Deep Learning Implementation **Lab 2 : AutoGrad Implementation**

20230683 박한비, 20230308 최승희

각 클래스에서 공통으로 사용되는 주요 함수에 대한 설명이다 : ctx.save_for_backward()은 forward 단계에서 backward 단계에 필요한 것들을 저장한다. ctx.saved_tensors는 backward 단계에서 앞서 저장한 것들 튜플로 꺼내준다.reduce_grad_to_sate는 broadcasting이 개입되는 경우, 원래 입력의 shape로 gradient를 맞춰준다.

1. Implement Mathematical Operations

1) class Mul

Mul 클래스는 입력 a, b의 원소별 곱 연산을 수행한다. forward propagation 연산은 $y = a \odot b$ 로 정의된다. backward propagation에서는 chain rule을 적용하면, output y가 a와 b의 곱이므로, $\partial a/\partial y=b$, $\partial b/\partial y=a$가 된다. 따라서, $\partial L/\partial y$가 주어졌을 때 입력에 대한 기울기는 $\partial L/\partial a=\partial L/\partial y \odot b$, $\partial L/\partial b=\partial L/\partial y \odot a$이다.

<u>Code Interpretation</u>: forward에서는 a * b를 반환한다. backward에서 기울기를 계산할 때 입력값 a, b가 사용되고, broadcasting을 되돌리기 위해 shape 정보를 ctx에 저장한다. grad_output은 $\partial L/\partial y$이다. 위의 수식에 따라 계산한 $\partial L/\partial a$, $\partial L/\partial b$ 값을 grad_a_raw, grad_b_raw에 저장한다. 각각 본래 shape으로 되돌려주고, a,b에 대한 기울기를 반환한다.

2) class Matmul

Matmul 클래스는 두 입력 행렬 A,B에 대해 행렬 곱셈 연산을 수행한다. forward에서의 연산은 $Y=A\times B$ 로 정의된다. backward에서는 chain rule을 적용하여 각 입력에 대한 기울기를 구한다. 행렬 곱의 도함수 성질을 이용하여 입력의 기울기를 전치 행렬과 곱해서 구하게 된다. $\partial L/\partial Y$가 주어졌을 때, A에 대한 기울기는 $\partial L/\partial A=\partial L/\partial Y\times B^T$, B에 대한 기울기는 $\partial L/\partial B=A^T\times\partial L/\partial Y$이다.

<u>Code Interpretation</u>: forward에서는 np.matmul(a,b)를 수행하고, 그 결과를 반환한다. backward 단계에서 기울기 계산에 활용할 입력 행렬 a,b를 미리 저장해둔다. chain rule과 위의 수식에 따라 grad_a_raw, grad_b_raw 값을 계산한다. 마지막으로 원래 입력의 shape으로 맞춘 후 반환한다.

3) class Pow

Pow 클래스는 원소별 a의 b 거듭제곱 연산을 수행한다. Pow의 forward propagation 연산 수식은 $y = a^{\odot b}$으로, $(y)_{i,j} = (a)_{i,j}^{(b)_{i,j}}$이다. backward propagation에서 Pow의 a, b에 대한 각 gradience는, 미분법칙에 따라 $\partial y/\partial a = b * a^{(b-1)}$, $\partial y/\partial b = a^b * \log(a)$ 이고, chain rule에 의하여 $\partial L/\partial a = \partial L/\partial y * \partial y/\partial a = \partial L/\partial y * (b * a^{(b-1)})$, $\partial L/\partial b = \partial L/\partial y * \partial y/\partial b = \partial L/\partial y * (a^b * \log(a))$이다.

<u>Code Interpretation</u>: forward에서는 np.power(a,b)의 형태로 거듭제곱 연산을 수행, 결과값을 반환한다. 또한 gradience 계산에 a와 b의 값이 이용되기 때문에 a, b, a.shape, b.shape를 ctx에 저장한다. backward에서는 변수 a, b 각각에 대한 gradience를 구한다. 위의 수식에 따라 연산을 수행하여 grad_a_raw, grad_b_raw에 저장한다. 이후 a, b의 본래 shape으로 gradient를 맞추고 반환한다.

