(1)

<코드>

```
def logit(z):
    return 1/(1 + np.exp(-z))
```

⇨ 로지스틱 시그모이드 함수

```
def relu(z):
    return np.maximum(0, z)
```

⇨ Relu 함수

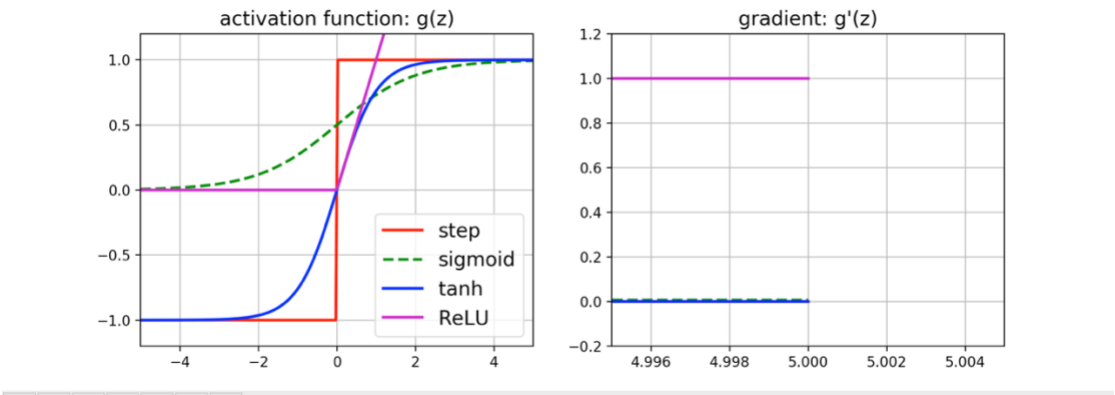
```
def derivative(f, z, eps = 0.000001):
    return (f(z + eps) - f(z - eps))/(2 * eps)
```

⇨ 도함수를 연산하는 함수

```
plt.plot(z, np.sign(z), "r-", linewidth = 2, label = "step")
plt.plot(z, np.tanh(z), "b-", linewidth = 2, label = "tanh")
```

⇨ 계단함수와 하이퍼볼릭 탄젠트함수는 numpy의 내장함수를 사용함

<출력 결과>



⇨ 계단함수

- 계단함수는 딱딱한 의사결정을 하며 영역을 점으로 변환한다.
- 함수값의 범위는 -1과 1 이다.

⇨ 로지스틱 시그모이드 함수

- 비선형 함수이다
- gradient vanishing 문제를 갖고있어 gradient 0이 곱해지기 때문에 그 다음 layer로 전파되지 않아 학습되지 않는 문제점을 갖는다.
- 지수함수로 연산이 복잡하다
- 함수값의 범위는 0부터 1 이다.

⇨ 하이퍼볼릭 탄젠트 함수

- 비선형함수이다
- 함수값의 범위는 -1부터 1 이며 중심값은 0이다.
- 시그모이드와 유사하며 시그모이드 함수를 이용하여 유도할 수 있다.
- 시그모이드와 마찬가지로 gradient vanishing 문제를 갖고있다.

⇨ 렐류 함수

- 비선형 함수이다.
- 양 극단값이 포화되지않는다(양수 지역은 선형적)
- 연산이 간단하다(최대값 연산 1개)
- 직관적으로 에러를 감지하기 쉽다.
- 함수값의 범위는 0 부터 무한대이며 중심값이 0이 아니다.

➔ 렐류를 제외하고는 z값이 충분히 커질때 도함수의 값이 0이 되어 입력값이 에러를 갖고있어도 0을 출력해 에러 전달에 문제점을 갖고있어 최근에는 렐류를 많이 사용한다

(2)

