Deep Learning, Mini-projects II

Noise2Noise model: image denoising network trained without clean references

Writing our own modules

Authors:
Steven Brown, Guillaume Briand and
Paulin de Schoulepnikoff

Professor:
Prof. François Fleuret

May 26, 2022

1 Introduction

The goal of this second miniproject is to build our framework for denoising images without using autograd and torch.nn modules. In order to achieve this, a framework with convolution and upsampling modules, ReLU and sigmoid activations has been implemented with forward and backward passes. Our framework also uses the Mean squared error loss as well as Stochastic gradient descent in order to minimise the loss.

2 Design and implementation

Here, the aim is to design modules and an optimiser in a similar way to the Pytorch implementation, with some modules inheriting from other classes.

2.1 Module class

The Module class is the parent class of the following classes : Convolution layer, Upsampling, the activation functions (ReLU and sigmoid), MSE loss function and the Sequential. It sets the main structure for the different modules. By default the module class has a forward pass, backward pass and param method. The param method returns a list containing pairs of the module's parameters and gradients. This will be useful for accessing and updating the module's parameters.

For our project we will consider 4 dimensional tensor inputs, which is standard for the tensor representation of sequences of images. The forward and backward pass will therefore be implemented according to this type of input but also accept single images (3 dimensonal tensor input).

2.2 Convolution layer

In order to compute the forward and backward passes of the convolution layer, 2 special operation from Pytorch are used : `fold` and `unfold`. With these operators, forward and backward passes can be evaluated as linear operations, avoiding loops. For the forward pass, we used the same implementation as the one given in the appendix of the project description, using `unfold`.
The gradient of the weights can be seen as a convolution operation between the input and the loss gradient from previous layer. Thus we can use the same operations as for the forward pass.
The gradient (with respect to the input) can be seen as a convolution between the loss gradient and the filter, and one can use `fold` to retrieve the good dimension (same as the input).
The weights and the bias are initialised with a Xavier normal distribution, close to the one Pytorch use for their initialization.
We also add two functions that are used in the SGD module : *set_params* to update the weights and *zero_grad* to set the gradients to zero.

2.3 Upsampling layer

The Upsampling layer is usually implemented with the transposed convolution, but in this mini-project it has been chosen to use a combination of Nearest neighbour upsampling (or NNUpsampling) and the convolution layer from section 2.2.
Nearest neighbour upsampling takes as an input a scale factor (by default 2) and return a tensor with an image scaled up by this factor. The nearest neighbour mode is used, as shown

in Equation 1.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \tag{1}$$

For the backward pass, we take as input the gradient of the loss and return the sum over the nearest neighbours with the scale factor used. Equation 2 shows how it has been implemented.

$$\begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 8 \\ 12 & 16 \end{bmatrix} \tag{2}$$

The implementation of the forward pass for the Upsampling simply consists of the forward pass of NNUpsampling followed by the forward of the 2d convolution. The 2d convolution uses as input the output of the NNUpsampling's forward. For the backward, it is the opposite.

Since only the convolution has parameters, the only parameters for the upsampling layer will be the weight and the bias. In our code, the module is implemented as **TransposeConv2d**.

2.4 Activation functions : ReLU and Sigmoid

The activation functions ReLU and Sigmoid have the standard implementation with no parameters, the forward and backward passes to compute the gradient with respect to input.

2.5 Loss function : Mean squared error

In order to implement the *loss.backward()*, we have chosen that the MSELoss class is linked to the Sequential. With this it accumulate the gradients for each layer in their *self.weight_grad*. The function use the *Sequential.backward()* function with the gradient of the loss as input.
To compute the loss between the predicted value and the target, we use python *__call__* function to call directly the class and not a forward pass.

2.6 Optimiser : Stochastic gradient descent

As for the MSELoss, the optimiser needs to have access to the sequence of modules. This is similarly implemented to the MSELoss. During the initialisation of the optimiser, the sequential is passed as an argument and it's address is therefore saved.

The optimiser has the tasks of setting the gradients to zero before the training of a batch and also updating the parameters according to the calculated gradients. For our project, the stochastic gradient descent (SGD) was selected. The SGD iterates over the modules contained in the sequential and accesses the parameters and their gradient from the *param*() method of each module. With the *set_params*() method in convolution the weights and biases are then updated according the classical gradient descent with learning rate lr.

