Noise2Noise project report 1

AGNAOU Zineb, JAAKIK Marouane, SHENG Haoyu

May 27, 2022

Abstract

In our deep learning course (EE-559), we implement a Noise2Noise model initially presented in the article (Lehtinen et al. (2018)). The idea is the following, from two corrupted images, we seek to restore the denoised image. To do this, the model is built according to a U-net structure consisting of convolution layers followed by deconvolution. The final selected model has a $PSNR$ score of 25.48 dB.

1 Introduction

To conduct our project, we first focused on the framework of our plan and started with a three-layer model. Once this stage was achieved, we used the original model as a benchmark and optimised it according to the data and information provided to us. We first generated new data from the available data to make our domain more robust. We then tried several error evaluation methods. After that, we tuned the model with several optimizers and different values for the hyperparameters.

2 Data Description

We start with data of dimensions two tensors of the size $(50,000 \times 3 \times 32 \times 32)$. This contains 50,000 pairs of tensors that are both compromised by a noise that we will assume throughout our project to be Gaussian. We divide this database into train and validation datasets. The first part (constituting 80%) will be used to train our model, the remaining data (constituting 20%) is left for validation. This division is done randomly. We also have at our disposal two tensors of dimensions $(1,000 \times 3 \times 32 \times 32)$ which are composed of noisy images as well as an image which has been denoised. We reserve this dataset for testing the performance of our models. For this we use the following recommended matrix: $PSNR(x, y) = -10log_{10}(MSE(x, y) + 10^{-8})$ where $MSE(x, y) = \sum_{i=1}^{N}(x_i - y_i)^2$ for $x = (x_1, x_2, ..., x_N)$, $y = (y_1, y_2, ..., y_N)$.

3 Data Augmentation

Data augmentation in the context of data analysis is one technique used to increase the amount of data by adding slightly altered duplicates of existing data or newly created synthetic data from existing data. It is useful for reducing overfitting and assisting us in achieving a more robust model. For this project, we performed this augmentation using the `torchvision.transforms` package of the `pyTorch`(Paszke et al. (2019)) package. For this, we have combined five transformations. The first one is done by the function `RandomInvert` which inverts the colours of the image with a probability p. In our case, we chose $p = 0.5$. Next, we use the function `CenterCrop` which crops the given image to the centre. The output image has a dimension of $(1 \times 3 \times 16 \times 16)$ after. For our third transformation, we transform our image to grey scale with a probability of 0.5. For this we use the function `RandomGrayscale`. After that, we resize our images to their original size. To do this, we use the function `Resize`. Finally, we use the function `RandomAdjustSharpness` which adjusts the sharpness of the image with a probability p. In our case, we chose $p = 0.5$.
We randomly select 10,000 data and perform the transformations on them before merging them with the rest of the data. In the end, we obtain tensors of dimension $(60,000 \times 3 \times 32 \times 32)$ to train our model.

4 Model

To first check that our structure and functions work, we experimented with relatively simple models with few layers. Once this was validated, we decided to start with the model presented in the article (Lehtinen et al. (2018)) as a benchmark and optimise it to get better results.

We use U-net (Ronneberger et al. (2015)) to implement a similar model presented in the paper (Lehtinen et al. (2018)) since U-net contains an encoder network followed by a decoder network. All convolutional layers in our network with (3×3) kernel, padding size 1. We used leaky ReLU with 0.1 negative slope as our activation function for all layers except the last layer. We use transposed convolutional layer with stride size (2×2) instead of nearest neighbour upsampling in the model as we think transposed convolutional layer perform both upsamples and convolutions. We initialise weights of our network following He et al. (2015) which is the same approach adopted by the original paper (Lehtinen et al. (2018)). To ensure that our output images' pixels are within the range [0, 255], we applied Hardtanh with minimum value of the linear region range 0 and maximum value of the linear region range 255 as the activation function in the last layer. Figure 3 shows the structure of our network model.

5 Optimization

To optimise our model, we proceeded as follows. We first opted to optimise on our batch size, for this we compared the values 8, 16, 32, 64 and 128. We noticed that the running time was too long for the small batch sizes and given our ten minute constraint for the project scoring, we decided to continue only with 32, 64 and 128. We then wanted to try different optimizers. We tried the stochatic gradient descent (SGD), the Root Mean Squared Propagation optimizer (RMSProp) and finally we tried the adaptive moment estimation optimizer (Adam). To improve our methods, we tune the hyperparameters. We then compared the performance of the models with an l2-regularisation with a $\lambda \in [10^{-4}, 0.001, 0.01, 0.1, 1, 5]$. We also refer to λ as weight decay (wd) throughout the report. Regarding the learning rate (lr), we saw that the results were really bad with large factors and we set ourselves a rate of 0.001 for Adam and RMSProp and use 10^{-6} for SGD. Considering of the ten minutes' time constraint, we only train the models for 20 epochs.

