PART 1

단층 퍼셉트론(SLP)

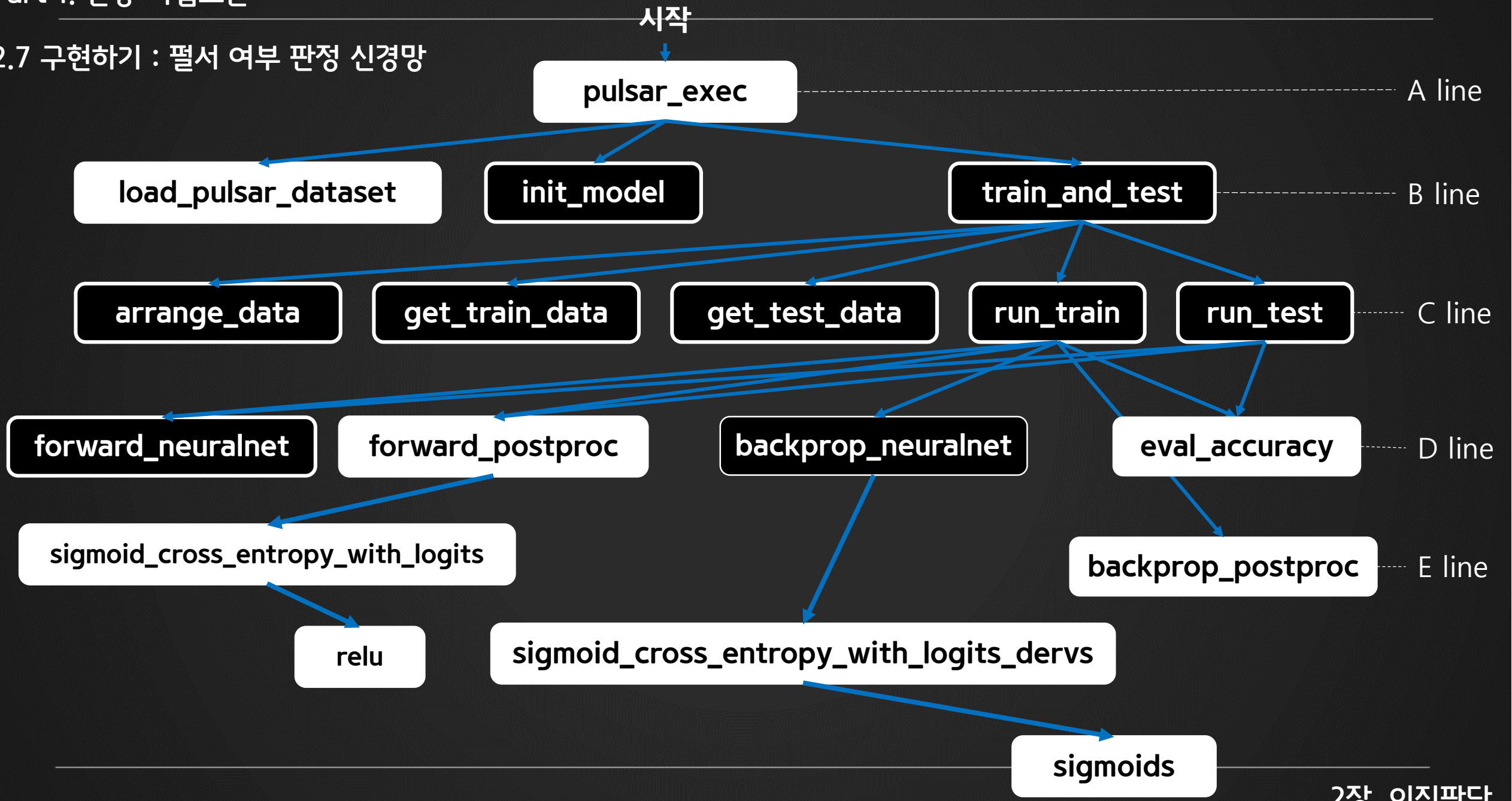
1장 회귀분석

2장 이진판단

3장 선택분류

딥러닝 & 강화학습 담당
이재화 강사

2.7 구현하기 : 펄서 여부 판정 신경망



2.0 코드 재활용을 위한 파이썬 파일 실행시키기

```
%run ../../leeyua/AI_CODE/AI_abalone.ipynb
```

 # 기존에 구축해 놓은 함수를 재활용
(앞장의 검은색 블록)

에폭(총 학습 반복 횟수)

