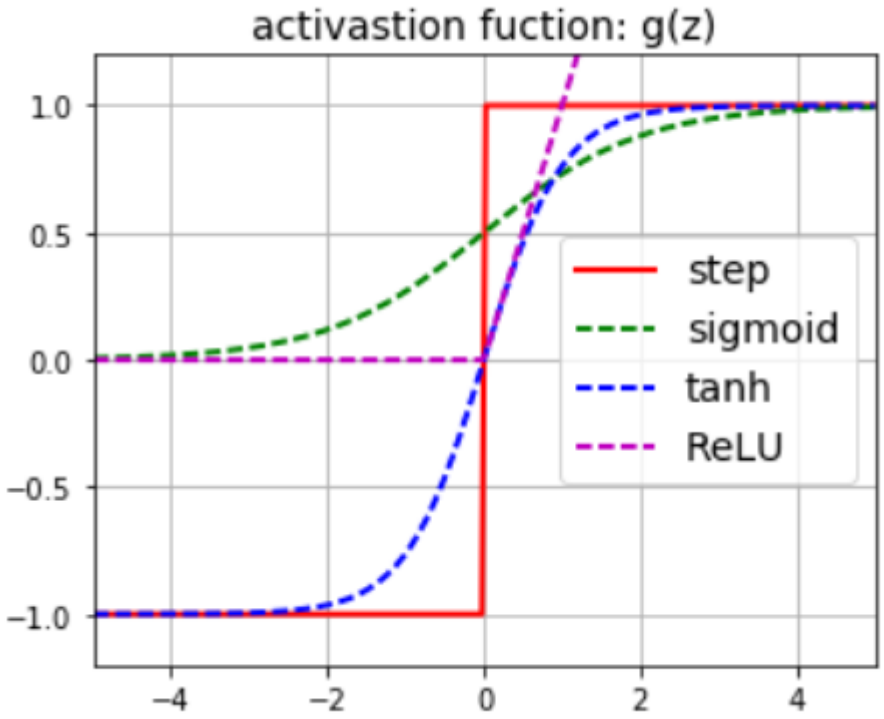
인공지능2

20130940 장용훈

1-1

```
# 로지스틱 함수 의 정의
def logit(z):
    return 1 / (1+np.exp(-z))

#relu 함수의 정의
def relu(z):
    return np.maximum(0,z)
# 각 활성 함수들의 미분
def derivative(f, z, eps=0.000001):
    return (f(z+eps)-f(z-eps))/(2*eps)

z = np.linspace(-5,5,200) # -5~5 200개의 간격 생성

plt.figure(figsize=(11,4))
plt.subplot(121)
#  활성함수로 계단함수
plt.plot(z,np.sign(z),"r-",linewidth=2,label="step")
# 활성함수로 로지스틱함수
plt.plot(z,logit(z),"g--",linewidth=2,label="sigmoid")
```

```
# 활성함수로 하이퍼블릭 탄젠트
plt.plot(z,np.tanh(z),"b--",linewidth=2,label="tanh")
# 활성함수로 relu함수
plt.plot(z,relu(z),"m--",linewidth=2,label="ReLU")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.title("activastion fuction: g(z)", fontsize=14)
plt.axis([-5,5,-1.2,1.2])
plt.show()
```

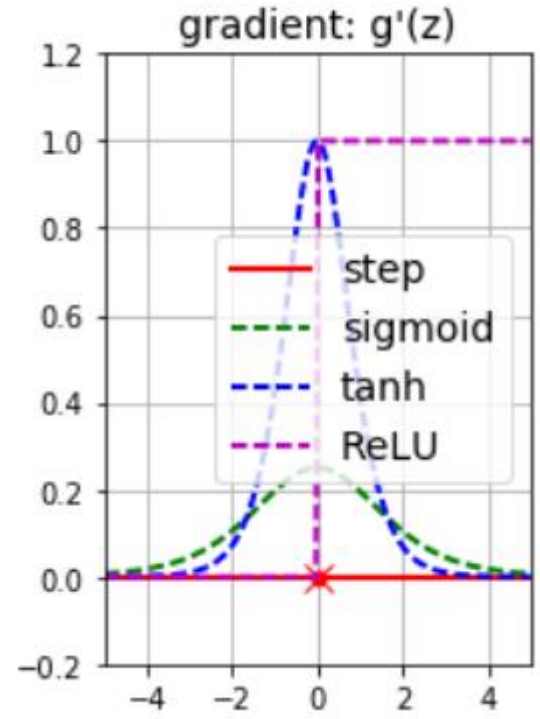


흔히 쓰이는 활성함수 4가지를 나타내주었다. 계단함수는 -1과 1의 범위를 가지고, 로지스틱 시그모이드는 (0,1) 하이퍼블릭 탄젠트는 (-1,1)

relu함수는 [0,inf] 의 값을 가진다. 계단함수는 딱딱한 의사결정을 이를 다른 함수들은 부드러운 의사결정으로 확장하였다.

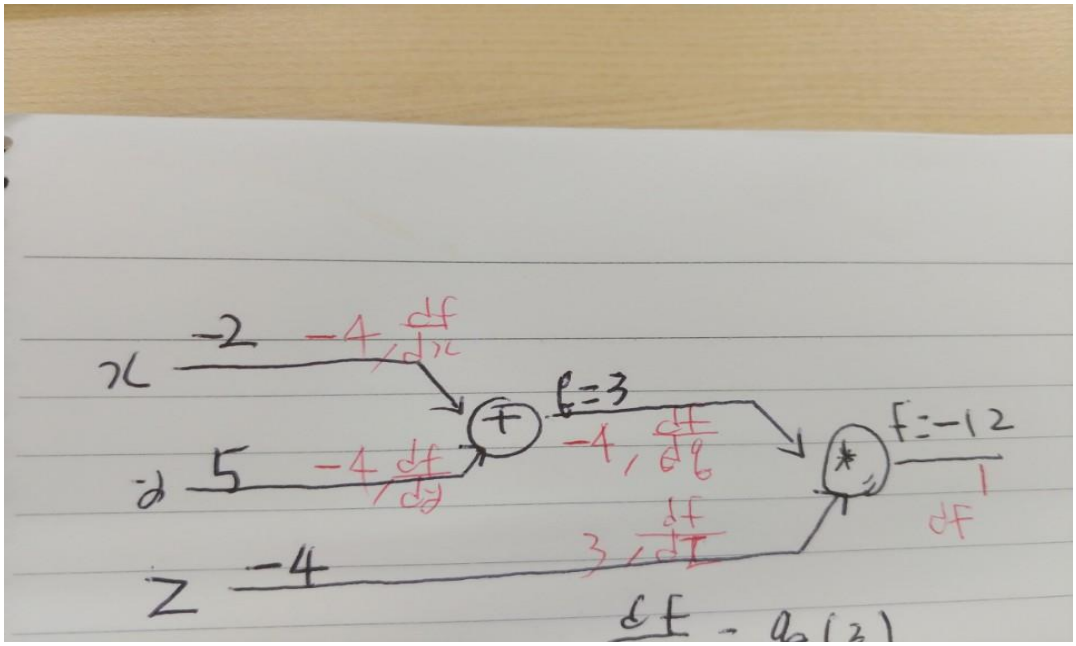
로지스틱 시그모이드나 하이퍼볼릭 탄젠트 시그모이드는 매끄러운 곡선 모양이며 모든 구간에서 미분 할 수 있다. 매개변수에 따라 기울기가 결정되며 이 매개 변수가 클수록 가파르며 무한대이면 계단함수가 된다

```
plt.subplot(122)
#1차 도함수를 이용하여 계단함수, 로지스틱 시그모이드, 하이퍼볼릭 탄젠트, relu 함수를 그래프로 나타냈다.
plt.plot(z,derivative(np.sign,z),"r-",linewidth=2,label="step")
plt.plot(0,0,"ro",markersize=5)
plt.plot(0,0,"rx",markersize=10)
plt.plot(z,derivative(logit,z),"g--",linewidth=2,label="sigmoid")
plt.plot(z,derivative(np.tanh,z),"b--",linewidth=2,label="tanh")
plt.plot(z,derivative(relu,z),"m--",linewidth=2,label="ReLU")
plt.grid(True)
plt.title("gradient: g'(z)", fontsize=14)
plt.legend(loc="center right", fontsize=14)
plt.axis([-5,5,-0.2,1.2])
plt.show()
```



퍼셉트론은 계단함수를 사용하였고 다층 퍼셉트론은 로지스틱 시그모이드나 하이퍼볼릭 탄젠트를 사용하였다. Relu 함수는 딥러닝에 주로 이용되는데, 그래디언트 소멸 문제 완화에 크게 도움이 된다. 또한 relu의 그래디언트는 비교 연산 한 번으로 계산 가능하여 속도가 빠르다.

2-1



15번 문제와 동일한 연산임으로 이를 통해 보이고자 한다. 처음 x,y,z의 값이 검은색과 같이 연산이 된다. 그 후 오른쪽에서 왼쪽으로 오류역전을 통해 gradient가 갱신이 된다.

