

Category 5

Sequences & Time Series



Sequences & Time Series

모델용 데이터 세트 분석

“ 1749년 01월 31일 부터 2018년 07월 31일 까지 관측된 월별 태양의 흑점개수 평균값이다. 전체 데이터는 3235개 시간 순서대로 얻어진 **Time Series(시계열) 데이터**이다

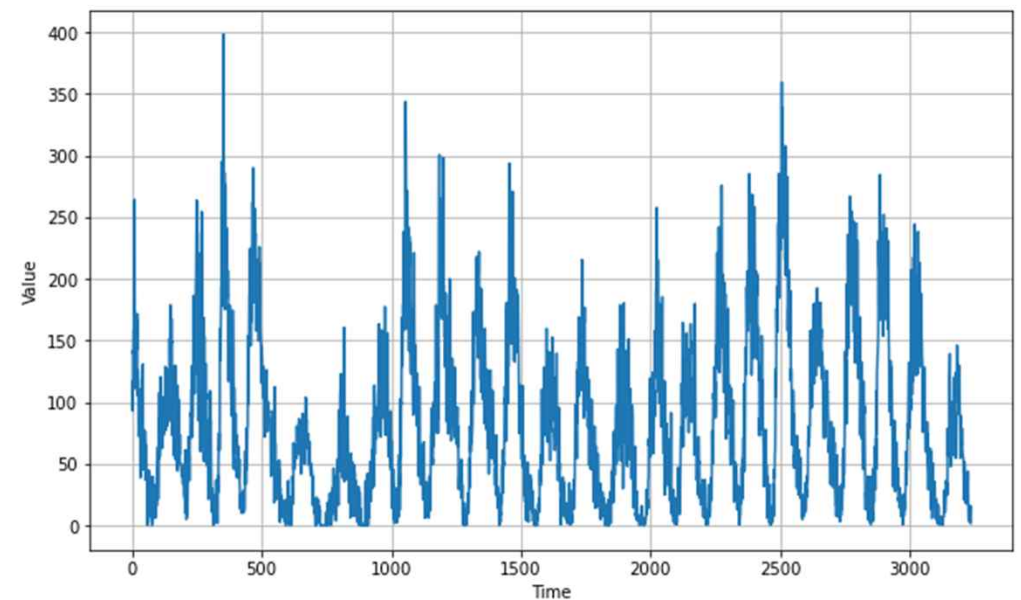
흑점은 주변 지역보다 어두운 점으로 나타나는 태양 광구의 일시적인 현상입니다. 대류를 억제하는 자기장 플럭스의 농도로 인해 표면 온도가 감소하는 영역입니다. 흑점은 일반적으로 반대의 자기 극성 쌍으로 나타납니다. 그들의 수는 대략 11 년의 태양주기에 따라 다릅니다.

<https://en.wikipedia.org/wiki/Sunspot>

sunspots.csv

Unnamed: 0		Date	Monthly Mean Total Sunspot Number
0	0	1749-01-31	96.7
1	1	1749-02-28	104.3
2	2	1749-03-31	116.7
3	3	1749-04-30	92.8
4	4	1749-05-31	141.7
...
3230	3230	2018-03-31	2.5
3231	3231	2018-04-30	8.9
3232	3232	2018-05-31	13.2
3233	3233	2018-06-30	15.9
3234	3234	2018-07-31	1.6

3235 rows × 3 columns



Sequences & Time Series

모델 개요

시계열 예측(time series prediction)

