# 도형을 활용한 파킨슨 진단

제출일: 2023.12.07
학부/학과: 휴먼기계바이오공학부,
뇌인지과학과, 인공지능학과
학번: 1970096, 2070044, 2076274, 2277037
제출자: 홍지우, 오예인, 윤서영, 홍지영

# 목차

# 1. Introduction

## 1.1 파킨슨 병 정의

: 파킨슨 병은 중추 신경계의 신경세포 손상으로 인해 발생하는 만성 질환을 말한다. 주로 운동 기능에 영향을 미치며 다리 떨림, 근육 경직, 움직임의 둔화와 불안정성을 증상으로 한다. 특히 움직임 둔화와 근육 경직의 현상은 질병 초기에 나타난다.

## 1.2 문제점

: 파킨슨 발병을 예방하기 위해서는 초기 단계에서의 정확한 진단이 매우 중요하다. 그러나 많은 연구에서 펜이 주는 압력과 속도, 각도 등 전문적인 검사를 해야지만 얻을 수 있는, 계산된 데이터 셋을 사용해서만 파킨슨 병을 진단한다. 이는 초기 단계에서 빠르게 검사를 받아 파킨슨 발병을 줄이는 목표에 부합하지 않는 매우 부적절한 방식이다.

## 1.3 목표

: 아르키메데스 나선이라는 도형 그림 데이터를 활용하여 사용자가 직접 자가 진단을 수행할 수 있는 모델을 개발하는 것을 목표로 한다.

# 2. Related Work

## 2.1. Handwriting-Based ADHD Detection for Children Having ASD Using Machine Learning Approaches

: 그림을 이용하여 Attention Deficit Hyperactivity Disorder(ADHD) 어린이를 감별하는 방법을 머신러닝에 적용한 연구이다. 본 논문에서는 Autism Spectrum Disease (ASD)을 동반하는 ADHD 어린이 환자와 건강한 어린이 각각 14, 15명을 대상으로 실험을 진행하였다. 연구자들은 실험 대상자들에게 태블릿 스크린에 지그재그 선과 주기적인 선을 그리도록 하였는데, 처음에는 왼쪽 아래의 그림처럼 패턴의 앞 부분만을 보여주어 뒷부분을 예측하여 그리게 하고, 두번째로는 선이 그려져있는 그림을 따라 그리도록 하였다(Fig 1). 이를 3번 반복하여 환자들의 데이터를 얻고, 각 데이터에서 펜의 좌표, 속도, 압력, 기울기 등의 특징을 추출하였다(Fig 2). 이 추출한 특징을 머신러닝을 통해 학습시켜 환자그룹과 건강한 그룹을 분류하는 모델을 구현하였다.

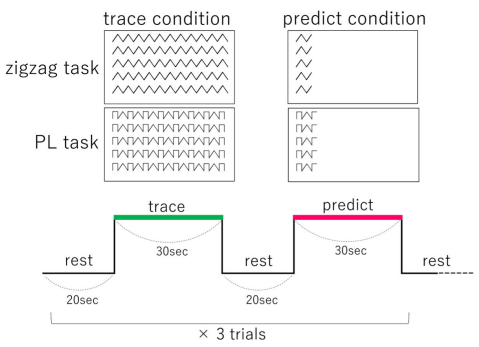



Figure 1. Handwritten data collection procedure for four tasks

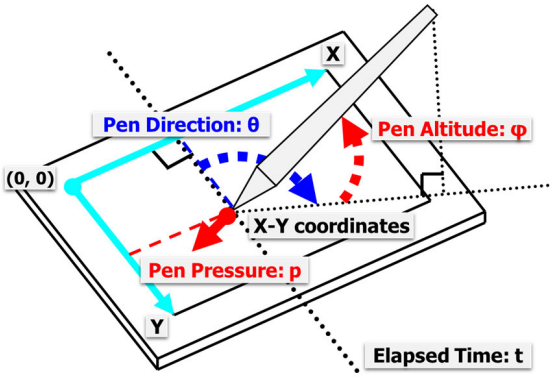



Figure 2. Pen tablet device

### 2.2 Distinguishing Different Stages of Parkinson's Disease Using Composite Index of Speed and Pen-Pressure of Sketching a Spiral

: 위 논문에서는 점으로 가이드된 아르키메데스 나선을 그릴 때 속도와 필압을 활용해 파킨슨병을 진단하고 심각도를 구분한다. 기존 연구들에서는 나선형 스케치를 통해 amplitude of tremor, extent of bradykinesia and dyskinesia와 같은 생리학적 매개변수들을 추출하고 이를 활용해 patient group과 control group을 성공적으로 구별했다. 이 연구는 더 나아가 질병의 중증도를 평가하기 위해 위와 같은 지표를 활용하고자 했으며 기존 연구들과는 달리 속도와 필압 각각을 개별의 매개변수로 활용하는 것이 아니라 두 특징의 스칼라 곱을 사용해 스케치 속도와 펜 압력의 합성 지수는 CISP를 얻었고 이를 통해 patients와 controls를 구분하고 중증도를 비교 검사했다.

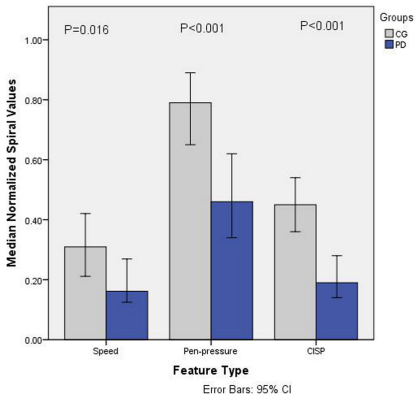



**FIGURE 2** | Barchart showing median normalized values (0–1) of speed, pen-pressure, and Composite Index of Speed and Pen-pressure (CISP) for Parkinso disease (PD) and control group (CG).

