EE-559: Deep Learning mini-project 2

Chen JIAWEI
Chrysanthou MELINA
Su JINGRAN
May 27, 2022

***Abstract*—The implementation in a variety of architectures of methods such as Noise2Noise has been an important tool in the fast-paced rate of improvement in the area of signal reconstruction by machine learning. The aim of this report is to implement our framework to denoise images from scratch in order to better understand the functioning of neural networks.**

I. INTRODUCTION

The objective of this project is to design a " denoising framework" using only Pytorch's tensor operations and the standard libraries. In other words the autograd or neural-network modules shouldn't be used. We will create our own blocks from scratch in order to follow the proposed architecture.

II. METHODOLOGY AND MODELS

Python 3.7.7 was used as the programming environment. The project was implemented with out autograd or torch.nn modules. The files *train_data.pkl* and *val_data.pkl* provided in the course were used as data sets for training and testing. The only packages used were the ones defined in the description of the project. To be more precise:

- Torch: empty
- Torch.nn.functional: fold, unfold
- Standard library: random, math

A. Data preprocessing

Our training data set corresponds to 50000 noisy pairs of images. This images have 3 channels corresponding to red, green and blue. Each channel was normalised in order to have values between [0,1]. This normalisation was done by a simple division by 255 (highest value of each channel).

B. Implemented methods

In this section we will explain the layers implemented from scratch, necessary for the image denoising.

1) Convolution: Makes use of filters or kernels to detect features in the images. It has learnable parameters, which are: input size (i), kernel size (k), padding (p), stride (s). In order to implement the convolutional layer, one of the most important for our framework, we treated it as a linear layer. Initially we computed the convolution with multiple for loops. Then, to decrease the complexity, we used the unfold operation.

2) Nearest Up-sampling: In image denoising we need to have an upsampling layer as these layers are used to upsample the input feature map to a desired output feature map using some learnable parameters. As the implementation of such function is much complicated, we used an alternative solution. The implementation of the Upsampling layer in our model is composed by Nearest Up-sampling followed by a Convolution layer. This combination works as the Nearest Up-sampling layer, upsamples the input and then the convolution layer learns how to fill in details during the model training process.

C. Loss functions and optimisation

As the main purpose of this project is to tackle a denoising problem, the following functions were used to train the model.

1) Mean Square Error loss: Measures how accurately our model is able to predict the expected outcome. To calculate the MSE, we take the difference between our model's predictions and the targets, square it, and average it out across the whole data-set. The MSE is great for ensuring that the trained model has no outlier predictions with huge errors, due to the squaring part of the function.

2) Stochastic Gradient Descent: A simple yet effective iterative approach for maximizing an objective function with appropriate smoothness criteria. Because it replaces the actual gradient with an estimate, it can be considered a stochastic approximation of gradient descent optimization. This minimizes the high computational cost, especially in high-dimensional optimization problems, allowing for faster iterations in exchange for a reduced convergence rate. A further improved algorithm is SGD with momentum. SGD with momentum is a method which helps accelerate gradients vectors in the right directions, thus leading to faster converging. To make the SGD with momentum more robust we can also add dampening. The dampening is used to weaken the effect of the momentum.

The SGD has been chosen as the optimizer due to its efficiency and ease of implementation. SGD with momentum and dampening has also been implemented.

D. Activation functions

1) Rectified Linear Unit Activation Function: The ReLU is the most used activation function, which has no shortcomings with gradient vanishing. The forward and backward implementation is as shown below:

- *Forward Propagation:*

$$Y_{ij} = \mathrm{ReLu}\left(X_{ij}\right) = \begin{cases} X_{ij}, & X_{ij} > 0 \\ 0, & \text{Otherwise} \end{cases} \tag{1}$$

- *Backward Propagation:* There is no parameters in ReLU layer, so chain rule is not needed. We can calculated it as following:

$$\left(\frac{\partial Y}{\partial X}\right)_{ij} = \left(\frac{\partial \,\mathrm{ReLu}(X)}{\partial X}\right)_{ij} = \begin{cases} 1, & X_{ij} > 0 \\ 0, & \text{Others} \end{cases} \tag{2}$$

