10장 시계열

개정 2판

- 온도 예측 문제
  - 데이터

코드 10-1 예나 날씨 데이터셋 조사하기

```
import os
fname = os.path.join("jena_climate_2009_2016.csv")

with open(fname) as f:
    data = f.read()

lines = data.split("\n")
header = lines[0].split(",")
lines = lines[1:]
print(header)
print(len(lines))
```

지정된 파일(fname)을 열고 파일 핸들을 f로 할당
with 구문을 사용하면 파일을 올바르게 닫을 수 있음
data 문자열로 반환

data 문자열을 줄 바꿈 문자("\n")를 기준으로 분할하여 리스트로 변환
첫 번째 줄을 쉼표(",")를 기준으로 분할하여 헤더 추출 (각 열의 이름)
첫 번째 줄을 제외한 나머지 줄을 데이터로 사용

```
['"Date Time"', '"p (mbar)"', '"T (degC)"', '"Tpot (K)"', '"Tdew (degC)"', '"rh (%)"', '"VPmax (mbar)"', '"VPact (mbar)"',
'"VPdef (mbar)"', '"sh (g/kg)"', '"H2OC (mmol/mol)"', '"rho (g/m**3)"', '"wv (m/s)"', '"max. wv (m/s)"', '"wd (deg)"']

420451
```

- 420,451개의 데이터 전체를 넘파이 배열로 바꿈
- 온도(섭씨)를 하나의 배열로 만들고 나머지 데이터를 또 다른 배열로 만듦
- 두 번째 배열이 미래 온도를 예측하기 위해 사용할 특성
- 'Date Time' 열은 제외시킴

코드 10-2 데이터 파싱

```
import numpy as np

temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i, :] = values[:]
```

(420451,) 온도를 나타내는 numpy 배열 생성
(420451, 15-1) numpy 배열 생성
두 번째 열을 'temperature' 배열에 저장합니다. (420451,)
(온도를 포함하여) 모든 열을 'raw_data' 배열에 저장합니다. (420451, 14)

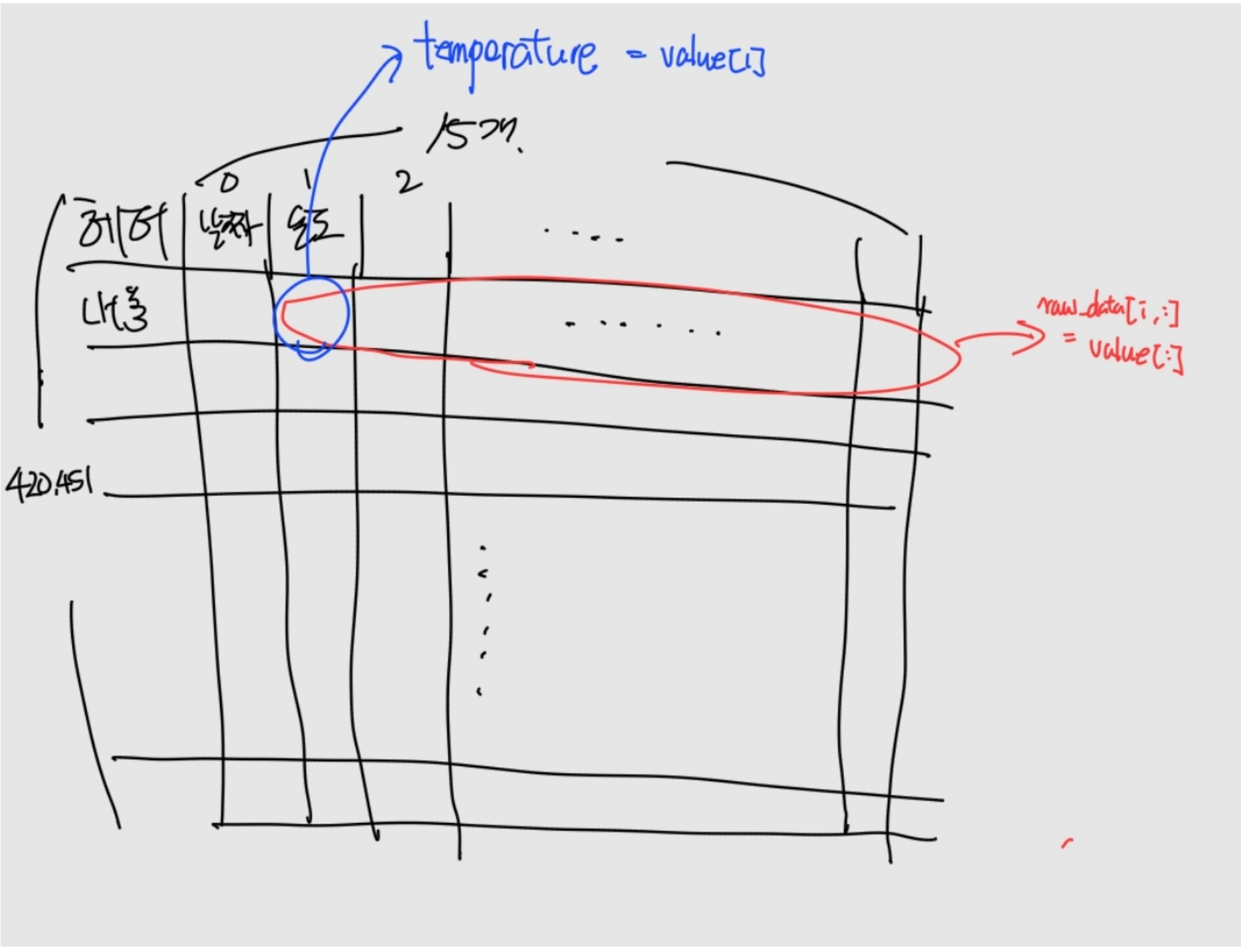
temperature라는 온도를 나타내는 numpy 배열생성 (420451,)

