EPFL EE-559 Deep Learning

Miniproject 2

Author:
Nicola Antonio Santacroce, SCIPER : 286331

Report : 3 pages, Appendix : 4 pages
May 27, 2022

1 Introduction

This report details the methods and results from the second miniproject. In this project, trainable modules are implemented with their forward and backward pass using only empty, cat, arange, fold and unfold from the Pytorch framework. a Noise2Noise deep learning model implementation using the created modules is then trained and tested. The training data consists in 50000 pairs of 32 pixels x 32 pixels images with different noise applied to both. The validation data consists in 1000 images pairs with a noisy input and a clean target.

2 Architecture

The implemented modules are briefly described here. Their implementation follows the same structure : class with a *__init__* method to define their attributes, a *forward* method that performs forward pass on an input and returns an output, a *backward* method that performs backward pass on the gradient w.r.t the output and returns the gradient on the input and a *param* method that returns the module parameters if existing. Additionally, some classes have a *to* method allowing to map needed attributes to the desired processing device.

Conv2D : The 2D convolutional layer has weights and bias declared in the *__init__* method and initialized following the Pytorch documentation. The Forward pass unfolds the input and multiplies it with the reshaped weights tensor. The result is then folded (NB using view instead of fold is also possible as the kernel size is equal to 1) back to have a shape (N,C,H,W). In the backward pass, the gradient bias is computed by summing the gradients for each output channel. The weights bias is obtained by multiplying the unfolded input with the reshaped gradient w.r.t. output. The input gradient is obtained by multiplying the reshaped weights with the reshaped gradient w.r.t. output. The

param method returns the weights and bias parameters paired with their graidents.

NNUpsampling : The 2D nearest neighbout upsampling layer has an attribute scale. In the forward pass, the input is reshaped to (N,C,H*W), repeated along the last channel using the squared scale as factor and then folded using the shape as kernel size. The backward pass sums the values of the unfolded gradient w.r.t output.

Upsampling : The upsampling layer combines NNUpsampling and Conv2D in a sequential way, allows to perform forward and backward passes and to return the Conv2D parameters.

ReLU : The layer sets negative inputs to zero in the forward pass and kills the gradient where values have been zeroed during the backward pass.

Sigmoid : The layer applies a sigmoid activation on the input in the forward pass and applies the derivative of the sigmoid on the gradients w.r.t ouput in the backward pass.

Sequential : This module is a container that allows to perform successively the forward passes of the defined sequence of modules. Backward passes are performed in the reverse sequence order. It allows to retrieve modules parameters in a sequential way.

MSE : The mean squared error modules returns the mean of the squared difference of input and targets in the forward pass. In the backward pass, it returns two times the difference.

SGD : The stochastic gradient descend allows to update the models parameters by substracting their accumulated gradients to their current value when the method *step* is called. It also sets gradients to zeros when the method *zero_grad* is called.

Parameter : Finally, a parameter class has been set with a value as attribute and methods allowing to set its value, increase it by the desired value and set it to zero (*set_value*, *add_value*, *zero_value*).

The architecture is GPU enabled by adding *to* method to all classes defined above except for the MSE and SGD. Training on random normal distributed inputs and targets of size (100,3,32,32) for 100 epochs is roughly 2.5 times faster using GPU than CPU as the computing device (see modules/modules_device_enabled.py) as shown on screenshot in appendix A.

The model implementation follows the instructions given in the project instructions. namely : Conv2D (stride 2), ReLU, Conv2D (stride 2), ReLU, Upsampling, ReLU, Upsamplng, Sigmoid. In addition, the output is multiplied by 255 to reach the range of the targets during the forward pass and divided by 255 during backward pass as the sigmoid only outputs values from 0 to 1.