2-2

```
#2
np.random.seed(0)
N,D = 3,4

#3x4행렬 생성
x = np.random.randn(N,D)
y = np.random.randn(N,D)
z = np.random.randn(N,D)

a = x*y
b = a+z
c = np.sum(b)
# 출력에서 의 gradient는 1로 임의로 설정
grad_c = 1.0
grad_b = grad_c * np.ones((N,D))
# 미분에서 연쇄법칙을 이용하기에 grad_b를 복사한다.
grad_a = grad_b.copy()
grad_z = grad_b.copy()
# 최종적인 x와 y의 gradient이다. 이는 연산을 모두 수행후 다시 되돌아가는 오류역전파 이다.
grad_x = grad_a*y
grad_y = grad_a*x
print("grad_c: ",grad_c,"\ngrad_b: ",grad_b,"\ngrad_a: ",grad_a,"\ngrad_z: ",grad_z,"\ngrad_x: ",grad_x,"\ngrad_y: ",grad_y)
```

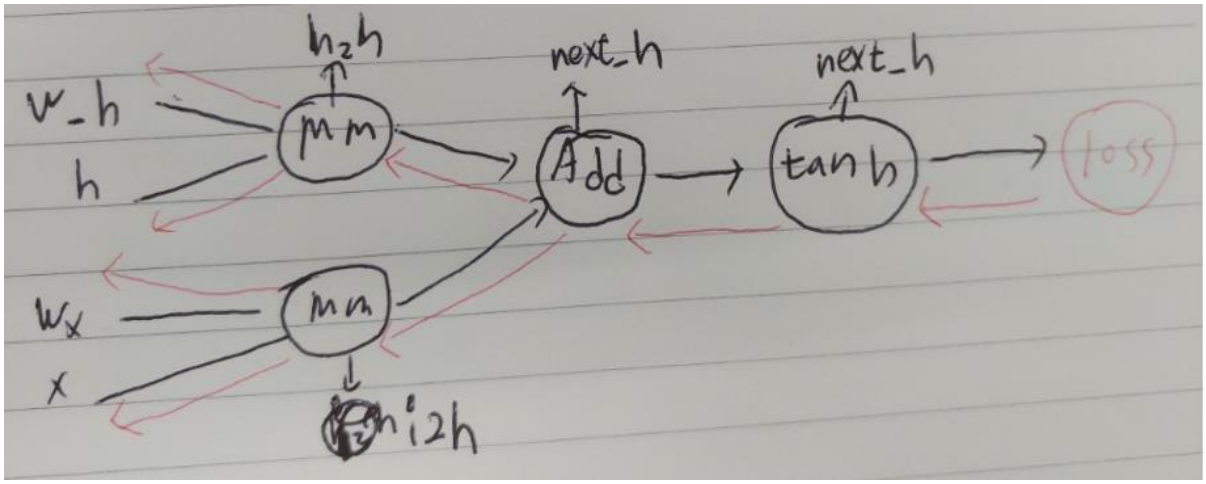
```
grad_c:  1.0
grad_b:  [[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
grad_a:  [[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
grad_z:  [[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
grad_x:  [[ 0.76103773  0.12167502  0.44386323  0.33367433]
 [ 1.49407907 -0.20515826  0.3130677  -0.85409574]
 [-2.55298982  0.6536186   0.8644362  -0.74216502]]
grad_y:  [[ 1.76405235  0.40015721  0.97873798  2.2408932 ]
 [ 1.86755799 -0.97727788  0.95008842 -0.15135721]
 [-0.10321885  0.4105985   0.14404357  1.45427351]]
```

2-3

```
import torch
N,D = 3,4
#tensor를 생성하고 연산을 추적하기 위해 requires_grad=True로 설정, gradient를 자동으로 계산해준다.
x = torch.randn(N,D,requires_grad=True)
y = torch.randn(N,D,requires_grad=True)
z = torch.randn(N,D,requires_grad=True)

a = x*y
b = a+z
c = torch.sum(b)
print(c.backward)
print(x.backward,"\n",y.backward)
```

```
<bound method Tensor.backward of tensor(-0.5948, grad_fn=<SumBackward0>)>
<bound method Tensor.backward of tensor([[-0.1727,  2.2792,  1.1866,  0.5243],
        [-0.2176, -1.5115,  1.0111, -0.3339],
        [ 0.3056, -1.2726, -1.2117,  0.1414]], requires_grad=True)>
 <bound method Tensor.backward of tensor([[-0.8094, -1.0744,  0.6850,  1.6515],
        [-0.1589, -1.5817, -0.0858, -0.3532],
        [-0.1780, -0.2835,  0.2819, -0.5667]], requires_grad=True)>
```

3-1



처음에는 왼쪽에서 오른쪽으로 검정색을 따라 forward 연산이 진행이 된다. 그 후 loss를 계산하고 그 후 backward 연산이 수행된다.(빨간색) backward 연산에서 gradient들을 구하고 이에 따라 가중치 w가 갱신이 된다.

3-2

```
#3
import torch
x = torch.randn(1,10)
prev_h = torch.randn(1,20)
w_h = torch.randn(20,20)
w_x = torch.randn(20,10)
# mm은 행렬의 곱셈 i2h와 h2h를 구함
i2h = torch.mm(w_x,x.t())
h2h = torch.mm(w_h, prev_h.t())

next_h = i2h + h2h
# 활성함수로 하이퍼볼릭 탄젠트 사용, next_h에 적용
next_h = next_h.tanh()
loss = next_h.sum()
# gradient 계산 loss부터(마지막) 처음까지 역방향으로 계산
loss.backward
# 처음 일차적으로 계산이 이뤄 진 후 다시 뒤에서 부터 처음으로 gradient를 구하게 되고 이에 따라 loss가 갱신이 된다.
```

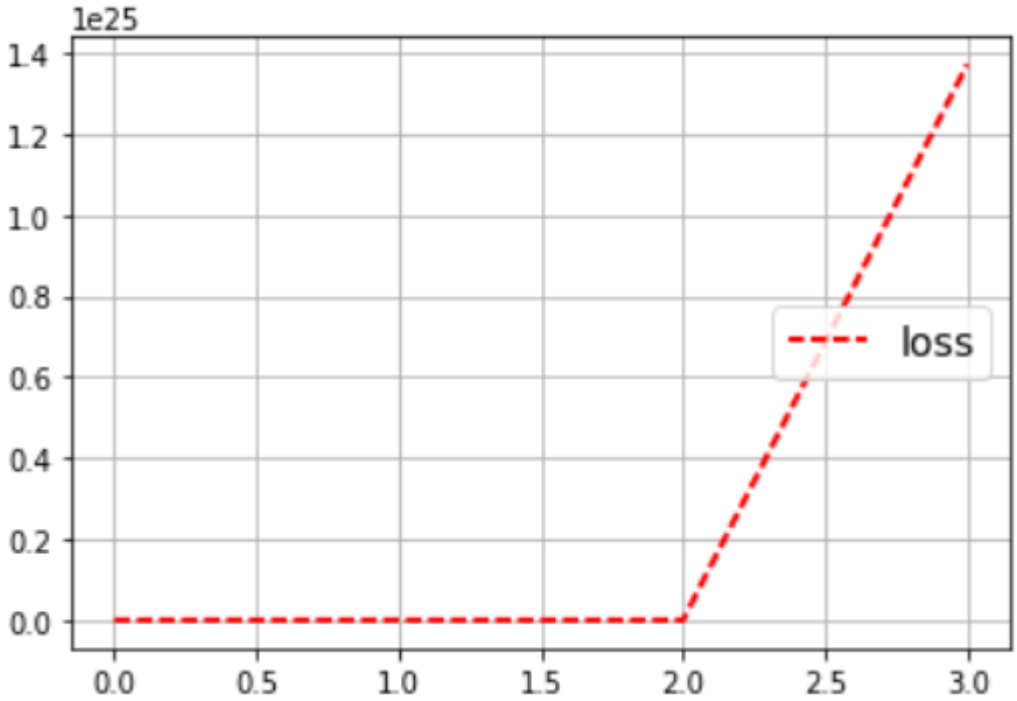
```
<bound method Tensor.backward of tensor(4.1723)>
```

여기서 각 matrix를 생성 할 때, ***requires_grad*** 옵션을 true로 설정해 주지 않았기 때문에 backward과정에서 gradient값들이 자동으로 저장되지는 않는다. 여기서 보여주고자 하는 것은 간단한 예로부터 backward 연산에따라 gradient 나아가 weight가 변함을 보여준다.

