Manual implementation of a ConvNet

Matteo Calafà, Paolo Motta, Thomas Rimbot
Second project for the Deep Learning (EE-559) course at EPFL Lausanne, Switzerland

***Abstract*—This second part consists in the explicit implementation of the main blocks for a simple convolutional network. In other words, we aim at implementing blocks which could replace the `PyTorch` modules and allow to define a standard network as a sequence of those. While this does not present any practical utility because of the lower computational performance with respect to optimized `PyTorch` modules, this project's goal is to clearly comprehend how convolutional networks work and some basic ideas about their implementation in the most common libraries.**

I. Introduction

The goal of this project is to explicitly implement a few modules that are useful for the construction of a simple convolutional neural network. These modules are: Sigmoid, ReLU, 2D Convolutional Layer, Nearest Neighbor Upsampling. In addition to that, we aim to implement a standard optimizer (SGD) and a standard loss (MSE) in order to provide a fully functional Deep Learning pipeline. Finally, to define the network as a sequence of modules, we will define a class named `Sequential` presenting very similar characteristics with the one provided by `PyTorch`.

II. General aspects about the definition and construction of the network

A. Definition of the network

The network we aim to implement is shown in Figure 1:

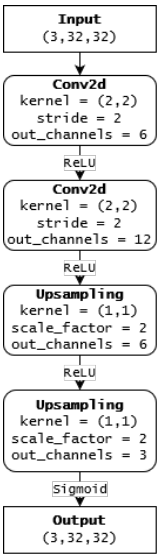

Fig. 1: Diagram of the network.

`Upsampling` can be defined as the composition of a Nearest Neighbor Upsampling and a 2D Convolutional Layer (respectively the equivalent of `UpsamplingNearest2d` and `Conv2d` in `PyTorch`).

B. The `Module` class

Every module has different inner tensors and will define three main methods:

1) `forward()`: to perform the forward pass.
2) `backward()`: to compute the gradient of the loss with respect to the input of the module, given the gradient with respect to the output.
3) `param()`: to return the module parameters and the gradient of the loss with respect to the same parameters (it is empty for parameterless modules, e.g `ReLU`).

In addition, we aim to implement the MSE loss as another class with the `forward()` and `backward()` methods (in this case, the latter will not accept any input) and the SGD optimizer with the `step()` method.

Finally we need a definition of the `Sequential` class to wrap the different blocks together and build the network. Its overridden `forward()` and `backward()` methods are explained in section subsection II-C.

C. Basic functioning of the network

We stress again that every module/block contains various tensors which keep in memory the latest update of parameters such as weights, gradients, inputs and outputs. The training is based on the usual following steps:

1) Forward pass: from the current weights, each block computes the output from an input tensor. Therefore, the `Sequential` class can compute the output of the network by sequentially applying the `forward()` methods from all its building blocks. During this step, each block/module saves in memory its input and output.
2) Forward pass of the loss: from the output of the net, the MSE module can compute the error thanks to its `forward()` method.
3) Backward pass of the loss: the `backward()` method of the MSE module returns the gradient of the loss with respect to the output of the net.
4) Every block can compute the loss gradient with respect to its input from the gradient with respect to its output. Therefore, starting from the gradient returned by the loss, all the gradients can be recursively computed with the `backward()` methods from all the blocks, called

in reverse order. This is indeed the definition of the overridden `backward()` method in `Sequential`.

5) Optimization step: from the gradients with respects to the states, we can compute the gradients with respect to the weights/parameters through the `param()` methods of each module. Having the current values and gradients of all the weights and biases, the `SGD` class can perform the optimization step via its `step()` method.

III. Operating definitions of the modules

In this section, we present how `Conv2d` and `Upsampling` explicitly perform their forward operation. The remaining blocks follow instead the standard definitions. Before proceeding, we explicit the following nomenclature:

- I and O are the input and output tensors of a generic layer.
- W and B are the weight and bias tensors of a convolutional layer.
- $\mathcal{L}$ indicates the loss evaluated in a current state.
- B, C, N_X, N_Y are the batch size, the channels number and the x-y lengths of the image at a generic layer.

