Building a CNN from Scratch

Mihai David, Ioan-Florin-Cătălin Nițu, Ana-Arina Răileanu
{mihai.david, ioan.nitu, ana-arina.raileanu}@epfl.ch
School of Computer and Communication Sciences, EPFL, Switzerland

***Abstract*—We used the Python Standard library to build a Noise2Noise model from scratch. We obtained comparable results and running time to our previous PyTorch implementation. Our model achieves a Peak Signal- to-Noise Ratio of 24.46 in 50 epochs.**

I. Introduction

In this report, we summarize our approach at building a network, using only the Python Standard library[1] and five of the functions of the PyTorch library [1]: `empty`, `cat`, `arange`, `fold`, and `unfold`. We assess the utility of the model on an image denoising task.

II. General Implementation Details

We have two different main classes: the `Model` class and the `Module` class. The model represents the architecture that we use for training and generating predictions, whereas the modules are the the blocks that make up the model.
When developing each module, we aim for a structure that is similar to that of the Pytorch modules [1]. We use function names and parameter lists that mimic the package's standards for ease of use. While most of the modules we implemented have intuitive and well-known formulas and derivations, developing the convolutional layer and the transposed convolutional layer required a bit more effort and creativity.
In order to assess our network, we use the provided dataset containing 50.000 image pairs for training, and 1000 image pairs for testing. The images are RGB images, of size 32×32, with values between 0 and 255. For a faster convergence[2], we normalize the images to change their scale to $[0, 1]$.
While developing the layers, we have used unit-tests to compare their results with those of the Pytorch implementation.

III. Model & Modules in Depth

In this section, we will discuss the main features of each of the classes we developed.

A. Model

The `Model` class contains three main variables:

- `model` - A pre-defined network, whose layers are instantiated when a new `Model` instance is created.
- `optimizer` - An optimizer, which is in charge of performing gradient descent steps.
- `loss_function` - A cost function used to determine the direction and size of the gradient descent steps.

This class provides functionalities to store and load a model, train it, and generate predictions for a batch of images.

B. Module

Each of the blocks building the model are inherit from the `Module` class. Therefore, they all implement the following four functions:

- `forward` - Performs a single step in the forward direction of the network, receiving as input the output of the previous layer and applying a transformation on it.
- `backward` - Performs a single step in the backward direction of the network, receiving as input the gradients of the next layer. In the backward phase, each module computes the gradient needed by the previous layer, and the gradients of its parameters, if any.
- `param` - Returns a list of the parameters of the model and their gradients in tensor form.
- `zero_grad` - Resets the gradients to 0.

In the next few subsections, we will discuss the modules.

C. Sequential

On a high-level view, the `Sequential` module represents the network. Much like PyTorch's Sequential[3], its role is to group the other modules together. It stores a dictionary that maps the name of the module to its instance. In the forward pass, it passes the input from one module to the next, starting from the first module. In the backward pass, it propagates the gradient updates from the last layer to the first.

D. Convolution

When implementing the convolutional layer, we made use of the formulas presented in class for the backpropagation

[1] https://docs.python.org/3/library/

[2] https://becominghuman.ai/image-data-pre-processing-for-neural-networks-498289068258

[3] https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html

of the multi-layer perceptron[4] to understand how they would apply to a patch of the image. By unfolding a patch, we can see the result as an input to a linear layer, which was also stated in the project description. In the backward pass, we compute the gradients with respect to the bias and the weight, and we store them for the optimizer's update. Also, we compute the gradient of the cost function with relation to the layer's input, which will later be fed into the module preceding it.

E. Transposed Convolution

For the upsampling operation, we decided to use the transposed convolution, which performs the operations of the convolution in reverse order (the forward pass of transposed convolution is the backward pass of convolution, respectively the same for relation for the backward pass). We started from the formulas given in an article [5]. The reverse symmetry with the convolution helped us rethink the logic behind the mathematical formulas when coding the forward and the backward passes.