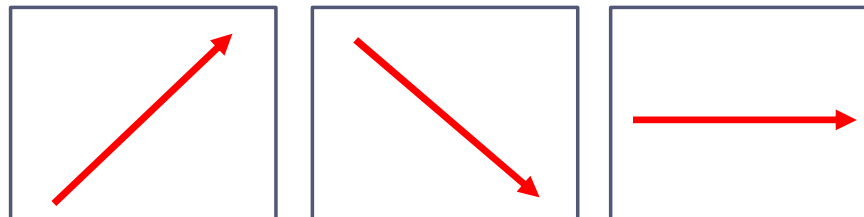
- “ **시계열 예측**(time series prediction)이라고 하는 것은 주어진 **시계열**을 보고 수학적인 **모델**을 만들어서 미래에 일어날 것들을 예측하는 것을 뜻하는 말이다
- “ 시간에 종속적으로 측정된 모든 데이터들은 시계열로 볼 수 있다. 물론 이런 때에는 시계열 데이터가 일정 시간 간격으로 주어진 것이 아닐 수도 있다. 종합 주가지수, 매일매일의 유가 변동사항, 환율 등 모든 데이터들은 시계열 데이터로 볼 수 있다. 따라서 시계열 해석은 미래를 예측하는 데에 중요한 도구가 될 수 있다.

Trend(추세)와 Seasonality(계절성)

“ **추세(trend)** 는 어떤 현상이 일정한 방향으로 나아가는 경향을 말한다

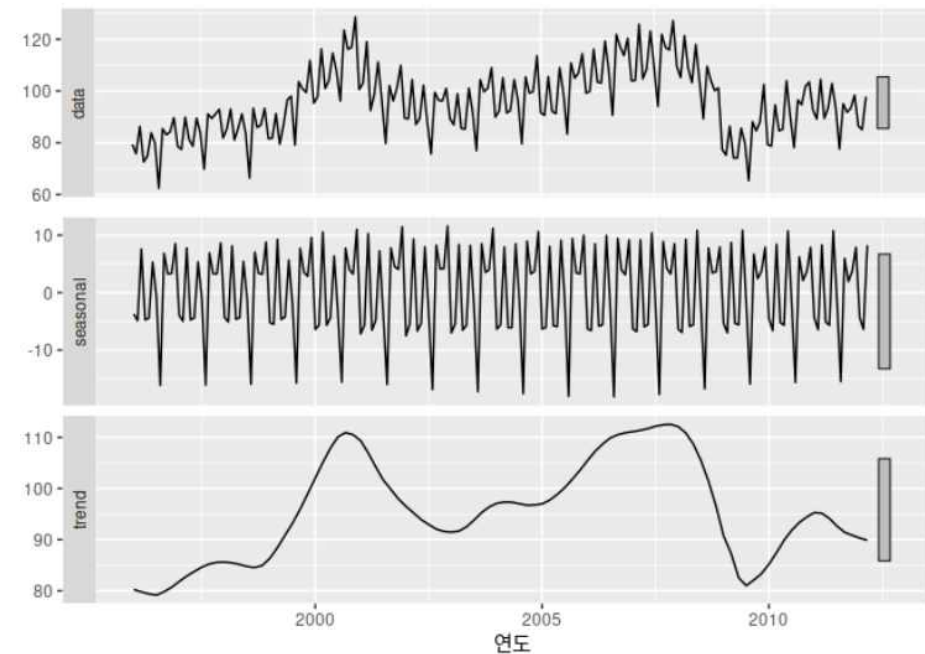
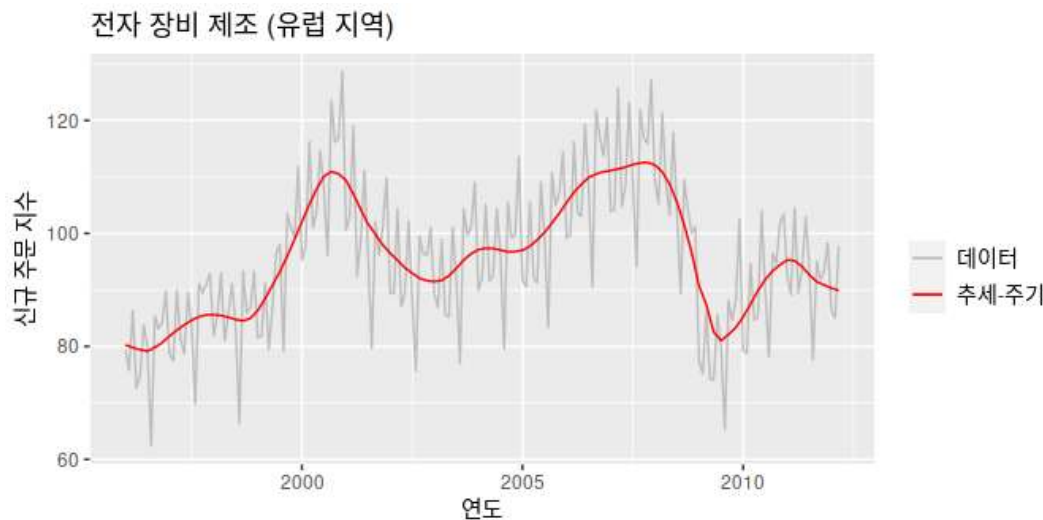
추세의 3가지 유형