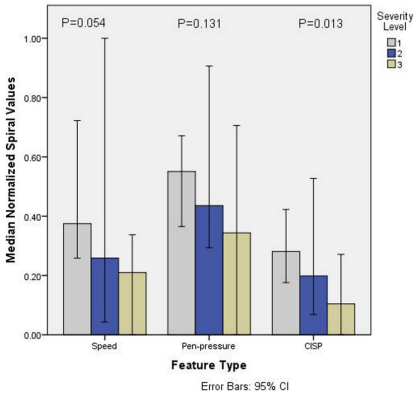



**FIGURE 3** | Barchart showing median normalized values (0–1) of speed, pen-pressure, and Composite Index of Speed and Pen-pressure (CISP) versus severity level (SL) (1–3) of Parkinson's disease.

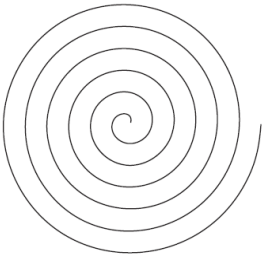
왼쪽 표는 patients와 controls의 속도, 필압, CISP를 정규화한 자료이며 오른쪽 표는 이를 심각도에 따라 시각화한 자료이다. 연구 결과 patients와 controls 사이의 구분은 확실했지만 1, 3단계 사이의 심각도를 구분하는 것은 성공했으나 2단계와의 큰 차이를 구별하지 못했다.

## 3. Data

### 3.1. Data introduction and Statistical Analysis

a) 데이터 ①

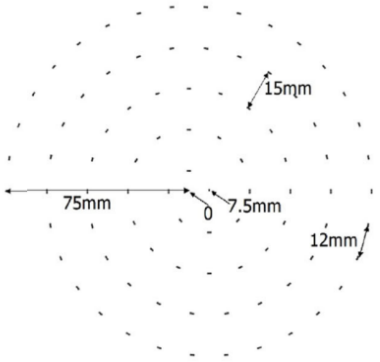
: 파킨슨 환자의 핸드라이팅 데이터셋을 BDALab에서 승인받아 사용하였다. 데이터의 형식은 svc 파일로서, 35명의 파킨슨 환자와 36명의 건강한 사람이 아르키메데스 나선을 그릴 때 측정한 X, Y 좌표, 시간, 압력 등의 값들로 구성되어 있다. 모델의 입력값은 이미지가 되어야 하므로 이 정보값들을 이미지로 변환하는 과정이 필요하였고, MATLAB을 이용하여 png파일로 저장하였다.

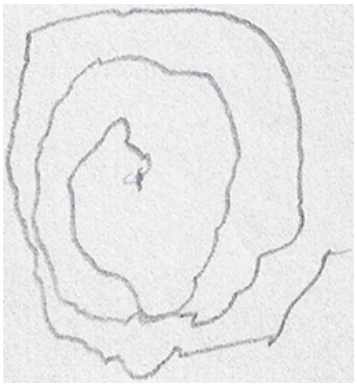


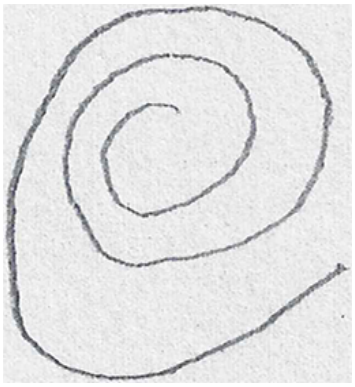
b) 데이터 ②

: 앞서 소개한 두번째 선행연구논문에서 제공한 이미지 데이터를 받아 사용했다. 이 데이터는 필압, 속도 등 선행연구에서 활용한 전체 데이터가 아닌 결과적으로 피실험자들이 그려낸 이미지 PNG 파일만으로 구성되어 있었다.

피실험자들은 태블릿 위에 종이를 얹고 밝은 색으로 가이드된 점들을 따라 아르키메데스 나선을 그리도록 하였는데 이는 질병 유무 외의 요소들이 가능한 개입되지 않도록 하기 위해서였다. 피실험자들이 받은 '밝은 색으로 가이드된 점 도안'은 왼쪽 사진과 같으며 나머지 사진은 완성된 이미지 파일이다. 순서대로 파킨슨병 환자와 정상인의 데이터이다.





