Homework#4

2020314482 인공지능융합전공 황선우

0. Part0

학습데이터 $(x,t)=(1,1)$

$w_1^0=w_2^0=w_3^0=w_4^0=w_5^0=1$

$\eta=0.1$

			net_{ni}	h_{ni}	net_{nj}		h_{nj}		net_{nk}	h_{nk}
n	x_{n1}	t_{n1}	net_{n1}	$h_{n1}=h_1$	net_{n1}	net_{n2}	$h_{n1}=h_2$	$h_{n2}=h_3$	net_{n1}	$h_{n1}=0$
1	1	1	1	0.731	0.731	0.731	0.675	0.675	1.350	0.794

$$\frac{\partial E}{\partial w_4^0}=\frac{\partial E}{\partial net_{nk}}\cdot\frac{\partial net_{nk}}{\partial w_4^0}=-(1-0.794)0.794(1-0.794)\cdot 0.675=-0.023$$

$$\frac{\partial E}{\partial w_5^0}=\frac{\partial E}{\partial net_{nk}}\cdot\frac{\partial net_{nk}}{\partial w_5^0}=-(1-0.794)0.794(1-0.794)\cdot 0.675=-0.023$$

$$\frac{\partial E}{\partial w_2^0}=\frac{\partial E}{\partial net_{nj}}\cdot\frac{\partial net_{nj}}{\partial w_2^0}=[-0.034\cdot 1)0.675(1-0.675)\cdot 0.731=-0.005$$

$$\frac{\partial E}{\partial w_3^0}=\frac{\partial E}{\partial net_{nj}}\cdot\frac{\partial net_{nj}}{\partial w_3^0}=(-0.034\cdot 1)0.675(1-0.675)\cdot 0.731=-0.005$$

$$\frac{\partial E}{\partial w_i^0}=\frac{\partial E}{\partial net_{ni}}\cdot\frac{\partial net_{ni}}{\partial w_i^0}=(-0.007\cdot 1-0.007\cdot 1)1(1-0.731)\cdot 1=-0.004$$

$$w_1^1=w_1^0-0.1\times\frac{\partial E}{\partial w_1^0}=1-0.1\times(-0.004)=1.0004$$

$$w_2^1=w_2^0-0.1\times\frac{\partial E}{\partial w_2^0}=1-0.1\times(-0.005)=1.0005$$

$$w_3^1=w_3^0-0.1\times\frac{\partial E}{\partial w_3^0}=1-0.1\times(-0.005)=1.0005$$

$$w_4^1=w_4^0-0.1\times\frac{\partial E}{\partial w_4^0}=1-0.1\times(-0.023)=1.0023$$

$$w_5^1=w_5^0-0.1\times\frac{\partial E}{\partial w_5^0}=1-0.1\times(-0.023)=1.0023$$

1. Part1 – Python

1-1. Python: List

```
list1 = [1, 2, 3]

#인덱스가 음수일 경우 리스트의 끝에서부터 셈
type(list1)
```

```
list
```

```
#0번 인덱스와 제일 마지막 인덱
list1[0] = 3
list1
```

```
(1, 3)
```

```
#리스트는 자료형이 다른 요소도 저장 가능
list1[1] = "str1"
list1
```

```
[1, 'str1', 3]
```

```
#range Type출력
list2 = range(10)
type(list2)
```

```
range
```

```
#List Type으로 변환
list2 = list(list2)
type(list2)
```

```
list
```

```
list2
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1-2. Python: List

```
# 인덱스 2에서 4(제외)까지 슬라이싱
a = list2[2:4]
print(f"list2[2:4] 결과: {a} => 인덱스 2~3까지만 출력")