6 Results

Looking at results in Table 2, we could find that almost all models' $PSNR$ scores are above 25dB. Performances of models with same hyperparameters which are trained on different dataset - original dataset and dataset after data augmentation are similar in terms of $PSNR$ scores. Training and testing models with batch size 128 are faster than models with batch size 32 and 64. Models with weight decay values 0.001 or 0.01 are usually perform better. Since weight decay 1 and 5 are too high values and high weight decay values penalize high value weights of network too much and even prevent it from learning.

The model with batch size 128, weight decay 0.1 and learning rate 0.001 trained on original dataset perform best in terms of $PSNR$ score (25.62 dB) and a relative short training and testing time (7 minutes 26 seconds). However, considering the fact that the dataset after augmentation contains more data points (10,000 more than original data points) and data augmentation could prevent the neural network from learning irrelevant patterns, and further boost overall performance of the network. We chose the model with batch size 128, weight decay 0.01, learning rate 0.001 as our final best model. Compared with the best model we have trained on the original dataset, our picked best model trained on the dataset after data augmentation with $PSNR$ score 25.48 dB and needs 8 minutes and 28 seconds to train and test.

To visualise the training and validation losses changes for different batch sizes, we plot losses for models with different batch sizes with learning rate 0.001 and weight decay 0.01 training on the dataset after data augmentation as shown in the Figure 1. Both training and validation losses fluctuates up and down, but the general trend is that losses keep decreasing. Since we only trained our models for 20 epochs, the fluctuations of losses are normal at the beginning of the training periods. Besides, we notice that models with smaller batch sizes have larger extent of fluctuations in both training and testing losses as updating the weights are more noisy with smaller batch sizes.

We also tried to find the best optimizer and results are presented in Table 1. Originally, we want to compare performance of models with same learning rate and weight decay but different optimizers. However, SGD optimizer with learning rate 0.001 always return NAN loss as it diverge into infinity. Then we change the learning rate of SGD optimizer to much lower learning rate. Based on the $PSNR$ scores, the model with Adam optimizer perform much better than models with RMSProp and SGD optimizers. Since the learning rate of SGD is much smaller than learning rate of Adam, the model with SGD optimizer needs longer time to converge. However, there are only 20 epochs. Compared with RMSProp, Adam not only keeps an exponentially decaying average of past squared gradients but also stores an exponentially decaying average of past gradients. Therefore, Models with Adam optimizer have a much better performance than those with SGD and RMSProp in our experiments.

7 Final Model

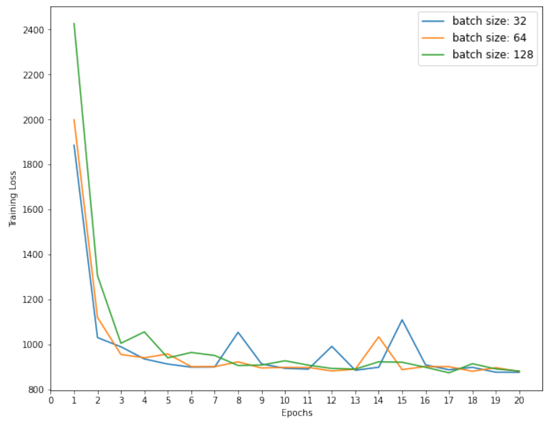
We compared the denoised image produced by our final model with the clean image, and original image in Figure 2. The Panel (a) represents the original noised image, the Panel (b) is the image resulting from our model and finally the Panel (c) represents the clean image provided. We can see that though our model is able to remove some noises in the original image, our image is clearly blurrier than the original image.