## 3.2. Data Preprocessing

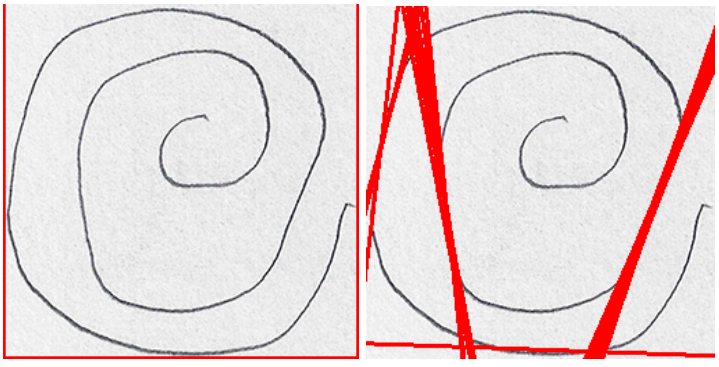
a) Segmentation

: 아르키메데스 나선 분할을 위해 segmentation에 사용되는 cv2를 import했다. 원활한 분할을 위해 이미지를 gray scale로 바꾸었다. 그후 차례로 GaussianBlur, Canny, Houghlines를 진행했다. 이때 Houghlines의 경우, 직선을 검출하는 허프변환 함수이다. 찾고자 하는 것은 곡선인 아르키메데스 나선이므로 여러개의 직선을 검출하고 통합하여 허프 웨이브 변환과 유사한 결과를 얻고자 시도했다.

```
import cv2
from google.colab.patches import cv2_imshow
import os
import numpy as np
from google.colab import files
from IPython.display import Image, display

# Upload image
uploaded = files.upload()
img      =      cv2.imdecode(np.frombuffer(uploaded['V01HE01.png'],
np.uint8), cv2.IMREAD_COLOR)

# Convert image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply GaussianBlur to reduce noise and improve edge detection
```

```
blurred = cv2.GaussianBlur(gray, (5, 5), 0)

# Use Canny edge detector
edges = cv2.Canny(blurred, 30, 100)  # Adjust the parameters as
needed

# Use Hough line transform to detect lines
lines = cv2.HoughLines(edges, 1, np.pi / 180, threshold=50)  #
Adjust the threshold as needed

# Create a folder
output_folder = 'Image_0'
os.makedirs(output_folder, exist_ok=True)

# Check if lines are detected
if lines is not None:
    # Draw detected lines on the image
    for line in lines:
        rho, theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * (a))
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * (a))
        cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 2)

    cv2_imshow(img)
else:
    print("No lines detected.")
```

**Segmentation 결과**



위의 결과를 보면 왼쪽 그림처럼 나선이 아닌 이미지의 테두리를 직사각형으로 분할하거나 오른쪽 그림처럼 소수의 접선만을 그려내어 나선의 특징을 대표할 수 없도록 segmentation이 제대로 진행되지 않았다. 나선 이미지를 segmentation하는 것은 쉽지 않고 원본 데이터가 인식하기에 어려움이 없어 추가적인 preprocessing이 필요 없다고 판단하여 위에서 작성한 segmentation코드는 최종 프로젝트에 사용하지 않았다.

b) Flipping

: 이미지 데이터의 경로를 매개변수로 받아서 가로로 플립해준 후, 또다른 매개변수로 받은 저장 경로로 플립된 데이터를 저장하는 'flip_image'함수를 정의하였다.

그 후, 원본 이미지가 저장된 디렉토리(input_directory)에 접근하여, 디렉토리 안의 모든 이미지를 for 반복문을 통해 접근하였고, 각 이미지마다 'flip_image'함수를 적용해주어 flipped 이미지를 생성하였다.

```
from PIL import Image
import numpy as np
import os

# 가로로 플립하는 함수
def flip_image(image_path, save_path):
    image = Image.open(image_path)
    flipped_image = image.transpose(Image.FLIP_LEFT_RIGHT)
    flipped_image.save(save_path)

# 이미지가 저장된 디렉토리
input_directory = "./case_application_dataset/controls"

# 플립된 이미지를 저장할 디렉토리
output_directory = "./case_application_dataset/controls"

# 디렉토리가 없다면 생성
os.makedirs(output_directory, exist_ok=True)

# 디렉토리 내의 각 이미지에 대해 플립 적용 및 저장
for filename in os.listdir(input_directory):
    if filename.endswith(".jpg"): # 혹은 다른 이미지 형식
        image_path = os.path.join(input_directory, filename)
        save_path = os.path.join(output_directory, "flipped_" +
filename)
        flip_image(image_path, save_path)

print("플립된 이미지가 성공적으로 저장되었습니다.")
```

c) Noise

: flip을 이용한 데이터 증강에도 아직 데이터 양이 부족하다고 생각되어 더 많은 데이터를 생성을 위해, 이전 단계에서 생성된 flipped 이미지에도 noise를 적용하여 주었다.

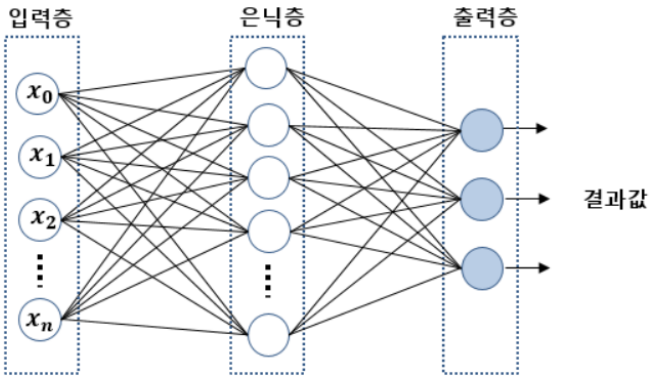
이미지에 노이즈를 추가하는 함수 'add_noise'를 정의하였고, 이 함수 또한 앞의 'flip_image'함수와 동일하게 noise를 적용한 이미지를 디렉토리에 저장하도록 코드를 작성하였다.

```
from PIL import Image, ImageFilter
import numpy as np
import os

# 이미지에 노이즈를 추가하는 함수
def add_noise(image_path, save_path, noise_factor=0.5):
    image = Image.open(image_path)

    # 이미지 크기를 가져오고 노이즈를 생성
    width, height = image.size
    channels = len(image.getbands())
    noise = np.random.normal(scale=noise_factor, size=(height, width,
            channels))

    # 노이즈를 이미지에 더하고 값을 0과 255 사이로 클리핑
    noisy_image = np.clip(np.array(image) + noise * 255, 0,
            255).astype(np.uint8)

    # 노이즈가 추가된 이미지를 저장
    Image.fromarray(noisy_image).save(save_path)

# 이미지가 저장된 디렉토리
input_directory = "./case_application_dataset/controls"

# 노이즈가 추가된 이미지를 저장할 디렉토리
output_directory = "./case_application_dataset/controls"

# 디렉토리가 없다면 생성
os.makedirs(output_directory, exist_ok=True)

# 디렉토리 내의 각 이미지에 대해 노이즈를 추가하고 저장
for filename in os.listdir(input_directory):
    if filename.endswith(".jpg"):  # 혹은 다른 이미지 형식
        image_path = os.path.join(input_directory, filename)
        save_path = os.path.join(output_directory, "noisy_" +
```

```
filename)
        add_noise(image_path, save_path)

print("노이즈가 추가된 이미지가 성공적으로 저장되었습니다.")
```

