Project 2 - Noise2Noise model from Scratch

EE-559 Deep Learning Course

Naravich Chutisilp (341752), Veniamin Veselovsky (337188), Yasmin El Euch (250222)

May 2022

1 Introduction

In this project, the goal is to create a framework from the ground up that can be used to apply the Noise2Noise model to denoise a picture that lacks clean data [1]. To do this, we will construct several modules such as the convolution module, activation functions, loss function, and optimizers while only using the main Python libraries, as well as the Pytorch's tensor, unfold, and fold.

2 Theory and Network Design

The Module class is the basic foundation class around which the system revolves. A neural network as a whole may be thought of as an ordered collection of these modules. In this section we will briefly explain how we implemented each of the modules.

2.1 Conv2d

Conv2d is an "invariance in translation" operation that preserves the dimension of the input signal, i.e. an image will remain an image. By passing a kernel over the input space we receive an output that is the kernel multiplied by the input. In our case, the goal is to learn the weights of the kernel such that the forward propogation will produce a denoised image. Given an input of size (B, in channel, height (S_1), width (S_2)) the output will be

$$H_i = \left\lfloor \frac{S_i + 2 \times \text{ pad}[i] - \text{ dil}[i] \times ((\text{ks}[i] - 1))}{\text{stride}[i]} + 1 \right\rfloor \quad (1)$$

Where padding (pad[i]), dilation (dil[i]), kernel size (ks[i]), and stride[i] are the values for the corresponding parameters of the ith dimension, where $i \in \{1, 2\}$.

Forward. The forward pass of the convolution can be done through the "unfold" operation, which extracts local blocks from a batched input signal. Specifically, unfold takes an input of size (B, C_I, S_1, S_2) and a weight of size (C_O, C_I, ks_1, ks_2) and produces a matrix of size $(B, C_O \cdot ks_1 \cdot ks_2, L)$ where $L = H_1 \cdot H_2$. This matrix can then be multiplied by a transformed weight matrix $(C_O, C_I \cdot ks_1 \cdot ks_2)$ to get the final output matrix (B, C_O, H_1, H_2).

Backward. To run the backward pass of the convolution we take in an output gradient from the following layer and get (1) a weight gradient for weight updates, (2) an input gradient, (3) a bias gradient to update the biases. The weight gradients are simply calculated by multiplying the unfolded matrix from the forward pass with the output gradient. However, to get the input gradients we are required to reverse the steps of the forward pass first by reshaping the matrix, then multiplying with the weights, and finally folding the matrix. The bias gradient is simply the output gradient summed along the batches and output size.

Initialization. We initialize all the weight gradients to $W \sim \mathcal{U}(-1/\sqrt{N}, 1/\sqrt{N})$ where N is the batch size of the input, and the bias gradients to $B \sim \mathcal{N}(0, 1)$. After, we perform each update using the weight and bias gradients we calculated in the backward step, alongside the learning rate, which will be discussed in the stochastic gradient descent section.

2.2 Upsampling

Since in noise2noise we need recover the original input dimensions, we implement an upsampling module to increase the dimension of the output space. Upsampling gives a result that is a close approximation of the sequence that would have been produced if the signal had been sampled at a greater rate or density. The output is a repeated tensor which has the same shape as input but has a different size. Specifically, upsamping consists of first conducting a nearest neighbour upsampling and then running the convolution on the output with padding equal to kernel size - 1. The forward step of the nearest neighours was done using the weave_interleave tensor function and the backward step summed over the neighbouring pixels.

After running the convolution with the padding on the upsampled image we get two times the input size. So, if we input an image of size (B, C_I, H, W) we'll get $(B, C_O, 2H, 2W)$.

2.3 Activations and Loss

Sigmoid The sigmoid is used as our final activation layer and returns a value $x \in (0, 1)$. For images, this can be interpreted as the strength of the resepctive pixel. Mathematically, it will be

$$y = \sigma(x) = \frac{1}{1 + e^{-x}} \quad (2)$$

Which will give gradient:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \sigma(x)(1 - \sigma(x)) \quad (3)$$

Where x is our input into the forward pass and $\frac{\partial \mathcal{L}}{\partial y}$ is the output gradient passed in.

