영상처리와 딥러닝 프로젝트 01

: cnn-numpy-project

20165306 김승현

1. Training 과정 코드분석

매개변수로 받은 배치 입력, 출력데이터를 mini_batch_inputs, mini_batch_outputs 변수에 저장

#forward

- 입력 받은 데이터를 deque모델을 이용해 스택 자료구조형으로 z_stack 변수에 저장
- 입력 받은 데이터를 activation 초기값으로 지정 (forward와 loss 연산에 사용)
- 반복문을 사용해서 Layers에 저장된 각각의 레이어에 대해 forward 연산을 한다. 반복문 내 변수 z에는 input값에 대한 레이어의 forward 연산 한 결과를 저장하고, 변수 activation에는 forward연산 결과인 z를 레이어의 activation 연산한 결과를 저장한다. 여기서 activation은 레이어들의 forward연산을 진행하기 위해 사용하고 z는 backward연산을 진행하기 위해 사용된다.

#calculate loss

- forward연산 이후 네트워크를 통과한 결과값인 activation과 레이블 outputs와의 loss값을 Network 객체를 선언할 때 지정한 loss연산을 통해 변수 loss_err에 저장한다.

#backward pass

- backward연산을 위해서 z_stack에 레이어의 역순으로 저장해 놓은 각 레이어의 forward 연산 결과들 중 맨 처음 값을 변수 lz의 초기값으로 지정
- 변수 upstream_gradient에 초기값으로 전체 loss 값을 지정
- grads를 빈 스택으로 지정
- 반복문을 사용해서 Layer의 역순으로 변화율을 구한다. 변수 layer_err에는 경사를 수정하기 위해 local gradient와 연산할 값(=상위 레이어에서 내려오는 upstream gradient가 미분된 activation을 통과한 값)을 저장한다.

- 해당 레이어의 결과값을 z_stack에서 바깥쪽부터 하나 추출해서 변수 lz를 최신화하고 layer의 가중치를 수정할 변화율을 구하는 get_grad연산으로 가중치 경사를 grads에 추가해 저장한다. 그리고 layer의 backward 연산으로 입력방향으로의 Downstrem gradient를 구해 Upstream gradient에 저장하고 하위 레이어로 내려 반복한다.

#update

- grads에 레이어 역순으로 값(=각 레이어 가중치의 변화율)을 optimizer의 update함수를 이용해 Regularization한 값을 레이어의 기존 w에 반영한다.
- assert 문을 통해 grads에 남아 있는 데이터가 있는지 점검한다. 남아있는 경우, 예외를 발생시킨다.

2. ReLU 구현

```
class Relu(AbstractActivation):
    def compute(self, x):
        ########################################################################
        # TODO: Implement the ReLU forward pass.                               #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        output = x
        output[output < 0] = 0

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
        #                             END OF YOUR CODE                         #
        ########################################################################

        return output

    def deriv(self, x):
        ########################################################################
        # TODO: Implement the ReLU backward pass.                              #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        local_grad = x
        local_grad[local_grad < 0] = 0
        local_grad[local_grad > 0] = 1

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
        #                             END OF YOUR CODE                         #
        ########################################################################
        return local_grad
```

forward 연산에서 ReLU activation은 if(x<0): 0, else : x 의 결과를 출력한다. input값을 저장한 output에서 0보다 작은 원소의 인덱스에는 0을 넣음으로 compute를 구현하였다. backward연산에서 ReLU activation은 if(x<0) : 0, else : 1의 결과를 출력한다. input값을 저장한 local_grad에서 0보다 작으면 0, 크면 1을 넣음으로 deriv를 구현하였다.