# 4. Model

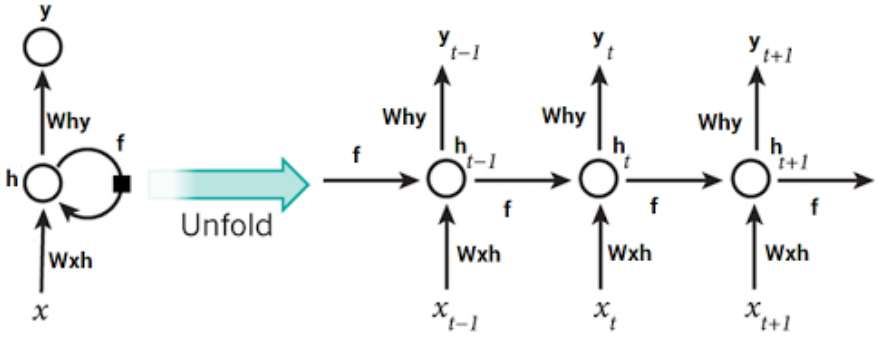
## 4.1 MLP




: MLP는 다층 퍼셉트론으로 지도학습에 사용되는 인공 신경망의 한 형태이다. 퍼셉트론으로 이루어진 층(layer) 여러 개를 순차적으로 붙여놓은 형태로, 입력층과 출력층 사이에 하나 이상의 중간층을 두어 비선형적으로 분리되는 데이터에 대해서도 학습이 가능하도록 만든 것이다. MLP에서 가장 중요한점은 인접한 층의 퍼셉트론간의 연결은 있어도 같은 층의 퍼셉트론끼리의 연결은 없다는 점이다. 더하여 MLP에서는 한번 지나간 층으로 다시 연결되는 피드백(feedback)도 없다는 특징을 가진다.

하지만 다층의 복잡한 모델 구현의 결과로 매번 input 데이터가 들어올때마다 모델에 부담이 많이 가게 되어 안정성이 떨어지는 이슈가 발생할 수 있다.

## 4.2 RNN




: RNN(Recurrent Neural Network)은 노드 간 연결이 시간 순서에 따라 방향 그래프를 형성하는 인공 신경망이다. 여기서 노드는 하나의 인공 신경망 셀을 의미한다. 즉, 시간

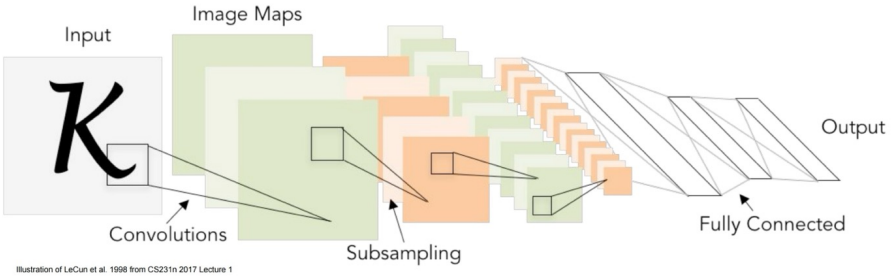
순서상 이전 셀에서 다음 셀로 정보가 전달되는 연결이 존재한다는 것이다. RNN은 입력값 x와 이전 은닉상태 ht-1을 함께 입력 받아 파라미터를 통해 결과값과 함께 현재의 은닉 상태를 반환하는 구조를 가진다. 그렇기 때문에 RNN은 과거의 정보를 기억한다는 강점을 가지며 이는 문장의 단어들이나 시계열과 같이 시간 흐름에 따라 변하는 데이터를 처리를 용이하게 할 수 있도록 도와준다. 또한 RNN은 각 시간 단계별로 가중치를 공유하고 시퀀스 길이의 관계없이 무작위 길이의 정보를 처리할 수 있다.

이미지는 인접한 픽셀들이 깊이 연관되어 있는 데이터이다. 다시 말해 RNN을 통해 이전 픽셀에서의 정보를 기억할 수 있다면 객체들의 순서가 중요한 이미지 데이터의 경우 image classification이 효과적일 수 있다는 것이다. 또한 RNN은 입력 시퀀스의 길이에 매우 유연하다. 이는 서로 다른 크기의 이미지를 다루는데 유리할 수 있다.

하지만 RNN은 Long-Term Dependencies를 잘 학습하지 못하는 단점을 가지고 있다. 이를 보완하기 위해 본 보고서에서는 LSTM(Long Short-Term Memory)라는 변형 구조를 활용해 layer가 deep해질 때 발생하는 희석 문제를 해결하고자 한다.

### 4.3 CNN(Convolutional Neural Network)

: CNN은 ‘Spatial locality’와 ‘Positional invariance’라는 주요 가정을 기반으로 하여 이미지 데이터 분류에서 좋은 성능을 보이는 딥러닝 모델이다. 여기서 ‘Spatial locality’란 각 필터가 오직 주변의 pixels에만 집중함을 의미하며, ‘Positional invariance’는 같은 필터가 이미지의 모든 위치에 적용됨을 의미한다.