raw__data는 헤더를 제외한 모든 행 데이터를 가질 numpy 배열 (420451, 14) * DateTime 열 제외

values는 하나의 행 데이터를 ,로 토큰화해서 가져온 것. 여기서 1번 째 위치가 temperature이므로 temperature[i]에 저장한다.

raw_data[i, :] 에는 하나의 행 데이터 전체를 저장한다. (온도 포함) → = values[:]

참고 그림

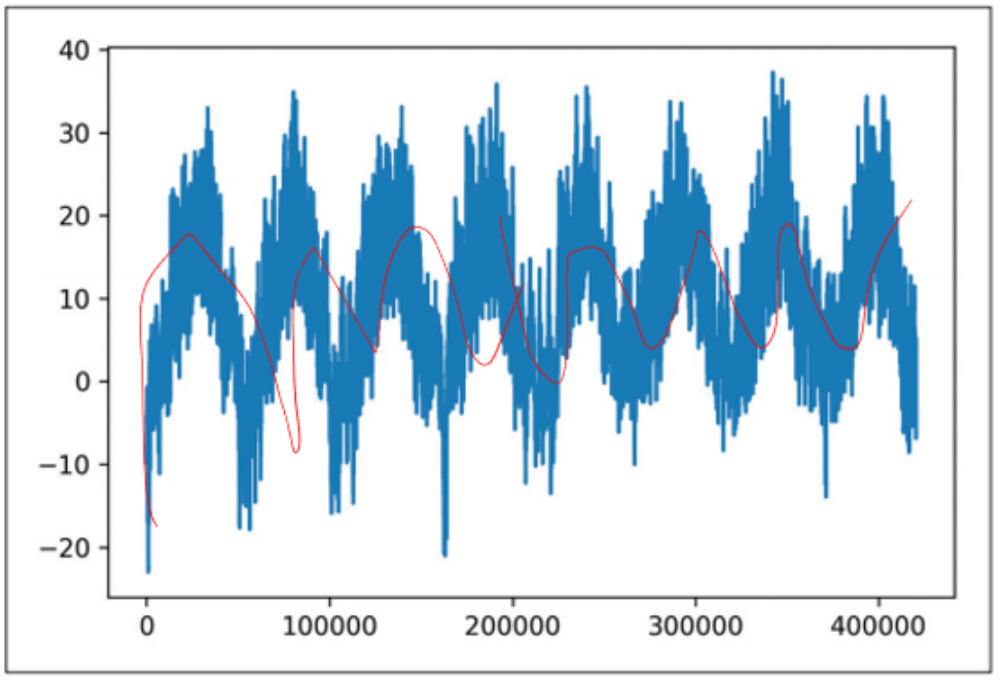


- 그림 10-1은 시간에 따른 온도(섭씨) 그래프
- 이 그래프에서 매년 온도에 주기성이 있다는 것을 잘 볼 수 있음(이 데이터의 범위는 8년)

코드 10-3 전체 온도를 그래프로 그리기

```
from matplotlib import pyplot as plt

plt.plot(range(len(temperature)), temperature)
plt.show()
```



