Chapter 7

Code Total

Ⅰ. Backpropagation <Toy model>

본 장은 간단한 Toy model에 대한 역전파를 구현할 것이다.
(참고 7_1_Backpropagation_in_Toy_Model)

1. Toy medel

• Toy model 정의

```
def fn(x, beta0, beta1, beta2, beta3, omega0, omega1, omega2, omega3):

  return beta3 + omega3 * np.cos(beta2 + omega2 * np.exp(beta1 + omega1
        * np.sin(beta0 + omega0 * x)))
```

$$f(x, \boldsymbol{\phi}) = \beta_3 + \omega_3 \cdot \cos[\beta_2 + \omega_2 \cdot \exp[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]]]$$

• 손실함수 정의

```
def loss(x, y, beta0, beta1, beta2, beta3, omega0, omega1, omega2,
          omega3):

  diff = fn(x, beta0, beta1, beta2, beta3, omega0, omega1, omega2
            , omega3) - y

  return diff * diff
```

$$\ell_i = (f(x, \boldsymbol{\phi}) - y_i)^2$$
$$= (\beta_3 + \omega_3 \cdot \cos[\beta_2 + \omega_2 \cdot \exp[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]]] - y_i)^2$$

• 파라미터 정의 후 손실함수 계산

```
beta0 = 1.0; beta1 = 2.0; beta2 = -3.0; beta3 = 0.4
omega0 = 0.1; omega1 = -0.4; omega2 = 2.0; omega3 = 3.0
x = 2.3; y =2.0

l_i_func = loss(x,y,beta0,beta1,beta2,beta3,omega0,omega1,omega2,omega3)
```

2. 도함수 계산

도함수에 대한 식을 손으로 계산하고 코드를 써서 직접 계산할 수도 있지만, 비교적 간단한 원래 방정식에 대해서도 일부는 매우 복잡한 표현을 가지고 있다. 도함수를 더 간단하게 계산하기 위해서는 아래와 같은 단계를 따라야 한다.

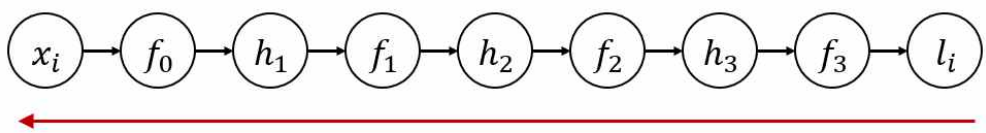
Step1 : Forward pass - f_k와 h_k의 값을 계산하고 저장함.

```
f0 = beta0 + omega0 * x
h1 = math.sin(f0)
f1 = beta1 + omega1 * h1
h2 = math.exp(f1)
f2 = beta2 + omega2 * h2
h3 = math.cos(f2)
f3 = beta3 + omega3 * h3
l_i = (f3 - y)**2
```

$$
\begin{aligned}
f_0 &= \beta_0 + \omega_0 \cdot x_i \\
h_1 &= \sin[f_0] \\
f_1 &= \beta_1 + \omega_1 \cdot h_1 \\
h_2 &= \exp[f_1] \\
f_2 &= \beta_2 + \omega_2 \cdot h_2 \\
h_3 &= \cos[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3 \\
\ell_i &= (f_3 - y_i)^2
\end{aligned}
$$

Step2 : Backward pass - ℓ_i의 도함수를 역순으로 계산

```
dldf3 = 2* (f3 - y)
dldh3 = omega3 * dldf3
dldf2 = -math.sin(f2) * dldh3
dldh2 = omega2 * dldf2
dldf1 = math.exp(f1) * dldh2
dldh1 = omega1 * dldf1
dldf0 = math.cos(f0) * dldh1
```



<Chain Rule>

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3}\left(\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial f_2} = \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial h_2} = \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial f_1} = \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial h_1} = \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial f_0} = \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

-> 이전에 사용한 도함수를 재 사용할 수 있다.

Step3 : Backward pass – 파라미터에 대한 손실의 도함수 계산

```
dldbeta3 = 2* (f3 - y)
dldomega3 = h3 * 2* (f3 - y)
dldbeta2 = dldf2
dldomega2 = h2 * dldf2
dldbeta1 = dldf1
dldomega1 = h1 * dldf1
dldbeta0 = dldf0
dldomega0 = x * dldf0
```

< k > 0 인 경우 >

$$\frac{\partial \ell_i}{\partial \beta_k} = \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k}, \quad \left(\frac{\partial f_k}{\partial \beta_k} = \frac{\partial}{\partial \beta_k}(\beta_k + \omega_k \cdot h_k) = 1\right)$$

$$= \frac{\partial \ell_i}{\partial f_k}$$

$$\frac{\partial \ell_i}{\partial \omega_k} = \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}, \quad \left(\frac{\partial f_k}{\partial \omega_k} = \frac{\partial}{\partial \omega_k}(\beta_k + \omega_k \cdot h_k) = h_k\right)$$

$$= h_k \cdot \frac{\partial \ell_i}{\partial f_k}$$

Ⅱ. Backpropagation

본 장에서는 파라미터을 업데이트 하기 위해 역전파 알고리즘을 이용하여 깊은 신경망의 파라미터에 대한 손실함수 도함수를 계산할 것이다. (참고 code_7_2_Backpropagation)

