EE-559 Mini-project 1: Using the Standard PyTorch Framework

Bastien Le Lan
Data Science
bastien.lelan@epfl.ch

Yuxiao Li
Digital Humanities
yuxiao.li@epfl.ch

Rui Huang
Computer Science
rui.huang@epfl.ch

***Abstract*—In this first mini-project, we used PyTorch framework to build build a Noise2Noise image denoising network. In the first place, we experimented with several different architectures and evaluated their performances to get the best architecture. Then, we explored the various parts of the hyper parameters tuning, including learning rate, mini batch size, activation function, loss function, and weight initialisation.**

I. Introduction

A. Problem Statement

Image denoising has been one of the greatest beneficiaries from development of deep learning algorithms, it is useful in many fields both in public and scientific community. The construction of an image denoising model with a deep learning model[1] would traditionally take two data sets: one noisy image set and one ground truth image set. and also have an objective noise level to get rid of. However, it could often be the case that there is no ground truth data. Therefore, traditional methods cannot be used in this case. In [2], the authors applied statistical reasoning to show that it is possible to learn to restore images by only looking at corrupted examples. In this project, we created such denoising model to to be able to get clean image with only noisy images pairs. We implemented this with a PyTorch framework called noise2noise model, and explored the architecture and parameter tuning to obtain the best denoising model possible.

B. Data

The data at our disposal was in two .pkl files: train_data.pkl and val_data.pkl. We extracted from train_data.pkl two tensors of size 50000 x 3 x H x W, corresponding to 50000 noisy pairs of images, each of the them corresponding to a downsampled, 32 by 32 pixelated images. We are used train_data.pkl to train a denoising network. We extracted from the second file val_data.pkl two tensors of size 1000 x 3 x H x W, corresponding to 1000 pairs of noisy and clean images.

II. Methodology

A. Evaluation metric

To track the performance of the model, the evaluation method that we adopted is Peak Signal to Noise Ratio (PSNR), a quantity expressed in decibel that measures the logarithmic ratio between the maximum power of a signal compared to its noise (MSE). PSNR is commonly used to quantify reconstruction quality of images, and a clearer image usually corresponds to a higher PSNR value. The aim of the hyperparameter tuning and model tweaking was to get the best PSNR, thus the best visual quality of denoised image.

B. Model Selection

1) U-Net: Originally designed for Image Segmentation[U-Net: Convolutional Networks for Biomedical Image Segmentation], U-Net has achieved good performances on other tasks, including Image Denoising. Its architecture could be generally regarded as an encoder followed by a decoder, with a concatenation between the higher resolution feature maps from the encoder network and the upsampled features maps in the decoder network, which is designed for better learning the representations with the following convolutions. Here, we have adopted U-Net structure similar to what is introduced in [2].

2) Benchmark: In order to understand if U-Net has a good performance, we implemented two benchmarks: The architecture of Benchmark 1 is Conv2d + ReLU + Maxpool2d + Conv2d + ReLU + Nearest Neighbor Upsample(scale_factor=2) + Conv2d + ReLU, Conv2d has kernel size of 3, padding 'same', and channel for hidden units equals 48. The architecture of Benchmark2 is Conv2d + ReLU + Conv2d +ReLU + Upsampling + ReLU + Upsampling + Sigmoid, Conv2d has kernel size of 2 and stride of 2, and channel for hidden units equals 48.

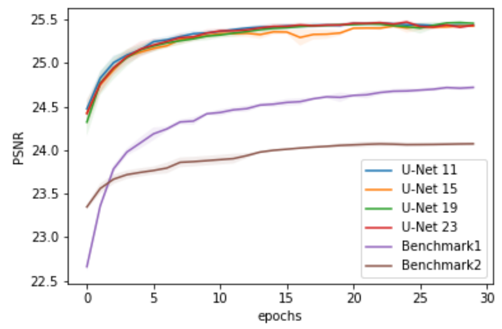


Fig. 1. Performance of different models

We can see that with U-Net architecture, the denoising model has a better performance regarding to PSNR score. And for the number of layers, U-Net with more layers generates slightly better performance. However, the U-Net that we have adopted is slightly different from the one proposed in the original paper. In the next section 'Hyper Parameter Tuning', we will using empirical result to support our choice.