2) Sigmoid: A bounded, differentiable function that is defined for all real input values and has a non-negative derivative at each point. The forward and backward implementation is as shown below:

- *Forward Propagation:*

$$Y_{ij} = \mathrm{Sigmoid}\left(X_{ij}\right) = \frac{1}{1+e^{-X_{ij}}} \tag{3}$$

- *Backward Propagation:*

$$\left(\frac{\partial Y}{\partial X}\right)_{ij} = \left(\frac{\partial \,\mathrm{Sigmoid}(X)}{\partial X}\right)_{ij} = \frac{e^{-X_{ij}}}{\left(1+e^{-X_{ij}}\right)^2} \tag{4}$$

E. Network structure

Our network follows the structure proposed by the description of the project and receives as input a 3 x 32 x 32 tensor corresponding to pairs of 32 x 32 RGB images. The details of our structure are as seen below.

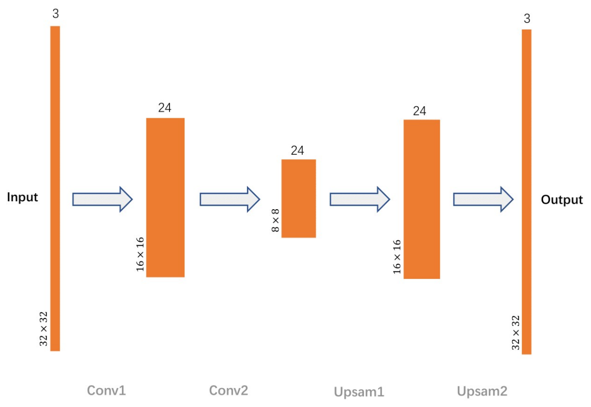

Fig. 1. Architecture of the model

The first layer is composed of a convolutional layer of 3 channels to 24, a kernel of size 3, stride of 2 and padding of 1 and then a ReLU non linearity. Subsequently we have a second convolutional layer of 24 to 24 channels with a kernel of size, stride of 2 and padding of 1, followed a ReLU non linearity. Then we have the upsampling layer. As mentioned before the upsampling layer is composed by a Nearest neighbour upsampling followed by a convolution. Our first upsampling layer has a Nearest neighbour upsampling with a scaling factor of 2 and is then followed by a convolutional layer of 24 to 24 channels with a kernel of size 3 and stride and padding of 1. Then we have a ReLu activation function followed by the second upsampling layer. This layer has a Nearest neighbour upsampling with a scaling factor of 2 and is then followed by a convolutional layer of 24 to 3 channels with a kernel of size 3 and stride and padding of 1. And finally we have a sigmoid activation.

F. Other techniques

In our effort to optimise the output of our network we have implemented different strategies.

1) Weight and bias initialisation: In the first versions of our model, we initialised weights and biases using the Xavier distribution. We realised that the results obtained were not satisfactory as the output wasn't stable and the MSE loss decreased at a very small rate. We then tried with the Uniform distribution and the results were satisfactory as you will see in the next section.

2) Stochastic Gradient Descent with momentum and dampening: Before realising that we could optimise the weight and bias initialisation, we tried to adjust the SGD optimizer. We therefore implemented SGD with momentum and dampening. However the results weren't significantly better. Therefore for our results we used a value of zero for momentum and dampening.

III. Results

A. Tuning parameters

After determining the network structure, we tuned the hyper-parameters of learning rate and batch size. Both of this parameters are essential for the optimisation of the training process. This two parameters are directly linked, we therefore had to train the model with different parameters simultaneously. The values used are as seen below:

- *Number of epochs*: 5
- *Learning rate*: [0.001, 0.01, 0.1, 0.5, 1, 1.5, 2]
- *Batch size*: [4, 12, 24, 48]

Below we can see the obtained results.

