Deep learning - miniproject II

Hilsberg Hendrik, Pillet Maxime, Savioz Baptiste

27th May 2022

1 Introduction

The goal of this project is to implement an image denoiser (as for the miniproject I) but without using autograd and torch.nn modules. Thus we had to implement the different modules of a convolutional neural network ourselves. The different modules to implement are : convolutions and upsampling as layers, as well as ReLU and Sigmoid as activation function. Moreover, we used Mean Square Error for the loss function and Stochastic Gradient Descent as optimizer.

2 Implementation

We structured our code around these different blocks as suggested in the project's description. MSE is implemented as a class with a forward and backward function. Then the layers and activation functions are implemented as independent class with the following architecture :

— *forward* : computes the forward pass
— *backward* : calculates, saves and returns the derivatives
— *train* : updates the weights and biases with the gradient descent
— *param* : returns the weights and biases and their derivatives
— *load* : load the pre-trained weights and biases and their derivatives

This, in order to build a sequential module that can work with an arbitrary configuration of those simple modules. The sequential has the same functions. In addition, we didn't implement the stochastic gradient descent as a separated module, but instead the gradient descent is directly implemented in the layer with trainable parameters, and the stochastic part comes from the batches used in the model training. This simplified the overall architecture of our implementation and avoid back and forth movement of parameters for training.

2.1 Convolution

The convolution can be seen as a linear transformation of the input. Thus in theory, it could be achieved by a simple matrix multiplication if we can arrange correctly our matrices. This is where the difficulty lies, but hopefully the unfold function from pytorch greatly simplifies this.
First, the input tensor of the convolution a^l needs to be unfolded so we can use the linearity of the convolution and a simple matrix product with our weight. We use the "view" method on the weight and bias in order to have the right shape, and we reshape everything after the operation is done to recover the wanted shape of the output tensor.
For the backward pass of the convolution, we start by computing the derivative of the loss with respect to the bias. From the chain rule, $\frac{dL}{db^l} = \frac{dL}{dz^l}\frac{dz^l}{db^l}$, with z^l being the output of the convolution. In 1D, it is

straightforward that $\frac{dz^l}{db^l} = 1$ thus $\frac{dL}{db^l} = \frac{dL}{dz^l}$. In 4D, we just have to sum $\frac{dL}{dz^l}$ over the right dimensions. we compute the sum over the last two dimensions of the tensor (containing the values of the gradient), before performing a sum for each images of the batch.
Then, we compute the derivative of weights with respect to the loss $\frac{dL}{dw^l}$. By carefully deriving $\frac{dz^l}{dw^l}$, one may find eventually that it is equivalent to a convolution between $\frac{dL}{dz^l}$ and the input of the forward pass. Alternatively, one can use results from calculus which shows similar conclusion. To perform this convolution we proceed as for the forward pass.
Finally we have to compute the loss gradient from the layer, $\frac{dL}{da^l}$, in order to back propagate it to the other layers. In this case, we can benefit from the function fold which is the reverse function from unfold, used in forward pass. We used this between the kernel and the input of the forward pass, but in reversed order. As for the forward pass, we have to take care of the shape and dimensions.

2.2 Upsampling

For the upsampling layer, we decided to implement it using a Nearest Neighboor upsampling followed by a convolution. The forward propagation of the NN upsampling simply consists of upsampling according to a given scale factor, using the *repeat_ interleave(scale_factor, dim=3)* function followed by the same command but in dimension 2 (corresponding to increasing in the lines and columns dimensions respectively). Regarding the backpropagation, we unfold $\frac{dL}{dx^{l-1}}$ in order to perform the downsampling. We use the scale factor of the upsampling as a parameter to get a convenient shape (unfold, parameters "kernel size" and the stride). We still have to reshape the unfolded tensor, because the shape of the unfolded tensor is $(N, C * SF^2, L)$, with C being the number of channels of the input tensor, and SF the scale factor. But we only want to sum SF^2 shape for the downsampling. Afterwards, we do a last reshape to get back to the right dimensions and we have $\frac{dL}{dx^l}$. This class has no learning parameters so the learning function is empty. Then, the final upsampling layer is the combination of NN upsampling and convolution.

2.3 ReLU

The forward implemention of the ReLU activation function is pretty straightforward. We just multiply each pixel by '0' if it is smaller than zero and let the value of the pixel if its equal or greater than '0'. This is done using *zl*(zl > 0)*, with zl the input tensor of the activation layer. For the backward propagation, we calculate the derivative of the function evaluated for the input. The train function is also empty, since there is no learnable parameters.

2.4 Sigmoid

This activation function is based on the same reasoning than for the ReLU activation but with the definition of the sigmoid.

2.5 Mean Square Error

We implemented MSE as loss function. To do so, we calculated the mean of the difference for the forward pass and the derivative of this for the backward. Again, the learning function is empty.

2.6 Stochastic Gradient Descent

The optimizer used for this project is SGD. It's used only in the convolution, which is the only module with trainable parameters. In our case, this gave us the following python expression :

```
self.weight = self.weight - self.dL_dw*eta
self.bias = self.bias - self.dL_db*eta
```

with dL_dw and dL_db the derivative of the loss with respect to the weights and bias respectively and eta being the learning rate.

3 Results

As a result, we obtain a PSNR score of 24.07 dB with the simple architecture imposed. After tuning the parameters we ended up with the following respective values : a batch size of 50 images, a learning rate of 2 and 20 epochs. Moreover, the parameters of the layers are :

layer	activation function	in channels	out channels	kernel size
Conv2d	ReLu	3	32	(3,3)
Conv2d	ReLu	32	64	(5,5)
NearestUpsampling	ReLu	64	32	(5,5)
NearestUpsampling	Sigmoid	32	3	(3,3)

Table 1 – Network parameters

As for the miniproject I, we observed a decrease of the loss with the number of epochs. We checked the result of the manually implemented convolution since it was the most complex part of this project. It gives the same results as the one from pyTorch.
The total training time is under 10 minutes using the GPU's from google colab. Our implementation should be working on all type of device, with and without GPU.

4 Conclusion

To conclude, we manually implemented different blocks which allowed us to better understand how these operations work. Moreover, the final result is satisfying for us since we reached a PSNR score pretty high for this kind of network.