Week 5: Custom Operators

20230071 정다은
20230309 최건

1. batchnorm_forward()

```
# Implement Here
reduce_dims = (0, 2, 3)
def _chan_view(t: Tensor) -> Tensor:
    return t.view(1, -1, 1, 1)
```

이 함수는 입력 x 에 대해 채널 C 별로 배치하고 공간 축(N, H, W)에 대해 통계를 내는 함수이다. 먼저, 채널을 남기고 N, H, W 로 평균/합산하기 위해 reduce_dims = (0, 2, 3)을 통해 축을 지정한다. _chan_view(t) 함수는 mean/invstd 와 같은 채널 벡터를 NCHW 와 연산 가능하게 하기 위해 지정한 함수이다. 이를 통해 채널 벡터를 [1 C 1 1]로 바꿔 NCHW tensor 와 브로드캐스팅이 가능해진다.

```
if training:
    batch_mean = input.mean(dim=reduce_dims)                    # [C]
    batch_var  = input.var(dim=reduce_dims, unbiased=False)     # [C]
    invstd     = torch.rsqrt(batch_var + eps)                   # [C]

    # running = (1 - m) * running + m * batch_stat
    running_mean.mul_(1.0 - momentum).add_(momentum * batch_mean)
    running_var.mul_(1.0 - momentum).add_(momentum * batch_var)

    mean_for_norm   = batch_mean
    invstd_for_norm = invstd

    save_mean   = batch_mean.detach()
    save_invstd = invstd.detach()
```

학습 모드일 때는, 채널별 배치 평균과 분산을 계산한다. 분산 계산 시, unbiased=False(모집단 분산)으로 해야 PyTorch nn.BatchNorm2d 과 일치한다. running = (1 - m) * running + m * batch_stat 임과 running_mean 과 running_var 은 buffer 임을 고려해 이들을 수식에 따라 in-place 업데이트 한다. 정규화에 사용할 mean 과 invstd 를 방금 계산한 배치 통계로 업데이트하고, save_mean 과 save_invstd 는 backward 에만 쓰는 캐시이므로 연산 그래프에서 detach 하여 저장한다.

```
else:
    mean_for_norm   = running_mean.detach().clone()
    invstd_for_norm = torch.rsqrt(running_var + eps)

    save_mean   = running_mean.detach().clone()
    save_invstd = invstd_for_norm.detach().clone()

x_hat  = (input - _chan_view(mean_for_norm)) * _chan_view(invstd_for_norm)  # [N,C,H,W]
output = _chan_view(gamma) * x_hat + _chan_view(beta)                       # [N,C,H,W]

return output, save_mean, save_invstd
```

추론 모드일때는 정규화에 running 통계를 사용한다. 이때, torch.library 커스텀 연산자는 출력과 입력이 메모리 공유되면 안 되므로 .detach().clone()을 통해 복제해서 반환한다. 마지막으로 $\hat{x} = (x - \mu) \cdot invstd$, $y = \gamma \cdot \hat{x} + \beta$을 구현하여 output 을 계산한다.

마지막으로 save_mean 과 save_invstd 는 backward 공식에 사용되므로 output 과 함께 반환한다.

2. batchnorm_backward()

```
# Implement Here
C = input.shape[1]
N_H_W = torch.tensor(input.shape[0] * input.shape[2] * input.shape[3],
                     dtype=input.dtype, device=input.device)

x_hat = (input - save_mean.view(1, -1, 1, 1)) * save_invstd.view(1, -1, 1, 1)
```

먼저 backpropagation 을 위한 기본 변수들을 설정한다. N_H_W 는 batch size(N)와 spatial dimensions(H,W)의 총합을 계산한다. 이 값은 batch normalization(BN)에서 통계량을 계산할 때 사용되는 normalization factor 다. 정확히는, gradient 를 구하는 과정에서 평균이나 분산을 통한 미분 항을 처리하기 위해 사용된다.

x_hat 은 normalized input tensor 를 계산하는 γ, x 의 gradient 를 계산하는 데에 있어 핵심이 되는 변수다. x 에서 save_mean 을 빼고 save_invstd 를 곱하여 얻는다. save_mean 과 save_invstd 는 shape [C]의 one-dimensional tensor 이므로, 4-dimensional tensor [N, C, H, W]와의 연산을 위해 .view(1, -1, 1, 1)을 사용하여 broadcasting 이 가능하게 했다.

```
    # Gradient w.r.t. beta (shift)
    grad_beta = grad_output.sum(dim=(0, 2, 3))

    # Gradient w.r.t. gamma (scale)
    grad_gamma = (grad_output * x_hat).sum(dim=(0, 2, 3))
```