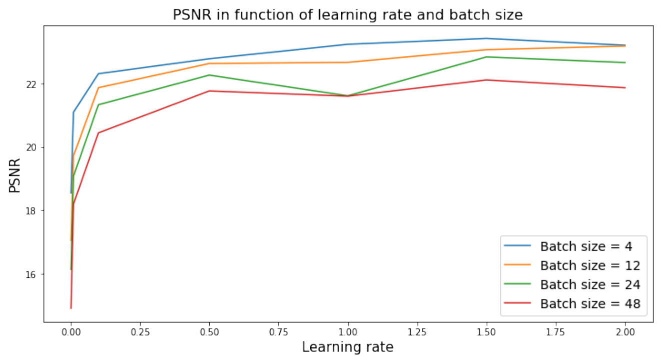

Fig. 2. PSNR in function of learning rate for different batch sizes

Here, we use grid search on learning rate and batch size, the best outcome was given by lr=1.5, and batch size=4. It is clear that a larger learning rate with a smaller batch size gives better results.

B. Training loss

We tested 20 epochs on the *val_data.pkl* dataset, and obtained the following test loss and PSNR evolution:

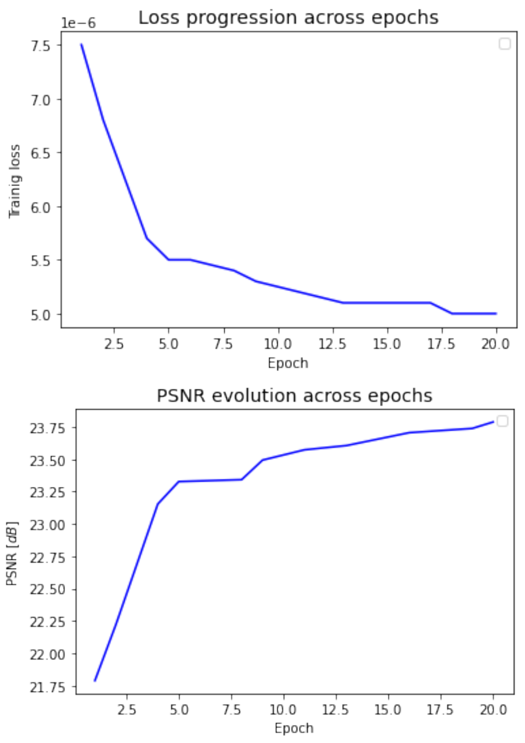

Fig. 3. Test loss and test PSNR

C. Performance of the model

By using the best model and testing our outcome in the *val_data.pkl*, we got the average PSNR of 23.53 dB. Here we compare it with the input image with an average PSNR of 20.72 dB and the ground truth.

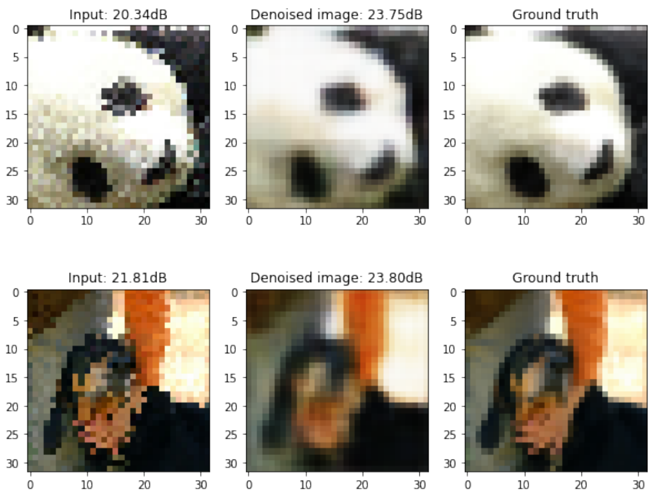

Fig. 4. Example of final results

IV. Summary and discussion

In this project, we implemented a network from scratch, and denoised images only by using pairs of noisy data as input. The trained model can be used as a denoiser and returns clean images with average 23.2dB PSNR. Given the computational complexity of the layers, the time limit, did not allow us to get better results. As further improvement we could consider optimizing the current algorithm to run each epoch in a smaller time. We could also considering a deeper network, adding more channels or getting more training data but this is also computationally costly.