EE 559 Mini-Project 2: Deep Learning Framework

Hao Zhao, Xufeng Gao, Qiming Sun
EPFL, Switzerland

***Abstract*—In this project, a mini deep learning framework has been developed, which can give sufficient flexibility and functionalities as the standard PyTorch framework. Based on the project instructions, our framework has been built upon PyTorch's basic Data Structure, Tensor without the Auto Differentiation function activated. All required modules, such as Convolution layer, ReLu and MSE loss, etc., are implemented to build a complete neural network and handle different situations in Deep Learning. Finally, these modules are combined into a model trained on noisy image pairs based on Noise2Noise theory. We compared the performance of our framework to the same model built with PyTorch modules.**

I. General Interface

The framework is mainly composed of 7 specific modules, and all these modules inherit from the base class `Module` that defines the following structure.

```
class Module(object):
  def forward(self, *input):
      raise NotImplementedError
  def forward(self, *gradwrtoutput):
      raise NotImplementedError
  def param(self):
      return []
```

There are two types of modules in our framework, called Parametric module and Non-Parametric module. The main difference between them is that the Non-Parametric module contains parameters (e.g., weight and bias) to optimize, while the Parametric module does not. Each specific module will implement the `forward` to perform forward pass, the `backward` to do backpropagation and `params` to get a list of module parameters.

- The `forward` method is used to perform the forward pass of the module, which accepts and returns a tensor or a tuple of tensors. It includes not only the calculations from the input layer to the output layer, but also the storage of all intermediate variables which are helpful in the backward pass.
- The `backward` method is used to run the backward pass of the module, which gets as input a tensor or a tuple of tensors containing the gradient of the loss w.r.t the module's output variables. The intermediate variables saved in forward pass are used together with the method inputs to accumulate the gradient of parameters and calculate the gradient w.r.t the input variables of the module as the method outputs.
- The `params` method returns a list of tuples, each composed of a parameter tensor and a gradient tensor of the same size. For the module without parameters (such as `ReLU`), an empty list is returned.
- The `zero_grad` method provides a beneficial to zero out gradients when building a neural network. For modules without gradient optimization (such as `ReLU`), this method is passed.

II. Implementation

In our framework, Parametric modules include `Conv2d` and `ConvTranspose2d` while modules like `ReLU` and `Sigmoid` are Non-Parametric. To build a complete deep neural network, the `Sequential` module is implemented.

A. *Conv2d*

2D convolution over an input signal composed of several input planes (in_channels). Depending on the out_channels, multiple convolution kernel will be used. Down-sample input with stride parameter is enabled in our implementation. The padding option is also available, meaning both the full and valid convolution are allowed. The methods of this module are as follows:

- *__init__()*: Initialize the layer by creating a weight tensor of size (out_channel $\times$ in_channel $\times$ kernel_size) and optionally a bias tensor of size (out_channel). It provides three different weight initialization based on the string parameter `weightsinit`:
  - `normal`: This is the default initialization similar to PyTorch, where the weights are sampled from a uniform distribution in the range $(-\sqrt{1/fan\_in}, \sqrt{1/fan\_in})$ where $fan\_in = in\_channel * \prod_{i=0}^{1} kernel\_size[i]$
  - `xavier`: The weights are initialized from a normal distribution with mean 0 and standard deviation of $\sqrt{2/(fan\_in + fan\_out)}$, where $fan\_out = out\_channel * \prod_{i=0}^{1} kernel\_size[i]$. This type of initialization helps prevent the gradient from exploding or vanishing.
  - `kaiming`: The weights are initialized from a kaiming normal distribution with mean 0 and standard deviation of $\sqrt{2/(fan\_in)}$. This type of initialization is preferred when using `ReLU` activation for the layers.
  - bias tensors are all initialized to zero.
- *forward*: Given the input batch x, saves it (for further use), then calculates the output of layer l. To simplify

our implementation, the PyTorch function `unfold` is used so that the 2D convolution works as matrix operations, as follows:

$$x_{l+1} = w_l x_l + b_l \tag{1}$$

where w_l and b_l are weights and biases of layer l.