CNN은 Convolution layer와 pooling layer를 반복적으로 사용하는 구조로 이루어져 있다. 여기서 Convolution layer는 입력 이미지에서의 패턴을 감지한다. 즉, Convolution 과정이 수행되는데 각각의 특징을 찾는 filter들이 input image를 돌아다니면서 계산을 수행하고, 그 계산 결과로 feature map을 생성한다.

Pooling layer는 공간을 다운샘플링하여 계산 효율성을 높이고 overfitting을 방지하는 역할을 수행한다. Pooling에는 Max Pooling과 Average Pooling이 있는데, Max Pooling은 Pooling을 수행하는 필터 위의 값들 중 가장 큰 값을 반환하고 Average Pooling은 필터 위의 값들을 평균낸 값을 반환한다.

## 5. Modeling

### 5.1 MLP

: MLP는 먼저 이미지 데이터를 직접 불러와서 이미지 데이터의 각 픽셀 값을 Numpy 배열로 변환한다. 그리고 Numpy 배열로 불러온 데이터 값들의 레이블을 원-핫 인코딩을 이용해 최종적으로 컴퓨터가 쉽게 이해할 수 있도록 데이터를 정리해준다. 더하여 뒤에 정의할 모델 코드에 바로 데이터를 넣을 수 있게 훈련 및 테스트 세트를 나누는 과정도 이 순서에서 거쳤다.

```python
# 이미지를 로드하고 NumPy 배열로 변환하는 함수
def load_and_preprocess_image(image_path, target_size=(128, 128)):
    img = load_img(image_path, target_size=target_size)
    img_array = img_to_array(img)
    img_array /= 255.0  # 이미지를 [0, 1] 범위로 정규화
    return img_array



# 데이터 생성
data = []

# 폴더 탐색
for root, dirs, files in os.walk(data_folder):
    for file in files:
        if file.endswith((".jpg", ".jpeg", ".png")):
            # 이미지 파일 경로
            image_path = os.path.join(root, file)

            # 레이블 (폴더 이름을 사용할 수도 있음)
            label = os.path.basename(root)

            # 데이터 딕셔너리에 추가
            data.append({"image_path": image_path, "label": label})


images = []
labels = []

for sample in data:
    image_path = sample["image_path"]
    label = sample["label"]

    img_array = load_and_preprocess_image(image_path)
    images.append(img_array)
    labels.append(label)


# 이미지 데이터를 NumPy 배열로 변환
images = np.array(images)
labels = np.array(labels)
```

```
# 레이블을 숫자로 인코딩
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)

# 레이블을 원-핫 인코딩
num_classes = len(np.unique(encoded_labels))
one_hot_labels = tf.keras.utils.to_categorical(encoded_labels,
num_classes)

# 훈련 및 테스트 세트로 데이터 분할
train_images, test_images, train_labels, test_labels =
train_test_split(
    images, one_hot_labels, test_size=0.2, random_state=42
)
```

MLP 모델 구축 코드 작성 단계에서, 처음에는 layer를 3개로만 쌓아서 코드를 작성해 돌려보았다. 그러나 그 결과 0.44 정도의 좋지 못한 정확도를 보였다. 그래서 아래와 같이 dense를 높여 총 5개의 layer를 쌓아 좀 더 깊은 모델을 사용했다. 더하여 모델 compile 단계에서 loss function을 이진 분류에 더 효과적인 성능을 가진 binary_crossentropy를 사용하여 정확도를 높였다.

하지만 이 코드는 다층의 복잡한 모델 구현으로 매번 input 데이터가 들어올때마다 모델에 부담이 많이 가도록 설계되어 있는 단점이 있다. 그래서 activation function이 진행되기 전에 batchnormalization을 추가로 넣어 이전 층 파라미터에 덜 의존적이게 만들어 주어 가중치 초기화에도 덜 민감하게 반응할 수 있도록 했다. 더하여 계속 깊어지는 학습에 MLP 모델의 과적합을 방지하기 위해서 Kernel_regulation 코드도 넣었다. Kernel_regulation 종류인 L1과 L2 중 저희는 L2를 선택하였는데, 그 이유는 저희가 사용하는 이미지 데이터가 흰 바탕에 까만색 선 하나만 그려진 모습으로, 이 데이터가 주는 특징이 다른 일반적인 이미지 데이터들이 주는 특징에 비해 양이 매우 적다고 분석되었기 때문이다. L2는 L1보다 더 많은 특성을 포함할 수 있다는 장점이 있다고 일반적으로 알려져 있다.