<코드>

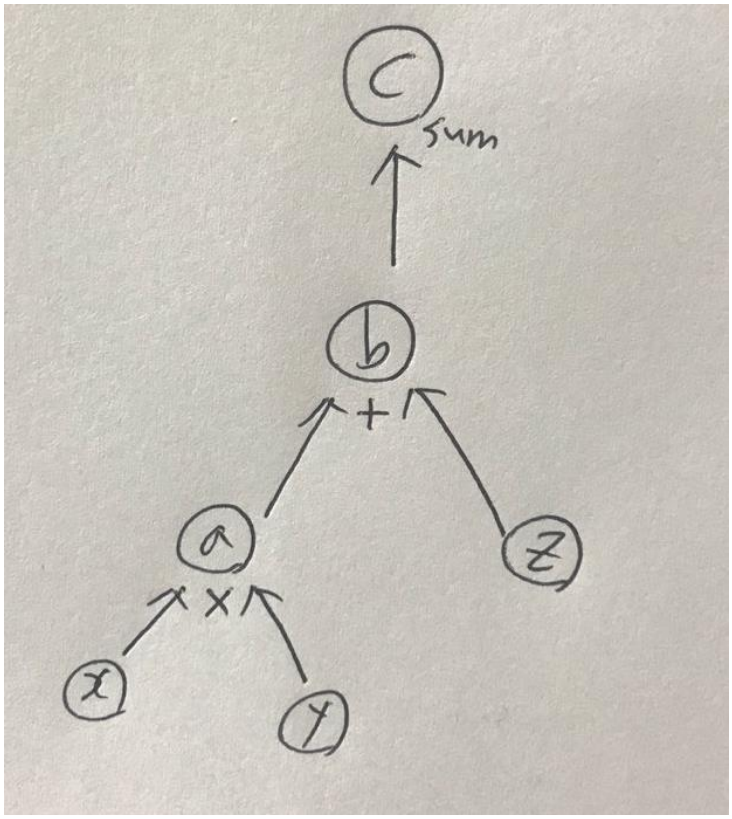
```python
# -*- coding: utf -8 -*-

import numpy as np

# 0 근처의 난수를 생성
np.random.seed(0)

N, D = 3, 4

# 평균이 0 이고 분산이 1인 3행 4열짜리 난수를 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)

# (1) 해당 연산망의 그래프 연산을 손으로 작성

# 행렬로 표현
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a*y
grad_y = grad_a*grad_x
```

```
43
44  import torch
45  x = torch.randn(N, D, requires_grad = True)
46  y = torch.randn(N, D, requires_grad = True)
47  z = torch.randn(N, D)
48
49  a = x * y
50  b = a + z
51  c = torch.sum(b)
52  c.backward()
```

⇨ 오류역전파를 직접 연산한 것과 torch의 backward함수를 이용한 두가지 방법으로 오류역전파를 구현한 코드이다

<출력 결과>

(1)



```
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
[[ 0.76103773  0.12167502  0.44386323  0.33367433]
 [ 1.49407907 -0.20515826  0.3130677  -0.85409574]
 [-2.55298982  0.6536186   0.8644362  -0.74216502]]
c.item
-1.91700839996
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
[[ 0.76103773  0.12167502  0.44386323  0.33367433]
 [ 1.49407907 -0.20515826  0.3130677  -0.85409574]
 [-2.55298982  0.6536186   0.8644362  -0.74216502]]
```

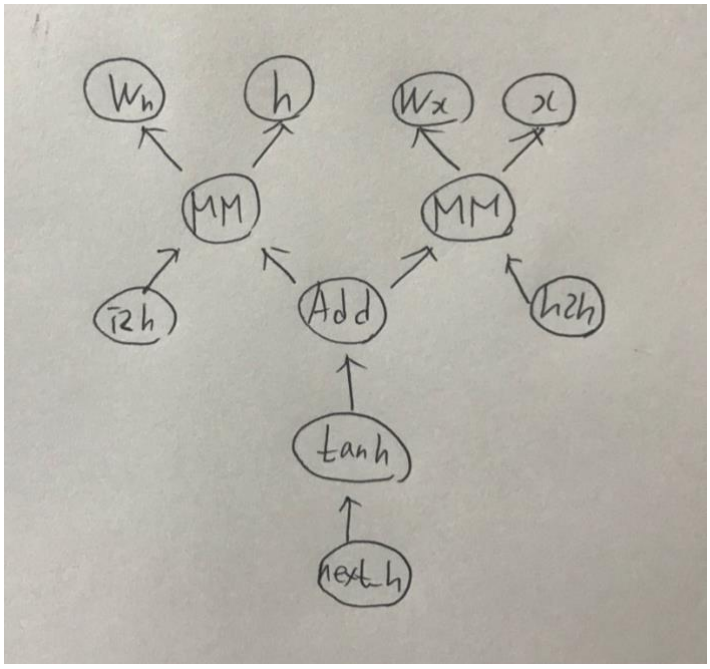
(3)

<코드>

```python
# -*- coding: utf -8 -*-

import torch

x = torch.randn(1, 10, requires_grad = True)
prev_h = torch.randn(1, 20, requires_grad = True)
w_h = torch.randn(20, 20, requires_grad = True)
w_x = torch.randn(20, 10, requires_grad = True)

i2h = torch.mm(w_x,x.t())
h2h = torch.mm(w_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

# (1) 해당 신경망의 그래프 연산을 손으로 작성

loss = next_h.sum()
loss.backward()

# (2) loss 출력 확인

print("loss :"),
print(loss.item())
```

<출력 결과>

(1)



(2)

```
loss : 6.75950574875
```

(4)

