DL Miniproject 2 - Implementing our own Modules

PHILIPPE, Jocelyn (288228) jocelyn.philippe@epfl.ch
DEVAUD, Quentin (287452) quentin.devaud@epfl.ch
KREMER, Iris (337635) iris.kremer@epfl.ch

I. Introduction

The goal of this mini-project 2 is to understand in depth the computations and implementation of convolutional neural networks by re-implementing some modules of the pytorch framework from scratch, using only tensors and their methods. In particular, we are implementing the following modules and classes:

- `Conv2d`: a 2D convolution layer
- `Upsampling`: a layer composed of nearest neighbor upsampling + convolution
- The `ReLU` activation function
- The `Sigmoid` activation function
- The `Sequential` model container
- `MSE`: the mean-squared error function
- An `SGD` optimizer

II. Code architecture

In this project, we try to follow the pytorch architecture as closely as possible. Therefore, we create a superclass `Module` with methods `forward()`, `backward()`, `param()`, `update_param()` and `zero_grad()`. All subclasses of `Module` are required to implement `forward()` and `backward()` methods, so we raise a `NotImplementedError` by default for these methods. Additionally to the modules, we also implement the `SGD` optimizer in a separate class, as well as the `Model` itself in which we implement training, loading and evaluation of our demonstration model.

III. Implementation

A. Conv2d

Our 2D convolution layer supports tensor inputs of size `(B, C_in, H_in, W_in)` or `(C_in, H_in, W_in)`, where $B = batch\_size$, $C\_in = in\_channels$, $H\_in = in\_height$ and $W\_in = in\_width$.

Mandatory parameters to pass to the convolution are `in_channels`, `out_channels` and `kernel_size`. Our implementation additionally supports the presence or absence of a `bias`, and arbitrary `stride`, `dilation` and `padding`. The only restriction is that the stride must be element-wise smaller than or equal to the kernel size, as implementing support for strides strictly greater than the kernel size was too complicated in the backward pass and deemed useless, since a stride greater than the kernel size would imply that a grid over the image would not be convolved at all.

For its implementation, we create the class `Conv2d` which subclasses `Module`, and implement the methods `forward()` and `backward()`. Additionally, we override the methods `param()` to return the weight and bias values of the layer along with their respective gradients, `update_param()` to update the weights and biases with new values, and `zero_grad()` to zero the gradients of the weights and bias. The weights of the convolution are of size `(C_out, C_in, k1, k2)`, where $C\_out = out\_channels$, $k1 = kernel\_height$ and $k2 = kernel\_width$, and the bias is of size (C_out). Weights and biases are initialized following the Pytorch initialisation, i.e. from a uniform distribution with interval between $-k$ and k, where k is defined as

$$k = \frac{1}{C_{in} \prod_{i=0}^{1} kernel\_size[i]}$$

Note that we have $groups = 1$, because we do not have this parameter in our layer arguments.

1) The forward pass: takes a tensor input of size `(B, C_in, H_in, W_in)` or `(C_in, H_in, W_in)`. It convolves the kernel, i.e. the weights of the layer, over each image in the input tensor, and adds the bias if any, then returns the output of this convolution. Convolution of a kernel K of size `(C_in, k1, k2)` over an image I of size `(C_in, H_in + k1 - 1, W_in + k2 - 1)` is defined in [1] as following:

$$(I \circledast K)_{ij} = \sum_{c}^{C_{in}} \sum_{m=0}^{k1-1} \sum_{n=0}^{k2-1} K_{c,m,n} I_{c,i-m,j-n}$$

Therefore, the `forward()` method outputs a new tensor of size `(B, C_out, H_out, out_width)`, where

$$H\_out = \frac{(H + 2p[0] - d[0](k[0] - 1) - 1)}{s[0]} + 1,$$

$$W\_out = \frac{(W + 2p[1] - d[1](k[1] - 1) - 1)}{s[1]} + 1$$

(where $k = kernel\_size$, $p = padding$, $d = dilation$, $s = stride$) and similarly for W_out, or without batch size dimension if the input does not have one.