#변경 전

```
# 심플 MLP 예제
lr = 0.01
layers = [
    Flatten((28, 28, 1)), # fully connected 전에 입력을 1차원으로 만들어주는 flatten() 삽입
    FullyConnected((28*28, 100), activation=sigmoid, optimizer = SGDOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (28*28))),
    FullyConnected((100, 50), activation=sigmoid, optimizer = SGDOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (100.))),
    FullyConnected((50, 10), activation=linear, optimizer = SGDOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (50.)))
 ]
```

```
Iteration: 0, loss : 2.338924
Iteration: 10, loss : 2.364508
Iteration: 20, loss : 2.347564
Iteration: 30, loss : 2.314345
Iteration: 40, loss : 2.296761
Iteration: 50, loss : 2.353226
Iteration: 60, loss : 2.352185
Iteration: 70, loss : 2.321809
Iteration: 80, loss : 2.379591
Iteration: 90, loss : 2.414772
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 10.46
```

➔ 위 세 그림은 기존에 주어진 심플 MLP 예제의 코드 그대로 Sigmoid activation을 이용하여 학습을 진행한 코드와 결과이다. 테스트를 하기 위해 iteration을 100회로 설정하고 진행하였다.

#변경 후

```
# 심플 MLP 예제
lr = 0.01
layers = [
    Flatten((28, 28, 1)), # fully connected 전에 입력을 1차원으로 만들어주는 flatten() 삽입
    FullyConnected((28*28, 100), activation=relu, optimizer = SGDOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (28*28))),
    FullyConnected((100, 50), activation=relu, optimizer = SGDOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (100.))),
    FullyConnected((50, 10), activation=linear, optimizer = SGDOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (50.)))
 ]
```

```
Iteration: 0, loss : 2.328836
Iteration: 10, loss : 2.288739
Iteration: 20, loss : 2.244332
Iteration: 30, loss : 2.259751
Iteration: 40, loss : 2.259415
Iteration: 50, loss : 2.238492
Iteration: 60, loss : 2.175522
Iteration: 70, loss : 2.184389
Iteration: 80, loss : 2.129590
Iteration: 90, loss : 2.182612
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 46.58
```

➔ 기존에 주어진 FC Layer들중 Sigmoid activation을 사용한 Layer들을 ReLU activation으로 교체하여학습을 진행, 다른 변화없이 activation만의 변화로 정확도 20정도 상승한 모습으로 보아 ReLU 구현이 잘 동작하는 것으로 볼 수 있다.

3. Adam Optimizer 구현

```
class AdamOptimizer():
    def __init__(self):
        # parameters for Adam
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.m = 0.0
        self.v = 0.0
        self.eps = 1e-8
        self.t = 0

    def update(self, dx, lr = 0.001):
        """
        A implementation of the Adam optimizer.

        Input:
        - dx: gradient of the target weight
        - lr: learning rate

        Returns a tuple of:
        - out: update_value
        """
        ###########################################################################
        # TODO: Implement the Adam optimizer                                      #
        ###########################################################################

        #update_value = None
        self.t += 1
        self.m = self.beta1 * self.m + lr * dx
        self.v = self.beta2 * self.v + (1 - self.beta2) * dx * dx
        first_unbias = self.m / (1 - self.beta1 ** self.t)
        second_unbias = self.v / (1 - self.beta2 ** self.t)
        update_value = lr * first_unbias / (np.sqrt(second_unbias) + self.eps)


        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ###########################################################################
        #                             END OF YOUR CODE                            #
        ###########################################################################
        return update_value
```

Adam Optimizer는 Momentum과 RMSProp을 0부터 시작하는 moment 때문에 gradient가 0에 치우치는 것을 방지하기 위해 bias에 저장해 weight를 update하는 방식으로 동작한다.

- 가장 처음 Momentum과 RMSProp이 0으로 나누어지는 것을 막기 위해 t를 가장 먼저 1 증가시켜준다.

- dx * learning rate에 beta1(람다) * m(momentum) 더해 m에 대입한다.

- beta2(decay) * v(velocity) + (1-beta2) * dx * dx 를 v에 대입한다.