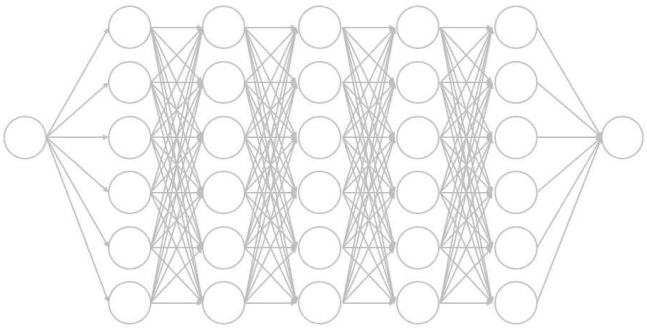
1. 신경망 구현

• **정의된 신경망 구조에 따라 초기 파라미터의값을 랜덤으로 설정**

```
np.random.seed(0)

K = 5
D = 6
D_i = 1
D_o = 1

# Initialize parameters
all_weights = [None] * (K+1)
all_biases = [None] * (K+1)
# [None, None, None, None, None, None]

# input layer
all_weights[0] = np.random.normal(size=(D, D_i))
all_biases[0] = np.random.normal(size =(D,1))

# output layer
all_weights[-1] = np.random.normal(size=(D_o, D))
all_biases[-1]= np.random.normal(size =(D_o,1))

# intermediate layer
for layer in range(1,K):
  all_weights[layer] = np.random.normal(size=(D,D))
  all_biases[layer] = np.random.normal(size=(D,1))
```

▶ Define neural network

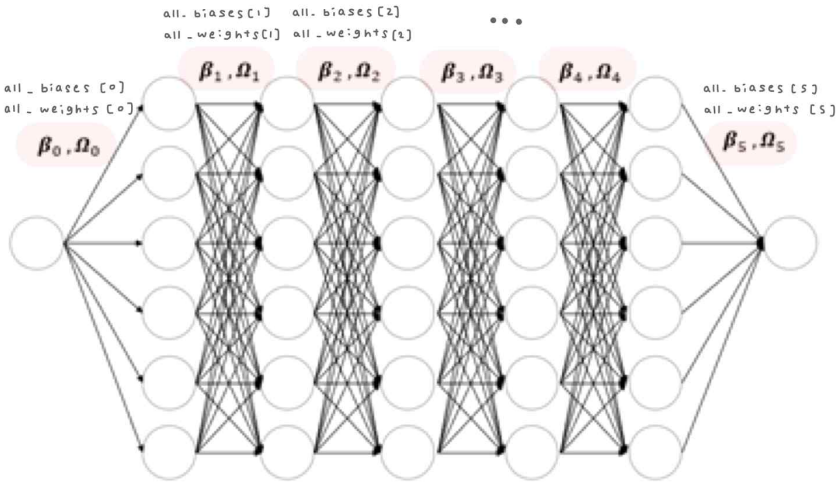
신경망은 은닉층 5개, 각 층마다의 뉴런 6개, 입력층 1개, 출력층 1개로 정의하였다.



▶ Initialize parameters

신경망의 각 층에 해당하는 가중치와 편향을 저장하기 위한 공간을 리스트로 저장하였다. 리스트는 요소들의 크기를 다양하게 저장할 수 있지만, array는 요소들의 크기를 다양하게 저장할 수 없다. 또한 가변적이며 유연하게 데이터를 조작하기 위해 리스트로 저장하였다.

```
all_biases = [None, None, None, None, None, None]
all_weights = [None, None, None, None, None, None]
```



▶ input layer ($f_0 = \beta_0 + \omega_0 h_0$)

ω_o는 입력의 가중치로 입력층에 영향을 받고, h_1의 사전 활성화 함수인 f_0에 전달되기 위해서는 (D, D_i) 행렬 만큼 필요하고, β_0는 f_0의 편향이고, h_1의 각 은닉유닛의 사전 활성화인 f_0에 전달되기 위해서는 (D, 1) 행렬 만큼 필요하다.

▶ ouput layer ($f_k = \beta_k + \omega_k h_k$)

ω_k는 h_k에서 f_k로 전달되는 가중치 즉, 출력의 가중치로 h_k(D, 1) 에 영향을 받음. f_k에 전달되기 위해서 (D_0, D) 행렬 만큼 필요하고, β_k는 f_k의 편향이고, f_k에 전달되기 위해서 (D_0, 1) 행렬 만큼 필요하다.

▶ intermediate layer ($f_i = \beta_i + \omega_i h_i$)

ω_i는 h_i에서 f_i로 전달되는 가중치로 h_i(D, 1) 에 영향을 받음. f_i(h_{i+1}의 사전 활성화 함수) 에 전달되기 위해서 (D, D) 행렬 만큼 필요하고, β_i는 f_i의 편향이고, f_i에 전달되기 위해서 (D, 1) 행렬 만큼 필요하다.

→ 랜덤 시드를 통해 위에서 정한 각 파라미터 크기만큼 랜덤 난수를 넣음.

2. Forward pass

위에서 정의한 파라미터를 이용하여 신경망의 순방향 계산을 할 것이다. 이 과정에서는 입력부터 출력까지의 계산을 단계적으로 수행하여 각 층의 활성화 값을 계산하는 코드이다.

• **ReLU Function 정의**

```
# ReLU function
def ReLU(preactivation):
  activation = preactivation.clip(0.0)
  return activation
```

▶ ReLU Function

정의한 ReLU Function에서 인자로 받는 preactivation 값이 0 이하일 때는 0으로, 양수일 때는 그 값을 그대로 출력하도록 함수 정의하였다.

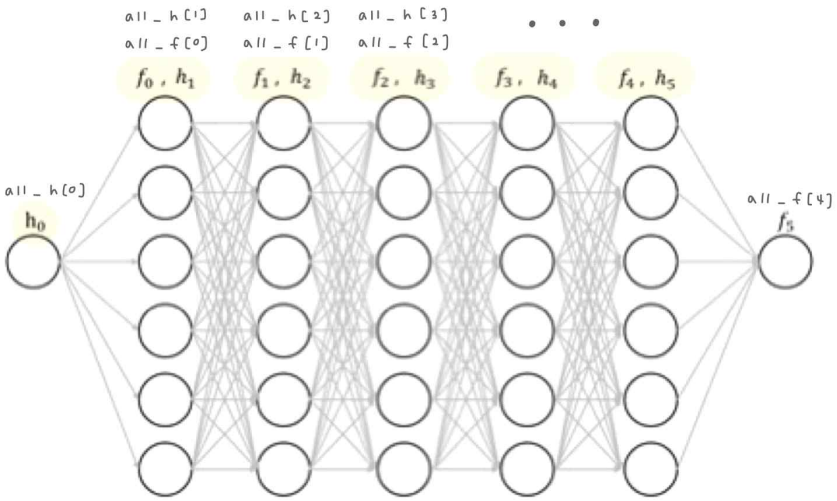
• **신경망의 순방향 계산**

```
def compute_network_output(net_input, all_weights, all_biases):

  # Initialize layer
  all_f = [None] * (K+1)
  all_h = [None] * (K+1)

  # input layer
  all_h[0] = net_input

  # intermediate layer
  for layer in range(K):  # 0~4
    all_f[layer] = all_biases[layer]
                  + np.matmul(all_weights[layer], all_h[layer])
    all_h[layer+1] = ReLU(all_f[layer])

  # Compute last hidden layer(output)
  all_f[K] = all_biases[K] + np.matmul(all_weights[K], all_h[K])
  net_output = all_f[K]

  return net_output, all_f, all_h
```