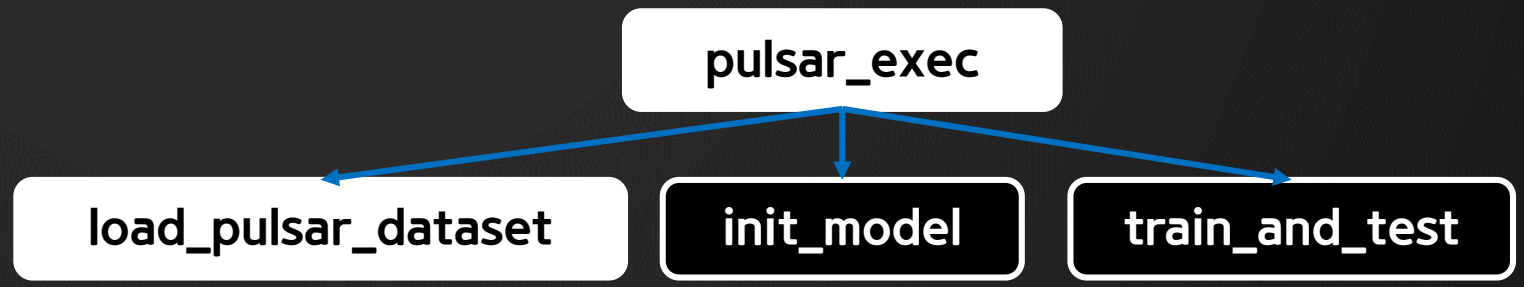
미니배치 크기(데이터 분할 단위)

보고 주기(얼마나 자주 보고할 것인가)

2.1 메인 함수 정의하기

```
def pulsar_exec(epoch_count = 10, mb_size = 10, report = 1):
  load_pulsar_dataset()
  init_model()
  train_and_test(epoch_count, mb_size, report)
```

 # 3가지 함수 실행



2.2 데이터 적재 함수의 정의

```
def load_pulsar_dataset():
  with open('pulsar_stars.csv') as csvfile:
    csvreader = csv.reader(csvfile)
    next(csvreader, None)
    rows = []
    for row in csvreader:
      rows.append(row)

  global data, input_cnt, output_cnt
  input_cnt, output_cnt, = 8,1
  data = np.asarray(rows, dtype='float32')
```

#with 명령어로 데이터를 열어주고 별칭 지정
#csv.reader()로 읽어들입니다.
#첫번째 행에 컬럼명 들을 **건너뛰어** 줍니다.
#데이터를 읽어들이기 위한 **빈 리스트 생성**
#반복문 for를 활용한 데이터 읽어들이기
#append()를 활용한 데이터 누적 저장

#전역변수 생성
#입력 크기와 출력 크기 (독립변수 8개, 종속변수 1개)

#np.asarray()를 활용하여 rows변수를 리스트 구조에서 배열로 변환하는 과정

rows의 파이썬의 리스트 구조는현재 list 구조. 즉, numpy에서 사용하는 다양한 일괄 산술 연산에는
비효율적 및 부적합 np.asarray()를 활용하여 배열로 변환

input_cnt = 8

output_cnt = 1

	n of the integrated profile	n of the integrated profile	s of the integrated profile	s of the integrated profile	ean of the DM-SNR curve	tion of the DM-SNR curve	osis of the DM-SNR curve	ess of the DM-SNR curve	target_class
1	140.5625	55.68378214	-0.234571412	-0.699648398	3.199832776	19.11042633	7.975531794	74.24222492	0
2	102.5078125	58.88243001	0.465318154	-0.515087909	1.677257525	14.86014572	10.57648674	127.3935796	0
3	103.015625	39.34164944	0.323328365	1.051164429	3.121237458	21.74466875	7.735822015	63.17190911	0
4	136.75	57.17844874	-0.068414638	-0.636238369	3.642976589	20.9592803	6.89649891	53.59366067	0
5	88.7265625	40.67222541	0.600866079	1.123491692	1.178929766	11.4687196	14.26957284	252.5673058	0