III. Hyper Parameter Tuning

A. Learning Rate of Optimizer

In order to understand the influence of learning rate over the training process, we experimented with three different learning rate: 0.01, 0.001, 0.0001.

When the learning rate is too high, the object function is prone to get stuck in a local minima, therefore we could only obtain a suboptimal solution. When the learning rate is too low, it requires more updates to reach the minimum point.

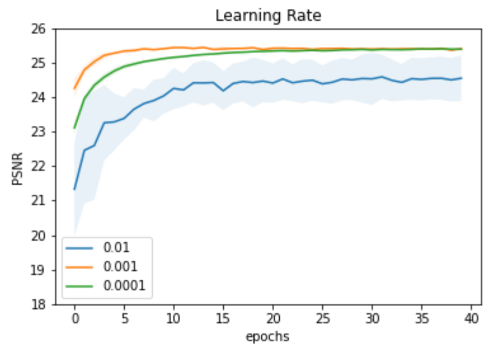


Fig. 2. Learning Rate of Optimizer

B. Mini Batch Size

One of the hyperparameter that has to be tuned is the mini batch size. A smaller batch size learns very quickly, nevertheless, the model bounces around the global optima and leads to a worse result. While a larger batch size took too much time to converge. We therefore ran the model using batch size 40,50,80,100,200.

From the plot, we can see that with smaller batch sizes, the training converges quicker, and there's no obvious difference among batch sizes smaller than 100. However, when the batch size is big (200), it takes longer to converge, and the PSNR is not as good as training with smaller batch sizes.

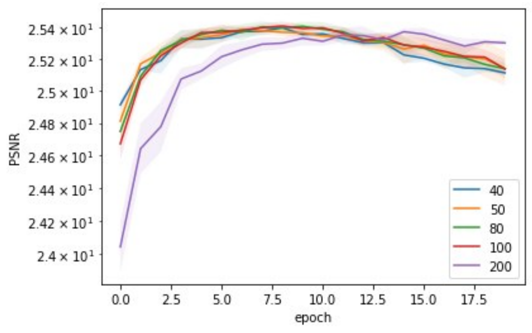


Fig. 3. Mini Batch Size

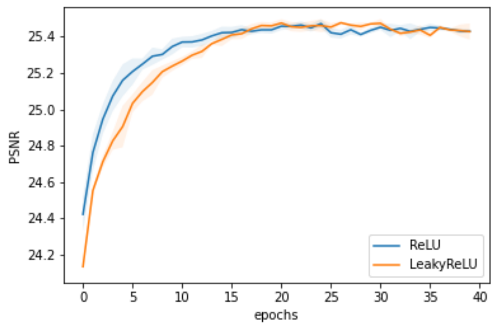


Fig. 4. Activation Function.

C. Activation Function

Leaky Rectified Linear Unit, or Leaky ReLU, is a variant of ReLU. Unlike ReLU, it has a small slope for negative values instead of a flat slope. We have tested these two activation functions to see if one has advantages over another.

The result shows that LeakyReLU doesn't get a better PSNR value for the images, while it takes longer to run. Therefore, we decide to forward with ReLU activation function.

D. Loss Function

We tried to run the model using Mean Square Errors (L2 Loss) and Least Absolute Deviations (L1 Loss), the latter is better suited for dealing with outlisers while L2 Loss is perfered in most cases.

Empirical result showed that the model using L2 Loss generates a much better PSNR. Therefore, we decide to adopt L2 Loss for the training.

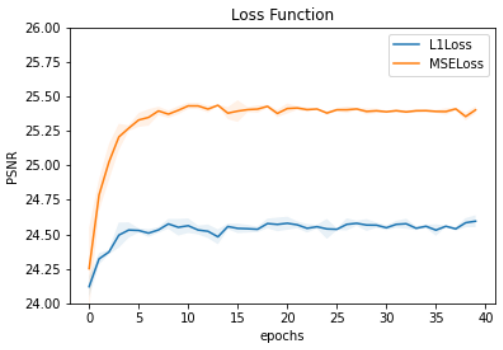


Fig. 5. Loss Function

E. Weight Initialisation

We experimented different weight initialisation method during the training process: Xavier initialisation, Kaiming initialisation, and blank control group.