▶ Initialize layer

신경망의 각 층에 해당하는 사전 활성화 값과 활성화 값을 저장하기 위한 공간을 리스트로 저장하였다.

```
all_h = [None, None, None, None, None, None]
all_f = [None, None, None, None, None, None]
```



▶ input layer

입력층은 하나인 h_0는 net_input(신경망 입력)이다.

▶ intermediate layer

은닉층이 i번째 층일 때 (i = 0, 1, 2, 3, 4) - D=6, k=5

$$\underset{6\times1}{\boldsymbol{f}}[i] = \underset{6\times1}{\boldsymbol{\beta}}[i] + \underset{6\times6}{\Omega}[i] \cdot \underset{6\times1}{\boldsymbol{h}}[i]$$

$$\underset{6\times1}{\boldsymbol{h}}[i+1] = \mathrm{ReLU}(\underset{6\times1}{\boldsymbol{f}}[i])$$

▶ output layer

$$\underset{1\times1}{f_{k=5}} = \underset{1\times1}{\beta_5} + \underset{1\times6}{\Omega_5} \cdot \underset{6\times1}{\boldsymbol{h}_5}$$

3. Backward pass

위에서 계산한 단계별 활성화 전 값과 활성화 값을 이용하여 신경망의 역방향 계산을 할 것이다. 이 과정은 주어진 손실 함수에 대해 각 층의 가중치와 편향에 대한 손실의 도함수를 계산하는 코드이다.

• loss Function 정의와 출력에 대한 손심함수의 도함수 계산

```
# least squares loss function
def least_squares_loss(net_output, y):
  return np.sum((net_output-y) * (net_output-y))

# compute dloss/doutput
def d_loss_d_output(net_output, y):
  return 2 * (net_output - y)

# y와 loss 계산
y = np.ones((D_o,1)) * 20.0
loss = least_squares_loss(net_output, y)
```

• 신경망의 backward pass

```
def indicator_function(x):
  x_in = np.array(x)
  x_in[x_in>=0] = 1
  x_in[x_in<0] = 0
  return x_in

def backward_pass(all_weights, all_biases, all_f, all_h, y):
  # 손실에 대한 wiehgts, bias, f와 h 값의 미분 저장
  all_dl_dweights = [None] * (K+1)
  all_dl_dbiases = [None] * (K+1)
  all_dl_df = [None] * (K+1)
  all_dl_dh = [None] * (K+1)

  # 출력에 대한 손실의 도함수 계산
  all_dl_df[K] = np.array(d_loss_d_output(all_f[K],y))

  # Backward pass
  for layer in range(K,-1,-1):  # k(5)~0 (역방향으로 작동)
    all_dl_dbiases[layer] = all_dl_df[layer]
    all_dl_dweights[layer] = np.matmul(all_dl_df[layer], all_h[layer].T)
    all_dl_dh[layer] = np.matmul(all_weights[layer].T, all_dl_df[layer])

    if layer > 0:
      all_dl_df[layer-1] = all_dl_dh[layer] \
                           * indicator_function(all_f[layer-1])
return all_dl_dweights, all_dl_dbiases
```

```
all_dl_dweights, all_dl_dbiases \
            = backward_pass(all_weights, all_biases, all_f, all_h, y)
```

▶ indicator function

ReLU 함수의 도함수이므로 입력이 0보다 작을 땐 0으로 반환하고 0보다 클 때 1을 반환한다.

-> $\frac{\partial ReLU[x]}{\partial x}$

▶ 손실에 대한 미분값 저장

weights, bias, 사전 활성화 값, 활성화 값에 대한 손실함수의 도함수를 저장하기 위한 공간을 리스트로 저장하였다.

```
all_dl_dweights = [None, None, None, None, None, None]
  all_dl_biases = [None, None, None, None, None, None]
     all_dl_df  = [None, None, None, None, None, None]
     all_dl_dh  = [None, None, None, None, None, None]
```

▶ 출력(output)에 대한 손실함수의 도함수

all_dl_df[5] : $\frac{\partial \ell_i}{\partial f_5} = \frac{\partial}{\partial f_5}(f_5 - y)^2 = 2(f_5 - y)$

▶ Backward pass 자세한 과정은 < I . Backpropagaion_Toymodel > 참고

- all_dl_dbiases[i] : $\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_i} = \frac{\partial \ell_i}{\partial \mathbf{f}_i}$, k = 5, 4, 3, 2, 1, 0
- all_dl_dweights[i] : $\frac{\partial \ell_i}{\partial \Omega_i} = \frac{\partial \ell_i}{\partial \mathbf{f}_i}\mathbf{h}_i^T$
- all_dl_dh[i] : $\frac{\partial \ell_i}{\partial \boldsymbol{h}_i} = \frac{\partial \ell_i}{\partial \boldsymbol{f}_i}\Omega_i^T$
- all_dl_df[i-1]

-> $\frac{\partial \ell_i}{\partial \boldsymbol{f}_{k-1}} = \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{f}_{k-1}} \cdot \frac{\partial \ell_i}{\partial \boldsymbol{h}_k}$

