Deep Learning Implementation Lab 5 : Design Custom Operator

20230683 박한비, 20230308 최승희

해당 랩에서는 PyTorch 상에서 사용자 정의 operator 의 형태로 Batch Normalization 을 구현하였다. Batch Normalization 은 input activation 을 정규화하며 평균과 분산를 조정한다. 이는 output data 가 평균 0, 분산 1 의 정규 분포를 따르도록 만든다. 결국 Batch Normalization 은 Train set 과 Test set 의 분포 차에 따른 overfitting 을 줄여준다.

1. Implementation

a. Forward Pass

Forward 에서 Batch Normalization 의 수식은 아래와 같다.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

torch.library.custom_op 를 활용하여 미리 구현된 skeleton code 내부를 배치 정규화의 forward propagation 기능을 하도록 코드를 작성해 채웠다.

```python
# Step 1: Define custom operators using torch.library API
@torch.library.custom_op("my_ops::batchnorm_forward", mutates_args=("running_mean", "running_var"))
def batchnorm_forward(
    input: Tensor,          # [N, C, H, W]
    gamma: Tensor,          # [C]
    beta: Tensor,           # [C]
    running_mean: Tensor,   # [C]
    running_var: Tensor,    # [C]
    training: bool,
    momentum: float,
    eps: float
) -> Tuple[Tensor, Tensor, Tensor]:
    """forward pass of BatchNorm for 4D input [N, C, H, W]."""

    # Implement Here
    N, C, H, W = input.shape
    dims = (0, 2, 3)  # reduce over N,H,W
    g = gamma.view(1, -1, 1, 1)
    b = beta.view(1, -1, 1, 1)

    if training:
        mean = input.mean(dims)                                 # [C]
        var = input.var(dims, unbiased=False)                   # [C]
        invstd = torch.rsqrt(var + eps)                         # [C]

        x_hat = (input - mean.view(1, -1, 1, 1)) * invstd.view(1, -1, 1, 1)
        output = x_hat * g + b

        # In-place running stats update
        running_mean.mul_(1.0 - momentum).add_(momentum * mean)
        running_var.mul_(1.0 - momentum).add_(momentum * var)

        save_mean = mean
        save_invstd = invstd
    else:
        invstd = torch.rsqrt(running_var + eps)                 # [C]
        x_hat = (input - running_mean.view(1, -1, 1, 1)) * invstd.view(1, -1, 1, 1)
        output = x_hat * g + b

        # Save the exact stats used for normalization for backward
        save_mean = running_mean.detach().clone()
        save_invstd = invstd  # freshly computed; no need to clone

    return output, save_mean, save_invstd
```

각 채널에 대하여 연산을 진행할 것이기 때문에 채널 C 에 대한 dimension 으로 설정해준다. 그 다음 training 단계와 test 단계에서 일어날 동작이 다르기 때문에 분기를 나누어준다. training 에서는 위 수식을 따라 input 의 채널 별 평균(mean)과 분산(var)을 구해주고 정규화된 input(x_hat)과 이에 scale, shift 도 적용한 값(output)까지 구해준다. 또한 mean 과 var 를 이용하여 이를 running_mean, running_var 에 적용해준다. test 에서는 running_mean, running_var 를 활용하여 x_hat 과 output 을 구해준다. 두 단계 모두 backward 를 위해 mean, invstd 를 저장해준다(save_mean, save_invstd).

b. Backward Pass

Backward 에서 Batch Normalization 의 수식은 아래와 같다.