4-1

```
N,D_in, H, D_out = 64, 1000, 100, 10
# requires_grad=False gradient가 자동적으로 저장되지 않는다.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H)
w2 = torch.randn(H, D_out)
learning_rate = 1e-6
k = []
for t in range(500):
    # Forward pass, y이 예측값을 구한다.
    h = x.mm(w1)
    #relu 함수정의. h의 값들은 0이상이다.
    h_relu = h.clamp(min=0)
    #relu 와  w2 의 곱은 y pred
    y_pred = h_relu.mm(w2)

    # MSE로 loss를 구한다.
    loss = (y_pred - y).pow(2).sum()

    k.append(loss.data[0])
```

```
    # 경사하강법(Gradient Descent)을 사용하여 가중치를 수동으로 갱신
    w1 = w1-learning_rate * grad_w1
    w2 = w2-learning_rate * grad_w2
# 2개의 층을 이용하여 신경망 학습을 진행, 이때 autogradient 를 하지 않고 수동으로 갱신.
plt.plot(k,"r--",linewidth=2,label="loss")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.show()
# 학습이 잘 이루어 지지 않는다. 왜냐하면 학습률이 너무 높기 때문이다. 학습률을 낮출 필요가 있다.
# 활성함수로 LELU를 사용 하였고, LOSS 함수로 MSE를 사용 하였으며 gradient는 경사하강법을 이용하여 구하였다,
```



4-2

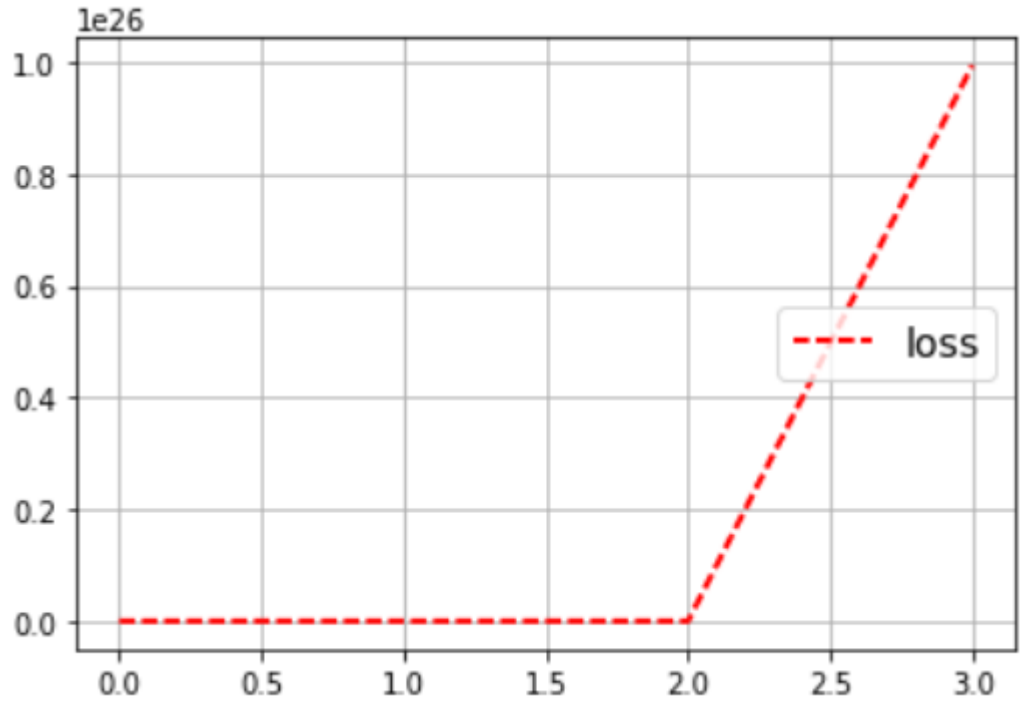
학습이 잘 이루어 지지 않는다. 왜냐하면 학습률이 너무 높기 때문이다. 학습률을 낮추어야 필요가 있다. Forward 과정 후 backward과정이 이루어 지는데 이때, gradient값이 계속해서 초기화 되기 때문에 누적이 이루어 지지 않기 때문이다. 학습을 할 때, RELU 활성함수로 이용하였고 손실함수로 MSE를 사용하였다. 특히 backward과정이 수동으로 이루어 지기 때문에, 그에 따른 weight 또한 매 수행마다 갱신이 된다.

5-1

```
#5
N, D_in, H, D_out = 64, 1000, 100, 10

# 가중치를 저장하기 위해 무작위 값을 갖는 Tensor를 생성
#requires_grad=False gradient를 계산할 필요가 없다.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# requires_grad=True로 설정하여 역전파 중에 이 Tensor들에 대한 gradient를 계산 함.
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)
learning_rate = 10e-6
k = []
```

```python
for t in range(500):
    # fowarad 연산을 사용하여 y 값을 예측
    # 4번과 같이 foward 단계가 완전히 동일하지만, backward는 직접 구현 하지 않음.
    #4번과 동일한 y_pred 값
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # Tensor 연산을 사용하여 손실을 계산하고 출력
    # loss.data[0]은 손실(loss)의 스칼라 값
    loss = (y_pred - y).pow(2).sum()
    k.append(loss.data[0])
    # autograde를 사용하여 역전파 단계를 계산. 이는 requires_grad=True를 w1, w2에 대하여 모두 행해진다.
    loss.backward()
    # requires_grad flag 를 false로 만들고 가중치를 갱신, 이 때는 gradient가 자동으로 계산되지 않는다.
    # 경사하강법(Gradient Descent)을 사용하여 가중치를 수동으로 갱신
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
```

```python
    # 가중치 갱신 후에는 수동으로 변화도를 0으로 만든다. 가중치의 누적을 막기 위해서이다.
        w1.grad.zero_()
        w2.grad.zero_()
plt.plot(k,"r--",linewidth=2,label="loss")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.show()
#  4번과 동일한 역활을 하지만 다르게 구현하였다. 5번의 경우 weight의 gradient 계산을 torch의 autograd 를 이용한 점
# (이는 연산 gradient를 추적하여 저장해준다)이 가장 큰 차이점이다.
# 나아가 5번에서는 가중치 갱신을 할 때, grad 값이 자동으로 저장 되지 않도록 grad flag를 false로 변경해 준다. 4번에서는 backward과정을
# 직접 정의하여 수행했다면, 5번에서는 backward과정을 torch 에서 정의 된 함수를 이용 하였다. 학습 결과는 동일하다.
# 마찬가지로 학습률이 너무 높기 때문에 올바르게 학습이 되지 않고 있다.
```



5-2

4번과 동일한 역할을 하지만 다르게 구현하였다. 5번의 경우 weight의 gradient 계산을 torch의 autograd 를 이용한 점(이는 연산 gradient를 추적하여 저장해준다)이 가장 큰 차이점이다.

나아가 5번에서는 가중치 갱신을 할때, grad 값이 자동으로 저장되지 않도록 grad flag를 false로

변경해 준다. 4번에서는 backward과정을직접 정의하여 수행했다면, 5번에서는 backward과정을 torch 에서 정의 된 함수를 이용 하였다. 학습 결과는 동일하다. 마찬가지로 학습률이 너무 높기 때문에 올바르게 학습이 되지 않고 있다. 학습률을 낮춘다면 학습이 더 잘될 것이다. 복잡한 코드를 간소화 했다.

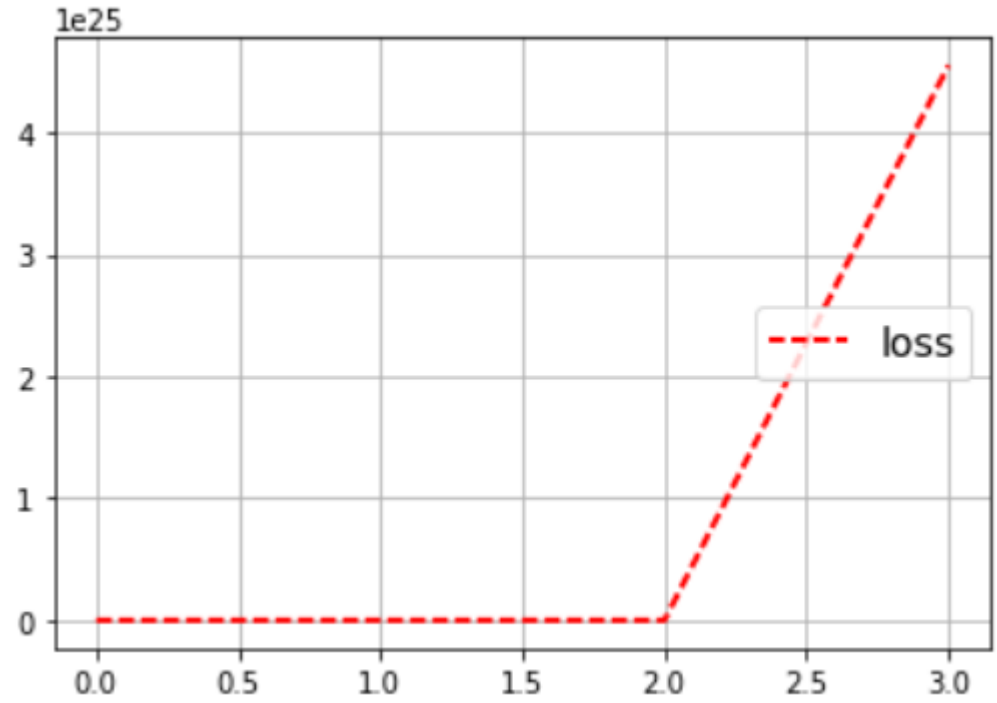
6-1

```
#autograd 함수를 정의
class MyReLU(torch.autograd.Function):
    # torch.autograd.Function을 상속받아 사용자 정의 autograd 함수를 구현하고, fowarad 와 backward를 구현
  @staticmethod
  def forward(ctx, x):
    """
        fowarad 단계에서는 입력을 갖는 Tensor를 받아 출력을 갖는 Tensor를 반환한다.
        ctx는 역전파 연산을 위한 정보를 저장하기 위해 사용하는 Context Object이다.
        ctx.save_for_backward method를 사용하여 역전파 단계에서 사용할 어떠한
        객체(object)도 저장해 둘 수 있다.
    """
    ctx.save_for_backward(x)
    #x 의 값은 0이상이다 relu를 사용
    return x.clamp(min=0)

  def backward(ctx, grad_output):
    """
        역전파 단계에서는 출력에 대한 손실의 변화도를 갖는 Tensor를 받고, 입력에
        대한 손실의 변화도를 계산한다다
    """
    #foward 에서 저장한 값을 backward에서 받아온다. 3번의 backward과정과 동일하다.
    #이때 값은 0이상이다.
    x, = ctx.saved_tensors
    grad_x = grad_output.clone()
    grad_x[x < 0] = 0
    return grad_x
```

```
N, D_in, H, D_out = 64, 1000, 100, 10

# 가중치를 저장하기 위해 무작위 값을 갖는 Tensor를 생성
#requires_grad=False gradient를 계산할 필요가 없다.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# requires_grad=True로 설정하여 역전파 중에 이 Tensor들에 대한 gradient를 계산 함.
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)
k = []
learning_rate = 10e-6
```

```
for t in range(500):

  ###Forward 과정 y의 예측치를 계산하는데, 이때 위에서 정의한 MyRelu를 이용한다
    y_pred = MyReLU.apply(x.mm(w1)).mm(w2)

  # mse 를 비용함수로 loss를 구해준다.
    loss = (y_pred - y).pow(2).sum()
    k.append(loss.data[0])
#a = loss.item()
    #k = np.append(k,a)
    #t1 = np.append(t1,t)
# autograde를 사용하여 backward 를 계산
    loss.backward()
# requires_grad flag 를 false로 만들고 가중치를 갱신
    with torch.no_grad():
# 경사하강법을 이용하여 가중치(w) 를 갱신
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

# 가중치 갱신 후에는 수동으로 변화도를 0으로 만든다. backward 연산시 누적이 되기 때문.
    w1.grad.zero_()
    w2.grad.zero_()
```

```
plt.plot(k,"r--",linewidth=2,label="loss")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.show()
```



