EE-559 Deep Learning
Report on mini-project 2

Spring 2022

Yixuan Xu
Master student in computer science
EPFL
Lausanne, Switzerland
yixuan.xu@epfl.ch

Yifei Song
Master student in computer science
EPFL
Lausanne, Switzerland
yifei.song@epfl.ch

Kanlai Peng
Master student in applied physics
EPFL
Lausanne, Switzerland
kanlai.peng@epfl.ch

***Abstract*—In this report, we briefly explain our implementation of different blocks of the network and make some analysis on the results we get from the experiments.**

I. Introduction

In this mini-project, our goal is still to implement a Noise2Noise model with a simplified network architecture comparing to U-net. However, unlike what we have done in the previous mini-project using the PyTorch framework, this time we have to implement all the blocks we have used before by ourselves and only use the Python Standard Library. The main task of this mini-project is to check that all the blocks implemented by ourselves work in the same way as their PyTorch counterparts.

II. Implementation of different blocks

A. Convolution layer

The basic idea of convolution layer is to repeat a dot product between the weight of a filter called "kernel" and part of the input tensor with the same size. We swipe across the entire input signal with the kernel and in case of multiple channels signal, the kernel is not swiped across channels. Instead, we simply perform D such convolutions where D corresponds to the number of filters in the layer. To implement the convolution layer, we choose to use $fold$ and $unfold$ functions provided by the $torch.nn.functional$ library.

1) Forward Pass: Concerning the forward pass, the idea is that these dot products between the kernel (size: $D\times C\times h\times w$) and parts of input tensor (size: $B \times C \times H \times W$) can be seen as matrix multiplications after some reshaping. We start with initializing the kernel and then unfold the input tensor according to the kernel size. Next, we perform a matrix multiplication and add bias to the results if needed (Equation 1). Finally, we reshape the output tensor to a specific size determined by parameters of the layer such as: padding, dilation and stride. The general operation can be summarized using the following equation:

$$s_i^{(l)} = \sum_j w_{i,j}^{(l)} \circledast x_j^{(l-1)} + b_i^{(l)} \tag{1}$$

2) Backward Pass: During the backward pass, we first use operations like fold, unfold, reshape and transpose to transform the convolution to a linear function. Then, same as in the linear layer, the gradients of loss with respect to weight and bias can be computed using the following equations, which are all based on the chain rule of derivation:

$$\frac{\partial l}{\partial \tilde{w_{i,j}}^{(l)}} = \frac{\partial l}{\partial \tilde{s_i}^{(l)}} \frac{\partial s_i^{(l)}}{\partial \tilde{w_{i,j}}^{(l)}} = \frac{\partial l}{\partial \tilde{s_i}^{(l)}} \tilde{x_j}^{(l-1)} \tag{2}$$

$$\frac{\partial l}{\partial \tilde{b_i}^{(l)}} = \frac{\partial l}{\partial \tilde{s_i}^{(l)}} \tag{3}$$

where the wave lines denote that all the variables in the above equations have already undergone certain conversions. Also, for the gradient of the loss with respect to the module's input, we have:

$$\frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial l}{\partial y} \cdot w \tag{4}$$

So we can see that backward pass is essentially applying the chain rule again and again to calculate and accumulate the gradients. In brief, the main trick to implement convolution layer is to treat it as a linear layer by using linear transformations.

B. Transposed convolution layer

Contrary to what the convolution layer does, the transposed convolution layer provides a way of increasing the size of the signal. In this case we have:

$$\left[\frac{\partial l}{\partial x}\right]_u = \sum_i \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial x_u} = \sum_i \frac{\partial l}{\partial y_i} k_{u-i+1} \tag{5}$$

where k denotes the kernel coefficients. From equation 5 we can see that the derivative $\frac{\partial l}{\partial y}$ and the filter k are visited in opposite ways, as indicated by the index i. So, this operation corresponds to transposing the weight matrix of the equivalent fully-connected layer and thus the transposed convolution can be seen as a weighted sum of translated kernels. We simply compute a linear combination of kernels at every location of the input signal.

1) Forward Pass: The general steps remain the same as in the forward pass of convolution layer. We initialize the kernel and make a matrix product between the flattened input tensor and weight. The main difference is that here we determine the height and width of our output images just as what we do in the backward pass of convolution layer, i.e. the forward pass of transposed convolution layer corresponds to a convolution layer's backward pass. Besides, unlike the alternative nearest neighbour up-sampling with convolution layer, where the convolution layer takes care of any given image height and width, transposed convolution requires additional output padding to guarantee the output shape when stride is not 1.

