EE-559 Miniproject 2 – Spring 2022

Ali Garjani (Sciper: 336721), Sepehr Mousavi (Sciper: 338673), Hossein Taji (Sciper: 322030)

I. Introduction

The core part of this project lies on creating a deep learning framework with train and test capabilities, without relying on PyTorch's autograd functionalities. The use of PyTorch is only restricted to tensor operations and tensor methods. An exception is the use of `torch.nn.functional.fold` and `torch.nn.functional.unfold` The implemented modules are the `ReLU` and the `Sigmoid` activation functions, the `MSE` loss, 2D convolutional layer (`Conv2d`), 2D transposed convolutional layer (`TransposeConv2d`), a wrapper which we call `Sequential` (as the name in PyTorch's module), and the `SGD` optimizer. These modules are available in `./Miniproject_2/others/framework.py`. In the following sections, the methodologies and the details of implementations are described, the validation of the framework is presented, and the results of using the framework for training a convolutional neural network (CNN) with image denoising purposes are presented.

II. Implementations

All the implemented modules except the SGD optimizer inherit from the base class `Module`. The `__init__` method should get the configurations and initialize the parameters and their gradients, if any. The `forward` method of these modules take a tensor as input, calculate the output, store the input, and return the output. The `backward` pass of these modules take as input the gradient of a scalar with respect to its output, calculate the gradient of the same scalar with respect to its input (stored in the forward pass) and with respect to its parameters (if any), accumulate the latter, and return the former. The `param` method should return a list of pairs of parameters and their gradients (empty if no parameter), the `zero_grad` method should clear the parameter gradients, and the `cuda` method should move the parameters of the module and their gradients to GPU.

A. Activation Functions

The ReLU (Rectified Linear Unit) and the sigmoid activation functions are implemented. The ReLU activation function is defined as $ReLU(x) = max(0, x)$, and its gradient is a unit step function (heaviside). The sigmoid activation function is defined as $\sigma(x) = \exp(x)/(1 + \exp(x))$, and its derivation yields $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. The forward passes of the `ReLU` and `Sigmoid` classes are trivial. In the backward passes, the chain rule is applied to calculate the gradient with respect to the input from the gradient with respect to the output. These two modules don't have any parameters.

B. MSE loss

The mean squared error (MSE) defined as $\frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)^2$ is implemented in the `MSE` class, where X is the predicted tensor and Y is the target tensor. As this loss function gets two tensors as input, the backward pass also returns two tensors, the gradient with respect to each of the inputs. Since the output of the loss function is usually a scalar that is supposed to be minimized, passing a gradient tensor is optional when calling the backward pass of this module. This module does not have any parameter.

C. 2D Convolutional Layer

Just like its counterpart in PyTorch (`torch.nn.Conv2d`) [2], the `Conv2d` class gets as a setting, the number of input channels (C_{in}), the number of output channels (C_{out}), kernel size ($h \times w$), stride ($s_0 \times s_1$), padding ($p_0 \times p_1$), and dilation $d_0 \times d_1$. The parameters (weight and bias) of the module are initialized with samples from the uniform distribution $\mathcal{U}(-\sqrt{k}, +\sqrt{k})$, where $k = C_{in}hw$. The accumulated gradients with respect to the parameters are also initialized with zeros. The dimensions of each variable are presented in Table I. The relation between the output dimensions and the settings is available in [2]. The main calculations are done in a different representation of the tensors, which is indicated with an underscore in the variable's name.

Variable Name	Shape
`input`, `gradx`	(N, C_{in}, H_{in}, W_{in})
`output`, `grad`	(N, C_{out}, H_{out}, W_{out})
`weight`, `gradw`	(C_{out}, C_{in}, h, w)
`bias`, `gradb`	(C_{out},)
`input_`, `gradx_`	(N, $C_{in}hw$, $H_{out}W_{out}$)
`output_`, `grad_`	(N, C_{out}, $H_{out}W_{out}$)
`weight_`, `gradw_`	(C_{out}, $C_{in}hw$)
`bias_`	(1, C_{out}, 1)

Table I: Dimensions of the variables corresponding to `Conv2d`.

In the forward pass, the `unfold` method changes the representation of `input`, to get the variable `input_`. Using this new representation we can simply apply convolution using matrix multiplication. If we consider the matrix of this tensor for one sample (first dimension), each column (second dimension, $C_{in}hw$) contains the elements of the kernel applied on all channels of the original tensor. The

convolution is applied on this variable to get the output by calculating `output_` $=$ `weight_@input_+bias_`, where @ indicates matrix multiplication. It is notable that this matrix multiplication is broadcasted in the first dimension of `input_`, and adding the bias is broadcasted in the first and the third dimensions. The final `output` is then created with reshaping this variable.