- *backward*: Calculates the loss gradient w.r.t x_l, denoted as dx_l. As suggested in [1], dx_l is the full convolution between a 180-degree rotated kernel and loss gradient from the next layer. With `unfold`, the computation can be simplified as:

$$dx_l = w_l^R dx_{l+1} \tag{2}$$

Moreover, the loss gradients w.r.t the weight and bias (e.g., dw_l and db_l) are accumulated to support parameter updating during training, by following concepts of dw_l is the convolution between x_l and dx_{l+1} and db_l is the sum of dx_{l+1} for each kernel [1].

B. `Upsampling`

We use the transposed convolution as Upsampling layer. Similar to the convolution layer, the parameters of the transposed convolution are set and calculated in a similar way, with the same stride and padding, while in PyTorch the parameters of the convolution and transposed convolution are set in such a way that the convolution layer's output, after passing through the transposed convolution layer with the same parameters, has the same size as the initial input, We noticed this and implemented it in our code.The detailed information is as follows:

- *__init__()*: This approach is essentially the same as the convolution layer, with 2 main differences. The first is the change in the dimension of the weights. In transpose layer, the weight tensor size is (in_channel $\times$ out_channel $\times$ kernel_size), which is the same with *nn.ConvTranspose2d*. The second is the additional parameter of output_padding in order to keep convolution and transposed convolution consistent as we mentioned before.
- *forward*: The forward method is similar to the backward of convolution layer when calculating loss gradient w.r.t x_l. The detailed function is to expand the input tensor according to the output size we want. The combination of stride, padding and output_padding is described in detail in the PyTorch documentation and therefore leads to different methods of expansion. Stride interpolates the input with a value of 0, resulting in an almost stride times larger input size, according to kernel_size While padding is padding, output_padding is also padding, but unlike the previous padding which reduces the size, output_padding only increases the size of the rows and columns on one side. In fact the forward method is equivalent to the convolution of the expanded input tensor with the constant stride of the kernel, so the implementation also uses a similar method to the unfold in the convolution layer forward [2].
- *backward*: The backward approach consists in treating the forward as a linear equation, similar to the convolution layer.

$$dx_l = {w_l}^T dx_{l+1} \tag{3}$$

$$dw_l = dx_{l+1} x_l^T \tag{4}$$

The unfold function used in the forward could also be reversed by function fold.

C. `Sigmoid`

Sigmoide can can avoid gradient explosion. Its forward implementation follows $y = \frac{1}{1+\exp(-x)}$, which maps the input to between 0 and 1. In the backward pass, $dx_{l+1} * (y - y^2)$ is returned.

D. `ReLU` and `LeakyReLU`

The implementation of ReLU is based on $y = x * (x > 0)$, which is the faster one among different alternatives. Compared to `Sigmoid`, it can avoid gradient vanishing because the gradient is always 1 for positive input. To increase the applicability of our model, the Leaky ReLU is also developed which is popular in tasks where we we may suffer from sparse gradients. The implementation of `LeakyReLU` is based on $y = x * (x \geq 0) + \alpha * x * (x < 0)$ where α is a small negative slop determined before training.

E. `MSE`

MSE is used to compute the error between the predicted network output and the ground truth, which is implemented with $\frac{1}{N}\sum(error)^2$, where N is the number of predicted network outputs and $error = output - truth$ for each sample.

F. `SGD`

The Stochastic Gradient Descent is implemented to update the weights and biases of the layers during training, and the following methods are included:

- It is initialized by passing a series of parameters, i.e., the learning rate (η), the gradient tensors and optionally with momentum.
- the `step` method updates parameters using the same form as Pytorch:

$$v_{t+1} = \mu * v_t + g_{t+1},\ p_{t+1} = p_t - \eta * v_{t+1} \tag{5}$$

where p, g, v and μ denote the parameters (e.g., weight and bias), gradient, velocity (that is initialized to 0) and momentum respectively.