1. 상승 추세(uptrends)
2. 하강 추세(downtrends)
3. 보합(sideways) or 수평(horizontal) 추세



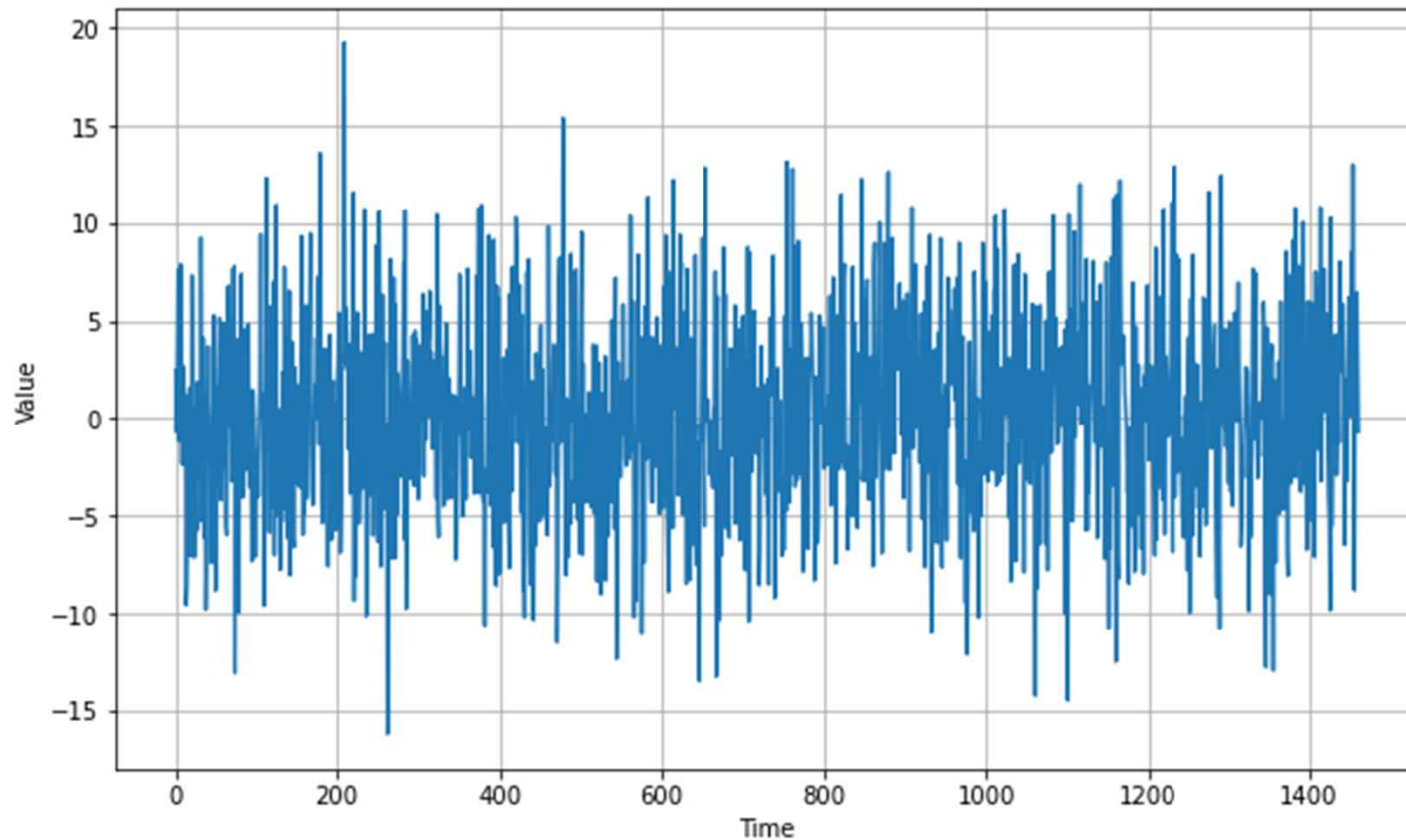
Trend(추세)와 Seasonality(계절성)