- 그림 10-2는 기간을 좁혀서 처음 10일간 온도 데이터를 나타낸 그래프
- 10분마다 데이터가 기록되므로 하루에 144개의 데이터 포인트가 있음(총 1440개)

코드 10-4 처음 10일간의 온도를 그래프로 그리기

```
plt.plot(range(1440), temperature[:1440])
plt.show()
```

10분 마다 기록시 1시간은 6개 → 24시간은 하루에 144개(24 * 6) → 10일이면 1440

온도 예측 데이터 샘플 나누기

코드 10-5 각 분할에 사용할 샘플 개수 계산하기

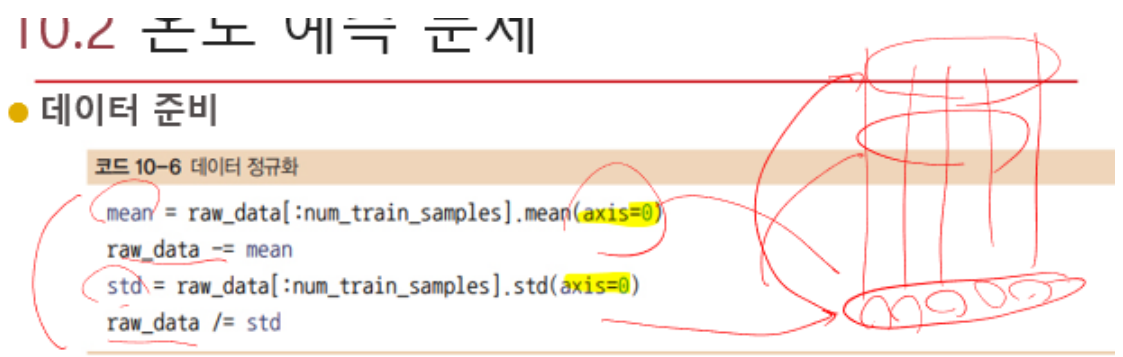
```
>>> num_train_samples = int(0.5 * len(raw_data))
>>> num_val_samples = int(0.25 * len(raw_data))
>>> num_test_samples = len(raw_data) - num_train_samples - num_val_samples
>>> print("num_train_samples:", num_train_samples)
>>> print("num_val_samples:", num_val_samples)
>>> print("num_test_samples:", num_test_samples)
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

처음 50% 데이터는 훈련 그 다음 25%를 검증 마지막(최신) 25%를 테스트에 사용한다.

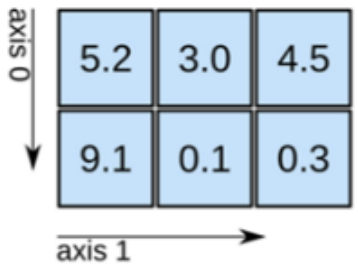
검증과 테스트가 훈련보다 최신의 데이터여야한다. → 과거를 바탕으로 미래를 예측하기 때문이다.

데이터 정규화

이 데이터에 있는 **시계열은 특성(기압, 온도, H20C 등..)마다 스케일이 다름** (기압은 1,000이고 온도는 화씨, H20C는 mmol/mol로 측정되오 약 3정도) → **따라서 각 시계열을 독립적으로 정규화 하여 비슷한 범위를 가진 작은 값으로 바꾼다. (axis = 0 기준)**

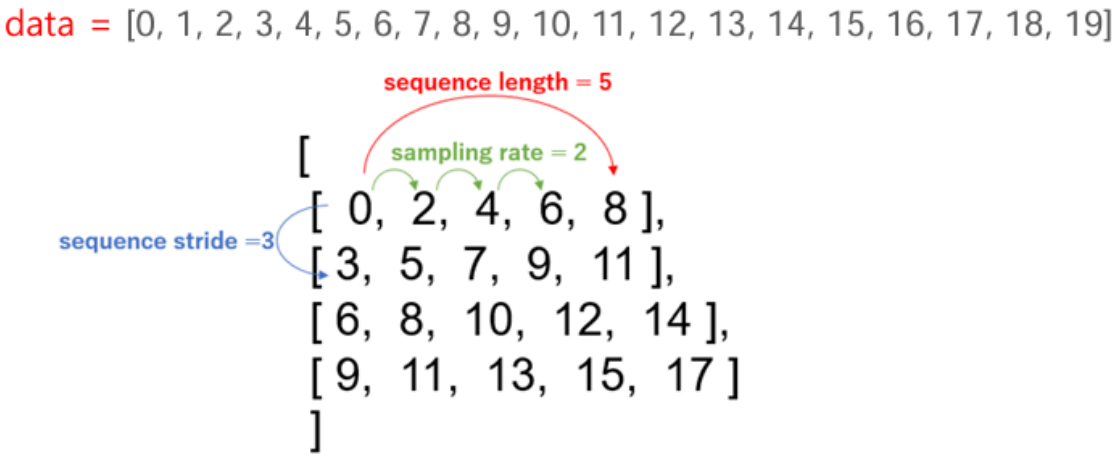


각 열을 기준으로 (axis=0) mean을 계산하고 평균을 뺀다. 그리고 표준 편차로 나눈다.



데이터 준비

- 이론