<코드>

```python
# -*- coding: utf-8 -*-
import torch
import matplotlib.pyplot as plt

#N은 배치크기, D_in 은 입력의 차원
#H는 은닉 계층의 차원, D_out 은 출력 차원
N, D_in, H, D_out = 64, 1000, 100, 10

# 무작위로 입력과 출력 데이터를 생성한다
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 무작위로 가중치를 초기화한다.
w1 = torch.randn(D_in, H)
w2 = torch.randn(H, D_out)

learning_rate = 1e-6

for t in range(500):
    # 순전파 단계 : 예측값 y를 계산한다
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # 손실을 계산하고 출력한다
    loss = (y_pred - y).pow(2).sum()

    plt.plot(t, loss, "b.")
    plt.xlabel("$t$",fontsize = 18)
    plt.ylabel("$loss$",fontsize = 18)

    print(t, loss)

    # 손실에 따른 w1, w2의 변화도를 계산하고 역전파 한다.
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    # 경사하강법을 사용하여 가중치를 갱신한다.
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
plt.show()
# (1) 매 t마다 y_pred에 따른 loss(accuracy) 변화를 화면 출력 확인 (plot)

# (2) 해당 학습이 적절히 진행되고 있는지 서술
```

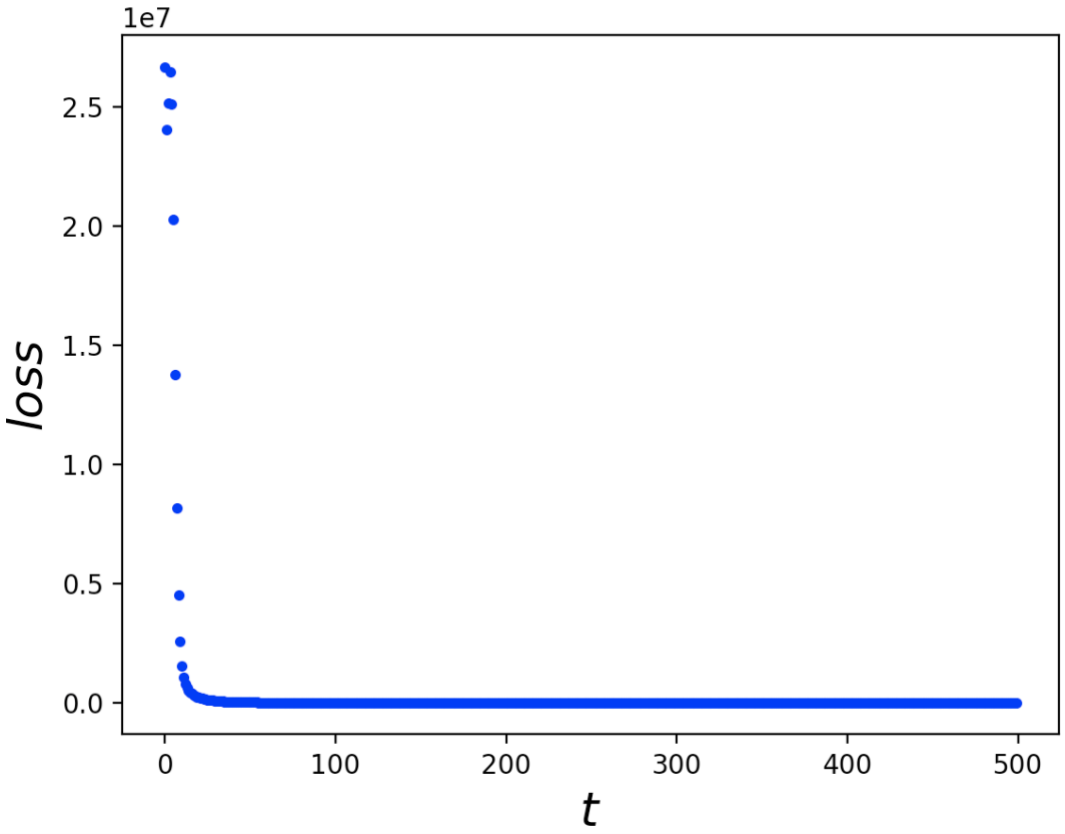
⇨ 코드 설명은 주석

⇨ ReLu신경망 학습 코드

⇨ 신경망의 순전파단계와 역전파 단계를 수동으로 구현

⇨ 수동으로 이루어졌기 때문에 역전파 단계를 위해 연산의 중간값들을 별도로 저장함

<출력 결과>



⇨ 이 신경망은 하나의 은닉 계층을 갖고 있으며, 신경망의 출력과 정답 사이의 유클리드 거리를 최소화하는 식으로 경사하강법을 사용하여 무작위의 데이터를 맞추도록 학습한다.

⇨ Loss 값이 거의 0에 수렴하므로 올바르게 학습되었다.