6-2

5번과 비교하자면, 유사한 결과를 보여준다. 하지만 학습을 할 때, 제공되는 것이 아닌 직접 forward 연산과 backward 연산을 정의하였다. 이를 정의한 클래스는 myRelu이다. 결론적으로 말하자면 4,5,6번의 경우 동일한 학습결과를 보여준다. 여전히 학습률 때문에 학습이 잘 이루어 지지 않는다.

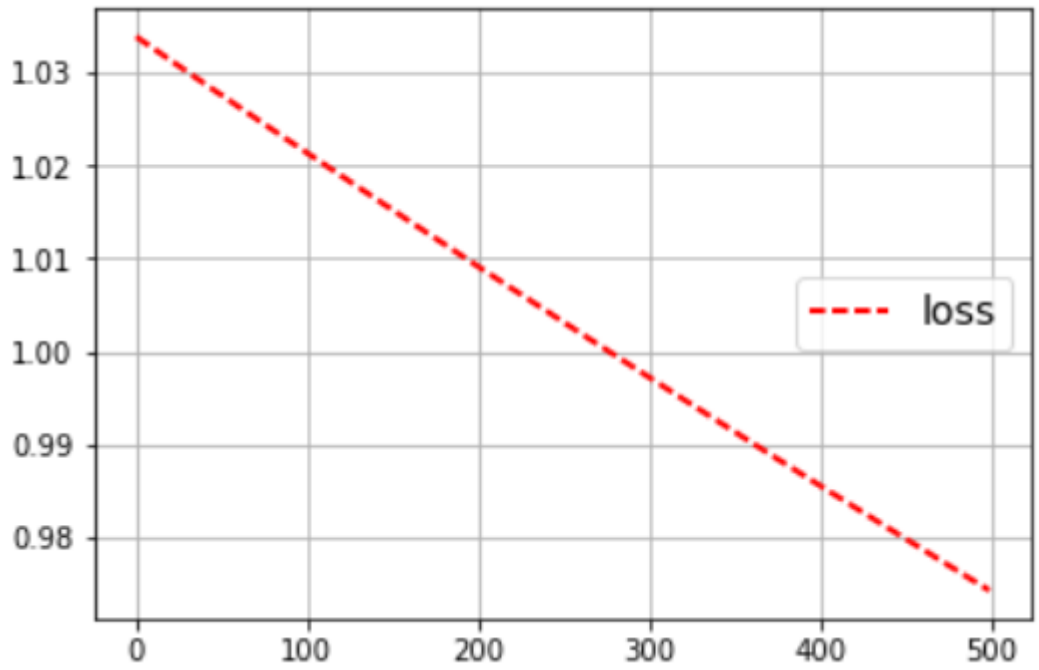
7-1

```
#7
class TwoLayerNet(torch.nn.Module):
  def __init__(self, D_in, H, D_out):
    """
    생성자에서 2개의 nn.Linear 모듈을 생성(Instantiate)하고, 멤버 변수로
    지정한다.
    """
    super(TwoLayerNet, self).__init__()
    self.linear1 = torch.nn.Linear(D_in, H)
    self.linear2 = torch.nn.Linear(H, D_out)

  def forward(self, x):
    """
    forward에서 입력 데이터의 Tensor를 받아서 출력 데이터의 Tensor를
    반환한다. Tensor 상의 임의의 연산자뿐만 아니라 생성자에서 정의한
    모듈을 사용할 수 있다. 활성함수로는 relu함수를 이용하였다.
    y에 대한 예측치 y_pred를 리턴한다.
    """
    h_relu = self.linear1(x).clamp(min=0)
    y_pred = self.linear2(h_relu)
    return y_pred
```

```
N, D_in, H, D_out = 64, 1000, 100, 10

# 가중치를 저장하기 위해 무작위 값을 갖는 Tensor를 생성
# requires_grad=False gradient를 계산할 필요가 없다.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 클래스를 생성해서 모델을 구성한다.
model = TwoLayerNet(D_in, H, D_out)

# 손실함수와 Optimizer를 만든다. SGD 생성자에서 model.parameters()를 호출하면
# 모델의 멤버인 2개의 nnLinear 모듈의 학습 가능한 매개변수들이 포함된다.
# 손실 함수를 nn에서 호출하여 사용한다.
# 스토캐스틱 경사 하강법을 사용. 램덤하게 추출한 일부 데이터에 대해 가중치를 조절. 뿐만아니라 학습률이 다르다.
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
```

```
k = []
for t in range(500):
#Forwardr과정, 앞서 정의한 TwoLayerNet을 이용한다.
    y_pred = model(x)

# loss를 계산한다.
    loss = loss_fn(y_pred, y)
    k.append(loss.data[0])

# 갱신할 Variable들에 대한 모든 변화도를 0으로 만든다. backward 연산을 수행 할 때마다 변화도가 누적되기 때문이다.
    optimizer.zero_grad()
# backward 단계: 모델의 매개변수에 대한 손실의 변화도를 계산
    loss.backward()
# backward과정에서 얻은 gradient에 따라 parameter를 업데이트 해준다.
    optimizer.step()
```

```
plt.plot(k,"r--",linewidth=2,label="loss")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.show()
```



7-2

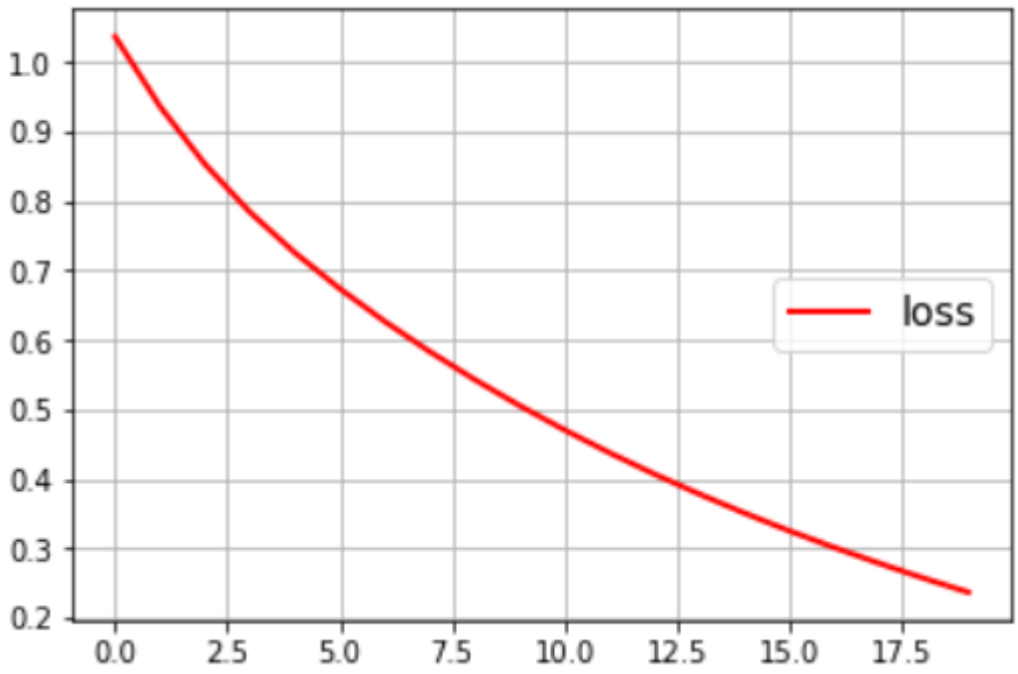
6번과는 다른 방식으로 학습을 수행했다. 우선 TwoLayerNet이라는 클래스를 정의하여 앞에서는 backward과정을 위해 context를 저장 하는 것이 있었다면 여기서는 존재하지 않는다. 가장 큰 차이점은 다른 학습률과, weight를 조정할 때 스토캐스틱 경사 하강법을 사용 하여 그 결과가 다를 수 있다는 것이다. 다만 스토캐스틱 경사하강법을 사용했기에 속도는 더 빠르다. 즉 많은 학습데이터를 처리할 때 더 효율적이다. 6번에 비하여 학습률을 높였기 때문에 loss가 원하는 방향으로 향하고 있다. 따라서 학습이 6번에 비하여 잘 되고 있다고 할 수 있다. 하지만 더 높인 다면 원하

는 방향에 더 가까워 질 것이다.