```
int_sequence = np.arange(10)            [0,1,2,3,4,5,6,7,8,9]
dummy_dataset = keras.utils.timeseries_dataset_from_array(
        data=int_sequence[:-3],         [0, 1, 2, 3, 4, 5, 6]
        targets=int_sequence[3:],       [3, 4, 5, 6, 7, 8, 9]
        sequence_length=3,              이 시퀀스의 길이는 3입니다.
        batch_size=2,                   이 시퀀스의 배치 크기는 2입니다.
)

for inputs, targets in dummy_dataset:  # inputs – 6개의 생성된 시퀀스
    for i in range(inputs.shape[0]):   #  inputs.shape[0] : (2, 3) (batch_size=2), i=0,1
        print([int(x) for x in inputs[i]], int(targets[i]))
```

첫 번째 배치: - inputs [0, 1, 2] [1, 2, 3] targets [3] [4]
두 번째 배치: - inputs [2, 3, 4] [3, 4, 5] targets [5] [6]
세 번째 배치: - inputs [4, 5, 6] targets [7]

[0,1,2,3,4,5,6,7,8,9] 에서 시퀀스 길이가 3이고 배치 크기가 2이면

첫 번째 배치 때 아래와 같이 배치 된다.

[0,1,2] target[0] (=3)
[1,2,3] target[1] (=4)

실제 데이터 준비

sampling_rate = 6 : 시간당 하나의 데이터 포인트를 샘플링 (10분 * 6 = 1시간)

sequence_length = 120 : 이전 5일간(120시간) 데이터를 사용한다.

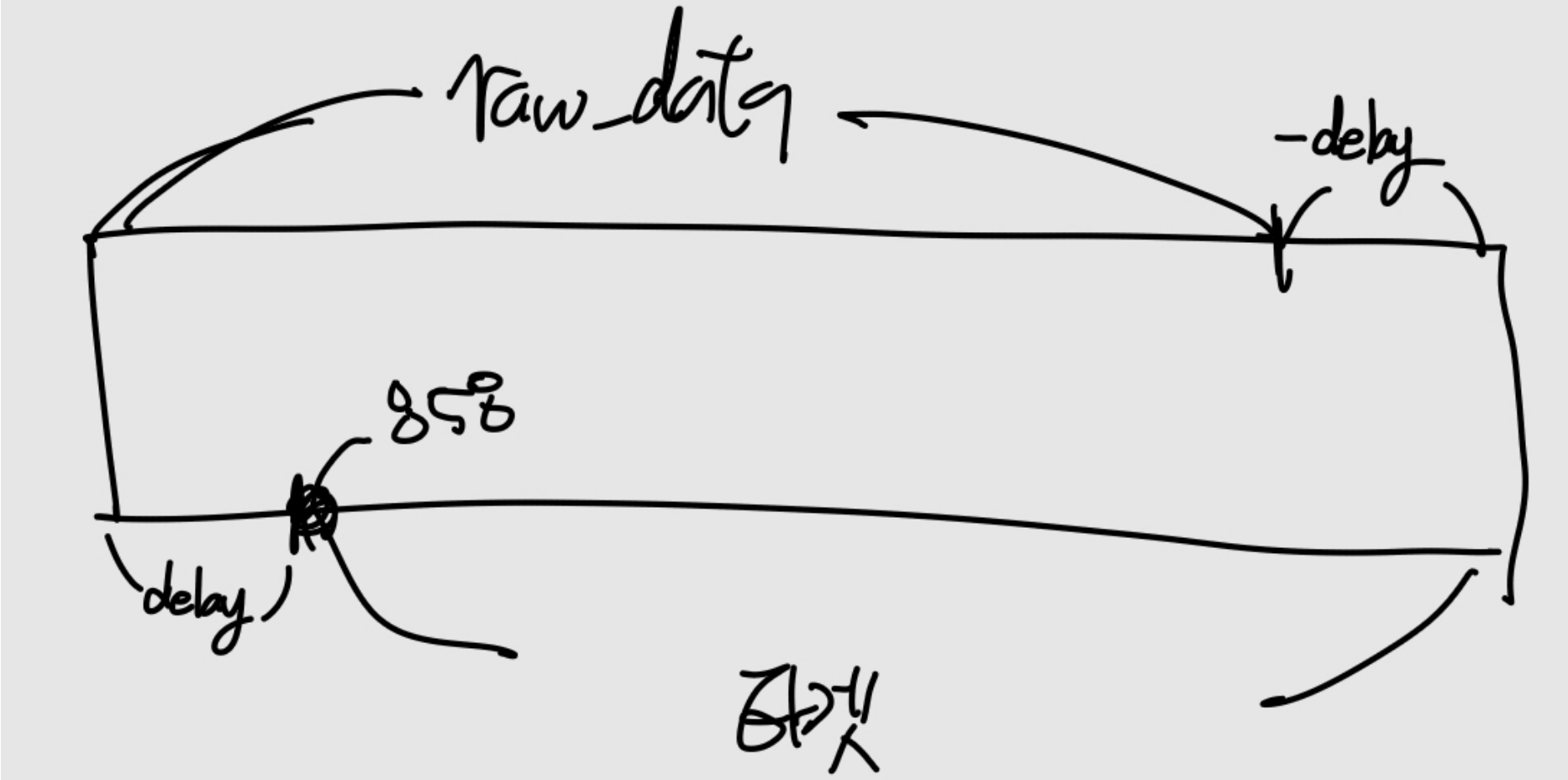
delay = sampling_rate * (sequence_length + 24 - 1): -1을 한 이유는 24시간 안의 데이터를 사용하기 위함이다. 이 값 이후는 시퀀스의 타킷에 해당한다.

코드 10-7 훈련, 검증, 테스트 데이터셋 만들기

