Deep Learning EE-559 - Mini-project 2

Antoine Olivier Salaün, Henrik Øberg Myhre, Utku Görkem Ertürk
27th of May 2022

***Abstract*—In this project, we built a network that predicts the clean versions of noisy images. This is done without the use of autograd and torch.nn modules. We construct 2D-convolution, Upsampling, activation functions, Stochastic Gradient Descent as an optimizer and Mean Squared Error as a loss function. In the end, the goal is to denoise images using only training data that consists of pairs of noisy images. The DL-model has hence never seen a clean reference image while training, but it still manages to remove noise from images because the noise in the pairs of noisy images is statistically independent and unbiased. A solid understanding of the underlying math was important to create all the necessary modules.**

I. Introduction

Denoising images without the use of clean images is a process called Noise2Noise [1]. When using frameworks like PyTorch and Tensorflow to build and train models, much of the the math and processes involved are hidden from the developer. In this report, we want to provide a deeper understanding of the network struture and modules behind a Noise2Noise-model. We do this by making our own framework with modules that can denoise images using the Noise2Noise-technique.

II. Noise2Noise

See our report "Deep Learning EE-559 - Mini-project 1" for more information and mathematical proof of the Noise2Noise-technique.

III. Building the Modules

A. 2D-Convolution

The convolution takes the following parameters as input when initializing a layer: in_channels, out_channels, kernel_size, stride, padding and dilation. The forward pass uses the dimesions of the input to get the batch size, and the width and height of the image. Both the backward pass and forward pass uses linear algebra and matrix multiplication to calculate the outputs. The forward pass returns the ouput after reshaping the input, and the backward pass returns the gradient with respect to the input. Additionally, the gradients with respect to the weights and biases are stored in the convolutional layer object, as these are needed to update the weights and biases later.

The key aspect of the implementation of the convolution is the `unfold` function. The convolution can be seen as a matrix multiplication

$$O = W \cdot \texttt{unfold}(\mathbf{I}) + \mathbf{b} \tag{1}$$

where O is the output, W is the weight matrix, $\mathbf{I}$ is the intput vector and $\mathbf{b}$ is the bias vector. The trick is that the input is not multiplied as it is (it would be a simple linear layer). It is patched by the kernel. Each combination of kernel squares are expanded by the `unfold` function. Then, the weight matrix multiplication and bias adding are applied. Finally, the `fold` function is used to put it back in place. The main challenge of the implementation is to manage the transposition and reshaping before and after the matrix multiplication.

B. Upsampling

The Upsampling layer is implemented with the use of a 2D-transpose convolution. The transpose convolution takes the same input parameters as the normal 2D-convolution, and is known as both deconvolution and upsampled convolution. This is because it has the opposite effect of the convolution.

C. Activation functions

We have implemented two different activation functions: Sigmoid and ReLU, both of which takes any batch of images as an input.

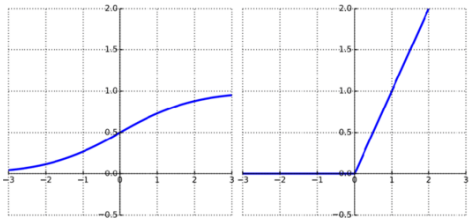
Sigmoid as shown in graph a in figure 1, returns numbers between 0 and 1. Input less than 0 returns output less than 0.5, and input more than 0 returns more than 0.5. See equation 2 for the equation. The input data has to be normalized first.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

ReLU, as shown in graph b in figure 1, returns a number equal to or higher than 0. Input less than 0

returns 0, and input more than 0 returns the input itself. See equation 3 for the equation.

$$ReLU(x) = max(0, x) \tag{3}$$



(a) (b)

Fig. 1: The plots for the activation functions Sigmoid (a) and ReLU (b)

D. Optimizer

The implementation uses Stochastic Gradient Descent (SGD) as the optimizer. The SGD-object keeps track of all the layers in the network and hence also all the gradients with respect to weight or bias, since these are stored in the layers. For updating the weights and biases, it iterates through all the layers and updates the values by substracting the multiple of the gradient by the learning rate. This is because we want to move the values in the opposite direction of the gradient in a step proportional to the est learning rate.

E. Loss-function

Mean Squared Error is used as the loss function. It calculates the distance between predicted values and actual values using the L_2-norm.

IV. Network Structure

The network structure we were tasked to implement is shown in figure 2. It includes 2D-convolutions, Upsampling, and Sigmoid and ReLU as activation functions. The stride-parameter for the convolutional network must, according to the assignmet description, be set to two, and the other parameters are set so that the upsampling and convolutions revert each-others modifications on the image dimensions. The 2D-convolution sets kernel size, stride and padding to 2, 2 and 0 respectivelly, to dvide the height and width by two. The Upsampling does the opposite when using 2D-transpose convolution and hence multiplies the dimensions by two. This is done with kernel size, stride and padding as 3, 2 and 1 respectivelly.

