Mini-project 1: PyTorch implementation of Noise2Noise model

Cezary Januszek, Tikhon Parshikov, Michael Roust
Deep Learning, EE-559, EPFL, Switzerland

***Abstract*—In this paper, we develop a model to solve Noise2Noise problem[1]. The model was created using modules provided by PyTorch framework. Building such a model allows us to train our skill in building neural networks as well as understanding the basic aspects of Noise2Noise.**

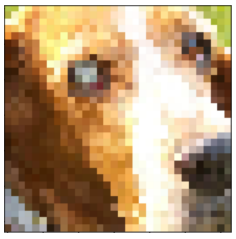
I. Introduction

The main idea of Noise2Noise is that (under certain conditions) noisy images have values distributed around the actual values. This property, which we will explain more in depth in below, allows us to train a model for denoising images using only noisy input data. Thus, our task for this mini-project is to develop an optimal architecture for the given task. Since, we are working with image data a convolutional neural network is the natural way to go. Which, we will build using the well known PyTorch library[2].

In the process of finding the optimal model we experimented with various different models and architectures and we report our findings here. Our final model can be trained in only about 10min on a single computer with a mid-range GPU. Yet, achieves a good performance.

II. Data set

The provided data set is divided to train and validation parts. The train part consists of 50000 pairs of noisy images with 3 channels and size of $32*32$. The elements of the pair are got by downsampling and pixelating of the same initial image. The validation part consists of 5000 pairs where the first element is a noisy image and the second one is a clean image.



(a) Noisy image 1

(b) Noisy image 2

III. Noise2Noise explanation

We'll begin by showing the mathematical intuition behind Noise2Noise. As stated in the Noise2Noise paper denoising can be achieved without clean samples, if the noise is additive and unbiased.

Considering ϵ and δ two unbiased and independent noises. We have

$$\begin{aligned}&\mathbb{E}[||\phi(X+\epsilon;\theta)-(X-\delta)||^2]\\&=\mathbb{E}[||\phi(X+\epsilon;\theta)-X||^2]+\mathbb{E}[||\delta||^2]\end{aligned}$$

Hence

$$\begin{aligned}&\text{argmin}_\theta\,\mathbb{E}\left[||\phi(X+\epsilon;\theta)-(X+\delta)||^2\right]\\&=\text{argmin}_\theta\,\mathbb{E}\left[||\phi(X+\epsilon;\theta)-X||^2\right]\end{aligned}$$

Basically, the equations above tells us that by training a model to minimize the distance between two noisy images (with unbiased independent noises) it would be statistically equivalent to training a model that minimizes the distance between noisy and clean images.

As a result effective denoising models can be trained without any clean data. This is the approach the Noise2Noise paper uses and we will use.

IV. Performance metric

For tasks where we need to estimate the quality of image reconstruction usual metrics such as accuracy are not very informative. In these cases it is better to use such a specific metric as *Point signal-to-noise ratio* that represents a measure of peak error. In this project we use a simplified version of this metric calculated by:

$$\text{PSNR}(im1,im2)=-10\cdot log_{10}(\text{MSE}(im1,im2)+10^{-8}),$$

where

$$\text{MSE}(im1,im2)=\frac{1}{N}\sum(im1-im2)^2$$

V. Model architecture and experiments

We used and iterative approach to find the best models and model components. Initially, we start with very simple models built with simple modules. Then, we progressively increased the complexity of each model and analysed results. Approaches that yielded good improvements in results were then carried over to more complex models. Whereas, approaches that decreased performance were discarded.

An important constraint in this project was the limited training time available. Since, our final model is required to have a training time of 10 minutes on a machine with a mid-range GPU we had to extract the maximum performance from a relatively simple model.

Below are descriptions of the first few models we used:

```
model1 = nn.Sequential(
    nn.Linear(32, 128),
    nn.Linear(128, 32)
)

model2 = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3,
        padding=3),
    nn.ReLU(),
    nn.ConvTranspose2d(32, 3,
        kernel_size=3, padding=3),
    nn.ReLU(),
)

model2v1 = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3,
        padding=3, bias=False),
    nn.ReLU(),
    nn.ConvTranspose2d(32, 3,
        kernel_size=3,
        padding=3, bias=False),
    nn.ReLU(),
)

model2v2 = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3,
        padding=3, bias=False),
    nn.LeakyReLU(),
    nn.ConvTranspose2d(32, 3,
        kernel_size=3,
        padding=3, bias=False),
    nn.ReLU(),
)

model3 = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3,
        padding=3),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.ConvTranspose2d(32, 3,
        kernel_size=3, padding=3),
    nn.ReLU(),
)
```

Initially, we start with `model1` which only includes 2 linear layers. This already achieves a PSNR of nearly 24. Obviously, this isn't the approach to use as image data has spatial properties which convolutional layers are great at capturing. Next, `model2` shows us that this is indeed that case as we achieve a PSNR of nearly 25.

In our next iteration `model2v1` we decided to remove the bias term from out convolutions. The reasoning behind this is that we are working with unbiased noises and thus having a bias term in our model should either have no effect or only reduce the performance of our model. This hypothesis is confirmed as we see a negligible change in performance compared to `model2`.