⇨ (학습률을 1e-6으로 고쳤음)

(5)

<코드>

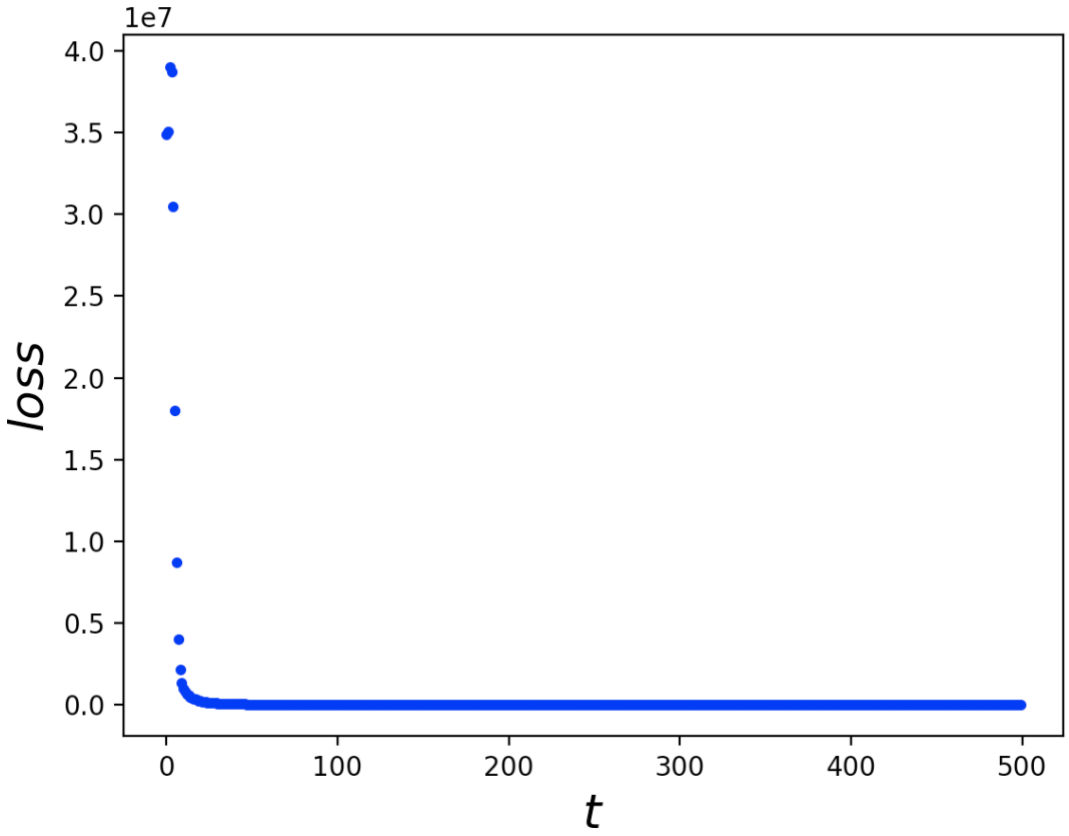
```python
# -*- coding: utf -8 -*-

import torch
import matplotlib.pyplot as plt

# N은 배치 크기이며, D_in은 입력의 차원이다
# H는 은닉 계층의 차원이며, D_out은 출력 차원이다
N, D_in, H, D_out = 64, 1000, 100, 10

# 입력과 출력을 저장하기 위해 무작위 값을 갖는 tensor를 생성한다.
# requires_grades=False로  설정하여 역전파 중에 이 값들에 대한 변화도를
# 계산할 필요가 없음을 나타낸다
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# requires_grades=True로 설정하여 역전파 중에 이 값들에 대한
# 변화도를 계산할 필요가 있음을 나타낸다.
w1 = torch.randn(D_in, H, requires_grad =True)
w2 = torch.randn(H, D_out, requires_grad =True)

# 원래 코드 학습률 (10e-6)을 1e-6으로 수정함
learning_rate = 1e-6
for t in range(500):
    # 순정파 단계 : y값을 예측한다
    # 이는 tensor를 사용한 순전파 단계와 완전히 동일하지만 역전파 단계를
    # 별도로 구현하지 않기위해 중간값들에 대한 참조를 갖고 있을 필요가 없다.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # 손실을 계산한다.
    loss = (y_pred - y).pow(2).sum()

    plt.plot(t, loss.data[0], "b.")
    plt.xlabel("$t$",fontsize = 18)
    plt.ylabel("$loss$",fontsize = 18)

    # autograde를 사용하여 역전파 단계를 계산한다. 이것은 requires_grad = True를
    # 갖는 모든 값들에 대한 손실의 변화도를 계산한다. 이후 w1.grad와 w2.grad는
    # w1과 w2 각각에 대한 손실의 변화도를 갖게된다.
    loss.backward()

    # 경사하강법
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
plt.show()

# (1) 매 t마다 y_pred에 따른 loss(accuracy) 변화를 화면 출력 확인 (plot)

# (2) 앞 문제의 코드와 비교

```

⇨ 위의 예제코드에서 수동으로 행해주었던 순전파단계와 역전파 단계를 자동화한 코드이다.

⇨ 신경망의 순전파 단계는 연산그래프를 정의하며 그래프의 노드는 tensor 이고 엣지는 입력 tensor로부터 출력 tensor를 만들어내는 함수이다.