8-1

```
#8
from torch.utils.data import TensorDataset, DataLoader
N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
# 랜덤으로 생성한 배열들을 DataLoader 를 이용하여 학습시킬 준비를함. 이때 배치 사이즈는 8
loader = DataLoader(TensorDataset(x,y),batch_size=8)
# 7번에서 정의한 TwoLayerNet을 모델로 사용
model = TwoLayerNet(D_in,H,D_out)
# 스토캐스틱 하강법 사용, 학습률은 1e-2, 배치사이즈는 위에서 정의한 8
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
```

```
k = []
for epoch in range(20):
# y예측 값을 7번에서 정의한 모델로 구한다.
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
# loss를 mse를 이용하여 구함.
        loss = torch.nn.functional.mse_loss(y_pred,y_batch)
# backward 과정(gradient 를 자동으로 계산해준다.)
        loss.backward()
# backward과정에서 얻은 gradient에 따라 parameter를 업데이트 해준다.
# step을 진행 시마다 gradient가 누적되기 때문에, step 진행후에 gradient를 zero_grad()를 호출 해주어야 한다.
        optimizer.step()
        optimizer.zero_grad()
    k.append(loss.data[0])

plt.plot(k,"r-",linewidth=2,label="loss")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.show()
```



8-2

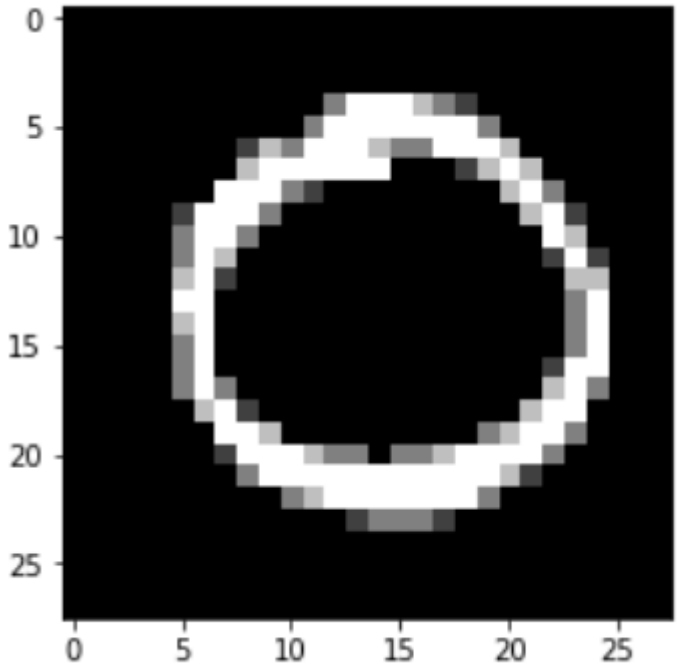
7번과 비교 했을 때, 동일한 모델을 사용하여 예측값을 만들었고, 동일한 gradient algoritm(SGD)를 사용했다. 다만 배치 사이즈를 정의 해주었고 학습이 수행되는 횟수와, 학습률이 다르다. 결론적으로 배치 사이즈와 더 적절한 학습률 덕분에 loss가 점점 0으로 수렴해 가는 모습을 볼 수 있다. 따라서 학습이 잘 되어가고 있다고 말할 수 있다.

9.

```
#9
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets,transforms
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline

# cuda의 사용 여부를 나타낸다.
is_cuda=False
if torch.cuda.is_available():
    is_cuda = True

# Compose 함수를 이용하여 입력데이터를 텐서로 변환 시키고 정규화 시킨다.
transformation = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.1307,), (0.3081,))])

# 훈련과 평가에 쓰일 MNIST데이터를 다운받고
# 훈련과 평가에 쓰일 데이터 셋을 정의 한다. 이때, 데이터를 앞서 정의한 transformation에 따라 변환시킨다.
train_dataset = datasets.MNIST('data/',train=True,transform=transformation,download=True)
test_dataset = datasets.MNIST('data/',train=False,transform=transformation,download=True)

# dataloader를 이용하여 data를 load 한다.
train_loader = torch.utils.data.DataLoader(train_dataset,batch_size=32,shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset,batch_size=32,shuffle=True)
```

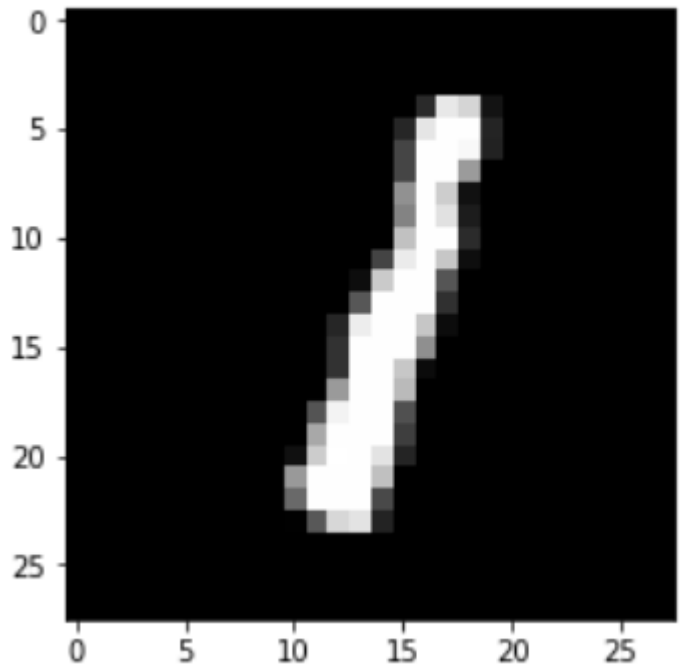
9-1

```
# 무작위로 data를 추출한다.
sample_data = next(iter(train_loader))

def plot_img(image):
    image = image.numpy()[0]
    mean = 0.1307
    std = 0.3082
# mean 과 std를 이용하여 image를 정규화
    image = ((mean * image) + std)
# 이미지를 뿌려준다.
    plt.imshow(image,cmap='gray')

plot_img(sample_data[0][2])
```



9-2

```
plot_img(sample_data[0][1])
```



9-3

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
# 2D convolution으로 conv1, conv2를 만들어 준다. weight 혹은 bias는 자동으로 처리해준다.
# 커널의 사이즈는 5X5로 생성을 한다.
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
# dropout을 설정해 준다. 입력 텐서의 전체 채널을 무작위로 0으로 만든다.
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
# 컨볼루션 층을 지난 후 maxpooling을 2x2 사이즈로 진행하고(이때 최대 값을 뽑아냄), 이를 relu함수로 처리한다
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
# 더 정교하게 진행 하기 위하여, drop out maxpooling을 2x2 사이즈로 진행하고(이때 최대 값을 뽑아냄), 이를 relu함수로 처리한다
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
# x를 다른 사이즈의 텐서로 생성. -1은 n*320 형태의 사이즈로 해줌을 의미. n은 자동으로 계산된다.
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        #x = F.dropout(x,p=0.1, training=self.training)
        x = self.fc2(x)
# 그 후 얻은 값을 소프트맥스 함수를 이용하여 출력 값을 정규화
        return F.log_softmax(x,dim=1)
```

```
model = Net()
if is_cuda:
    model.cuda()
# 스토캐스틱 경사하강법을 0.01의 학습률적용 하는 optimizer 선언
optimizer = optim.SGD(model.parameters(),lr=0.01)
# 학습데이터를 불러온다.
data , target = next(iter(train_loader))
# model을 적용
output = model(Variable(data))
```