$= \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{f}_{k-1}} \cdot \frac{\partial \boldsymbol{f}_k}{\partial \boldsymbol{h}_k} \cdot \frac{\partial \ell_i}{\partial \boldsymbol{f}_k}$

$$= \mathbb{I}[f_{k-1} > 0] \cdot \Omega_k^T \cdot \frac{\partial \ell_i}{\partial f_k}$$

$\frac{\partial h_k}{\partial f_{k-1}}$은 f_{k-1}에 대한 f_{k-1}의 활성화 함수의 도함수로 f_{k-1}가 0보다 작으면 0이 반환되고 0보다 크면 1이 반환된다. (indicator function 참고)

4. 유한차분법을 사용한 역전파 구현의 정확성 검증

신경망의 역전파 구현의 정확성을 유한차분법을 사용하여 검증할 것이다. 이를 통해 역전파로 계산된 기울기가 올바르게 계산되었는지 확인하는 코드이다.

- **유한 차분법(finite difference method)**

함수의 도함수를 근사하기 위해 사용하는 수치적 방법이다. 신경망에서 유한차분법은 주로 역전파 알고리즘이 제대로 작동하는지, 즉 기울기 계산이 정확한지 검증하는 데 사용된다. 밑은 유한 차분법으로 도함수를 구하는 공식이다.

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- **유한차분법을 하기 위한 사전 설정**

```
# 출력 설정
np.set_printoptions(precision=3)

# 유한차분법으로 계산된 파라미터에 대한 손실함수의 도함수 저장
all_dl_dweights_fd = [None] * (K+1)
all_dl_dbiases_fd = [None] * (K+1)

# 미세변화량 설정
delta_fd = 0.000001
```

▶ 출력 설정

출력 시 소수점 셋째 자리까지 출력하도록 설정했다.

▶ 유한차분법으로 계산한 파라미터 값 저장

유한차분법으로 계산된 파라미터에 대한 손실함수의 도함수를 저장할 공간을 리스트를 지정했다.

▶ 미세변화량 설정

기울기를 근사하기 위해 사용할 작은 변화량 ϵ을 설정했다. 이 값은 보통 매우 작은 값으로 설정 해야한다.

- **바이어스(β)의 기울기 검정**

```
for layer in range(K):

  # 현재 층의 바이어스의 도함수 저장 공간 설정
  dl_dbias  = np.zeros_like(all_dl_dbiases[layer])
```

```
    for row in range(all_biases[layer].shape[0]):

      # 현재 바이어스의 변화
      all_biases_copy = [np.array(x) for x in all_biases]
      all_biases_copy[layer][row] += delta_fd

      # 신경망 계산
      network_output_1, *_ = compute_network_output
                             (net_input, all_weights, all_biases_copy)
      network_output_2, *_ = compute_network_output
                             (net_input, all_weights, all_biases)

      # 유한차분법을 이용한 도함수 계산
      dl_dbias[row] = (least_squares_loss(network_output_1, y)
                     - least_squares_loss(network_output_2,y))/delta_fd


      all_dl_dbiases_fd[layer] = np.array(dl_dbias)

    # 결과 출력 및 비교
    print("-------------------------------------------------")
    print("Bias %d, derivatives from backprop:"%(layer))
    print(all_dl_dbiases[layer])
    print("Bias %d, derivatives from finite differences"%(layer))
    print(all_dl_dbiases_fd[layer])

    if np.allclose(all_dl_dbiases_fd[layer],all_dl_dbiases[layer],
                   rtol=1e-05, atol=1e-08, equal_nan=False):
      print("Success!  Derivatives match.")
    else:
      print("Failure!  Derivatives different.")
```

▶ 현재 층 바이어스의 도함수 저장 공간 설정

0 ~ K-1(4)만큼 단계적으로 반복을 하고, 현재 층(해당 K)의 dl_dbias를 현재 층의 all_dl_dbiases의 배열 크기만큼 저장 공간을 설정했다.

▶ 현재 바이어스의 변화

- for row in range(all_biases[layer].shape[0]):
  현재 층의 바이어스의 각 요소에 대해 반복하는 코드이다.
  (예. all_dbiases[0].shape -> (6,1) / all_dbiases[0].shape[0] -> 6)
  ex) b[1, 2]와 b[1][2]는 같음
- all_biases_copy
  현재의 바이어스(all_biases)의 배열을 그대로 복사하고, 현재 층(K)과 현재 요소(row)에 해당되는 바이어스 요소에 작은 변화량(delta_fd) 추가했다. 즉, 현재 바이어스의 모든 요소에 작은 변화량을 준 변수이다.

▶ 신경망 계산

network_output_1은 all_biases_copy를 가지고 신경망을 계산한 출력값이고 network_output_2는 all_biases를 가지고 신경망을 계산한 출력값이다.

▶ 유한 차분법을 이용한 도함수 계산

해당 층, 해당 요소의 기울기를 구하여 저장한 변수이다. 이때 기울기는 유한 차분법을 이용하여 계산하였다.

▶ all_dl_dbiases_fd

위에서 구한 현재 층(해당 K)의 바이어스 기울기 값을 저장한 배열 dl_dbias을 all_dl_dbiases_fd의 k번째 리스트에 저장한다.

▶ 결과 출력 및 비교

현재 층의 변화량을 주지 않은 바이어스의 도함수 배열과 변화량을 준 바이어스의 도함수 배열을 프린트 하고 만약 all_dl_dbiases[layer] 값과 all_dl_dbiases_fd[layer]값이 허용 오차내에서 근접하면 "Success! Derivatives match." 아니면 "Failure! Derivatives different." 반환

※ np.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)

a : 첫 번째 입력 배열.

b : 두 번째 입력 배열.

rtol : 상대 허용 오차 (relative tolerance). 기본값은 1e-05.

$\frac{|a-b|}{|b|}$ 두 배열의 상대적인 차이를 의미함. 이는 비교 대상 값의 크기에 비례하여 오차를 평가함.

atol : 절대 허용 오차 (absolute tolerance). 기본값은 1e-08.