2) Backward Pass: The idea of backward pass of transposed convolution layer is still using equation 3 to calculate and accumulated the gradient of loss with respect to the model's parameters. And use equation 4 to compute the gradient of loss with respect to the model's input.

C. Upsampling layer

The alternative way to approach the result is to replace the transposed convolution layer with upsampling layer, which is a combination of nearest neighbor upsampling and convolution. Instead of calculating an average value by some weighting criteria, nearest neighbor upsampling simply determines the "nearest" neighbouring pixel, and assumes the intensity value of it. This may be a possible alternative of transposed convolution layer and we will test it's performance in the experiments section.

D. ReLU & Sigmoid

In this mini-project, we use ReLU and Sigmoid as the activation functions, which are respectively defined as:

$$f(x) = max(0, x) \tag{6}$$

$$S(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

with the derivatives given by:

$$f'(x) = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x > 0 \end{cases} \tag{8}$$

$$S'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = S(x)(1 - S(x)) \tag{9}$$

E. Mean Squared Error

The L_2 loss, which is also called the mean squared error loss, is defined as:

$$mse = \frac{1}{n}\sum_{i=1}^{n}(y_{i,pred} - y_{i,target})^2 \tag{10}$$

and accordingly, the backward pass of MSE is given by:

$$\frac{\partial mse}{\partial y_{i,pred}} = \frac{2}{n}\sum_{i=1}^{n}(y_{i,pred} - y_{i,target}) \tag{11}$$

Here we need to pay attention that, in our case, n is the product of all dimensions of our 4d input tensor.

F. SGD Optimizer

In our work, we use the mini-batch stochastic gradient descent as the optimizer to update the parameters of our model during training. The mini-batch SGD consists of updating the parameters w_t after each batch:

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla l_{n(t,b)}(w_t) \tag{12}$$

in our case, the order $n(t, b)$ to visit the samples is sequential and without replacement. To be similar to the PyTorch counterpart, we have also implemented the `zero_grad()` function, which zeros the gradient accumulated.

III. Architecture of the Network

As mentioned previously, the transposed convolution layer can be replaced by upsampling layer, which is a combination of nearest neighbor upsampling and convolution. So, there are two possible architectures of the network, as shown in table I.

Architecture 1	Architecture 2
Conv	Conv
ReLU	ReLU
Conv	Conv
ReLU	ReLU
TransposedConv	UpsamplingConv
ReLU	ReLU
TransposedConv	UpsamplingConv
Sigmoid	Sigmoid

TABLE I
2 POSSIBLE ARCHITECTURES OF THE NETWORK

In the following experiments, we will test the performance of these two architectures respectively and choose the one which performs better on the validation set as the final architecture.

IV. Experiments & Analysis

A. Performance of the model with different choices of architectures

1) Architecture 1: To test which architecture of the network works better on the denoising task, first we train architecture 1 for 15 epochs (not too many to avoid over-fitting) and plot the evolution of the training loss and validation loss during the training process, as well as the evolution of the PSNR value. The results we get are shown in Fig.1.

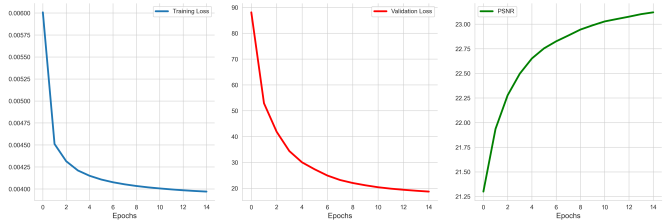

Fig. 1. Evolution of training loss (left), validation loss (middle) and PSNR value (right) of architecture 1

From Fig.1 we can see that both training loss and validation loss drops quickly at the beginning of training and reach a certain minimum at the end with a lower drop rate. The absolute value of the validation loss is much larger than the absolute value of the training loss, which means our model doesn't generalize well on other dataset.

Looking at the evolution of PSNR value, we observe that the PSNR value produced by the model has an obvious increase during training and finally exceeds 23dB. Notice that the original PSNR value of the validation data is 19.62 dB, we can conclude that architecture 1 truly works for denoising tasks.