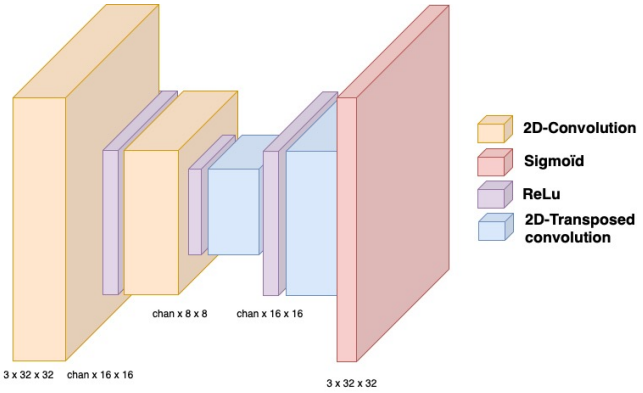


Fig. 2: Network architecture

V. Tuning of Hyperparameters

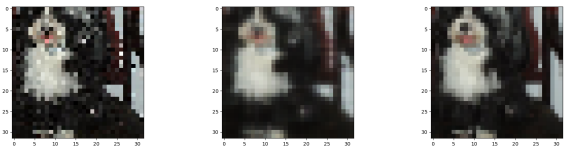
Hyperparameter tuning is done to optimize the custom Noise2Noise-model. In this paper, we tune the hyperparameters batch size and learning rate, in addition to manual network testing with different in and out channel-sizes. The hyperparameters and the values they were tested with are shown in table I. We will test every possible combination to find the values that produce the model with the highest PSNR-score.

Batch sizes:	8	16	32	64
Learing rates:	0,1	0,2	0,4	0.8

TABLE I: Vaues for tuning hyperparameters

VI. Results

The network we were tasked to implement is shown in figure 2 along with the dimensions and channel sizes that proved to be performing good. The resulting scores and losses by this network is shown in table II, for different batch sizes and learning rates. The best performing model has a batch size of 8 and a learning rate of 0.8.



(a) Noised (b) Denoised (c) Clean

Fig. 3: Model demonstration on an example image

We also see in figure 4 that the performance (PSNR-score) is strongly increasing and the training loss is decreasing when the number of hidden channels increases.

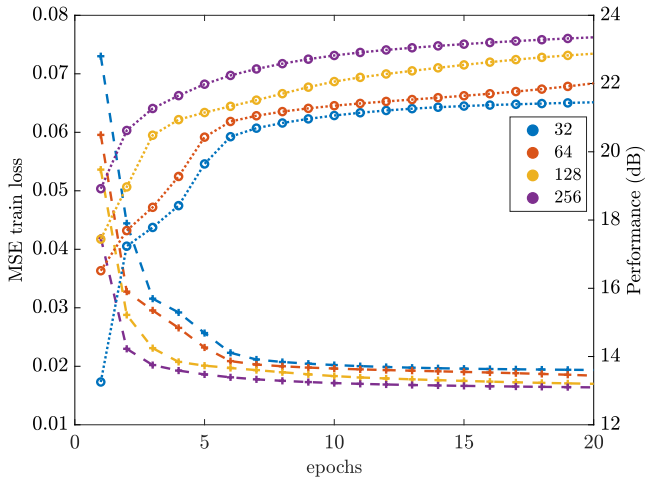


Fig. 4: Perfomance (in dB) and MSE training loss in function of epochs for each number of hidden channels. Circles represent the performance and crosses represent losses

Batch size	Learning rate	Train loss	Performance (dB)
8	0.1	0.01576	23.94
8	0.2	0.01557	24.14
8	0.4	0.01538	24.35
8	0.8	0.01526	**24.49**
16	0.1	0.01598	23.71
16	0.2	0.01578	23.94
16	0.4	0.01557	24.16
16	0.8	0.01540	24.36
32	0.1	0.01638	23.36
32	0.2	0.01600	23.70
32	0.4	0.01581	23.90
32	0.8	0.01557	24.10
64	0.1	0.01694	22.92
64	0.2	0.01636	23.38
64	0.4	0.01602	23.70
64	0.8	0.01586	23.88

TABLE II: Tuning of hyperparameters

VII. Discussion

A. Batch Size

The performance improves and the training loss decreases when the batch size decreases, as can be seen in table II. Further decreasing the batch size would probably improve the model. One reason for this is internal competition within a batch, which is further explained in the mini-project 1 report or in this article [2].

B. Blurring

As seen in 3, the use of a MSE loss tends to predict blurry images. This defect is generalized to all predictions done with this model. It could be solved by using a loss function that take in consideration the sharpness of line. Another improvement could be to train another network to asses the probability of an image to be real. The two networks could be trained in an adversarial architecture.

C. Hidden Channels

In figure 4, we see that more hidden channels perform better, at least up until 256. The resons for this is that it increses complexity and the model can hence find more patterns. The increased complexity is done without increasing the depth of the network, which could have decreased the networks learing speed.

D. Time Limit

The total training time limit was set to ten minutes for this project. The best score for this model with that time limitation is 24.49 dB, which is better than what the best model in mini-project 1 achieved. This is because of a smaller network that learns quicker, but will also stagnate at a lower PSNR-score given enough training time.

E. Improvements

To further improve the model, the first priority would be to create a deeper network of convolutions and increase the time restriction. In addition, the hyperparameters can be tuned more finley and the data can be augmented by e.g. cropping and rotations.

VIII. Summary

The model managed to remove noise from 32×32 images using the Noise2Noise-method without the use of PyTorch modules. The network, as shown in figure 2, was very simple, but still manages to achieve a PSNR of 24.49. This was achieved with the hyperparameters batch size and learning rate set to 8 and 0.8 respectivelly.

References

[1] Jaakko Lehtinen et al. *Noise2Noise: Learning Image Restoration without Clean Data*. 2018. DOI: 10.48550/ARXIV.1803.04189. URL: https://arxiv.org/abs/1803.04189.

[2] Kevin Shen. *Effect of batch size on training dynamics*. 2018. URL: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e.