$|a-b|$ 두 배열의 절대적인 차이를 의미함. 이는 비교 대상의 값의 크기에 상관없이 일정한 값을 기준으로 비교함.

equal_nan : True로 설정하면 NaN 값들도 서로 같다고 간주함. 기본값은 False.

두 배열의 요소들이 모두 주어진 허용 오차 범위 내에서 가까운지 여부를 확인하는 데 사용하는 함수이며, 두 배열의 모든 요소가 허용 오차 내에서 가까우면 True, 그렇지 않으면 False를 반환함.

- **가중치(Ω)의 기울기 검정**

가중치의 기울기 검정도 바이어스의 기울기 검정과 같게 진행하였다.

Ⅲ. Initialization

본 장은 파라미터의 초기 설정을 β_i는 0으로 초기화 하고, Ω_{ij}는 평균이 0이고 분산이 σ_Ω^2인 정규분포로 초기화를 하고, 역전파 알고리즘을 이용하여 파라미터에 대한 손실함수의 도함수 계산하고 이때 발생하는 기울기 문제를 확인할 것이다. (참고 7_3_Initialization)

1. 신경망 계산을 위한 함수 지정

• **파라미터 초기화 함수**

```
def init_params(K, D, sigma_sq_omega):

  np.random.seed(0)
  # 입력층 및 출력층
  D_i = 1
  D_o = 1
  # 가중치와 바이어스의 리스트 생성
  all_weights = [None] * (K+1)
  all_biases = [None] * (K+1)
  # 입력층과 출력층의 가중치 및 바이어스 초기화
  # np.random.normal: 평균이 0이고 표준편차가 1인 정규분포
  # 표준편차는 np.sqrt(sigma_sq_omega)
  all_weights[0] = np.random.normal(size=(D, D_i))
                                          * np.sqrt(sigma_sq_omega)
  all_weights[-1] = np.random.normal(size=(D_o, D))
                                          * np.sqrt(sigma_sq_omega)
  all_biases[0] = np.zeros((D,1))
  all_biases[-1]= np.zeros((D_o,1))
  # 은닉층의 가중치 및 바이어스 초기화
  for layer in range(1,K):
    all_weights[layer] = np.random.normal(size=(D,D))
                                          * np.sqrt(sigma_sq_omega)
    all_biases[layer] = np.zeros((D,1))

return all_weights, all_biases
```

<Ⅱ. Backpropagation >의 파라미터 초기와 코드와 유사하지만 β_i는 0으로 초기화 하고, Ω_{ij}는 평균이 0이고 분산은 sigma_sq_omega인 정규분포로 초기화를 하였다. 또한, 이번 코드는 K(신경망 층)을 인자로 받아서 신경망 층 수에 맞게 파라미터 값을 초기화 하였다.

• **neural network 계산 함수**

```
def ReLU(preactivation):
  activation = preactivation.clip(0.0)
  return activation
```

```
def compute_network_output(net_input, all_weights, all_biases):

  K = len(all_weights) -1

  all_f = [None] * (K + 1)
  all_h = [None] * (K + 1)

  all_h[0] = net_input

  for layer in range(K):

    all_f[layer] = all_biases[layer]
                          + np.matmul(all_weights[layer], all_h[layer])
    all_h[layer+1] = ReLU(all_f[layer])

  all_f[K] = all_biases[K] + np.matmul(all_weights[K], all_h[K])

  net_output = all_f[K]

return net_output, all_f, all_h
```

위에서 지정한 파라피터 값을 이용하여 신경망을 계산하였다.
자세한 과정은 <Ⅱ. Backpropagaion - 2 > 참고

2. Initialization for forward pass

• neural network 계산

```
# 신경망 구조
K = 5
D = 8
D_i = 1
D_o = 1

# 초기 분산
sigma_sq_omega = 1.0

# Initialize parameters
all_weights, all_biases = init_params(K,D,sigma_sq_omega)

# 입력값 지정
n_data = 1000
data_in = np.random.normal(size=(1,n_data))

# 신경망 계산
net_output, all_f, all_h = compute_network_output
                                    (data_in, all_weights, all_biases)
```

```
# 각 은닉층의 표준편차 비교
for layer in range(K):
print("Layer %d, std of hidden units = %3.3f“
                                        %(layer, np.std(all_h[layer])))
```

▶ 신경망 구조

은닉층 5개(K=5), 은닉뉴런 8개(D=8), 입력층 1개(Di=1), 출력층 1개(D0=1)로 지정하였다.

▶ 초기 분산

초기 분산은 가중치를 $N(0, \sigma_\Omega^2)$의 정규분포로 초기화하였기 때문에 분산을 1.0으로 지정하여 가중치를 $N(0, 1)$의 정규분포에서 랜덤하게 추출할 것이다.

▶ Initialize parameter

위에서 신경망 구조, 초기분산을 가지고 파라미터를 초기화 하여 랜덤하게 추출하였다.

- all_weights

  입력층: (D, D_i) → (8, 1)

  중간층: (D, D) → (8, 8)

  출력층: (D_0, D) → (1, 8)

- all_biases

  입력층: (D, 1) → (8, 1)

  중간층: (D, 1) → (8, 1)

  출력층: (D_0, 1) → (1, 1)

▶ 입력값 지정

입력값을 (1, 1000)형태의 행렬로 지정하였다. 즉, 1개의 입력층에 1000개의 데이터 포인트가 있는 형태이다.