In the backward pass, the gradient of the loss with respect to the output, `grad`, is reshaped to get `grad_`. The unfolded gradient of the loss with respect to the input is calculated as `gradx_` $=$ `weight_`^{T} @ `grad_`, where the T superscript denotes the transpose of the tensor. Applying the `fold` operation on this tensor yields `gradx` in the same dimensions as the input. Similarly, the gradient of the loss with respect to the weights, `gradw_`, is calculated by the summation of `grad_` @ `input_`^{T} in the first dimension. The summation is due to the fact that the weights are broadcasted in the first dimension when calculating the output, as mentioned above. Similarly, the gradient of the loss with respect to the bias, `gradb`, is calculated by summing `grad_` in the first and the third dimensions.

D. 2D Transposed Convolutional Layer

The settings of the `TransposeConv2d` are consistent with its Pytorch counterpart (`torch.nn.ConvTranspose2d`) [3] and similar to the ones of `Conv2d` described in subsection II-C. The stride, padding, and the dilation settings could be passed to this layer. However, the dilation is not implemented and it is restricted to 1×1. The initialization of the parameters is similar to `Conv2d` (uniform), and the dimensions of each variable are listed in Table II. Note that in this table we are considering zero padding for simplicity. When padding is not zero, it is applied on the final output and we have $H'_{out} = H_{out} - 2p_0$ and $W'_{out} = W_{out} - 2p_1$. The detailed relation between the output dimensions and the settings is available in [3].

Variable Name	Shape
`input`, `gradx`	(N, C_{in}, H_{in}, W_{in})
`output`, `grad`	(N, C_{out}, H_{out}, W_{out})
`weight`, `gradw`	(C_{in}, C_{out}, h, w)
`bias`, `gradb`	(C_{out},)
`input_`, `gradx_`	(N, $C_{in}hw$, $H_{out}W_{out}$)
`output_`, `grad_`	(N, C_{out}, $H_{out}W_{out}$)
`weight_`, `gradw_`	(C_{out}, $C_{in}hw$)
`bias_`	(1, C_{out}, 1)

Table II: Dimensions of the variables corresponding to `TransposeConv2d`.

Transposing `weight` in the first and the second dimension, flipping it in the third and the fourth dimension, and reshaping it, gives `weight_`. The flips in this stage correspond to the nature of transposed convolution, where each of the elements of the output is comprised of the summation of the weights multiplied by the elements of the original input in the reverse order. `bias_` is simply a reshaped version of `bias`. To get `input_`, the original input is first dilated by the size of stride. Meaning that, for instance, $s_0 - 1$ zero rows are added between the rows of `input`. The resulting tensor is then padded by $(k_0 - 1) \times (k_1 - 1)$ and is unfolded to get `input_`. The padding is originated in the boundary elements of the output which are not summed up. To get `grad_`, `grad` is padded by $p_0 \times p_1$ and reshaped. The padding here is necessary because the output is padded in the last step of the forward pass, as will be described in the following. This structure allows us to calculate the forward and backward passes very similarly to the forward and backward passes of `Conv2d`.

In the forward pass, `output_` is calculated by `weight_@input_ + bias_`, which is then reshaped and cropped (by $p_0 \times p_1$) to yield `output`. It is notable that the same broadcasting as in `Conv2d` are applied here. The gradient of the loss with respect to the input, `gradx`, is calculated by folding and undilating `gradx_` $=$ `weight_`^{T} @ `grad_`. Undilating refers to do the inverse of dilating, i.e., removing the zero rows and zero columns. The procedure to get the gradient of the loss with respect to the weight and the bias are identical with `Conv2d`, with an extra transposing for `gradw` because of its different shape.

E. The Sequential Wrapper

The `Sequential` module is provided as a wrapper for multiple ordered modules like its Pytorch's counterpart [4]. Let us consider wrapping N modules $f_1(x)$ to $f_N(x)$. The forward pass of the sequential wrapper should return

$$Seq(x) = f_N \circ f_{N_1} \circ \cdots \circ f_1(x). \tag{1}$$

This is done by calling the forward pass of each module and passing its output to the forward pass of the next module in the sequence. Applying the chain rule on Equation 1 gives

$$\frac{\partial Seq}{\partial x}(x) = \prod_{i=1}^{N} \frac{\partial f_i(h_i(x))}{\partial h_i(x)}, \tag{2}$$

where $h_i(x) = f_{i-1} \circ \cdots \circ f_1(x)$ is the output of the previous module in the sequence. The backward pass of `Sequential` calculates this by calling the backward pass of each module in the reverse order of the sequence. Since the input of each module is stored in the forward pass, there's no need to pass the input again.