A. Nearest Neighbor Upsampling

The goal of this module is to increase the tensor's spatial size (based on a `scale_factor` parameter) by filling it with repeated values of the original tensor. In particular, let us assume that the original shape of the tensor is (B, C, N_x, N_y). If `scale_factor` = k, the output tensor's shape should be (B, C, kN_x, kN_y). The Nearest Neighbor Upsampling fills up the newly added $k \times k$ squares with the corresponding original values in the input tensor. We refer below a 2D example, inspired from the PyTorch documentation page:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \xrightarrow{\text{k=2}} \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \tag{1}$$

B. `Conv2d` as a linear layer

One can prove that convolution is in fact a linear operation. However, we a priori do not have access tp suitable tensors for the linear products. Rather, they need a specific construction. We claim that there exist reshaped versions of the previously mentioned tensors (called $\tilde{W}, \tilde{B}, \tilde{O}$) such that:

$$\tilde{O} = \tilde{W} \otimes U + \tilde{B} \tag{2}$$

which, in components, it corresponds to:

$$\tilde{O}_{p,q,r} = \sum_m \tilde{W}_{q,m} U_{p,m,r} + \tilde{B}_{0,q,0} \tag{3}$$

where U is the unfolded version of I (see `PyTorch`'s `unfold()` method). Therefore, the following forward and backward operations for the `Conv2d` block will be based on the equivalent linear operation in Equation 2.

IV. Computation of the gradients

A. MSE loss

The definition of the MSE loss for a CNN is:

$$MSE(T, O) := \frac{1}{B \cdot C \cdot N_X \cdot N_Y} \cdot \sum_{B,C,N_X,N_Y} (T_{B,C,N_X,N_Y} - O_{B,C,N_X,N_Y})^2$$

where T and O are the target and output tensors. Therefore, the gradient can be easily computed as:

$$\frac{\partial \mathcal{L}}{\partial O} = \frac{2}{B \cdot C \cdot N_X \cdot N_Y} \cdot \sum_{B,C,N_X,N_Y} (T_{B,C,N_X,N_Y} - O_{B,C,N_X,N_Y})$$

B. Sigmoid

Since the forward operation is by definition $O = \sigma(I)$ (component-wise), where:

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

one can easily compute the gradient with respect to the input with:

$$\frac{\partial \mathcal{L}}{\partial I} = \frac{\partial \mathcal{L}}{\partial O} \odot \sigma'(I)$$

where $\odot$ indicates the Hadamard product and the derivative of the sigmoid can be directly computed with:

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

C. ReLU

By definition, the layer forward operation is $O = \text{ReLU}(I)$, where:

$$\text{ReLU}(x) := \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Hence, as before, the gradient with respect to the input can be computed as:

$$\frac{\partial \mathcal{L}}{\partial I} = \frac{\partial \mathcal{L}}{\partial O} \odot H(I)$$

where, indeed, the Heavyside function H coincides with the ReLU subderivative.

D. Nearest neighbor upsampling

The general idea of the forward operation was already presented in section III. To compute the backward pass, we instead need to sum the derivatives in each $k \times k$ square following intuitively the concept of *weight-sharing*. This intuition can also be mathematically demonstrated by the chain rule:

$$\frac{\partial \mathcal{L}}{\partial I_{i,j}} = \sum_{p,q} \frac{\partial \mathcal{L}}{\partial O_{p,q}} \frac{\partial O_{p,q}}{\partial I_{i,j}}$$

However, following Equation 1, $\frac{\partial O_{p,q}}{\partial I_{i,j}}$ can only have value 1 when (p, q) are in the (i, j) k-square (=: $M^k_{i,j}$) and 0 otherwise. Therefore:

$$\frac{\partial \mathcal{L}}{\partial I_{i,j}} = \sum_{p,q \in M^k_{i,j}} \frac{\partial \mathcal{L}}{\partial O_{p,q}}$$

as expected.

E. Conv2d - gradient of states

Gradients with respects to the input I can be computed component-wise from Equation 3, which implies:

$$\begin{gathered}\frac{\partial \mathcal{L}}{\partial U_{i,j,k}} = \sum_{p,q,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,q,r}} \frac{\partial \tilde{O}_{p,q,r}}{\partial U_{i,j,k}} \overset{(3)}{=} \\ \sum_{p,q,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,q,r}} \tilde{W}_{q,j} \delta_{p,i} \delta_{r,k} = \sum_{q} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{i,q,k}} \tilde{W}_{q,j}\end{gathered} \tag{4}$$

This component-wise operation can be computed with a direct tensor product, being careful to transpose some components. Moreover, notice that $\partial \mathcal{L}/\partial \tilde{O}$ can be easily obtained by reshaping $\partial \mathcal{L}/\partial O$. The last missing step is the computation of the gradient, not with respect to the unfolded input but to the very input instead. The `unfold()` operation creates a new tensor filled with the same original values but repeated multiple times. Therefore, to compute the last step, we once again need a *weigh-sharing* operation as in subsection IV-D. Fortunately, `PyTorch` provides a method called `fold()`, which consists in the inverse of the `unfold()` operation, with the additional multiplication for the repetition times. Therefore, we can directly go from $\partial \mathcal{L}/\partial U$ to $\partial \mathcal{L}/\partial I$ using `fold()`. Schematically:

$$\boxed{W, \frac{\partial \mathcal{L}}{\partial O} \overset{\texttt{reshape}}{\rightarrow} \tilde{W}, \frac{\partial \mathcal{L}}{\partial \tilde{O}} \overset{(4)}{\rightarrow} \frac{\partial \mathcal{L}}{\partial U} \overset{\texttt{fold}}{\rightarrow} \frac{\partial \mathcal{L}}{\partial I}}$$