▶ 신경망 계산

$$\underset{8\times1000}{\boldsymbol{f}_0} = \underset{8\times1}{\boldsymbol{\beta}_0} + \underset{8\times1}{\boldsymbol{\Omega}_0} \cdot \underset{1\times1000}{\boldsymbol{x}}$$

- $\boldsymbol{\beta}_0$와 $\boldsymbol{\Omega}_0 \cdot \boldsymbol{x}$은 연산이 불가하지만, 브로드 캐스팅(몇쪽 첨부) 규칙 2번에 따라 $\boldsymbol{\beta}_0$가 (8,1000)으로 확장되어 연산이 가능해진다.

$$\underset{8\times1000}{\boldsymbol{f}_i} = \underset{8\times1}{\boldsymbol{\beta}_i} + \underset{8\times8}{\boldsymbol{\Omega}_i} \cdot \underset{8\times1000}{\boldsymbol{h}_i}$$

$$\underset{1\times1000}{\boldsymbol{f}_k} = \underset{1\times1}{\boldsymbol{\beta}_k} + \underset{1\times8}{\boldsymbol{\Omega}_k} \cdot \underset{8\times1000}{\boldsymbol{h}_k}$$

- 신경망 계산을 통해 최종 출력은 (1, 1000) 형태로 출력된다.

▶ 각 은닉층의 표준편차 비교

for 문을 통해 신경망의 각 은닉층에서 활성화 값의 표준편차를 계산하고 출력한다. 이를 통해 신경망의 학습 과정에서 각 층의 출력이 어떻게 변화하는지를 평가할 수 있다.

```
>>> Layer 0, std of hidden units = 0.981
    Layer 1, std of hidden units = 0.811
    Layer 2, std of hidden units = 1.472
    Layer 3, std of hidden units = 4.547
    Layer 4, std of hidden units = 8.896
```

- 네트워크를 통과할수록 표준편차가 점점 커지는 것을 볼 수 있다. 이는 *exploding gradient* 의 문제로 판단된다.

2-1. 초기화 문제 - forward pass

TO DO

1. 층 당 80개의 은닉유닛이 있는 50개의 층으로 변경해라
2. sigma_sq_omega를 조절하여 forward pass 계산의 분산이 폭발하는 것을 막아라

층 당 80개의 은닉유닛을 갖는 50개의 은닉 층으로 변경했을 때 네트워크를 통과할수록 표준편차가 무수히 커지는 것을 보면 e*xploding gradient*의 문제가 생겼다고 볼 수 있다.

```
Layer 0, std of hidden units = 0.981
Layer 1, std of hidden units = 0.622
Layer 2, std of hidden units = 3.108
Layer 3, std of hidden units = 21.075
Layer 4, std of hidden units = 161.638
Layer 5, std of hidden units = 1125.582
Layer 6, std of hidden units = 6319.072
Layer 7, std of hidden units = 37275.665
Layer 8, std of hidden units = 243387.814
Layer 9, std of hidden units = 1339835.231
Layer 10, std of hidden units = 7366234.399
Layer 11, std of hidden units = 49006173.785
Layer 12, std of hidden units = 272845366.658
```

이를 해결하기 위해서 He 초기화 기법을 이용할 것이다.

• **He 초기화를 활용한 forward pass**

```
def he_init_params(K, D):
  np.random.seed(0)

  D_i = 1
  D_o = 1

  all_weights = [None] * (K+1)
  all_biases = [None] * (K+1)
```

```
    all_weights[0] = np.random.normal(size=(D, D_i)) * np.sqrt(2/ D_i)
    all_weights[-1] = np.random.normal(size=(D_o, D)) * np.sqrt(2/ D)
    all_biases[0] = np.zeros((D,1))
    all_biases[-1]= np.zeros((D_o,1))

    for layer in range(1,K):
      all_weights[layer] = np.random.normal(size=(D,D)) * np.sqrt(2/ D)
      all_biases[layer] = np.zeros((D,1))

  return all_weights, all_biases
```

▶ 가중치의 초기화

$\sigma_{\Omega}^{2} = \frac{2}{D_h}$ -> D_h 는 가중치가 적용되기 전 층 의 차원이다.

때문에 입력층과 첫 번째 은닉층 사이의 가중치 행렬을 $\frac{2}{D_i}$으로 초기화 하고, 마지막 은닉층과 출력층 사이의 가중치 행렬을 $\frac{2}{D}$로 초기화 하였다. 그리고 은닉층과 은닉층 사이의 가중치 행렬도 마찬가지로 $\frac{2}{D}$로 초기화 하였다. 그 후 Ⅱ-2 방식으로 신경망을 계산하고 각 은닉층의 표준편차를 비교해봤다.

▶ 각 은닉층의 표준편차 비교

은닉층 50, 은닉뉴런 80으로 이루어진 신경망의 가중치를 He 초기화를 활용하여 파라미터를 초기화해서 계산한 신경망의 각 은닉층에서 활성화 값의 표준편차의 값은 밑과 같이 비교적 안정적인 표준편차가 나왔다.

```
Layer 0, std of hidden units = 0.981
Layer 1, std of hidden units = 0.886
Layer 2, std of hidden units = 0.698
Layer 3, std of hidden units = 0.741
Layer 4, std of hidden units = 0.899
Layer 5, std of hidden units = 0.995
Layer 6, std of hidden units = 0.886
Layer 7, std of hidden units = 0.826
Layer 8, std of hidden units = 0.851
Layer 9, std of hidden units = 0.737
Layer 10, std of hidden units = 0.646
Layer 11, std of hidden units = 0.677
Layer 12, std of hidden units = 0.597
```

3. Initialization for backward pass

• backward pass 함수 정의

```
def indicator_function(x):
  x_in = np.array(x)
  x_in[x_in>=0] = 1
  x_in[x_in<0] = 0
return x_in

def backward_pass(all_weights, all_biases, all_f, all_h, y):

  all_dl_dweights = [None] * (K+1)
  all_dl_dbiases = [None] * (K+1)

  all_dl_df = [None] * (K+1)
  all_dl_dh = [None] * (K+1)

  all_dl_df[K] = np.array(d_loss_d_output(all_f[K],y))

  for layer in range(K,-1,-1):

    all_dl_dbiases[layer] = np.array(all_dl_df[layer])
    all_dl_dweights[layer] = np.matmul(all_dl_df[layer], all_h[layer].T)
    all_dl_dh[layer] = np.matmul(all_weights[layer].T, all_dl_df[layer])

    if layer > 0:
      all_dl_df[layer-1] = indicator_function(all_f[layer-1])
                                                * all_dl_dh[layer]

return all_dl_dweights, all_dl_dbiases, all_dl_dh, all_dl_df
```