그러나 아래의 코드 학습률 0.003, epochs 100에서 training accuracy는 0.8273, test accuracy는 0.5143으로 아쉬운 정확도를 보였다.

```
# MLP 모델 정의
from keras.layers import BatchNormalization
from keras.regularizers import l2
from keras.callbacks import EarlyStopping

# 입력 모양을 (img_height, img_width, num_channels)라고 가정
input_shape = (128, 128, 3)

model = keras.Sequential([
    keras.layers.Flatten(input_shape=input_shape),
    BatchNormalization(),  # 활성화 함수 이전에 배치 정규화 추가
    keras.layers.Dense(512, activation='relu',
kernel_regularizer=l2(0.01)),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(256, activation='relu',
kernel_regularizer=l2(0.01)),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(128, activation='relu',
kernel_regularizer=l2(0.01)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(64, activation='relu',
kernel_regularizer=l2(0.01)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(32, activation='relu',
kernel_regularizer=l2(0.01)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(2, activation='softmax')
])

# 모델 컴파일
optimizer = keras.optimizers.Adam(learning_rate=0.0003)
model.compile(optimizer=optimizer, loss='binary_crossentropy',
metrics=['accuracy'])

# 모델 요약 출력
model.summary()

model.fit(train_images, train_labels, epochs=100, batch_size=10)
model.evaluate(test_images, test_labels)
```

```
 Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 flatten_1 (Flatten)          (None, 49152)             0

 batch_normalization_1 (Bat   (None, 49152)             196608
 chNormalization)

 dense_6 (Dense)              (None, 512)               25166336

 dropout_5 (Dropout)          (None, 512)               0

 dense_7 (Dense)              (None, 256)               131328

 dropout_6 (Dropout)          (None, 256)               0

 dense_8 (Dense)              (None, 128)               32896

 dropout_7 (Dropout)          (None, 128)               0

 dense_9 (Dense)              (None, 64)                8256

 dropout_8 (Dropout)          (None, 64)                0

 dense_10 (Dense)             (None, 32)                2080

 dropout_9 (Dropout)          (None, 32)                0

 dense_11 (Dense)             (None, 2)                 66

=================================================================
Total params: 25537570 (97.42 MB)
Trainable params: 25439266 (97.04 MB)
Non-trainable params: 98304 (384.00 KB)
_________________________________________________________________

Epoch 100/100
56/56 [==============================] - 31s 564ms/step - loss:
1.1147 - accuracy: 0.8273
5/5 [==============================] - 1s 55ms/step - loss: 1.6859 -
accuracy: 0.5143
[1.6858874559402466, 0.5142857432365417]
```

## 5.2 RNN

: RNN은 다른 모델과는 달리 시계열 데이터 혹은 순차적인 데이터를 다루기 위해 설계된 신경망 구조다. 하지만 이미지는 공간적인 특성을 가지고 있기 때문에 이미지를 RNN에 입력하기 위해서는 시계열 데이터로 변환해야 한다. LSTM의 경우 3D 배열의 입력을 요구한다. 고로 128*128의 이미지 데이터를 시간축으로 처리하기 위해서는 (샘플 수, 타임스텝, 특징)으로 reshape 해주어야 한다.

```
# 데이터 전처리 및 LSTM layer에 맞게 reshape
X_train = X_train.astype('float32') / 255.0
```

```
X_test = X_test.astype('float32') / 255.0

X_train = X_train.reshape((X_train.shape[0], 128, 128))
X_test = X_test.reshape((X_test.shape[0], 128, 128))
```

앞선 모델 소개에서 언급했듯 RNN은 layer가 deep해질수록 과거의 정보를 희석되기 때문에 이를 보완하기 위해 LSTM을 사용해 모델을 구축했다. LSTM, Dropout, 그리고 Dense를 활용해 구축한 모델을 Adam으로 학습률을 조정해가며 학습을 반복했다.

그 결과 학습률 0.001, epoch 100에서 training accuracy 0.4622, test accuracy 0.5000라는 가장 좋은 성능을 보였다.

```
# 모델 구축
model = Sequential()
model.add(LSTM(50,input_shape=(X_train.shape[1:]),activation='relu',
return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(50, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

# 모델 컴파일
optimizer = Adam(learning_rate=0.001)
model.compile(loss='binary_crossentropy',optimizer=optimizer,
metrics=['accuracy'])

# 모델 요약

model.summary()
# 모델 훈련
model.fit(X_train, y_train, epochs=100, batch_size=10)

# 모델 평가
evaluation_result = model.evaluate(X_test, y_test)

loss = evaluation_result[0]
accuracy = evaluation_result[1]
print(f"Test Loss: {loss}")
print(f"Test Accuracy: {accuracy}")

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

```
 lstm (LSTM)                 (None, 128, 50)           35800

 dropout (Dropout)           (None, 128, 50)           0

 lstm_1 (LSTM)               (None, 50)                20200

 dropout_1 (Dropout)         (None, 50)                0

 dense (Dense)               (None, 1)                 51

=================================================================
Total params: 56051 (218.95 KB)
Trainable params: 56051 (218.95 KB)
Non-trainable params: 0 (0.00 Byte)

.
.
.

Epoch 100/100
56/56 [==============================] - 10s 178ms/step - loss: 0.6932 -
accuracy: 0.4622
5/5 [==============================] - 1s 68ms/step - loss: 0.6931 -
accuracy: 0.5000
Test Loss: 0.6931496262550354
Test Accuracy: 0.5
```

## 5.3 CNN

: CNN의 가장 첫번째 layer는 파라미터 'input_shape'을 반드시 지정해주어야 하므로, dataset 크기에 맞추어 (120,160,3) 형태로 지정해주었다. 총 3번의 convolution layer와 maxpooling layer를 추가해주었고, activation function으로는 'relu'를 적용해주었다. Dropout layer도 추가하여 0.25의 확률로 노드들을 drop해주었는데, 이는 모델이 다양한 clues로부터 학습할 수 있도록 하기 위함이다.

이후에는 Flatten layer를 추가해주었고, class가 controls인지 patients인지를 구별하는 이진분류 문제이므로 마지막 layer에 'num_classes' 파라미터를 1로, 'activation' 파라미터도 sigmoid로 지정해주었다.

model.compile 단계에서는 마찬가지로 이진분류 문제이므로, 'loss' 파라미터를 binary_crossentropy로 지정해주었다. 마지막으로 model.fit 단계에서 epochs를 100을 주어 충분히 학습할 수 있도록 해주었다.

그 결과 마지막에 가서 평균 0.87정도의 accuracy를 보였고, test_dataset에 대해서는 0.92정도의 성능을 보였다.