3 Data Augmentation

The Model class in the model.py file contains an additional method called *augment_data()* that allows to augment data before using it for training, allowing to mitigate overfitting

risks. The implemented data augmentation consists in three transformations. The first one is the addition of gaussian noise centered in zero and with a standard deviation of 10% of that of the batch. Then, random transpose and random rotation (with a step of 90 degrees) are applied to both input and target.

4 Architecture and Parameters Tuning

The architecture in the project guidelines does not specify the number of channels in the intermediate steps of the model passes. The number of channels in the signal as then been defined as 3 to 64 to 128 and then back to 64 and 3. Some trials have been performed with intermediate channels set as (128, 512) and (512, 1024). The performance on the PSNR score was slightly enhanced but the computation time increased drastically with the number of parameters. It has thus been decided to stick to (64, 128). Results can be seen in appendix B.

Similarly, context sizes of convolution layers have been tuned. Convolutions in the Conv2D layers have been tested with (kernel_size : 2, padding : 0, stride : 2), (kernel_size : 4, padding : 1, stride : 2) and (kernel_size : 6, padding : 2, stride : 2) while the ones in the Upsampling layers were tested with (kernel_size : 3, padding : 1, stride : 1), (kernel_size : 5, padding : 2, stride : 1), (kernel_size : 7, padding : 3, stride : 2). Results are in appendix C. Increasing the context offers a great enhancement to the PSNR performance of the model and only has a little influence on the training time as the number of added parameters is quite low.

Once the preferred architecture defined, learning rates of (0.01, 0.001 and 0.0001) have been tried with both batches sizes of 100 and 10.

5 Final Model

The best model has the following configuration :

```
Sequential(
  Conv2d(in_channels=3, out_channels=64, kernel_size=6, padding=2, stride
    =2, dilation=1),
  ReLU(),
  Conv2d(in_channels=64, out_channels=128, kernel_size=6, padding=2,
    stride=2, dilation=1),
  ReLU(),
  Upsampling(in_channels=128, out_channels=64, kernel_size=7, padding=3,
    stride=1, dilation=1, scale_factor=2),
  ReLU(),
  Upsampling(in_channels=64, out_channels=3, kernel_size=7, padding=3,
    stride=1, dilation=1, scale_factor=2),
  Sigmoid(),
    )
```

This configuration resulted in the best model, achieving a 22.797 PSNR after 100 epochs with a learning rate of 0.01 and a batck size of 10. The model's parameters are saved in the *bestmodel.pkl* file and the model training can be visualised on the figures in the appendix D. Predictions on the first three validation images are displayed in the appendix E.

A Appendix : GPU vs CPU training

```
> python modules/modules_device_enabled.py
Loss at epoch 100 :      1.0098403692245483
time for device cuda : 21.091716051101685
Loss at epoch 100 :      1.0095473527908325
time for device cpu : 52.04283571243286
```

Figure 1: GPU vs CPU training speed comparison on 100 epochs

B Appendix : Intermediate channels tuning

Figure 2: PSNR with different channels sizes (chA and chB)

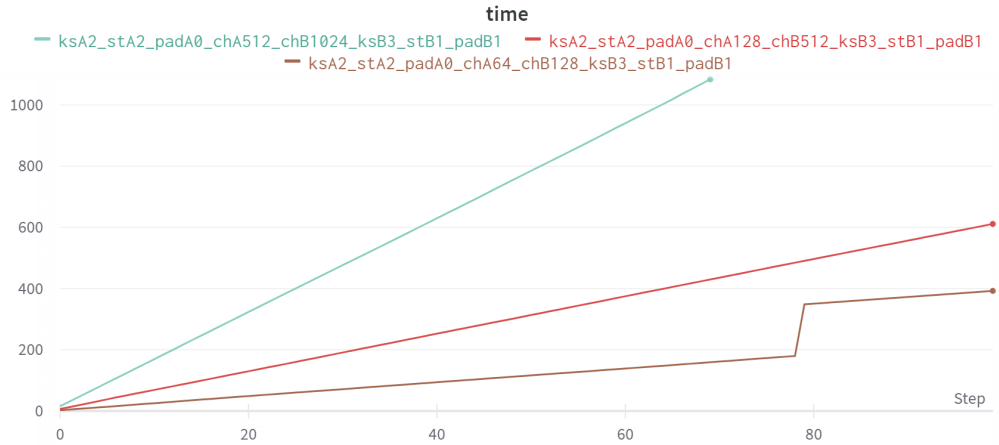

Figure 3: Training time with different channels sizes (chA and chB)