- 구해진 m을 (1-beta1^{t}) 로 나눠 first_bias에 저장

- 구해진 v를 (1-beta2^{t}) 로 나눠 second_bias에 저장

- learning rate * first_bias / (second_bias$^{0.5}$ + eps(1e-7)를 layer로 출력해 weight를 update 한다

```python
# 심플 MLP 예제
lr = 0.01
layers = [
    Flatten((28, 28, 1)), # fully connected 전에 입력을 1차원으로 만들어주는 flatten() 삽입
    FullyConnected((28*28, 100), activation=relu, optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (28*28))),
    FullyConnected((100, 50), activation=relu, optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (100.))),
    FullyConnected((50, 10), activation=linear, optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(1.0 / (50.)))
]
```

```
Iteration: 0, loss : 2.258938
Iteration: 10, loss : 1.984354
Iteration: 20, loss : 1.319482
Iteration: 30, loss : 1.074512
Iteration: 40, loss : 0.809883
Iteration: 50, loss : 0.701885
Iteration: 60, loss : 0.454337
Iteration: 70, loss : 0.434623
Iteration: 80, loss : 0.457611
Iteration: 90, loss : 0.338652
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 88.68

Process finished with exit code 0
```

➔ 위 2번(ReLU 구현) 문제에서 ReLU activation을 사용하여 학습한 것과 비교해 보았을 때 optimizer가 SGDOptimizer인 것과 AdamOptimizer인 것의 차이만으로 정확도가 40정도 오른 것을 볼 수 있다. Adam Optimizer 구현이 잘 동작하는 것으로 볼 수 있다.

4. Weight initialization

기존에 주어진 LeCun, Xavier, He initialization 중 더 나은 선택지를 찾기 위해 단순하게 10회 학습을 통해 비교했을 때 정확도로 최선의 방법을 찾기는 어려웠다. 또한, 여러 번 많은 학습을 하는 것은 어렵다고 판단해 알려진 지식을 이용하기로 했다. 우선, Xavier는 다음 층의 노드 수를 사용하여 LeCun에 비해 더 나은 초기화 방법이라고 생각했다. 그렇지만, Sigmoid같은 비선형함수에 효과적인 결과를 보여주는 Xavier 함수는 ReLU activation에 비효율적인 결과를 보여주는데 이런 경우 He initialization을 사용하는 것이 효과적이라고 알려져 있다.

```
layers = [
    Conv((3, 3, 1, 16), strides=1, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (28*28))),
    Conv((4, 4, 16, 32), strides=2, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (16*26*26))),
    #POOL(pshape = 2),
    Flatten((12, 12, 32)),
    FullyConnected((12*12*32, 256), activation=leaky_relu,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (12*12*32))),
    FullyConnected((256, 10), activation=linear,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (256.)))
]
```

```
Iteration: 0, loss : 2.300276
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 13.58
```

lambda shape : np.random.normal(size=shape) * np.sqrt(2.0 / (n_in))

으로 weight initialization을 구현했다.

5. Build network

```
#Conv(LReLU)-Conv(LReLU)-POOL-flatten-FC-FC
layers = [
    Conv((3, 3, 1, 16), strides=1, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (28*28))),
    Conv((4, 4, 16, 32), strides=2, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (16*26*26))),
    #POOL(pshape = 2),
    Flatten((12, 12, 32)),
    FullyConnected((12*12*32, 256), activation=relu,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (12*12*32))),
    FullyConnected((256, 10), activation=linear,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (256.)))
]
```

```
Iteration: 0, loss : 2.302841
Iteration: 10, loss : 2.300939
Iteration: 20, loss : 2.299861
Iteration: 30, loss : 2.299051
Iteration: 40, loss : 2.300240
Iteration: 50, loss : 2.297511
Iteration: 60, loss : 2.293551
Iteration: 70, loss : 2.294204
Iteration: 80, loss : 2.290968
Iteration: 90, loss : 2.291978
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 43.57
```

➔ 위 결과는 레이어를 Conv-LReLU-Conv-LReLU-faltten-FC-ReLU-FC-Linear-로 하여 100회 학습시킨 결과이다.

