- 사람의 활동을 인식하는 1D CNN(1차원 컨볼루션 신경망)을 구축하고 학습시키는 데 사용
- 주요 단계: 데이터 전처리, 모델 구성, 학습 및 평가, 결과 시각화

1. 라이브러리 임포트

In [ ]:
```
import tensorflow as tf # TensorFlow를 사용하여 딥러닝 모델 구축 및 학습
import numpy as np # NumPy로 수치 연산을 수행
import matplotlib.pyplot as plt # 결과를 시각화
```

2. 데이터 파싱 및 전처리

In [ ]:
```
def parse_end(s):
    try:
        return float(s[-1])
    except:
        return np.nan
```

> - 입력 문자열의 마지막 값을 추출하여 `float`으로 변환, 만약 변환이 실패하면 `NaN`을 반환

In [ ]:
```
# columns: 'user', 'activity', 'timestamp', 'x-accl', 'y-accl', 'z-accl';
def read_data(file_path):
    labels = {'Walking': 0,
              'Jogging': 1,
              'Upstairs': 2,
              'Sitting': 3,
              'Downstairs': 4,
              'Standing': 5}
    data = np.loadtxt(file_path,
                      delimiter=",",
                      usecols=(0,1,3,4,5),
                      converters={1: lambda name: labels[name.decode()], 5: parse_
    data = data[~np.isnan(data).any(axis=1)]
    return data
```

> - 이 함수는 CSV 파일에서 데이터를 읽어오고, 특정 열(`user`, `activity`, `x-accl`, `y-accl`, `z-accl`)만 선택
> - `activity` 컬럼의 문자열 값을 `labels` 딕셔너리를 이용해 정수로 변환
> - `NaN` 값을 포함하는 행을 제거한 후, 전처리된 데이터를 반환

3. 데이터 로드 및 정규화

In [ ]:
```
data = read_data("./DATA/WISDM_ar_v1.1/WISDM_ar_v1.1_raw.txt")

mean = np.mean(data[:,2:], axis=0)
std = np.std(data[:,2:], axis=0)
data[:,2:] = (data[:,2:] - mean) / std
```

> - `read_data` 함수를 사용해 데이터를 불러온 후, `x`, `y`, `z` 가속도 데이터를 정규화 함

4. 데이터 분할 및 세그먼트화

- 신호 처리에서 주로 사용하는 방법
- 원시 데이터를 일정한 길이의 "세그먼트"로 나누고, 각각의 세그먼트를 모델에 입력할 수 있는 형식으로 변환하는 과정
- 이 과정은 시계열 데이터를 처리할 때 매우 중요한 단계
- 세그먼터(Segment): 데이터를 일정한 길이로 자른 부분, 각 세그먼트는 일정한 시간 동안의 데이터 포인트를 포함하며, 이 경우 `TIME_PERIODS`라는 변수로 세그먼트의 길이를 정의
- 스템 거리(STEP_DISTANCE): 세그먼트를 자를 때 겹치는 부분을 얼마나 유지할지를 결정하는 값, 이 값을 통해 데이터의 중복이 결정됨
- 원-핫 인코딩(One-Hot Encoding): 다중 클래스 레이블을 이진 벡터로 변환하는 방법, ex) 6개의 클래스가 있다면 각 클래스는 길이 6의 벡터에서 하나의 값만 1이고 나머지는 0인 벡터로 표시

In [ ]:
```
x_train = data[data[:,0] <= 28] #[28, 36]
x_test = data[data[:,0] > 28]
```

> - 사용자 ID를 기준으로 데이터를 훈련 데이터(`x_train`)와 테스트 데이터(`x_test`)로 나눔

In [ ]:
```
TIME_PERIODS = 80 # 1초 동안 80개의 데이터 포인터가 나타남
STEP_DISTANCE = 40 # 세그먼트를 생성할 때 40개의 데이터 포인트마다 세그먼트를 시작

#4. 데이터 세그먼트화 및 라벨링 (-1, TIME_PERIODS, 3)
def data_segments(data):
    segments = []
    labels = []
    for i in range(0, len(data) - TIME_PERIODS, STEP_DISTANCE): #  데이터를 순차적
        X = data[i:i+TIME_PERIODS, 2:].tolist() # 세그먼트 내의 데이터를 슬라이싱

        # 해당 세그먼트 내의 활동 라벨들을 확인하고, 각각의 라벨이 몇 번 나타나는
        values, counts = np.unique(data[i:i+TIME_PERIODS, 1], return_counts=True)
        # 가장 빈번하게 나타나는 활동 라벨을 이 세그먼트의 대표 라벨로 지정
        label = values[np.argmax(counts)]

        segments.append(X)
        labels.append(label)
```

```
    # reshape (-1, TIME_PERIODS, 3)
    # -1은 자동으로 배열의 첫 번째 차원을 결정, 두 번째 차원은 TIME_PERIODS(세그먼
    segments = np.array(segments, dtype=np.float32).reshape(-1, TIME_PERIODS, 3) #
    labels = np.asarray(labels)
    return segments, labels
```