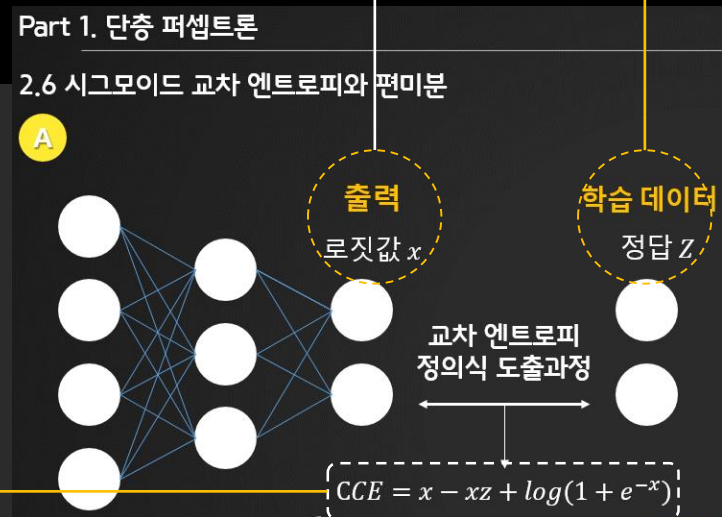
2.3 후처리 과정에 대한 순전파와 역전파 함수 재정의

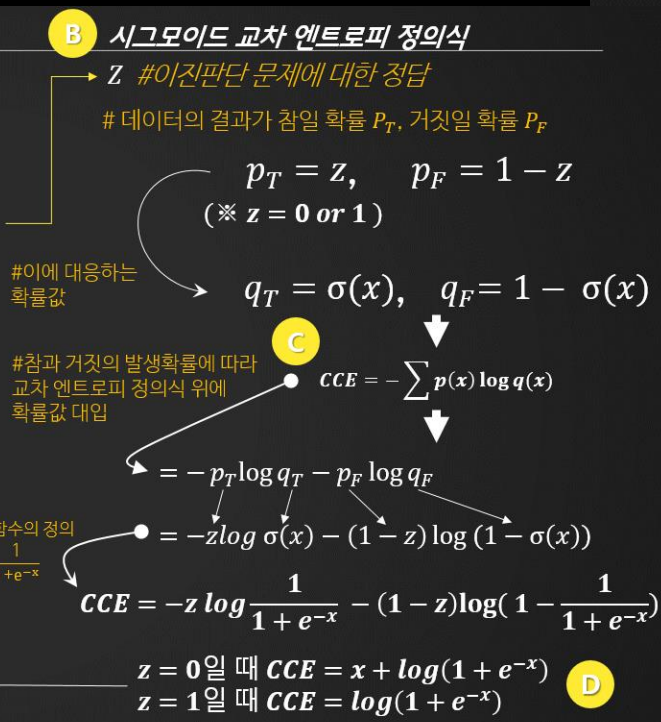
```
def forward_postproc(output, y): # 시그모이드 교차 엔트로피값을 구하는 2단계 연산 수행
    entropy = sigmoid_cross_entropy_with_logits(y, output)
    loss = np.mean(entropy)
    # 각 수행값들에 대한 평균값 연산
    return loss, [y, output, entropy]
```

1

2

#바로 다음에 구축할 함수에서 활용하고자 함





2.3 후처리 과정에 대한 순전파와 역전파 함수 재정의

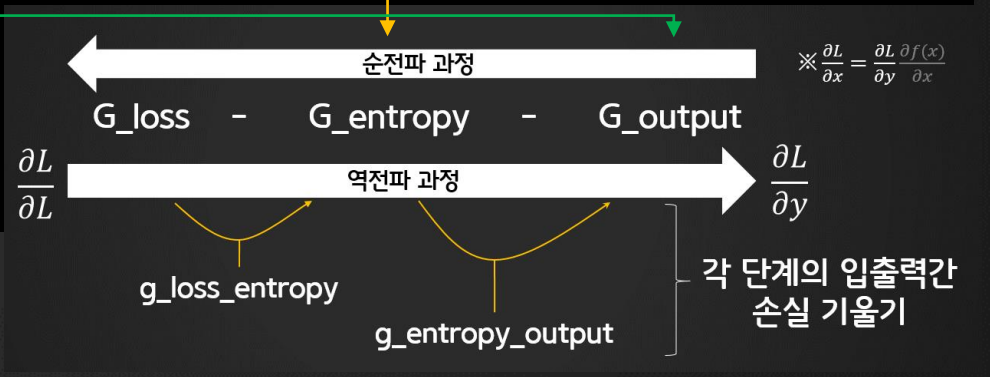
순전파의 역순에 대항하는 일련의 과정을 거쳐 G_output 도출 $(\frac{\partial L}{\partial Y})$

$$L = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N} square_{ij}}{MN} \quad \blacktriangleright \quad \frac{\partial L}{\partial square_{ij}} = \frac{1}{MN}$$

```
def backprop_postproc(G_loss, aux):     #G_loss = 1  ※∂L/∂L = 1.0
  y, output, entropy = aux              # 바로 이전 순전파 과정에서 받아온 [y, output, entropy]

  g_loss_entropy = 1.0 / np.prod(entropy.shape) #제곱 연산에 대한 역전파 처리/ backward_postproc()에서 정의

  g_entropy_output = sigmoid_cross_entropy_with_logits_derv(y,output)
  #entrop와 output 사이의 부분 기울기

  G_entropy = g_loss_entropy * G_loss

  G_output = g_entropy_output * G_entropy

  return G_output
```