# 인덱스 2에서 끝까지 슬라이싱
a = list2[2:]
print(f"list2[2:] 결과: {a} => 인덱스 2~끝까지 출력")

# 처음부터 인덱스 2(제외)까지 슬라이싱
a = list2[:2]
print(f"list2[:2] 결과: {a} => 인덱스 시작~1까지만 출력")

# 전체 리스트 슬라이싱
a = list2[:]
print(f"list2[:] 결과: {a} => 리스트 전체 출력")

# 2씩 증가하면서 리스트 출력
a = list2[::2]
print(f"list2[::2] 결과: {a} => 인덱스 0부터 2씩 증가하면서 리스트 출력")

# 슬라이싱 인덱스는 음수도 가능
a = list2[:-1]
print(f"list2[:-1] 결과: {a} => 인덱스 시작~마지막에서 2번째까지 출력")

# 슬라이스된 리스트에 새로운 리스트 할당
list2[2:4] = [8, 9]
print(f"list2 결과: {list2} => 인덱스 2~3의 값이 [8,9]로 대체되어 출력")

# 결과를 예측해 보시오
list2[2:5] = [8, 9]
print(f"list2 결과: {list2} => 인덱스 2~4의 값이 [8,9]로 대체되어 출력")
```

```
list2[2:4] 결과: [2, 3] => 인덱스 2~3까지만 출력
list2[2:] 결과: [2, 3, 4, 5, 6, 7, 8, 9] => 인덱스 2~끝까지 출력
list2[:2] 결과: [0, 1] => 인덱스 시작~1까지만 출력
list2[:] 결과: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] => 리스트 전체 출력
list2[::2] 결과: [0, 2, 4, 6, 8] => 인덱스 0부터 2씩 증가하면서 리스트 출력
list2[:-1] 결과: [0, 1, 2, 3, 4, 5, 6, 7, 8] => 인덱스 시작~마지막에서 2번째까지 출력
list2 결과: [0, 1, 8, 9, 4, 5, 6, 7, 8, 9] => 인덱스 2~3의 값이 [8,9]로 대체되어 출력
list2 결과: [0, 1, 8, 9, 5, 6, 7, 8, 9] => 인덱스 2~4의 값이 [8,9]로 대체되어 출력
```

1-3. Python: For Loop with List

```
# List는 순서를 갖기 때문에 for loop 시 index 0 부터 순서대로 접근
animals = ['cat', 'dog', 'monkey']
# 실행 결과를 확인하시오
for animal in animals:
  print(animal)
```

```
cat
dog
monkey
```

```
nums = [0, 1, 2, 3, 4]
squares = []

# 실행 결과를 확인하시오
for x in nums:
  squares.append(x ** 2)
  print(squares)
```

```
[0]
[0, 1]
[0, 1, 4]
[0, 1, 4, 9]
[0, 1, 4, 9, 16]
```

1-4. Python: Function

```
def sign(x):
  if x > 0: #x가 양수면 positive 반환
    return 'positive'
  elif x < 0: #x가 음수면 negative 반환
    return 'negative'
  else: #x가 양수 또는 음수가 아니면 zero 반환
    return 'zero'

# 실행 결과를 확인하시오.
for x in [-1, 0, 1]:
  print(sign(x))
```

```
negative
zero
positive
```

1-5. Python: Function

```
def hello(name, loud=False):
  if loud:  #loud가 True면 전체 대문자로 인사출력
    print ('HELLO, %s!' % name.upper())
  else: #loud가 False면 첫 글자만 대문자로 인사출력
    print ('Hello, %s' % name)

# 실행 결과를 확인하시오
hello('Bob')

# 실행 결과를 확인하시오
hello('Fred', loud=True)
```

```
Hello, Bob
HELLO, FRED!
```

1-6. Python: Zip

에러가 나는 이유: 함수 내에 3개의 반복가능 자료형이 있으므로 변수 역시 3개가 되어야 에러가 발생하지 않는다.