ReLU Given a loss function $\mathcal{L}$, input x, output y and activation function f, we are interested in $\frac{\partial \mathcal{L}}{\partial x}$ given that $y = f(x)$:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x}$$

For ReLU, we calculate the output y and input gradient using the following formula.

$$\text{y} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \quad \frac{\partial \mathcal{L}}{\partial x} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Mean Square Error Mean Square Error is used to model how far off the estimation is. MSE is used as Criterion in the forward and backward pass as described in the table below.

Loss	Forward pass	Backward pass
MSE	$\frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y_i})^2$	$2(Y - \hat{Y})$

2.4 Sequential

This module, is the container block for all neural networks of our framework.

- **Instantiation Parameters:** *modules*
  The Sequential organizes the modules - the l (layers) and activation functions - passed as arguments in a sequential (ordered) manner.
- **Forward**($X \in R^{dim_{in}(1)}$): $\in R^{dim_{out}(l)}$
  Iterates over all of the layers in ascending order, applying the current module's forward function on the preceding module's output. (1 to l)
- **Backward**($X \in R^{dim_{out}(l)}$): $\in R^{dim_{in}(1)}$
  Iterates over all of the layers in reverse order, applying the current module's backward function on the preceding module's output. It gives the loss gradient with respect to the parameters.

2.5 Stochastic Gradient Descent Optimizer

By updating the model's parameters in the opposite direction of the objective function's gradient, gradient descent is used to minimize the loss function L() with respect to the model's parameters. On top of that, there are various algorithms to optimize this approach in order to obtain fast convergence while avoiding getting locked in a local minimum. The stochastic gradient descent, which we implement, offers a good compromise between accuracy and computation time.

- **SGD:**

$$\theta^t + 1 = \theta^t - \mu\nabla_\theta \mathcal{L}(\theta; x_j, y_j)$$

given μ the learning rate.

- **SGD mini batch version:**

$$\theta^t + 1 = \theta^t - \mu\nabla_\theta \mathcal{L}(\theta; x_{j,j+n}, y_{j,j+n})$$

Sequential	**Nout**
conv1 (2x2 kernel, stride 2)	48
relu1	48
conv2 (2x2 kernel, stride 2)	96
relu2	96
upsampling3	48
relu3	48
upsampling4	3
activation	3

Table 1: The summary of the sequential model.

Sequential	**Nout**
conv1 (2x2 kernel, stride 2)	48
relu1	48
conv2 (2x2 kernel, stride 2)	48
relu2	48
conv3 (2x2 kernel, stride 2)	48
relu3	48
conv4 (2x2 kernel, stride 2)	96
relu4	96
upsampling5	96
relu5	96
upsampling6	96
relu6	96
upsampling7	48
relu7	48
upsampling8	3
activation	3

Table 2: The summary of the deep sequential model.

3 Training

With the custom-made Pytorch implementation in-hand, we turned to empirically testing the model using a dataset consisting of two sets of the same images with normal noise added to them. The first step was to determine the optimal set of parameters for the model.

3.1 Hyperparameter Tuning

We test a few different channels and learning rates, iterating through the recommended channels ($(3 \to 48 \to 96 \to 48 \to 3)$, as well as $(3 \to 60 \to 120 \to 60 \to 3)$, $(3 \to 24 \to 48 \to 24 \to 3)$, and $(3 \to 10 \to 20 \to 10 \to 3)$. Since tuning is computationally intensive, we restricted ourselves to 3,000 of the inputs for training and a batch size of 100. All results provided surprisingly similar on the channel comparison results resulting in a MSE error between 0.730 and 0.744. We ran through several learning rates between [0.001,0.8] and found that they all performed comparably, saturating quite rapidly.

We classify our model as sequential (see Table 1) and deep sequential (see Table 2). We call each model based on its last activation layer. For example, deep sequential

model with Sigmoid as its last activation layer is called Deep NN + Sigmoid.

4 Result

We trained our model on a Google Colab GPU on the full training set and validated using the validate set. To begin, we find that the model performs better with more training, which is a relief given initial technical difficulties. But after several epochs, the model saturates very quickly (see Figure 1) at a MSE of 0.073 and a PSNR of 12.68 db. We include an image of the performance in Figure 3 where as we can see, the model predicts a uniform colour across the samples. To understand this disappointing result we tested a few different architectures to see if they could offer superior results. All the models with ReLU as their last activation layer do not converge (see Figure 2). The best model is NN + Sigmoid with the PSNR of 12.68 db. We present a summary of the results in Table 3.