8 Conclusion

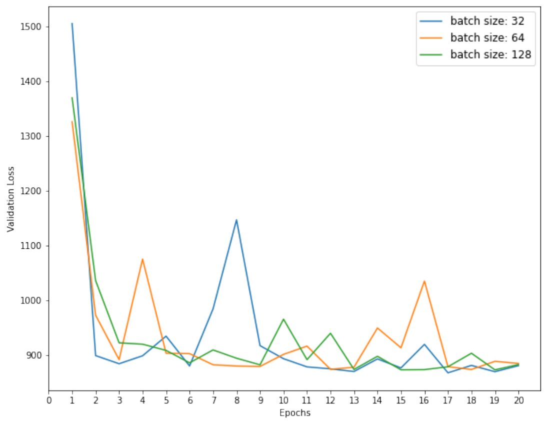
The first goal of this project was to develop a model that could denoise images without any access to the clean image. Over the course of our project, we experimented with different ways to improve our model. We discovered that the Adam optimizer had overall better performances. We were able to experiment with different hyperparameter values. To further expand our data, we also performed a data augmentation on a random sample to counteract the overfit. A possible implementation for the future would be to do the augmentation at the batch level, as opposed to the very beginning of the code. Furthermore, the ten minute rule was our main constraint as it greatly dictated the complexity of our model but also the number of epochs we could train our model on. We have also trained our model on the MSE metric as a loss function but are evaluating it on $PSNR$. As a future improvement, we wish to obtain a model that is more consistent. Finally, we have considered that the noise follows a Gaussian distribution. We may also consider other types of noise.

9 Note

Note that we took inspiration from the pytorch tutorials (Inkawhich; Heintz) and the website (Sharma; Litalien (2020)) to build our model. We used the website (Ltd) to created our Figure 3.

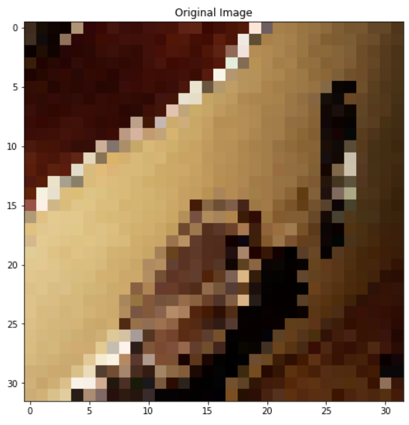


(a) Training losses of different batch sizes

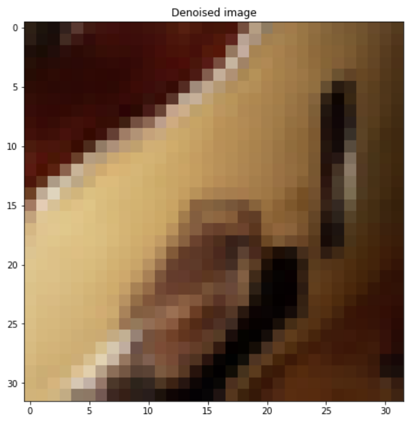


(b) Validation losses of different batch sizes

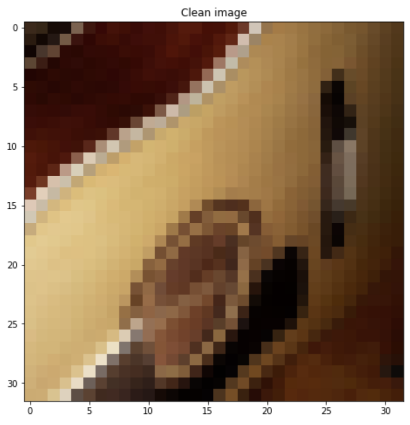
Figure 1: Training and validation losses for different batch sizes with model lr=0.001 wd=0.01



(a)



(b)



(c)

Figure 2: Example of results. From left to right, Panel (a) represents the original noised image, Panel (b) represents the image resulting from our model and finally the Panel (c) represents the clean image provided.

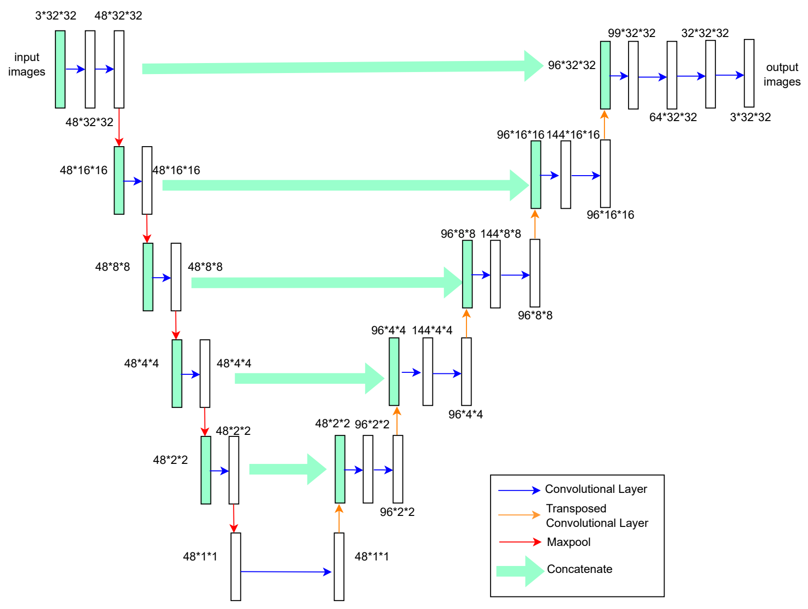


Figure 3: Structure of our Final model. We start from left to right, up to down. The dimension at each time point is mentioned on top of the cell. The green arrow represent concatenation, the blue arrow represent convolutional layers, red arrow represent maxpooling and the orange arrow represent Transposed Convolutional layers.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[2] B. Heintz. Training with pytorch. `https://pytorch.org/tutorials/beginner/introyt/trainingyt.html`.