Table I: Model training times and performance. Epochs=30, Mini-batch_size= 100, Adam optimizer lr=1e-3

	Training time (minutes)	Test set PSNR
`model1`	2	23.78
`model2`	2	24.92
`model2v1`	2	24.98
`model2v2`	2	24.94
`model3`	3	24.52
`model4`	3	25.01
`model5`	7	24.87
`model6`	10	25.44
`model_pooling`	5	25.11
`model_final`	7	25.45

Our next experiment in `model2v2` was to introduce a LeakyReLU function in the middle of our model. This could help capture some extra information from the first to the second layer by allowing negative values to pass through the first convolutional layer. Also, LeakyReLU tends to reduce training times as it doesn't have zero-slope parts. Albeit, it seemed to produce a negligible change for this model. However, as we will see later we achieved a small improvement by using it in our final model. The greater complexity of the model must have allowed LeakyReLU this time have a non-negligible effect.

We attempted to use batch normalization in `model3` as this can sometimes help in learning models. However, we see a significant decrease in PSNR of 0.5db and slightly longer training time. We suppose batch normalization provided negative results as we are dealing with unbiased noise for which no such normalization would be needed. Instead batch normalization only deviates out model from the correct behaviour when training inputs aren't perfectly centered around the correct value.

The next two models `model4` and `model5` correspond to an identical structure to `model2` but with wider convolutional layers of 64 or 128 channels respectively. We can observe a significant increase in training times of these models as the number of parameters grows and that more than 64 channels in the hidden layers appears to be counter productive.

In `model6` we simply double that number of convolution layers to 4. The potential descriptivity of the model, training time and performance are significantly increased as expected.

Finally, we inspired ourselves for the architecture of the U-Net[3] and attempted to use pooling and upsampling operations. However, this unfortunately decreased the performance of our model. We assume this could be explained in the following manner. The U-Net architecture is designed for segmentation. That means the model needs to pick up on

features such as shapes or objects in the input image. This is achieved with a set of convolutions and pooling. Then, in order to produce an output with a size proportional to the input the pooling operations are "so to say" reversed with a set of convolutions and upsampling operations. Our hypothesis, is that the task of our model here isn't to pick up on larger features of the input but to just denoise our image and thus that could explain why a U-Net like architecture may not be the best for our task.

Our final and most effective model combines all the most effective methods we used so far into one model. 6 convolutional layers with a number of channels in hidden layers ranging from 48 to 64 and LeakyReLU (with a negative section slope of 0.01) at the end. With 30 epochs this model achieves a PSNR of 25.45db.

VI. Optimizing hyper-parameters

We tried to use different optimizers such as SGD or Adam. Unlike SGD which has a fixed learning rate, Adam has a dynamic learning rate with adjustments based on the first two moments of the gradients. As observed the results of the Adam optimizer are generally better than most other optimization algorithms, have faster computation time, and require fewer parameters for tuning.

We further tried to optimize next parameters:

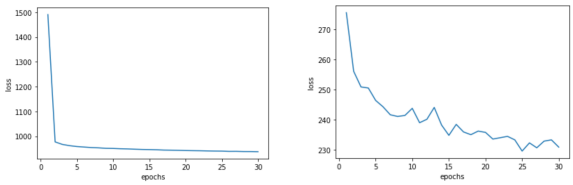
- learning rate
- batch size
- slope (for the last activation function, LeakyReLU)

To do this we ran a grid search. During this tuning we noticed several points that should be mentioned:

1) Learning rate higher than 0.01 does not allow to achieve good results. It can be caused by the fact that for relatively high learning rate it is more possible converge to a suboptimal solution.
2) As seen with the functioning of SGD a batch size of 1 is ideal but to leverage hardware parallelism larger batch sizes are used in practice. We settled for a batch size of 100 to keep training times on GPU machines optimal.
3) Oddly enough, a LeakyReLU with negative section slope closer to -0.5 leads to better results. A positive slope appeared to be counter productive as outputs (an image) should be in the range [0,255] and not be negative. Our final model uses a more conservative value of -0.1.

Finally, we found several combinations of these parameters that allow our model to perform better on the given data set than with standard parameters.

VII. Results



(a) Train losses (b) Test losses

In order to obtain our final model we trained the most effective architecture we found for more epochs. As we can see in the previous graphs train losses converge very rapidly. Whereas, test loss continues to decrease even after 30 epochs. It should also be noted that the test and train losses can't be directly compared as we train on a set of only noisy images and but test on a set of noisy and clean validation images.

Finally, `model_final` achieves PSNR of 25.57 when trained for 50 epochs (Adam optimizer lr=1e-3, Mini-batch_size=100, LeakyReLU_negative_slope=-0.1).

Below is an example to show the model's in action:

(a) Noisy image (b) Cleaned image (c) Original image

VIII. Conclusion

Overall, we are happy with the result of this project. We maximized the performance of a complexity constrained CNN. We explored a number of different architectures as well as hyperparameters and found the optimal ones. The model performs significantly better than baseline (23/24 db). We can clearly see its performance on the above images.

References

[1] J. Lehtinen and Munkberg, "Noise2noise: Learning image restoration without clean data," 2018. [Online]. Available: https://arxiv.org/abs/1803.04189

[2] Paszke, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[3] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597