⇨ 이 연산그래프를 통해 역전파를 하게되면 변화도를 쉽게 계산할 수 있다.

<출력 결과>



⇨ 앞의 코드와 같은 결과를 출력한다.

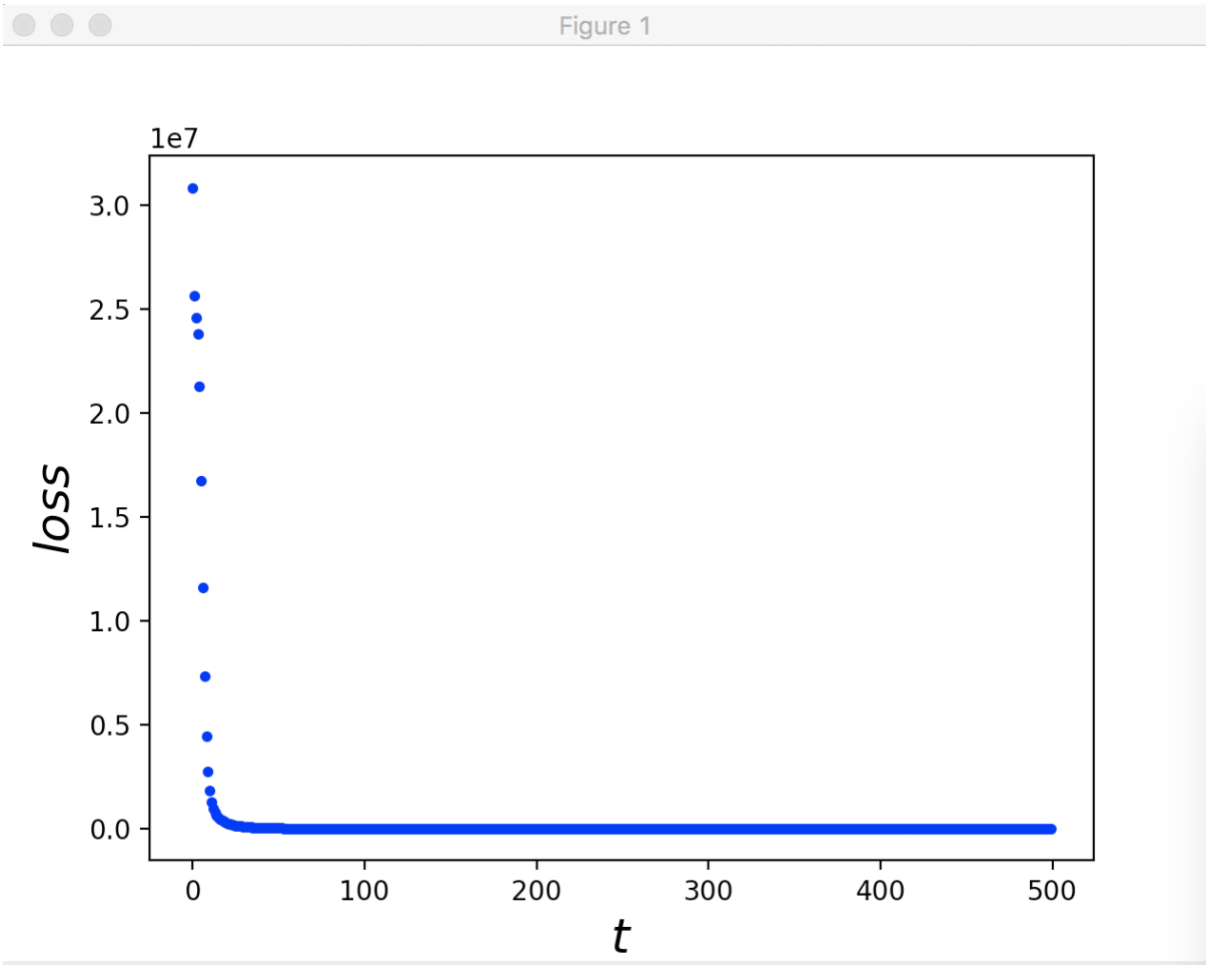
(6)