With this structure, the back-propagation algorithm is applied by calling the forward pass and the backward pass of the model, wrapped in a `Sequential` object. In the forward pass, the input of each layer is stored, and in the backward pass, the stored input and the gradient with respect to the output are used to calculate the gradient with respect to each parameter.

F. SGD Optimizer

The SGD optimizer is implemented in the `SGD` class which gives as input a list of pairs of the parameters to optimize and their accumulated gradients, a learning rate, and a weight decay factor. It's `step` method updates each parameter by subtracting its gradient times the learning rate from it.

III. Validation

To validate our framework, the results of the forward and the backward methods of each module (output and all the gradients) have been compared to the results from their Pytorch's counterpart with random inputs and arbitrary configurations (input dimensions, output channels, kernel sizes, etc.). The test cases are available in `./Miniproject_2/others/testcases.ipynb`. The gradients for the Pytorch modules are calculated with the `torch.autograd.grad` method. All the modules successfully pass the testcases if the configurations are valid (e.g., kernel size can't be bigger than the dimensions of the input tensor).

IV. Convolutional Neural Network

Using the developed framework, a 4-layer convolutional neural network (CNN) is built in `./Miniproject_2/model.py` for image denoising purposes. The details of the layers are listed in Table III. The output dimensions in this table correspond to the input of size $N \times 3 \times 32 \times 32$. The model is trained on a dataset of 50000 pairs of noisy images, based on the conclusions in [1], and validated on a similar validation dataset of 1000 pairs. The MSE loss between the predictions of the model and the ground-truth is used for the loss function. Several optional data augmentation techniques are implemented in the training. First, the training data is shuffled on each epoch. Second, each image and its ground-truth is randomly flipped horizontally, vertically, both, or none. Third, each image and its ground-truth are randomly rotated by one 0, 90, 180, or 270 degrees.

Layer Type	Kernel	Stride	Padding	Output Size
`Conv2d`	3×3	2×2	1×1	$N \times 32 \times 16 \times 16$
`Conv2d`	3×3	2×2	1×1	$N \times 32 \times 8 \times 8$
`TransConv2d`	3×3	2×2	0×0	$N \times 32 \times 17 \times 17$
`TransConv2d`	6×6	2×2	3×3	$N \times 32 \times 32 \times 32$

Table III: Details of each layer of the CNN.

The script `./Miniproject_2/others/run.py` is used for training the network, with the training configurations (e.g., learning rate, features, batch size, CUDA, etc.) set in `./Miniproject_2/others/configs.py`. The framework does a reasonable job by changing the parameters in the right direction, decreasing the training loss, and increasing the validation PSNR score [1]. However, the final validation PSNR score is not good compared to more sophisticated models trained on the same dataset. This could be explained by the simplicity of the structure, since the upsampling layers are not using any information of the previous layers, and the network has a hard time recreating the features of the original image. The network is trained with different kernel sizes and number of features (in convolutional layers), each with different learning rates and batch sizes, and the best obtained validation PSNR score was $14.82dB$, which corresponds to the model described in Table III and the training configurations in `configs.py`. The validation PSNR score before training this structure is around $12.50dB$. The trained model is provided in `./Miniproject_2/bestmodel.pth`, and could be loaded by calling the `Model.load_pretrained_model` method. It is notable that increasing the number of features (32) of the convolutional layers could increase the final PSNR score but training such a network requires more memory resources. To ensure that our implemented modules work correctly, we tested these different architectures and settings using `torch.nn` modules and we got the same results as the ones from our implementation.

V. Conclusion

Using the PyTorch tensor methods and tensor operations, a deep learning framework is developed and validated without relying on the PyTorch's autograd functionalities. We demonstrated how the back-propagation algorithm could be easily implemented using an appropriate architecture, and how the convolutional layers could be efficiently implemented with tensor operations. Training one epoch of the proposed model structure for image denoising takes less than 1 minute on Google Colab GPU's. The optimizer can successfully drive the parameters in the right directions to improve the validation PSNR score from around $12.50dB$ to $14.82dB$ However, in order to be able to train more sophisticated networks which are more suitable for this purpose, further considerations are needed.

References

[1] Jaakko Lehtinen et al. "Noise2Noise: Learning Image Restoration without Clean Data". In: *CoRR* abs/1803.04189 (2018). arXiv: 1803.04189. URL: http://arxiv.org/abs/1803.04189.

[2] *Pytorch's Documentation for nn.Conv2d*. https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html. Accessed: 2022-05-27.

[3] *Pytorch's Documentation for nn.ConvTranspose2d*. https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html. Accessed: 2022-05-27.

[4] *Pytorch's Documentation for nn.Sequential*. https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html. Accessed: 2022-05-27.