Mini-project 2: Mini deep-learning framework development and application on Noise2Noise

Cezary Januszek, Tikhon Parshikov, Michael Roust
Deep Learning, EE-559, EPFL, Switzerland

***Abstract*—In this paper, we revise the approach we adopted to build the Noise2Noise model in part 1, by going more into depth and creating our own framework to create such models. Understanding how all these layers are implemented from scratch is a difficult task, but worth it to be able to fully get the work done behind denoising neural networks.**

I. Introduction

The main goal of this second part was to build, train and test a convolutional neural network built from scratch and apply it to the Noise2Noise [1] task. We focused on several modules and explored them in depth to be able to build them without using the existing Pytorch framework (expect its tensor implementation). In particular, that meant not using autograd or existing nn modules. In order to do that, we were restricted to basic tensor operations as well as the *fold* and *unfold* functions from *torch.nn.functional*.

II. Data set

The data set used in this part of the project is the same as the one for the first part. We have a training set made with 50000 pairs of noisy images. Each image has 3 channels and width 32 and height 32. Each of these pairs provided correspond to downsampled, pixelated images. A validation data set is also provided, with noisy images and its clean counterparts.

III. Our framework

Using our own framework, the goal was to build a functional denoising network, consisting of a Convolutional layer, Upsampling layer, ReLU and Sigmoid activation functions. To put together all these layers, a sequential container was also needed to create the main model, as well as a loss function and the Stochastic Gradient Descent optimizer step for training.

Each of the following modules has an initialization function which initializes the modules parameters and create the parameters gradients needed for the back-propagation. Then, except for the SGD optimizer, all the modules also have a *forward* function that applies the layer or activation function on a tensor input and give an output that will be used by the next layer in the model. The *backward* pass of these modules takes the gradient of the w.r.t the output and updates the gradients of the modules parameters. Lastly, each module also have a *param* function which returns a list of tuples consisting of the module parameter tensor and its gradient.

A. *Convolutional layer*

Our convolutional layer (Conv2d) as the original Pytorch implementation takes into account the following parameters: number of input channels, number of output channels, kernel size, stride and padding. We did not take into account the dilation, as we did not use at all this parameter neither for this part nor part 1.
When implementing this layer, the first step we had to decide on was how to initialize its parameters: the weights and the bias. The natural solution was to initialize those exactly like in the Pytorch implementation - using a uniform distribution U(-k, k), where

$$k = \frac{groups}{C_{in} * \prod_{i=0}^{1} kernelsize[i]}$$

and groups=1 by default.

The Kaiming initialization [2] is usually recommended when using ReLU activation functions and to prevent the vanishing or exploding gradient problem. We tested the use of the Kaiming normal and uniform initialization but in both cases, we got worse results during training and inference than with our initial initialization.
The forward pass of the Conv2d layer was implemented using the fold and unfold operations on tensors. These two operations take care on their own of the kernel size, padding and stride parameters. The input had to be first unfolded and then a linear operation could be applied to it using the weights and bias of the model. To make this forward pass coherent with the different parameters the module supports (kernel size, stride, padding), the correct output size has to be ensured, by reshaping the resulting matrix. Using the hints provided in the Pytorch implementation of this module, the required output size is the following:

$$H_{out} = \lfloor \frac{H_{in} + 2 * padding[0] - kernel[0]}{stride[0]} + 1 \rfloor$$

$$W_{out} = \lfloor \frac{W_{in} + 2 * padding[1] - kernel[1]}{stride[1]} + 1 \rfloor$$

The forward pass could have been also implemented using for loops on the tensors, but this matrix multiplication approach using unfold/fold is more performant. The input to the forward pass has also to be saved to be later used in the backward.
In the backward pass, the gradient w.r.t the output passed as input to the function is first reshaped to match the shape of the linear operation result form the forward, the input o the forward is also loaded and unfolded. With these elements the weight and bias gradients can be updated, by computing the gradient of loss w.r.t to the weights and the bias accordingly. The backward pass outputs the gradient w.r.t the input, by applying first a linear operation on the backward pass input and then a fold to match the required shape.
The *param* function returns simply the the tuple of the weights and its gradient and the bias and its gradient.

B. *Upsampling layer*

The Upsampling layer was implemented with the Transpose Convolutional layer, similar to the ConvTranspose2d from Pytorch. The parameters of the module were initialized in the same way as in the Convolutional layer described above. The forward pass could have been observed to be similar to the backward pass of Conv2d. A linear operation with the weights of the module is first applied on the input and the result of it is folded next. The folded output has to be then reshaped to match the output size:

$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel[0]$$

$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel[1]$$

In the end the bias is added to the output and reshaped accordingly. In this case again, the input to the forward has to be stored to be later used in the backward
The backward pass on the other hand looks more similar to the forward pass of Conv2d. The gradient w.r.t the output taken as input to the backward pass, has to be first unfolded and then a linear operation with the weights is applied on it to give the output - gradient w.r.t to the input, that will be later used in the chain rule when doing the backward pass through the whole network. The weights and bias gradients are accumulated similarly to what occurs in the convolution. Lastly the *param* function of the module returns the same module parameters and their gradients as in Conv2d.