C Appendix : Convolution context tuning

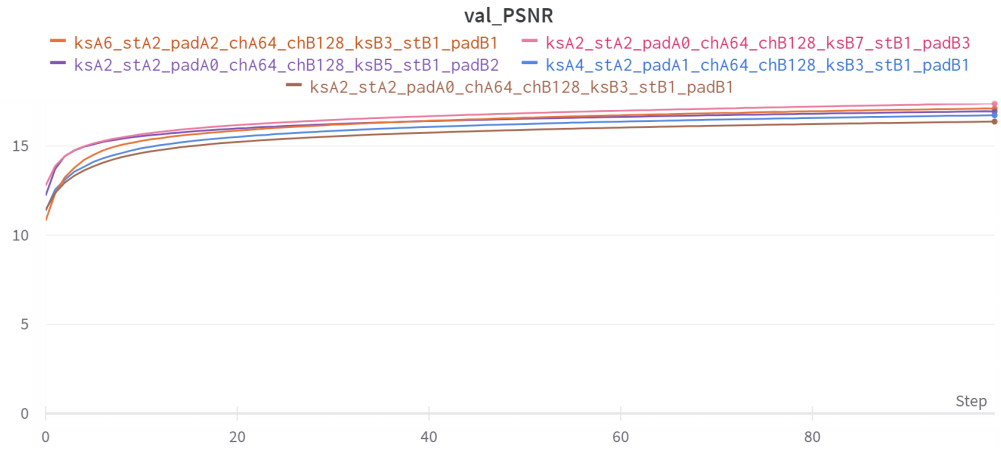

Figure 4: Training time with different context sizes

D Appendix : Best model training

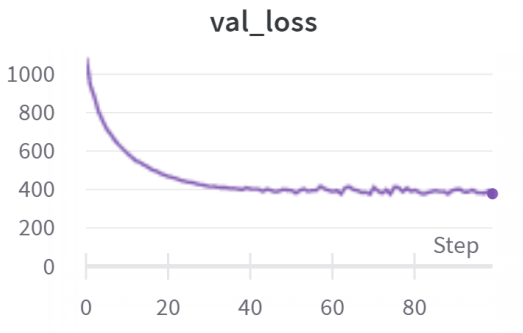

Figure 5: Validation loss

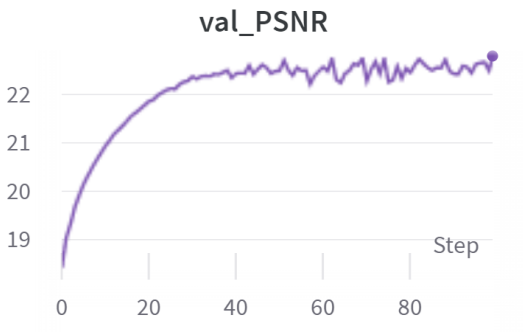

Figure 6: Validation PSNR

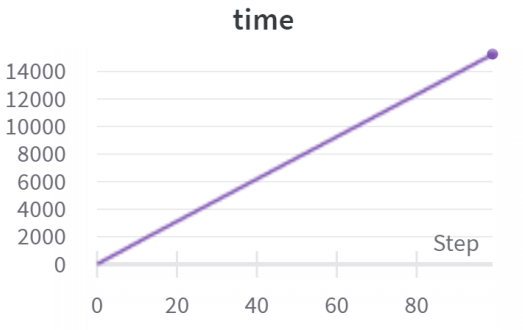

