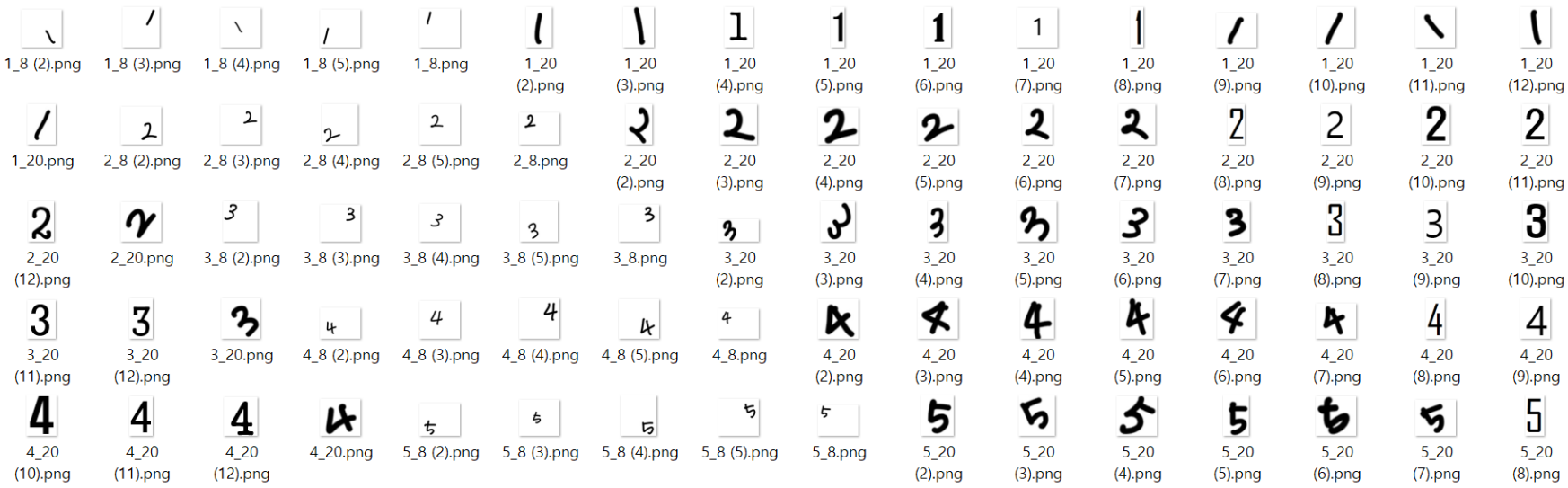
인공지능 프로젝트

작성일시	@2022년 11월 24일 오후 2:08
강의 번호	
유형	
복습	☐
자료	
텍스트	

▼ Image Labeling

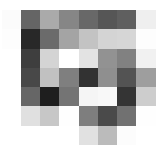
- 데이터 셋은 각자 생성하여 공유할 것




```
import numpy as np
import os
import cv2
import re
import csv

path_dir = 'C:/Users/YongJun/To Student 2022/ProjectSource2'      # 저장되어 있는 경로 (모여있는 공간)
path_dir_20x20 = 'C:/Users/YongJun/To Student 2022/Data_20'    # 20x20이 저장될 경로
path_dir_8x8 = 'C:/Users/YongJun/To Student 2022/Data_8'    # 8x8이 저장될 경로
file_list = os.listdir(path_dir)
file_list_png = [file for file in file_list if file.endswith(".png")]

for i in range((np.shape(file_list_png)[0])):
    src = cv2.imread(path_dir+'/'+file_list_png[i],cv2.IMREAD_GRAYSCALE)

    scr_resized = cv2.resize(src,dsize = (20,20), interpolation=cv2.INTER_AREA)
    img_split_str = re.split('[_| |.]',file_list_png[i])

    imarr = np.array(scr_resized)
    labeling = np.ones((1, 20)) * 255

    labeling[0, 0] = img_split_str[0]  # 1,2,3,4,5 중에 하나
    labeling[0, 1] = img_split_str[1]  # 20 혹은 8

    final = np.vstack([labeling, imarr])

    if(img_split_str[1] == '20'):
        cv2.imwrite(path_dir_20x20+'/'+ img_split_str[0] + '_' +img_split_str[1] + '_' +str(i%12) +'.png',final)
    elif(img_split_str[1] == '8'):
        cv2.imwrite(path_dir_8x8+'/'+ img_split_str[0] + '_' + img_split_str[1] + '_' +str(i%5) +'.png',final)
```

- 손글씨 / 폰트체 로 데이터를 만들고 python cv2 모듈을 통한 Resize (보간법 : INTER_AREA)

  - 21x20 이미지에 첫번째 행의
    1번 픽셀은 정답 정보 (1,2,3,4,5)
    2번 픽셀은 size 정보가 담겨져있다 (8,20)
  - 학습을 위한 데이터 가공시 첫번째 을 Next함수로 무시하게 된다.



▼ Image To CSV

▼ Image_to_csv.py

```
import csv
import os
import cv2
import numpy as np

path_dir_20x20 = 'C:/Users/YongJun/AI_TMP/Data_20'    # 20x20이 저장된 경로
path_dir_8x8 = 'C:/Users/YongJun/AI_TMP/Data_8'    # 8x8이 저장된 경로

file_list_20x20 = os.listdir(path_dir_20x20)
file_list_8x8 = os.listdir(path_dir_8x8)

file_list_png_20x20 = [file for file in file_list_20x20 if file.endswith(".png")]
file_list_png_8x8 = [file for file in file_list_8x8 if file.endswith(".png")]

f_20x20 = open('./20x20_info.csv', 'w', encoding='utf-8', newline='')
f_8x8 = open('./8x8_info.csv', 'w', encoding='utf-8', newline='')
wr_20x20 = csv.writer(f_20x20)
wr_8x8 = csv.writer(f_8x8)
attribute = (["pixel" + str(j) for j in range(1,401)])
# attribute.append("format")
attribute.append("isOne")
attribute.append("isTwo")
attribute.append("isThree")
attribute.append("isFour")
attribute.append("isFive")

#print(attribute)

wr_20x20.writerow(attribute)
wr_8x8.writerow(attribute)

instance = list()
for i in range(np.shape(file_list_png_20x20)[0]):
    src = cv2.imread(path_dir_20x20+'/'+file_list_png_20x20[i], cv2.IMREAD_GRAYSCALE)
    for j in range(1,21):
        for k in range(0,20):
            instance.append(src[j,k])
    #instance.append(src[0,1]) # format
    if src[0,0] == 1: instance.extend([1,0,0,0,0])
    if src[0,0] == 2: instance.extend([0,1,0,0,0])
    if src[0,0] == 3: instance.extend([0,0,1,0,0])
    if src[0,0] == 4: instance.extend([0,0,0,1,0])
    if src[0,0] == 5: instance.extend([0,0,0,0,1])

    wr_20x20.writerow(instance)
    instance.clear()

for i in range(np.shape(file_list_png_8x8)[0]):
    src = cv2.imread(path_dir_8x8+'/'+file_list_png_8x8[i], cv2.IMREAD_GRAYSCALE)
    for j in range(1,21):
        for k in range(0,20):
            instance.append(src[j,k])
    #instance.append(src[0,1]) # format
    if src[0,0] == 1: instance.extend([1,0,0,0,0])
    if src[0,0] == 2: instance.extend([0,1,0,0,0])
    if src[0,0] == 3: instance.extend([0,0,1,0,0])
    if src[0,0] == 4: instance.extend([0,0,0,1,0])
    if src[0,0] == 5: instance.extend([0,0,0,0,1])

    wr_8x8.writerow(instance)
    instance.clear()
```

- 21x20 이미지를 첫번째 행은 무시하고, 1x400 으로 형태로 csv파일에 입력
- 정답 정보 (1,2,3,4,5) 은 (1 0 0 0 0) (0 1 0 0 0) … (0 0 0 0 1) 형태로 구분되어 저장된다.