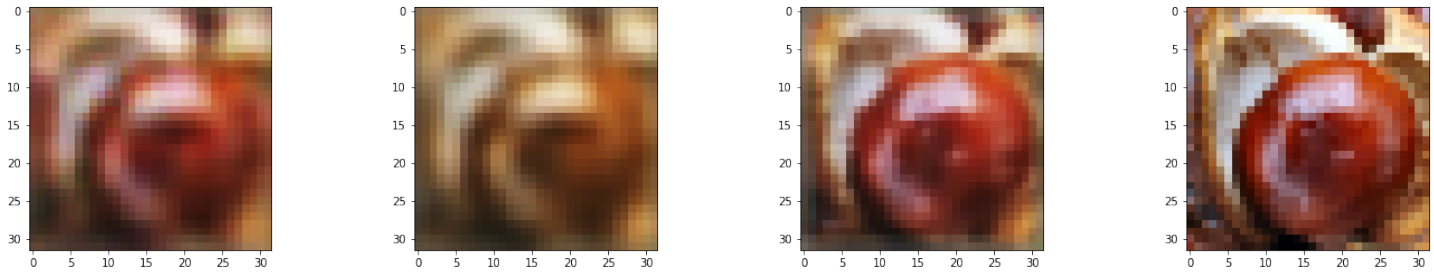
2.7 Container : Sequential

For the Sequential class, we wanted to have as input a list of modules to build the neural network and then being able to call the function for the forward pass. We also add a *backward* function that use the gradient of the loss computed with MSELoss class and then accumulates

the gradients of each layer. This function is used in *loss.backward()* as mentioned above. Two other functions have been implemented to deal with the parameters : *params* to return the weight and bias as well as their gradients, *set_params* to set the weights and biases (it is used in the *load_pretrained_model()* function).

3 Results

For our model we used the network architecture proposed in the project description. Kernel sizes, padding, stride and channel sizes have been tweaked in order to obtain the same image dimension as output.
The **Model** class has been slightly changed compared to the first mini-project. We chose to not run our system on a CPU since a *.to* method equivalent to the pytorch *.to* would have been necessary. The *load_pretrained_model()* and *save_model()* functions have been modified to work with our framework. We also use `pickle` to save the result in `bestmodel.pth` and load them from the same file.
Figure 1 shows how our framework performs for denoising images in comparison to Pytorch nn modules and the clean image. One can see that our denoised image looks like the clean image but blurry. The overall noise2noise achieves a low PSNR of 20.6 dB. However the Pytorch image (b) shows similar results with our framework with PSNR of 20.4 dB. This shows that our framework must be a big limiting factor. One would need to implement more layers as in the first miniproject. Nevertheless, our framework needs 45 minutes on a CPU to train our model with 8 epochs on the full dataset, while Pytorch implementation only need 2 minutes for the same condition. Figure 1c has a reduced noise with PSNR of 21.4 and was implemented with the same framework but with nn.ConvTranspose2d.



(a) (b) (c) (d)

Figure 1: Example of a denoised image with our framework (a), with pytorch and the same network (b), Pytorch with nn.ConvTranspose2d (c) and the clean one (d).

4 Improvements and final notes

We could further improve our framework by implementing the transpose convolution which is much faster and more efficient than Upsampling. Moreover, one can also further improve our modules with the ability to use the GPU. Finally the model we use is quite simple and a more evolved one, as done in the first Mini-project or in the Noise2noise paper, could increase the PSNR and the quality of the denoised images. As a final remark, this project shows the usefulness of the pytorch library which shouldn't be taken for granted. The modules are well optimised and avoid having to calculate manually all the gradients.

References

[1] Sutskever, Ilya and Martens, James and Dahl, George and Hinton, Geoffrey, *On the importance of initialization and momentum in deep learning*, PMLR, 2013, https://proceedings.mlr.press/v28/sutskever13.html

[2] Fangyu Zou and Li Shen and Zequn Jie and Ju Sun and Wei Liu, *Weighted AdaGrad with Unified Momentum*, 2019, arXiv:1808.03408

[3] Diederik P. Kingma, University of Amsterdam, OpenAI , Jimmy Lei Ba, University of Toronto, *adam: a method for stochastic optimization*, Published as a conference paper at ICLR 2015, 30 Jan 2017 , arXiv:1412.6980v9

[4] Glorot, A. Bordes, and Y. Bengio, *Deep sparse rectifier neural networks*, In International Conference on Artificial Intelligence and Statistics (AISTATS), 2011.

[5] Krizhevsky, I. Sutskever, and G. Hinton, *Imagenet classification with deep convolutional neural networks*, Neural Information Processing Systems (NIPS), 2012.

[6] L. Maas, A. Y. Hannun, and A. Y. Ng, *Rectifier nonlinearities improve neural network acoustic models*, ICML Workshop on Deep Learning for Audio, Speech and Language Processing, 2013.

[7] He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, CoRR, abs/1502.01852, 2015.

[8] Xu, N. Wang, T. Chen, and M. Li, *Empirical evaluation of rectified activations in convolutional network*, CoRR, abs/1505.00853, 2015.

[9] Clevert, T. Unterthiner, and S. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*, CoRR, abs/1511.07289, 2015.

[10] simonyan and zisser 2014

[11] Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.

[12] Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*, pages 237–243. IEEE Press, 2001.

[13] Balduzzi, M. Frean, L. Leary, J. Lewis, K. Wan-Duo Ma, and B. McWilliams, *The shattered gradients problem: If resnets are the answer, then what is the question?*, CoRR, abs/1702.08591, 2017