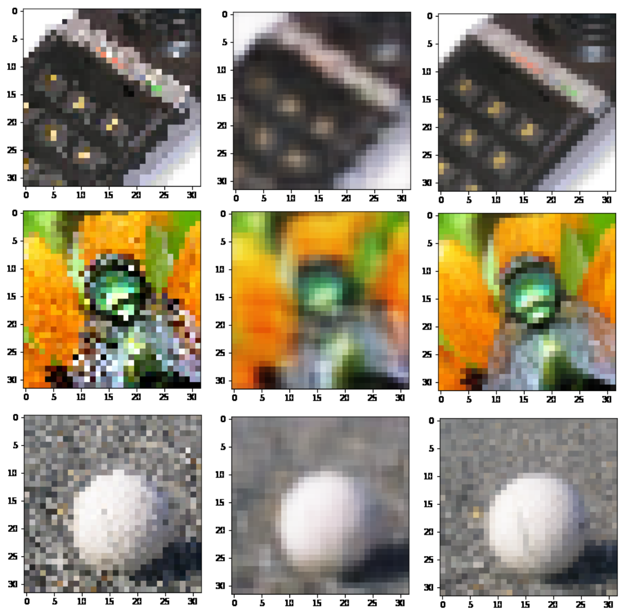

Fig. 3. *Result of the Model on 3 images selected from the data set.For each image, we plot the image given in input with the noise (left), the image obtained by the model (middle) and the image without noise (right)*

V. Conclusion

In summary, we built a deep learning framework from scratch during this project. Different modules such as convolutions and upsampling, typically available with PyTorch, were re-implemented to build a network.

The convolutional layer plays an important part to reduce the size of the input as we need it in Noise2Noise procedure where it is specially in the encoder part. The upsampling layer is a main technique to increase the size of the input as it is required in decoder part.

We also implemented different activation functions, one optimizer (SGD) and one loss function (MSE). To connect all of these modules in order to test a Noise2Noise model, we implemented a class *Sequential* to put together an arbitrary configuration of modules together.

Finally, to test our new framework, we implemented a Noise2Noise model with optimized parameters to compare the following model to the one constructed with the PyTorch framework.