▼ 실험환경(참조코드)

- 해당 과제는 기본적으로 선택분류의 성질을 가지고 있음(정답이 이진이 아닌 여러개의 상태)
- 따라서 기반코드는 챕터1의 코드에 부분적으로 챕터3의 코드를 사용하였고, 추가로 다층 구조에 대한 실험을 위해 챕터4의 코드를 사용하였다.

```
# chapter 1
import numpy as np
import csv
import time
np.random.seed(1234)

f = open('test_result.csv','w+',encoding='utf-8', newline='')
wr = csv.writer(f)

def randomize(): np.random.seed(time.time())
```

- Chpater1.
  모듈/라이브러리 import 및 가중치 부여를 위한 Random 난수 생성함수 정의

```
# chapter 1
RND_MEAN = 0     #정규분포 난숫값 평균
RND_STD = 0.0030     #정규분포 난숫값 표준편차
LEARNING_RATE = 0.001     #학습율
```

- Chapter1.
  가중치 초기화를 위한 정규분포평균, 표준편차, 학습을 위한 학습률 정의
  초기 가중치의 변화에 따라 차이를 알아보기 위해 수정될 수 있음.

```
# chapter 1
# 실험용 메인 함수
def NUM_exec(epoch_count=10, mb_size=10, report=1):
    load_NUM_dataset()
    init_model()
    train_and_test(epoch_count, mb_size, report)
```

- Chapter1.
  실험용 메인함수
  전체적은 코드의 진행에 해당한다.
  epoch_count 와, 미니배치 사이즈, 리포트 주기를 파라미터로 받고있으며
  1. csv로 이루어진 데이터셋을 학습 부분과 테스트 부분을 구분 짓는다.
  2. 초기 가중치에 대한 설정을 하기위해 init_model을 호출한다.
  3. 이후, 파라미터로 입력한 변수 그대로 학습/테스트 를 위해 train_and_test 함수를 호출한다.

```
# chapter3
def load_NUM_dataset():
    with open('./data/20x20_info.csv') as csvfile:
        csvreader = csv.reader(csvfile)
        next(csvreader, None)     # 첫행을 읽지 않고 건너뛴다
        rows = []
        for row in csvreader:
            rows.append(row)

    global data, input_cnt, output_cnt
    input_cnt, output_cnt = 400, 5  # 20x20 img dataset 형식에 맞춤
    data = np.asarray(rows, dtype='float32')
```

- Chapter3.
  데이터 적재 함수.
  - csv로 변환된 이미지 파일을 불러오며 Attribute Name으로 이루어진 첫행을 건너 띄게 된다.

- 이 후, 20x20 img data 형식에 맞게 정보부분(Input_cnt), 정답 부분(out_cnt) 를 정의한다.
- csv를 읽을 땐, List형태로 이루어지기 때문에 data 변수에 asarray() 함수를 사용,
계산에 용이한 Numpy 형식으로 Copy 하게 된

```
# chapter4
def set_hidden(info):
    global hidden_cnt, hidden_config
    if isinstance(info, int):
        hidden_cnt = info
        hidden_config = None
    else:
        hidden_config = info
```

- Chapter4.
  은닉 계층구조의 형태를 지정하는 함수이다.
  사용자는 NUM_exec() 를 호출하기 전 set_hidden() 함수를 통해 은닉계층의 구조를 설정해주어야 한다.

```
# Chapter 4
# hidden_config는 단층/다층 을 구분하기 위한 전역변수 (set_hidden() 함수에서 정의됨)
global hidden_config

def init_model():
    # 값이 설정되어 있으면
    if hidden_config is not None:
        print('은닉 계층 {}개를 갖는 다층 퍼셉트론이 작동되었습니다.'. \
              format(len(hidden_config)))
        init_model_hiddens()

    # 설정 되어있지 않으면
    else:
        print('은닉 계층 하나를 갖는 다층 퍼셉트론이 작동되었습니다.')
        init_model_hidden1()

def forward_neuralnet(x):
    if hidden_config is not None:
        return forward_neuralnet_hiddens(x)
    else:
        return forward_neuralnet_hidden1(x)

def backprop_neuralnet(G_output, hiddens):
    if hidden_config is not None:
        backprop_neuralnet_hiddens(G_output, hiddens)
    else:
        backprop_neuralnet_hidden1(G_output, hiddens)
```

- Chapter4.
  사용자가 설정한 신경망 구조(다층/단층) 를 구분하기 위한 Switch 함수이다.
  hidden_config는 전역변수로 선언되어 있으며
  hidden_config에 저장된 값의 유무에 조건문을 통해 수행하는 함수가 다르게 된다.

```
# chapter1
def train_and_test(epoch_count, mb_size, report):
    step_count = arrange_data(mb_size) # 데이터를 섞고, 학습/테스트 용 데이터셋을 분리
    global test_x, test_y
    test_x , test_y = get_test_data()

    for epoch in range(epoch_count):     # epoch만큼 학습 반복
        losses, accs = [], []            # Report를 위한 리스트형 변수

        for n in range(step_count):     # 값 만큼 미니배치 처리
            train_x, train_y = get_train_data(mb_size, n)
            # train_x, train_y 는 학습이 진행될 데이터셋과 정답 데이터셋

            loss, acc = run_train(train_x, train_y) # 학습을 진행할 run_train 함수 호출
            losses.append(loss)
            accs.append(acc)

        if report > 0 and (epoch+1) % report == 0:
            acc = run_test(test_x, test_y)
            print('Epoch {}: loss={:5.3f}, accuracy={:5.3f}/{:5.3f}'. \
                  format(epoch+1, np.mean(losses), np.mean(accs), acc))
```

```
    final_acc = run_test(test_x, test_y) # final_acc는 마지막 정확도만 존재
    print('\nFinal Test: final accuracy = {:5.3f}'.format(final_acc))
    f.close()
```

- Chapter1.
  불러온 데이터셋을 이용해 학습/테스트 진행
  더 나은 학습을 위해 DataSet을 재배열 하게 된다.
  외곽 반복문은 지정한 Epoch수 만큼 학습을 진행하겠다는 의미
  내각 반복문은 설정한 미니배치(덩어리) 만큼 처리하겠다는 뜻이다.
  get_train_data() 를 통해 학습부분과 정답 부분을 반환받으며, 보고를 위한 리스트에 지속 저장한다.
  학습이 끝난 이후, 곧바로 run_test() 함수를 통해 테스트 함수가 진행된다.

```
# Chapter 1
def arrange_data(mb_size):
    global data, shuffle_map, test_begin_idx
    shuffle_map = np.arange(data.shape[0]) # data 는 load() 함수에서 만든 넘파이 배열
    np.random.shuffle(shuffle_map)
    step_count = int(data.shape[0] * 0.8) // mb_size   # 미니배치 학습에 필요한 배치 수
    test_begin_idx = step_count * mb_size    # 테스트 데이터셋의 시작 부분을 정의한다.
    return step_count

def get_test_data():
    global data, shuffle_map, test_begin_idx, output_cnt
    test_data = data[shuffle_map[test_begin_idx:]]
    return test_data[:, :-output_cnt], test_data[:, -output_cnt:]

def get_train_data(mb_size, nth):
    global data, shuffle_map, test_begin_idx, output_cnt
    if nth == 0:
        np.random.shuffle(shuffle_map[:test_begin_idx]) # 매 Epoch마다 순서를 섞음
    train_data = data[shuffle_map[mb_size*nth:mb_size*(nth+1)]]     #미니배치 구간의 위치를 따져 그 구간에 해당하는 shuffle_map만 가져옴
    return train_data[:, :-output_cnt], train_data[:, -output_cnt:]
```

- Chapter1.
  - arrange_data() : 원할한 학습을 위해 데이터셋의 순서를 뒤섞는다.
  이후, 학습 데이터셋 과 테스트 데이터셋을 구분하기 위해 인덱스를 설정하게 된다.
  - get_test_data() : arrange_data() 에서 인덱싱한 데스트데이터셋의 경계를 통해 하나의 데이터셋을 추출해낸다. 이후, Input_cnt 와 output_cnt를 통해 입력 벡터와 정답벡터를 반환한다.
  - get_train_data() : 매 Epoch마다 순서를 뒤섞으며 진행, 미니 배치 사이즈를 통해 위치를 반영하여 그 구간에 해당하는 shuffle_map만 가져온다. 이후, 입력벡터와 정답벡터를 반환한다.

```
# Chapter 1
def run_train(x, y):
    output, aux_nn = forward_neuralnet(x)      #순전파 처리
    loss, aux_pp = forward_postproc(output, y) #순전파 후처리
    accuracy = eval_accuracy(output, y)        #정확도 계산
    #aux_pp, aux_nn은 순전파를 통해 얻은 값중 일부를 역전파 함수에 전달
    #불필요한 계산을 줄여 효율을 높임
    G_loss = 1.0 # 손실 기울기 1
    G_output = backprop_postproc(G_loss, aux_pp)
    backprop_neuralnet(G_output, aux_nn)

    return loss, accuracy

def run_test(x, y):
    output, _ = forward_neuralnet(x)
    accuracy = eval_accuracy(output, y)
    return accuracy
```

