Training a Noise2Noise Denoising Autoencoder from scratch

Deep Learning MiniProject 2

Francesco Salvi Lukas Van Den Heuvel Jiří Lhotka

EPFL, Lausanne, Switzerland
{cornelius.vandenheuvel,francesco.salvi,jiri.lhotka}@epfl.ch

Abstract

In this report, we implement a denoising autoencoder from scratch, using only the Python Standard Library and basic utility function from PyTorch, with *autograd* functionalities always disabled.

1 Implementation choices

In this section, we briefly discuss our implementation choices, in terms of modules, optimizers and weight initialization.

1.1 Modules

All the modules are implemented to support batched inputs, which are made of PyTorch tensors representing images. To refer to such inputs, we will make use in the following of the notation (N, C, H, W), where N represents the batch size, C the number of channels, H the height and W the width of the image.

1.1.1 Module structure

Following the recommendations in (Fleuret, 2022a), we implemented a `Module` base class, from which all our layers and losses inherit, specializing the abstract methods `forward`, `backward` and `param`. In particular, the `backward` method takes as input the gradient with respect to the output of the layer, and it uses them to update the layer's parameters and to calculate and return the gradient with respect to its input. To mimic PyTorch implementations, we also implement the `__call__` function, which enables an instance of the class to be called directly like a function (which, in our case, calls the `forward` method in the background). Finally, we implement a `zero_grad` method, which resets the gradient of all the parameters. We implement `zero_grad` here rather than in the optimizer (as PyTorch does) as it allows easier testing. Conveniently, `__call__` and `zero_grad` are can be implemented already in the `Module` base class as the implementation is the same for all of `Module`'s children.

1.1.2 Conv2d

To implement a convolutional layer, we created the class `Conv2d`, taking as input the same parameters as the analogous PyTorch module. The weights and biases of each kernel are initialized as described in §1.3, mimicking the standard PyTorch initialization.

In the forward pass, the output of the layer is computed by interpreting convolutions as matrix multiplications, which can be done by recasting each parameter to an appropriate shape. This operation is realized by `torch.unfold`, which extracts local *patches* of the input images and transforms them to column vectors, depending on the size of the chosen kernel. In addition, weights and biases are reshaped to match the unfolded input on the dimension of the contraction, and the output of the inner product is reshaped to its final dimension using the shape formulae (Torch Contributors, 2019).

Similarly, the backward pass can be implemented in the unfolded space, in the same way as it would be for a linear layer. In practice, the reshaping operation applied to the output of the forward pass is reversed for its gradient, and after carrying the necessary computations the gradient with respect to inputs is folded back to the original input dimensions, using `torch.fold`. While doing that, the gradients of the parameters are accumulated over the batch and stored using the attribute `grad` of each tensor, initially set to 0. In addition, the gradients of the biases are accumulated also over the width and the height of the output gradient.

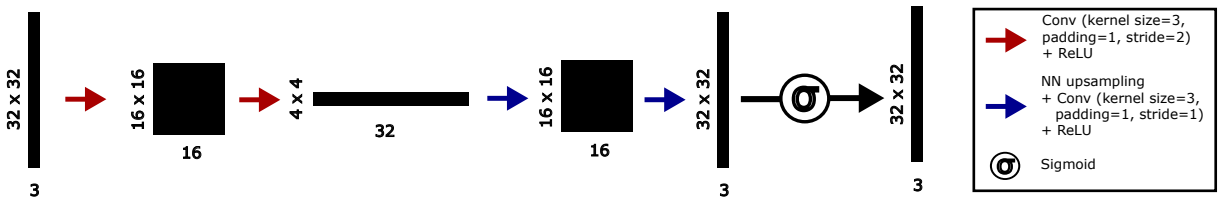

Figure 1: Model architecture

1.1.3 Upsampling

As a crucial component of the decoder, we implemented an upsampling strategy that combines an upsampling by Nearest Neighbors interpolation with a 2d convolutional layer (§1.1.2). The `NearestUpsampling`, which implements Nearest Neighbours, takes as input a `scale_factor` and, in the forward pass, construct an output by repeating the elements[1] of the input images over their height and width dimension (H, W). In the backward pass, we sum the gradients corresponding to all the upsampled repetitions of each input element by unfolding the gradients with respect to the outputs, with kernel size and stride equal to `scale_factor`. The two components are then combined in an `Upsampling` class, which inherits from `Sequential` (cf §1.1.6) and assigns as layers an instance of `NearestUpsampling` and an instance of `Conv2d`.

1.1.4 Activations

The activation functions of the neural network, `ReLU` and `Sigmoid`, are also implemented as classes that inherit from `Module`. In `Sigmoid`, all the input elements x such that $|x| > 100$ were clipped, to improve numerical stability.

1.1.5 Loss

We implemented a `MSE` class, realizing the Mean Squared Error. Being a loss, this is the only class that takes more than one input tensor, comparing predictions to respective target images. The loss is computed by averaging the squares of residuals over all dimensions of the input tensors, hence both over the batch and over C, H and W.

1.1.6 Sequential

To combine multiple modules, we implement a `Sequential` class, whose constructor takes as input a list of layers to be stacked. The class concatenates the layers' parameters and, for both the forward and the backward pass, loops through the layers to iteratively pass the results of computations through them.

[1]The repetition is performed using the tensor function `repeat_interleave`.

1.2 Optimizers

Similar to Modules (cf §1.1.1), we implemented an abstract class `Optimizer`, from which later on all the optimizers inherit. In this case, only the abstract method `step` is needed.

1.2.1 SGD

The class `SGD` implements a simple version of the Stochastic Gradient Descent algorithm, taking as input the parameters to track and the learning rate. At each step, the parameters are updated by subtracting their gradient, multiplied by the learning rate.