C. *ReLU*

ReLU is the first of the two activation functions implemented for the model. Its forward pass simply applies the ReLU function on the input it takes.

$$ReLU(x) = max(0, x)$$

The backward pass on the other hand, computes the derivative of the function, as follows:

$$\frac{dReLU}{dx} = 0 \quad if \quad x \leq 0 \quad and \quad \frac{dReLU}{dx} = 1 \quad if \quad x > 0$$

Backward outputs then the product of the computed derivative with its input (gradient w.r.t the output). As the activation layers in our model have no parameters, there are no parameter gradients to be updated here.
The *param* function returns an empty array.

D. *Sigmoid*

The Sigmoid layer applies the Sigmoid function in its forward pass and computes the backward pass using its derivative.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{dSigmoid}{dx} = Sigmoid(x) * (1 - Sigmoid(x))$$

Like for ReLU, it has no module parameters, thus its *param* function outputs an empty list.

E. *Sequential container*

The Sequential container is the module that assembles all the layers in the model in a sequential manner. The forward pass of this module takes the input tensor an pass it to the forward step of the first layer. Then, each layer one by one applies its corresponding forward pass on the output of the previous one. The last layer returns the output of the sequential module.
In the backward pass the same applies, but in the reversed order. The loss gradient w.r.t to the output is given to the last activation function, which applies backward on it. Then its output it passed to the layer in the model just before it.
The *param* returns the tuples (parameter, gradient of parameter) of all the layers in the module. Additionally, we decided to add a function *appendLayer*, which adds a layer to the end of the sequential list of the module layers.

F. *Loss function - Mean Squared Error*

The loss function used in the model is the Mean Squared Error. Its module forward pass applies the loss function to the input, which in this case consists in two tensors (the obtained tensor and the target). The backward simply computes the derivative of the same function.

G. *Stochastic Gradient Descent*

The SGD is our optimizer module. It takes the model as input, as well as the number of epochs, the learning rate, the mini batch size and the loss the user chooses. Then it performs the training steps for the chosen number of epochs, on a mini batch, by doing a forward pass, computing the loss and doing a backward pass through the model. Once the backward pass is done, our optimizer call the *step* function to update the weights and biases of the layers. Once all the training is done it outputs the resulting model with the optimized parameters for all layers.

IV. Our model

With the modules described in the previous section we created a model with the following layers in our Sequential container:

- Conv2d (3, 32, 3, stride=2)
- ReLU
- Conv2d (32, 64, 2, stride=2)
- ReLU
- Upsampling(64, 32, 2, stride=2)
- ReLU
- Upsampling(32, 3, 2, stride=2)
- Sigmoid

We experimented with different numbers of channels in the hidden layers. Testing with 8, 16, 32, 64 in the first Conv2d and 16, 32, 64, 128 in the second Conv2d respectively. We found the above configuration of 32 output channels in the first Conv2d and 64 in the second Conv2d to have the best trade-off between training time and accuracy. Larger values gave negligible performance increases for a significantly larger training time. The number of channels in the Upsampling layers reflect the numbers decided for the Conv2d layers, as we want to have our output image back in the same dimensions and number of channels as we passed in as input to the model. Thus, we also use the stride parameter in the Upsampling layers to be sure to obtain the same dimensions as we got when going through the convolutions.

A. Training

Training was performed using our implementation of SGD described above. Again we optimized the parameters of SGD to achieve maximum performance.

We found a learning rate of $1 \cdot 10^{-3}$ the best. Larger learning rate values achieved lower performance as SGD wouldn't converge as well to the right parameters. Lower values increased training time without reducing significantly the training loss.

In our opinion, mini-batch size only improves the training time as it doesn't affect much the mathematical behavior of the algorithm. Several mini-batch sizes were tested from 10 to 200, but the value of 100 was found to be the most reasonable choice. We started training our model, with 100 epochs, but quickly noticed that after more than 30 epochs the model converged, we therefore decided to limit the training to 30 epochs.

Unfortunately, when training the model, we notice that we might have a vanishing gradient issue that we did not manage to solve. In some cases that occurs around 20 epochs and the loss does not decrease from that point. That also makes our psnr metric on the predictions always be around 12.5dB. We tried to identify the issue

B. Performance

Our fully trained model achieves a PSNR of around 12.7 db on the test set. This comes as a bit of a concern, as it was expected to achieve a PSNR of 22-23dB. This is due to the problem mentioned before we encountered during training and that we were unfortunately not able to solve on time.

Our model from mini-project 1 achieves a higher performance and better training time as PyTorch is a mature library with much more code running in C++ than in python as our model does. Using similar approaches we could far improve our custom implementation.

V. Conclusion

The custom implemented model we build leaves place for further improvement. We are aware of the issues we faced during the implementation of its main modules, but those also helped us a lot to understand deeper the functioning of such frameworks. Overall, our model is almost functional and correctly applies the forward pass. The backward pass also works in most cases, but it can be that fails at some place making our training do what it does now. This model sets a good basis for implementing a fully functional denoiser and can motivate for further development and improvement of parameters.

References

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

[2] S. R. J. S. Kaiming He, Xiangyu Zhang, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015. [Online]. Available: https://arxiv.org/pdf/1502.01852v1.pdf