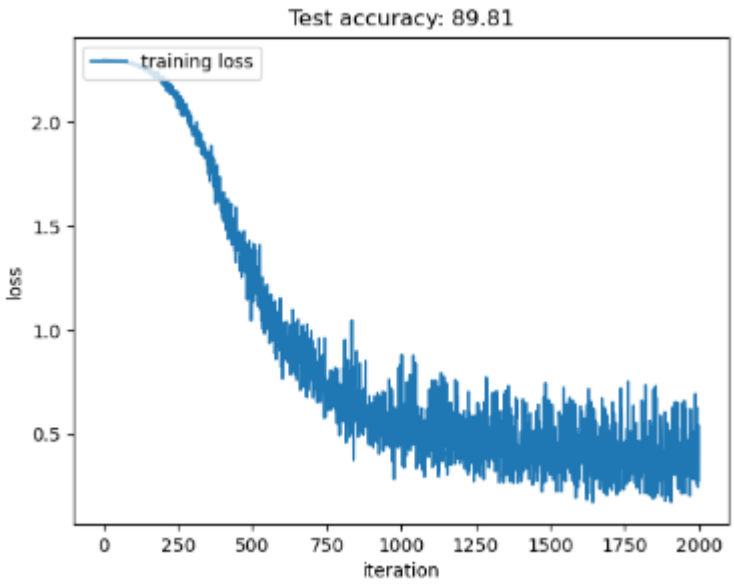
```
layers = [
    Conv((3, 3, 1, 16), strides=1, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (28*28))),
    Conv((4, 4, 16, 32), strides=2, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (16*26*26))),
    POOL(pshape = 2),
    Flatten((6, 6, 32)),
    FullyConnected((6*6*32, 256), activation=relu,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (6*6*32))),
    FullyConnected((256, 10), activation=linear,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (256.)))
]
```

```
Iteration: 0, loss : 2.303371
Iteration: 10, loss : 2.301294
Iteration: 20, loss : 2.306706
Iteration: 30, loss : 2.300957
Iteration: 40, loss : 2.299775
Iteration: 50, loss : 2.304579
Iteration: 60, loss : 2.303073
Iteration: 70, loss : 2.294793
Iteration: 80, loss : 2.298459
Iteration: 90, loss : 2.297250
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 18.71
```

➔ 위 결과는 레이어를 Conv-LReLU-Conv-LReLU-POOL-faltten-FC-ReLU-FC-Linear-로 하여 100회 학습시킨 결과이다.

Max Pooling layer를 네트워크에 추가하여 학습시킨 정확도가 추가하지 않은 네트워크에 비해서 상당하게 떨어지는 모습을 볼 수 있다. 마찬가지로 100회 이상 학습시켰을 때도 Max pooling layer를 잘못 설계했는지 loss가 잘 떨어지지 않는 모습을 보여준다. 따라서, Conv-LReLU-Conv-LReLU-faltten-FC-ReLU-FC-Linear- 구조의 네트워크를 사용해 2000회 학습을 진행했다.

```
Iteration: 1910, loss : 0.628885
Iteration: 1920, loss : 0.656511
Iteration: 1930, loss : 0.239919
Iteration: 1940, loss : 0.460586
Iteration: 1950, loss : 0.660754
Iteration: 1960, loss : 0.310334
Iteration: 1970, loss : 0.496974
Iteration: 1980, loss : 0.285450
Iteration: 1990, loss : 0.450810
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
Accuracy over all test set 89.81
```



➔ 2000회 정도의 학습에 89.81의 정확도가 나타나는 것으로 보아 잘 학습되었다고 볼 수 있다.