이 부분은 BN 의 learnable parameter γ 와 β 에 대한 gradient 를 계산한다. 먼저, β 는 단순히 BN 의 출력에 단순히 더해지는 bias 역할을 한다. 미분 규칙에 따라 β 에 대한 gradient 는 upstream gradient 인 grad_output 을 β 가 적용된 모든 element 들에 대해 합산하여 구한다. 따라서 batch, height, width 로 정의되는 차원 (0, 2, 3)에 대해 합산하여 [C] 형태의 grad_beta 를 얻는다. γ 는 정규화된 입력 $\hat{x}$와 곱해지는 scale 역할을 한다. chain rule 에 따라 γ 에 대한 기울기는 grad_output 과 x 에 element-wise multiplication 을 수행한 후, β 에서와 같이 (0, 2, 3) 차원에서 합산하여 [C] 형태의 grad_gamma 를 계산하도록 설계했다.

```
# Gradient w.r.t. input (x)
dL_dx_hat = grad_output * gamma.view(1, -1, 1, 1)

mean_dL_dx_hat = dL_dx_hat.sum(dim=(0, 2, 3)).view(1, -1, 1, 1)
mean_dL_dx_hat_x_hat = (dL_dx_hat * x_hat).sum(dim=(0, 2, 3)).view(1, -1, 1, 1)

grad_input = (save_invstd.view(1, -1, 1, 1) / N_H_W) * (
    N_H_W * dL_dx_hat
    - mean_dL_dx_hat
    - x_hat * mean_dL_dx_hat_x_hat
)

return grad_input, grad_gamma, grad_beta
```

이제 x 에 대한 최종 기울기 grad_input 를 계산한다. 먼저 grad_input 계산의 기반이 되는 정규화된 입력 $\hat{x}$에 대한 기울기 dL_dx_hat 를 계산한다. 이 연산은 Batch Normalization $y = \gamma \cdot \hat{x} + \beta$에서 $\hat{x}$의 다음 단계가 γ의 곱셈인 것에 착안하여 Chain Rule 을 적용한 것이다. 이때, γ는 4-dimension tensor 와의 broadcasting 을 위해 명시적으로 dimension 을 확장한다.

이후 x 의 변화가 평균 μ와 분산 σ^2 을 통해 간접적으로 손실에 미치는 영향을 반영하기 위해 dL_dx_hat 을 batch 와 공간 차원(0, 2, 3)에 대해 합산한 값인 mean_dL_dx_hat 은 x 의 변화가 평균 μ를 통해 손실에 미치는 영향을 보정하는 데 사용된다. 또한, dL_dx_hat 과 정규화된 입력 $\hat{x}$를 곱한 후 합산한 값인 mean_dL_dx_hat_x_hat 은 x 의 변화가 분산 σ^2 을 통해 손실에 미치는 영향을 상쇄하는 데 필요하다. 이 두 값은 grad_input 의 최종 계산에 사용되는 channel 별 통계값 역할을 수행하며, 계산 후 다시 broadcasting 형태로 변경된다.

최종 grad_input 은 앞서 계산된 모든 항을 결합하여 완성된다. 이 수식은 BN backpropagation 의 표준 구현 공식을 따른다. N_H_W · dL_dx_hat 은 $\hat{x}$를 통한 직접적인 영향을 반영하며, 나머지 두 항은 평균과 분산을 통한 간접적 미분 효과를 상쇄하는 보정 항 역할을 한다. 최종적으로 전체 괄호 항은 save_invstd 를 곱하고 정규화 계수 N_H_W 로 나누어 정규화한다. 이 과정을 통해 [N, C, H, W] 형태의 정확한 grad_input 이 계산되며, 이는 이전 계층으로 전달되어 backpropagation 을 수행할 수 있게 한다.

3. conclusion

```
root@c4a583a0f4d8:/workspace/python-custom-ops-bn-student# python test/test_batchnorm.py command
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!
```

위의 테스트를 통해 우리 조가 작성한 코드와 nn.BatchNorm2d 의 최대 차이가 9.54e-07 임을 확인하였다. 이는 test script 의 허용치 1e-5 보다 더 작은 수치이다. 테스트를 통해 우리 조의 구현이 nn.BatchNorm2d 와 사실상 동일함을 확인할 수 있었다.