Lab2: AutoGrad

20230071 정다은
20230309 최건

1-A. Mathematical Operations

Add 연산자는 Chain Rule에 따라 Loss 값이 그대로 유지된다.

$$\text{In } a+b=z,\ \frac{\partial L}{\partial a}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial a}=\frac{\partial L}{\partial z},\ \frac{\partial L}{\partial b}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial b}=\frac{\partial L}{\partial z}\ (\because \frac{\partial z}{\partial a}=1, \frac{\partial z}{\partial b}=1)$$

```
a_shape, b_shape = ctx.saved_tensors
grad_a = reduce_grad_to_shape(grad_output, a_shape)
grad_b = reduce_grad_to_shape(grad_output, b_shape)
return grad_a, grad_b
```

Mul 연산자는 Chain Rule에 따라 Loss 값에 상대 변수 값을 곱한 값이 전달된다.

$$\text{In } ab=z,\ \frac{\partial L}{\partial a}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial a}=b\cdot\frac{\partial L}{\partial z},\ \frac{\partial L}{\partial b}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial b}=a\cdot\frac{\partial L}{\partial z}\ (\because \frac{\partial z}{\partial a}=b, \frac{\partial z}{\partial b}=a)$$

```
a, b, a_shape, b_shape = ctx.saved_tensors
grad_a_tmp = grad_output * b
grad_b_tmp = grad_output * a
grad_a = reduce_grad_to_shape(grad_a_tmp, a_shape)
grad_b = reduce_grad_to_shape(grad_b_tmp, b_shape)
return grad_a, grad_b
```

Power 연산자 역시 Chain Rule 적용을 위해 각 변수에 편미분을 수행한다.

$$\text{In } a^b=z,\ \frac{\partial L}{\partial a}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial a}=a^{b-1}b\cdot\frac{\partial L}{\partial z},\ \frac{\partial L}{\partial b}=\frac{\partial L}{\partial z}\frac{\partial z}{\partial b}=a^b ln\ a\cdot\frac{\partial L}{\partial z}$$

```
a, b, a_shape, b_shape = ctx.saved_tensors
grad_a_tmp = grad_output * b * (a ** (b-1))
grad_b_tmp = grad_output * (a**b) * np.log(a)
grad_a = reduce_grad_to_shape(grad_a_tmp, a_shape)
grad_b = reduce_grad_to_shape(grad_b_tmp, b_shape)
return grad_a, grad_b
```

Sum 연산자는 피연산 행렬의 크기가 결과와 다를 수 있기 때문에, parameter들을 고려해야 한다. 예를 들어, axis 옵션이 켜져 있다면, entry마다 연산에 참여한 값이 다르므로 이를 고려하여 Loss 값을 entry들에게 각각 신중하게 broadcast해야 한다.

```
a_shape, axis, keepdims = ctx.saved_tensors
if axis is None:
    grad_a = np.ones(a_shape, dtype=grad_output.dtype) * grad_output
else:
    if not keepdims:
        grad_output = np.expand_dims(grad_output, axis=axis)
    grad_a = np.ones(a_shape, dtype=grad_output.dtype) * grad_output
return grad_a,
```

1-B. Activation Functions

ReLU 연산은 $\max(a, 0)$과 같으므로, max 연산에 대한 Back Propagation을 진행한다.

$$\text{In } ReLU(0,A)=Z,\ \frac{\partial L}{\partial a_{ij}}=\begin{cases}0 & \text{if } a_{ij}\le 0\\ \frac{\partial L}{\partial z_{ij}} & \text{if } a_{ij}>0\ (\because \text{In } a_{ij}=z_{ij}, \frac{\partial z_{ij}}{\partial a_{ij}}=1)\end{cases}$$

```
def forward(ctx, a):
    mask = a > 0
    ctx.save_for_backward(mask)
    return np.where(mask, a, 0.0)
```

```
def backward(ctx, grad_output):
    (mask,) = ctx.saved_tensors
    return grad_output * mask
```

Softmax 연산은 각 element들을 e^x의 확률분포로 나타내는 연산으로, 다음과 같다.

$$\text{In Softmax},\ z_i=\frac{e^{a_i}}{\sum_j e^{a_j}},\ \frac{\partial L}{\partial a_i}=\sum_k \frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial a_i}=\left(\frac{\partial L}{\partial z_i}-\sum_k \frac{\partial L}{\partial z_k}z_k\right)z_i$$

```
(probs,) = ctx.saved_tensors
grad_a = (grad_output - np.sum(grad_output * probs, axis=-1, keepdims=True)) * probs
return grad_a,
```

Log 연산도 역시 Chain Rule을 적용시켰고, 다음과 같다.

$$ln\ a = z,\ \frac{\partial L}{\partial z}\frac{\partial z}{\partial a} = \frac{1}{a}\cdot\frac{\partial L}{\partial z}$$

```python
(x,) = ctx.saved_tensors
grad_a = grad_output / x
return grad_a,
```