<코드>

```python
# -*- coding: utf-8 -*-

import torch
import matplotlib.pyplot as plt

class MyReLU(torch.autograd.Function):
    @staticmethod
    #torch.autograd.Function를 상속받아 사용자 정의 autograd
    #tensor 연산을 하는 순전파와 역전파 단계를 구현함.
    def forward(ctx, x):
        # 순전파 단계에서는 입력을 갖는 tensor를 받아 출력 tensor를 반환해야한다.
        # ctx는 역전파 연산을 위한 정보를 저장하기 위해 사용하는 context object
        # ctx.save_for_backward method를 사용하여 역전파 단계에서 사용할 어떠한
        # 객체도 저장해 둘 수 있다.
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        # 역전파 단계에서는 출력에 대한 손실의 변화도를 갖는 tensor를 받고,
        # 입력에대한 손실의 변화도를 계산한다
        x, = ctx.saved_tensors
        grad_input =grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)

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

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6

for t in range(500):
    y_pred = my_relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    plt.plot(t, loss.item(), "b.")
    plt.xlabel("$t$",fontsize = 18)
    plt.ylabel("$loss$",fontsize = 18)

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        # 가중치 갱신 후 수동으로 변화도를 0으로 만들어준다
        w1.grad.zero_()
        w2.grad.zero_()
plt.show()

# (1) 매 t마다 y_pred에 따른 loss(accyracy) 변화를 화면 출력 확인

# (2) 앞 문제의 코드와 비교

```

⇨ Torch.autograd.Function의 서브클래스를 정의하고 forward와 backward함수를 구현해 사용자 정의 autograd 연산자를 정의한 코드이다.

⇨ <출력 결과>



(7)

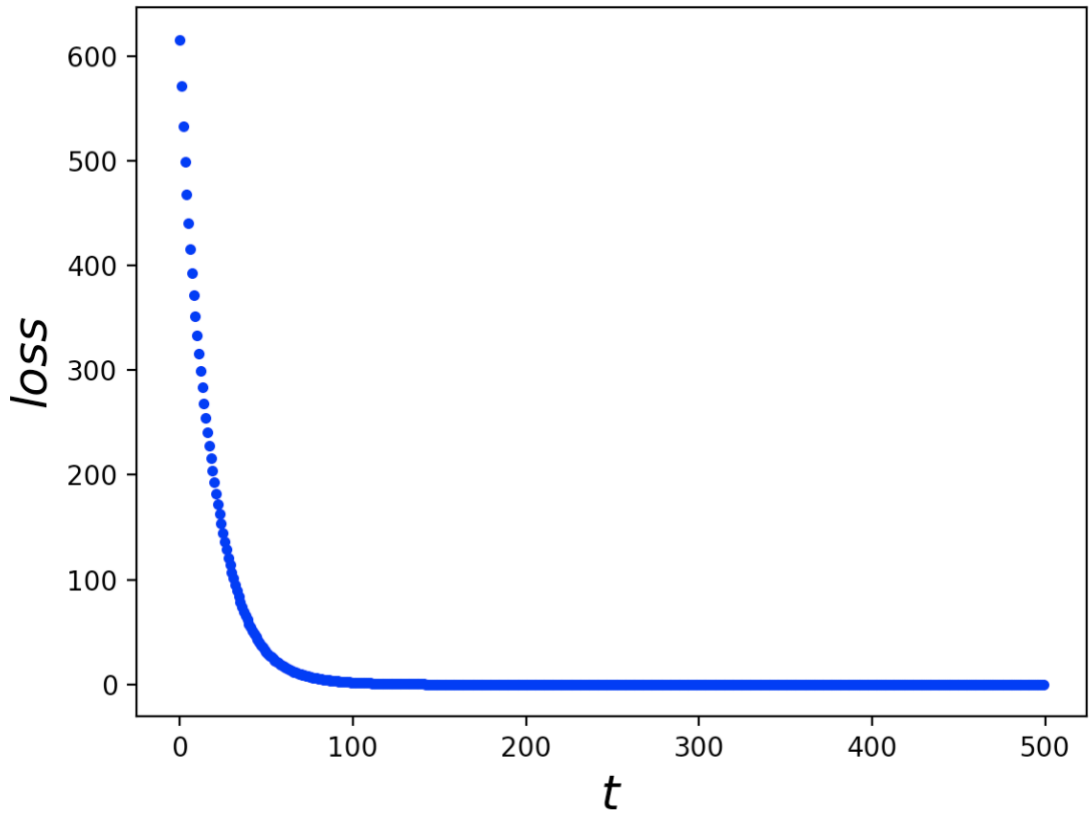
<코드>

```
import torch
import matplotlib.pyplot as plt

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        # 생성자에서 2개의 nn.linear모듈을 생성하고 멤버변수로 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):
        # 순전파 함수에서는 입렵데이터를 받아서 출력데이터를 반환해야한다.
        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

x = torch.randn(N, D_in)
y = torch.randn(N, D_out, requires_grad = False)

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
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

for t in range(500):
    # 순전파 단계 : 모델에 x를 전달하여 예상하는 y값을 계산한다.
    y_pred = model(x)

    # 손실을 계산한다.
    loss = criterion(y_pred, y)
    plt.plot(t, loss.data[0], "b.")
    plt.xlabel("$t$",fontsize = 18)
    plt.ylabel("$loss$",fontsize = 18)

    # 변화도를 0으로 만들고, 역전파 단계를 수행하고, 가중치를 갱신한다.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
plt.show()
```

⇨ 코드설명 주석

⇨ 최적화 알고리즘의 아이디어를 추상화하고 일반적으로 사용하는 최적화 알고리즘의 구현체를 제공하는 Optim 패키지 사용

<출력 결과>



⇨ 앞의 코드에서는 복잡한 연산자를 정의하고 도함수를 자동으로 계산하는데 매우 강력했다. 그러나 규모가 큰 신경망에서는 그 자체만으로는 너무 낮은 수준일 수 있다.