> - 데이터를 `TIME_PERIODS` 만큼의 길이로 세그먼트화하고, 세그먼트 내에서 가장 빈번한 활동으로 라벨링
> - 최종적으로 `(세그먼트 수, TIME_PERIODS, 3)` 형태의 데이터 배열과 라벨 배열을 반환
> - 이유: 신경망에 입력되는 데이터는 (배치 크기, 타임 스템, 피처 수) 형식, `reshape`를 사용해 데이터를 이 형식에 맞게 조정

In [ ]:
```
x_train, y_train = data_segments(x_train)
x_test, y_test = data_segments(x_test)
```

> - 훈련 데이터와 테스트 데이터를 세그먼트화하여 모델 학습에 사용할 준비를 함

5. 라벨 원-핫 인코딩

In [ ]:
```
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
```

> - 라벨 데이터를 원-핫 인코딩하여 다중 클래스 분류가 가능하도록 변환

6. 1D CNN 모델 생성

- 1D CNN 모델은 시계열 데이터나 연속적인 데이터를 처리하는데 매우 효과적
- 각 데이터 포인트 간의 지역적 상관관계를 학습하는 데 주로 사용되며, 특히 시간 축에 따라 패턴을 인식하는 데 유용
- 특히 시간 축에 따라 패턴을 인식하는 데 유용

In [ ]:
```
model = tf.keras.Sequential()
# 모델의 입력 형태를 정의, `TIME_PERIODS`=80 길이의 시계열 데이터를 받고, 각 타임
model.add(tf.keras.layers.Input(shape=(TIME_PERIODS, 3)))
# 100개의 필터를 사용하여 입력 데이터를 컨볼루션함, 11개의 연속된 데이터 포인트를
model.add(tf.keras.layers.Conv1D(filters=100, kernel_size=11, activation='relu'))
# 맥스 풀링 층은 컨볼루션 층의 출력에서 가장 큰 값을 추출하여 데이터의 크기를 줄이
model.add(tf.keras.layers.MaxPool1D())
# 각 배치마다 활성화 값을 정규화하여 학습 과정을 안정화하고 가속화
model.add(tf.keras.layers.BatchNormalization())

# 두 번째 컨볼루션 층은 앞서 추출된 특징을 기반으로 더 작은 범위의 패턴을 학습, 10
model.add(tf.keras.layers.Conv1D(filters=10, kernel_size=5, activation='relu'))
```

```
# 중요한 특징을 강조하며 데이터의 크기를 줄임
model.add(tf.keras.layers.MaxPool1D())
# 학습 과정에서 일부 뉴런을 무작위로 비활성화, rate=0.5는 전체 뉴련의 50%를 드롭아
model.add(tf.keras.layers.Dropout(rate=0.5))

# 다차원 배열을 일차원으로 변환, 완전 연결 층(Dense Layer)에 데이터를 전달하기 위
model.add(tf.keras.layers.Flatten())
# 6개의 뉴런을 가진 완전 연결층, `softmax`활성화 함수를 사용해 출력값을 다중 클래스
model.add(tf.keras.layers.Dense(units=6, activation='softmax'))
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param
conv1d (Conv1D)	(None, 70, 100)	3,4
max_pooling1d (MaxPooling1D)	(None, 35, 100)	
batch_normalization (BatchNormalization)	(None, 35, 100)	4
conv1d_1 (Conv1D)	(None, 31, 10)	5,0
max_pooling1d_1 (MaxPooling1D)	(None, 15, 10)	
dropout (Dropout)	(None, 15, 10)	
flatten (Flatten)	(None, 150)	
dense (Dense)	(None, 6)	9

Total params: 9,716 (37.95 KB)

Trainable params: 9,516 (37.17 KB)

Non-trainable params: 200 (800.00 B)

히든 레이어(은닉층)의 역할

- 컨볼루션 층과 맥스 풀링 층은 모델의 히든 레이어 역할을 함
- 히든 레이어는 입력 데이터에서 점점 더 높은 수준의 특징을 학습하도록 도와줌
- 각 히든 레이어는 이전 레이어의 출력에서 중요한 패턴을 학습하며, 이 패턴들이 쌓여 최종적으로 데이터의 분류를 수행하는 데 필요한 복잡한 특징을 학습

  > - 컨볼루션 층: 입력 데이터의 국소적인 패턴을 학습
  > - 맥스 풀링 층: 이 패턴에서 중요한 부분을 선택하고, 데이터의 크기를 줄여 다음 층에서 효율적으로 처리
  > - 배치 정규화 층: 학습 과정을 안정화하고 가속화
  > - 드롭아웃 층: 과적합을 방지하여 모델의 일반화 성능을 향상

- 히든 레이어들을 통해 모델은 입력 데이터를 점진적으로 높은 수준의 특징으로 변환하며, 마지막에는 이 특징들을 기반으로 입력 데이터의 클래스(활동)를 예측

7. 모델 컴파일 및 학습

In [ ]:
```
# RMSprop (Root Mean Square Propagation) 옵티마이저
opt = tf.keras.optimizers.RMSprop(learning_rate=0.01)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy']
ret = model.fit(x_train, y_train, epochs=100, batch_size=400,
                validation_data=(x_test, y_test), verbose=2)
```