시그모이드 교차 엔트로피의 편미분

$$\frac{\partial CCE}{\partial x} = -z + \sigma(x)$$



연쇄적 계산을 통해 손실기울기 도출

※ np.prod() : 배열의 요소들을 곱하는 함수

2.4 정확도 계산 함수의 재정의

#이진 판단 문제에서 정확도
신경망이 추정한 로짓값에 따른 판단 <-> 정답으로 주어진 판단이 일치하는 비율

```
def eval_accuracy(output, y):

    estimate = np.greater(output,0)    # output에 담긴 로짓값들의 부호를 확인하면 신경망의 판단을 확인가능

    answer = np.greater(y, 0.5)        # 정답으로 주어진 판단은 참 일때 1, 거짓일 때 0 값을 갖도록
                                         (안전하게 0.5기준으로 참 거짓 구분 )
    correct = np.equal(estimate,answer)  # 두 판단의 일치여부를 비교하여 정답을 correct에 저장

    return np.mean(correct)              # 평균 연산을 수행하여 올바른 판단의 비율 측정
```

※ np.greater(x1,x2): x2 기준보다 값이 크면 True 반환
※ np.equal(x1,x2): x1과 x2가 같으면 True 반환

2.4_1 예제

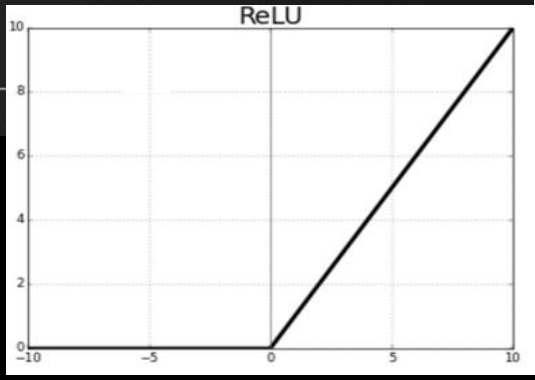
```
x = [1,0,0,1,1]
y = [1,1,1,1,0]
correct = np.equal(x,y)
print(correct)
np.mean(correct)
```

```
[ True False False True False]
0.4
```

"output에 담긴 로짓값들의 부호만으로 신경망의 판단이 가능할까?"

'이진판단' 문제에 한에서는 가능해!

2.4 시그모이드 관련 함수 정의




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

relu()는 음수인 원소들을 모두 찾아내 0 으로 대치하는 효과
입력값이 음수인 경우 0을 출력, 양수는 양수를 그대로 출력

```
def sigmoid(x): # 시그모이드 함수
    return np.exp(-relu(-x)) / (1.0 + np.exp(-np.abs(x)))
```

2.3 시그모이드 함수

$$\sigma(x) = \frac{1}{1+e^{-x}}$$ A

C
※ $f(x) = 1$
※ $g(x) = 1 + e^{-x}$

D 시그모이드 함수의 도함수

$$\frac{-(1+e^{-x})'}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} \cdots = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right)$$

※ $\sigma(x) = y$ $\quad \sigma(x)(1-\sigma(x))$

$$y(1-y)$$ E

```
def sigmoid_derv(x,y): # 시그모이드 함수의 도함수
    return y * (1-y)
```

```
# 시그모이드 교차 엔트로피 함수
def sigmoid_cross_entropy_with_logits(z,x):
    return relu(x) - x * z + np.log(1 + np.exp(-np.abs(x)))
```

```
# 시그모이드 교차 엔트로피 함수의 도함수
def sigmoid_cross_entropy_with_logits_derv(z,x):
    return -z + sigmoid(x)
```

E

$$\frac{\partial CCE}{\partial x} = -z + \sigma(x)$$

출력 로짓값 x

학습 데이터 정답 z

교차 엔트로피 정의식 도출과정

$$CCE = x - xz + log(1+e^{-x})$$

2.5 실행하기

```
pulsar_exec()
```

```
Epoch 1: loss=0.154, accuracy=0.959/0.972
Epoch 2: loss=0.131, accuracy=0.966/0.972
Epoch 3: loss=0.136, accuracy=0.967/0.970
Epoch 4: loss=0.133, accuracy=0.968/0.970
Epoch 5: loss=0.121, accuracy=0.968/0.969
Epoch 6: loss=0.145, accuracy=0.968/0.974
Epoch 7: loss=0.122, accuracy=0.970/0.975
Epoch 8: loss=0.127, accuracy=0.970/0.976
Epoch 9: loss=0.125, accuracy=0.970/0.976
Epoch 10: loss=0.134, accuracy=0.968/0.976

Final Test: final accuracy = 0.976
```

#다만 이 결과에는 눈속임이 들어 있습니다.