2) The backward pass: takes the gradient with respect to the output of the current layer l, denoted $\frac{\partial L}{\partial x_l}$, as input, and computes the following three gradients:

$$\frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial x_l} \frac{\partial x_l}{\partial w_l} = \frac{\partial L}{\partial x_l} x_l$$

$$\frac{\partial L}{\partial b_l} = \frac{\partial L}{\partial x_l}\frac{\partial x_l}{\partial b_l} = \frac{\partial L}{\partial x_l}$$

$$\frac{\partial L}{\partial x_{l-1}} = \frac{\partial L}{\partial x_l}\frac{\partial x_l}{\partial x_{l-1}} = \frac{\partial L}{\partial x_l} w_l$$

where L is the loss, w is the weights of layer l, b its bias and x_{l-1} the input to layer l. The gradients with the respect to the weights and to the bias are stored as attributes in the layer, while the gradient with respect to the input is returned by the backward pass, such that it can be passed to the previous layer to compute its gradients.

To obtain $\frac{\partial L}{\partial w_l}$, we need to convolve $\frac{\partial L}{\partial x_l}$ by x_l. This is because $\frac{\partial L}{\partial w_l}$ has each of its entries defined by the sum of gradients. This is implemented as a convolution of $\frac{\partial L}{\partial x_l}$ over x_{l-1}. Similarly, to compute $\frac{\partial L}{\partial x_{l-1}}$, we must convolve w_l over $\frac{\partial L}{\partial x_l}$. As for the bias gradient $\frac{\partial L}{\partial b_l}$, we simply need to sum $\frac{\partial L}{\partial x_l}$ over all the batches, height and width, keeping only the channels dimensions.

To implement support for the stride of the forward pass in the backward pass, we have to adapt x_{l-1} for the computation of $\frac{\partial L}{\partial w_l}$ by cutting away the parts of x_{l-1} that were not convolved at all during the forward pass due to the stride, since their contribution to the gradient is 0. The cut-off q for height and width respectively is calculated as

$$q_{height} = (H_{in} - k1) \bmod s[0],$$

$$q_{width} = (W_{in} - k2) \bmod s[1]$$

We then convolve the cut version of x_{l-1} with $\frac{\partial L}{\partial x_l}$ using a dilation equals to the stride of the backward pass [2]. As for the computation of $\frac{\partial L}{\partial x_{l-1}}$, we dilate $\frac{\partial L}{\partial x_l}$ by the stride before convolving it with w_l [3].

We implement support for dilation following [4], and for padding support, we simply pad x_{l-1} with zeros when convolving it with $\frac{\partial L}{\partial x_l}$, and remove some padding from $\frac{\partial L}{\partial x_l}$ when convolving it with w_l.

B. Upsampling

The `Upsampling` layer subclasses `Module` and is composed in our case of a nearest neighbor upsampling layer called `NNUpsampling` followed by a `Conv2d` layer. It overrides the methods `forward()`, `backward()`, `param()`, `update_param()` and `zero_grad()` to correctly forward the inputs and outputs of the layers.

We chose to implement the nearest neighbor upsampling followed by a convolution, because this combination avoids the typical checkerboard artifacts that are often created by transpose convolutions, especially when the kernel size is not a multiple of the stride [5].

`NNUpsampling` subclasses `Module` and has a single argument `scale_factor` determining by how much it upsamples its input. In the forward pass, it therefore takes an input tensor of size `(B, C_in, H_in, W_in)` or `(C_in, H_in, W_in)` and outputs a tensor of size `(B, C_in, H_in * scale_factor, W_in * scale_factor)` or `(C_in, H_in * scale_factor, W_in * scale_factor)`. The backward pass consists of computing $\frac{\partial L}{\partial x_{l-1}}$ by summing $\frac{\partial L}{\partial x_l}$ in windows of size $scale\_factor \cdot scale\_factor$ to aggregate the gradients to which each input pixel has contributed. This is implemented as a convolution with C_in kernels of size $scale\_factor \cdot scale\_factor$, each with values of 1 for a single channel and 0 otherwise.