G. `Sequential`

As mentioned above, the `Sequential` module is a container module which can combine the forward and backward computations and parameter updates of multiple layers (e.g., multiple sub-modules with their own implementations). It feeds the outputs of sub-modules as inputs to the next modules during forward pass and passes the gradients in the reverse order during the backward propagation.

III. Experiments and Results

As instructed, we build a simple network with our modules to cope the Noise2Noise task, and the corresponding hyper-parameters are listed as following.

```
model = Sequential(
    Conv2d(3, 48, 3, stride=2, padding=1),
    ReLU(),
    Conv2d(48, 48, 3, stride=2, padding=1),
    ReLU(),
    Upsampling(48,48,3,padding=1, stride=2,
    output_padding=1),
    ReLU(),
    Upsampling(48, 3,3,padding=1, stride=2,
    output_padding=1),
    Sigmoid()
)
```

We set a SGD optimizer with learning rate of 0.01, momentum of 0.9 and batch size of 16 to improve our model during the training. The train images have pixels range from 0 to 255, and We convert it to the range of 0 to 1, by dividing by 255, which is the same as required in test.py.

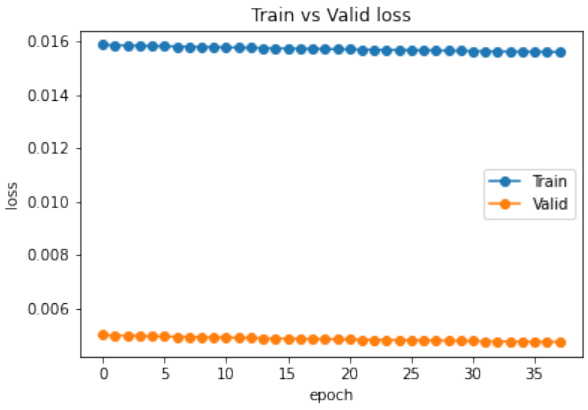

Figure 1. Train and validation loss over epoch

For each epoch, the training process will cost about 10 minutes in CPU, 3 minutes in GPU with Google Colab. The loss of train data and validation data are showed in Fig.1. The loss change are quite small and the PSNR on validation data in Fig.2 are more obvious.

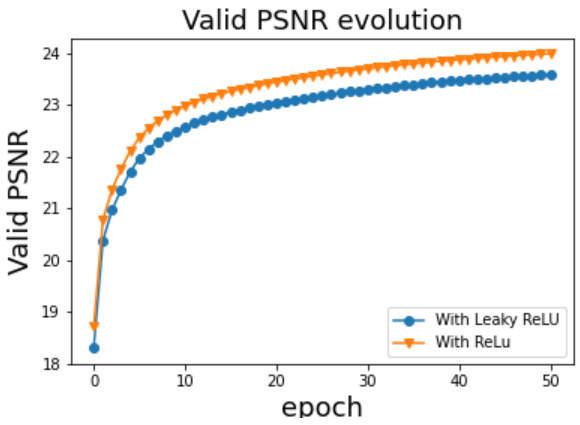

Figure 2. Validation PSNR over epoch

Our model is trained for about 50 epochs, each time calculating the PSNR in the validation set and saving the model if we get a better result. Each training session loads the best saved model from the previous session, and the final result is obtained, Fig.2 shows results of the different training session. The final PSNR we get from our best model is 24.20dB. We also tried another architecture where we changed all ReLU to Leaky ReLU. However, this model gave worse performances (only 23.66dB).

IV. Summary

Overall, we implement the convolution layer, the transposed convolution layer and activation functions such as ReLU and Sigmoid as well as the MSE loss function, build and test the noise2noise model. After about 50 epochs of training, the model finally achieved a PSNR of 24.20 dB. To improve this and get closer to the loss of performance of the model we proposed in mini-project 1 that uses official PyTorch, we could develop more complicated models and implement different optimizers such as AdamW that we used in mini-project 1 to aim to converge faster.

References

[1] P. Solai, "Convolutions and backpropagations," 2018. [Online]. Available: https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

[2] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *ArXiv e-prints*, mar 2016.