Mini Project II: Implementation of PyTorch Modules

1 Implementation of Modules: Base class

All Modules in our implementation are based on the following base class `Module` structure:

```
class Module(object):
    def __init__(self) -> None:
        pass
    def forward(self, *input):
        raise NotImplementedError
    def backward(self, *gradwrtoutput):
        raise NotImplementedError
    def param(self):
        return []
    def to(self, device):
        return self
    def load_param(self, *param, device):
        return None
```

Listing 1: Base class.

- `param()` function returns the list of parameters in a module, e.g. for `Conv2d` and Upsampling, the parameters are `[(self.weight, self.dl_dw), (self.bias, self.dl_db)]`.
- `to(self, device)` function moves the module to the specified device, possible devices are `'cpu'` and `'cuda'`. We used `'cuda'` during training to increase matrix calculation speed (about 30× faster, takes about 5 second per epoch to train our models in Section 8).
- For each module, we also provide a `load_param(self, *param)` to load saved weights back to the modules.

2 Conv2d Module

The major tools we use in this implementation are: `torch.nn.functional.unfold/fold` and `torch.einsum` for multidimensional tensor multiplication. Here we illustrate the forward/backward pass of `Conv2d`.

The forward pass of our `Conv2d` module is in the following order:

1. **Reshape kernel weights**: $[C_{out}, C_{in}, ks, ks] \rightarrow [C_{out}, C_{in} \times ks^2]$.
   Where $C_{in/out}$ stands for numbers of in/output channels and ks stands for kernel_size.
2. **Unfold input tensor**: $[bz, C_{in}, H_{in}, W_{in}] \rightarrow [bz, C_{in} \times ks^2, H_{out} \times W_{out}]$.
   Where bz stands for batch size, and H, W stands for height and width of in/output images. The output of this step is saved in the model class as `self.unfolded` for later use in backpropagation. And the relationship between input and output widths and heights are (we assume hyperparameter inputs are `int` instead of `tuple`):
   $$H_{out} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{ padding } - \text{ dilation } \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$
   $$W_{out} = \left\lfloor \frac{W_{\text{in}} + 2 \times \text{ padding } - \text{ dilation } \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$
3. **Matrix Multiplication $\mathbf{Y} = \mathbf{XW} + \mathbf{b}$**:
   $[bz, C_{in} \times ks^2, H_{out} \times W_{out}] \times [C_{out}, C_{in} \times ks^2] + [C_{out}] = [bz, C_{out}, H_{out} \times W_{out}]$.
4. **Reshape output**: $[bz, C_{out}, H_{out} \times W_{out}] \rightarrow [bz, C_{out}, H_{out}, W_{out}]$

The entire forward pass could be done within a few lines of code (only the relevant parts are kept):

```
self.unfolded = unfold(x, kernel_size = self.kernel_size, dilation=self.dilation, padding=self.padding, stride=self.stride)
weight = self.weight.reshape(self.out_channels, -1)
wxb = einsum('ow,bws->bso', weight, self.unfolded)
if self.use_bias: wxb += self.bias
wxb = einsum('bso->bos', wxb)
out_h = math.floor((x.shape[-2]-(self.kernel_size-1)*self.dilation+2*self.padding-1)/self.stride+1)
out_w = math.floor((x.shape[-1]-(self.kernel_size-1)*self.dilation+2*self.padding-1)/self.stride+1)
ret = wxb.reshape(-1, self.out_channels, out_h, out_w)
```

Listing 2: Forward pass of `Conv2d`.

The backward pass of our `Conv2d` module is in the following order:

1. **Reshape upstream gradients $\partial\mathbf{L}/\partial\mathbf{Y}$**:
   $[bz, C_{out}, H_{out}, W_{out}] \to [bz, C_{out}, H_{out} \times W_{out}]$.
2. **Reshape kernel weights**:
   $[C_{out}, C_{in}, ks, ks] \to [C_{out}, C_{in} \times ks^2]$.
3. **Apply chain rule $\frac{\partial\mathbf{L}}{\partial\mathbf{X}} = \frac{\partial\mathbf{L}}{\partial\mathbf{Y}}\mathbf{W}^T$:**
   $[C_{out}, C_{in} \times ks^2] \times [bz, C_{out}, H_{out} \times W_{out}] \to [bz, C_{in} \times ks^2, H_{out} \times W_{out}]$.
4. **Apply `fold` to recover shape of $\frac{\partial\mathbf{L}}{\partial\mathbf{X}}$ to $\mathbf{X}$**:
   $[bz, C_{in} \times ks^2, H_{out} \times W_{out}] \xrightarrow{\texttt{fold}} [bz, C_{in}, H_{in}, W_{in}]$
5. **Acquire gradients w.r.t weights in `self.unfolded` $\frac{\partial\mathbf{L}}{\partial\mathbf{W}} = \mathbf{X}^{\mathbf{T}}\frac{\partial\mathbf{L}}{\partial\mathbf{Y}}$ and transform to the original weight shape**:
   $[bz, C_{in} \times ks^2, H_{out} \times W_{out}] \times [bz, C_{out}, H_{out} \times W_{out}] = [C_{out}, C_{in} \times ks^2] \to [C_{out}, C_{in}, ks, ks]$
6. **Acquire gradients w.r.t bias: $\frac{\partial\mathbf{L}}{\partial\mathbf{b}} = \frac{\partial\mathbf{L}}{\partial\mathbf{Y}}$**:
   $[bz, C_{out}, H_{out}, W_{out}] \xrightarrow{\texttt{transpose+flatten}} [C_{out}, bz \times H_{out} \times W_{out}] \xrightarrow{\texttt{mean}} [C_{out},]$

