Report: Mini-project 1: Using the standard PyTorch framework

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 allowed.**

I. Introduction

Nowadays, it is possible to train convolutional neural networks to denoise images, without using clean 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 using the standard PyTorch framework. In order to analyse our models, a data set file of size 50000 noisy pairs of images is given. Each of the 50000 pairs provided corresponds to downsampled, pixelated images. Our aim is to train a network that uses these two sets of images to succesfully train a denoiser.

II. Model construction

In this section, we introduce the structures of our models. A denoising Autoencoder consists of three steps. First, the data is passed into the network, this is called the Input layer. Secondly, the information needs to be processed by applying weights, biases and activation functions. This consists of Encoder and Decoder, which together form the so called Hidden layer. Finally, we also implement the Output layer, which matches the input neurons and is, as its name suggests, the output of the model.
First, simple models consisting only of layers have been implemented but the score of theses models is very bad. Therefore, we decided to implement a **U-Net** architecture, inspired by the following paper [1], who presents a model based on the Noise2Noise technique. It consists of an encoder network followed by a decoder network, where pooling operations are replaced by upsampling operators.

A. Model 1

The first model we implemented corresponds to a quite basic **U-Net** architecture, and we detail its components in the following part:

1) **Encoder part :** composed of 2 convolutions layers (with kernel size of dimension 3×3, an output channel of size of a given number of channel (21 by default) and other default setting) combined with *ReLU* activation function. This part terminates with a *MaxPooling* with kernel size of 2, which reduces the dimension of the image from 32×32 to 16×16.

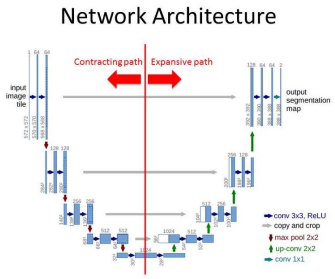

Fig. 1: U-net architecture

2) **Bottleneck part :** This step corresponds to the middle of the "U" shape that we can see in 1. This section has 3 convolutions layers with same parameters as in the Encoder part, where we just also increase the number of channel by a factor 2. Again, we combine this convolutions by *ReLU* activation function. Finally, the step ends with a transposed convolution with kernel size of dimension 3×3 and a stride 2 which enables our images to increase the dimension and get their initial dimension meaning 32×32, however we don't consider the channel size yet.
3) **Decoder part :** It consists of the right part of the "U" shape in 1. It is composed by 2 convolution layers with same settings as in the Encoder part, except input and output channels, and same activation function. The first convolution take the output of the Bottleneck step and also the input image at the beginning of the model and thus increase the number of channels by the channel's number of the input image i.e 3. Then, the second convolution decreases the number of channel of the input by a factor 2.
4) **Output layer :** Here, we just add a convolution layer which allows us to decrease the number of channel of the input (of the decoder part) to 3, which is the number of channel of the initial image.

B. Model 2

This model can be seen as an update of the *Model 1*. Instead of considering just one encode and decode level, we consider two levels of encode and decode by repeating the same structure than in *Model 1*. In other words, each encoder level has 3 convolutions layer combined with *ReLU* activation function and ends with a *MaxPooling*. Therefore, we decrease

the input image size from 32×32 to 8×8. The bottleneck part stays unchanged and the decoder part (as the encoder part) we just add one level to re-increase the image size from 8×8 to 32×32. Lastly, as a rule of thumb, the model doubles the amount of channels for each downsampling operation which, in our case, is done by a factor of 2.

C. Model 3

Finally, a last model has been implemented as an update of the second model. It is quite similar to the model presented in the paper [1]. The structure of the model is composed of three levels of encode and decode. Each part is repeating the same architecture as before, i.e each composed of convolution layers and ended with a *MaxPooling* of dimension 2 for encode level and a transposed convolution with a kernel size $(3, 3)$ and stride 2 (in order to increase the dimension of the input by a factor 2)). An other specificity of this model is that the number of channels is directly increased from 3 to 48 and then increased again in the decoder part when we add the images from encoder part, and finally we decrease it again to reach the initial number of channel of the images, 3 in this case.