F. Conv2d - gradient of weights and biases

With similar calculations, we get:

$$\begin{gathered}\frac{\partial \mathcal{L}}{\partial \tilde{W}_{i,j}} = \sum_{p,q,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,q,r}} \frac{\partial \tilde{O}_{p,q,r}}{\partial \tilde{W}_{i,j}} \overset{(3)}{=} \\ \sum_{p,q,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,q,r}} U_{p,j,r} \delta_{q,i} = \sum_{p,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,i,r}} U_{p,j,r}\end{gathered} \tag{5}$$

which can be implemented with a standard tensor product if tensors are before properly translated and reshaped. Then,

$$\begin{gathered}\frac{\partial \mathcal{L}}{\partial B_i} = \frac{\partial \mathcal{L}}{\partial \tilde{B}_{0,i,0}} = \sum_{p,q,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,q,r}} \frac{\partial \tilde{O}_{p,q,r}}{\partial \tilde{B}_{0,i,0}} \overset{(3)}{=} \\ \sum_{p,q,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,q,r}} \delta_{q,i} = \sum_{p,r} \frac{\partial \mathcal{L}}{\partial \tilde{O}_{p,i,r}}\end{gathered} \tag{6}$$

wihch can be implemented with a sum over the first and third component. At this point, we show the sequential steps to get the final gradients:

$$\boxed{\frac{\partial \mathcal{L}}{\partial O} \overset{\texttt{reshape}}{\rightarrow} \frac{\partial \mathcal{L}}{\partial \tilde{O}} \overset{(5)}{\rightarrow} \frac{\partial \mathcal{L}}{\partial \tilde{W}} \overset{\texttt{reshape}}{\rightarrow} \frac{\partial \mathcal{L}}{\partial W}}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial O} \overset{\texttt{reshape}}{\rightarrow} \frac{\partial \mathcal{L}}{\partial \tilde{O}} \overset{(5)}{\rightarrow} \frac{\partial \mathcal{L}}{\partial B}}$$

G. SGD

In order to actually update the network parameters from the gradients, we implement the SGD optimizer in the `SGD` class. In particular, this class provides a `step()` method, whose role is to loop through the model's modules and recursively update their parameters by calling their `update_params()` methods. The update rule is the SGD one, namely going in the opposite direction of the gradient with a tuneable step size:

$$p_{t+1} = p_t - \gamma \nabla_{p_t} \mathcal{L} \tag{7}$$

V. Training, testing, saving and loading

The `Model` class is the main interface for using our modules. It implements the sequential structure represented in Figure 1, stored in its `self.model` attribute, and defines the loss criterion (`MSE`) and optimizer (`SGD`). It also provides a `train()` method, responsible for running the full training loop. This method iterates on the input samples by batches, performs the prediction with `model.forward()`, computes the loss and the gradients with the `backward()` methods, and updates the parameters with the `optimizer.step()` method.

This class also provides support for saving and loading the model. Through the `load_pretrained_model()` and `save_pickle_state()` methods, the parameters and their gradients in each modules can be saved in and loaded from a pickle file *bestmodel.pth*.

VI. Conclusion

In order to assess the correctness of our implementation, these blocks have been compared to the `PyTorch` ones, both individually and sequentially. After careful examination, we concluded that the implementation was in fact correct, and attained an almost complete similarity with the library's performance. However, we point out that because of computational details, which are unrelated to the mathematics described in this report, it is very hard to actually get a fully precise implementation. Some examples are the floating point precision for digits, or different implementations of the same mathematical expressions, which have been proven to change results. In fact, this has been apparent when using an equivalent definition for the sigmoid function and a strict inequality instead of an inequality condition for the ReLU.
Regarding the implementation of the CNN in Figure 1 for a Noise2Noise model, we obtained decent results and a PSNR of 19.5, by training the model for about two minutes on a CPU. While the score in the first part of the project is much higher, we remind that the proposed network as well as its optimizer are of very low complexity and could be more finely tuned to get better performances. Moreover, the higher computational times resulting from the manual implementation of the blocks instead of modules taken from optimized libraries prevented us from running large simulations. Despite this, we confirmed the correctness of the implementations and mathematical calculations.