```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256
```

delay = sampling_rate * (sequence_length + 24 - 1)

sequence_stride는 1



- 데이터 준비

```
train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],   (419593, 14)
    targets=temperature[delay:],   (419593,)
    sampling_rate=sampling_rate,   6
    sequence_length=sequence_length,   120
    shuffle=True,
    batch_size=batch_size,   256
    start_index=0,
    end_index=num_train_samples)   210225
```

- [0,6,12,...,719] 858
- [1,7,13,...,720] 859
- [2,8,14,...,721] 860

  ...

- [209506,...,210224] 211,083

shuffle

- 셔플을 하면 시계을의 데이터 특징을 고려하지 않는다? → 좋은 결과는 아닐 듯

- **데이터 준비**

```
val_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples,    210225
    end_index=num_train_samples + num_val_samples)    315337

test_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples + num_val_samples)    315337
```

sampling_rate는 1시간 당 데이터를 사용 (10분 * 6)

sequence_length는 120시간 (1시간 기준으로 자른 데이터가 120개 있으니 120시간= 1시간 * 120)

한번 배치에 256개의 데이터 셋을 가져옴 (batch_size =256)

- **데이터 준비**
  - 각 데이터셋 - (samples, targets) 크기의 튜플
  - samples는 256개의 샘플로 이루어진 배치
  - 각 샘플 - 연속된 120시간의 입력 데이터를 담고 있음
  - Targets - 256개의 타깃 온도
  - 샘플 (shuffle) - 연속된 두 샘플(예를 들어 samples[0]과 samples[1])이 꼭 시간적으로 가까운 것은 아님

코드 10-8 훈련 데이터셋의 배치 크기 확인하기

```
>>> for samples, targets in train_dataset:
>>>     print("샘플 크기:", samples.shape)
>>>     print("타깃 크기:", targets.shape)
>>>     break
샘플 크기: (256, 120, 14)
타깃 크기: (256,)
```

한 배치에 256 샘플을 가져오고 각 샘플을 연속된 120시간의 입력 데이터를 담고 있고 256개의 타겟 온도를 가지고 있다.

따라서 훈련 데이터셋의 배치 크기를 확인하면 한 번 배치에 256개를 가져오니 256이고 120시간 기준으로 14개의 특성을 가지고 있다. 따라서 (256,120,14)이다. 타깃은 (256,)

상식 수준의 기준점 - MAE 사용

모델 성능의 좋고 나쁨을 정할 기준을 알아야 하는데 이를 위해 통상적인 상식 수준의 모델을 만들어서 기준을 만든다.

상식 수준의 모델의 정확도 보다 내가 만든 모델의 정확도가 높으면 모델 성능이 좋은 것!

평균 절댓값 오차 (MAE)를 사용

- 다음 코드 10-9는 평가 루프

코드 10-9 상식 수준 모델의 MAE 계산하기