F. ReLU

The forward layer applies the function given in Equation 1. This means that entries above 0 will be stored, while the rest will become 0. The backward pass computes the gradient of the loss with respect to the input it received in the forward pass, by multiplying the gradient from the next-in-line layer with the gradient of the function it applies with respect to its input. The multiplication between the two is explained by the chain rule. The formula of its derivative is given in Equation 2.

$$R(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{1}$$

$$R'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2}$$

G. Sigmoid

In a similar manner to the ReLU function, implementing the sigmoid function was straightforward. The formula applied in the forward pass is given in Equation 3, while the formula for the backward pass is given in Equation 4.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{4}$$

[4]https://fleuret.org/dlc/materials/dlc-handout-3-6-backprop.pdf

[5]https://towardsdatascience.com/only-numpy-understanding-back-propagation-for-transpose-convolution-in-multi-layer-cnn-with-c0a07d191981

H. Mean Squared Error

The Mean Squared Error (MSE) module will be the last module in the network. Its purpose is to compute a cost function, given the outputs of the last layer and the targets. In the forward method, it computes the loss of the model on the batch according to Equation 5, where B is the batch size. We use the the loss per batch to obtain the global loss, by summing up all of the losses. In the backward pass, it returns the gradient of the loss function, whose simplistic form is given in Equation 6. In reality, the gradient of the loss function is computed in the forward pass and stored by the object, and it is returned in the backward pass.

$$L(y\_true, y\_pred) = \frac{1}{|B|} \sum_{i=1}^{|B|} (y\_true_i - y\_pred_i)^2 \tag{5}$$

$$L'(y\_true, y\_pred) = \frac{2}{|B|} \sum_{i=1}^{|B|} (y\_true_i - y\_pred_i) \tag{6}$$

I. Stochastic Gradient Descent

This module does not have any forward or backward method. Instead, its goal is to iterate through the parameters and the gradients of the model and to update their value. Even though the module is named `SGD`, it works with batches too, not only single image inputs. The `step` function is implemented according to the pseudocode offered in the PyTorch documentation [6]. Each epoch of the training phase ends with an SGD step, where the parameters of the model are updated based on the computed gradients. Each epoch starts with setting the gradient variables to zero, such that previously computed gradients would not affect the current run.

IV. Results

After trying various combinations of parameters, we have decided to use the model described by Figure 1. The figure was generated with the PyTorchViz package [7], and the last two convolutions of the model are transposed convolutions. We trained the model with a learning rate equal to 5, even though it is not generally recommended to use a learning rate above 1[8] as it seemed to obtain the best results after relatively few epochs. We measured its performance on the train and test sets after each of the first 50 epochs and plotted the results. We can see that both the PSNR 2 and the MSE 3 error improve with time, while the gap between the curves of the train error and the test error seems to stabilize, suggesting that the model is probably not overfitting. Table I shows a comparison between an input image from the test set, a

[6]https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

[7]https://github.com/szagoruyko/pytorchviz

[8]https://ai.stackexchange.com/questions/23740/why-is-the-learning-rate-generally-beneath-1

prediction generated by the model after a number of epochs, and the target. The results are consistent with the goal of image denoising. After 50 epochs we achieve a PSNR of 24.46 on the test set.