The `Upsampling` layer is implemented to emulate a transpose convolution, therefore it takes arguments equivalent to a transpose convolution, i.e. `in_channels`, `out_channels` and `kernel_size` are required arguments, and optional arguments `stride`, `padding`, `dilation` and `bias`. It also has two additional arguments `transposeconvargs`, by default `True`, and `scale_factor`, by default `None`. When `transposeconvargs` is `True`, the arguments passed to the Upsampling layer are treated as transpose convolution arguments and mapped to different argument values for the `NNUpsampling` and `Conv2d` classes such that the size of the output of the `Upsampling` layer matches the size of the output of a transpose convolution layer with the same arguments. In our implementation, this only works perfectly if the dilation argument is set to 1, because we were unable to find how to properly map this argument to match the output shape of a transpose convolution. The mapping is performed as follows (left the final argument values for `NNUpsample` or `Conv2d`, on the right arguments passed to `Upsampling` or new value):

- `scale_factor` = `stride * dilation`
- `stride` = `dilation`
- `padding` = `dilation * (kernel_size - 1) - padding`
- `dilation` = 1

and the remaining arguments keep their values. If `transposeconvargs` is set to `False`, the `scale_factor` argument is required, and all arguments passed to the `Upsampling` layer are treated as direct arguments for the `NNUpsampling` and `Conv2d` layers according to the nomenclature used for their respective arguments.

C. ReLU

`ReLU` subclasses `Module` and overrides the methods `forward()` and `backward()` only. The forward applies the ReLU activation function defined as following to the input:

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

The backward pass receives as always $\frac{\partial L}{\partial x_l}$ returns the derivative of the loss with respect to each input, defined as

$$\frac{\partial L}{\partial x_{l-1}} = \frac{\partial L}{\partial x_l}\frac{\partial x_l}{\partial x_{l-1}}$$

and $\frac{\partial x_l}{\partial x_{l-1}}$ is the derivative of $ReLU(x_{l-1})$ with respect to x_{l-1}, which gives one for positive entries and zero otherwise.

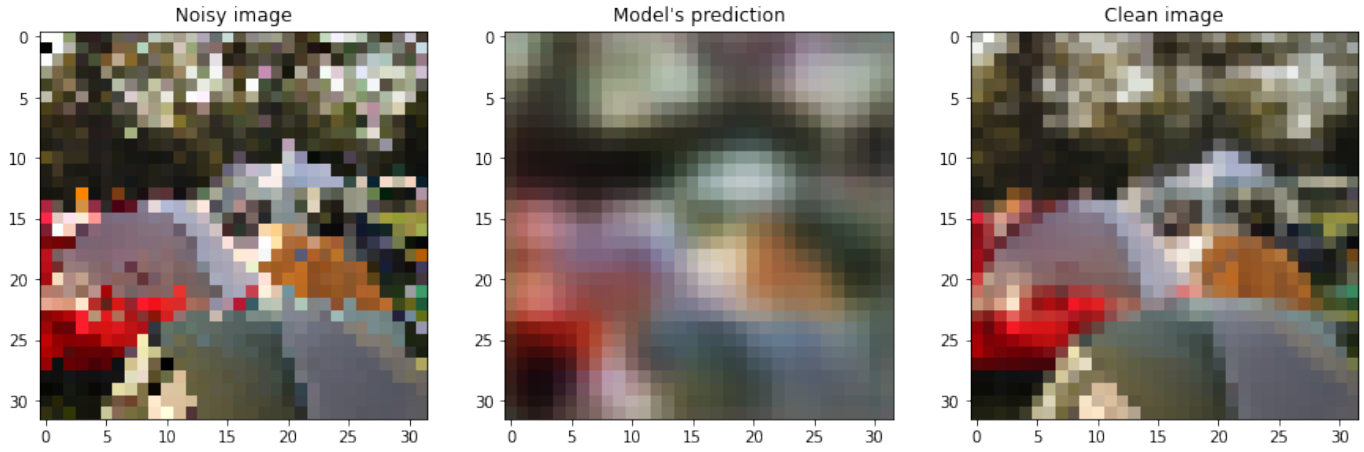

Fig. 1: Example of output of the sample model.