2.5_1 새로운 예제로의 실험

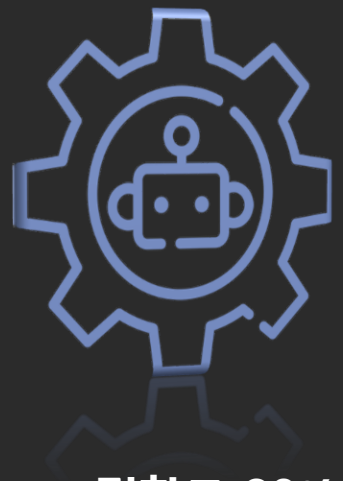
```
x = np.array([130,52,0.4,0.6,3.1,20,8,72]) #가상의 새로운 데이터 입력
output = forward_neuralnet(x)              #신경망 연산을 활용한 결과 출력
print(output)
x_sig= sigmoid(output[0])                  #앞서 구축한 sigmoid() 통과
new_data = np.greater(x_sig,0.5)           #np.greater()를 활용한 기준
print(new_data)                            #결과 출력
```

```
(array([-6.51508291]), array([130. ,  52. ,   0.4,   0.6,   3.1,  20. ,   8. ,  72. ]))
[False] #우리가 찾는 펄서가 아니다.
```

2.6 확장하기

: 균형잡힌 데이터셋과 착시 없는 평가방법

```
pulsar_exec()

Epoch 1: loss=0.138, accuracy=0.960/0.970
Epoch 2: loss=0.134, accuracy=0.966/0.976
Epoch 3: loss=0.134, accuracy=0.967/0.966
Epoch 4: loss=0.140, accuracy=0.967/0.975
Epoch 5: loss=0.129, accuracy=0.966/0.977
Epoch 6: loss=0.118, accuracy=0.968/0.967
Epoch 7: loss=0.127, accuracy=0.968/0.973
Epoch 8: loss=0.126, accuracy=0.969/0.976
Epoch 9: loss=0.124, accuracy=0.969/0.977
Epoch 10: loss=0.131, accuracy=0.968/0.975

Final Test: final accuracy = 0.975
```



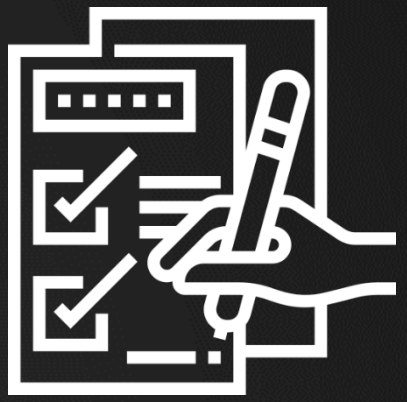
"정답은 언제나 Star 입니다."

정확도 90%의 신경망 모델

새로운 신경망 평가지표의 필요성

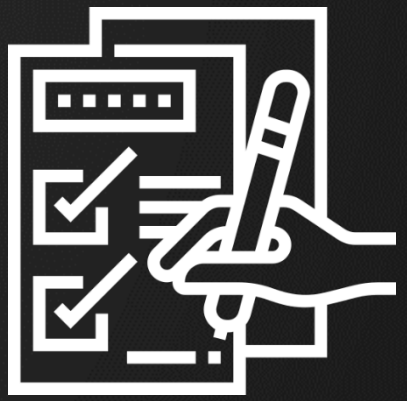
Pulsar = 10%

Star = 90%



Pulsar = 50%

Star = 50%



2.8 확장하기 : 균형잡힌 데이터셋과 착시 없는 평가방법

정밀도

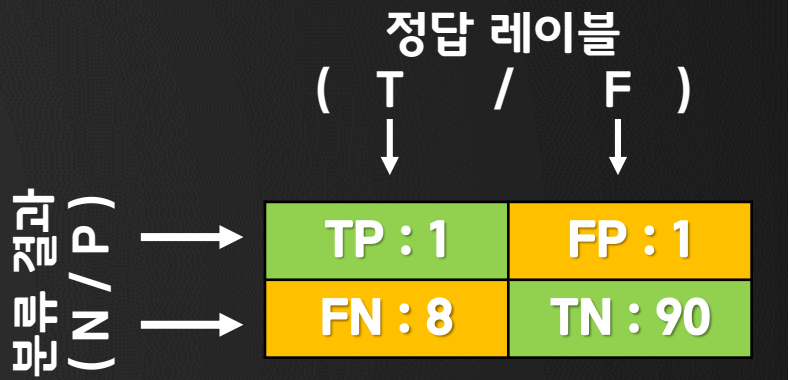
"분류 결과 양성으로 식별된 사례 중 실제로 양성이었던 사례의 비율은 어느 정도인가요?"