D. Skip Connections

We also added skip connections to every single model. These connections ensure feature reusability, and proved to be very effective in order to optimize the performance of the denoiser. The skip connections can also be seen in 1. They correspond to the gray arrows, which correspond to already learned features, litteraly "skipping" a whole part of the network to join it again on a later point.

III. Model selection

In order to improve the score of our models, one idea has been to augment our training data. Indeed, Data Augmentation consists on forming new and different examples to train data sets. In addition, a second idea has been to implement a Grid search on our hyper parameters. One can see that there are a lot of hyper parameters in a neural network, and making them vary will have an impact on the effectiveness of the denoiser. In addition, we also implemented some functions in order to optimize our model, such as the initialization of the model parameters. We also tried several optimizers to find which one was best suited to minimize the loss function, taking also time to optimize into account. **Note that** the PSNR scores given in this section were all computed with a different function that the one given in *test.py* file. The function is defined as follow:

A. Initialization of parameters

Before optimizing our model, in addition of the default parameter initialization of PyTorch, we also performed a *Xavier normal* initialization.

B. Data Augmentation

Data augmentation can be done with different methods. We decided to implement two data augmentations and studied their performance. First, we created a new data set by only switching the input and the target images (Data augmentation 1). This means that for each pair provided, the target image becomes the one without the noise. A second data augmentation is to rotate some images that we add to the data set (Data augmentation 2). These two implementations are located in the same function, that can be found in the *others* folder.

Data augmentation 1	Data augmentation 2	PSNR score
No	No	24.4926
Yes	No	24.6371
No	Yes	24.6286
Yes	Yes (180)	24.6910
Yes	Yes (90)	24.7218
Yes	Yes (45)	24.6730

TABLE I: Results of data augmentation techniques

In the table **??**, we can see different scores when we used Model 3 with a learning rate of 10^3, *Adamax* as the optimizer and the *MSE* loss. The numbers in bracket in column "Data augmentation 2" correspond to the angle in degrees the images were rotated. One can notice that both data augmentation techniques seem to positively affect the final PSNR, which varies from 24,49 to around 24.63. When we combine both techniques together the PSNR still increases a little. We can also observe that the rotation of 90 degrees seems to perform best.

C. Optimizer selection

The decision of the optimizer is important to increase the score of the model. We decided to study the efficiency of each optimizer in order to select the optimizer.

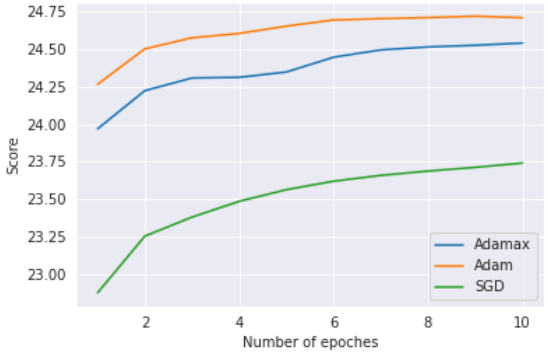

Fig. 2: Comparison between the different Optimizers

As can be seen in figure2, the SGD optimizer operates really bad compared to the other two others optimizers (Adam and Adamax). The best one is Adam but one can notice that there are not a lot of difference of scores between Adam and Adamax. Therefore, we decided to work with both and compare the results at the end of our model selection.

D. Grid Search

As we all know, a neural network uses many parameters. One way can be to arbitrarily choose a value for each one of them. But one can also implement a grid search algorithm.

It consists of performing hyper-parameters tuning on a predefined range in order to determine the optimal combination of parameter values.

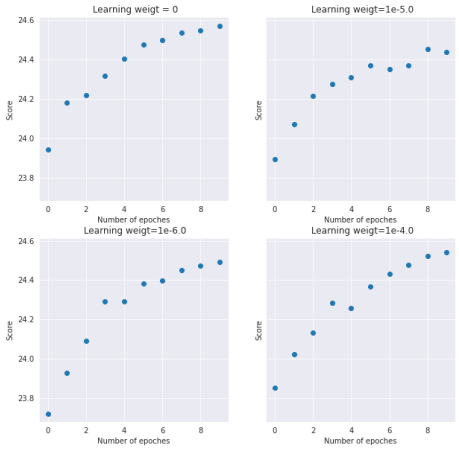

Fig. 3: Grid Search on the number of epochs and the learning weight

In the figure 3, a grid search is performed on both the number of epochs and the learning weight. For each value of learning weight, we study the score of the neural network depending on the number of epochs since we know that it can reduce over-fitting and generalizes our model. One can see that however the value of learning weight, more there are epochs, the more the value of the score increases. So we decided to take 10 epochs, which is also reasonable in terms of computation time. The maximum value of the score is obtained with a learning weight equal to 0.