Figure 7: Training time

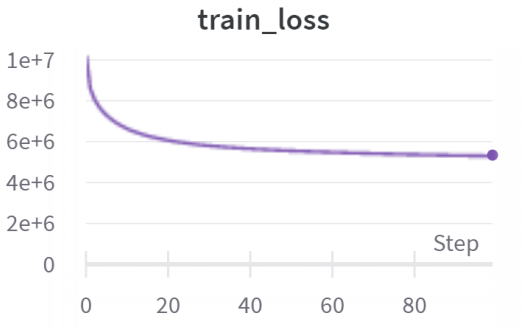

Figure 8: Training loss

E Appendix : Best model output examples

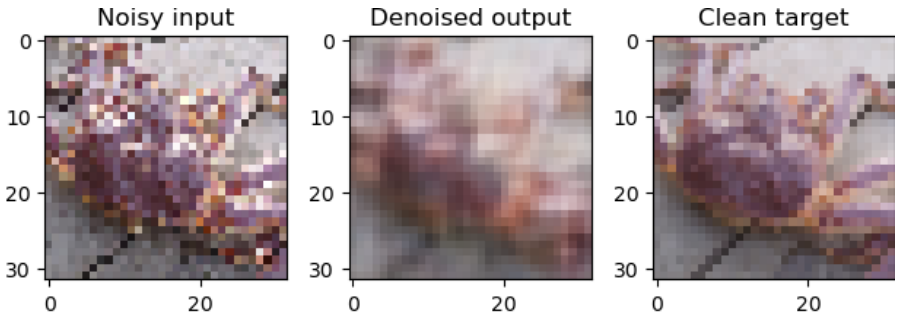

Figure 9: validation image 1

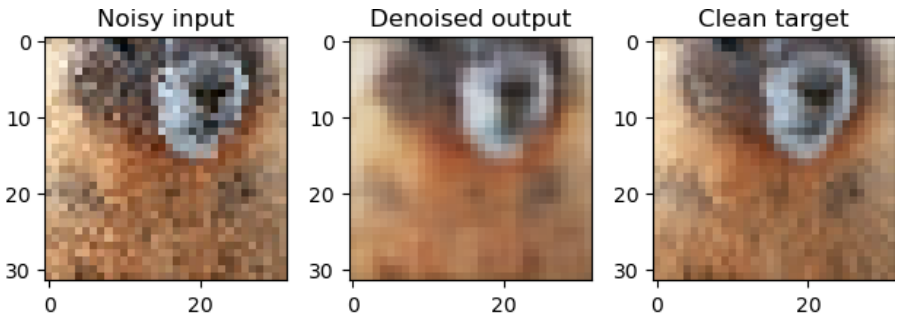

Figure 10: validation image 2

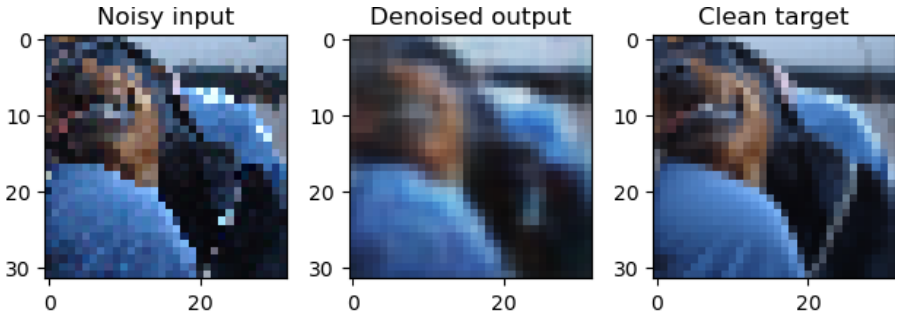

Figure 11: validation image 3