```
output.size()
```

```
torch.Size([32, 10])
```

9-4

```
target.size()
```

```
torch.Size([32])
```

9-5

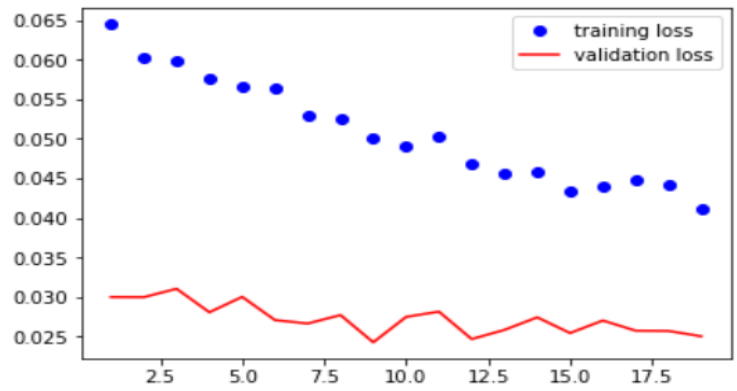
```
# model을 training 하기 위해서 정의한다.
def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
# test를 하기 위해 eval함수를 호출, train을 fale로 변경 시킨다.
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , (data,target) in enumerate(data_loader):
        if is_cuda:
            data,target = data.cuda(),target.cuda()
        data , target = Variable(data,volatile),Variable(target)
# training의 경우 gradient를 초기화 해준다.
        if phase == 'training':
            optimizer.zero_grad()
        output = model(data)
# loss를 구한다. 분류 문제를 훈련시킬 때 많이 사용.nll 방식을 적용
        loss = F.nll_loss(output,target)
# batch loss 를 더한다.
        running_loss = running_loss+F.nll_loss(output,target,size_average=False).data[0]
# max log-probability 를 구한다.
        preds = output.data.max(dim=1,keepdim=True)[1]
```

```
# 예측하여 얻은 값과 주어진 데이터가 얼마나 같은지를 비교하여 이를 누적해준다.
        running_correct = running_correct+preds.eq(target.data.view_as(preds)).cpu().sum()
# training 의 경우 backpropogation 과정이 필요하다. 테스트의 경우에는 분류를 실제로 하기때문에 불필요하다.
        if phase == 'training':
            loss.backward()
            optimizer.step()
# loss를 구해준다
    loss = running_loss/len(data_loader.dataset)
# 정확도가 얼마인지 구한다.
    accuracy = 100. *running_correct/len(data_loader.dataset)
# {accuracy:{10}.{4}} 는 에러때문에 출력이 되지 않아서, 임시로 다른 방법을 사용했습니다. 그러나 loss와
# accuacy는 정확하게 구해졌습니다.
    print(F'{phase} loss is {loss:{5}.{2}} and {phase} accuracy is {running_correct}/{len(data_loader.dataset)}',format(accuracy,

    return loss,accuracy
```

9-6

```
plt.plot(range(1,len(train_losses)+1),train_losses,'bo',label = 'training loss')
plt.plot(range(1,len(val_losses)+1),val_losses,'r',label = 'validation loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d90470dd68>
```



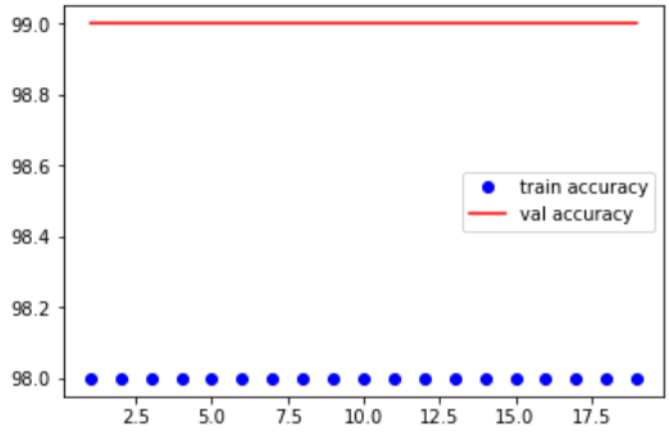
Epoch 가 수행될 때마다 loss의 변화를 보이는 그래프이다.

학습 초기에는 loss가 비교적 크지만, 학습이 진행에 따라 loss의 값이 0으로 수렴함을 볼 수 있다. 학습 이후에 테스트 시행 결과, validation loss 더 작아 진 것이 보인다. 이는 테스트시 분류를 잘고, 학습이 잘 수행되었음을 보인다.

9-7

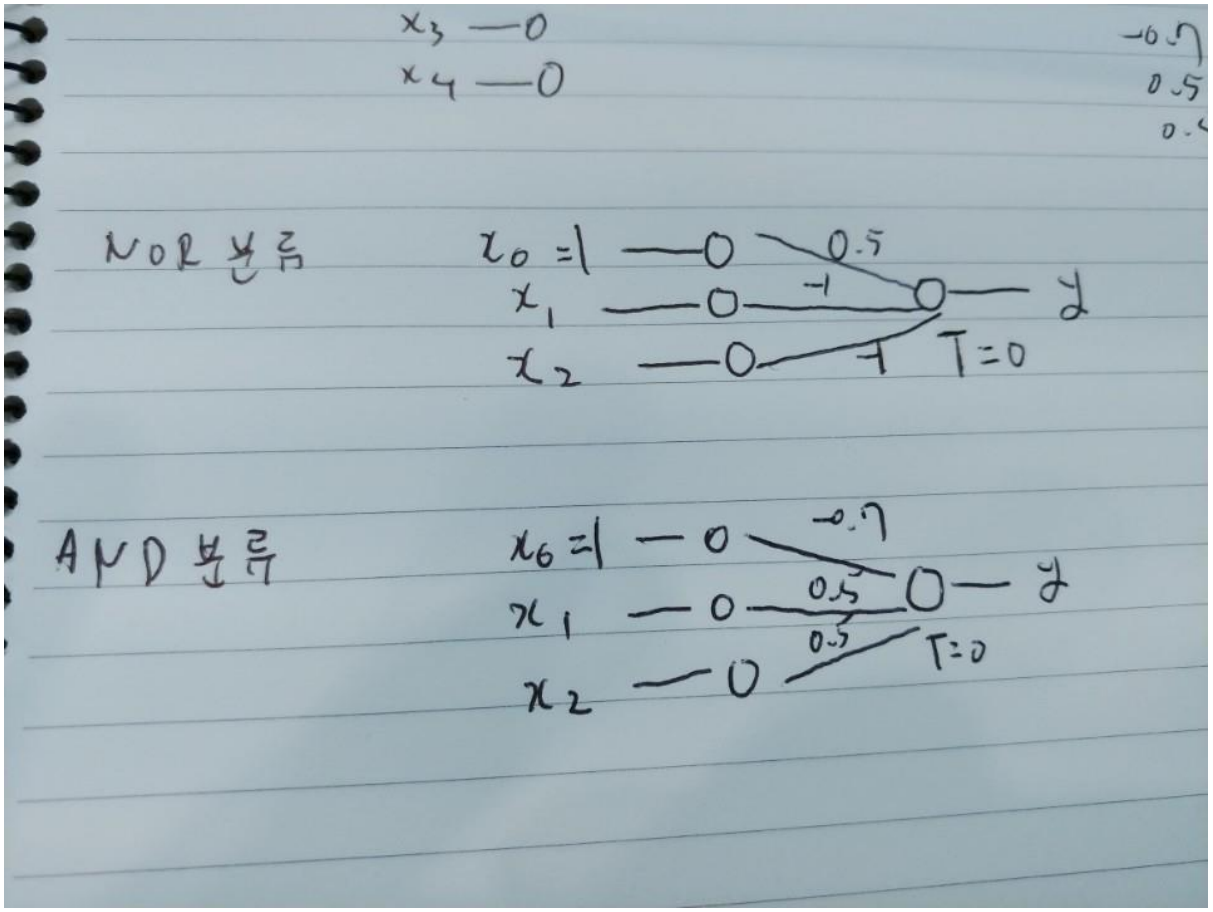
```
plt.plot(range(1,len(train_accuracy)+1),train_accuracy,'bo',label = 'train accuracy')
plt.plot(range(1,len(val_accuracy)+1),val_accuracy,'r',label = 'val accuracy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d903cc2ac8>
```



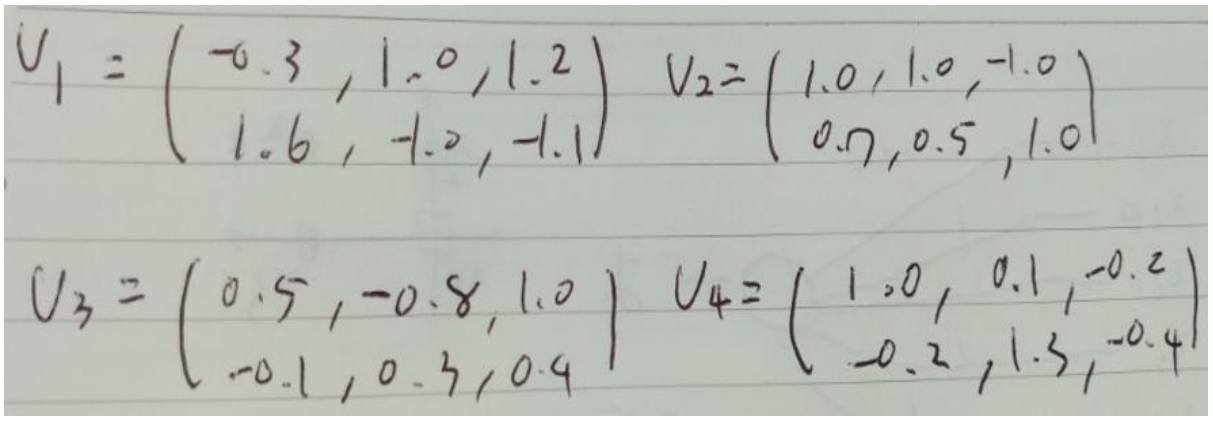
Epoch 수행시의 정확도를 나타낸 그래프이다. 처음 학습부터 피팅이 잘되어 있어서 우수한 적중율을 보인다. 나아가 학습을 통하여 정확도를 더 높여주었고, 이에 따라 테스트에서 더 우수한 적중률을 보여준다.

10.



11.

1) 가중치 행렬



2) 로지스틱 시그모이드 함수 사용