4) class Sum

Sum 클래스는 a의 원소들의 합계 연산을 수행한다. Sum의 forward propagation 연산 수식은 $y = \sum(a)_{i,j,k,...}$이다. axis 값에 따라 특정 축을 따라 합 연산도 가능하며 keepdims는 합을 낸 후에도 축을 유지할지 결정한다. 역전파에서 Sum의 a에 대한 gradience는 $\partial L/\partial(a)_{i,j,k,..} = \partial L/\partial y * \partial y/\partial(a)_{i,j,k,..} = \partial L/\partial y$이다.

<u>Code Interpretation</u>: forward에서는 np.sum(a, axis = axis, keepdims = keepdims)의 형태로 합계 연산을 수행, 결과값을 반환한다. backward에서는 위의 수식에 따라 변수 a에 대한 gradience를 구한다. a의 본래 shape으로 gradient 또한 맞추기 위해 axis의 설정 여부와 keepdims의 여부에 따라 다른 처리를 진행한다.

2. Implement Activation Functions

1) class ReLU

ReLU의 forward propagation은 $y=\max(0,x)$로 정의된다. 이는 음수는 0으로, 양수는 그대로 통과시키는 함수이다. backward propagation에서의 도함수는 $\partial y/\partial x=1(x>0)$이며, $\partial L/\partial y$가 주어질 때 입력 기울기는 $\partial L/\partial x=\partial L/\partial y\odot 1(x>0)$가 된다.

<u>Code Interpretation</u>:forward에서는 np.maximum(0,x)로 원소별 ReLU를 계산해 반환한다. backward에서는 저장된 x와 grad_output을 받아, x>0 인 위치에서만 기울기를 통과시키는 방식으로 grad_x를 계산한다. ReLU는 보통 출력이 입력과 동일한 shape을 유지하므로, shape 축소는 필요하지 않다.

2) class Softmax

softmax는 입력받은 값들을 0과 1 사이의 확률값으로 변환하는 활성화함수이다. 이때 출력값의 총합은 1이 되도록 정규화한다. forward propagation은 $y_i=e^{z_i} / \sum_j e^{z_j}$ 으로 정의된다. 수치 안정성을 위해 같은

axis의 최댓값 $m=\max_j z_j$를 빼서 e^{z_i-m}으로 계산하며, 이때 값은 동일하다. backward propagation에서 softmax의 야코비안 J는 $J_{ij}=y_i(\delta_{ij}-y_j)$이고, $g=\partial y/\partial L$에 대한 $\partial L/\partial z$는 $\partial L/\partial z=J^{\top}g=y\odot(g-\langle g,y\rangle_{axis})$가 된다.
<u>Code Interpretation</u>: 위의 수식에서 설명한 내용을 코드로 구현했다. forward에서는 입력에서 최댓값을 빼서 시프트하고(언더,오버플로우 방지), exp_z를 계산하고, 해당 값을 정규화 해서 반환한다. backward에서는 야코비안-벡터곱 연산을 보다 간단하게 수행하기 위해 입력 z 대신 출력값 y자체를 저장한다. 저장된 y와 grad_output을 받아, 위의 수식 연산을 수행하고 grad_z를 얻는다.

3) class Log

Log 클래스는 x의 원소별 로그 연산을 수행한다. Log의 forward propagation 연산 수식은 $y = \log(x)$이다. 여기서 로그 연산의 안정성을 위해 0 혹은 더 작은 값이 지수가 되지 않도록 주의가 필요하다. backward propagation에서 Log의 x에 대한 gradience는 $\partial L/\partial x = \partial L/\partial y * \partial y/\partial x = (\partial L/\partial y)/x$이다. 여기서도 앞과 비슷하게 연산의 안정성을 위해 0이 분모가 되어선 안 되도록 처리해주어야 한다.
<u>Code Interpretation</u>: forward에서는 np.log(a, axis = axis, keepdims = keepdims)의 형태로 로그 연산을 수행, 결과값을 반환한다. 여기서 np.clip으로 $(x)_{i,j,k,...} < eps\ (= 10^{-12})$인 원소는 eps로 바꾸어서 연산이 수행되도록 한다. backward에서는 위의 수식에 따라 변수 x에 대한 gradience를 구한다. 해당 연산에서도 eps를 활용한다.