▶ 자세한 코드 설명은 <Ⅱ.Backpropagation – 3. backpass ward > 참고

• 신경망 구조에 따른 backward pass를 통한 각 층의 미분 값

```
# 신경망 구조
K = 5
D = 8
D_i = 1
D_o = 1
sigma_sq_omega = 1.0

# 파라미터 초기화
all_weights, all_biases = init_params(K,D,sigma_sq_omega)
```

```
# 각 층의 사전 활성화 함수에 대한 손실함수의 미분값 저장
n_data = 100
aggregate_dl_df = [None] * (K+1)

for layer in range(1,K): # 1~4 만큼 반복
  aggregate_dl_df[layer] = np.zeros((D,n_data))


# 데이터 포인트 수 만큼 반복
for c_data in range(n_data): # 0~99 만큼 반복

  data_in = np.random.normal(size=(1,1))
  y = np.zeros((1,1))

  net_output, all_f, all_h = compute_network_output
                                (data_in, all_weights, all_biases)

  all_dl_dweights, all_dl_dbiases, all_dl_dh, all_dl_df =
           backward_pass(all_weights, all_biases, all_f, all_h, y)

  for layer in range(1,K):
    aggregate_dl_df[layer][:,c_data] = np.squeeze(all_dl_df[layer])

# 각 층의 미분값에 대한 손실함수의 도함수의 표준편차 비교
for layer in range(1,K):
  print("Layer %d, std of dl_dh = %3.3f“
                   %(layer, np.std(aggregate_dl_df[layer].ravel())))
```

▶ 신경망 구조

은닉층 5개, 각 층의 뉴런 수 8개, 입력층 1개, 출력층 1개, 초기 가중치 분산 값은 1로 설정하여 신경망의 구조를 정의하였다.

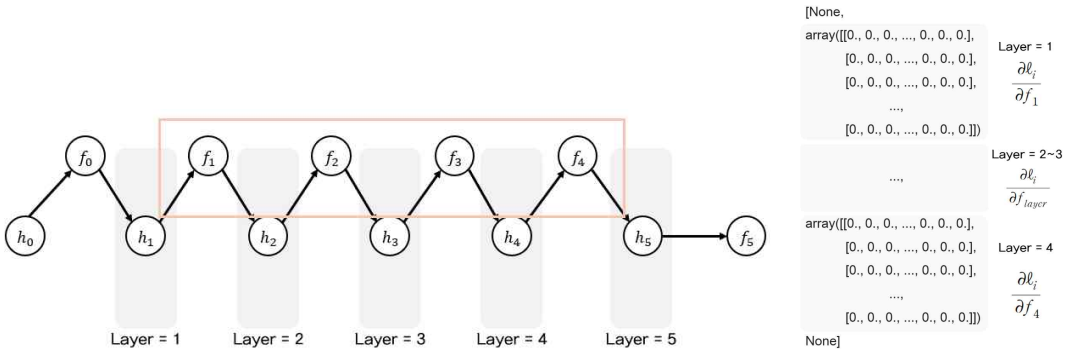
▶ 파라미터 초기화

위에서 정의한 신경망 구조와 초기 가중치 값으로 파라미터를 초기화하였다.

▶ 각 층의 사전 활성화 함수에 대한 손실함수의 미분값 저장

1개의 입력층에 100개의 데이터 포인트를 지정하였고, 각 층의 사전 활성화 함수에 대한 손실함수의 미분값을 저장하기 위한 리스트를 aggregate_dl_df 로 지정 하였다.

단순화를 위해 첫 번째 은닉층과 마지막 은닉층 사이의 파라미터의 기울기만 고려를 할 것이다.



aggregate_dl_df는 은닉뉴런수가 8개고 데이터 포인터가 100개 이기 때문에 층별로 (8, 100) 만큼 공간 확보하였다.

▶ 데이터 포인터 수 만큼 반복

모든 데이터 포인트에 대해 동시에 네트워크의 그래디언트를 계산하기 위해 각 데이터 포인트에 대한 매개 변수의 도함수를 개별적으로 계산 해야한다.

1)

for 문을 통해 n_data만큼(0~99) 반복문을 만들어서 data_in을 정규 분포를 따르는 난수(하나의 데이터 포인트)로 초기화하고, y는 0(스칼라)으로 초기화하여 신경망을 계산하고, 파라미터에 대한 손실함수의 미분을 계산했다.

2)

또한 for문을 통해 layer 1~4 까지 반복하여 aggregate_dl_df 리스트에 있는 layer 번째 각 층의 c_data열에 all_dl_df[layer](c_data로 계산된 도함수 값)의 값을 열벡터로 저장한다. 이때 c_data는 데이터 포인트의 인덱스를 나타내며, 해당 데이터 포인트에 대한 도함수 값을 저장할 열의 인덱스로 사용된다.

이를 통해 모든 데이터 포인트에 대한 그래디언트를 한 번에 계산했다.