$$\frac{TP}{TP + FP}$$

재현율

"실제 양성 중 정확히 양성이라고 식별된 사례의 비율은 어느 정도 인가요?"

$$\frac{TP}{TP + FN}$$



F1

$$\frac{2}{\frac{1}{\text{정밀도}} + \frac{1}{\text{재현율}}} = 2 * \frac{\text{정밀도} * \text{재현율}}{\text{정밀도} + \text{재현율}}$$

2.7 코드 재활용을 위한 파이썬 파일 실행시키기

```
%run ../../leeyua/AI_CODE/AI_pulsar.ipynb
```

2.8 메인 실행 함수 재정의

```
def pulsar_exec(epoch_count=10,mb_size=10, report=1, adjust_ratio = False):

                                        # False : 펄서 데이터보다 별 데이터가 훨씬 많은 데이터셋을 활용(기본값)
                                        # True : 펄서 데이터와 별 데이터가 같은 데이터셋을 활용


    load_pulsar_dataset(adjust_ratio)

    init_model()

    train_and_test(epoch_count, mb_size, report)
```

2.9 데이터 적재 함수의 재정의_1

불균형한 데이터의 균형을 맞춰주기 위해 펄서 데이터를 반복복사해주는 기능구현
인자값으로 adjust_ratio 부분이 지정되었습니다.

```
def load_pulsar_dataset(adjust_ratio):
    pulsars, stars = [], []                                  #펄서 데이터와 별 데이터를 담아주기 위한 빈 리스트 생성
    with open('pulsar_stars.csv') as csvfile:
        csvreader = csv.reader(csvfile)
        next(csvreader, None)
        rows = []
        for row in csvreader:                                #만약 받아들이는 데이터의 9번째 열이 '1'(펄서)인 경우 append()를 통해
            if row[8] == '1' : pulsars.append(row)           pulsars에 데이터 저장
            else : stars.append(row)                         #아닌경우(별) append()를 통해 stars에 데이터 저장
    global data, input_cnt, output_cnt
    input_cnt, output_cnt, = 8,1
    star_cnt, pulsar_cnt = len(stars), len(pulsars)          #전역변수의 값을 각 starts와 pulsars의 행 크기에 맞춰서 지정

    if adjust_ratio:
        data = np.zeros([2*star_cnt , 9])
        data[0:star_cnt, :] = np.asarray(stars, dtype='float32')
        for n in range(star_cnt):
            data[star_cnt+n]=np.asarray(pulsars[n % pulsar_cnt], dtype='float32')

    else:
        data = np.zeros([star_cnt+pulsar_cnt,9])
        data[0:star_cnt, :] = np.asarray(stars, dtype="float32")
        data[star_cnt:, :] = np.asarray(pulsars, dtype="float32")
```

다음페이지에서 설명!

2.9 데이터 적재 함수의 재정의_2

… #데이터를 모두 읽고 나서는 매개변수값이 TRUE인 경우 즉,
데이터를 늘려주는 방법을 선택했을 때의 코드 구축.

```
if adjust_ratio:

    data = np.zeros([2*star_cnt , 9])
```

#데이터에 행의 크기가 두배가 되는 data라는 '임시공간'(버퍼) 를 생성.
버퍼의 용도는 펄서 데이터와 별 데이터를 각각 5:5로 채워 균형을 맞춰주기 위함 .

```
    data[0:star_cnt, :] = np.asarray(stars, dtype='float32')
```

#다음으로 버퍼의 절반 공간 즉, 별 데이터만큼의 공간에는 **별 데이터**를 행렬형태로 넣어준다.

```
    #star_cnt만큼 반복문 수행
    for n in range(star_cnt):
        # 반복문을 통해 별 데이터 '이후' 행 부터 하여 '펄서 데이터' 가 반복적으로 저장.
        data[star_cnt + n] = np.asarray(pulsars[n % pulsar_cnt], dtype='float32')
        # '%(나머지) 연산자'를 활용하여, 펄서 데이터 수를 계속 반복 출력
        # 나머지 버퍼 공간에 펄서 데이터가 들어가게 되면서 데이터의 균형이 맞춰짐.

else: #하지만 인자값이 True가 아니라면
```

펄서와 별 데이터를 원 데이터의 크기에 맞춰 버퍼를 생성
데이터를 그대로 넣어주도록 코드를 작성.

```
    data = np.zeros([star_cnt+pulsar_cnt,9])

    data[0:star_cnt, :] = np.asarray(stars, dtype="float32")

    data[star_cnt:, :] = np.asarray(pulsars, dtype="float32")
```