```
Epoch 1/100
53/53 - 5s - 102ms/step - accuracy: 0.6235 - loss: 1.3124 - val_accuracy: 0.5272 -
val_loss: 1.3405
Epoch 2/100
53/53 - 4s - 69ms/step - accuracy: 0.7295 - loss: 0.7288 - val_accuracy: 0.6739 - v
al_loss: 1.0938
Epoch 3/100
53/53 - 3s - 63ms/step - accuracy: 0.7563 - loss: 0.6166 - val_accuracy: 0.5293 - v
al_loss: 1.6130
Epoch 4/100
53/53 - 3s - 65ms/step - accuracy: 0.8034 - loss: 0.5100 - val_accuracy: 0.6428 - v
al_loss: 1.3745
Epoch 5/100
53/53 - 4s - 67ms/step - accuracy: 0.8501 - loss: 0.4705 - val_accuracy: 0.7485 - v
al_loss: 0.9446
Epoch 6/100
53/53 - 3s - 64ms/step - accuracy: 0.8807 - loss: 0.3633 - val_accuracy: 0.7464 - v
al_loss: 1.2955
Epoch 7/100
53/53 - 3s - 66ms/step - accuracy: 0.8909 - loss: 0.3455 - val_accuracy: 0.7257 - v
al_loss: 0.9908
Epoch 8/100
53/53 - 4s - 73ms/step - accuracy: 0.8957 - loss: 0.3146 - val_accuracy: 0.8316 - v
al_loss: 1.0873
Epoch 9/100
53/53 - 4s - 68ms/step - accuracy: 0.9069 - loss: 0.2806 - val_accuracy: 0.7868 - v
al_loss: 0.9039
Epoch 10/100
53/53 - 4s - 68ms/step - accuracy: 0.9052 - loss: 0.2794 - val_accuracy: 0.7792 - v
al_loss: 0.8292
Epoch 11/100
53/53 - 3s - 65ms/step - accuracy: 0.9160 - loss: 0.3006 - val_accuracy: 0.7866 - v
al_loss: 1.3984
Epoch 12/100
53/53 - 4s - 67ms/step - accuracy: 0.9145 - loss: 0.2545 - val_accuracy: 0.7886 - v
al_loss: 1.9175
Epoch 13/100
53/53 - 3s - 64ms/step - accuracy: 0.9241 - loss: 0.2340 - val_accuracy: 0.7637 - v
al_loss: 1.1764
Epoch 14/100
53/53 - 3s - 65ms/step - accuracy: 0.9248 - loss: 0.2566 - val_accuracy: 0.7975 - v
al_loss: 0.7777
Epoch 15/100
53/53 - 3s - 65ms/step - accuracy: 0.9241 - loss: 0.2350 - val_accuracy: 0.7947 - v
al_loss: 0.9083
Epoch 16/100
53/53 - 3s - 62ms/step - accuracy: 0.9315 - loss: 0.2160 - val_accuracy: 0.8024 - v
al_loss: 1.0983
Epoch 17/100
53/53 - 4s - 67ms/step - accuracy: 0.9323 - loss: 0.2164 - val_accuracy: 0.8185 - v
al_loss: 1.4807
Epoch 18/100
53/53 - 4s - 67ms/step - accuracy: 0.9321 - loss: 0.2131 - val_accuracy: 0.8097 - v
al_loss: 0.8809
Epoch 19/100
53/53 - 3s - 64ms/step - accuracy: 0.9312 - loss: 0.2234 - val_accuracy: 0.8030 - v
al_loss: 2.8731
Epoch 20/100
53/53 - 3s - 65ms/step - accuracy: 0.9338 - loss: 0.2447 - val_accuracy: 0.8343 - v
al_loss: 1.0409
```