From our empirical experiments, we find out that even without weight initialisation, the training still converges quickly, but not as good as the other two methods. For Xavier initialisation, the performance of converging rate is the best, however it is less stable. As for Kaiming initialisation, it took longer to plateau, but the stability is the better than the other two methods.

The Kaiming initialisation is initialising weights normally distributed with mean zero and standard deviation defined as a gain divided the square root of the input, and the Xavier initialisation is defined as the square root of two over the sum of inputs and outputs.

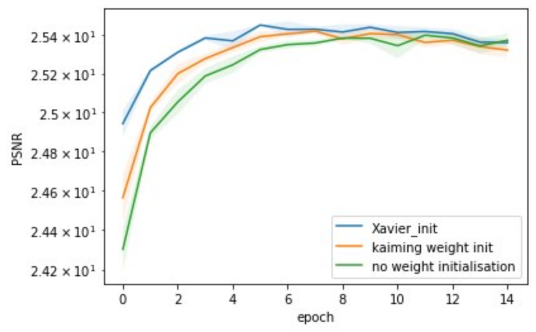


Fig. 6. Weight Initialisation

IV. Results

In this section, we present the results of our Noise2Noise model. After the hyper parameters tuning, we obtained a PSNR around 25.48±0.02 dB, which is satisfying considering that the PSNR from our benchmark reaches around 24.5 dB and a black image has a PSNR of around 6 dB.

We also did a qualitative analysis of the result. According to Figure 7, we can observe that our Noise2Noise model does a good job denoising the noisy images in general. From the first sight, the model could remove the most part of the noise. However, as we can observe in the fourth image, the model tends to over soften some local pattern.

:

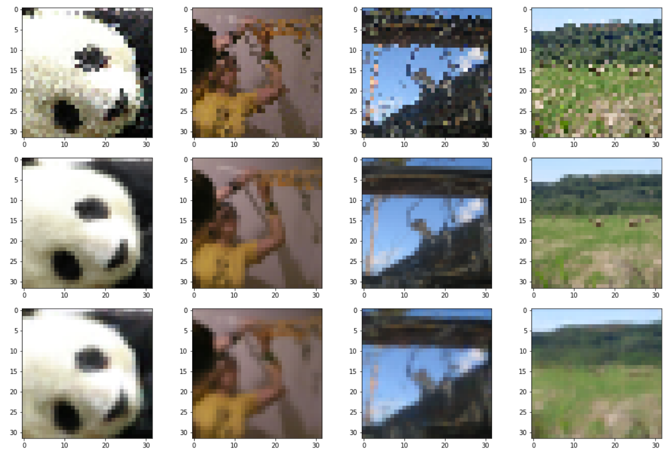


Fig. 7. The first row is noisy image, the second row is clean images, the third row is denoised images.

V. Conclusion

This mini-project allowed us to build an image denoiser using PyTorch framework, and we managed to get a decent result. We experimented two architectures: U-Net and one based on trial and error. After we compared the PSNR of these two architectures, we decided to forward with U-Net architecture. Afterwards, we explored the tuning of hyper parameters to understand the role played by different hyper parameters during the training process, meanwhile, we also found the best combination of hyper parameters that gives us the best performance.