```
# 1 4 7 / 2 5 8 / 3 6 9 출력
#각 리스트(순회가능한 객체)의 원소들을 순차적으로 튜플로 반환
for x, y, z in zip([1, 2, 3], [4, 5, 6], [7, 8, 9]):
  print(x, y, z)
```

```
1 4 7
2 5 8
3 6 9
```

```
# 에러가 나는 이유는 ?
#zip(): 동일한 개수로 이루어진 반복 가능(iterable)한 자료형을 묶어 주는 역할을 하는 함수
#함수 내에 3개의 자료형이 있으므로 변수 역시 3개가 되어야 에러가 나지 않는다.
for x, y in zip([1, 2, 3], [4, 5, 6], [7, 8, 9]):
  print(x, y, z)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-39-d7f2424fb504> in <module>()
      2 #zip(): 동일한 개수로 이루어진 반복 가능(iterable)한 자료형을 묶어 주는 역할을 하
는 함수
      3 #함수 내에 3개의 자료형이 있으므로 변수 역시 3개가 되어야 에러가 나지 않는다.
----> 4 for x, y in zip([1, 2, 3], [4, 5, 6], [7, 8, 9]):
      5   print(x, y, z)

ValueError: too many values to unpack (expected 2)

 SEARCH STACK OVERFLOW
```

1-7. Python: Magic method __init__()

```
class HelloWorld():
  def __init__(self): #__init__은 자동 실행
    print("init")

# 실행 결과를 확인하시오
helloworld = HelloWorld()
```

```
init
```

1-8. Python: Magic method __call__()

__init__: 객체를 생성할 때 불리는 초기화(initialize) 메서드

__call__: 객체 변수를 사용해서 호출을 가능하게 하는 메서드

__init__함수는 인스턴스를 생성할 때 불리는 초기화 메서드이다. 한편 __call__함수는 인스턴스 변수를 사용해서 호출을 가능하게 하는 메서드로, 호출할 때 불리는 함수이다. __call__함수가 정의되어 있지 않으면 인스턴스를 변수로 호출할 수 없으므로 에러가 난다.

```
class HelloWorld():
  def __init__(self):
    print("init")

helloworld = HelloWorld() #객체 초기화가 이루어져 init출력
#callable(HelloWorld) #True: class 자체는 호출을 통해 객체를 생성하므로 callable

# 실행 결과를 확인하시오
callable(helloworld) #False: 생성된 객체는 __call__() 이 class 내에 정의되어 있지 않기 때문에 callable 하지 않음
```

```
init
False
```

```
#객체를 호출 가능하게 해주는 __call__이 정의되어 있지 않아 에러가 난다.
helloworld() # Error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-53-e11ab98f9e43> in <module>()
      1 #객체를 호출 가능하게 해주는 __call__이 정의되어 있지 않아 에러가 난다.
----> 2 helloworld() # Error

TypeError: 'HelloWorld' object is not callable

 SEARCH STACK OVERFLOW
```

```
class HelloWorld():
  def __init__(self): #객체(인스턴스) 초기화할 때 실행
    print("init")

  def __call__(self): #인스턴스가 호출되었을 때 실행
    #객체를 함수로써 호출 가능하게 활용하기 위하여 사용하는 특수한 메소드
    print("Hello world")

helloworld = HelloWorld()

#개체를 호출 가능하게 해주는 __call__이 정의되어 있어 callable하다.
# 실행 결과를 확인하시오
helloworld()  #객체 호출
```

```
init
Hello world
```

1. Part 1 – Numpy

1-9. Numpy: array

```
import numpy as np
# rank가 1인 배열 생성
a = np.array([1, 2, 3])
# 각 라인 별 실행 결과를 확인하시오
type(a) #numpy자료형 확인
```

```
numpy.ndarray
```

```
a.shape #배열의 크기 (3,0)
```

```
(3,)
```

```
a.ndim  #1차원
```

```
1
```

```
a[0], a[1], a[2]  #배열의 요소들을 튜플로 출력
```

```
(5, 2, 3)
```

```
# 요소 변경 후 실행 결과를 확인하시오.
a[0] = 5  #배열의 0번째 요소를 5로 대체
a
```

```
array([5, 2, 3])
```

```
# rank가 2인 배열 생성
b = np.array([[1,2,3],[4,5,6]])
# 각 라인 별 실행 결과를 확인하시오
b.shape #배열의 크기 (2,3)
```

