인공신경망의 Back-propagation 을 해봅시다

이번 과제의 목표는 Back-propagation 을 계산하고 이를 torch의 자동 미분과 비교하여 맞는지 확인하는 것입니다.

1) 필수 라이브러리와 함수를 불러 옵니다.

In [1]:
```
import torch
import numpy as np
```

In [2]:
```
def ReLU_func(outputs):
    zero_tensor = torch.zeros(outputs.size())
    final_outputs = torch.maximum(outputs,zero_tensor)

    return final_outputs

def softmax(outputs):
    numerator = torch.exp(outputs - torch.max(outputs,axis=1)[0].view(-1,1))
    denominator = torch.sum(numerator, axis=1).view(-1,1)
    softmax = numerator/denominator

    return softmax

def cross_entropy(outputs, labels):
    return torch.sum(-1*labels*torch.log(outputs),axis=1)
```

2) 인공신경망 계산을 합니다.

입력은 [4,1], Label은 [0,1], W_0 은 $\begin{pmatrix} 1, & -2 \\ 2, & 5 \end{pmatrix}$, W_1은 $\begin{pmatrix} 3, & -3 \\ -1, & 1 \end{pmatrix}$ 로 주어졌을 때

각 단계를 h, L, O, s, l 를 각각 역치 전 값, 히든 레이어 값, 출력값, 소프트맥스 후, loss로 하여 계산합니다. 각 값은 torch.tensor 클래스의 객체로 만들어 자동미분이 가능하게 만드세요.

In [3]:
```
I = torch.Tensor([4,1])
label = torch.Tensor([0,1])
W_0 = torch.Tensor([[1,-2],
                    [2,5]])
W_1 = torch.Tensor([[3,-3],
                    [-1,1]])
W_0.requires_grad_(True)
W_1.requires_grad_(True)
```

Out[3]:
```
tensor([[ 3., -3.],
        [-1.,  1.]], requires_grad=True)
```

In [4]:
```
h =  I.matmul(W_0)
L =  ReLU_func(h)
O =  L.matmul(W_1)
s =  softmax(O.reshape(1,2))
l =  cross_entropy(s,label)
l.requires_grad_(True)
```

Out[4]:
```
tensor([36.], grad_fn=<SumBackward1>)
```

In [5]:
```
print(l)
```

```
tensor([36.], grad_fn=<SumBackward1>)
```

In [6]:
```
l.backward()
```

3) $\nabla_{W_0} l$ 계산하기

위에서 주어진 h, L, O, s, l을 이용하여 $\nabla_{W_0} l$를 계산해봅시다.

numpy를 이용하여 계산하세요!

참고로

$$\nabla_s l = \left(-\frac{label_0}{s_0}, -\frac{label_1}{s_1} \right)$$

$$\nabla_o s = \begin{pmatrix} s_0(1-s_0), & -s_0 s_1 \\ -s_1 s_0, & s_1(1-s_1) \end{pmatrix}$$

$$\nabla_{w_1} o = \begin{pmatrix} L_0, 0, L_1, 0 \\ 0, L_0, 0, L_1 \end{pmatrix}$$

In [7]:
```
#numpy 계산시에는 W_0, W_1을 requires_grad 를 False로 변경하고 실행할 것
with torch.no_grad():
    s_l= np.array([-label[0]/s[0][0],-label[1]/s[0][1]])
    o_s= np.array([[s[0][0]*(1-s[0][0]),-s[0][0]*s[0][1]],
                   [-s[0][1]*s[0][0],s[0][1]*(1-s[0][1])]])
    w1_o= np.array([[L[0],0,L[1],0],
                    [0,L[0],0,L[1]]])
```

In [8]:
```
w1_l = s_l@o_s@w1_o
print(w1_l)
```

```
[ 6. -6.  0.  0.]
```

자동 미분 결과와 비교해 봅시다.

In [9]:
```
print(W_1.grad)
```

```
tensor([[ 6., -6.],
        [ 0., -0.]])
```

자동 미분 결과와 비교해보고 차이점이 무엇인지 그리고 왜 그런 결과가 생겼는지 서술하시오.

정답) numpy로 계산한 결과는 벡터와 자동미분한 결과는 매트릭스로 결과가 나왔다. numpy로 계산한 결과는 w1_o의 정의에 따라 벡터로 출력되며, 자동 미분은 입력의 차원으로 계산되기 때문이다.

4) $\nabla_{w_0} l$ 계산하기

위에서 주어진 h, L, O, s, l을 이용하여 $\nabla_{W_0} l$를 계산해봅시다.

numpy를 이용하여 계산하세요!

참고로

$$\nabla_L o = W_1^T$$

$$\nabla_h L = \begin{pmatrix} 1 or 0, & 0 \\ 0, & 1 or 0 \end{pmatrix}$$

$$\nabla_{w_0} h = \begin{pmatrix} I_0, 0, I_1, 0 \\ 0, I_0, 0, I_1 \end{pmatrix}$$

In [10]:
```
L_o = np.array([[3,-1],
                [-3,1]])
h_L = np.array([[1,0],
                [0,0]])
w0_h = np.array([[4,0,1,0],
                 [0,4,0,1]])
```

In [11]:
```
w0_l = s_l@o_s@L_o@h_L@w0_h
print(w0_l)
```

```
[24.  0.  6.  0.]
```

자동미분과 비교해봅시다.

In [12]:
```
W_0.grad # 채워넣으시오
```

Out[12]:
```
tensor([[24.,  0.],
        [ 6.,  0.]])
```

In [ ]:
```

```