2.10 정확도 계산 정의 함수 재정의

기존 함수에서는 신경망의 추측값과 실제값이 일치하는 비율에 맞춰 정확도 추출.
하지만 이번에는 앞서 배웠던 '정밀도'와 '재현율' 그리고 'F1 수치'에 맞춰 정확도 추출.

```
def eval_accuracy(output, y):
    est_yes = np.greater(output, 0)
    est_no = np.logical_not(est_yes)
    ans_yes = np.greater(y, 0.5)
    ans_no = np.logical_not(ans_yes)
```

정밀도와 재현율을 구하기 위해 '추측값(output)'과 '실제값(y)'을 나타내는 값 들을 먼저 준비
#np.greater() :
0보다 output 이 더 크면 TRUE 값을 반환 ,output 이 음수인 경우에는 False를 반환
즉, output 혹은 y가 이 '참'인지를 확인
np.logical_not() 함수는 배열 원소가 조건을 만족하지 않는 경우에 '참'을 반환
즉, output 혹은 y가 '거짓'인지를 확인

 # tp ~ tn까지 계산할 수 있는 식 구현.

```
    tp = np.sum(np.logical_and(est_yes, ans_yes))
    fp = np.sum(np.logical_and(est_no, ans_yes))
    fn = np.sum(np.logical_and(est_yes,ans_no))
    tn = np.sum(np.logical_and(est_no,ans_no))

    accuracy = safe_div(tp+tn,tp+tn+fn+fp)

    precision = safe_div(tp, tp + fp)

    recall = safe_div(tp, tp + fn)

    f1 = 2 * safe_div(recall * precision, recall + precision)

    return [accuracy, precision, recall, f1]
```

tp : 예측값과 실제값이 모두 참인 경우
fp : 분류결과가 거짓이지만, 정답은 참인 경우
fn : 분류결과는 참인데, 정답은 거짓인 경우
tn : 정답과 추측 모두 거짓을 의미

safe_div() : 나눗셈을 수행하던 중 형변환이나 0을 나누게 될 경우 에러를 방지하는 나눗셈

정밀도

"분류 결과 양성으로 식별된 사례 중 실제로 양성이었던 사례의 비율은 어느 정도인가요?"

$$\frac{TP}{TP + FP}$$

재현율

"실제 양성 중 정확히 양성이라고 식별된 사례의 비율은 어느 정도 인가요?"

$$\frac{TP}{TP + FN}$$

F1

$$\frac{2}{\frac{1}{\text{정밀도}} + \frac{1}{\text{재현율}}} = 2 * \frac{\text{정밀도} * \text{재현율}}{\text{정밀도} + \text{재현율}}$$

2.11 안전한 계산을 위한 나눗셈 정의

```
def safe_div(p,q):
    p,q = float(p),float(q)  #모두 float타입으로 형변환
    if np.abs(q) < 1.0e-20: return np.sign(p)  #모두 float타입으로 형변환

    # 만약 q 즉, 분모가 너무 '작은 수' 라서나눗셈을 진행할 때 '에러가 날 수도 있겠다. 하는 경우라면,
    # 조건을 걸어 '절댓값' 자체가 너무 작은 값인 경우라면 ,
    # np.sign()를 활용하여, 분자의 값을 0,-1,+1 로 조정하여 나눗셈을 진행하도록 합니다.

    return p / q
```

※ np.sign() : 0 이면 0, 양수는 1, 음수면 -1을 출력

일반적인 나숫셈 연산자를 통해 연산을 진행할 경우 많은 문제를 보인다.

```
C:\Users\leeyua\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_launcher.py:41: RuntimeWarning: divide by zero encountered in long_scalars
C:\Users\leeyua\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_launcher.py:47: RuntimeWarning: invalid value encountered in long_scalars
C:\Users\leeyua\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_launcher.py:50: RuntimeWarning: invalid value encountered in long_scalars
Epoch1:loss = 0.139,result=3602.358,108.000,11.000,324.000
Epoch2:loss = 0.130,result=3600.752,80.000,17.000,240.000
Epoch3:loss = 0.130,result=3606.715,162.000,37.000,486.000
Epoch4:loss = 0.128,result=3601.179,88.000,14.000,264.000
Epoch5:loss = 0.137,result=3600.755,81.000,29.000,243.000
C:\Users\leeyua\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_launcher.py:41: RuntimeWarning: invalid value encountered in long_scalars
Epoch6:loss = 0.118,result=3602.363,108.000,10.000,324.000
Epoch7:loss = 0.123,result=3598.037,33.000,235.000,99.000
Epoch8:loss = 0.120,result=3600.030,66.000,32.000,198.000
Epoch9:loss = 0.128,result=3600.128,68.000,30.000,204.000
Epoch10:loss = 0.124,result=3601.303,90.000,10.000,270.000

 Final Test: final result = 3601.303,90.000,10.000,270.000
```