```
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples, targets in dataset:
        preds = samples[:, -1, 1] * std[1] + mean[1]
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

print(f"검증 MAE: {evaluate_naive_method(val_dataset):.2f}")
print(f"테스트 MAE: {evaluate_naive_method(test_dataset):.2f}")
```

온도 특성은 칼럼 인덱스 1에 있습니다. 따라서 samples[:, -1, 1]이 입력 시퀀스에 있는 마지막 온도 측정값입니다. 특성을 정규화했기 때문에 온도를 섭씨로 바꾸려면 표준 편차를 곱하고 평균을 더해야 합니다.[7]

samples shape: (256, 120, 14)
std[1] - temperature

Validation MAE: 2.44
Test MAE: 2.62

샘플 시퀀스의 120시간째의 온도를 가져오기 위해서 samples[:, -1, 1]을 사용한다.
sample (256, 120, 14) = (배치, 총 시간, 특성 수) 온도는 두번째 특성이라 인덱스 1

특성을 **정규화 했으므로 온도를 섭씨로 바꾸기 위해 온도의 표준 편차를 곱하고 온도의 평균을 다시 더해준다.**

total_abs_err += np.sum(np.abs(preds - targets)) **예측값과 실제 타겟 값의 절댓값 차이를 누적합하여 저장한다.**

그리고 **샘플의 배치값**인 samples_seen(samples.shape[0])**으로 나눠준다.**

기본적인 머신 러닝 모델로 시도

완전 연결 네트워크 사용, 손실 함수는 평균 제곱 오차를 사용

코드 10-10 밀집 연결 모델 훈련하고 평가하기

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"테스트 MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

120 14 (420451, 14)
1680 120*14개로 변환 – 시간적 구조가 훼손

콜백을 사용해서 최상의 모델을 저장합니다.

timeseries_dataset_from_array()에 의해 (samples, targets) 형태의 튜플을 반복(iterate)할 수 있는 데이터셋

최상의 모델을 다시 로드하고 테스트 데이터에서 평가합니다.

x = layers.Flatten()(inputs)에서 120 * 14개로 데이터를 일자로 늘려서 시간적 구조가 훼손됐다. 이는 120개의 타임 스탭과 14개의 특징 간의 관계가 고려되지 않아 성능이 향상되지 않는다.

1D 합성곱 모델 시도해보기

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
```

1680

input_9 (Inp
conv1d_16 (C
max_pooling1

밀집 연결 모델보다 성능이 더 나쁘다

시계열은 순서가 중요한데 최대 풀링과 전역 풀링 층 때문에 순서 정보가 삭제된다.

순환 신경망 사용하기

우리가 기존에 배운 MAE, MSE, 합성곱 모델을 사용해도 상식수준의 모델보다 성능이 좋지 않다.

따라서 새로운 모델인 순환 신경망(RNN, LSTM, GRU)를 배운다.