- Chapter1.
  학습 함수와 Test 함수 정의
  run_train() : 순전파 → 순전파 처리 (마지막 계층에 대한) → 정확도 계산 → 역전파 처리(마지막 계층에 대한) → 역전파 로 이루어 진다.
  run_test() : 순전파만 진행하며 결과에 따른 정확도만 측정한다.

```
# chapter 3
# 후처리 과정에 대한 순전파와 역전파 함수의 재정의
```

```
def forward_postproc(output, y):
    # output은 로짓값
    # 로짓값 -> 소프트맥스 교차 엔트로피를 -> loss 연산
    entropy = softmax_cross_entropy_with_logits(y, output)
    loss = np.mean(entropy)
    return loss, [y, output, entropy]

def backprop_postproc(G_loss, aux):
    y, output, entropy = aux
    #평균 loss 편미분
    g_loss_entropy = 1.0 / np.prod(entropy.shape)
    #엔트로피 함수 역산 편미분
    g_entropy_output = softmax_cross_entropy_with_logits_derv(y, output)

    G_entropy = g_loss_entropy * G_loss
    G_output = g_entropy_output * G_entropy
    #손실 기울기
    return G_output
```

- Chapter3.
  후 처리 과정을 대한 순전파/역전파 함수 재정의
  - 소프트맥스 교차 엔트로피를 통해 순전파로 구해진 값을 확률값으로 변경, 확률 분포 하나로부터 엔트로피 값 구함
  - Output_cnt에 따라 소프트맥스 교차 엔트로피는 여러 형태로 나타날수 있음
  - 프로젝트에선 (미니배치 사이즈, output_cnt) 형태로 이루어 진다.

  역전파의 postproc 의 경우 순전파의 역산이기 때문에 편미분을 통해 구하게 된다.

```
# Chapter 3
# 소프트맥스 관련 함수 정의
def softmax(x)
    max_elem = np.max(x, axis=1)
    diff = (x.transpose() - max_elem).transpose()
    exp = np.exp(diff)
    sum_exp = np.sum(exp, axis=1)
    probs = (exp.transpose() / sum_exp).transpose()
    return probs

def softmax_derv(x, y):
    mb_size, nom_size = x.shape
    derv = np.ndarray([mb_size, nom_size, nom_size])
    for n in range(mb_size):
        for i in range(nom_size):
            for j in range(nom_size):
                derv[n, i, j] = -y[n,i] * y[n,j]
            derv[n, i, i] += y[n,i
    return derv

# 로그 폭주를 막기 위한 상수 더하기
def softmax_cross_entropy_with_logits(labels, logits):
    probs = softmax(logits)
    return -np.sum(labels * np.log(probs+1.0e-10), axis=1)

def softmax_cross_entropy_with_logits_derv(labels, logits):
    return softmax(logits) - labels
```

- Chapter3.
  소프트맥스 함수는 출력계층에서 확률 을 나타내게 된다.
  5개의 숫자(1,2,3,4,5) 에서 선택분류를 하기 때문에 구현된다.
  각 행에서 최대항을 골라 자연상주 처리를 하여 구현된다.

```
# Chapter4
def init_model_hidden1():
    global pm_output, pm_hidden, input_cnt, output_cnt, hidden_cnt


    pm_hidden = alloc_param_pair([input_cnt, hidden_cnt])

    pm_output = alloc_param_pair([hidden_cnt, output_cnt])

def alloc_param_pair(shape):

    weight = np.random.normal(RND_MEAN, RND_STD, shape)
    #print(shape)
```

```
    bias = np.zeros(shape[-1])
    return {'w':weight, 'b':bias}
```

- 은닉 계층 하나를 위한 파라미터 생성 함수 정의
  두 쌍의 파라미터를 저장하고 hidden_cnt라는 파라미터를 전달 받음

```
# Chapter 4
def forward_neuralnet_hidden1(x
    global pm_output, pm_hidden


    hidden = relu(np.matmul(x, pm_hidden['w']) + pm_hidden['b'])

    output = np.matmul(hidden, pm_output['w']) + pm_output['b']


    return output, [x, hidden]

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

- Chapter 4.
  은닉 계층 하나를 위한 순전파 함수를 정의한다.
  Chapter4를 기반으로 작성되었기 때문에 Relu 함수를 활성화 함수로 사용하였다.
  은닉 계층하나를 거쳐 출력되는 최종 출력 Output 을 구하게 된다.

```
# Chapter 4
def backprop_neuralnet_hidden1(G_output, aux):

    global pm_output, pm_hidden


    x, hidden = aux


    #
    g_output_w_out = hidden.transpose()
    G_w_out = np.matmul(g_output_w_out, G_output)
    G_b_out = np.sum(G_output, axis=0)

    #
    g_output_hidden = pm_output['w'].transpose()

    G_hidden = np.matmul(G_output, g_output_hidden)


    pm_output['w'] -= LEARNING_RATE * G_w_out
    pm_output['b'] -= LEARNING_RATE * G_b_out


    G_hidden = G_hidden * relu_derv(hidden)


    g_hidden_w_hid = x.transpose()
    G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
    G_b_hid = np.sum(G_hidden, axis=0)

    # 은닉층 역전파
    pm_hidden['w'] -= LEARNING_RATE * G_w_hid
    pm_hidden['b'] -= LEARNING_RATE * G_b_hid


def relu_derv(y):
    return np.sign(y)
```

- Chapter 4.
  히든레이어 하나에 대한 역전파를 진행한다.

```
# Chapter4
def init_model_hiddens():

    global pm_output, pm_hiddens, input_cnt, output_cnt, hidden_config

    pm_hiddens = []
```

```
    prev_cnt = input_cnt

    for hidden_cnt in hidden_config:
        pm_hiddens.append(alloc_param_pair([prev_cnt, hidden_cnt]))
        prev_cnt = hidden_cnt

    pm_output = alloc_param_pair([prev_cnt, output_cnt])
```

- Chapter 4.
  파라미터에 대한 은닉계층을 구성하는 함수
  리스트 성분의 개수는 계층의 수와 같으며, 리스트의 성분은 계층별 노드의 수와 같다.
  반복문을 통해 prev_cnt를 갱신, 은닉층의 벡터크기에 맞게 갱신한다.

```
# Chapter4
def forward_neuralnet_hiddens(x):
    global pm_output, pm_hiddens
    hidden = x

    # input들이 정리된 리스트
    hiddens = [x]

    for pm_hidden in pm_hiddens:
        hidden = relu(np.matmul(hidden, pm_hidden['w']) + pm_hidden['b'])
        hiddens.append(hidden)

    output = np.matmul(hidden, pm_output['w']) + pm_output['b']

    return output, hiddens
```

- Chapter 4.
  은닉 계층 구성을 위한 순전파 함수를 정의한다.
  계층 하나에 대한 순전파를 구현한 함수 foward_neuralnet_hidden1 함수를 반복한다.
  활성화 함수는 Relu함수를 사용하며, pm_hidden의 가중치 성분과 hidden 의 행렬곱을 하며, pm_hidden의 바이어스 성분을 더해준

```
#chapter 4

def backprop_neuralnet_hiddens(G_output, aux):
    global pm_output, pm_hiddens

    hiddens = aux


    g_output_w_out = hiddens[-1].transpose()
    G_w_out = np.matmul(g_output_w_out, G_output)
    G_b_out = np.sum(G_output, axis=0)


    g_output_hidden = pm_output['w'].transpose()
    G_hidden = np.matmul(G_output, g_output_hidden)


    pm_output['w'] -= LEARNING_RATE * G_w_out
    pm_output['b'] -= LEARNING_RATE * G_b_out


    for n in reversed(range(len(pm_hiddens))):

        G_hidden = G_hidden * relu_derv(hiddens[n+1])

        g_hidden_w_hid = hiddens[n].transpose()
        G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
        G_b_hid = np.sum(G_hidden, axis=0)

        g_hidden_hidden = pm_hiddens[n]['w'].transpose()
        G_hidden = np.matmul(G_hidden, g_hidden_hidden)

        pm_hiddens[n]['w'] -= LEARNING_RATE * G_w_hid
        pm_hiddens[n]['b'] -= LEARNING_RATE * G_b_hid
```

- Chapter 4.
  은닉 계층 구성을 위한 역전파 함수를 정의한다.

▼ 실험을 통한 최적의 신경망 구조 찾아보기

▼ 1. 20x20 에 그려진 숫자

- **Sigmoid**

▼ Code

> 💡 Chapter 3/4 기반의 소스는 Relu함수를 사용하였다.
> 시그모이드를 적용하기 위해 함수를 활성화 함수가 이용되는 함수를 재 정의 하였다.