In terms of initialization, we use the same method as in the `reset_parameters(self)` function in PyTorch repository. Entire implementation of `Conv2d` module is shown in Section 8.

3 Upsampling Module

In our implementation, the `Upsampling` module is implemented in the way of *nearest neighbor* sampling (NN Sample) + convolution. Two tricky parts in this implementation are: 1) implement nearest neighbor sampling efficiently. 2) implement backward gradient pass of nearest neighbor sampling.

In terms of nearest neighbor sampling, we implement it with `torch.repeat_interleave` as follows (simplified due to paragraph limit):

```python
def nearest_neighbor_sampling(self, inp, scale_factor):
    inter = repeat_interleave(inp, repeats=scale_factor, dim=-1)
    ret = repeat_interleave(inter, repeats=scale_factor, dim=-2)
    return ret
```

Listing 3: Nearest neighbor sampling.

The back propagation of `Upsampling` consists of two parts: 1) backprop the upstream gradients from output to convolution layer input $\partial\mathbf{L}/\partial\mathbf{X}_{up}$, and 2) back prop from convolution input to input $\partial\mathbf{X}_{up}/\partial\mathbf{X}$ (passing the NN Sample module). Where 1) can simply be achieved by backpropagating the convolution layer in `Upsampling`, whereas, the second requires a special type of gradient accumulation way which depends on the `scale_factor` in the NN Sample part. An example of this type of accumulation is shown in Eq. (1) (with `scale_factor=2`):

$$\underbrace{\left[\begin{array}{cc|cc} 1 & 0 & 0 & 2 \\ -3 & 4 & 2 & 1 \\ \hline -2 & 0 & 0 & 1 \\ 0 & 3 & 0 & 1 \end{array}\right]}_{\partial\mathbf{L}/\partial\mathbf{X}_{up}} \to \underbrace{\begin{bmatrix} 2 & 5 \\ 1 & 2 \end{bmatrix}}_{\partial\mathbf{L}/\partial\mathbf{X}} \tag{1}$$

This is implemented with two for loops, we use tensor operations to accelerate on GPU:

```python
def backward(self, grdwrtoutput):
    dl_dw = self.conv.backward(grdwrtoutput)
    N, C, H, W = dl_dw.size()
    out_h, out_w = H // self.scale_factor, W // self.scale_factor
    rows = arange(0, H, self.scale_factor).repeat(out_h)
    cols = arange(0, W, self.scale_factor).repeat_interleave(out_w)
    dl_dx = zeros(dl_dw[..., cols+0, rows+0].size()).to(grdwrtoutput.device)
    for i in range(self.scale_factor):
        for j in range(self.scale_factor):
            dl_dx = dl_dx + dl_dw[..., cols+i, rows+j]
    dl_dx = dl_dx.reshape(N, C, out_h, out_w)
    return dl_dx
```

Listing 4: Backprop of `Upsampling` module.

4 ReLU Module

`torch.clamp` and `torch.sign` are mainly used to implement `ReLU` module. Module implement main steps: clamp input $\rightarrow$ output 0 or input itself $\rightarrow$ sign input and then clamp $\rightarrow$ 1 or 0, notice this module has no parameters.

5 Sigmoid Module

`torch.exp` is used to implement `sigmoid` module with following equations. $\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$, $\text{Sigmoid'}(x) = \frac{(e^{-x}+1)-1}{1+e^{-x}} \frac{1}{1+e^{-x}} = (1 - \text{Sigmoid}(x))\,\text{Sigmoid}(x)$, this module also has no parameters.

6 Mean Squared Error Loss Function Module

`MSELoss` module is implemented with `torch.pow`, `torch.mean`, and `torch.size` with the following equations. $\text{MSELoss}(y,t) = \frac{1}{N}\sum_{i=0}^{N}(y_i - t_i)^2$, $\text{MSELoss'}(y,t) = \frac{2}{N}\sum_{i=0}^{N}(y_i - t_i)$, where N is total elements in y and t. The module does following things: initialize the prediction *y* and target *t* $\Rightarrow$ compute the loss and compute the gradient w.r.t prediction *y*.