코드 10-12 간단한 LSTM 기반 모델

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
```

과거 정보를 사용하여 새롭게 얻은 정보를 업데이트 한다. (잔차 연결과 비슷하다)

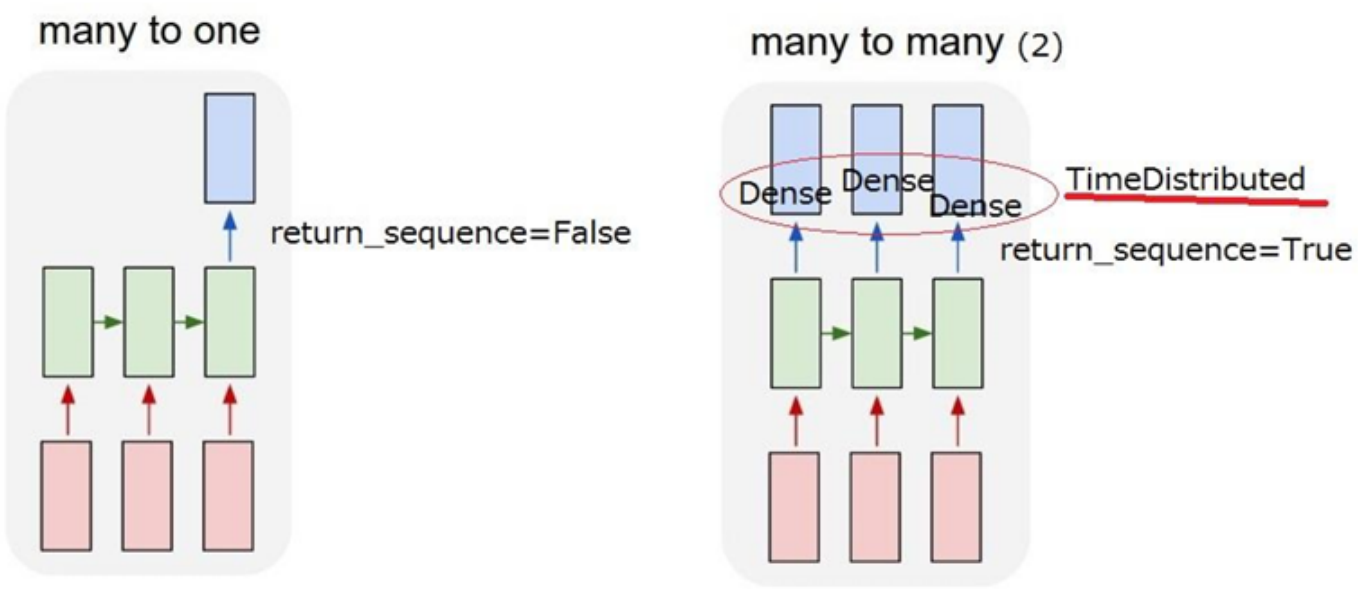
RNN은 새롭게 얻은 정보를 과거 정보로 업데이트 하여 시간적 순서를 고려한다.?

순환 신경망 이해하기

- Data - (batch_size, timesteps, input_features)
- Input() 함수의 timesteps 항목을 None으로 지정 - 가변 길이를 처리할 수 있음

코드 10-16 어떤 길이의 시퀀스도 처리할 수 있는 RNN 층

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))   (None, 16)
outputs = layers.SimpleRNN(16)(inputs)
```



return_sequence = False는 **순환층의 마지막 출력 스탭만 반환**하는 RNN 층을 만들 때 사용한다.
마지막 출력 스탭만 반환해서 outputs.shape 출력하면 (None , 16)이다.

return_sequence = True는 **순환층의 모든 출력 스텝 시퀀스를 반환**하는 RNN층을 만들 때 사용한다.
전체 출력 스탭을 반환해 outputs.shape을 출력하면 (120, 16)이다.

코드 10-17 마지막 출력 스텝만 반환하는 RNN 층

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
>>> print(outputs.shape)
(None, 16)
```

return_sequences=False가 기본값입니다.

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_53 (InputLayer)        [(None, 120, 14)]         0

simple_rnn_25 (SimpleRNN)    (None, 16)                496
```

코드 10-18 전체 출력 시퀀스를 반환하는 RNN 층

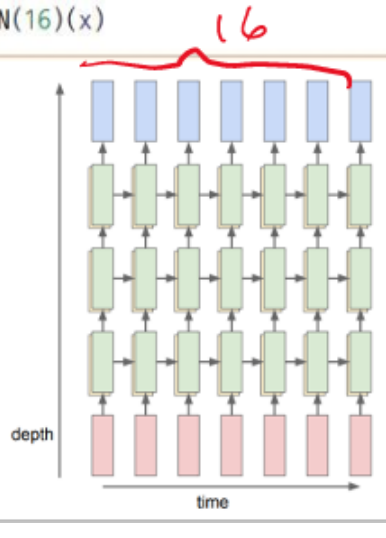
```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
>>> print(outputs.shape)
(120, 16)
```

RNN층의 네트워크 표현력 높이기

네트워크 표현력을 증가시키기 위해 여러 개의 순환층을 쌓는다. 이런 설정에서는 **중간 층들이 전체 출력 시퀀스를 반환하도록 한다. (return_sequence = True)**

코드 10-19 스태킹(stacking) RNN 층

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```



SimpleRNN은 너무 단순하고 Vanishing Gradient Problem(그래디언트 소실 문제)로 잘 쓰지 않는다.

이를 위해 고안된 LSTM과 GRU 층이 이다.

LSTM

시퀀스의 어느 지점에서 추출된 정보가 컨베이어 벨트 위로 올라가 필요한 시점의 타임 스텝으로 이동하여 떨구는것이 LSTM이 하는 일이다.