We also verified that the modules themselves were implemented the same as Pytorch and found that each of them gave the same results. Thus, we turned to brainstorming other issues that may have arose.

5 Limitations and Discussion

The source of the issue may be in the training process. In particular, we suspect that the SGD weight update might be the problem. The first possibility, is that the SGD model may not be able to arrive at the global minimum effectively given our data, and more recent developments like Adam may have performed better. The second possibility, is that the weight updates are not being properly done, but tests reveal that this is unlikely to be the case.

6 Conclusion

In this assignment we implemented a Pytorch-esque framework of convolution, upsampling, and various activation and loss functions. We use these custom-made modules to denoise images using the noise2noise framework outlined above. Overall, while the individual modules are correct, our model is able to learn a version of the de-noised image that is quite a poor representation of the clean signal.

	Best PSNR (db)
NN + Sigmoid	**12.68**
Deep NN + Sigmoid	12.67
NN + ReLU (lr=0.001)	6.73
NN + ReLU (lr=0.0001)	6.99
Deep NN + ReLU	6.52

Table 3: The summary of PSNR results of all the model variations

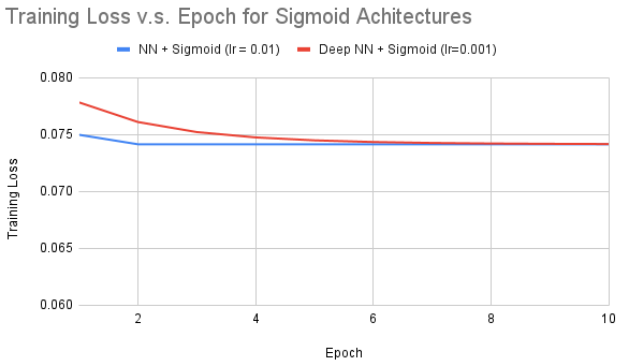

Figure 1: Training Loss v.s. Epoch for Different Model Architecture with Sigmoid as the last activation layer

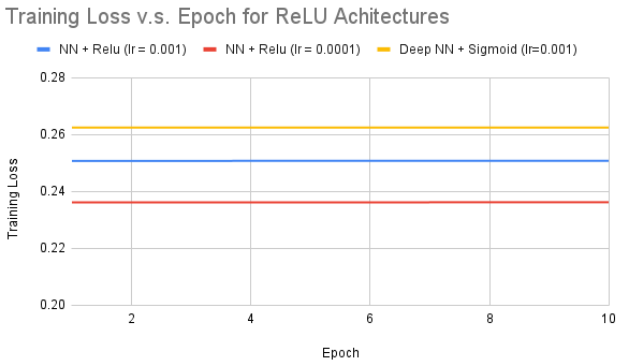

Figure 2: Loss v.s. Epoch for Different Model Architecture with ReLU as the last activation layer

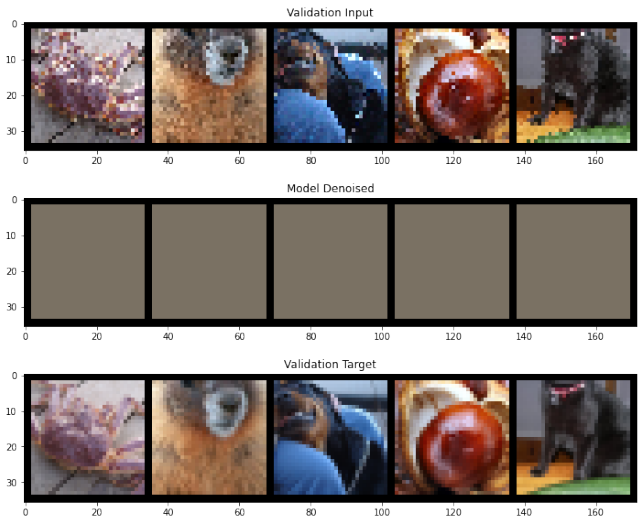

Figure 3: Qualitative Result of NN + Sigmoid

References

[1] J. Lehtinen, J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, and T. Aila, "Noise2noise: Learning image restoration without clean data," *CoRR*, vol. abs/1803.04189, 2018. [Online]. Available: http://arxiv.org/abs/1803.04189