```
(2, 3)
```

```
b.ndim  #배열의 차원은 2차원
```

```
2
```

```
b[0, 0], b[0, 1], b[1, 0] #배열의 요소들을 튜플로 출력
```

```
(1, 2, 4)
```

1-10. Numpy: zeros, ones, full, eye, random

```
import numpy as np
# 모든 값이 0인 배열 생성 후 실행 결과를 확인하시오.
a = np.zeros((2,2))
a # 모든 값이 0인 2x2 행렬
```

```
array([[0., 0.],
       [0., 0.]])
```

```
# 모든 값이 1인 배열 생성 후 실행 결과를 확인하시오.
b = np.ones((1,2))
b #모든 값이 1인 1x2 행렬
```

```
array([[1., 1.]])
```

```
# 모든 값이 특정 상수인 배열 생성 후 실행 결과를 확인하시오.
c = np.full((2,2), 7)
c #모든 값이 7인 2x2 행렬
```

```
array([[7, 7],
       [7, 7]])
```

```
# 2x2 단위행렬 생성 후 실행 결과를 확인하시오.
d = np.eye(2)
d #2x2 단위행렬
```

```
array([[1., 0.],
       [0., 1.]])
```

```
# 임의의 값으로 채워진 배열 생성 후 실행 결과를 확인하시오.
e = np.random.random((2,2))
e #랜덤으로 값이 채워진 2x2 행렬
```

```
array([[0.28923233, 0.95138202],
       [0.38237336, 0.27952743]])
```

1-11. Numpy: where

```
import numpy as np
# 0~9를 갖는 배열 생성
a = np.arange(10)
# 실행 결과를 확인하시오.
np.where(a<5) #요소 중 값이 5보다 작은 요소들의 인덱스만을 갖는 배열
```

```
(array([0, 1, 2, 3, 4]),)
```

```
# 실행 결과를 확인하시오.
np.where(a<5, a, 10*a)  #a가 5보다 작으면 그대로 두고, 그렇지 않으면 10을 곱한 배열
```

```
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

```
# 실행 결과를 확인하시오.
np.where(a<4, a, -1)  #a가 4보다 작으면 그대로 두고, 그렇지 않으면 -1로 대체한 배열
```

```
array([ 0,  1,  2,  3, -1, -1, -1, -1, -1, -1])
```

1-12. Numpy: array slice

```
import numpy as np

# 3x4 배열 생성
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# 슬라이싱을 이용하여 첫 두 행과 1열, 2열로 이루어진 부분배열 생성
# b는 shape가 (2,2)인 배열이 됨
# 실행 결과를 확인하시오.
b = a[:2, 1:3]
b
```

```
array([[2, 3],
       [6, 7]])
```

```
# 슬라이싱된 배열은 원본 배열과 같은 데이터를 참조, 즉 슬라이싱 된 배열을 수정하면 원본 배열 역시 수정됨
# 실행 결과를 확인하시오.
a[0, 1] #a배열의 0행 1열
```

```
2
```

```
# b[0, 0]은 a[0, 1]과 같은 데이터
b[0, 0] = 77  #b배열의 0행 0열 값을 77로 수정
b
```

```
array([[77,  3],
       [ 6,  7]])
```

```
# 실행 결과를 확인하시오.
a[0, 1] #b배열을 수정함에 따라 a배열의 같은 위치에 해당하는 0행 1열의 값이 77로 변경
```

```
77
```

1-13. Numpy: array reshape

```
import numpy as np

# 3x4 배열 생성
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# 3x4 배열을 2x6으로 변형
b = np.reshape(a, (2, 6))
```

```
# 실행 결과를 확인하시오.
b #2행 6열
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

```
# 슬라이싱된 배열과 마찬가지고 reshape된 배열도 원본 배열과 같은 데이터를 참조,
#즉 슬라이싱 된 배열을 수정하면 원본 배열 역시 수정됨
# 실행 결과를 확인하시오.
a[0, 1] #0행 1열의 값
```