▶ 각 층의 사전 활성화 값 대한 손실함수의 도함수 표준편차 비교

```
>>>  Layer 1, std of dl_dh = 446.654
     Layer 2, std of dl_dh = 340.657
     Layer 3, std of dl_dh = 109.132
     Layer 4, std of dl_dh = 56.472
```

3-1. 초기화 문제 - Backward pass

TO DO

> 1. 층 당 80개의 은닉유닛이 있는 50개의 층으로 변경해라
> 2. sigma_sq_omega를 조절하여 도함수의 분산이 폭발하는 것을 막아라

층 당 80개의 은닉유닛을 갖는 50개의 은닉 층으로 변경했을 때 각 층에서 계산된 손실 함수의 미분 값의 표준 편차가 매우 커서 그레디언트 폭발이 나타났다.

```
Layer 1, std of dl_dh = 3864161615668244381461267510373904423054174106385627368088294218476908925943808.000
Layer 2, std of dl_dh = 6442974627408047134425819000266806338082499927324410975720219504344175151677 44.000
Layer 3, std of dl_dh = 1162565785064209495300348366930262783326207759062014229095636577550399753420 80.000
Layer 4, std of dl_dh = 1801240688308829057482062480059250614667726002506238644877431697728425112371 2.000
Layer 5, std of dl_dh = 2657382923492155237517743551772839757190473680951583847484196418047631687680.000
Layer 6, std of dl_dh = 4371155742182451938079088378987767650753499333007811746446175975380006993 92.000
Layer 7, std of dl_dh = 7228942568902238369495992250213394896648163318088503669840271552689052057 6.000
Layer 8, std of dl_dh = 1022904557936987705989316058556701151511423010164593474160760302146368307 2.000
Layer 9, std of dl_dh = 1622769590787357221165998426801933037786842056906651889376730795250548736.000
Layer 10, std of dl_dh = 2916772668608704377802325761901884432415144886978681186752010450202787 84.000
Layer 11, std of dl_dh = 4704092500482571009257735041289515545332156369752601850448921647854387 2.000
Layer 12, std of dl_dh = 7477794785556511029375003658625364545843316026986261679990214882230272.000
```

이를 해결하기 위해서 He 초기화 기법을 이용할 것이다.

- He 초기화를 활용한 Backward pass

```python
def re_init_params(K, D):

  np.random.seed(0)

  D_i = 1
  D_o = 1

  all_weights = [None] * (K+1)
  all_biases = [None] * (K+1)

  all_weights[0] = np.random.normal(size=(D, D_i)) * np.sqrt(2/ D)
  all_weights[-1] = np.random.normal(size=(D_o, D)) * np.sqrt(2/ D_o)
  all_biases[0] = np.zeros((D,1))
  all_biases[-1]= np.zeros((D_o,1))

  for layer in range(1,K):
    all_weights[layer] = np.random.normal(size=(D,D)) * np.sqrt(2/ D)
    all_biases[layer] = np.zeros((D,1))

  return all_weights, all_biases
```

▶ 가중치의 초기화

$\sigma_{\Omega}^{2} = \frac{2}{D_{h'}}$ -> $D_{h'}$ 는 가중치가 적용된 후 층의 차원이다.

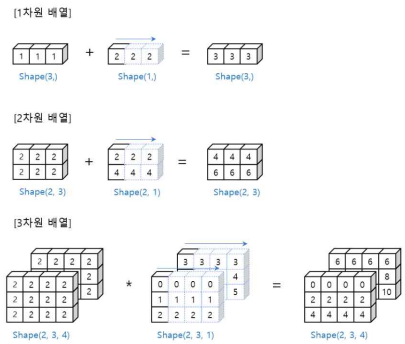
때문에 입력층과 첫 번째 은닉층 사이의 가중치 행렬을 $\frac{2}{D}$으로 초기화 하고, 마지막은닉층과 출력층 사이의 가중치 행렬을 $\frac{2}{D_0}$로 초기화 하였다. 그리고 은닉층과 은닉층 사이의 가중치 행렬도 마찬가지로 $\frac{2}{D}$로 초기화 하였다. 그 후 Ⅱ-3 방식으로 신경망을 계산하고 각 은닉층의 표준편차를 비교해봤다.

▶ 각 은닉층의 표준편차 비교

은닉층 50, 은닉뉴런 80으로 이루어진 신경망의 가중치를 He 초기화를 활용하여 파라미터를 초기화해서 계산한 신경망의 각 은닉층에서 활성화 값의 표준편차의 값은 밑과 같이 비교적 안정적인 표준편차가 나왔다.

```
Layer 1, std of dl_dh = 2.439
Layer 2, std of dl_dh = 2.572
Layer 3, std of dl_dh = 2.935
Layer 4, std of dl_dh = 2.876
Layer 5, std of dl_dh = 2.683
Layer 6, std of dl_dh = 2.791
Layer 7, std of dl_dh = 2.920
Layer 8, std of dl_dh = 2.613
Layer 9, std of dl_dh = 2.622
Layer 10, std of dl_dh = 2.980
Layer 11, std of dl_dh = 3.040
Layer 12, std of dl_dh = 3.056
```

※ 브로드 캐스팅(Broadcasting)



산술연산이 되는 배열(array)에서 모양(shape)이 다른 경우에도, 연산이 가능하도록 배열들의 모양을 처리하는 방법을 의미한다. 즉, 일정 조건을 부합하는 다른 형태의 배열끼리 연산을 수행하게 해준다.

1. 원소가 하나인 배열은 어떤 배열이나 브로드캐스팅이 가능
   - (4, 4) + 1

```
[34]  Value = np.array([[1,2,3,4],[2,5,6,7],[8,9,10,11],[12,13,14,15]])

      Value1 = np.array([1])

      Value + Value1
```

```
array([[ 2,  3,  4,  5],
       [ 3,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

2. 하나의 배열이 1차원 배열인 경우, 브로드캐스팅이 가능
   - (4, 4) + (1, 4) - 열/행 둘 중 하나의 차원이 1이여야 함.

```
[35]  Value = np.array([[1,2,3,4],[2,5,6,7],[8,9,10,11],[12,13,14,15]])

      Value2 = np.array([[3,3,3,3]])

      Value + Value2
```

```
array([[ 4,  5,  6,  7],
       [ 5,  8,  9, 10],
       [11, 12, 13, 14],
       [15, 16, 17, 18]])
```

3. 차원의 짝이 맞을때 브로드캐스팅이 가능
   - (1, 4) + (4, 1)

```
[36]  Value = np.array([[1,2,3,4],[2,5,6,7],[8,9,10,11],[12,13,14,15]])

      Value3 = np.array([[4],[5],[6],[7]])

      Value + Value3
```

```
array([[ 5,  6,  7,  8],
       [ 7, 10, 11, 12],
       [14, 15, 16, 17],
       [19, 20, 21, 22]])
```