3. Implement Loss Functions

1) class CrossEntropyLoss

CrossEntropyLoss 클래스는 loss 함수 중 하나인 cross entropy loss 함수의 기능을 한다. CrossEntropyLoss의 forward propagation 연산 수식은 $y = 1/N * \sum(-(t)_{i,j,k,...} * \log(p)_{i,j,k,...})$ (p is logits, t is targets)이다. backward propagation에서 Sum의 p, t에 대한 각 gradience는 $\partial L/\partial p = \partial L/\partial y * \partial y/\partial p = \partial L/\partial y * (-t / (N * p))$, $\partial L/\partial t = \partial L/\partial y * \partial y/\partial t = \partial L/\partial y * (-\log(p) / N)$이다.
<u>Code Interpretation</u>: forward에서는 np의 mean, sum, log, clip등을 이용하여 위의 수식과 같은 형태로 연산을 만들어 수행, 결과값을 반환한다. np.mean은 산술 평균을 구해준다. backward에서는 위의 수식으로 logits, targets에 대한 각 gradience를 구한다. 연산에 이용하는 배치 크기는 logits의 shape를 이용해 구한다.

2) class NLLLoss

NLLLoss 클래스의 forward 단계 수식은 $L=-1/N\sum\log(p_{i,t_i})$로 정의된다. 이때 N은 배치 사이즈, p_{i,t_i}는 sample i의 정답 클래스 ti에 해당하는 확률값을 의미한다. backward 에서는 p의 각 원소에 대한 기울기를 구하게 된다. 정답 클래스에 해당하는 위치에서는 $\partial L/\partial p_{i,t_i}=-1/(N\cdot p_{i,t_i})$이고, 그 외 위치에서는 0이 된다.
<u>Code Interpretation</u>: forward에서는 각 샘플의 정답 클래스 확률을 위의 수식 연산을 수행한 뒤 반환한다. 이때 np.clip을 이용해서 확률이 0이 되지 않게 한다. backward에서는 배치 사이즈 N, p와 동일한 shape의 배열을 생성한 뒤, 각 샘플의 정답 클래스 위치에만 위의 수식에서 설명한 기울기 값을 채워넣는다. 이 배열이 p에 대한 기울기가 된다.

4. Training MLPs with MNIST Dataset

1) Implement L2 regularizer & Compare with 'w/o L2 regularizer version.'

MLP3 model과 우리가 위에서 구현한 함수들을 이용하여 MNIST 데이터셋(training set, test set으로 나눔)을 훈련(training)하고 정확도를 검증(test)하여 보았다. loss function으로는 CrossEntropyLoss를 이용하였다. L2 regularizer를 추가로 구현하지 않았을 때 training의 일부인 accuracy는 다음과 같다.

```
Epoch 1 / Test accuracy: 81.54% Epoch 10 / Test accuracy: 97.22%
```

L2 regularization은 모델의 overfitting을 방지하기 위한 방안으로, loss에 추가로 더하여 가중치가 너무 커지지 않도록 규제한다. L2 regularizar의 적용은 보통 다음과 같다, $Loss = L(W) + \lambda * \|W\|^2$ *(λ is hyper parameter(정규화 강도), W is weight)*. 여기서 L(W)는 기존의 loss function을 의미하며, L2 regularization에 해당하는 부분은 $\lambda * \|W\|^2$ 이다. 또한 W에 대한 gradience는, 미분법칙에 따라 $\partial L/\partial W = 2 * \lambda * \|W\|$이다.

L2 regularization 구현을 위해 위의 식을 따라 연산 수행, 그 값을 total_loss에 추가하였다. 또한 backward pass에서 L2 regularizar의 gradience 또한 반영하기 위해서 위의 gradience 식을 따라 연산 수행 후 param의 grad에 추가하였다. L2 regularizer를 추가로 구현하였을 때 training의 일부인 accuracy는 다음과 같다.

```
Epoch 10 / Test accuracy: 97.23% Epoch 1 / Test accuracy: 80.80%
```

결과 비교 시 MNIST 데이터셋에서 L2 regularizer 유무에 따른 accuracy 차이가 크지 않다. 이는 MNIST의 데이터 양이 overfitting을 낮출 수 있을 만큼 많아, L2 regularizer의 영향이 낮아 보이는 것이라 추측했다.