2) Architecture 2: Having tested architecture 1, we now move on to architecture 2 and to see whether it achieves a good result. One thing we need to pay attention is the learning rate, for architecture 2, we need to have a larger learning rate to avoid falling into a local minimum. We have tried to train it for 5 epochs, however after the first epoch, the PSNR value we get is only around 19dB, whereas the original PSNR value of the validation data is 19.62 dB. Even after 5 epochs, the PSNR value produced is still only around 20dB. To further test whether this architecture works or not, we have printed some example images but the imaging results are not ideal and there might be some problems with the functionality of architecture 2. Therefore, we no longer discuss this architecture in the rest of our report.

B. Qualitative results

To intuitively understand how our model (architecture 1) works for denoising task, we have printed some image examples, as show in Fig.2 and Fig.3. The left one is the clean target of the validation set which doesn't have noise. The image in the middle is the corrupted image which suffers from noise. Lastly, the right one is the output of our model.

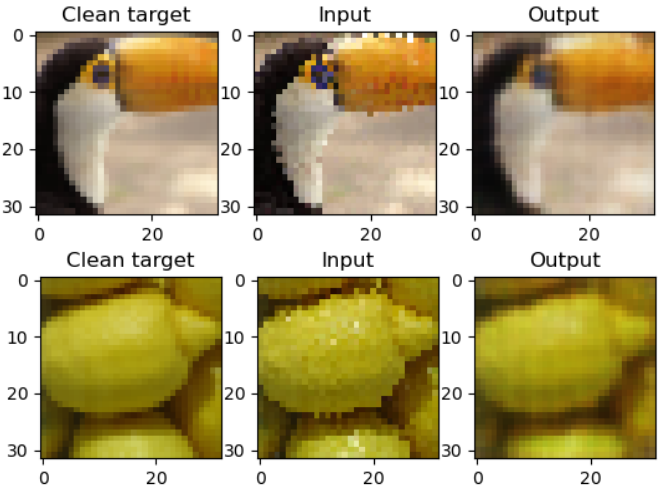

Fig. 2. Examples of output images with simple patterns

From Fig.2 we can see that when the target images have simple patterns and large blocks of the same color, and when the noise doesn't severely influence the input images, the output images of our model have quite good quality, except for a bit lower resolution.

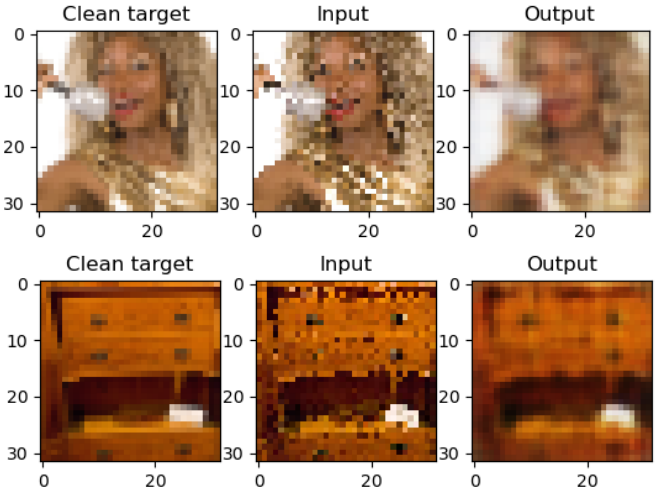

Fig. 3. Examples of losing details

However, as shown in Fig.3, when the input images have some import details which are affected by the noise, our model is not able to correctly recover these details and this leads to an obvious difference between the output images and the target ones. For example, if we look at the upper image in Fig.3, it is clear that the singer is opening her mouth because we can see some white pixels which corresponds to her teeth. However, these white pixels are severely affected by the noise and in the output images they just disappear, and we aren't sure about whether she is opening her mouth from the output image. The same thing happens to the lower image of Fig.3. In the output image we can only see five drawer handles while the target image indicates that there should be six.

V. Problem Encountered

During the training process, the most challenging problem we encountered is the vanishing gradient, which is a common issue for a gradient-based learning method, especially when the output activation function of the architecture is sigmoid. To mitigate the impact of vanishing gradient to our model, we take two approaches: adjust learning rate and standardise data before feeding into model. The resulting performance of the model indicates that standardisation is more essential to our model.

VI. Conclusion

In this mini-project, we have successfully implemented all the related blocks needed to construct a simple denoising autoencoder. Our final model works for denoising tasks, but because its structure is too simple, the final performance is not as good as that of deep autoencoders. The model is able to reduce the impact of noise, but it cannot fully recover all the details of the input images, which occasionally leads to quite blurry and low resolution output images. Overall, the

performance of the model is acceptable and in compliance with its architecture.