```
from keras.models import Sequential
from keras.layers import Dropout, Activation, Dense
from keras.layers import Flatten, Convolution2D, MaxPooling2D
from keras.models import load_model
import tensorflow as tf
```

```
import cv2

batch_size = 20
num_classes = 1
model = Sequential()
model.add(Convolution2D(64, 3, 3,padding='same',
activation='relu',input_shape=(120,160,3)))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 3, 3,padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 3, 3,padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation = 'relu')) #relu
model.add(Dropout(0.5))
model.add(Dense(num_classes,activation = 'sigmoid'))

model.compile(loss='binary_crossentropy',optimizer='Adam',
              metrics=['accuracy'])
model.fit(train_dataset, batch_size=20, epochs=100)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 40, 54, 64)        1792

 max_pooling2d (MaxPooling2  (None, 20, 27, 64)        0
 D)

 dropout (Dropout)           (None, 20, 27, 64)        0

 conv2d_1 (Conv2D)           (None, 7, 9, 128)         73856

 max_pooling2d_1 (MaxPoolin  (None, 4, 5, 128)         0
 g2D)

 dropout_1 (Dropout)         (None, 4, 5, 128)         0

 conv2d_2 (Conv2D)           (None, 2, 2, 128)         147584
```

```
 max_pooling2d_2 (MaxPoolin   (None, 1, 1, 128)         0
 g2D)

 dropout_2 (Dropout)          (None, 1, 1, 128)         0

 flatten (Flatten)            (None, 128)               0

 dense (Dense)                (None, 256)               33024

 dropout_3 (Dropout)          (None, 256)               0

 dense_1 (Dense)              (None, 1)                 257

=================================================================
Total params: 256513 (1002.00 KB)
Trainable params: 256513 (1002.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.evaluate(test_dataset)
```

```
2/2 [==============================] - 9s 57ms/step
- loss: 0.1645 - accuracy: 0.9235
[0.1644611805677414, 0.9234693646430969]
```

## 5.4 Additional modeling - CNN + RNN

: CNN과 RNN 모델을 하나로 합친 CNN-RNN architecture을 이용한 모델을 구현하였다. 1D 벡터로 펼쳐져 있는 시계열 데이터를 사용하기 때문에 Conv1D layer로 구성된 CNN과, GRU layer로 구성된 RNN을 이용하였다. GRU란 두 개의 게이트를 사용하여 LSTM 네트워크를 개선한 모델로, 빠른 계산 시간과 낮은 계산 복잡성을 가지는 장점이 있다.

train, validation, test 이미지를 layer에 알맞은 사이즈로 reshape한 후, 한 번의 Conv1D와 maxpooling layer을 적용하였다. CNN에서 activation function은 ReLu을 사용하였다. 그 후 GRU layer을 적용해줌으로써 RNN을 추가하였다. 마지막 activation function은 sigmoid을 사용하였다. loss값은 sparse_categorical_crossentropy, 옵티마이저는 adam으로 지정해주었다.

```
from tensorflow.keras import layers

# reshape input data appropriately for the GRU layer
# Flatten spatial dimensions

X_train_flat = X_train.reshape((X_train.shape[0], -1,
X_train.shape[-1]))
X_val_flat = X_val.reshape((X_val.shape[0], -1, X_val.shape[-1]))
X_test_flat = X_test.reshape((X_test.shape[0], -1, X_test.shape[-1]))

# Utility for our sequence model.
```

```
def get_model(input_shape, num_classes):
    frame_features_input = keras.Input(shape=(X_train_flat.shape[1],
X_train_flat.shape[2]))

    # Add a Convolutional layer for spatial feature extraction
    x = layers.Conv1D(32, kernel_size=3,
activation='relu')(frame_features_input)
    x = layers.MaxPooling1D(pool_size=2)(x)

    # ADD RNN
    x = layers.GRU(16)(frame_features_input)
    x = layers.Dropout(0.5)(x)
    x = layers.Dense(64, activation="relu")(x)
    output = layers.Dense(num_classes, activation="softmax")(x)

    cnn_rnn_model = keras.Model(inputs=frame_features_input,
outputs=output)

    cnn_rnn_model.compile(
        loss="sparse_categorical_crossentropy", optimizer="adam",
metrics=["accuracy"]
    )
    return cnn_rnn_model

input_shape = (64, 64, 3)
num_classes = 2  # Binary classification for patients and controls

# Example usage
cnn_rnn_model = get_model(input_shape, num_classes)
cnn_rnn_model.summary()
```

모델을 학습시키는 과정에서는, ModelCheckpoint 콜백을 사용하여 모델 가중치를 저장하였다. 이 콜백은 모델의 특정 지표가 향상되면 모델을 저장하여 나중에 재사용하거나 성능을 모니터링할 수 있게 한다.
그 결과 test accuracy는 0.6286정도를 보였다.

```
# Utility for running experiments.
def run_experiment():
    filepath = "/tmp/video_classifier"
    checkpoint = keras.callbacks.ModelCheckpoint(
        filepath, save_weights_only=True, save_best_only=True,
verbose=1
```

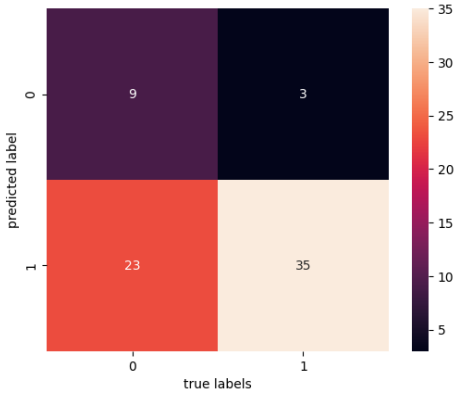
```
    )

    pa_model = get_model(input_shape, num_classes)
# Pass input_shape and num_classes to the function
    history = pa_model.fit(
        X_train_flat,  # Use X_train_flat instead of X_train
        y_train,
        validation_split=0.5,
        epochs=EPOCHS,
        callbacks=[checkpoint],
    )