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

torch.library.custom_op 를 활용하여 미리 구현된 skeleton code 내부를 배치 정규화의 backward propagation 기능을 하도록 코드를 작성해 채웠다.

```
@torch.library.custom_op("my_ops::batchnorm_backward", mutates_args=())
def batchnorm_backward(
    grad_output: Tensor,     # [N, C, H, W]
    input: Tensor,           # [N, C, H, W]
    gamma: Tensor,           # [C]
    save_mean: Tensor,       # [C]
    save_invstd: Tensor      # [C]
) -> Tuple[Tensor, Tensor, Tensor]:
    """backward pass of BatchNorm for 4D input."""

    # Implement Here
    N, C, H, W = input.shape
    m = float(N * H * W)
    dims = (0, 2, 3)

    g = gamma.view(1, -1, 1, 1)
    mean = save_mean.view(1, -1, 1, 1)
    invstd = save_invstd.view(1, -1, 1, 1)

    x_hat = (input - mean) * invstd

    # Gradients for scale (gamma) and shift (beta)
    grad_beta = grad_output.sum(dims)                # [C]
    grad_gamma = (grad_output * x_hat).sum(dims)     # [C]

    # Determine if stats came from current batch (training) or running stats (eval)
    # If they match (within tolerance), treat as training; otherwise eval-path.
    sample_mean = input.mean(dims)
    is_training_like = torch.allclose(sample_mean, save_mean, rtol=1e-5, atol=1e-5)

    dy = grad_output
    dy_g = dy * g                                    # [N,C,H,W]

    if is_training_like:
        # Training-mode backward:
        # dX = (1/m) * invstd * ( m*dy_g - sum(dy_g) - x_hat*sum(dy_g*x_hat) )
        sum_dy = dy_g.sum(dims).view(1, -1, 1, 1)                # [1,C,1,1]
        sum_dy_xhat = (dy_g * x_hat).sum(dims).view(1, -1, 1, 1)  # [1,C,1,1]
        grad_input = (invstd / m) * (m * dy_g - sum_dy - x_hat * sum_dy_xhat)
    else:
        # Eval-mode backward (normalization constants are treated as constants)
        grad_input = dy_g * invstd

    return grad_input, grad_gamma, grad_beta
```

각 채널에 대하여 연산을 진행할 것이기 때문에 채널 C 에 대한 dimension 으로 설정해준다. 주어진 수식과 forward 에서 저장한 값을 이용하여 dl/dβ(grad_beta),

dl/dγ(grad_gamma)를 구해준다. 그 다음에는 해당 backward pass 가 training 단계에 속하는지 test 단계에 속하는지를 판별한다. input 을 이용해 구한 평균(sample_mean)이 save_mean 과 거의 같다면 forward 에서도 input 의 평균을 그대로 저장했다는 것이므로 training 단계라고 추측할 수 있다(is_training_like = True). 이제 training 단계와 test 단계에서 dl/dx 를 다르게 구하므로 분기를 나누어준다. training 에서 output 을 구할 때 이용하는 평균, 분산은 input 을 이용해 구한 것이므로 input 에 의존하나 test 에서는 running_mean, running_var 를 평균, 분산으로 이용하므로 input 에 의존하지 않는 상수이다. 따라서 미분식이 달라지기 때문에 이를 유의하여 각각 dl/dx(grad_input)을 구해준다.

위의 코드로 구현한 operator 를 Pytorch 의 autograd 로 연결하기 위한 단계는 미리 구현되어있던 코드를 이용하였다. 이러한 과정을 통해, 사용자가 직접 정의한 custom operator 를 Pytorch 에서 BatchNormCustom 클래스로 적용할 수 있다.

2. Result

Implementation 에서 구현한 custom operator 를 테스트하기 위해 이미 주어진 test_batchnorm.py 를 활용하였다. 테스트는 PyTorch 에서 Batch Normalization 을 제공하는 기존 클래스 torch.nn.BatchNorm2d 와의 비교를 통해 진행된다. BatchNormCustom 클래스를 통한 정규화와 torch.nn.BatchNorm2d 를 통한 정규화의 각 output 들의 차이가 설정된 임계값을 넘지 않는다면 테스트는 통과한다.

'python test/test_batchnorm.py' 명령어로 테스트를 실행하였고 그 결과는 아래와 같다.

```
========================================
All tests completed!
root@460746c9756c:~/python-custom-ops-bn-student# python test/test_batchnorm.py
Testing Custom BatchNorm Implementation
========================================
Device: cuda
Input shape: [32, 64, 56, 56]

1. Forward Pass
   Max difference: 9.54e-07
   ✓ Forward pass passed

✓ Test PASSED

========================================
Testing Inference Mode
Inference input shape: [8, 128, 28, 28]
✓ Inference test PASSED

========================================
All tests completed!
```

모든 테스트를 통과한 것을 보아 Custom Operator 가 성공적으로 구현되었다는 것을 알 수 있다.