6. Max pooling layer

```
def forward(self, inputs):

    s = int(inputs.shape[1] / self.strides)

    outputs = np.zeros((inputs.shape[0], s, s, inputs.shape[3]))

    for n in range(inputs.shape[0]):
        for c in range(inputs.shape[3]):
            for h in range(s):
                for w in range(s):
                    block = inputs[n, (w*self.strides) : (w*self.strides)+self.pshape, (h*self.strides) : (h*self.strides)+self.pshape, c]
                    outputs[n, w, h, c] = np.max(block)

                    maxpos = [n, int(np.argmax(block) / self.pshape), (np.argmax(block) % self.pshape), c]

                    self.cached_data.append(maxpos)
    print("maxpooling forward input shape",inputs.shape)
    print("maxpooling forward output shape",outputs.shape)

    return (outputs, outputs)
```

```
def backward(self, layer_err):

    s = int(layer_err.shape[1] * self.strides)

    dx = np.zeros((layer_err.shape[0], s, s, layer_err.shape[3]))

    for n in range(layer_err.shape[0]):
        for c in range(layer_err.shape[3]):
            for h in range(layer_err.shape[1]):
                for w,i in zip(range(layer_err.shape[1]), self.cached_data):
                    dx[i] = layer_err[n, w, h, c]

    print("maxpooling backward input shape",layer_err.shape)
    print("maxpooling backward output shape",dx.shape)

    return dx
```

forward에서는 output을 저장할 빈 배열을 생성하고 데이터개수(n), 채널(c), 높이(h), 너비(w) 순으로 반복문을 실행하여 각 n과 c의 2차원 평면에서 (stride * w, h) ~ (stride * w, h) + pooling shape 만큼의 크기를 추출해서 가장 큰 값을 output에 입력해 input을 max pooling해 output으로 출력하는 구조이다. 여기서 backward에서 사용하기 위한 max값의 위치를 빈 리스트 cached_data에 저장해 객체에 저장한다.

backward에서는 upstream gradient(=layer_err)로 들어온 forward의 output과 같은 크기의 경사가 저장된 데이터를 cached_data에 저장된 원래 max위치에 저장하고 나머지 공간에는 0으로 저장해 backward의 output으로 출력한다.

Maxpooling Layer 구현이 잘 동작하는지 확인하기 위해서 forward와 backward를 연산할 때 각 입력과 출력 데이터의 크기를 출력하도록 설정해서 실험해보았다.

```
layers = [
    Conv((3, 3, 1, 16), strides=1, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (28*28))),
    Conv((4, 4, 16, 32), strides=2, activation=leaky_relu, optimizer=AdamOptimizer(),
         filter_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (16*26*26))),
    POOL(pshape = 2),
    Flatten((6, 6, 32)),
    FullyConnected((6*6*32, 256), activation=relu,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (6*6*32))),
    FullyConnected((256, 10), activation=linear,
                   optimizer = AdamOptimizer(),
                   weight_init=lambda shp: np.random.normal(size=shp) * np.sqrt(2.0 / (256.)))
]
```

```
maxpooling forward input shape (50, 12, 12, 32)
maxpooling forward output shape (50, 6, 6, 32)
maxpooling backward input shape (50, 6, 6, 32)
maxpooling backward output shape (50, 12, 12, 32)
maxpooling forward input shape (50, 12, 12, 32)
maxpooling forward output shape (50, 6, 6, 32)
Iteration: 0, loss : 2.300590
#### 학습 종료 #####
Calculate accuracy over all test set (시간 소요)
maxpooling forward input shape (10000, 12, 12, 32)
maxpooling forward output shape (10000, 6, 6, 32)
Accuracy over all test set 10.40
```

➔ pool size : 2X2, stride 2 로 설정하고 진행한 결과 forward 연산에서 width와 height가 ÷2, backward 연산에서는 반대로 x2가 된 것으로 보아 max pooling layer 구현은 잘 동작되는 것으로 보인다.