즉, 나중을 위해 정보를 저장함으로써 처리 과정에서 오래된 시그널이 점차 소실되는 것을 막아준다. (잔차 연결과 비슷하다.)

또한 **LSTM은셀 상태라는 추가 메모리 공간으로 중요한 정보를 유지해 RNN보다 장기 의존성을 더 잘 학습합니다.**

이런 구조적 특징으로 LSTM은 RNN보다 긴 시퀀스를 더 효과적으로 처리할 수 있습니다.

순환 신경망의 고급 사용법

순환 드롭 아웃

순환층에서 과대적합을 방지하기 위해서 recurrent_dropout = 0.25(드롭아웃 비율)를 적용했다.
참고로

unroll = True는 계산을 더 쉽게 병렬화하기 위해 사용한다.

드롭아웃 적용 시 수렴되는데 오래 걸리므로 에포크를 2배 늘려 훈련한다.

유닛의 수도 2배늘린 32개를 사용 LSTM(32, recurrent_dropout = 0.25)

코드 10-22 드롭아웃 규제를 적용한 LSTM 모델 훈련하고 평가하기

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x) -------- Dense 층에 규제를 추가하기 위해 LSTM 층 뒤에도 Dropout 층을 추가합니다.
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=50,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

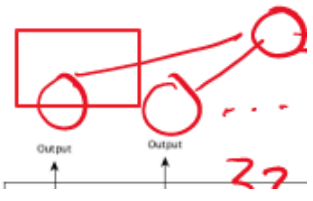
스태킹 순환 층

모델의 표현 능력을 증가시키기 위해 여러개의 순환층을 쌓는다. 이를 위해서는 return_sequence = True로 설정한다.

여기 예시에서는 GRU를 사용했다. GRU는 LSMT 구조보다 간단한 간소화 버전이다.

코드 10-23 드롭아웃 규제와 스태킹을 적용한 GRU 모델을 훈련하고 평가하기

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```



양방향 RNN 사용하기

양방향 RNN은 입력 시퀀스를 양쪽 방향으로 바라보기 때문에 잠재적으로 풍부한 표현을 얻을 수 있다. 또한 **단 방향 RNN이 놓치기 쉬운 패턴을 감지**할 수 있다.

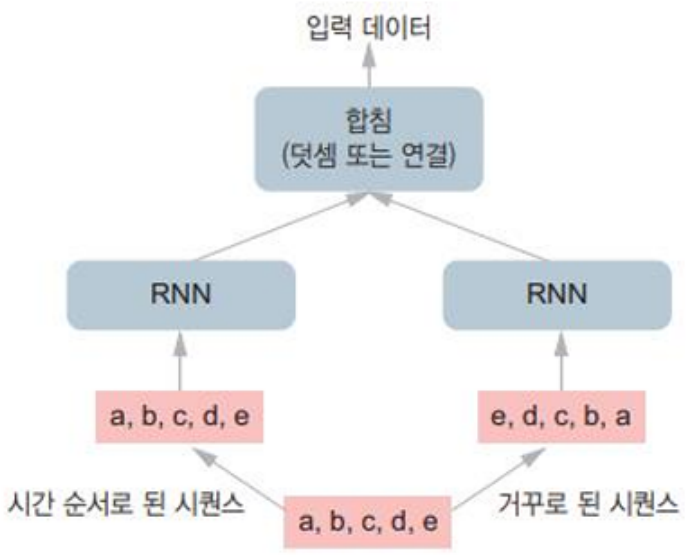
`양방향 RNN은 양쪽으로 입력 시퀀스를 바라보아 잠재적 표현을 더 잘 발견하고 단방향 RNN이 놓치기 쉬운 패턴을감지한다.`

코드 10-24 양방향 LSTM 모델 훈련하고 평가하기

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset)
```

Bidirectional 클래스는 전달받은 순환층으로 새로운 두 번째 객체를 만들어서 하나는 시간 순, 하나는 시간 역순으로 입력 시퀀스를 처리한다

`Bidirectional 클래스는 하나는 시간순, 하나는 시간의 역순으로 입력 시퀀스를 처리해 두 입력 시퀀스를 하나로 합친다.`



양방향 RNN은 평범한 LSTM 층 만큼 `성능이 좋지는 않고 네트워크 용량이 2배이고 일찍 과대 적합`이 된다. 하지만 `양방향 RNN은 텍스트 데이터 또는 순서가 중요한 다른 종류의 데이터에 적합` 합니다.