```
def logit(z):
    return 1 / (1+np.exp(-z)) # 로지스틱 함수 의 정의

def relu(z):
    return np.maximum(0,z) #relu 함수의 정의

def derivative(f, z, eps=0.000001):
    return (f(z+eps)-f(z-eps))/(2*eps) #

z = np.array([1.0,1.0,0])
x = np.array([],float)
u1 = np.array([[[-0.3,1.0,1.2],[1.6,-1.0,-1.1]],[[1.0,1.0,-1.0],[0.7,0.5,1.0]],[[0.5,-0.8,1.0],[-0.1,0.3,0.4]],[[1.0,0.1,-0.2],
#로지스틱 시그모이드 함수를 활성함수로 사용
for k in u1:
    for t in k:
        temp = np.dot(z,t.T)
        x = np.append(x,logit(temp))
    z = []
    z = np.append(1.0,x)
    x = np.array([],float)
print(z[1],z[2])
```

```
0.7202129099819784 0.6080707687639114
```

3) Relu 함수 사용

```
for k in u1:
    for t in k:
        temp = np.dot(z,t.T)
        x = np.append(x,relu(temp))
    z = []
    z = np.append(1.0,x)
    x = np.array([],float)
print(z[1],z[2])
```

```
0.949 1.095
```

---

4) weight 가 변했을 때

1)로지스틱 시그모이드 함수

```
# 로지스틱 시그 모이드 함수를 활성함수로 사용
u1 = np.array([[[-0.3,1.0,1.2],[1.6,-1.0,-1.1]],[[1.0,1.0,-1.0],[0.7,0.5,1.0]],[[0.5,-0.8,1.0],[-0.1,0.3,0.4]],[[1.0,0.1,-0.2],[-
z = np.array([1.0,1.0,0])
for k in u1:
    for t in k:
        temp = np.dot(z,t.T)
        x = np.append(x,logit(temp))
    z = []
    z = np.append(1.0,x)
    x = np.array([],float)
o1_pred = (0-z[1])*(0-z[1])
o2_pred = (1-z[2])*(1-z[2])
loss_1 = o1_pred + o2_pred

print(loss_1)
```

```
0.6723151580020207
```

```
#로지스틱 시그 모이드 함수를 활성함수로 사용
#가중치  u3의 12를 0.9로변경
u1 = np.array([[[-0.3,1.0,1.2],[1.6,-1.0,-1.1]],[[1.0,1.0,-1.0],[0.7,0.5,1.0]],[[0.5,-0.8,0.9],[-0.1,0.3,0.4]],[[1.0,0.1,-0
z = np.array([1.0,1.0,0])
for k in u1:
    for t in k:
        temp = np.dot(z,t.T)
        x = np.append(x,logit(temp))
    z = []
    z = np.append(1.0,x)
    x = np.array([],float)

o1_pred = (0-z[1])*(0-z[1])
o2_pred = (1-z[2])*(1-z[2])
loss_1 = o1_pred + o2_pred


print(loss_1)
```

```
0.6763389205358228
```

MSE를 이용하여 손실을 구하였다. 변경 전에는 0.6723151580020207 이고, 변경 후에는 0.6763389205358228 이다. 가중치가 작아졌을 때, 손실이 소폭 작아졌다.

2) Relu 함수를 이용

```
#Relu 함수를 활성함수로 사용
u1 = np.array([[[-0.3,1.0,1.2],[1.6,-1.0,-1.1]],[[1.0,1.0,-1.0],[0.7,0.5,1.0]],[[0.5,-0.8,1.0],[-0.1,0.3,0.4]],[[1.0,0.1,-0.2],[
z = np.array([1.0,1.0,0])
for k in u1:
    for t in k:
        temp = np.dot(z,t.T)
        x = np.append(x,relu(temp))
    z = []
    z = np.append(1.0,x)
    x = np.array([],float)
o1_pred = (0-z[1])*(0-z[1])
o2_pred = (1-z[2])*(1-z[2])
loss_1 = o1_pred + o2_pred

print(loss_1)
```

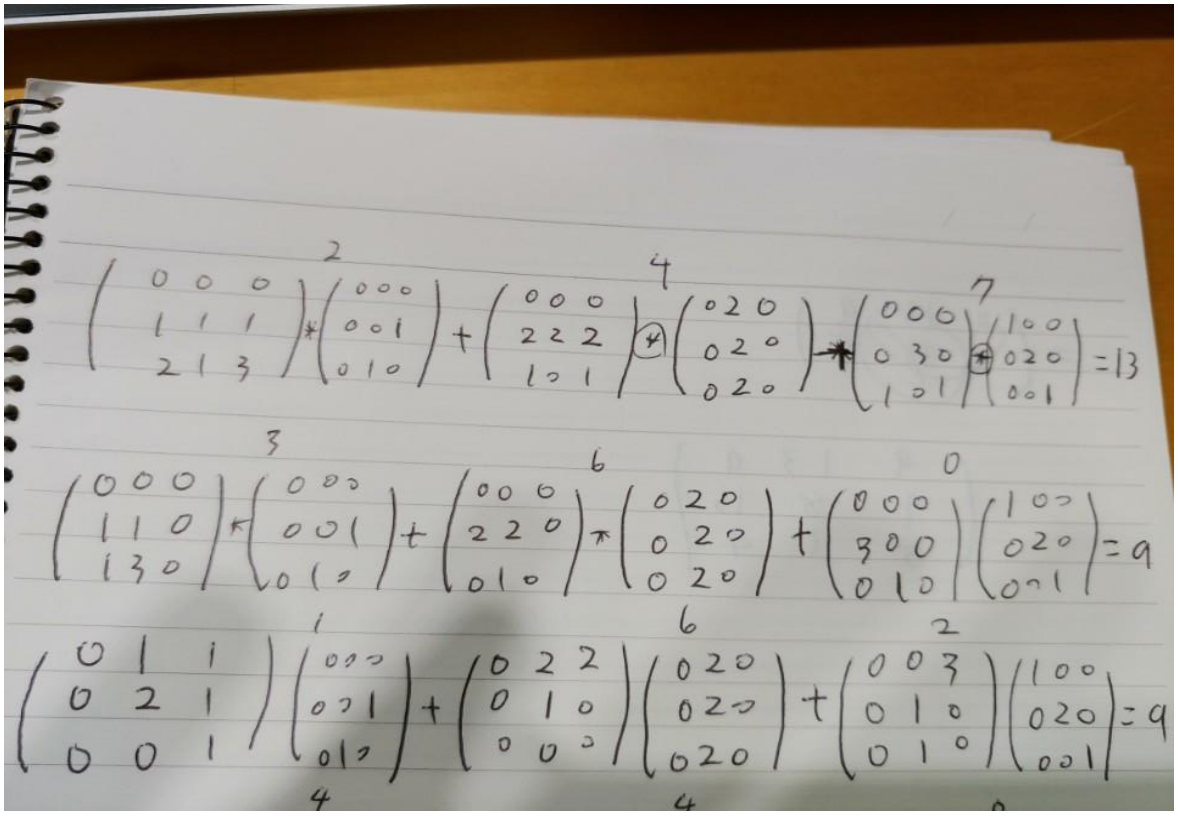
```
0.9096259999999998
```

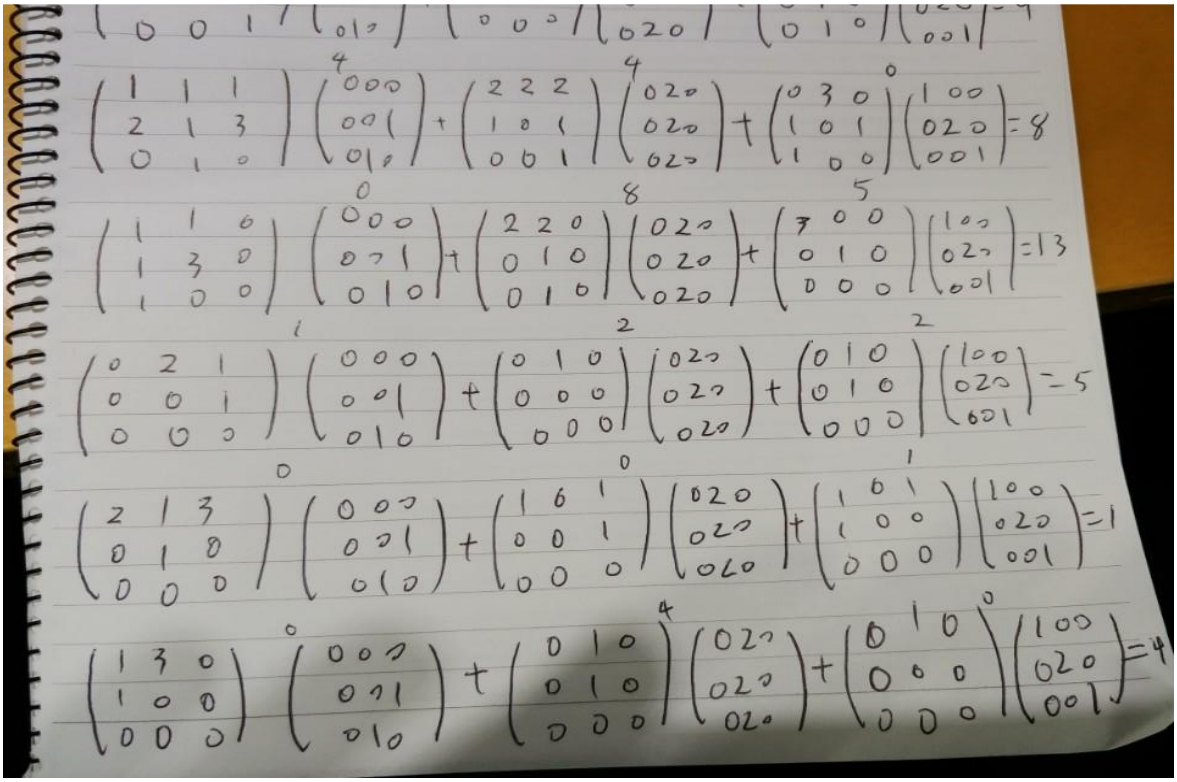
```
#Relu 함수를 활성함수로 사용
#가중치  u3의 12를 0.9로변경
u1 = np.array([[[-0.3,1.0,1.2],[1.6,-1.0,-1.1]],[[1.0,1.0,-1.0],[0.7,0.5,1.0]],[[0.5,-0.8,0.9],[-0.1,0.3,0.4]],[[1.0,0.1,-0.2],[-
z = np.array([1.0,1.0,0])
for k in u1:
    for t in k:
        temp = np.dot(z,t.T)
        x = np.append(x,relu(temp))
    z = []
    z = np.append(1.0,x)
    x = np.array([],float)