```
def sigmoid(x):
    return 1/1+np.exp(-x)
def sigmoid_derv(y):
    return sigmoid(y) * (1 - (1/sigmoid(y)))
```

```
# Chapter 4
# 은닉 계층 하나를 위한 순전파 함수 정의
def forward_neuralnet_hidden1(x):
    # init_model_hidden1 쌍의 파라미터에 접근
    global pm_output, pm_hidden

    #입력 x와 pm_hidden을 통해 은닉계층 출력 계산 Relu함수 사용
    hidden = sigmoid(np.matmul(x, pm_hidden['w']) + pm_hidden['b'])
    #hidden과 pm_output을 통해 출력계층 출력이자 최종출력 output 계산
    output = np.matmul(hidden, pm_output['w']) + pm_output['b']

    #출력 계층의 역전파 처리 때 가중치에 대한 편미분 정보로 hidden이 필요함
    return output, [x, hidden]
```

```
# Chapter 4
# 은닉 계층 하나를 위한 역전파 함수 정의
def backprop_neuralnet_hidden1(G_output, aux):
    # G_output은 역전파 후처리 후만들어지는 파라미터
    global pm_output, pm_hidden

    # 순전파에 사용됐던 input
    x, hidden = aux

    # 출력층에 대한 역전파
    # 출력층에서 사용하는 input은 hidden
    g_output_w_out = hidden.transpose()
    G_w_out = np.matmul(g_output_w_out, G_output)
    G_b_out = np.sum(G_output, axis=0)

    # 출력 계층과 은닉 계층 역전파 처리 매개하는 G_output으로부터 G_hidden을 구해내는 과정
    g_output_hidden = pm_output['w'].transpose()
    # pm_output['w']은 변환이 되는 계수이기 때문에 미리 기록한 것 위에 명시된 식에서 W_2
    G_hidden = np.matmul(G_output, g_output_hidden)
    print(G_w_out)
    # 출력층 역전파
    pm_output['w'] -= LEARNING_RATE * G_w_out
    pm_output['b'] -= LEARNING_RATE * G_b_out

    # 은닉층 -> relu -> 출력층 순서
    # 역전파는 그 반대로 가야하기 때문에 Relu 부분을 처리해줘야 함 위 식에서 편미분값
    G_hidden = G_hidden * sigmoid_derv(hidden)

    # 은닉층에 대한 역전파
    # 은닉층에서 사용하는 input은 x
    g_hidden_w_hid = x.transpose()
    G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
    G_b_hid = np.sum(G_hidden, axis=0)

    # 은닉층 역전파
    pm_hidden['w'] -= LEARNING_RATE * G_w_hid
    pm_hidden['b'] -= LEARNING_RATE * G_b_hid
```

▼ Result

> 💡 시그모이드 적용 전, 행렬곱의 합이 너무 크거나 작으면 문제가 생긴다.

```
4644  [[-0.10104006 -0.0037068   0.11268632  0.18762316 -0.19556262]]
4645  [[ 7.90995426e-02  1.76664602e-01  2.33912846e+29 -2.33912846e+29
4646     8.40650119e-02]]
4647  [[-0.1 -0.4 -0.4  1.  -0.1]]
4648  [[nan nan nan nan nan]]
4649  [[nan nan nan nan nan]]
```

```
>>> sigmoid(-2.33912846e+29)
inf
```

- 에폭 진행중 Output 시그모이드 함수 적용전, matmul 의 값을 print한 출력문이다. sigmoid 함수는 입력파라미터가 너무 큰 음수면 무한대를 출력해 버린다.

→ 다른 활성화함수를 사용해 보자.

- **하이퍼볼릭 탄젠트**

$$\tanh x = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$$= \frac{(e^{x}-e^{-x})\cdot e^{-x}}{(e^{x}+e^{-x})\cdot e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}}$$

$$= \frac{2-(1+e^{-2x})}{1+e^{-2x}}$$

$$= \frac{2}{1+e^{-2x}} - 1 = 2\,\mathrm{sigmoid}(2x) - 1$$

▼ Code

```
# 활성화 함수 정의
def sigmoid(x):
    return 1/1+np.exp(-x)

def tanh(x):
    return 2 * sigmoid(2*x) - 1

def tanh_derv(y):
    return 1 - np.power(tanh(y),2)
```

```
# Chapter 4
# 은닉 계층 하나를 위한 순전파 함수 정의
def forward_neuralnet_hidden1(x):
    # init_model_hidden1 쌍의 파라미터에 접근
    global pm_output, pm_hidden

    #입력 x와 pm_hidden을 통해 은닉계층 출력 계산 Relu함수 사용
    hidden = tanh(np.matmul(x, pm_hidden['w']) + pm_hidden['b'])
    #hidden과 pm_output을 통해 출력계층 출력이자 최종출력 output 계산
    output = np.matmul(hidden, pm_output['w']) + pm_output['b']

    #출력 계층의 역전파 처리 때 가중치에 대한 편미분 정보로 hidden이 필요함
    return output, [x, hidden]
```

```
# Chapter 4
# 은닉 계층 하나를 위한 역전파 함수 정의
def backprop_neuralnet_hidden1(G_output, aux):
    # G_output은 역전파 후처리 후만들어지는 파라미터
    global pm_output, pm_hidden

    # 순전파에 사용됐던 input
    x, hidden = aux
```

```
# 출력층에 대한 역전파
# 출력층에서 사용하는 input은 hidden
g_output_w_out = hidden.transpose()
G_w_out = np.matmul(g_output_w_out, G_output)
G_b_out = np.sum(G_output, axis=0)

# 출력 계층과 은닉 계층 역전파 처리 매개하는 G_output으로부터 G_hidden을 구해내는 과정
g_output_hidden = pm_output['w'].transpose()
# pm_output['w']은 변환이 되는 계수이기 때문에 미리 기록한 것 위에 명시된 식에서 W_2
G_hidden = np.matmul(G_output, g_output_hidden)

# 출력층 역전파
pm_output['w'] -= LEARNING_RATE * G_w_out
pm_output['b'] -= LEARNING_RATE * G_b_out

# 은닉층 -> relu -> 출력층 순서
# 역전파는 그 반대로 가야하기 때문에 tanh 부분을 처리해줘야 함 위 식에서 편미분값
G_hidden = G_hidden * tanh_derv(hidden)

# 은닉층에 대한 역전파
# 은닉층에서 사용하는 input은 x
g_hidden_w_hid = x.transpose()
G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
G_b_hid = np.sum(G_hidden, axis=0)

# 은닉층 역전파
pm_hidden['w'] -= LEARNING_RATE * G_w_hid
pm_hidden['b'] -= LEARNING_RATE * G_b_hid
```

▼ Result

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4a1d97c1-c7a9-4743-aeb5-88ebc59d20e0/1___ETA___STD___tanh___Normal.txt

```
Epoch 950: loss=1.608, accuracy=0.212/0.151
Epoch 960: loss=1.608, accuracy=0.212/0.151
Epoch 970: loss=1.608, accuracy=0.212/0.151
Epoch 980: loss=1.608, accuracy=0.212/0.151
Epoch 990: loss=1.608, accuracy=0.212/0.151
Epoch 1000: loss=1.608, accuracy=0.212/0.151