⇨ 이 코드는 신경망을 2-계층 신경망으로 구성하였다.

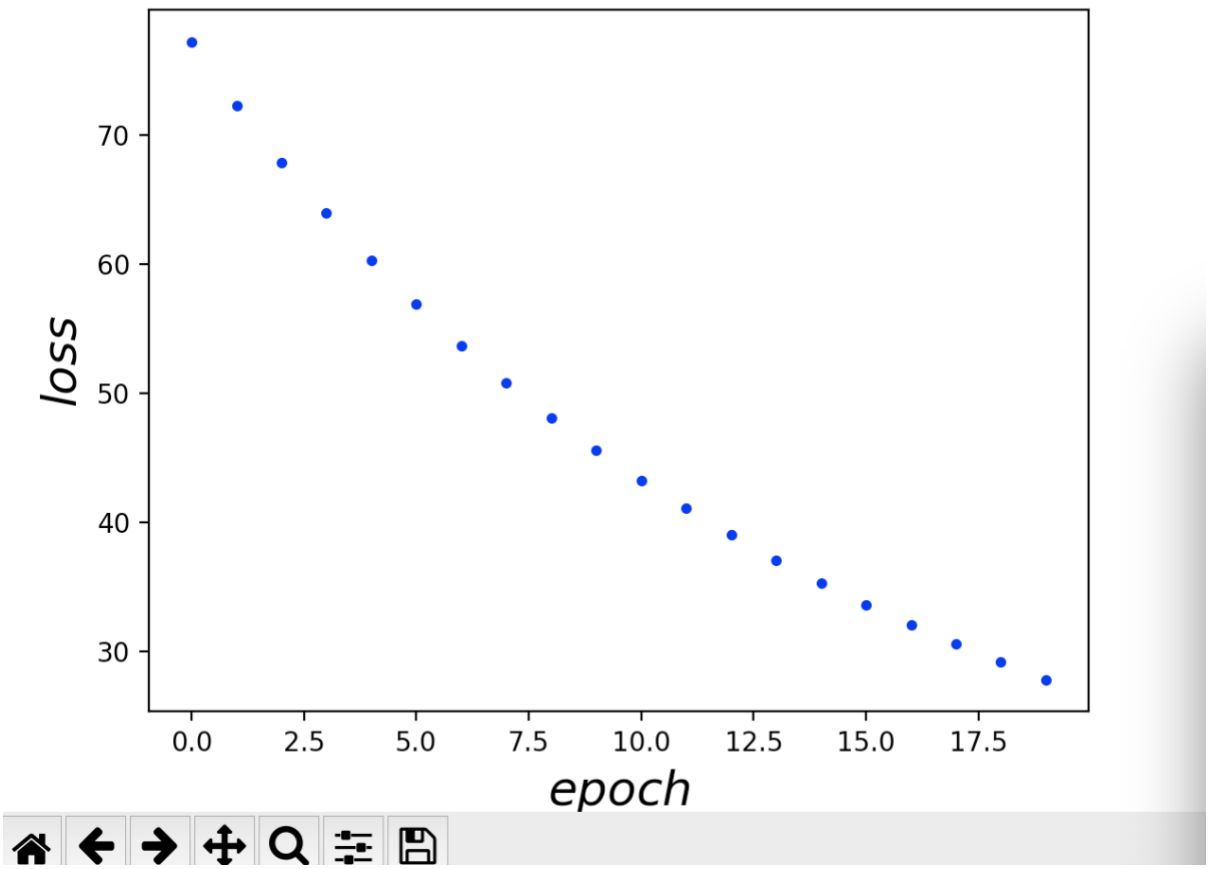
(8)

<코드>

```
import torch
import matplotlib.pyplot as plt
from torch.utils.data import TensorDataset, DataLoader

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        # 생성자에서 2개의 nn.linear모듈을 생성하고 멤버변수로 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):
        # 순전파 함수에서는 입력데이터를 받아서 출력데이터를 반환해야한다.
        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

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size = 8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
criterion = torch.nn.MSELoss(size_average=False)

for epoch in range(20):
    for x_batch,y_batch in loader:
        y_pred = model(x_batch)
        loss = criterion(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    plt.plot(epoch, loss.item(), "b.")
    plt.xlabel("$epoch$",fontsize = 18)
    plt.ylabel("$loss$",fontsize = 18)

plt.show()
```

⇨ 이전 코드들은 스토캐스틱법을 사용하였고 이 코드는 데이터 전처리 후 미니배치법을 사용한 코드이다.