```
Epoch 21/100
53/53 - 3s - 64ms/step - accuracy: 0.9362 - loss: 0.2046 - val_accuracy: 0.8133 - v
al_loss: 1.5249
Epoch 22/100
53/53 - 4s - 69ms/step - accuracy: 0.9362 - loss: 0.2615 - val_accuracy: 0.8039 - v
al_loss: 1.2440
Epoch 23/100
53/53 - 4s - 68ms/step - accuracy: 0.9368 - loss: 0.2002 - val_accuracy: 0.6651 - v
al_loss: 1.5114
Epoch 24/100
53/53 - 3s - 65ms/step - accuracy: 0.9373 - loss: 0.1976 - val_accuracy: 0.8214 - v
al_loss: 1.0391
Epoch 25/100
53/53 - 4s - 66ms/step - accuracy: 0.9408 - loss: 0.1883 - val_accuracy: 0.7945 - v
al_loss: 1.0370
Epoch 26/100
53/53 - 3s - 65ms/step - accuracy: 0.9413 - loss: 0.1893 - val_accuracy: 0.7632 - v
al_loss: 1.3628
Epoch 27/100
53/53 - 3s - 65ms/step - accuracy: 0.9400 - loss: 0.2297 - val_accuracy: 0.8159 - v
al_loss: 1.1417
Epoch 28/100
53/53 - 3s - 63ms/step - accuracy: 0.9442 - loss: 0.1800 - val_accuracy: 0.8317 - v
al_loss: 1.5888
Epoch 29/100
53/53 - 3s - 65ms/step - accuracy: 0.9408 - loss: 0.1872 - val_accuracy: 0.8077 - v
al_loss: 1.1485
Epoch 30/100
53/53 - 4s - 71ms/step - accuracy: 0.9437 - loss: 0.1806 - val_accuracy: 0.8420 - v
al_loss: 1.1994
Epoch 31/100
53/53 - 4s - 68ms/step - accuracy: 0.9427 - loss: 0.1906 - val_accuracy: 0.8234 - v
al_loss: 0.8927
Epoch 32/100
53/53 - 4s - 70ms/step - accuracy: 0.9440 - loss: 0.1767 - val_accuracy: 0.8364 - v
al_loss: 0.6945
Epoch 33/100
53/53 - 3s - 66ms/step - accuracy: 0.9447 - loss: 0.1948 - val_accuracy: 0.7734 - v
al_loss: 1.2355
Epoch 34/100
53/53 - 4s - 67ms/step - accuracy: 0.9429 - loss: 0.1842 - val_accuracy: 0.8200 - v
al_loss: 1.5008
Epoch 35/100
53/53 - 4s - 68ms/step - accuracy: 0.9469 - loss: 0.1685 - val_accuracy: 0.7700 - v
al_loss: 1.5770
Epoch 36/100
53/53 - 3s - 64ms/step - accuracy: 0.9441 - loss: 0.1782 - val_accuracy: 0.8285 - v
al_loss: 1.0115
Epoch 37/100
53/53 - 3s - 64ms/step - accuracy: 0.9444 - loss: 0.1771 - val_accuracy: 0.8387 - v
al_loss: 0.7391
Epoch 38/100
53/53 - 3s - 66ms/step - accuracy: 0.9476 - loss: 0.1697 - val_accuracy: 0.8376 - v
al_loss: 1.4682
Epoch 39/100
53/53 - 4s - 68ms/step - accuracy: 0.9470 - loss: 0.1702 - val_accuracy: 0.7543 - v
al_loss: 1.3740
Epoch 40/100
53/53 - 4s - 66ms/step - accuracy: 0.9452 - loss: 0.1737 - val_accuracy: 0.8235 - v
al_loss: 1.5770
```

```
Epoch 41/100
53/53 - 3s - 66ms/step - accuracy: 0.9482 - loss: 0.1613 - val_accuracy: 0.8351 - v
al_loss: 1.3337
Epoch 42/100
53/53 - 3s - 66ms/step - accuracy: 0.9498 - loss: 0.1634 - val_accuracy: 0.8594 - v
al_loss: 0.6603
Epoch 43/100
53/53 - 3s - 65ms/step - accuracy: 0.9482 - loss: 0.1690 - val_accuracy: 0.7937 - v
al_loss: 1.2343
Epoch 44/100
53/53 - 3s - 65ms/step - accuracy: 0.9499 - loss: 0.1832 - val_accuracy: 0.8076 - v
al_loss: 0.7794
Epoch 45/100
53/53 - 4s - 66ms/step - accuracy: 0.9504 - loss: 0.1538 - val_accuracy: 0.8378 - v
al_loss: 0.9208
Epoch 46/100
53/53 - 3s - 65ms/step - accuracy: 0.9487 - loss: 0.1616 - val_accuracy: 0.8187 - v
al_loss: 1.4928
Epoch 47/100
53/53 - 4s - 67ms/step - accuracy: 0.9508 - loss: 0.1620 - val_accuracy: 0.8206 - v
al_loss: 1.0339
Epoch 48/100
53/53 - 4s - 66ms/step - accuracy: 0.9507 - loss: 0.1592 - val_accuracy: 0.8212 - v
al_loss: 0.8660
Epoch 49/100
53/53 - 4s - 66ms/step - accuracy: 0.9495 - loss: 0.1618 - val_accuracy: 0.8344 - v
al_loss: 0.8419
Epoch 50/100
53/53 - 4s - 66ms/step - accuracy: 0.9492 - loss: 0.1589 - val_accuracy: 0.8074 - v
al_loss: 0.6478
Epoch 51/100
53/53 - 3s - 65ms/step - accuracy: 0.9497 - loss: 0.1624 - val_accuracy: 0.8382 - v
al_loss: 0.7978
Epoch 52/100
53/53 - 4s - 66ms/step - accuracy: 0.9521 - loss: 0.1530 - val_accuracy: 0.7957 - v
al_loss: 1.1482
Epoch 53/100
53/53 - 3s - 66ms/step - accuracy: 0.9497 - loss: 0.1573 - val_accuracy: 0.7860 - v
al_loss: 1.3143
Epoch 54/100
53/53 - 3s - 66ms/step - accuracy: 0.9532 - loss: 0.1546 - val_accuracy: 0.8259 - v
al_loss: 1.2755
Epoch 55/100
53/53 - 4s - 67ms/step - accuracy: 0.9521 - loss: 0.1556 - val_accuracy: 0.8199 - v
al_loss: 1.3108
Epoch 56/100
53/53 - 4s - 70ms/step - accuracy: 0.9530 - loss: 0.1551 - val_accuracy: 0.8373 - v
al_loss: 1.1221
Epoch 57/100
53/53 - 4s - 76ms/step - accuracy: 0.9529 - loss: 0.1501 - val_accuracy: 0.8326 - v
al_loss: 1.2122
Epoch 58/100
53/53 - 3s - 64ms/step - accuracy: 0.9531 - loss: 0.1485 - val_accuracy: 0.8098 - v
al_loss: 1.3292
Epoch 59/100
53/53 - 3s - 63ms/step - accuracy: 0.9513 - loss: 0.1580 - val_accuracy: 0.8050 - v
al_loss: 1.7334
Epoch 60/100
53/53 - 4s - 69ms/step - accuracy: 0.9528 - loss: 0.1514 - val_accuracy: 0.8240 - v
al_loss: 1.4883
```

```
Epoch 61/100
53/53 - 3s - 65ms/step - accuracy: 0.9524 - loss: 0.1541 - val_accuracy: 0.8279 - v
al_loss: 1.1641
Epoch 62/100
53/53 - 3s - 64ms/step - accuracy: 0.9510 - loss: 0.1576 - val_accuracy: 0.8375 - v
al_loss: 0.9982
Epoch 63/100
53/53 - 3s - 65ms/step - accuracy: 0.9536 - loss: 0.1496 - val_accuracy: 0.8170 - v
al_loss: 1.3592
Epoch 64/100
53/53 - 3s - 66ms/step - accuracy: 0.9535 - loss: 0.1495 - val_accuracy: 0.8228 - v
al_loss: 1.2538
Epoch 65/100
53/53 - 3s - 66ms/step - accuracy: 0.9536 - loss: 0.1476 - val_accuracy: 0.8288 - v
al_loss: 2.2673
Epoch 66/100
53/53 - 4s - 66ms/step - accuracy: 0.9532 - loss: 0.1475 - val_accuracy: 0.8259 - v
al_loss: 0.9479
Epoch 67/100
53/53 - 3s - 66ms/step - accuracy: 0.9573 - loss: 0.1424 - val_accuracy: 0.8287 - v
al_loss: 1.5098
Epoch 68/100
53/53 - 4s - 68ms/step - accuracy: 0.9533 - loss: 0.1528 - val_accuracy: 0.7653 - v
al_loss: 1.4380
Epoch 69/100
53/53 - 3s - 64ms/step - accuracy: 0.9534 - loss: 0.1469 - val_accuracy: 0.7846 - v
al_loss: 1.6865
Epoch 70/100
53/53 - 3s - 66ms/step - accuracy: 0.9543 - loss: 0.1573 - val_accuracy: 0.7962 - v
al_loss: 1.8342
Epoch 71/100
53/53 - 4s - 68ms/step - accuracy: 0.9556 - loss: 0.1514 - val_accuracy: 0.8296 - v
al_loss: 0.8009
Epoch 72/100
53/53 - 4s - 68ms/step - accuracy: 0.9551 - loss: 0.1424 - val_accuracy: 0.8326 - v
al_loss: 0.8728
Epoch 73/100
53/53 - 3s - 65ms/step - accuracy: 0.9545 - loss: 0.1466 - val_accuracy: 0.8337 - v
al_loss: 1.1999
Epoch 74/100
53/53 - 4s - 67ms/step - accuracy: 0.9542 - loss: 0.1462 - val_accuracy: 0.8065 - v
al_loss: 1.2412
Epoch 75/100
53/53 - 4s - 73ms/step - accuracy: 0.9544 - loss: 0.1475 - val_accuracy: 0.8325 - v
al_loss: 1.3678
Epoch 76/100
53/53 - 3s - 66ms/step - accuracy: 0.9558 - loss: 0.1428 - val_accuracy: 0.8071 - v
al_loss: 0.9519
Epoch 77/100
53/53 - 3s - 63ms/step - accuracy: 0.9549 - loss: 0.1614 - val_accuracy: 0.8108 - v
al_loss: 2.0948
Epoch 78/100
53/53 - 3s - 64ms/step - accuracy: 0.9541 - loss: 0.1500 - val_accuracy: 0.8433 - v
al_loss: 0.8419
Epoch 79/100
53/53 - 3s - 64ms/step - accuracy: 0.9556 - loss: 0.1451 - val_accuracy: 0.8132 - v
al_loss: 1.0847
Epoch 80/100
53/53 - 3s - 66ms/step - accuracy: 0.9567 - loss: 0.1373 - val_accuracy: 0.8106 - v
al_loss: 1.0479
```

```
Epoch 81/100
53/53 - 3s - 66ms/step - accuracy: 0.9547 - loss: 0.1476 - val_accuracy: 0.8112 - v
al_loss: 1.5327
Epoch 82/100
53/53 - 3s - 62ms/step - accuracy: 0.9567 - loss: 0.1410 - val_accuracy: 0.8384 - v
al_loss: 0.9048
Epoch 83/100
53/53 - 3s - 63ms/step - accuracy: 0.9572 - loss: 0.1414 - val_accuracy: 0.8197 - v
al_loss: 1.4630
Epoch 84/100
53/53 - 4s - 69ms/step - accuracy: 0.9551 - loss: 0.1454 - val_accuracy: 0.8354 - v
al_loss: 1.9333
Epoch 85/100
53/53 - 4s - 73ms/step - accuracy: 0.9555 - loss: 0.1439 - val_accuracy: 0.8142 - v
al_loss: 1.5286
Epoch 86/100
53/53 - 3s - 64ms/step - accuracy: 0.9570 - loss: 0.1390 - val_accuracy: 0.8392 - v
al_loss: 1.2435
Epoch 87/100
53/53 - 3s - 63ms/step - accuracy: 0.9593 - loss: 0.1319 - val_accuracy: 0.7899 - v
al_loss: 2.2095
Epoch 88/100
53/53 - 3s - 65ms/step - accuracy: 0.9564 - loss: 0.1384 - val_accuracy: 0.8095 - v
al_loss: 1.2640
Epoch 89/100
53/53 - 3s - 64ms/step - accuracy: 0.9563 - loss: 0.1442 - val_accuracy: 0.8401 - v
al_loss: 1.6403
Epoch 90/100
53/53 - 3s - 62ms/step - accuracy: 0.9575 - loss: 0.1370 - val_accuracy: 0.8259 - v
al_loss: 1.4935
Epoch 91/100
53/53 - 4s - 68ms/step - accuracy: 0.9560 - loss: 0.1412 - val_accuracy: 0.8244 - v
al_loss: 1.8370
Epoch 92/100
53/53 - 3s - 63ms/step - accuracy: 0.9571 - loss: 0.1430 - val_accuracy: 0.8364 - v
al_loss: 1.8224
Epoch 93/100
53/53 - 3s - 63ms/step - accuracy: 0.9581 - loss: 0.1445 - val_accuracy: 0.7925 - v
al_loss: 1.7559
Epoch 94/100
53/53 - 3s - 64ms/step - accuracy: 0.9547 - loss: 0.1444 - val_accuracy: 0.8287 - v
al_loss: 1.9348
Epoch 95/100
53/53 - 4s - 67ms/step - accuracy: 0.9532 - loss: 0.1480 - val_accuracy: 0.8279 - v
al_loss: 1.4749
Epoch 96/100
53/53 - 3s - 63ms/step - accuracy: 0.9579 - loss: 0.1355 - val_accuracy: 0.8395 - v
al_loss: 2.3559
Epoch 97/100
53/53 - 3s - 62ms/step - accuracy: 0.9595 - loss: 0.1362 - val_accuracy: 0.8299 - v
al_loss: 1.4046
Epoch 98/100
53/53 - 3s - 64ms/step - accuracy: 0.9587 - loss: 0.1295 - val_accuracy: 0.8220 - v
al_loss: 1.4238
Epoch 99/100
53/53 - 3s - 65ms/step - accuracy: 0.9563 - loss: 0.1412 - val_accuracy: 0.8433 - v
al_loss: 1.1167
Epoch 100/100
53/53 - 4s - 66ms/step - accuracy: 0.9589 - loss: 0.1332 - val_accuracy: 0.8173 - v
al_loss: 1.4130
```

- 모델을 RMSprop 옵티마이저와 교차 엔트로피 손실 함수로 컴파일하고, 훈련 데이터로 모델을 학습

8. 모델 평가

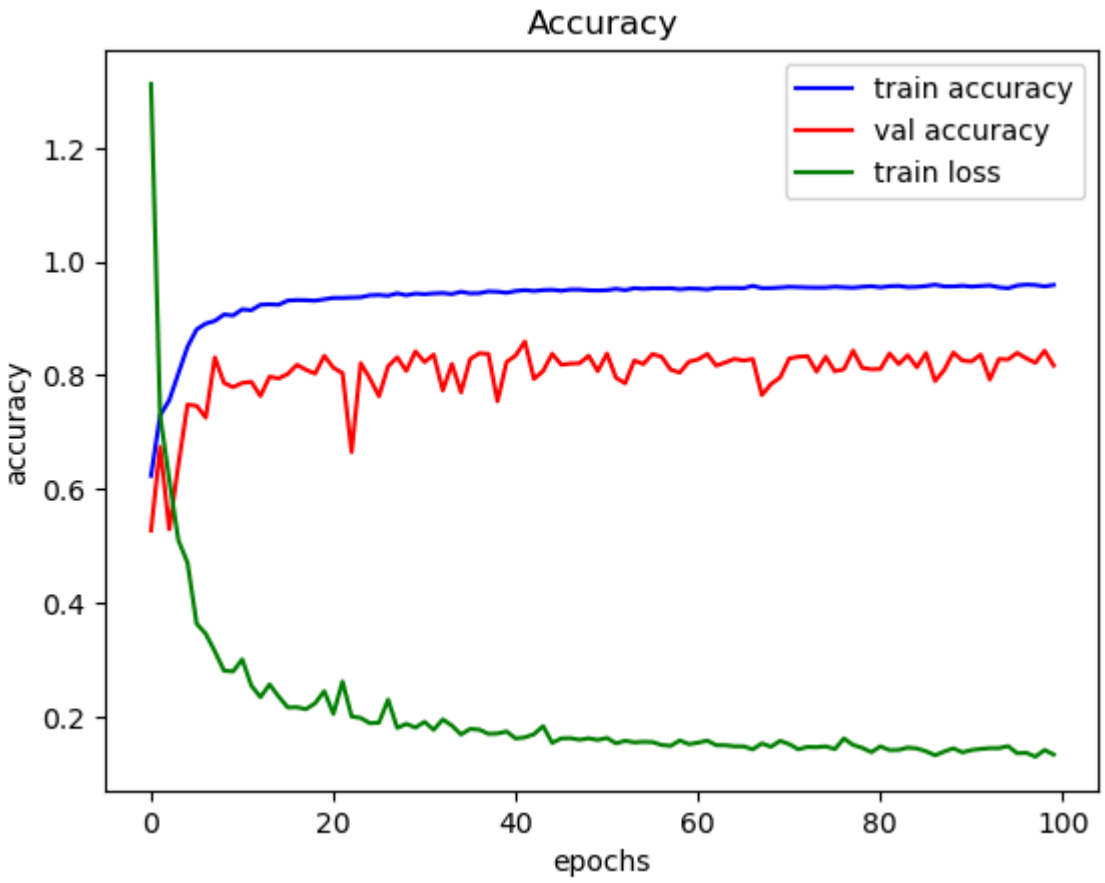
In [ ]:
```
train_loss, train_acc = model.evaluate(x_train, y_train, verbose=2)
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
```

```
653/653 - 1s - 2ms/step - accuracy: 0.9610 - loss: 0.1269
206/206 - 0s - 2ms/step - accuracy: 0.8173 - loss: 1.4130
```

- 훈련 데이터와 테스트 데이터에 대해 모델의 성능을 평가함

9. 정확도 및 손실 시각화

In [ ]:
```
plt.title("Accuracy")
plt.plot(ret.history['accuracy'], "b-", label="train accuracy")
plt.plot(ret.history['val_accuracy'], "r-", label="val accuracy")
plt.plot(ret.history['loss'], "g-", label="train loss")
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend(loc="best")
plt.show()
```



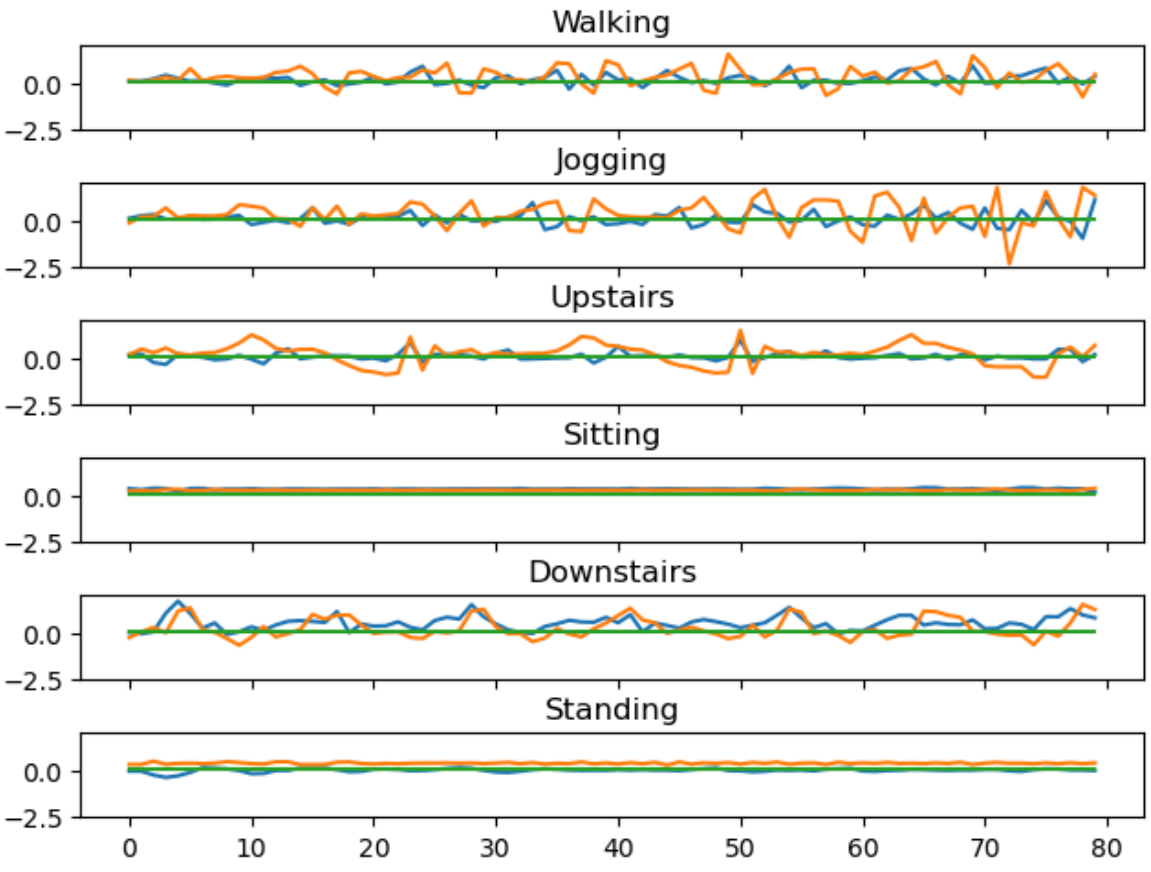
- 학습 과정에서의 정확도와 손실을 시각화하여 모델 성능을 분석

10. 샘플 활동 데이터 시각화

In [ ]:
```
activity = ('Walking', 'Jogging', 'Upstairs', 'Sitting', 'Downstairs', 'Standing')
train_label = np.argmax(y_train, axis=1)

plot_data = []
n = 1
for i in range(6):
    plot_data.append(np.where(train_label == i)[0][n])

fig, ax = plt.subplots(6, sharex=True, sharey=True)
fig.tight_layout()
for i in range(6):
    k = plot_data[i]
    ax[i].plot(x_train[k], label=activity[i])
    ax[i].set_title(activity[i])
plt.show()
```



- 각 활동에 대한 샘플 데이터를 시각화하여 모델이 학습한 결과를 확인

In [ ]:
```
```