Final Test: final accuracy = 0.151
Final Epoch : 1000 Running Time : 15.003068685531616
```

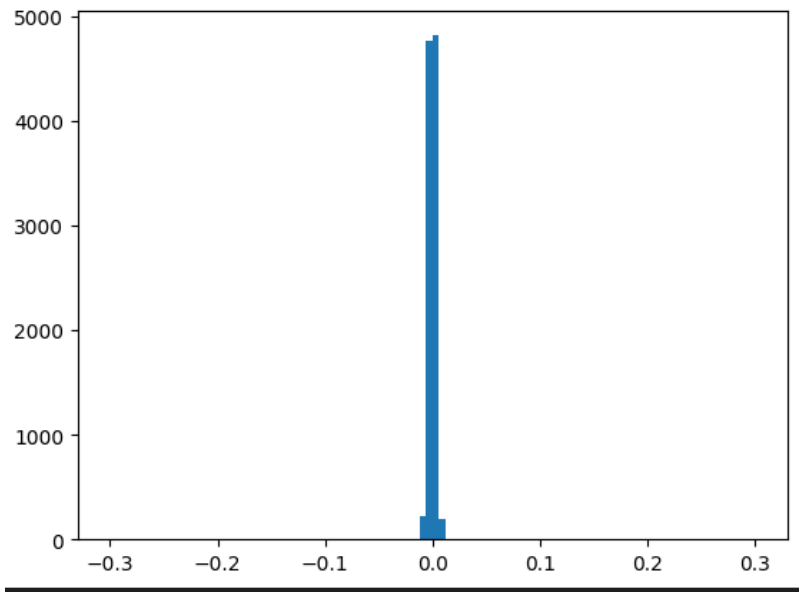
- 하나의 계층의 하나의 노드수일 때의 결과, 학습이 전혀 진행되지 않는다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/fc3b9af3-ad77-4bce-95a1-cc23d187d439/32___ETA___STD___tanh___Normal.txt

```
Epoch 950: loss=1.609, accuracy=0.208/0.168
Epoch 960: loss=1.609, accuracy=0.208/0.168
Epoch 970: loss=1.609, accuracy=0.208/0.168
Epoch 980: loss=1.609, accuracy=0.208/0.168
Epoch 990: loss=1.609, accuracy=0.208/0.168
Epoch 1000: loss=1.609, accuracy=0.208/0.168

Final Test: final accuracy = 0.168
Final Epoch : 1000 Running Time : 18.224365234375
```

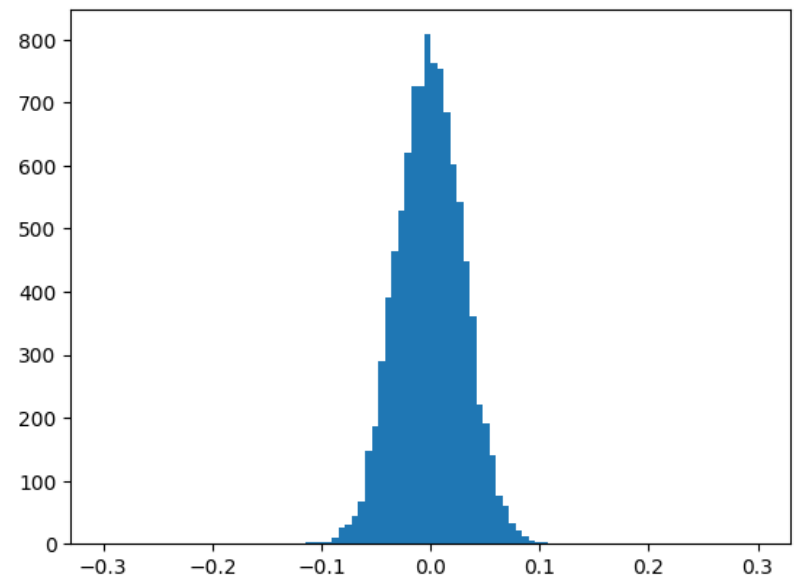
- 계층 수는 유지한 채 노드수만 증가시켜 보았다.
- 여전히 학습이 진행되지 않고 있다.
- → 가중치 초기화 단계에서 표준편차가 너무 작아 한곳에 몰려서 그런것이 아닐까?

```
RND_MEAN = 0
RND_STD = 0.0030
arr_normal = np.random.normal(RND_MEAN,RND_STD,10000)
plt.hist(arr_normal,range=(-0.3,0.3),bins=100)
plt.show()
```



- 표준편차 0.0030 의 그래프이다.
- 0~0.03 까지의 너무 400개의 가중치가 너무 몰려있다는것을 알수 있다.

```
RND_MEAN = 0
RND_STD = 0.0030 * 10
arr_normal = np.random.normal(RND_MEAN,RND_STD,10000)
plt.hist(arr_normal,range=(-0.3,0.3),bins=100)
plt.show()
```



- Default 표준편차 0.0030 을 x 10 하였다.
- 훨씬더 많은 영역으로 가중치가 분포될 것이다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/f5685daa-526a-4e43-ac26-37ac380cfeb8/32___ETA___STD_x_10___tanh___Normal.txt

```
Epoch 950: loss=1.604, accuracy=0.214/0.162
Epoch 960: loss=1.604, accuracy=0.214/0.162
Epoch 970: loss=1.604, accuracy=0.214/0.162
Epoch 980: loss=1.604, accuracy=0.214/0.162
Epoch 990: loss=1.604, accuracy=0.214/0.162
Epoch 1000: loss=1.604, accuracy=0.214/0.162

Final Test: final accuracy = 0.162
Final Epoch : 1000 Running Time : 18.50136685371399
```

- 정규분포를 더 넓게 설정하였지만 학습에는 큰 영향을 주지 않았다.
- → 다른 요인이 문제인가?
- → Learning Rate를 변경해보았다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/96cb0d25-02df-44ae-9379-2d09074518f1/32___ETA_x_0.1___STD_x_10___tanh___Normal.txt

```
# 단층구조 Node : 1
LEARNING_RATE = 0.001 * 0.1
RND_MEAN = 0
RND_STD = 0.0030 * 10
Layer_node_info = [32]
```

```
file_set('./Result/tanh20/%s _ ETA x 0.1 _ STD x 10 _ tanh _ Normal'%Layer_node_info)
set_hidden(Layer_node_info)
NUM_exec(epoch_count=1000, report=10)
f.close()
```

```
은닉 계층 1개를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 10: loss=0.893, accuracy=0.670/0.584
Epoch 20: loss=0.666, accuracy=0.757/0.622
Epoch 30: loss=0.537, accuracy=0.796/0.632
Epoch 40: loss=0.438, accuracy=0.834/0.659
Epoch 50: loss=0.355, accuracy=0.864/0.670
Epoch 60: loss=0.278, accuracy=0.891/0.697
Epoch 70: loss=0.229, accuracy=0.922/0.659
Epoch 80: loss=0.191, accuracy=0.923/0.654
Epoch 90: loss=0.164, accuracy=0.945/0.686
Epoch 100: loss=0.131, accuracy=0.957/0.703
Epoch 110: loss=0.115, accuracy=0.966/0.703
Epoch 120: loss=0.078, accuracy=0.980/0.692
Epoch 130: loss=0.073, accuracy=0.980/0.708
Epoch 140: loss=0.057, accuracy=0.989/0.735
Epoch 150: loss=0.055, accuracy=0.986/0.735
Epoch 160: loss=0.043, accuracy=0.995/0.714
Epoch 170: loss=0.036, accuracy=0.996/0.724
Epoch 180: loss=0.034, accuracy=0.993/0.719
Epoch 190: loss=0.030, accuracy=0.995/0.719
```

```
Epoch 950: loss=0.001, accuracy=1.000/0.730
Epoch 960: loss=0.001, accuracy=1.000/0.730
Epoch 970: loss=0.001, accuracy=1.000/0.730
Epoch 980: loss=0.001, accuracy=1.000/0.730
Epoch 990: loss=0.001, accuracy=1.000/0.730
Epoch 1000: loss=0.001, accuracy=1.000/0.730

Final Test: final accuracy = 0.730
Final Epoch : 67 loss_zero time : Time : 1.616436243057251
```

- 학습이 진행되었다.
- 로스함수의 값은 점점 줄었으며
- 정답률또한 점차 증가하다가 수렴하는 꼴을 보였다.

- 1000 Epoch 동안 진행시 정답률은 0.730 에 수렴하였으며,
- 첫 Loss가 설정한 값 0.005 보다 작게 나타난 시점은 67 Epoch 단계였고,
  그 때의 수행시간은 1.61 s 였습니다.
- → 비교를 위해 노드수를 더 증가하여 보았다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/77f83d26-7718-4893-80c4-fcb9c196acb3/128___ETA_x_0.1___STD_x_10___tanh___Normal.txt