E. Other methods

We also tried several techniques in order to generalize our model and improve the optimization of the latter, such as Dropout and Batchnorm.

Batch Normalization	Dropout (with probability)	PSNR score
No	No	24.6698
Yes	No	24.2704
Yes	Yes (0.3)	24.0307
No	Yes (0.3)	24.5388
No	Yes (0.2)	24.5719
No	Yes (0.1)	24.6172

TABLE II: Results of dropout and batch norm. techniques

In the table II, we can notice that the batch normalization did not perform really well. We can also observe that the dropout technique does not seem to be really useful on our model (Model 3). Another try was to change the activation function and tried to use *Sigmoid* and *Leaky ReLU* as activation functions. The first one was not really convincing. We tried the latter with a negative of slope of 0.1 which did not give any improvement. For these reasons, we decided to continue with the *ReLU* function as the activation function.

IV. Final Model

To conclude, the best score obtained thanks to these techniques is 25.68 . Indeed, when running the Model 3, with the following parameters:

- Learning rate : $\mathbf{1} \times \mathbf{10}^{-3}$
- Learning weight: **0**
- Number of Epochs : **10**
- Optimizer : **Adamax**
- Activation function : ***ReLU***

Model 3 was trained for 10 epochs, with a batch size of 100. Training was done using the online platform *Kaggle*, whose GPUs are *NVidia K80*s. Total training time is around 10 minutes. On CPU, one epoch takes around 15 minutes (MacBook Pro, 2.3 GHz Intel Core i5 dual core).

V. Implemented denoiser in action

In order to understand how the model performs, we compared by visualizing the different images : the noise image, the image returned by our model and the clean image as we can see in the figure 4. The results of our denoiser are quite impressive.

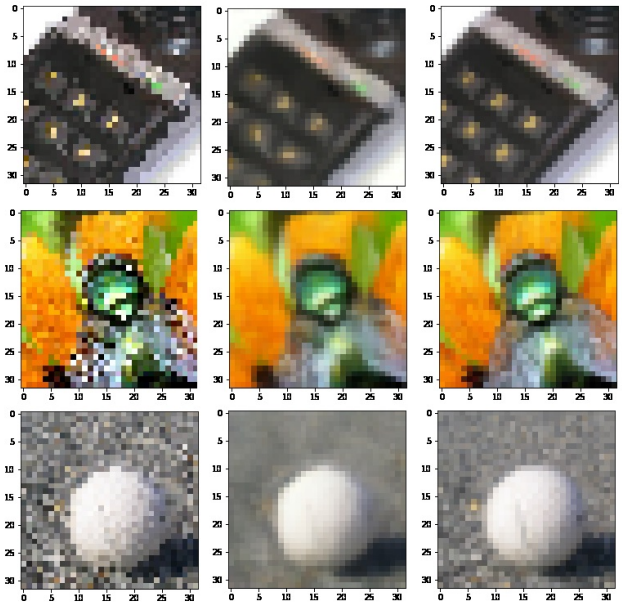

Fig. 4: *Result of the Model on 3 images selected from the data set. For each image, we plot the image given in input with the noise (left), the image obtained by the model (middle) and the image without noise (right)*

VI. Conclusion

Through this project, we constructed different models in order to optimise our PSNR score. Each of these models have been trained to denoise images in the fashion of a Noise2Noise model. As expected, U-Net architecture yields the best scores, that's why the only models presented in this report follow a U-Net architecture. The best model remains Model 3, whose architecture is based on the paper [1], with a PSNR of 25.68.

References

[1] Jaakko Lehtinen et al. “Noise2Noise: Learning Image Restoration without Clean Data”. In: *CoRR* abs/1803.04189 (2018). arXiv: 1803.04189. URL: http://arxiv.org/abs/1803.04189.