2.12 출력문 수정을 위한 실행함수 재정의

이 함수는 abalone_exec()에서 정의한 함수.
run_test() 에서 호출하는 eval_accuracy() 함수에서 반환값 형식이 accuacy에서 네 가지 평가지표로 달라졌기 때문에 재정의 필요

```
def train_and_test(epoch_count, mb_size, report):  # 동일
    step_count = arrange_data(mb_size)
    test_x, test_y = get_test_data()

    for epoch in range(epoch_count):  # accuracy를 이미 앞에서 구축했기 때문에 따로 정확도를 저장하지 않음.
        losses = []

        for n in range(step_count):
            train_x, train_y = get_train_data(mb_size, n)  # 동일
            loss,_ = run_train(train_x,train_y)
            losses.append(loss)

        if report > 0 and (epoch+1) % report == 0:
            acc = run_test(test_x,test_y)
            acc_str = ','.join(['%5.3f']*4) % tuple(acc)    # 각 지표는 쉼표를 기준으로 구분하도록 하고,
                                                            # join 함수를 통해 형식, 어느 정도 크기 정보 전달,
            print('Epoch{}:loss = {:5.3f},result={}'.format(epoch+1,np.mean(losses),acc_str))
            # 4가지 평가지표 리스트 형식으로 그에 맞게 문자열 포멧팅 형식을 변경.
    acc = run_test(test_x, test_y)
    acc_str = ','.join(['%5.3f']*4) % tuple(acc)
    print('\n Final Test: final result = {}'.format(acc_str))
```

2.12 출력문 수정을 위한 실행함수 재정의_예제

.join() : 리스트에 특정 구분자를 추가하여 문자열로 변환

```
acc_str = ','.join(['%5.3f']*4) % tuple([1,2,3,4])
print(acc_str)
```

동일한 규격으로 설정

```
1.000,2.000,3.000,4.000
```

2.13 실행하기

```
pulsar_exec(adjust_ratio = False)
```

불규형한 종속변수의 분포

```
Epoch1:loss = 0.139,result=0.965,0.624,0.975,0.761
Epoch2:loss = 0.128,result=0.971,0.696,0.969,0.810
Epoch3:loss = 0.130,result=0.973,0.734,0.951,0.828
Epoch4:loss = 0.131,result=0.973,0.762,0.917,0.832
Epoch5:loss = 0.130,result=0.975,0.755,0.953,0.843
Epoch6:loss = 0.127,result=0.974,0.809,0.893,0.849
Epoch7:loss = 0.130,result=0.972,0.718,0.950,0.818
Epoch8:loss = 0.119,result=0.974,0.743,0.960,0.837
Epoch9:loss = 0.116,result=0.974,0.727,0.971,0.832
Epoch10:loss = 0.120,result=0.967,0.649,0.976,0.780

 Final Test: final result = 0.967,0.649,0.976,0.780
```

```
pulsar_exec(adjust_ratio = True)
```

균형잡힌 종속변수의 분포

```
Epoch1:loss = 0.405,result=0.923,0.902,0.939,0.920
Epoch2:loss = 0.376,result=0.918,0.848,0.983,0.910
Epoch3:loss = 0.377,result=0.904,0.934,0.879,0.906
Epoch4:loss = 0.364,result=0.915,0.839,0.987,0.907
Epoch5:loss = 0.378,result=0.925,0.905,0.941,0.923
Epoch6:loss = 0.367,result=0.923,0.859,0.982,0.916
Epoch7:loss = 0.369,result=0.917,0.844,0.986,0.909
Epoch8:loss = 0.363,result=0.927,0.910,0.940,0.925
Epoch9:loss = 0.349,result=0.926,0.867,0.981,0.920
Epoch10:loss = 0.378,result=0.915,0.919,0.909,0.914

 Final Test: final result = 0.915,0.919,0.909,0.914
```

```
 Final Test: final result = 0.967,0.649,0.976,0.780
```

정확도 감소 ↕

f1 수치 증가 ↕

```
Final Test: final result = 0.915,0.919,0.909,0.914
```

#정밀도와 재현율에는 약간의 편차 존재

1. 데이터간의 균형은 매우 중요하구나!

2. 무조건적인 단순비교 정확도 추출보다는 다양한 지표를 활용하자!