<출력 결과>



(9)

<코드>

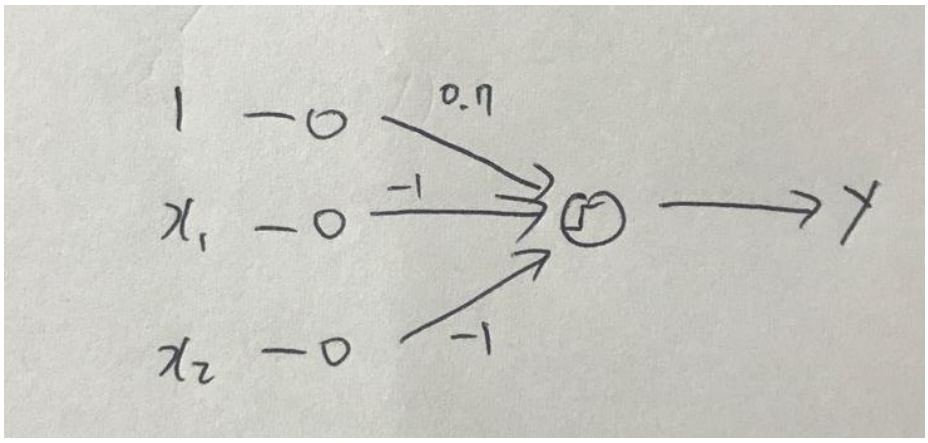
소스코드 제출

(컴파일이 안되요 .. data가 다운로드 되다가 멈춤)

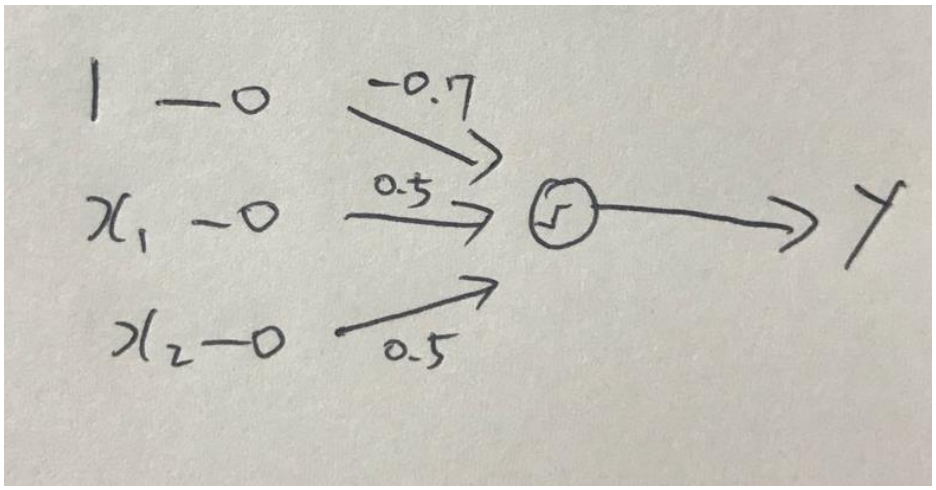
<출력 결과>

(10)

-NOR



-AND



(11)

1) $U^1 = \begin{bmatrix} 0 & 0 & 0 \\ -0.3 & 1.0 & 1.2 \\ 1.6 & -1.0 & -1.1 \end{bmatrix}$ $U^2 = \begin{bmatrix} 0 & 0 & 0 \\ 1.0 & 1.0 & -1.0 \\ 0.7 & 0.5 & 1.0 \end{bmatrix}$

$U^3 = \begin{bmatrix} 0 & 0 & 0 \\ 0.5 & -0.8 & 1.0 \\ -0.1 & 0.3 & 0.4 \end{bmatrix}$ $U^4 = \begin{bmatrix} 1.0 & 0.1 & -0.2 \\ -0.2 & 1.3 & -0.4 \end{bmatrix}$

(numpy를 이용함 – 코드제출)

2) o_1 = 0.60844973 o_2= 0.61685917

3) o_1 = 0.005 o_2= 0.967

4) u^3_{12}가중치의 경우 출력 o_1 과 o_2에 모두 영향을 미치기 때문에 오류를 증가시킨다.

(12)

⇨

$$\begin{bmatrix} 9 & 14 & 9 \\ 9 & 8 & 13 \\ 5 & 1 & 9 \end{bmatrix}$$

(13)

1) 출력의 크기 : ((32 – 5 + 2*2) / 1 +1)*((32 – 5 + 2*2) / 1 +1)*(10) = 10240
   매개변수의 수 : (5 * 5 * 3) * 10 + 10 = 760
2) 출력의 크기 : ((32 – 3 + 2*1) / 1 +1)*((32 – 3 + 2*1) / 1 +1)*(64) = 65536
   매개변수의 수 : (3 * 3* 3) * 64+ 64 = 110592

(14)