Deep Learning Projet 2 Report

Julian Blackwell, Erwan Serandour, Ghali Chraibi

May 2022

1 Introduction

The aim of this project is to build our own framework for denoising images without using *autograd* or `torch.nn` modules. We build our own modules and network building blocks such that we can use them to train a deep learning architecture on some training data given a loss function. We decide to stay as close as possible to Pytorch interface for ease of use.

2 Implementations

Each component inherits the Module class and has to implement a forward, a backward and a param method.

2.1 Activation functions

The forward pass for a given activation function $\sigma(x)$ simply corresponds to itself applied to its input. Generally, we denote the input of the activation function at layer l as $\mathbf{s}^{(l)} = \left(\mathbf{w}^{(l)}\right)^\intercal \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}$ where $\mathbf{w}^{(l)}, \mathbf{x}^{(l-1)}, \mathbf{b}^{(l)}$ are respectively the weights of layer l, the output of layer l−1, and the bias of layer l.
In the backwards pass, we propagate the derivative of the loss with respect to the activation:

$$\frac{\partial \ell}{\partial \mathbf{s}^{(l)}} = \frac{\partial \ell}{\partial \mathbf{x}^{(l)}} \sigma'(\mathbf{s}^{(l)})$$

The backward pass will take as input $\frac{\partial \ell}{\partial \mathbf{x}^{(l)}}$ and must be therefore multiplied by $\sigma'(\mathbf{s}^{(l)})$, where $\mathbf{s}^{(l)}$ was the input to the forward pass.

2.1.1 ReLU

This block implements the $\textsc{ReLU}(x) = \textsc{max}(0, x)$ activation function. Its derivative is:

$$\textsc{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

For the forward pass, we simply apply ReLU element-wise on the input, and observe that we need to save the input to compute the backward pass: the derivative is propagated if the input was positive, otherwise it is set to 0.

2.1.2 Sigmoid

This block implements the $\sigma(x) = \frac{1}{1+e^{-x}}$ activation function. Its derivative is:

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x))$$

For the forward pass we apply σ to the input which we also save for the backward pass (for which we will need to compute σ').

2.2 Sequential

This Module implements a container inspired from `torch.nn.Sequential` to put together an arbitrary configuration of modules together. The container is created by specifying the desired sequential configuration of modules in its constructor. The forward pass then delivers some input to the first underlying module whose output is then fed to the next module. This is done until the final module is reached, where its output is then returned. The backwards pass works similarly, with now the input (gradient) is fed to the last underlying module and this time outputs propagated backwards.

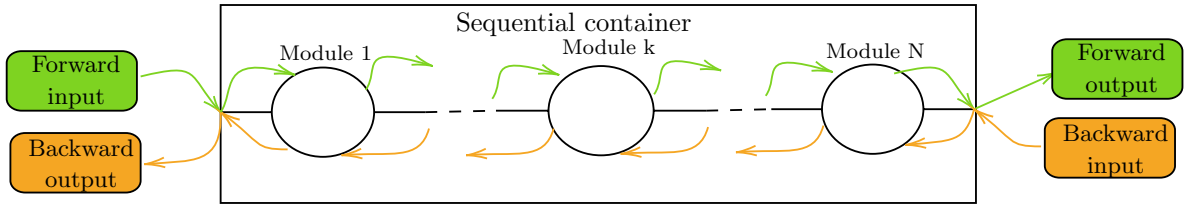

Figure 1: Sequential container with N underlying modules

2.3 Convolution

We add a block that implements similar functionality as the `torch.nn.Conv2d` module. This module has as a weight and bias parameter of shapes (C_o, C_i, K_0, K_1) and (C_o) respectively where C_o, C_i are the number of output and input channels, and the convolutional kernel is of size $K_0 \times K_1$. It takes as input a batch of N tensors of shape (N, C_i, H, W) where C_i is the number of channels, and H, W are the channels' height and width.

Under the hood, we consider the convolution operation as a matrix multiplication. For the forward pass, the 2d convolution is emulated by a simple linear transformation: we resize the weight parameter into a $C_o \times C_i K_0 K_1$ kernel matrix $\mathbf{W}$. We multiply this kernel matrix by the block input unfolded by `torch.nn.fucntional.unfold` $\mathbf{x}$ and add the bias $\mathbf{b}$. This method extracts the sliding blocks of the input tensor given the kernel size, padding and stride parameters. Multiplication of the kernel matrix and the sliding blocks is computationally identical to convolution, only shaped differently: $(C_o, N H_o W_o)$ where H_o, W_o are the height and width of the convolved channels (which depend on convolution parameters). This can then be appropriately resized to the desired output of shape (N, C_o, H_o, W_o). The unfolded input is also saved as it is needed for gradient computations.