```
2
```

```
# b[0, 1]은 a[0, 1]과 같은 데이터
b[0, 1] = 77
a[0, 1] #a[0,1]도 수정된 b[0,1]
```

```
77
```

1-14. Numpy: array reshape

```
import numpy as np

# 2x2 행렬 두 개 생성
a = np.array([[1, 0], [0, 1]])
b = np.array([[4, 1], [2, 2]])

# 2x2 행렬 간 dot product, 실행 결과를 확인하시오.
np.dot(a, b)  #두 행렬을 내적한 결과에 해당하는 행렬
```

```
array([[4, 1],
       [2, 2]])
```

1. Part 1 – Pyorch

1-15. Pytorch: Tensor

```
import torch
import numpy as np

# Float 형을 갖는 tensor 생성
t1 = torch.FloatTensor([0, 1, 2, 3, 4, 5, 6])
t2 = torch.tensor(np.arange(7)) # t1과 동일

# 각 라인 별 실행 결과를 확인하시오
t1.shape  #t1의 크기
```

```
torch.Size([7])
```

```
t2.shape  #t2의 크기
```

```
torch.Size([7])
```

```
t1.dim()  #t1의 차원
```

```
1
```

```
t1.size() #t1의 크기
```

```
torch.Size([7])
```

```
t1[:2]  #t1의 0~1번째 인덱스까지
```

```
tensor([0., 1.])
```

```
t1[3:]  #t1의 3~끝 인덱스까지
```

```
tensor([3., 4., 5., 6.])
```

1-16. Pytorch: numpy의 array <-> pytorch의 tensor

```
# 각 라인 별 실행 결과를 확인하시오
b = np.arange(7)
t1 = torch.FloatTensor([0,1,2,3,4,5,6])
```

```
type(b) #numpy array
```

```
numpy.ndarray
```

```
type(t1)  #torch tensor
```

```
torch.Tensor
```

```
# numpy의 array인 b를 torch의 tensor로 변환
tt = torch.tensor(b)
t_from = torch.from_numpy(b)
```

```
type(tt)  #tt의 자료형: tensor
```

```
torch.Tensor
```

```
type(t_from)   #t_fron의 자료형: tensor
```

```
torch.Tensor
```

```
tt
```

```
tensor([0, 1, 2, 3, 4, 5, 6])
```

```
t_from
```

```
tensor([0, 1, 2, 3, 4, 5, 6])
```

1-17. Pytorch: numpy의 array <-> pytorch의 tensor

tensor()를 사용하여 tensor로 변환했을 때는 원본 numpy array를 수정해도 함께 수정되진 않지만, from_numpy()를 사용하여 tensor로 변환했을 때는 함께 수정된다.

```
b[0]= -10
tt  #array b를 참조하지 않음
```

```
tensor([0, 1, 2, 3, 4, 5, 6])
```

```
t_from  #array b를 참조하여 b의 요소 변화가 반영됨
```

```
tensor([-10,   1,   2,   3,   4,   5,   6])
```

```
# torch의 tensor를 numpy의 array로 바꿈
t_to_np = t_from.numpy()
type(t_to_np) #tensor를 array로 변환
```

```
numpy.ndarray
```

1-18. Pytorch: Broadcasting

Broadcasting: 크기가 작은 행렬을 크기가 큰 행렬과 모양이 맞게끔 늘려주는 것.

m1이 broadcasting을 통해 m2의 크기와 동일한 모양으로 맞춰진 뒤, 행렬의 덧셈이 행해진다.

```
m1 = torch.FloatTensor([[3,3]])
m2 = torch.FloatTensor([[2,2]])

m1+m2 #두 tensor의 합
```

```
tensor([[5., 5.]])
```

```
# Vector + scalar
m1 = torch.FloatTensor([[1,2]])
m2 = torch.FloatTensor([[3]])

m1 + m2 #m1 tensor의 각 열에 스칼라값 더한 tensor
```

```
tensor([[4., 5.]])
```

```
# 2 x 1 Vector + 1 x 2 Vector
m1 = torch.FloatTensor([[1,2]])
m2 = torch.FloatTensor([[3],[4]])

m1 + m2 #m1이 broadcasting을 통해 m2의 크기와 동일한 크기로 맞춰진 뒤, 행렬의 덧셈이 행해짐
```

```
tensor([[4., 5.],
        [5., 6.]])
```

1-19. Pytorch: Mul vs MatMul

```
m1 = torch.FloatTensor([[1,2],[3,4]])
m2 = torch.FloatTensor([[1],[2]])

m1 * m2 #동일한 크기의 행렬에 대하여 동일한 위치에 있는 원소끼리 곱하는 element-wise 곱셈
#m2의 broadcasting이 이루어진 후에 element-wise 곱셈
```