1.2.2 Adam

For faster and more stable convergence, we also implemented the Adam algorithm, as described in (Kingma and Ba, 2015). The number of iterations (timestep) is also recorded while calling `step`, in order to correct for biases in the estimates of first and second momentum.

1.3 Weight initialization

To initialize the parameters of the modules described in 1.1, we implemented a method performing uniform initialization and reproducing either Xavier initialization (Glorot and Bengio, 2010), He initialization (He et al., 2015) or PyTorch standard initialization (Fleuret, 2022b). For each of them, the input and output dimensions are computed by multiplying respectively the number of input and output channels by the dimensions of the kernel. Specifically, the PyTorch standard initialization, used for the final model described in §2, is reproduced by sampling from a uniform distribution $\mathcal{U}(-a, a)$, with $a = 1/\sqrt{N_{in}}$ and N_{in} the input size.

2 Model

The final architecture of the model is shown in Figure 1, following the indication in (Fleuret, 2022a), and is implemented with a `Sequential` module. As optimizer, we used Adam with a learning rate of 0.01 and all other parameters kept at their standard values ($\beta_1 = 0.9$; $\beta_2 = 0.999$; $\epsilon = 10^{-8}$). For training, we mimicked the standard PyTorch approach with a batch size of 32: for each batch, (1) the input images go through the forward pass of the network and the loss is computed, (2) gradients are set to zero, calling `zero_grad`, and then accumulated during the backward pass, and (3) a `step` of the optimizer is taken. The only difference with respect to PyTorch is that here, since the loss is technically not part of the model, `backward` is called two times: once for the MSE loss, with an input of 1, and once for the `Sequential` model, taking as input the output of backpropagation through `MSE`. During all computations, both for training and for computing predictions, the input images are scaled to the interval $[0, 1]$, dividing them by 255. For predictions, then, the output images are rescaled to the original interval $[0, 255]$ and cast to `torch.uint8`, to match the input type.

3 Experiments and results

3.1 Denoising

To verify the correctness of our implementations, we tested both the forward and the backward pass of each module against their respective PyTorch equivalent, confirming the results up to 1×10^{-8} precision. In addition, we reproduced the same model using PyTorch, with the same architecture and optimizer. The two implementations revealed to be almost perfectly equivalent, with a similar peak signal-to-noise ratio (PSNR) of $23.30\,\mathrm{dB}$ for us and $23.24\,\mathrm{dB}$ for PyTorch, after 10 epochs. The only systematic difference that we noticed is training time, which is much shorter in PyTorch, whose implementation is likely much better optimized. Some examples of denoised predictions are shown in Figure 2, with the corresponding ground truths.

3.2 SGD vs Adam

In addition, we experimented with SGD and Adam, to understand how training is impacted by the choice of the optimizer. We observe that, for 10 epochs and with a learning rate of 1, SGD achieves a lower PSNR of $22.86\,\mathrm{dB}$ with respect to Adam who reached $23.30\,\mathrm{dB}$ (cf. §3.1). In addition, convergence to an optimum is slower, as can be seen from Figure 3 comparing the learning curves of the two models. Finally, getting SGD to work required longer fine-tuning of the learning rate, from which it critically depends, while Adam worked very well with the default parameters and gave rise to little variations when changing the learning rate, which corresponds to our expectations, as with Adam the learning rate is adaptive rather than fixed like with SGD.

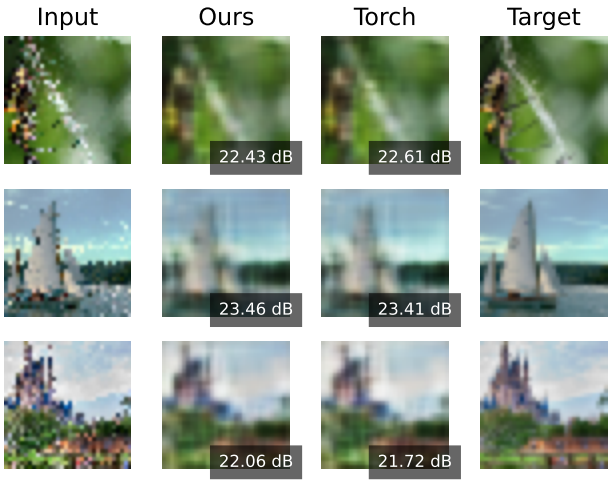

Figure 2: Examples of denoised predictions.

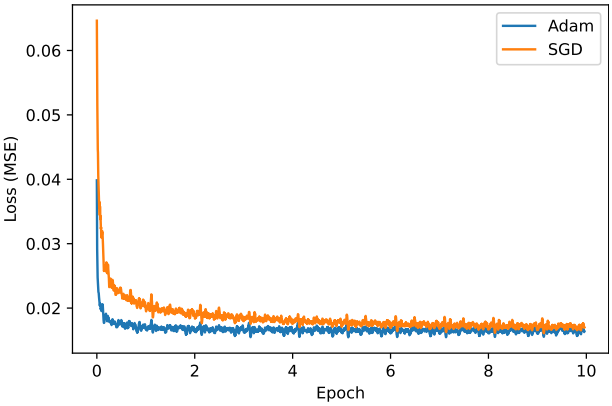

Figure 3: MSE loss as a function of epochs, using Adam or SGD optimizers. Each point in the plot correspond to a moving average of the loss over over 10 batches, with batch size 32.

References

François Fleuret. 2022a. Mini-project for EE-559.

François Fleuret. 2022b. Parameter initialization lecture notes.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural net-

works. In *AISTATS*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Torch Contributors. 2019. torch.nn.conv2d.