Since we considered convolution as a linear transformation, we similarly apply the backward pass as from a linear layer, only with added resizing steps at the start and end. It is equivalent to multiplying the transposed kernel matrix by the incoming loss derivative. The result is then resized by `torch.nn.functional.fold` to combine the sliding blocks back into their original shape. To summarize, the forward and backwards pass can then be seen as:

$$\texttt{Linear}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \quad ; \quad \frac{\partial l}{\partial \mathbf{x}^{(l)}} = \mathbf{W}^\intercal \frac{\partial l}{\partial \mathbf{s}^{(l)}}$$

where $\frac{\partial l}{\partial \mathbf{s}^{(l)}}$ is loss derivative with respect to the activation of the layer l (which lies right after `Linear`). Finally, we also accumulate the loss derivative w.r.t. the weight and bias which will be used in an optimization step to update the layer with new weights and bias: $\frac{\partial l}{\partial \mathbf{W}} = \mathbf{x}^{(l)\intercal} \frac{\partial l}{\partial \mathbf{s}^{(l)}}$, $\frac{\partial l}{\partial \mathbf{b}} = \mathbf{1} \frac{\partial l}{\partial \mathbf{s}^{(l)}}$.

2.4 Upsampling and Transpose convolution

We then add an Upsampling block, which really just is a transpose convolution block under the hood, similar to Pytorch torch.nn.ConvTranspose2d. The backward pass of transposed convolution is equivalent to the forward pass of the convolution, and vice versa. Therefore its implementation is very similar to regular convolution, and also emulates the operation as a matrix multiplication. This module is very important in taks where we need to have an output of the same shape as the input (such as noise2noise or GANs). Indeed, it allows to upsample the output of the convolution layers.

2.5 MSE Loss function

We also add a block which implements mean squared error (MSE) as a loss function. Our model will aim to minimize this quantity:

$$\text{MSE}(\mathbf{y}, \mathbf{target}) = \frac{1}{N} \sum_{i=1}^{N} (\text{target}_i - \text{y}_i)^2$$

where $\mathbf{y}$ is the vector of model outputs and **target** is the vector of target labels (both of length N). We consider the error vector $\mathbf{e} = \mathbf{target} - \mathbf{y}$. Then we can re-write the MSE function as a function of error: $\text{MSE}(\mathbf{e}) = \text{MEAN}(\mathbf{e}) = \frac{1}{N} \sum_{i=1}^{N} (\text{e}_i)^2$. The derivative is then:

$$\text{MSE}'(\mathbf{e}) = -\frac{2}{N} \sum_{i=1}^{N} \text{e}_i$$

Therefore, the forward pass takes two inputs: the model outputs and the labels. It then accordingly computes the MSE, and saves the error vector which is needed in the backward pass. The loss does not depend on any other input so one simply propagates the gradient as defined above.

2.6 SGD Optimizer

The optimizer is a special block as it is the only one that does not inherit from the Module class.

Indeed, it is not a layer of the model, but rather describes how an optimization step is done. Here we implement Stochastic Gradient Descent (SGD).

Our optimizer is composed of two methods. First, a `zero_grad()` method to set the gradients of all optimized tensors to 0. This must be called at the start of a backpropagation step to reset tensors that accumulate parameter gradients who already served to update the parameters at a previous step. Then, a `step()` method which performs the gradient descent step:

$$g_t \longleftarrow \nabla f_t(\theta_{t-1}) \quad ; \quad \theta_t \longleftarrow \theta_{t-1} - \gamma g_t$$

where $f(\theta)$ is the objective function to minimize, θ_0 are the parameters, and γ is the learning rate, which we pass as an argument at construction.

We also added optional functionality to the optimizer such as weight decay, momentum where we update the velocity of all the parameters in the model to incorporate a percentage μ (the momentum) of the previous gradients in the current gradient and dampening to offer similar arguments as the `torch.optim.SGD` module.

3 Results

We tested our framework on a noise2noise task with the same dataset used in the first project. We train a given sequential model to validate our implementation. Its architecture is as follow:

```
Sequential(
    Conv2d(3, 64, (4,4),
        stride=2, padding=2),
    ReLU(),
    Conv2d(64, 256, (4,4),
        stride=2, padding=2),
    ReLU(),
    Upsampling(256, 128, (5,5),
        stride=2, padding=2),
    ReLU(),
    Upsampling(128, 3, (4,4),
        stride=2, padding=2),
    Sigmoid()
)
```

For the training, we use a learning rate of 0.5 for the first 10 epochs, then a learning rate of 0.1 for 30 epochs. It is a surprisingly high learning rate, but having tested with other smaller learning rate, the training was too slow. We achieve a PSNR of 21.89 db proving our implementation works.

4 Conclusion

In this project, by implementing a small deep learning framework, we have learned to better understand how Pytorch works under the hood and some of the design choices they made. We were able to demonstrate the capability of our framework by producing a model that improves the quality of noisy images in a noise2noise task.

4.1 Limitations

While we were tuning our model, we observed that we needed a surprisingly high learning rate to obtain good performances. This may indicate a vanishing gradient problem even though our model is not very deep. This could be because we are not using Xavier's initialization when creating our tensors.

Another limitation we faced was due to the constraints given in the context of this project. As a specific structure was required, some choice of implementation could have been more adequate. For example, we would have liked that the `param` method of the modules return a dict or an object rather than a list of tuples. This way, we could have directly paired parameters with their velocity, gradient, and additional characteristics in a more robust way, instead of storing velocities in the optimizer as an example.

4.2 Future Work

This framework is a solid basis to implement a variety of neural networks. Due to the nature of the project, we mainly focused on modules related to convolutional neural networks, especially autoencoders. However, we would still need to implement several other components to build state-of-the-art models such as ResNet or U-Net that we implemented in the first project. Among the missing components, the ability to add skip connections would allow to construct much deeper architecture, but it is not possible with our current Sequential container.