```
은닉 계층 1개를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 10: loss=0.589, accuracy=0.784/0.670
Epoch 20: loss=0.354, accuracy=0.882/0.708
Epoch 30: loss=0.276, accuracy=0.904/0.719
Epoch 40: loss=0.177, accuracy=0.945/0.741
Epoch 50: loss=0.120, accuracy=0.972/0.719
Epoch 60: loss=0.091, accuracy=0.972/0.751
Epoch 70: loss=0.068, accuracy=0.986/0.762
Epoch 80: loss=0.042, accuracy=0.993/0.746
Epoch 90: loss=0.034, accuracy=0.999/0.751
Epoch 100: loss=0.023, accuracy=0.999/0.762
Epoch 110: loss=0.018, accuracy=1.000/0.778
Epoch 120: loss=0.014, accuracy=1.000/0.768
Epoch 130: loss=0.013, accuracy=1.000/0.778
Epoch 140: loss=0.011, accuracy=1.000/0.757
Epoch 150: loss=0.010, accuracy=1.000/0.773
Epoch 160: loss=0.008, accuracy=1.000/0.762
Epoch 170: loss=0.007, accuracy=1.000/0.773
Epoch 180: loss=0.007, accuracy=1.000/0.778
Epoch 190: loss=0.006, accuracy=1.000/0.768
```

```
Epoch 950: loss=0.001, accuracy=1.000/0.784
Epoch 960: loss=0.001, accuracy=1.000/0.784
Epoch 970: loss=0.001, accuracy=1.000/0.784
Epoch 980: loss=0.001, accuracy=1.000/0.784
Epoch 990: loss=0.001, accuracy=1.000/0.784
Epoch 1000: loss=0.001, accuracy=1.000/0.778

Final Test: final accuracy = 0.778
Final Epoch : 40 loss_zero time : Time : 3.248728036880493
```

- 역시나 학습이 진행되었다.
- 40 Epoch 시점에서 로스가 설정한 수준으로 떨어졌고 (학습이 완료되었고)
  그 시점까지 3.24s 의 시간이 걸렸습니다.
- 결과적으로 더 적은 노드를 가지는 신경망 구조보단 시간적으로 손해를 보았지만 정확도는 더 높다는 것을 확인 할 수 있었습니다.

→ 시간적으로도, 정확성 적으로도 이득을 보는 방법?

- **ReLu**

▼ Code

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

def relu_derv(y):
    return np.sign(y)
```

```
# Chapter 4
# 은닉 계층 하나를 위한 순전파 함수 정의
def forward_neuralnet_hidden1(x):
    # init_model_hidden1 쌍의 파라미터에 접근
    global pm_output, pm_hidden

    #입력 x와 pm_hidden을 통해 은닉계층 출력 계산 Relu함수 사용
    hidden = relu(np.matmul(x, pm_hidden['w']) + pm_hidden['b'])
    #hidden과 pm_output을 통해 출력계층 출력이자 최종출력 output 계산
    output = np.matmul(hidden, pm_output['w']) + pm_output['b']

    #출력 계층의 역전파 처리 때 가중치에 대한 편미분 정보로 hidden이 필요함
    return output, [x, hidden]
```

```
# Chapter 4
# 은닉 계층 하나를 위한 역전파 함수 정의
def backprop_neuralnet_hidden1(G_output, aux):
    # G_output은 역전파 후처리 후만들어지는 파라미터
    global pm_output, pm_hidden

    # 순전파에 사용됐던 input
    x, hidden = aux

    # 출력층에 대한 역전파
    # 출력층에서 사용하는 input은 hidden
    g_output_w_out = hidden.transpose()
    G_w_out = np.matmul(g_output_w_out, G_output)
    G_b_out = np.sum(G_output, axis=0)

    # 출력 계층과 은닉 계층 역전파 처리 매개하는 G_output으로부터 G_hidden을 구해내는 과정
    g_output_hidden = pm_output['w'].transpose()
    # pm_output['w']은 변환이 되는 계수이기 때문에 미리 기록한 것 위에 명시된 식에서 W_2
    G_hidden = np.matmul(G_output, g_output_hidden)

    # 출력층 역전파
    pm_output['w'] -= LEARNING_RATE * G_w_out
    pm_output['b'] -= LEARNING_RATE * G_b_out

    # 은닉층 -> relu -> 출력층 순서
    # 역전파는 그 반대로 가야하기 때문에 Relu 부분을 처리해줘야 함 위 식에서 편미분값
    G_hidden = G_hidden * relu_derv(hidden)

    # 은닉층에 대한 역전파
    # 은닉층에서 사용하는 input은 x
    g_hidden_w_hid = x.transpose()
    G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
    G_b_hid = np.sum(G_hidden, axis=0)

    # 은닉층 역전파
    pm_hidden['w'] -= LEARNING_RATE * G_w_hid
    pm_hidden['b'] -= LEARNING_RATE * G_b_hid
```

▼ Result

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/88a3f346-f53b-4c6b-8e5f-f15f4a7b07ca/1___ETA___STD___Relu___Normal.txt

- Relu 함수를 이용한 신경망 구조 역시 단일 계층의 단일 노드로는 학습이 되지 않았다.

```
Epoch 950: loss=1.608, accuracy=0.215/0.141
Epoch 960: loss=1.608, accuracy=0.215/0.141
Epoch 970: loss=1.608, accuracy=0.215/0.141
Epoch 980: loss=1.608, accuracy=0.215/0.141
Epoch 990: loss=1.608, accuracy=0.215/0.141
Epoch 1000: loss=1.608, accuracy=0.215/0.141

Final Test: final accuracy = 0.141
Final Epoch : 1000 Running Time : 15.215362548828125
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ed4d42ef-44cd-4e88-a012-493d725f6b42/32___ETA_x_0.1___STD_x_10___Relu___Normal.txt

```
Epoch 950: loss=0.001, accuracy=1.000/0.751
Epoch 960: loss=0.001, accuracy=1.000/0.751
Epoch 970: loss=0.001, accuracy=1.000/0.751
Epoch 980: loss=0.001, accuracy=1.000/0.751
Epoch 990: loss=0.001, accuracy=1.000/0.751
Epoch 1000: loss=0.001, accuracy=1.000/0.746

Final Test: final accuracy = 0.746
Final Epoch : 76 loss_zero time : Time : 1.7764441967010498
```

- 활성화 함수를 Relu로 변경하고 하이퍼볼릭탄젠트 함수로 실험하였을 때 학습이 진행되었던 환경 그대로 실험을 진행하였다.
- 코드 실행마다 shuffle이 변하기 때문에 결과는 조금씩 달라질 순 있지만 하이퍼볼릭탄젠트 함수와 비교하였을 때 큰 차이는 없었다.

→ 더 복잡한 구조라면 어떨까?

은닉 계층 2개 적용

```
LEARNING_RATE = 0.001
RND_MEAN = 0
RND_STD = 0.0030
Layer_node_info = [8, 8]
file_set('./Result/Relu20/%s _ ETA _ STD _ Relu _ Normal'%Layer_node_info)
set_hidden(Layer_node_info)
NUM_exec(epoch_count=1000, report=10)
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7e60d395-8aff-4349-a10e-2084a140856d/8_8___ETA___STD___Relu___Normal.txt

```
Epoch 950: loss=0.046, accuracy=0.978/0.751
Epoch 960: loss=0.044, accuracy=0.974/0.751
Epoch 970: loss=0.043, accuracy=0.976/0.741
Epoch 980: loss=0.067, accuracy=0.969/0.724
Epoch 990: loss=0.046, accuracy=0.973/0.746
Epoch 1000: loss=0.045, accuracy=0.976/0.746

Final Test: final accuracy = 0.746
Final Epoch : 175 loss_zero time : Time : 3.663877487182617
```

- 각 히든레이어 별 노드수가 8개로 이루어져 있는 신경망이다.
- 결과적으로 노드수가 32개인 단층 구조보다 성능적으로 좋지 않았다 (속도적으로)
- 하지만, 적은 노드로부터 꽤나 괜찮은 정확도를 보였다.

→ 계층별 노드수를 증가해 정확도 면에서 큰 이득을 보고자 하였다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/593fea18-f894-4f72-b5d1-2f32df170867/32_32___ETA___STD___Relu___Normal.txt

```
Epoch 950: loss=0.000, accuracy=1.000/0.778
Epoch 960: loss=0.000, accuracy=1.000/0.778
Epoch 970: loss=0.000, accuracy=1.000/0.778
Epoch 980: loss=0.000, accuracy=1.000/0.778
Epoch 990: loss=0.000, accuracy=1.000/0.778
Epoch 1000: loss=0.000, accuracy=1.000/0.778

Final Test: final accuracy = 0.778
Final Epoch : 47 loss_zero time : Time : 1.4193549156188965
```

- 각 히든레이어 별 노드수를 대폭 증가하였다.
- 노드수가 증가하니 학습이 진행되는 속도가 더욱 빨라졌다.

→ 속도와 정확성을 동시에 잡으려면 계층수는 많으면서 노드수는 상황에 맞게 변경해야하나?

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/971bcd7e-9e99-45fa-9eb9-3a08f6807d8b/32_5___ETA___STD___Relu___Normal.txt

```
Epoch 950: loss=0.007, accuracy=0.997/0.838
Epoch 960: loss=0.007, accuracy=0.999/0.843
Epoch 970: loss=0.005, accuracy=0.999/0.838
Epoch 980: loss=0.005, accuracy=0.999/0.849
Epoch 990: loss=0.005, accuracy=0.997/0.843
Epoch 1000: loss=0.010, accuracy=0.997/0.827

Final Test: final accuracy = 0.827
Final Epoch : 112 loss_zero time : Time : 2.9348435401916504
```

- 입력 단계에서 더 많은 정보를 뽑아내기 위해 히든레이어1의 노드수는 유지한채, 출력 단계에 나가는 히든레이2의 노드수를 줄여보았습니다.
- 높은 정확도를 기록하였지만 , 노드수 감소에 따라 학습속도는 느려졌습니다.

→ 계층을 더 늘려보자

은닉 계층 3개 적용

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b87d689e-6bc7-4cf7-9d13-4f098cf6112a/8_8_8___ETA___STD___Relu___Normal.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3a7028db-9f70-4f80-b52a-db959a2fdad2/32_16_8___ETA_x_0.1___STD_x_10__Relu___Normal.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/444216b9-b577-4f41-8440-4c5b55b26f0a/32_32_32___ETA_x_0.1___STD_x_10__Relu___Normal.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a20d483a-98b6-4bb8-b906-f8feb46650b6/64_64_64___ETA_x_0.1___STD_x_10__Relu___Normal.txt

```
Epoch 950: loss=1.608, accuracy=0.214/0.146
Epoch 960: loss=1.608, accuracy=0.214/0.146
Epoch 970: loss=1.608, accuracy=0.214/0.146
Epoch 980: loss=1.608, accuracy=0.214/0.146
Epoch 990: loss=1.608, accuracy=0.214/0.146
Epoch 1000: loss=1.608, accuracy=0.214/0.146

Final Test: final accuracy = 0.146
Final Epoch : 1000 Running Time : 20.691175937652588
```

```
Epoch 950: loss=0.226, accuracy=0.911/0.746
Epoch 960: loss=0.232, accuracy=0.914/0.714
Epoch 970: loss=0.217, accuracy=0.926/0.746
Epoch 980: loss=0.209, accuracy=0.924/0.751
Epoch 990: loss=0.242, accuracy=0.893/0.746
Epoch 1000: loss=0.240, accuracy=0.896/0.741

Final Test: final accuracy = 0.741
Final Epoch : 784 loss_zero time : Time : 15.940345764160156
```

- [8 8 8] 적용, 학습되지 않는다.
  → 학습률, 정규분포 변경

```
Epoch 950: loss=0.022, accuracy=0.997/0.730
Epoch 960: loss=0.021, accuracy=0.997/0.719
Epoch 970: loss=0.020, accuracy=0.997/0.730
Epoch 980: loss=0.020, accuracy=0.997/0.714
Epoch 990: loss=0.020, accuracy=0.997/0.735
Epoch 1000: loss=0.018, accuracy=0.997/0.730

Final Test: final accuracy = 0.730
Final Epoch : 438 loss_zero time : Time : 11.077585935592651
```

```
Epoch 950: loss=0.005, accuracy=1.000/0.816
Epoch 960: loss=0.005, accuracy=1.000/0.811
Epoch 970: loss=0.005, accuracy=1.000/0.816
Epoch 980: loss=0.005, accuracy=1.000/0.811
Epoch 990: loss=0.005, accuracy=1.000/0.822
Epoch 1000: loss=0.005, accuracy=1.000/0.805

Final Test: final accuracy = 0.805
Final Epoch : 195 loss_zero time : Time : 5.4192633628845215
```

- 4가지 Case 역시 , 학습은 진행되지만, 은닉 계층 2개를 사용할 때와 비교했을 때 큰 이점을 볼 수 없었다.

→ 초기화 방식의 문제인가?

- 현재 계층의 개수와 계층의 노드의 개수에 따라 표준편차와 학습률을 임의로 건드려 가져 학습이 되는 상황을 억지로 조작해왔다. (제공된 코드상 표준편차는 0.003 이었다.) 하지만 이는 학습모델에 따라 달라질 필요가 있으며 초기화 방법을 변화시킬 필요가 있을 것이라고 생각하였다.

- **ReLu : He Initailization 적용**

$$W = \mathcal{N}(0,\ std^2)$$

$$std = \frac{gain}{\sqrt{fan_{in}}}$$

nonlinearity	gain
Linear / Identity	1
Conv{1,2,3}D	1
Sigmoid	1
Tanh	$\frac{5}{3}$
ReLU	$\sqrt{2}$
Leaky Relu	$\sqrt{\frac{2}{1+\text{negative\_slope}^2}}$
SELU	$\frac{3}{4}$

  - 허초기화에 대한 공식이다.
  - 이전계층의 노드수에 따라 가중치의 표준편차가 달라지게 된다.
  - Gain의 경우 활성화함수 Relu에서 보편적으로 $\sqrt{2}$ 를 사용한다.

▼ Code

```
def alloc_param_pair(shape):
    # 허 초기화 방식 사용
    RND_STD = np.sqrt(2 / (shape[0]))
    weight = np.random.normal(RND_MEAN, np.power(RND_STD,2), shape)
    bias = np.zeros(shape[-1])
    return {'w':weight, 'b':bias}
```

▼ Result

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9492fb0b-c6dd-4617-aa62-e26383f9b8e3/64_64___ETA___STD___Relu___He.txt

```
Epoch 950: loss=0.000, accuracy=1.000/0.827
Epoch 960: loss=0.000, accuracy=1.000/0.827
Epoch 970: loss=0.000, accuracy=1.000/0.827
Epoch 980: loss=0.000, accuracy=1.000/0.827
Epoch 990: loss=0.000, accuracy=1.000/0.827
Epoch 1000: loss=0.000, accuracy=1.000/0.827

Final Test: final accuracy = 0.827
Final Epoch : 62 loss_zero time : Time : 2.050497531890869
```

- 허초기화 이후 학습을 진행하였을 때 대부분 좋은 정확도를 보이고 있다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/26c94c02-e809-41a6-af81-4393434b5fd7/128_128___ETA___STD___Relu___He.txt

```
Epoch 950: loss=0.000, accuracy=1.000/0.843
Epoch 960: loss=0.000, accuracy=1.000/0.843
Epoch 970: loss=0.000, accuracy=1.000/0.843
Epoch 980: loss=0.000, accuracy=1.000/0.843
Epoch 990: loss=0.000, accuracy=1.000/0.843
Epoch 1000: loss=0.000, accuracy=1.000/0.843

Final Test: final accuracy = 0.843
Final Epoch : 52 loss_zero time : Time : 5.347322463989258
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/89ea10d2-a4c6-415b-8059-4e173a4fea80/256_256___ETA___STD___Relu___He.txt

```
Epoch 950: loss=0.000, accuracy=1.000/0.892
Epoch 960: loss=0.000, accuracy=1.000/0.892
Epoch 970: loss=0.000, accuracy=1.000/0.892
Epoch 980: loss=0.000, accuracy=1.000/0.892
Epoch 990: loss=0.000, accuracy=1.000/0.892
Epoch 1000: loss=0.000, accuracy=1.000/0.892

Final Test: final accuracy = 0.892
Final Epoch : 47 loss_zero time : Time : 10.143116235733032
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e3ca47b8-c1be-4b61-9d27-d7b97817437b/64_64_64___ETA___STD___Relu___He.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/cea01d30-fb8e-4487-b011-cdbf052a310a/32_32_32_32___ETA___STD___Relu___He.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2c338cc1-ebbb-4e08-9d19-c87658b828ae/64_64_64_64___ETA___STD___Relu___He.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/236b83a2-25d3-4f91-b0d3-157fc108e4c7/64_32_16_4___ETA___STD___Relu___He.txt

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d6e65632-32ca-4bd1-a9f2-20830e7cca9d/128_64_32_16___ETA___STD___Relu___He.txt

- 이후, 히든레이어 계층의 개수를 3개, 4개 를 진행하였을 때 여전히 히든레이어 개수가 2개일 때가 가장 결과가 좋았다.

▼ 2. 8x8에 그려진 숫자

- Relu , He Initialization 사용

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/8608187c-07e2-4e90-ac9b-709745594bea/256_256___ETA___STD___Relu___He.txt

- 20x20 에 그려진 숫자를 학습시키는 과정에서 가장 정답률이 높았던 구조를 그대로 적용해 보았다.
  - Loss가 지속적으로 줄어 들긴 하였으나, Final Accuracy

```
Epoch 950: loss=1.196, accuracy=0.522/0.288
Epoch 960: loss=1.162, accuracy=0.506/0.295
Epoch 970: loss=1.167, accuracy=0.510/0.311
Epoch 980: loss=1.127, accuracy=0.529/0.356
Epoch 990: loss=1.161, accuracy=0.533/0.220
Epoch 1000: loss=1.155, accuracy=0.555/0.402

Final Test: final accuracy = 0.402
Final Epoch : 1000 Running Time : 116.56916761398315
```

가 0.402 로 제대로 학습이 이루어 지지 않은 것을 확인할 수 있었다.

- 8x8 이미지는 각 데이터가 위치도 다르고, 기울기 등등이 다르기 때문에 실제 파일의 량보다 데이터가 부족하기 때문이라고 생각한다.
- 따라서 한번에 학습이 진행되는 미니배치의 수를 조정해 보았다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3353b2aa-bd74-459a-a44d-1761af592f17/256_256___ETA___mb_size_5___STD___Relu___He.txt

```
Epoch 950: loss=0.462, accuracy=0.828/0.488
Epoch 960: loss=0.697, accuracy=0.754/0.496
Epoch 970: loss=0.722, accuracy=0.754/0.535
Epoch 980: loss=0.565, accuracy=0.784/0.465
Epoch 990: loss=0.596, accuracy=0.790/0.535
Epoch 1000: loss=0.676, accuracy=0.739/0.504

Final Test: final accuracy = 0.504
Final Epoch : 737 loss_zero time : Time : 196.6618893146515
```

- 정확도는 0.5 까지 향상되었다.
- Loss 가 점차줄어 들고 있다. 가끔식 오차가 튀는 경우가 있긴 하나, 점차 감소하는 추세이기 때문에 Epoch를 늘려 더 나은 결과를 기대해 보았다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/10f9ab57-e47a-4f11-aa66-d42a12171a7b/256_256___Epoch_2000___ETA___mb_size_5___STD___Relu___He.txt

```
Epoch 1950: loss=0.755, accuracy=0.800/0.441
Epoch 1960: loss=0.317, accuracy=0.901/0.386
Epoch 1970: loss=0.287, accuracy=0.905/0.307
Epoch 1980: loss=0.440, accuracy=0.859/0.378
Epoch 1990: loss=0.459, accuracy=0.840/0.402
Epoch 2000: loss=0.693, accuracy=0.792/0.402

Final Test: final accuracy = 0.402
Final Epoch : 749 loss_zero time : Time : 199.22180795669556
```

- Epoch을 향상시켬음에도 불구하고 중간지점 부터 loss 가 증가하며, 정확도가 이전보다 낮아지는 현상이 발생했다.

▼ 3. 20x20,8x8 이 섞여 있는 데이터 셋

- Relu, He Initialization 사용
  - 1,2 번과 마찬가지로, Hidden Layer의 개수는 2개, 노드의 개수는 입력 계층에서 히든 계층 까지는 입력노드의 수와 비슷한 개수로 생성하였다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c7df9f73-4d8d-4a6d-b55c-f66431544382/256_256___ETA___STD___Relu___He.txt

```
Epoch 950: loss=0.257, accuracy=0.914/0.536
Epoch 960: loss=0.186, accuracy=0.949/0.685
Epoch 970: loss=0.035, accuracy=0.988/0.678
Epoch 980: loss=0.035, accuracy=0.989/0.688
Epoch 990: loss=0.678, accuracy=0.837/0.571
Epoch 1000: loss=0.027, accuracy=0.991/0.694

Final Test: final accuracy = 0.694
Final Epoch : 94 loss_zero time : Time : 34.97575664520264
```

- 히든레이어는 2개로 이루어져 있고, 각 계층의 노드의 수는 이전 20x20에서 가장 좋은 결과를 냈던 256 x 256 로 형성하였다.
- 그 결과 마지막 테스트에서 0.694 라는 결과를 내었다.
- 8x8 이미지로만 이루어졌던 학습결과 보단 더 나은 결과를 냈다.

- 이는 상대적으로 20x20 에 해당하는 데이터셋이 더 많고, 8x8 데이터셋이 더 적어서 생긴 결과라고 생각된다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a8ca2c72-094c-44b8-a1dc-2012e5bae532/512_128___ETA___STD___Relu___He.txt

```
Epoch 950: loss=0.102, accuracy=0.964/0.744
Epoch 960: loss=0.080, accuracy=0.974/0.735
Epoch 970: loss=0.048, accuracy=0.984/0.751
Epoch 980: loss=0.114, accuracy=0.962/0.729
Epoch 990: loss=0.046, accuracy=0.985/0.732
Epoch 1000: loss=0.049, accuracy=0.981/0.722

Final Test: final accuracy = 0.722
Final Epoch : 242 loss_zero time : Time : 133.8845775127411
```

- 입력노드로 부터 더 많은 정보를 빼기 위해 히든레이어1의 노드수를 더 늘렸다.
- 그 결과 정확도가 더욱 상승하였다.

▼ 결론

1. Loss 가 0까지 수렴하는데 있어 중요한 요소는 레이어의 개수가 아닌 각 레이어당 노드의 개수이다.
2. 입력 레이어 단계에서 연결되는 히든레이어의 노드의 개수는 분류의 정확도에 큰 영향을 끼친다. (입력에 대한 정보를 더 많이 가져오기 때문인 듯하다.
3. 너무 많은 히든 레이어의 개수는 학습에 오히려 악 영향을 미친다.
4. 시그모이드 함수는 히든레이어 사이의 순전파/역전파 에서 사용시 가중치 손실/과적합이 일어날 수 있으므로 적절하지 않다.
5. 정규분포 가중치 초기화 방법은 단순한 신경망 구조이고, 입력 노드가 별로 없다면 큰 영향을 끼치지 않지만, 입력 노드수가 많아지고, 복잡한 신경망이라면 문제가 생길 수 있다. (표준편차가 너무 작으면 가중치가 몰리기 때문이다.)

프로젝트 진행하면서 개선할 점

8x8 실험을 진행할 때 학습이 잘 이루어지지 않았다.
왜 그런것인가에 대해 생각해 보았는데, 데이터의 수가 현저히 부족했다.
같은 숫자라도, 그 숫자의 위치정보에 따라 경우의수가 무수히 많아진다. (20x20 은 상대적으로 그러한 현상이 적다.)
따라서 학습을 진행하기 전 , 이미지 전처리가 필요할 것이라고 생각한다. (기울기 보정 또는 Canny Edge 등)
전처리 한 데이터를 가지고 다시 한번 입력에 데이터로 넣으면 더 좋은 결과가 나올 것 이라고 생각한다.