o1_pred = (0-z[1])*(0-z[1])
o2_pred = (1-z[2])*(1-z[2])
loss_1 = o1_pred + o2_pred


print(loss_1)
```

```
0.8838365
```

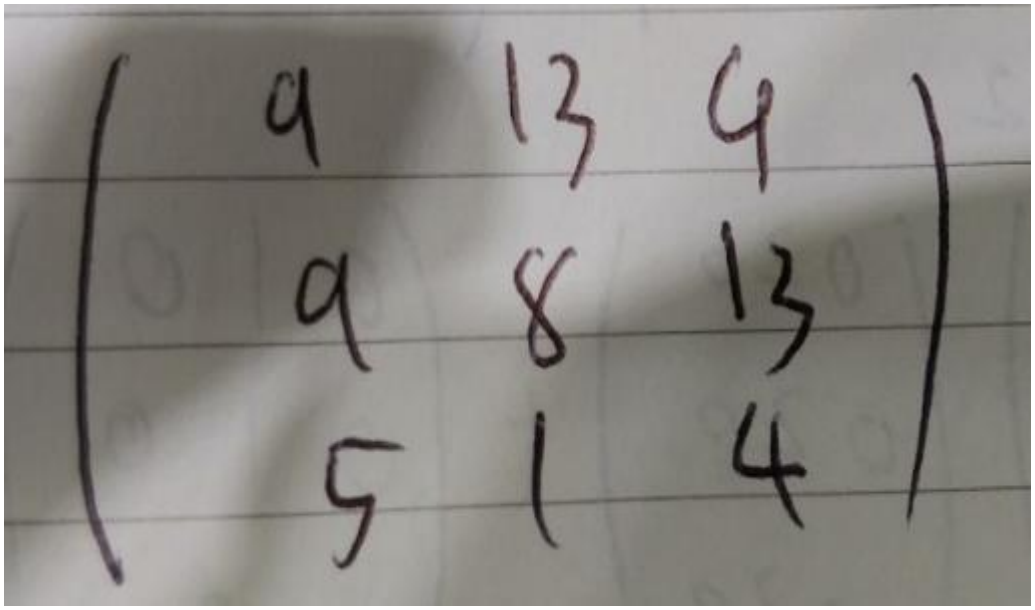
Relu를 적용했을 때도 동일하다. 손실 값이 `0.9096259999999998` 에서 `0.8838365`으로 작아진 것이 보인다. 따라서 가중치를 변경 한다면, 오류의 값에 영향을 줌을 알 수 있다. 이를 이용하여 `backward` 과정에서 `loss` 값을 이용하여 `weight`를 조정하는데 이는 타당함을 알 수 있다.

12. 계산과정은 다음과 같다.





결과는 다음과 같다.



13.

1) w2 = (32-5+2*2)/2+1

H2 = (32-5+2*2)/2+1

D2 = 10

출력의 크기 = 32*32*10=10,240

매개변수의 수 = (5*5*3)*10+10 = 760

2) w2 = (32-3+2)/1+1

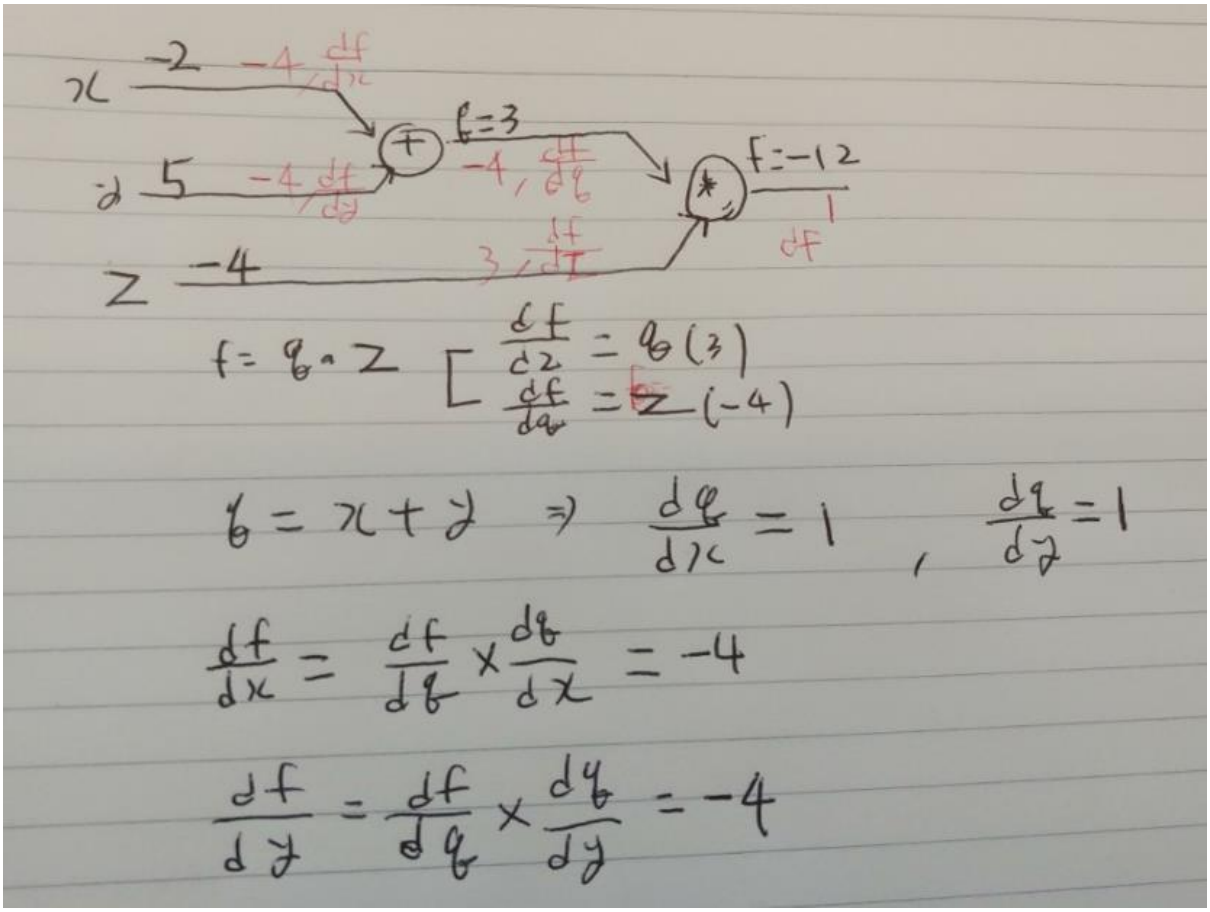
H2 = (32-3+2)/1+1

D2 = 64

출력의 크기 = 32*32*64 = 65,536

매개변수의 수 = (3*3*3)*64+64 = 1792

14.



$f = q \cdot z$ $\left[\frac{df}{dz} = q(3) \right.$
$\left. \frac{df}{dq} = z(-4) \right.$

$q = x + y \Rightarrow \frac{dq}{dx} = 1 \quad , \quad \frac{dq}{dy} = 1$

$$\frac{df}{dx} = \frac{df}{dq} \times \frac{dq}{dx} = -4$$

$$\frac{df}{dy} = \frac{df}{dq} \times \frac{dq}{dy} = -4$$