    #pa_model.load_weights(filepath)
    _, accuracy = pa_model.evaluate(X_test_flat, y_test)
    # Use X_test_flat instead of X_test
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")

    return history, pa_model

_, pa_model = run_experiment()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 64, 64)]          0

 gru (GRU)                   (None, 16)                3936

 dropout (Dropout)           (None, 16)                0

 dense (Dense)               (None, 64)                1088

 dense_1 (Dense)             (None, 2)                 130

=================================================================
Total params: 5154 (20.13 KB)
Trainable params: 5154 (20.13 KB)
Non-trainable params: 0 (0.00 Byte)

Epoch 100: val_loss did not improve from 0.68988
9/9 [==============================] - 0s 41ms/step - loss: 0.6832 - accuracy: 0.5468 - val_loss: 0.7016 - val_accuracy: 0.4604
3/3 [==============================] - 0s 13ms/step - loss: 0.6824 - accuracy: 0.6286
Test accuracy: 62.86%
```

classification_report 함수를 사용하여 분류 모델의 주요 분류 지표를 보여주는 텍스트 리포트를 생성하였고, confusion matrix를 생성하기 위해 predict함수를 사용하여 predict 결과를 담은 결과를 predicted에 저장하였다. confusion matrix를 통해 본 모델의 분류 오류를 시각화하였다.

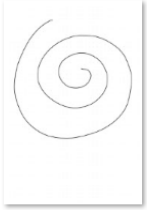
```
#Building the Decision Tree Model on our dataset
from sklearn.tree import DecisionTreeRegressor
predicted = pa_model.predict(X_test_flat)

# 각 행에서 더 큰 값의 인덱스를 추출하여 1 또는 0으로 구성된 리스트 생성
predicted_classes = (predicted[:, 1]>predicted[:,0]).astype(int)
from sklearn.metrics import classification_report


print(classification_report(y_test, predicted_classes))
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# plot the confusion matrix
mat = confusion_matrix(y_test, predicted_classes)
sns.heatmap(mat.T, square = True, annot=True, xticklabels =
sorted(set(y_test)), yticklabels = sorted(set(y_test)), fmt = "d")
plt.xlabel("true labels")
plt.ylabel("predicted label")
plt.show()
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.75 | 0.28 | 0.41 | 32 |
| 1 | 0.60 | 0.92 | 0.73 | 38 |
| accuracy | | | 0.63 | 70 |
| macro avg | 0.68 | 0.60 | 0.57 | 70 |
| weighted avg | 0.67 | 0.63 | 0.58 | 70 |




## 6. Real Case Study

: 가장 성능이 좋았던 CNN 모델로 Real case study를 진행했다. data는 아래 사진과 같이 팀원들이 실제로 그린 아르키메데스 나선 이미지이다.


KakaoTalk_20231128_014253727.jpg


KakaoTalk_20231128_014406966.jpg


KakaoTalk_20231128_014539529.jpg


KakaoTalk_20231128_014621713.jpg

위 데이터를 증강하여 16개의 데이터로 만들어 주었고, 팀원들 모두 파킨슨 환자가 아니므로 모두 controls로 labeling 해주었다. patients 폴더에는 임시로 16개의 데이터를 넣어주었다. 약 0.9의 accuracy를 보여서 모델이 상당히 잘 동작하고 있음을 알 수 있었다.

```
case_dataset = keras.utils.image_dataset_from_directory(
    "./case_application_dataset", labels="inferred",
label_mode="binary", image_size=(120, 160), batch_size=128,
class_names=["controls", "patients"]
)
def normalize_images(image, label):
    return image / 255.0, label

case_dataset = case_dataset.map(normalize_images)



model.evaluate(case_dataset)
```

```
1/1 [==============================] - 0s 445ms/step - loss: 0.2114 -
accuracy: 0.9062
[0.21136339008808136, 0.90625]
```

## 7. Conclusion

: 앞서 실험한 모델 MLP, RNN, CNN, CNN+RNN의 실행 결과에서 test accuracy는 0.9235가, real case study accuracy는 0.9062로 CNN이 가장 좋은 성능을 보였다. 다시 말해 도형 그림 데이터만을 활용해 파킨슨병을 진단하는 모델을 개발했고 성공적으로 작동했다는 것이다. 본 프로젝트는 전문장비와 의료전문가 없이도 누구나 간편하게 파킨슨병을 조기 감지하고 예방할 수 있는 가능성이 열렸다는 데에 의의가 있다. 하지만 학습 데이터의 양이 증강을 거쳤음에도 실용화하기에는 충분하지 않다는 점에서 상용화에는 정확도가 떨어질 수 있다. 향후에 사용자가 더욱 편리하게 모델을 사용할 수 있도록 하기 위해서는 앱 또는 웹 기반의 인터페이스를 개발해 더 많은 사용자를 모집하여 모델의 신뢰성을 높여야할 것이다.

2023학년도 2학기 <의학영상처리>

상호 평가 점수

| | |
|---|---|
| Introduction | 홍지우, 오예인, 윤서영, 홍지영 |
| Related Work | 오예인, 윤서영 |
| Data | 홍지우, 오예인, 윤서영, 홍지영 |
| Model | 홍지우, 오예인, 윤서영, 홍지영 |
| Modeling | 홍지우, 오예인, 윤서영, 홍지영 |
| Real Case Study | 홍지영 |
| Conclusion | 홍지우, 오예인, 윤서영, 홍지영 |

홍지우, 오예인, 윤서영, 홍지영
25%, 25%, 25%, 25%