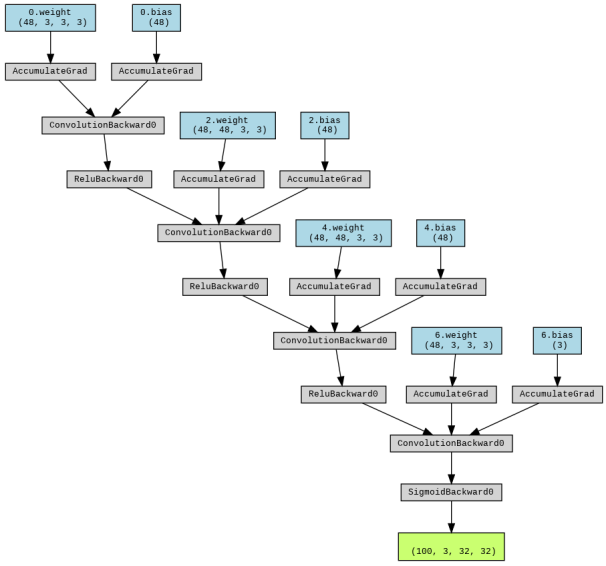

Figure 1: The structure of the final architecture used for image denoising.

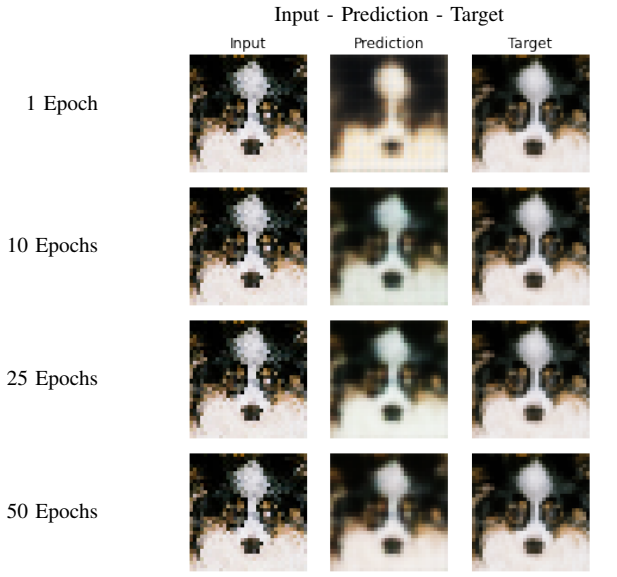

Table I: Test result comparison with a varying number of epochs.

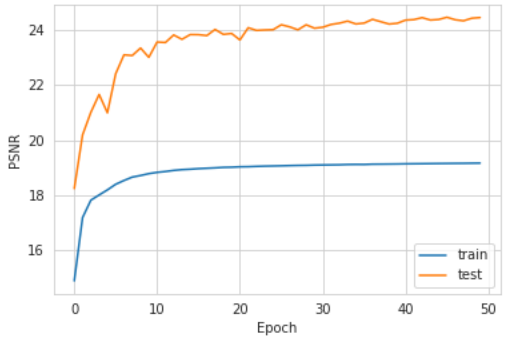

Figure 2: Peak Signal-to-Noise Ratio (PSNR) over 50 epochs.

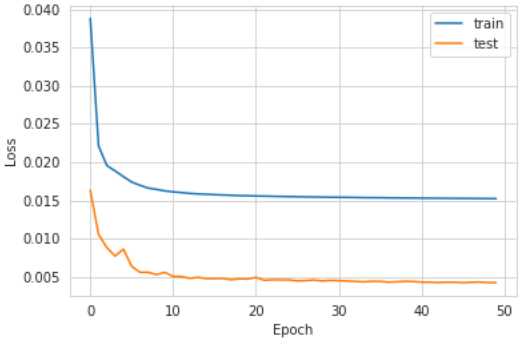

Figure 3: Mean Squared Error (MSE) over 50 epochs.

V. Conclusion

We relayed on mathematical, programming, and reverse engineering skills to implement multiple layers of the PyTorch library. We have obtained results that are consistent with the library's implementation, with a relative tolerance of 1e-03 and an absolute tolerance of 1e-06, and a running time comparable to our previous implementation of a Noise2Noise model - 115 seconds/epoch, instead of 50 seconds/epoch on a machine provided by Google Colaboratory. We have developed a model from scratch that achieves a Peak Signal-to-Noise Ratio of 24.46 in 50 epochs.

References

[1] 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*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf