FordA 데이터셋을 기반으로 한 ConvNet 모델 자동차 엔진 이상 이진 분류

An Binary Classification of detecting error at mobile engine using ConvNet

- **임규연 (가천대학교 컴퓨터공학부 학부과정, 서울대학교 의생명정보학연구실 학부인턴)**
  - 학번 : 202334734
  - e-mail : lky473736@gmail.com
  - GitHub : https://github.com/lky473736
  - 관심 분야 : Artificial Intelligence, AI for Signal Data, Time-Series Self-Supervised Learning, Human Activity Recognition

- **Keyword** : Anomaly Detection, Convolution Neural Network, Time-series data processing

- abstract -

Using the FordA dataset containing the engine noise measurements captured by the car engine sensor, a binary classification operation is performed with the CNN deep learning model to check if there is a specific problem with the engine.

---

1. introduction

본 문서에서는 FordA 데이터셋을 이용하여 시계열 데이터 기반의 이상 탐지 이진 분류를 진행한다. FordA 데이터셋은 자동차 엔진의 소음을 측정하였으며, 이를 통하여 엔진의 이상 여부를 확인할 수 있다. 기존 머신러닝에서의 시계열 데이터 기반 이상 탐지에서는 주로 특정 window의 파형을 서로 대조해가면서 이상 탐지를 수행하였으나, 일종의 Black-Box인 딥러닝에서는 3차원 데이터 형태로 데이터를 구성하고 Convolution 연산을 통해 window 자체의 특징을 추출한다.

https://keras.io/examples/timeseries/timeseries_classification_from_scratch/ 에서는 본 FordA 데이터셋을 이용하여 ConvNet을 구성하고 categorical crossentropy 손실함수를 두어 이진 분류를 해결하는 튜토리얼을 볼 수 있다. **하지만 나는 본 튜토리얼에서 시계열 처리에서 가장 큰 문제점을 발견하였다.**

- (1) **위 링크의 튜토리얼에서는 ConvNet에 데이터를 넣기 위하여 단순히 (X_train.shape[0], X_train.shape[1], 1)로 데이터를 reshape하였는데, reshape API가 어떻게 시간적 종속성을 반영할 수 있겠는가?** 원래라면, (# of windows, window의 크기, feature의 수)로 input data를 구성하는 것이 바람직하다.

- (2) **(1)에 이어서, 기존 코드에서는 sampling rate를 일절 반영하지 않았다.** sampling rate란 1초마다 센서가 데이터를 몇 번 수집하였는지를 의미한다. FordA의 sampling rate는 500으로 우리는 이것을 반영하여 모델의 입력을 구성해야 한다. (추후 슬라이딩 윈도우 알고리즘을 통해 이를 직접 구현하여 해결한다)

따라서 나는 튜토리얼 코드를 **다시 재구성**해서 처음부터 다시 data processing을 수행할 것이다. 그리하였을 때의 성능 또한 확인하겠다.

본 ConvNet 모델을 구현할 테크는 아래와 같다.

- **데이터 읽기 및 전처리 테크닉** : pandas
- **모델링 및 머신러닝-딥러닝 라이브러리** : scikit-learn, TensorFlow, Keras
- **선형대수학 및 tensor 이용** : numpy

- **데이터 시각화** : matplotlib, seaborn

데이터 분석은 아래와 같은 단계로 진행된다.

- **1) 데이터 구조 파악 및 전처리, 시각화**
  - 데이터의 구조 및 기초 통계량 확인, 특정 attribute에 대한 간단한 시각화를 진행한다.
- **2) ConvNet 기반의 classification 진행 및 성능 지표 출력, 시각화**

In [99]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn as sk
import numpy as np
import tensorflow as tf
```

---

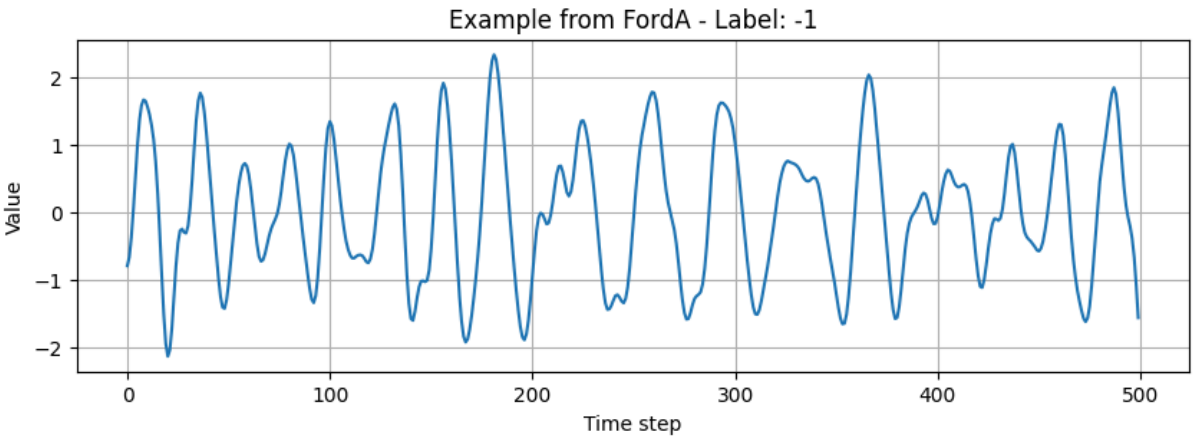
2. 데이터 구조 파악 및 전처리, 시각화

- 본 단계에서는 데이터의 기초적인 통계량 파악, 데이터 전처리 및 각 attribute에 대한 시각화를 진행한다.

In [100…
```
# numpy로 데이터를 읽는다

def readucr(filename):
    data = np.loadtxt(filename, delimiter="\t")
    y = data[:, 0]
    x = data[:, 1:]
    return x, y.astype(int)

root_url = "https://raw.githubusercontent.com/hfawaz/cd-diagram/master/FordA/"

X_train, y_train = readucr(root_url + "FordA_TRAIN.tsv")
X_test, y_test = readucr(root_url + "FordA_TEST.tsv")

print (X_train.shape, X_test.shape)
print (y_train.shape, y_test.shape)
print (set(y_train))
```

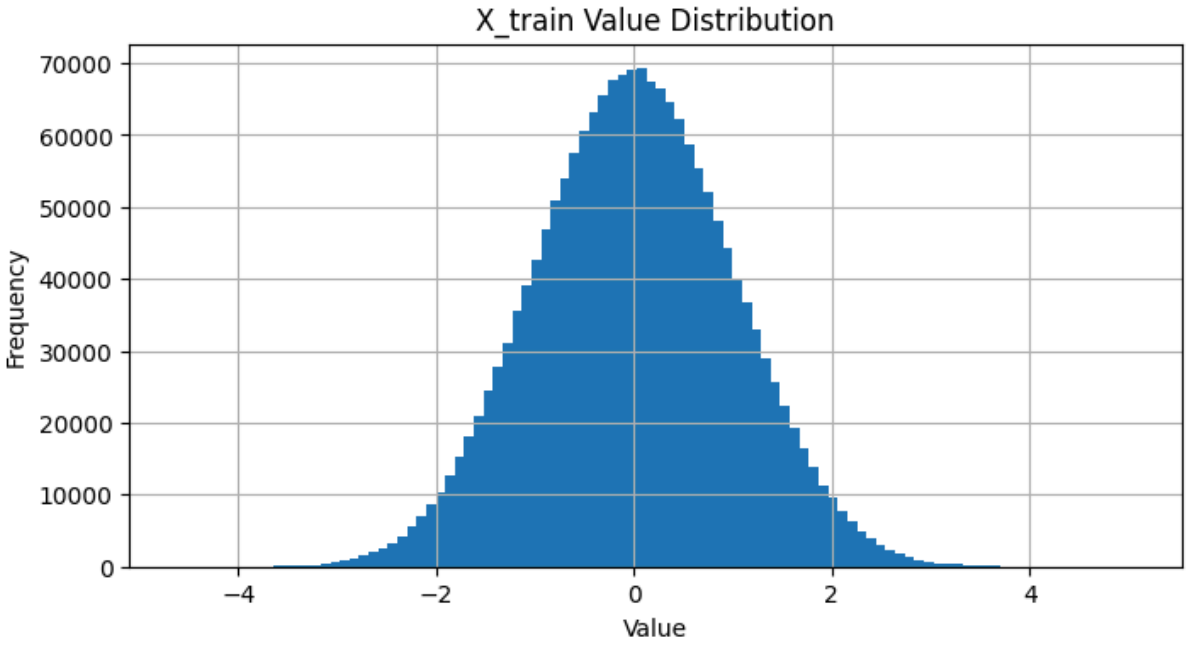
```
(3601, 500) (1320, 500)
(3601,) (1320,)
{1, -1}
```

In [101…
```
plt.figure(figsize=(10, 3))
plt.plot(X_train[0])
plt.title(f"Example from FordA - Label: {y_train[0]}")
plt.xlabel("Time step")
plt.ylabel("Value")
plt.grid()
plt.show()
```



In [102…
```
plt.figure(figsize=(8, 4))
plt.hist(X_train.flatten(), bins=100)
```

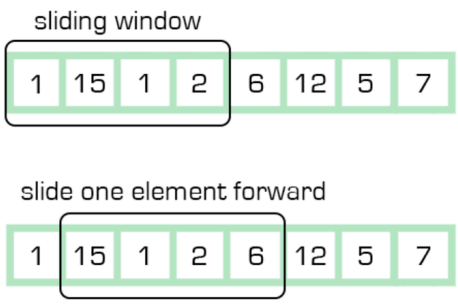
```
plt.title("X_train Value Distribution")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.grid()
plt.show()
```



- 위 출력문을 통하여 본 데이터셋은 feature가 1로 구성된 1차원 시계열 데이터임을 확인 가능하다. 원래 튜토리얼에서는 단지 시간종속성을 고려 안하고 reshape하여 구성하였지만, 난 신호처리에서 많이 사용되는 기법인 sliding window 방식을 채택하여 시계열 데이터를 window형태로 분리하려고 한다.
- y_train의 고윳값을 확인해보니 1과 -1로 확인되는것을 보아 정상은 1, 비정상은 -1로 구성되어 있는 것으로 유추된다.
- 위 시각화를 확인해보니, X_train의 전체적인 값들이 0을 중심으로 Z-score normalization되어있는 것을 확인할 수 있다. Standard Scaling을 따로 진행할 필요가 없어 보인다.

In [103…

```
def split_sequences(sequences, n_steps) :
        X, y = list(), list()
        for i in range(len(sequences)):
    # find the end of this pattern
                end_ix = i + n_steps
                # check if we are beyond the dataset
                if end_ix > len(sequences):
                        break
                # gather input and output parts of the pattern
                seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
                X.append(seq_x)
                y.append(seq_y)
        return np.array(X), np.array(y)
```



위 함수는 **sliding window 방식으로 구성** 되었다. sliding window란 투 포인터 풀이 기법에서 많이 쓰이는 테크닉으로, 마치 위 figure의 방식과 같이 좌우로 미끌어지면서 원소를 택하는 방법이다. 여기서 **window는 위 figure에서 원소 4개를 감싸는 박스** 를 의미하며, window의 사이즈는 4개가 된다.

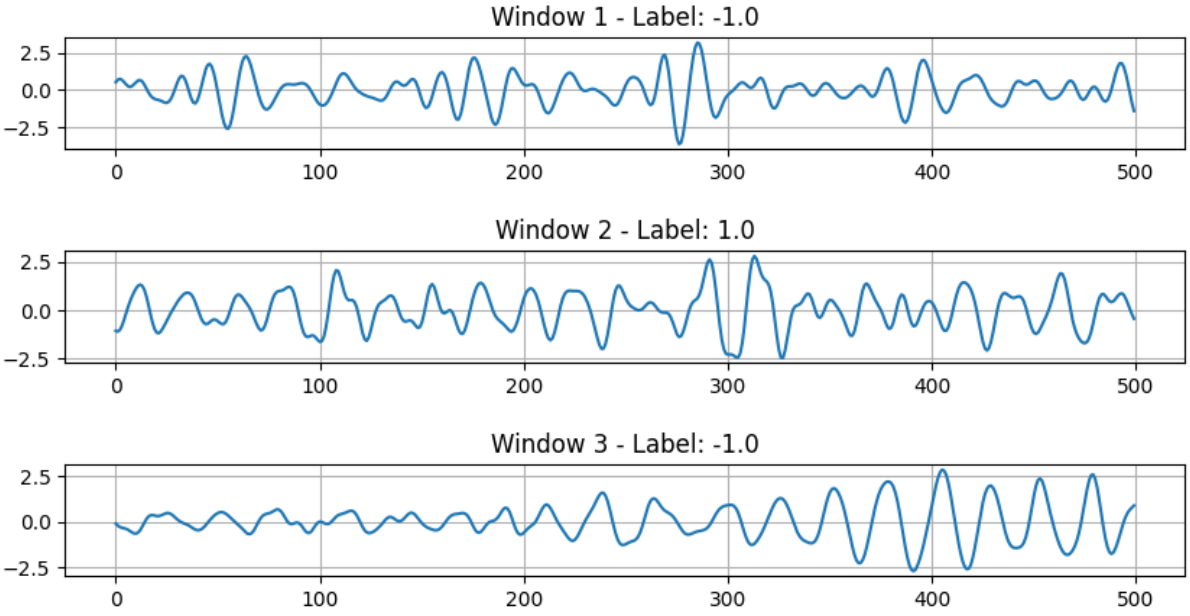
위 함수에서 **len(sequences)는 df의 records의 수 (행의 수)를 의미** 한다. 그니깐 행의 수만큼 반복문을 돌리겠다는 거고, 여기서 i는 행의 위치가 된다. end_ix는 현재 행의 위치와 n_steps를 더한 값으로 구성되고, 여기서 n_steps가 바로 window의 수이다. 1차원적으로 생각해보면, i는 위 figure에서 window의 첫번째 원소를 가리키는 index를 의미하고, n_steps를 i와 더하여 window의 마지막 원소를 가리키는 index를 end_ix라고 선언해 둔 것이다.

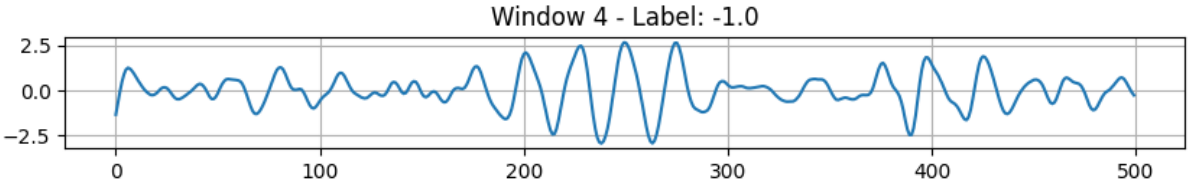
end_ix가 전체 데이터프레임의 행의 갯수를 넘어가면 함수가 종료되며, 그 전까지 함수를 진행하는데, seq_x와 seq_y에 각각 순환 데이터를 구성한 input, target을 집어 넣는다. sequences[i+end_ix, :-1]은 i열부터 end_ix - 1행까지, 가장 마지막 열인 target 열을 제외하고 split한 input 데이터를 의미하며, sequences[end_ix-1, -1]은 end_ix - 1행의 마지막 열인 target 열의 값을 target 데이터로 지정해둔 것이다. 여기서 알 수 있는 것은, **기존 split_sequences 함수는 각 window의 마지막 target 값을 순환 데이터의 target값으로 만든다는 사실** 이다. 위 경우에서 **'특정 window에서 마지막 레코드의 target 값을 선택'** 하는 경우이다.

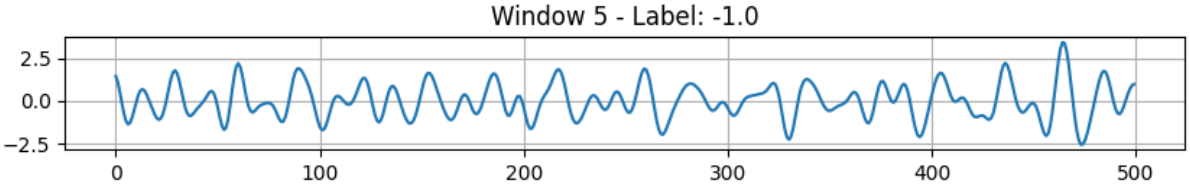
In [104…
```
# split_seqeunce를 적용한다.

def apply_sliding_window(X, y, n_steps): # STRIDE = 100
    X_all, y_all = [], []
    for sample, label in zip(X, y):
        combined = np.hstack([sample[:, np.newaxis], np.full((len(sample), 1), label)])
        x_seq, y_seq = split_sequences(combined, n_steps)
        X_all.append(x_seq)
        y_all.append(y_seq)
    return np.vstack(X_all), np.hstack(y_all)

n_steps = 500
X_train_seq, y_train_seq = apply_sliding_window(X_train, y_train, n_steps)
X_test_seq, y_test_seq = apply_sliding_window(X_test, y_test, n_steps)

print (X_train_seq.shape)
print (X_test_seq.shape)
```

```
(3601, 500, 1)
(1320, 500, 1)
```

In [105…
```
# 1차원 시계열 데이터를 시각화한다. (5개의 point)

def plot_windows(X_seq, y_seq, num_samples=5):
    for i in range(num_samples):
        plt.figure(figsize=(10, 1))
        plt.plot(X_seq[i+10, :, 0])
        plt.title(f"Window {i+1} - Label: {y_seq[i]}")
        plt.grid()
        plt.show()

plot_windows(X_train_seq, y_train_seq)
```

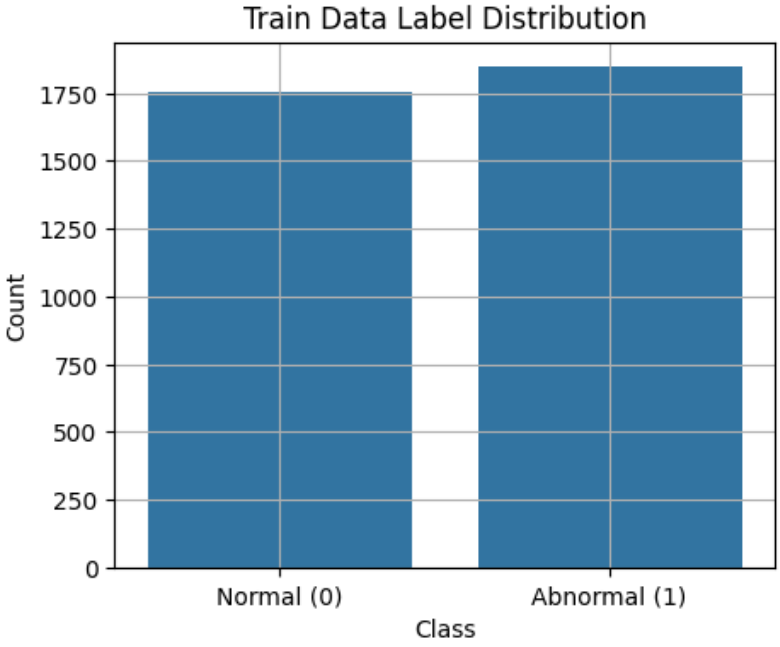






In [106…
```
# 라벨 변환: 1 → 0 (정상), -1 → 1 (이상)

y_train_seq = (y_train_seq == -1).astype(int)
y_test_seq = (y_test_seq == -1).astype(int)
```

In [107…
```
def plot_label_distribution(y_seq, title="Train Data Label Distribution"):
    plt.figure(figsize=(5, 4))
    sns.countplot(x=y_seq)
    plt.xticks([0, 1], ["Normal (0)", "Abnormal (1)"])
    plt.title(title)
    plt.xlabel("Class")
    plt.ylabel("Count")
    plt.grid(True)
    plt.show()

plot_label_distribution(y_train_seq)
```



위를 확인해보면, Outcome 열의 0과 1의 도수가 비슷하여 bias가 일어나지는 않을 것 같다. 만약에 불균형하다면, 추후 분류 시 편향이 발생할 가능성이 높다. 따라서 아래와 같은 방안을 생각해볼 수 있다.

- 오버 샘플링 (oversampling)
  - 소수 클래스인 1의 데이터 갯수를 늘려 불균형을 해소한다.
  - 데이터 손실이 없는 것이 장점이지만, 학습 시간이 증가할 수 있다.
    - 현재 데이터 수가 그리 많지 않기 때문에 괜찮을 듯 하다.
- 언더 샘플링 (undersampling)
  - 다수 클래스인 0의 데이터 갯수를 줄여 불균형을 해소한다.
  - 오버 샘플링과 반대로 작용한다.

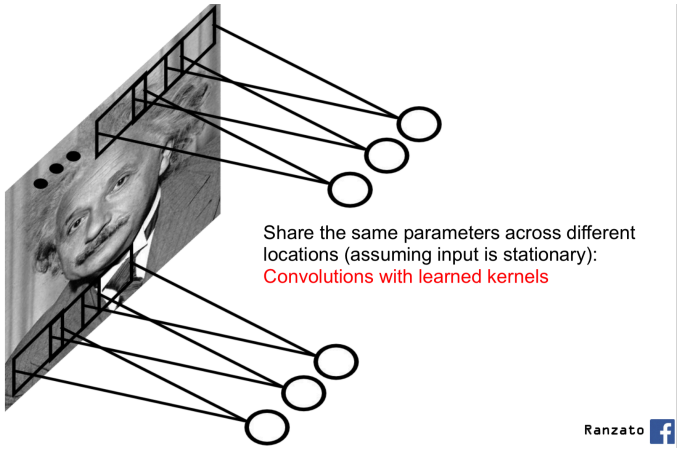
만약에 train, test가 나누어져 있지 않은 데이터셋이라면, 원래 같았으면 공평한 학습을 위해선, train_test_split 함수에서 stratify 옵션을 사용하는 것이 핵심이 되겠다. 이 옵션은 학습 데이터셋과 테스트 데이터셋 내에서 목표 변수의 분포를 동일하게 유지하는데, 만약 y 변수가 0과 1로 이루어진 이진 범주형 변수이고, 0의 비율이 25%, 1의 비율이 75%라면, stratify=y를 사용하면 랜덤 분할을 해도 0이 25%, 1이 75%인 비율을 유지할 수 있기 때문에 현재 target 분포에 꼭 필요한 옵션일 것이다.

In [108…

```
# One-hot encoding 수행

from tensorflow.keras.utils import to_categorical

y_train_cat = to_categorical(y_train_seq, num_classes=2)
y_test_cat = to_categorical(y_test_seq, num_classes=2)
```

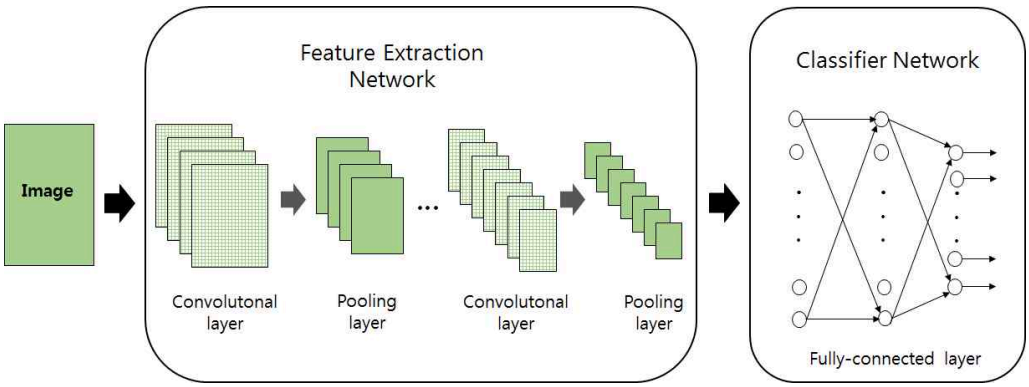
---

3. ConvNet 기반의 classification 진행 및 성능 지표 출력, 시각화



합성곱 신경망(CNN, Convolutional Neural Network)은 원래 이미지 처리 분야에서 뛰어난 성능을 보이며 널리 사용되어 온 딥러닝 모델이다. **CNN의 핵심은 필터(또는 커널)를 사용하여 입력 데이터의 지역적인 특징을 자동으로 추출하는 것이며, 이 특징 추출 과정에서 파라미터 수를 크게 줄일 수 있다는 장점**이 있다.합성곱 신경망(CNN, Convolutional Neural Network)은 원래 이미지 처리 분야에서 뛰어난 성능을 보이며 널리 사용되어 온 딥러닝 모델이다. CNN의 핵심은 필터(또는 커널)를 사용하여 입력 데이터의 지역적인 특징을 자동으로 추출하는 것이며, 이 특징 추출 과정에서 파라미터 수를 크게 줄일 수 있다는 장점이 있다.

이러한 CNN은 이미지뿐 아니라 시계열 데이터 분석에도 매우 적합한 모델이다. 그 이유는 **시계열 데이터 역시 시간 축을 따라 국소적이고 연속적인 패턴을 가지기 때문**이다. 예를 들어, 정상적인 센서 신호와 이상 징후는 특정 시점이나 구간에서 패턴 차이를 보이며, CNN은 이러한 지역적 특성을 잘 포착할 수 있다. **시계열 데이터에 CNN을 적용할 때는 Conv2D 대신 Conv1D를 사용하였다.** Conv1D는 시간 축을 따라 1차원 필터를 움직이며 특징을 추출하는데, 이는 시계열 데이터가 기본적으로 (시간, 값) 형태의 구조를 가지기 때문에 적합하다. 이렇게 하면 시간 순서를 유지하면서도, 작은 구간(윈도우) 내의 변화나 패턴을 효과적으로 포착할 수 있다.



Convolutional Neural Network, 줄여서 ConvNet은 입력 데이터에서 유용한 정보를 추출하고, 그 정보를 기반으로 예측이나 분류를 수행하는 딥러닝 모델 구조이다. 이 구조는 크게 두 가지 주요 구성 요소로 나뉘는데, 하나는 Feature

Extractor이고, 다른 하나는 Classifier이다.

Feature Extractor는 입력된 데이터를 받아 그 안에 포함된 의미 있는 특징들을 자동으로 추출하는 역할을 한다. 전통적인 기계학습에서는 사람이 직접 중요한 특징을 정의하거나 설계해야 했지만, ConvNet의 Feature Extractor는 데이터로부터 학습을 통해 스스로 특징을 찾아낸다. 이 부분은 주로 Convolution 계층과 활성화 함수, 그리고 Pooling 계층으로 구성된다. 예를 들어 시계열 데이터에 Conv1D 계층을 사용하면, 시간의 흐름에 따라 나타나는 패턴들을 작은 필터(커널)를 통해 탐지하게 된다. 이 필터는 데이터 위를 일정한 간격으로 슬라이딩하면서 국소적인 패턴을 인식하고, 이를 활성화 함수(ReLU)를 통해 비선형적으로 표현한다. 그 뒤에 이어지는 Pooling 계층은 공간적 또는 시간적 크기를 줄여주면서, 가장 중요한 정보만을 남기고 노이즈를 제거하는 효과를 가진다. 이렇게 Feature Extractor는 복잡한 원본 데이터를 간결하고 의미 있는 표현으로 변환해주는 역할을 한다.

Classifier는 Feature Extractor를 통해 추출된 특징들을 바탕으로 최종적인 예측을 수행하는 부분이다. 이 부분은 보통 Fully Connected 계층(밀집층)들로 구성되며, 입력된 특징 벡터들을 결합하고 가중치를 적용하여, 데이터가 어떤 클래스에 속하는지를 출력한다. 시계열 이진 분류 문제에서는 출력층에서 sigmoid 함수를 사용하여 0과 1 사이의 값을 예측하고, 다중 클래스 문제에서는 softmax 함수를 통해 각 클래스에 대한 확률을 예측한다. Classifier는 모델이 학습 과정에서 주어진 정답과 비교하면서 오차를 줄이는 방향으로 가중치를 조정하는데 사용된다.

In [109…

```python
# Model 정의

from tensorflow.keras import layers, models, regularizers

model = models.Sequential([
    # ConvNet 부분 (feature extractor)
    layers.Conv1D(64, kernel_size=3, activation='relu',
                  input_shape=(X_train_seq.shape[1], X_train_seq.shape[2])),
    layers.MaxPooling1D(pool_size=4),
    layers.Dropout(0.2),
    layers.Conv1D(128, kernel_size=3,
                  activation='relu'),
    layers.MaxPooling1D(pool_size=3),
    layers.Dropout(0.2),
    layers.Conv1D(256, kernel_size=3,
                  activation='relu'),
    layers.MaxPooling1D(pool_size=2),
    layers.Dropout(0.2),

    # feature 요약
    layers.GlobalAveragePooling1D(),

    # Dense (DNN) 부분
    layers.Dense(64, kernel_regularizer=regularizers.l2(0.001)),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.Dropout(0.2),

    layers.Dense(32, kernel_regularizer=regularizers.l2(0.001)),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.Dropout(0.1),

    layers.Dense(16, kernel_regularizer=regularizers.l2(0.001)),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.Dropout(0.1),

    # 출력층
    layers.Dense(2, activation='softmax')
])

model.summary()
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_8"

Layer (type)	Output Shape	Param #
conv1d_17 (Conv1D)	(None, 498, 64)	256
max_pooling1d_10 (MaxPooling1D)	(None, 124, 64)	0
dropout_16 (Dropout)	(None, 124, 64)	0
conv1d_18 (Conv1D)	(None, 122, 128)	24,704
max_pooling1d_11 (MaxPooling1D)	(None, 40, 128)	0
dropout_17 (Dropout)	(None, 40, 128)	0
conv1d_19 (Conv1D)	(None, 38, 256)	98,560
max_pooling1d_12 (MaxPooling1D)	(None, 19, 256)	0
dropout_18 (Dropout)	(None, 19, 256)	0
global_average_pooling1d_8 (GlobalAveragePooling1D)	(None, 256)	0
dense_23 (Dense)	(None, 64)	16,448
batch_normalization_13 (BatchNormalization)	(None, 64)	256
activation_13 (Activation)	(None, 64)	0
dropout_19 (Dropout)	(None, 64)	0
dense_24 (Dense)	(None, 32)	2,080
batch_normalization_14 (BatchNormalization)	(None, 32)	128
activation_14 (Activation)	(None, 32)	0
dropout_20 (Dropout)	(None, 32)	0
dense_25 (Dense)	(None, 16)	528
batch_normalization_15 (BatchNormalization)	(None, 16)	64
activation_15 (Activation)	(None, 16)	0
dropout_21 (Dropout)	(None, 16)	0
dense_26 (Dense)	(None, 2)	34

Total params: 143,058 (558.82 KB)
Trainable params: 142,834 (557.95 KB)
Non-trainable params: 224 (896.00 B)

모델의 전체 구조는 다음과 같은 이유로 구성되었다. **먼저 Conv1D 층을 세 번 쌓아, 저수준(local) 패턴에서 고수준(abstract) 패턴으로 점진적으로 특징을 추출하도록 했다. GlobalAveragePooling1D는 전체 시계열 구간의 평균적인 특징을 모아 하나의 벡터로 변환함으로써 flatten을 대체하고, 파라미터 수를 줄이는 동시에 과적합을 방지하는 효과**도 있다.

그 이후 Dense(완전 연결) 레이어에서는 **Batch Normalization과 Dropout을 함께 적용**하였다. Batch Normalization은 각 층의 입력 분포를 정규화하여 학습을 안정화하고 속도를 개선하며, Dropout은 학습 과정에서 일부 뉴런을 랜덤하게 제거함으로써 과적합을 방지한다. 또한, Dense 층에는 L2 정규화를 추가하여 가중치의 크기를 제한하고 모델의 일반화 능력을 높였다.

In [110…

```
model.compile(optimizer=tf.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

```
history = model.fit(X_train_seq, y_train_cat,
                    validation_data=(X_test_seq, y_test_cat),
                    epochs=100,
                    batch_size=64)
```

```
Epoch 1/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 7s 66ms/step - accuracy: 0.5783 - loss: 0.8327 - val_accuracy:
0.6402 - val_loss: 0.8234
Epoch 2/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.7687 - loss: 0.6183 - val_accuracy:
0.4871 - val_loss: 0.8258
Epoch 3/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 61ms/step - accuracy: 0.8406 - loss: 0.5011 - val_accuracy:
0.4841 - val_loss: 1.0170
Epoch 4/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.8641 - loss: 0.4444 - val_accuracy:
0.4864 - val_loss: 0.9605
Epoch 5/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.8714 - loss: 0.4096 - val_accuracy:
0.5333 - val_loss: 0.7964
Epoch 6/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.8901 - loss: 0.3763 - val_accuracy:
0.4932 - val_loss: 0.8894
Epoch 7/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step - accuracy: 0.8975 - loss: 0.3668 - val_accuracy:
0.8788 - val_loss: 0.4489
Epoch 8/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step - accuracy: 0.9082 - loss: 0.3336 - val_accuracy:
0.7508 - val_loss: 0.5300
Epoch 9/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9007 - loss: 0.3296 - val_accuracy:
0.9258 - val_loss: 0.3147
Epoch 10/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.8852 - loss: 0.3440 - val_accuracy:
0.8924 - val_loss: 0.3749
Epoch 11/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 56ms/step - accuracy: 0.9114 - loss: 0.2996 - val_accuracy:
0.7553 - val_loss: 0.5718
Epoch 12/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - accuracy: 0.9164 - loss: 0.2882 - val_accuracy:
0.9371 - val_loss: 0.2627
Epoch 13/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9175 - loss: 0.2781 - val_accuracy:
0.9235 - val_loss: 0.2850
Epoch 14/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 61ms/step - accuracy: 0.9224 - loss: 0.2706 - val_accuracy:
0.9152 - val_loss: 0.2683
Epoch 15/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9188 - loss: 0.2723 - val_accuracy:
0.7705 - val_loss: 0.5615
Epoch 16/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 61ms/step - accuracy: 0.9129 - loss: 0.2780 - val_accuracy:
0.9121 - val_loss: 0.2716
Epoch 17/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.9194 - loss: 0.2596 - val_accuracy:
0.8553 - val_loss: 0.4356
Epoch 18/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 57ms/step - accuracy: 0.9199 - loss: 0.2504 - val_accuracy:
0.9341 - val_loss: 0.2583
Epoch 19/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.9263 - loss: 0.2343 - val_accuracy:
0.9144 - val_loss: 0.2662
Epoch 20/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - accuracy: 0.9338 - loss: 0.2297 - val_accuracy:
0.9174 - val_loss: 0.2688
Epoch 21/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 61ms/step - accuracy: 0.9239 - loss: 0.2306 - val_accuracy:
0.9136 - val_loss: 0.2640
Epoch 22/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9174 - loss: 0.2421 - val_accuracy:
0.9341 - val_loss: 0.2277
Epoch 23/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 56ms/step - accuracy: 0.9400 - loss: 0.2030 - val_accuracy:
0.9023 - val_loss: 0.2907
Epoch 24/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 55ms/step - accuracy: 0.9275 - loss: 0.2120 - val_accuracy:
```

```
0.8742 - val_loss: 0.3609
Epoch 25/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 52ms/step - accuracy: 0.9330 - loss: 0.2134 - val_accuracy:
0.9129 - val_loss: 0.2573
Epoch 26/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 55ms/step - accuracy: 0.9212 - loss: 0.2393 - val_accuracy:
0.9114 - val_loss: 0.2627
Epoch 27/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 55ms/step - accuracy: 0.9257 - loss: 0.2240 - val_accuracy:
0.9258 - val_loss: 0.2181
Epoch 28/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 52ms/step - accuracy: 0.9183 - loss: 0.2293 - val_accuracy:
0.8788 - val_loss: 0.3542
Epoch 29/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 53ms/step - accuracy: 0.9322 - loss: 0.2016 - val_accuracy:
0.8667 - val_loss: 0.3107
Epoch 30/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step - accuracy: 0.9345 - loss: 0.1995 - val_accuracy:
0.9068 - val_loss: 0.2468
Epoch 31/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 54ms/step - accuracy: 0.9282 - loss: 0.2136 - val_accuracy:
0.9311 - val_loss: 0.2097
Epoch 32/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9318 - loss: 0.1887 - val_accuracy:
0.7326 - val_loss: 0.6831
Epoch 33/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 57ms/step - accuracy: 0.9347 - loss: 0.1931 - val_accuracy:
0.9371 - val_loss: 0.1967
Epoch 34/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - accuracy: 0.9308 - loss: 0.2105 - val_accuracy:
0.8129 - val_loss: 0.5135
Epoch 35/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9377 - loss: 0.1912 - val_accuracy:
0.9250 - val_loss: 0.2038
Epoch 36/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9241 - loss: 0.2136 - val_accuracy:
0.9379 - val_loss: 0.1852
Epoch 37/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9377 - loss: 0.1899 - val_accuracy:
0.9106 - val_loss: 0.2457
Epoch 38/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9365 - loss: 0.1890 - val_accuracy:
0.9303 - val_loss: 0.1978
Epoch 39/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - accuracy: 0.9329 - loss: 0.1838 - val_accuracy:
0.9189 - val_loss: 0.2711
Epoch 40/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9358 - loss: 0.1784 - val_accuracy:
0.9348 - val_loss: 0.2115
Epoch 41/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9367 - loss: 0.1813 - val_accuracy:
0.9091 - val_loss: 0.2351
Epoch 42/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9260 - loss: 0.1896 - val_accuracy:
0.9394 - val_loss: 0.1780
Epoch 43/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9337 - loss: 0.1905 - val_accuracy:
0.9265 - val_loss: 0.1972
Epoch 44/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9359 - loss: 0.1759 - val_accuracy:
0.9220 - val_loss: 0.2333
Epoch 45/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.9307 - loss: 0.1822 - val_accuracy:
0.9068 - val_loss: 0.2530
Epoch 46/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9404 - loss: 0.1776 - val_accuracy:
0.9439 - val_loss: 0.1882
Epoch 47/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9311 - loss: 0.1802 - val_accuracy:
0.9023 - val_loss: 0.2789
Epoch 48/100
```

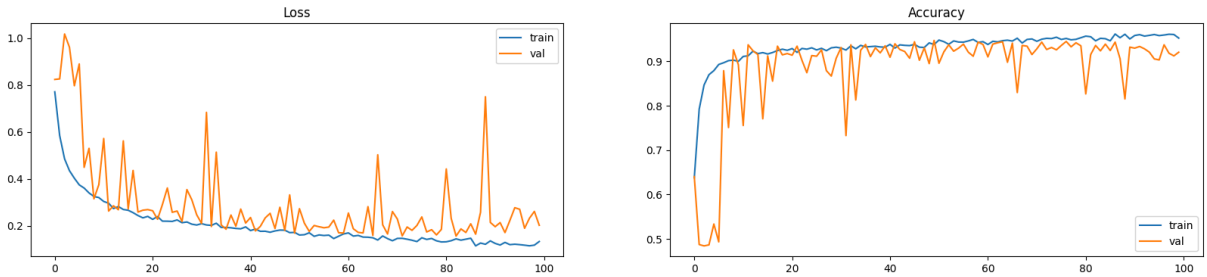
```
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.9293 - loss: 0.1761 - val_accuracy:
0.9303 - val_loss: 0.1799
Epoch 49/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9410 - loss: 0.1765 - val_accuracy:
0.8947 - val_loss: 0.3315
Epoch 50/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 70ms/step - accuracy: 0.9376 - loss: 0.1679 - val_accuracy:
0.9470 - val_loss: 0.1687
Epoch 51/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9461 - loss: 0.1620 - val_accuracy:
0.8955 - val_loss: 0.2727
Epoch 52/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9520 - loss: 0.1467 - val_accuracy:
0.9212 - val_loss: 0.2111
Epoch 53/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9422 - loss: 0.1669 - val_accuracy:
0.9371 - val_loss: 0.1759
Epoch 54/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9481 - loss: 0.1552 - val_accuracy:
0.9227 - val_loss: 0.2010
Epoch 55/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step - accuracy: 0.9484 - loss: 0.1499 - val_accuracy:
0.9295 - val_loss: 0.1959
Epoch 56/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9455 - loss: 0.1505 - val_accuracy:
0.9386 - val_loss: 0.1914
Epoch 57/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9486 - loss: 0.1532 - val_accuracy:
0.9205 - val_loss: 0.1944
Epoch 58/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 61ms/step - accuracy: 0.9469 - loss: 0.1487 - val_accuracy:
0.9114 - val_loss: 0.2240
Epoch 59/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9382 - loss: 0.1638 - val_accuracy:
0.9432 - val_loss: 0.1702
Epoch 60/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9443 - loss: 0.1634 - val_accuracy:
0.9379 - val_loss: 0.1690
Epoch 61/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9318 - loss: 0.1750 - val_accuracy:
0.9098 - val_loss: 0.2542
Epoch 62/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9431 - loss: 0.1559 - val_accuracy:
0.9386 - val_loss: 0.1873
Epoch 63/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9429 - loss: 0.1515 - val_accuracy:
0.9417 - val_loss: 0.1717
Epoch 64/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 66ms/step - accuracy: 0.9509 - loss: 0.1383 - val_accuracy:
0.9439 - val_loss: 0.1694
Epoch 65/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9484 - loss: 0.1401 - val_accuracy:
0.8977 - val_loss: 0.2815
Epoch 66/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9433 - loss: 0.1591 - val_accuracy:
0.9409 - val_loss: 0.1587
Epoch 67/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.9536 - loss: 0.1293 - val_accuracy:
0.8295 - val_loss: 0.5023
Epoch 68/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9408 - loss: 0.1596 - val_accuracy:
0.9356 - val_loss: 0.2043
Epoch 69/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 59ms/step - accuracy: 0.9476 - loss: 0.1487 - val_accuracy:
0.9341 - val_loss: 0.1648
Epoch 70/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9537 - loss: 0.1304 - val_accuracy:
0.9152 - val_loss: 0.2605
Epoch 71/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9450 - loss: 0.1451 - val_accuracy:
0.9280 - val_loss: 0.2287
```

```
Epoch 72/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 53ms/step - accuracy: 0.9407 - loss: 0.1645 - val_accuracy:
0.9432 - val_loss: 0.1568
Epoch 73/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9542 - loss: 0.1359 - val_accuracy:
0.9265 - val_loss: 0.1947
Epoch 74/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9466 - loss: 0.1433 - val_accuracy:
0.9311 - val_loss: 0.1804
Epoch 75/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9593 - loss: 0.1215 - val_accuracy:
0.9258 - val_loss: 0.2012
Epoch 76/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9513 - loss: 0.1461 - val_accuracy:
0.9356 - val_loss: 0.2376
Epoch 77/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9513 - loss: 0.1366 - val_accuracy:
0.9447 - val_loss: 0.1734
Epoch 78/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9460 - loss: 0.1543 - val_accuracy:
0.9326 - val_loss: 0.1832
Epoch 79/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 66ms/step - accuracy: 0.9435 - loss: 0.1502 - val_accuracy:
0.9417 - val_loss: 0.1609
Epoch 80/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9593 - loss: 0.1131 - val_accuracy:
0.9348 - val_loss: 0.1845
Epoch 81/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9579 - loss: 0.1246 - val_accuracy:
0.8265 - val_loss: 0.4419
Epoch 82/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 60ms/step - accuracy: 0.9547 - loss: 0.1310 - val_accuracy:
0.9152 - val_loss: 0.2319
Epoch 83/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step - accuracy: 0.9441 - loss: 0.1463 - val_accuracy:
0.9356 - val_loss: 0.1560
Epoch 84/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 55ms/step - accuracy: 0.9526 - loss: 0.1321 - val_accuracy:
0.9235 - val_loss: 0.1866
Epoch 85/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9564 - loss: 0.1345 - val_accuracy:
0.9386 - val_loss: 0.1712
Epoch 86/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9532 - loss: 0.1353 - val_accuracy:
0.9242 - val_loss: 0.2083
Epoch 87/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 68ms/step - accuracy: 0.9594 - loss: 0.1241 - val_accuracy:
0.9432 - val_loss: 0.1642
Epoch 88/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9491 - loss: 0.1366 - val_accuracy:
0.9061 - val_loss: 0.2574
Epoch 89/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9630 - loss: 0.1196 - val_accuracy:
0.8152 - val_loss: 0.7497
Epoch 90/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 66ms/step - accuracy: 0.9537 - loss: 0.1266 - val_accuracy:
0.9318 - val_loss: 0.2142
Epoch 91/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9565 - loss: 0.1218 - val_accuracy:
0.9295 - val_loss: 0.1959
Epoch 92/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9606 - loss: 0.1182 - val_accuracy:
0.9333 - val_loss: 0.2136
Epoch 93/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9578 - loss: 0.1285 - val_accuracy:
0.9280 - val_loss: 0.1709
Epoch 94/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.9585 - loss: 0.1200 - val_accuracy:
0.9205 - val_loss: 0.2227
Epoch 95/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9614 - loss: 0.1145 - val_accuracy:
```

```
0.9053 - val_loss: 0.2769
Epoch 96/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - accuracy: 0.9565 - loss: 0.1199 - val_accuracy:
0.9030 - val_loss: 0.2702
Epoch 97/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 3s 58ms/step - accuracy: 0.9534 - loss: 0.1252 - val_accuracy:
0.9371 - val_loss: 0.1892
Epoch 98/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 66ms/step - accuracy: 0.9642 - loss: 0.1056 - val_accuracy:
0.9182 - val_loss: 0.2321
Epoch 99/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9592 - loss: 0.1211 - val_accuracy:
0.9121 - val_loss: 0.2611
Epoch 100/100
57/57 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9579 - loss: 0.1252 - val_accuracy:
0.9205 - val_loss: 0.2019
```

In [111…

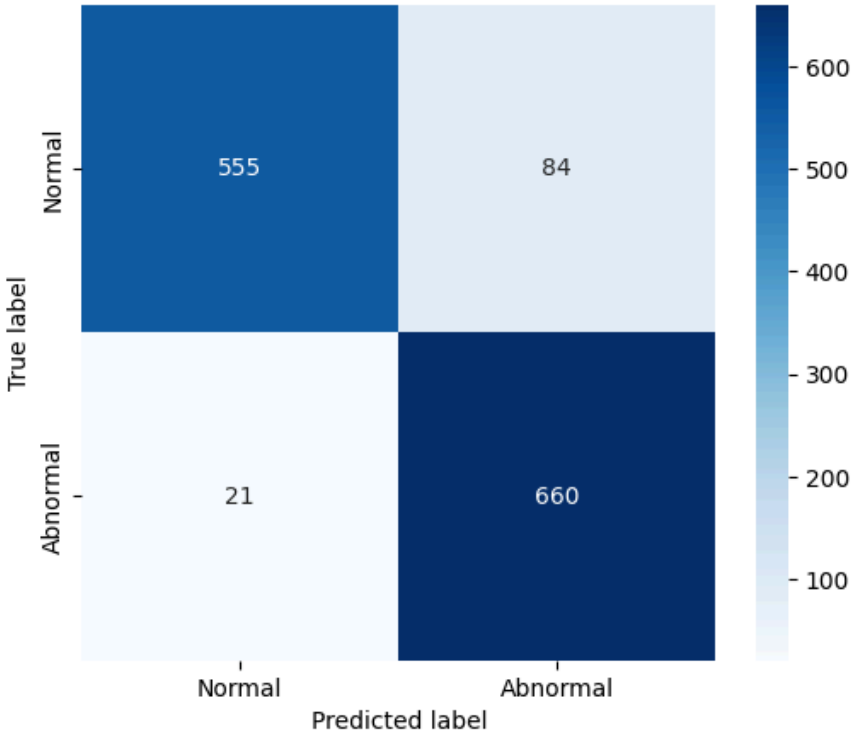
```python
plt.figure(figsize=(20, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.title("Loss")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='val')
plt.title("Accuracy")
plt.legend()

plt.show()
```



In [112…

```python
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, prec

# 예측
y_pred = model.predict(X_test_seq)
y_pred_labels = np.argmax(y_pred, axis=1)
y_true_labels = np.argmax(y_test_cat, axis=1)

# 지표 계산
print ("Accuracy:", accuracy_score(y_true_labels, y_pred_labels))
print ("Precision:", precision_score(y_true_labels, y_pred_labels))
print ("Recall:", recall_score(y_true_labels, y_pred_labels))
print ("F1 Score:", f1_score(y_true_labels, y_pred_labels))
print ("\nClassification Report:\n", classification_report(y_true_labels, y_pred_labels))

# Confusion Matrix
cm = confusion_matrix(y_true_labels, y_pred_labels)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Abnormal'], yt
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()
```

```
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step
Accuracy: 0.9204545454545454
Precision: 0.8870967741935484
Recall: 0.9691629955947136
F1 Score: 0.9263157894736842

Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.87      0.91       639
           1       0.89      0.97      0.93       681

    accuracy                           0.92      1320
   macro avg       0.93      0.92      0.92      1320
weighted avg       0.92      0.92      0.92      1320
```



- True Positive(TP), True Negative(TN), False Positive(FP), False Negative(FN)

True Positive는 실제로 정답이 긍정 클래스(예: 질병 있음)이며 모델도 이를 긍정으로 올바르게 예측한 경우를 의미한다. True Negative는 실제로 부정 클래스(예: 질병 없음)이며 모델도 이를 부정으로 정확히 예측한 경우이다. 반면 False Positive는 실제로는 부정인데 모델이 잘못 긍정으로 예측한 경우이며, 이를 **"Type I 오류"** 라고도 부른다. False Negative는 실제로는 긍정인데 모델이 이를 부정으로 잘못 예측한 경우로, **"Type II 오류"** 라고 한다.

위 confusion matrix를 기반으로 본 모델의 일반화 능력 (generalization error)를 판단하면 아래와 같다.

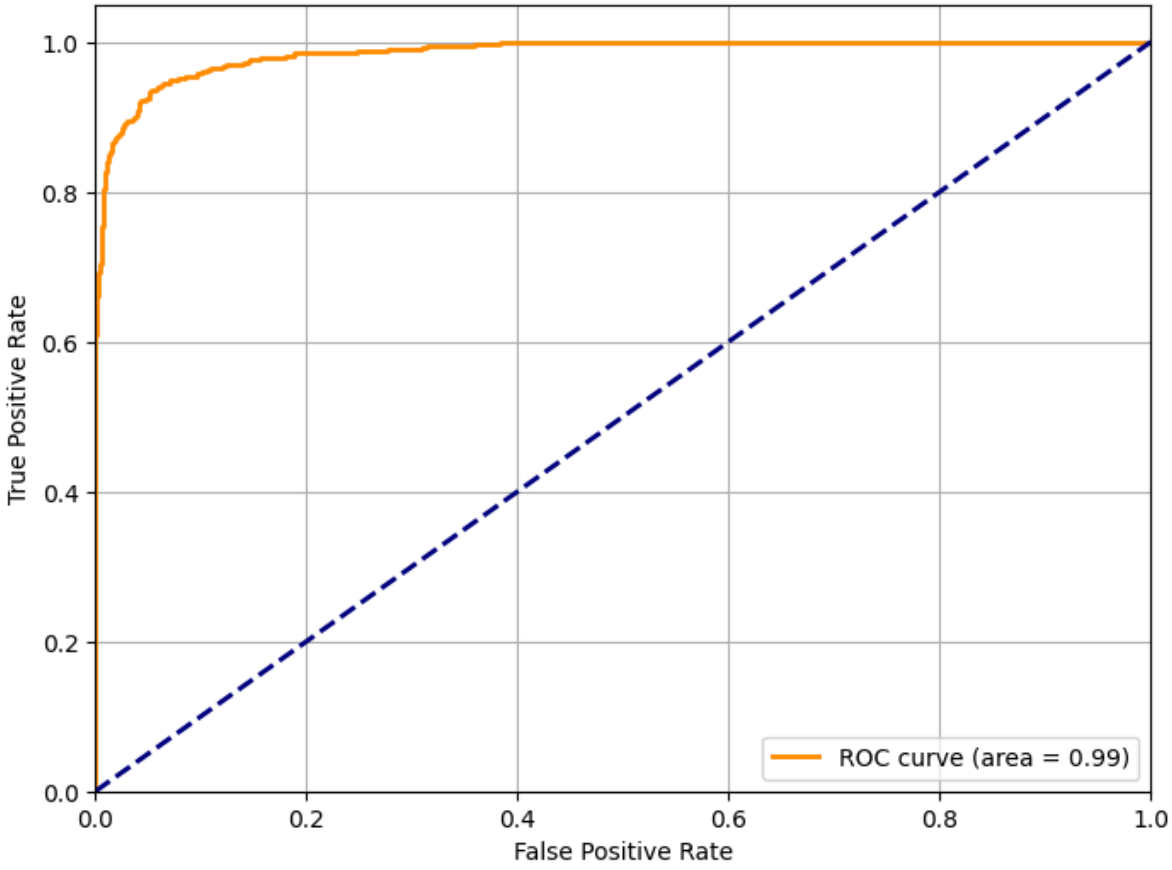
- 전체 정확도는 92.05%로 우수한 편이며, 모델이 대부분의 데이터를 올바르게 분류하고 있음을 보여준다.
- 이상을 놓치지 않는 능력인 **재현율(Recall)** 이 **96.91%** 로 매우 높아, 실제로 이상인 경우 거의 대부분을 정확히 탐지한다. 이는 의료, 보안, 이상 징후 감지 등에서 매우 중요한 요소이다.
- **정밀도(Precision)가 88.71%** 로 높지만 완벽한 수치는 아니다. 이는 Abnormal로 판단한 예측 중 일부가 실제로는 정상이라는 뜻이며, 정상 데이터를 이상으로 과잉 판단하는 경우가 존재함을 의미한다.

In [113…

```python
from sklearn.metrics import roc_curve, auc

y_pred_proba = model.predict(X_test_seq)[:, 1]

fpr, tpr, thresholds = roc_curve(y_true_labels, y_pred_proba)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
```

```
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.grid()
plt.show()
```

```
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step
```



- X축: FPR (False Positive Rate) = FP / (FP + TN)
- Y축: TPR (True Positive Rate) = TP / (TP + FN) = Recall

위 시각화는 ROC 곡선이다. ROC 곡선의 좌상단 모서리에 가까울수록 좋은 모델이다. 이 영역은 FPR은 낮고 TPR은 높은 상태를 의미한다. 대각선(y = x) 근처의 ROC 곡선은 **무작위 추측 수준(random guessing)** 을 의미하며, AUC 값이 0.5에 가까워진다. 이상적인 모델은 FPR이 거의 0이면서 TPR이 거의 1인 지점, 즉 왼쪽 위 꼭짓점에 ROC 곡선이 몰려 있는 형태이다.

현재 이 모델의 ROC는 왼쪽 위 꼭짓점에 ROC 곡선이 몰려 있음으로 test score가 잘 나오고 있음을 확인 가능하다.

---

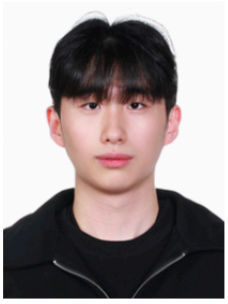
4. 결론

본 문서에서 나는 ConvNet을 이용하여 FordA dataset을 기반으로 차량 엔진 이상 케이스를 시계열 데이터 처리 방식과 함께 성능을 제시하였다. 다만, 시계열/시그널 AI를 전공분야로 삼아 연구하고 있는 학부연구생 입장에서 본 연습문제에 대한 아쉬움을 제시하고 싶다.

1. 사실 이런 이상 케이스를 탐지하는 건 anomaly detection이라고 하는 문제 해결 policy를 사용하는 것이 맞다. AutoEncoder 구조의 비지도 학습 모델을 이용해서 reconstruction error의 차이를 통해 이상 탐지를 수행하는 것이 지도학습보다 본 연습 문제에 더욱 적합한 방법이다.

2. 이미 본 데이터셋에서는 Z-score normalization이 되어 있으나, 보통 signal에서는 MinMax Scaling을 사용한다. 왜냐하면 Z-score 자체가 0-based이라 특정 개형에 대한 특이성을 반영하지 못하고 모두 기준이 0으로 맞추어지기 때문이다.

따라서 나는 모든 중간고사 대체 과제를 이행 후에 AutoEncoder 기반 anomaly detection을 이 데이터셋을 기반으로 수행해볼 예정이며, 실제 원본 FordA 데이터셋과 더불어 FordB 데이터셋 또한 구하여 더욱 더 발전된 모델 (U-Net, TF-Transformer...)등을 도입하여 실험하겠다.



Gyuyeon Lim (Member, IEEE) is an undergraduate student in the Department of Computer Science and Engineering at Gachon University, South Korea. He is currently pursuing a Bachelor of Science degree with a focus on artificial intelligence, AI for signal data, self-supervised learning, and human activity recognition.