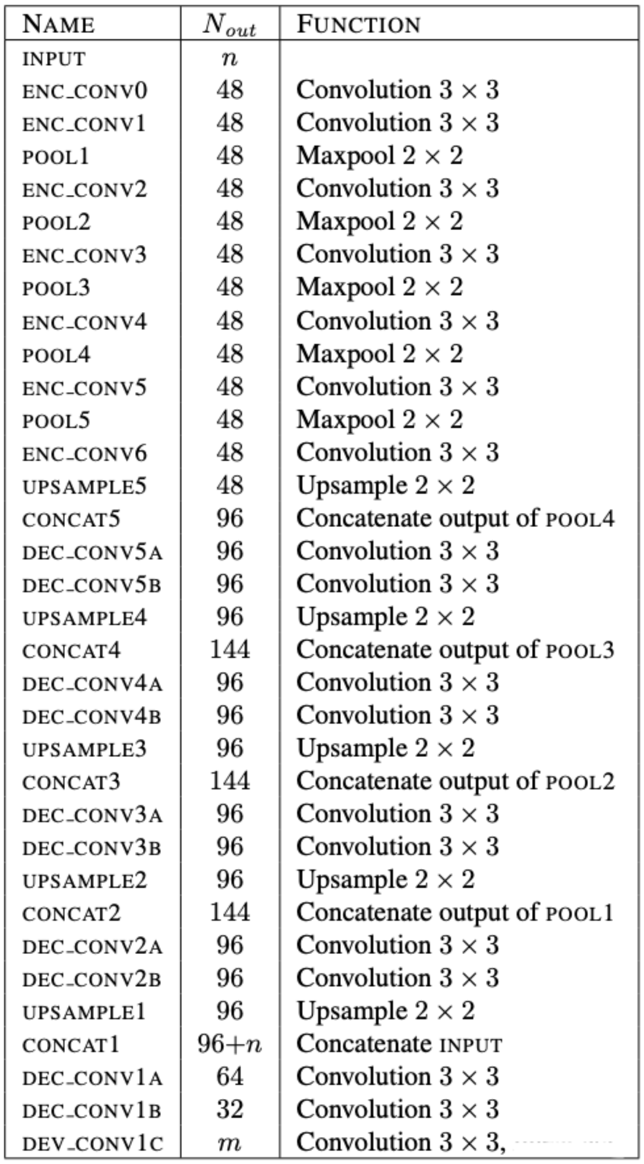
This is even more impressive keeping in mind that this model was not trained using some ground truth images but only by using pairs of noisy images. However, our model did not manage to get to a perfect reconstruction of the image we wanted, showing that further improvement could be done. One improvement path that could be taken easily is to improve the image dimension as well as the training set size.

References

[1] S. Y. C. Shakarim Soltanayev, "Training deep learning based denoisers without ground truth data," 2018, https://papers.nips.cc/paper/2018/file/c0560792e4a3c79e62f76cbf9fb277dd-Paper.pdf.

[2] J. H. S. L. T. K. M. A. T. A. Jaakko Lehtinen, Jacob Munkberg, "Noise2noise: Learning image restoration without clean data," 2018, arxiv cornell university.

VI. Appendix



Name	N_{out}	Function
INPUT	n	
ENC_CONV0	48	Convolution 3×3
ENC_CONV1	48	Convolution 3×3
POOL1	48	Maxpool 2×2
ENC_CONV2	48	Convolution 3×3
POOL2	48	Maxpool 2×2
ENC_CONV3	48	Convolution 3×3
POOL3	48	Maxpool 2×2
ENC_CONV4	48	Convolution 3×3
POOL4	48	Maxpool 2×2
ENC_CONV5	48	Convolution 3×3
POOL5	48	Maxpool 2×2
ENC_CONV6	48	Convolution 3×3
UPSAMPLE5	48	Upsample 2×2
CONCAT5	96	Concatenate output of POOL4
DEC_CONV5A	96	Convolution 3×3
DEC_CONV5B	96	Convolution 3×3
UPSAMPLE4	96	Upsample 2×2
CONCAT4	144	Concatenate output of POOL3
DEC_CONV4A	96	Convolution 3×3
DEC_CONV4B	96	Convolution 3×3
UPSAMPLE3	96	Upsample 2×2
CONCAT3	144	Concatenate output of POOL2
DEC_CONV3A	96	Convolution 3×3
DEC_CONV3B	96	Convolution 3×3
UPSAMPLE2	96	Upsample 2×2
CONCAT2	144	Concatenate output of POOL1
DEC_CONV2A	96	Convolution 3×3
DEC_CONV2B	96	Convolution 3×3
UPSAMPLE1	96	Upsample 2×2
CONCAT1	$96{+}n$	Concatenate INPUT
DEC_CONV1A	64	Convolution 3×3
DEC_CONV1B	32	Convolution 3×3
DEV_CONV1C	m	Convolution 3×3,

Fig. 8. Architecture of the U-Net