7 Stochastic Gradient Descent optimizer Module

`SGD` module is mainly implemented with `torch.add` and `torch.Tensor.zero_`. `SGD` module has `zero_grad()` to set all grad parameters in the input parameters to 0, and `step()` to update grad parameters in the input parameters by `w += -grad * lr`.

These are done with the following code:

```python
def zero_grad(self):
    for modules in self.params:
        weight, grad = modules
        if grad != None: grad.zero_()
```

Listing 5: `SGD zero_grad()` part.

```python
def step(self):
    for modules in self.params:
        weight, grad = modules
        if weight != None and (grad != None): weight.add_(-self.lr * grad)
```

Listing 6: `SGD step()` part.

8 Sequential Module

`Sequential` module needs to make sure that both forward inputs and backward gradients as well as backward gradients could flow smoothly during training. Morever, the `Sequential` module also needs to take care of gathering parameters from all sub modules and loading back the parameters to its sub modules. All these are done by:

```python
class Sequential(Module):
    def __init__(self, *layers) -> None:
        super().__init__()
        self.modules = []
        for layer in layers:
            self.modules.append(layer)

    def forward(self, x):
        ret = x
        for layer in self.modules:
            ret = layer.forward(ret)
        return ret

    def backward(self, gradwrtoutput):
        grad_from_back = gradwrtoutput
        for layer in reversed(self.modules):
            grad_from_back = layer.backward(
  grad_from_back)
    def __call__(self, input):
        return self.forward(input)
```

Listing 7: `Sequential` functioning part.

```python
def to(self, device):
    for i, module in enumerate(self.modules):
        self.modules[i] = module.to(device)
    return self
def param(self):
    ret = []
    for layer in self.modules:
        ret.append(layer.param()[0])
        if len(layer.param()) > 1:
            ret.append(layer.param()[1])
    return ret

def load_param(self, param):
    model_idx = param_idx = 0
    while model_idx < len(self.modules) and (param_idx < len(param)):
        required_length = len(self.modules[model_idx].param())
        self.modules[model_idx].load_param(param[param_idx:
  required_length+param_idx])
        param_idx += required_length
        model_idx += 1
```

Listing 8: Auxiliary: switch device save/load parameters.

Appendix 1: Experiment

In terms of experiment design, we implement several *Noise2Noise*-like model structures to test our modules' performance. Due to space limitation, we report validation performance on the given dataset of two typical models and their training curves as well as final visualizations.

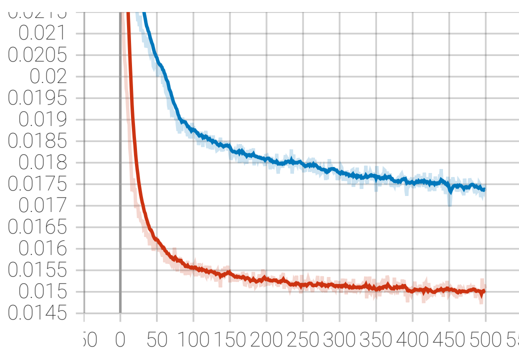
```
model = Sequential(
    Conv2d(3, 32, 3, stride=1),
    ReLU(),
    Conv2d(32, 64, 3, stride=1, padding=3),
    ReLU(),
    Upsampling(2, 64, 32, stride=2),
    ReLU(),
    Upsampling(2, 32, 3, stride=2),
    Sigmoid()
)
```

Listing 9: Best performance model 1.

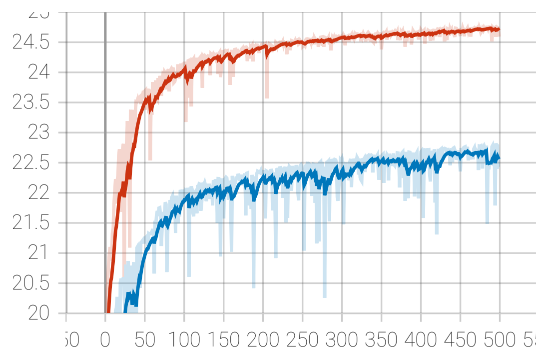
```
model = Sequential(
    Conv2d(3, 10, 3, stride=2, padding=2),
    ReLU(),
    Conv2d(10, 10, 3, stride=2, padding=2),
    ReLU(),
    Upsampling(2, 10, 10, kernel_size=4, stride=1),
    ReLU(),
    Upsampling(2, 10, 3, kernel_size=3, stride=1),
    Sigmoid()
)
```

Listing 10: Model 2: Conv2d stride 2.