[3] M. Inkawhich. Saving and loading models. `https://pytorch.org/tutorials/beginner/saving_loading_models.html?highlight=eval`.

[4] J. Lehtinen, J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, and T. Aila. Noise2noise: Learning image restoration without clean data, 2018. URL `https://arxiv.org/abs/1803.04189`.

[5] J. Litalien. noise2noise-pytorch. `https://github.com/joeylitalien/noise2noise-pytorch`, 2020.

[6] J. Ltd. App diagrams. `https://www.diagrams.net/`.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[8] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[9] P. Sharma. Build an image classification model using convolutional neural networks in pytorch. `https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/`.

A Tables of results

In this section we present some of our results. The results are presented in tables. The tables contain the following columns: The optimizer where we specify the optimizer used, the wd, the lr, the batch size, the $PNSR$ score, the time which refers to the training and testing time, the number of epochs and whether there the model was trained on augmented data.

optimizer	wd	lr	Batch size	$PSNR$	time	epochs
Adam	0.01	0.001	128	25.48	8m 28s	20
RMSprop	0.01	0.001	128	7.15	8m 23s	20
SGD	0.01	0.000001	128	6.83	8m 23s	20

Table 1: Table of comparing different optimizers results.

optimizer	wd	lr	Batch size	$PSNR$	time	epochs	Data augmentation
Adam	0.0001	0.001	32	25.52	12m 3s	20	No
Adam	0.001	0.001	32	25.55	12m 6s	20	No
Adam	0.01	0.001	32	25.60	12m 5s	20	No
Adam	0.1	0.001	32	25.40	12m 1s	20	No
Adam	1	0.001	32	25.06	11m 57s	20	No
Adam	5	0.001	32	25.50	11m 59s	20	No
Adam	0.0001	0.001	64	25.49	9m 39s	20	No
Adam	0.001	0.001	64	25.48	9m 40s	20	No
Adam	0.01	0.001	64	25.56	9m 39s	20	No
Adam	0.1	0.001	64	25.60	9m 40s	20	No
Adam	1	0.001	64	25.39	9m 41s	20	No
Adam	5	0.001	64	25.07	9m 41s	20	No
Adam	0.0001	0.001	128	25.41	7m 10s	20	No
Adam	0.001	0.001	128	25.46	7m 21s	20	No
Adam	0.01	0.001	128	25.53	7m 25s	20	No
Adam	0.1	0.001	128	25.62	7m 26s	20	No
Adam	1	0.001	128	25.41	7m 24s	20	No
Adam	5	0.001	128	24.99	7m 20s	20	No
Adam	0.0001	0.001	32	25.51	10m 4s	20	Yes
Adam	0.001	0.001	32	25.45	10m 6s	20	Yes
Adam	0.01	0.001	32	25.51	10m 6s	20	Yes
Adam	0.1	0.001	32	25.46	10m 3s	20	Yes
Adam	1	0.001	32	25.26	10m 3s	20	Yes
Adam	5	0.001	32	25.01	10m 2s	20	Yes
Adam	0.0001	0.001	64	25.44	11m 10s	20	Yes
Adam	0.001	0.001	64	25.47	11m 10s	20	Yes
Adam	0.01	0.001	64	25.40	11m 8s	20	Yes
Adam	0.1	0.001	64	25.45	11m 5s	20	Yes
Adam	1	0.001	64	25.12	11m 7s	20	Yes
Adam	5	0.001	64	25.03	11m 6s	20	Yes
Adam	0.0001	0.001	128	25.39	8m 28s	20	Yes
Adam	0.001	0.001	128	25.40	8m 28s	20	Yes
Adam	0.01	0.001	128	25.48	8m 28s	20	Yes
Adam	0.1	0.001	128	25.41	8m 28s	20	Yes
Adam	1	0.001	128	25.23	8m 28s	20	Yes
Adam	5	0.001	128	25.01	8m 28s	20	Yes

Table 2: Table of comparing batch sizes and weight decay results on orginal dataset and dataset after data augmentation. (Time refers to the training and testing time and we train and test our models on Google Colab GPU)