1-C. Loss Functions

NLLLoss와 CrossEntropyLoss에 대해 다음과 같은 식을 얻어, 아래와 같이 구현했다.

$$L = -\frac{1}{B}\sum_{n=1}^{B}\sum_{c=1}^{C} y_{n,c}\hat{y}_{n,c},\quad \frac{\partial L}{\partial \hat{y}_i} = -\frac{1}{B}y_i$$

```python
class NLLLoss(Op):
    @staticmethod
    def forward(ctx, log_probs, targets_one_hot):
        loss = -np.sum(targets_one_hot * log_probs, axis=-1)
        batch = log_probs.shape[0]
        ctx.save_for_backward(targets_one_hot, batch)
        return np.mean(loss)
    @staticmethod
    def backward(ctx, grad_output):
        targets_one_hot, batch = ctx.saved_tensors
        grad_log_probs = -(targets_one_hot/batch) * grad_output
        return grad_log_probs, None
```

NLLLoss의 식과 구현

$$L = -\frac{1}{B}\sum_{n=1}^{B}\sum_{c=1}^{C} y_{n,c}\log\frac{e^{a_{n,c}}}{\sum_{j=1}^{C} e^{a_{n,j}}},\quad \frac{\partial L}{\partial a_i} = \frac{1}{B}\left(p_i - y_i\right)$$

```python
class CrossEntropyLoss(Op):
    @staticmethod
    def forward(ctx, logits, targets):
        ctx.save_for_backward(logits, targets)
        max_logits = np.max(logits, axis = 1, keepdims = True)
        log_softmax = logits - np.log(np.sum(np.exp(logits - max_logits), axis = 1, keepdims = True)) - max_logits
        loss = -np.sum(targets * log_softmax, axis=1)
        return np.mean(loss)
    @staticmethod
    def backward(ctx, grad_output):
        logits, targets = ctx.saved_tensors
        exps = np.exp(logits - np.max(logits, axis=1, keepdims = True))
        softmax = exps / np.sum(exps, axis=1, keepdims = True)
        grad_logits = (softmax - targets) / logits.shape[0]
        grad_logits *= grad_output
        return grad_logits, None
```

CrossEntropyLoss의 식과 구현

2. L2 정규화

L2 정규화는 weight 값이 과도하게 커져 일부 특징에 의존하는 현상을 방지하고 데이터의 일반적인 특징을 잘 반영하게 하는 방법으로, overfitting 문제를 해결하기 위한 방법 중 하나이다. L2 regulation 은 모델의 손실함수에 L2 Loss Function 을 추가해준 것이다. L2 Loss Function 수식은 $L = \sum_{\{i=1\}}^{\{n\}}\left(Y_i - f(x_i)\right)^2$이다. 랩에서 우리 조는 $Loss = Loss_{data} + \left(\frac{\lambda}{2N}\right)\sum_{\{\ell\}}\left|\left|W_{\{\ell\}}\right|\right|_2^2$로 구현하였다. 이때, $Loss_{data}$는 NLLoss 를 사용하였고, N 은 배치 크기를, λ는 정규화 강도(weight_decay)를 의미한다.

```python
for i in range(0, X_train.shape[0], batch_size):
    # Mini-batch slice
    X_batch = Tensor(X_train[i:i+batch_size])        # input batch
    y_batch = Tensor(y_train[i:i+batch_size])        # target batch

    # Forward pass:
    logits = model(X_batch)
    probs  = Softmax.apply(logits)
    logp   = Log.apply(probs)

    data_loss    = NLLLoss.apply(logp, y_batch)
    #loss = CrossEntropyLoss.apply(logits, y_batch)  # Compute cross-entropy loss over the batch
    l2_W1 = Sum.apply(Mul.apply(model.W1, model.W1))
    l2_W2 = Sum.apply(Mul.apply(model.W2, model.W2))
    l2_W3 = Sum.apply(Mul.apply(model.W3, model.W3))
    l2_sum = Add.apply(Add.apply(l2_W1, l2_W2), l2_W3)

    reg_coef = Tensor(weight_decay / (2.0 * batch_size))
    reg = Mul.apply(l2_sum, reg_coef)

    loss = Add.apply(data_loss, reg)

    total_loss += float(loss.data)
    num_updates += 1
```

먼저, Mul.apply(model.W_i, model.W_i)로 각 가중치의 제곱을 구하고, 이어서 그 값을 Sum.apply함으로써 모든 원소를 합해 스칼라 값을 얻는다. 이 값은 $||w_i||^2$에 해당한다. 이때, 스칼라로 만드는 이유는 손실이 스칼라여야 loss.backward()가 정상적으로 동작하고, 배치 평균으로 계산된 $Loss_{data}$와 형상이 일치해 합산할 수 있기 때문이다.