D. Sigmoid

`Sigmoid` subclasses `Module` and overrides the methods `forward()` and `backward()` only. The forward applies the sigmoid activation function defined as following to the input:

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

Just like for `ReLU`, the backward pass of `Sigmoid` follows the same equation to compute $\frac{\partial L}{\partial x_{l-1}}$.

E. MSE

The mean-squared error (MSE) loss function is implemented in `MSE`, also by subclassing `Module` and overriding `forward()` and `backward()`. The forward pass computes the loss according to the function

$$MSE(x, y) = \frac{1}{N} \sum (x - y)^2$$

where $N = height \cdot width \cdot n\_channels \cdot batch\_size$, as is done by default in the torch MSE module. The backward pass returns the derivative of the MSE: $\frac{2}{N}(x - y)$, normalised by the same term as in the forward.

F. SGD

The `SGD` class is a standalone and takes a single argument `lr` for the learning rate, set by default to $1e-2$. We implemented a single method `step()` which takes as argument a list of pairs `(parameter, parameter_gradient)` and computes the updated value for each of them by descending a step of size `lr` in the direction opposite to the gradient following the formula $parameter = parameter - lr \cdot parameter\_gradient$.

IV. Demonstration of our results

As requested, we build the simple network containing two `Conv2d`, two `Upsampling`, three `ReLU` and one `Sigmoid`. The architecture and parameters are the following:

```
Sequential(
    Conv2d(in_channels=3, out_channels=6, stride=2, padding=1, dilation=1, kernel_size=3),
    ReLU(),
    Conv2d(in_channels=6, out_channels=9, stride=2, padding=1, dilation=1, kernel_size=3),
    ReLU(),
    Upsampling(factor=2, in_channels=9, out_channels=6, kernel_size=3, transposeconvargs=False),
    ReLU(),
    Upsampling(factor=2, in_channels=6, out_channels=3, kernel_size=3, transposeconvargs=False),
    Sigmoid()
)
```

Listing 1: Layers of our model

The optimisation algorithm is the implemented `SGD` with a varying learning rate and no momentum, since we did not implement this feature. We use `MSE` loss, with a batch size of 32, and 15 epochs. The training set containes 50,000 images, from which 20% are randomly assigned to validation set. The results obtained are displayed in table I

Learning rate	MSE	PSNR
0.05	0.0227	20.24
0.1	0.0198	21.09
0.2	0.0228	19.78
0.3	0.0207	20.75
0.4	0.0207	20.59

TABLE I: Validation MSE and PSNR scores for different learning rates.

However the best model that we have was trained on 10 epochs with a learning rate of 0.3 and a batch size of 32. We reached a validation loss `MSE = 0.01941` and `PSNR = 21.38` It took 10min 39s to train it on an M1 Mac. Figure 1 shows an example of noisy image, output of our model and ground truth. Although the output is very blurry, the amount of noise is significantly reduced.

References

[1] T. Dardoize, "Forward and backward propagation for 2d convolutional layers," https://towardsdatascience.com/forward-and-backward-

propagations-for-2d-convolutional-layers-ed970f8bf602, accessed 26th of May 2022, online.
[2] M. Kaushik, "Part 1: Backpropagation for convolution with strides," https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-8137e4fc2710, accessed 26th of May 2022, online.
[3] ——, "Part 2: Backpropagation for convolution with strides," https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-fb2f2efc4faa, accessed 26th of May 2022, online.
[4] A. Agrawal, "Back propagation in dilated convolution layers," https://www.adityaagrawal.net/blog/deep_learning/bprop_dilated_conv, accessed 26th of May 2022, online.
[5] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, 2016. [Online]. Available: http://distill.pub/2016/deconv-checkerboard