```
tensor([[1., 2.],
        [6., 8.]])
```

```
m1.mul(m2) #동일한 크기의 행렬에 대하여 동일한 위치에 있는 원소끼리 곱하는 element-wise 곱셈
#m2의 broadcasting이 이루어진 후에 element-wise 곱셈
```

```
tensor([[1., 2.],
        [6., 8.]])
```

```
m1 = torch.FloatTensor([[1,2],[3,4]])
m2 = torch.FloatTensor([[1],[2]])

m1.matmul(m2) #Matrix Multiplication, 즉 행렬곱셈
```

```
tensor([[ 5.],
        [11.]])
```

1-20. Pytorch: View (Reshape in Numpy)

view([]): reshape와 마찬가지로 행렬의 크기를 변경하는 기능을 수행

```
t = np.arange(12).reshape(-1,2,3) #0~11의 정수를 원소로 갖는 1차원 배열을 3차원 배열로 변경(tensor 크기 변경)
#(?,2,3)의 크기로 텐서의 크기를 변경
floatT = torch.FloatTensor(t) #array를 tensor로 변환

floatT.shape  #floatT의 크기 보기
```

```
torch.Size([2, 2, 3])
```

```
floatT.view([-1,3]) #floatT tensor의 크기를 (-1,3)으로 변경
```

```
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]])
```

2. Part 2- Exercise

2-1. numpy를 활용하여 10부터 49까지를 값으로 갖는 배열을 생성하시오. (ex. 10, 11, … 49)

```
import numpy as np


np.arange(10,50)
```

2-2. numpy를 활용하여 49부터 10까지를 값으로 갖는 배열을 생성하시오. (ex. 49, 48, … 10)

```
import numpy as np

np.arange(49,9,-1)
```

2-3. 아래 예제와 같이 [입력]이 주어졌을 때, 'A'와 'B'의 요소가 같은 위치(index)를 갖는 배열을 구하시오. (Hint: np.where)

```
import numpy as np
A = np.array([1,2,3,2,3,4,3,4,5,6])
B = np.array([7,2,10,2,7,4,9,4,9,8])

np.where(A==B)[0]
```

2-4. 아래 예제와 같이 [입력]이 주어졌을 때, Broadcasting 성질을 활용하여 행렬 'a'에 1을 더한 결과를 구하시오.

```
import numpy as np
a = np.arange(9).reshape(3, 3)

print(a + 1)
```

2-5. 아래 예제와 같이 [입력]이 주어졌을 때 [출력]이 나오도록, 1열과 2열을 바꾸는 코드를 작성하시오.

```
import numpy as np

a = np.arange(9).reshape(3, 3)
print(a[[1,0,2]])
```

2-6. 아래와 같은 결과가 나오기 위한 변수 'a'를 선언하시오. (Hint: np.arange(n))

```
import numpy as np

a = np.arange(24).reshape(-1,3,4) #Problem

print("Shape: ", a.shape)
print("Value: ", a)
```

2-7. 문제2-6에서 선언한 변수 'a'를 활용하여 다음과 같은 결과를 만드는 slice 부분을 작성하시오.

```
print(a[0,1:,2:]) #Problem
```

2-8. 아래와 같은 Output을 출력하도록 torch의 view 함수를 작성하시오.

```
import torch
import numpy as np

t = np.arange(24)
print(t.shape)

floatT = torch.FloatTensor(t)
print(floatT.view([4,-1])[2:,(1,4)])
```