We trained both models for 500 epochs with batch size 16 and SGD of learning rate 6e-2. The best performances they achieved were **24.85dB** for model 1 and **22.73dB** for model 2 (we submitted model 2 as per requirements). The training/validation curves are shown in Fig. 1.



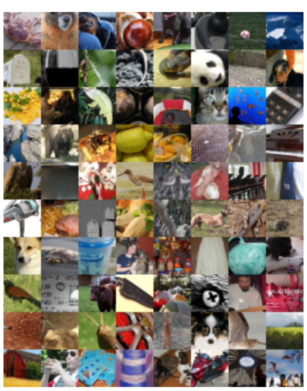
(a) Training loss (MSE).



(b) Validation PSNR.

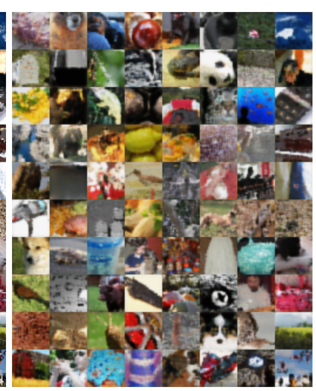
Figure 1: Training/Validation curves of Model 1 and Model 2.

The result visualizations are shown in Fig. 2.

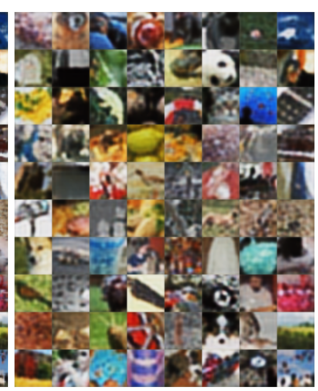


(a) Ground truth image

(b) Input noisy image



(c) Model 1 output



(d) Model 2 output

Figure 2: Visualization results at validation.

Appendix 2: Conv2d Implementation

```python
class Conv2d(Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, bias=True, dilation=1, stride=1, padding=0):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.dilation = dilation
        self.stride = stride
        self.padding = padding
        self.use_bias = bias

        # torch initialization
        n = in_channels * kernel_size**2
        stdv = 1. / math.sqrt(n)

        self.weight = empty(out_channels, in_channels, kernel_size, kernel_size).uniform_(-stdv, stdv)
        self.bias  = empty(out_channels).uniform_(-stdv, stdv)
        self.dilation = dilation
        self.padding = padding
        self.stride = stride
        self.unfolded = None
        self.x = None

        # gradient
        self.dl_dw = empty(self.weight.size())
        self.dl_db = empty(self.bias.size())

    def forward(self, x):
        self.x = x
        self.unfolded = unfold(x, kernel_size = self.kernel_size, dilation=self.dilation, padding=self.padding, stride=self.
    stride)
        weight = self.weight.reshape(self.out_channels, -1)
        wxb = einsum('ow,bws->bso', weight, self.unfolded)
        if self.use_bias:
            wxb += self.bias
        wxb = einsum('bso->bos', wxb)
        out_h = math.floor((x.shape[-2] - (self.kernel_size-1)*self.dilation + 2*self.padding - 1) / self.stride + 1)
        out_w = math.floor((x.shape[-1] - (self.kernel_size-1)*self.dilation + 2*self.padding - 1)  / self.stride + 1)

        #ret = fold(wxb, output_size=(out_h, out_w), kernel_size=(1,1))
        ret = wxb.reshape(-1, self.out_channels, out_h, out_w)
        return ret

    def backward(self, grdwrtoutput):
        grdwrtoutput = grdwrtoutput.flatten(2, -1)
        weight = self.weight.reshape(self.out_channels, -1)
        # here x is cin*kernelsize^2 and s is Hout*Wout
        dl_dx = einsum('ox,nos->nxs', weight, grdwrtoutput)
        out_size = (self.x.size(-2), self.x.size(-1))
        dl_dx = fold(dl_dx, output_size=out_size, kernel_size=self.kernel_size, dilation=self.dilation, padding=self.padding,
    stride=self.stride)

        self.dl_dw.add_(einsum('nos,nxs->ox', grdwrtoutput, self.unfolded).reshape(self.weight.size()))
        self.dl_db.add_(grdwrtoutput.transpose(0, 1).flatten(1, -1).mean(1))
        return dl_dx

    def param(self):
        return [(self.weight, self.dl_dw), (self.bias, self.dl_db)]

    def __call__(self, input):
        return self.forward(input)

    def to(self, device):
        self.dl_dw = self.dl_dw.to(device)
        self.dl_db = self.dl_db.to(device)
        self.weight = self.weight.to(device)
        self.bias = self.bias.to(device)
        return self

    def load_param(self, param, device):
        self.weight, _ = param[0]
        self.bias, _ = param[1]
        self = self.to(device)
```

Listing 11: Implemetation of `Conv2d`.