“ **계절성(Seasonality)** 는 일정한 주기를 두고 반복되는 패턴, 즉 주기성을 말한다



White Noise

시계열 데이터는 백색 잡음(white Noise)을 가질 수 있다



Naive Forecast / Moving Average Forecast

- naive forecast
 - 단순히 이전 값으로 예측
 - 이 값을 기준값으로 설정할 수 있음
- Moving Average (이동평균)
 - 단순하지만 노이즈를 줄일 수 있는 방법
 - 트렌드나 주기를 예측하진 않음
 - naive forecast 보다 정확도 낮음

Naive Forecast


A **naive forecast** is one in which the forecast for a given period is simply equal to the value observed in the previous period.

	A	B	C	D	E
1	Month	Actual Sales	Forecasted Sales	Formula used	
2	January	34			
3	February	37	34	=B2	
4	March	44	37	=B3	
5	April	47	44	=B4	
6	May	48	47	=B5	
7	June	48	48	=B6	
8	July	46	48	=B7	
9	August	43	46	=B8	
10	September	32	43	=B9	
11	October	27	32	=B10	
12	November	26	27	=B11	
13	December	24	26	=B12	
14					
15					
16					
17					
18					

Moving Average Forecast

Moving Averages

Week	Sales	3 MA				
1	39					
2	44					
3	40					
4	45	41				
5	38					
6	43					
7	39					
8						



1~3주의 평균값으로 4주차를 예측하고 2~4주차의 평균값으로 5주차를 예측...(반복)

Moving Average of Differencing Forecast

- Moving Average of Differencing (기간 차의 이동평균을 활용)
 - 예를 들어, 1년 전의 값의 차이를 계산하고, $V_{diff} = (V_t - V_{(t-365)})$
 - V_{diff} 의 이동평균을 계산 ($V_{t_diff_average}$)
 - 이전 날짜에 V_{diff} 를 이동평균한 값을 더해 다음 값을 예측: $V_{(t-365)} + V_{t_diff_average}$
- Forecasts = moving averaged differenced series + series($t - 365$)
- 이래도 노이즈는 남아있는데, 이럴 때 $V_{(t-365)}$ 대신 과거값을 이동평균한 값으로 사용하면 부드러워짐
- 즉, forecasts = $V_{moving_average}(t-365) + V_{t_diff_average}$
- Forecasts = trailing moving average of differenced series + centered moving average of past series($t-365$)
- 과거 값에 centered 이동 평균을 사용했고 (예: 시간 + 5와, 시간 - 5 사이의 평균) 이게 더 정확함. 현재값에 적용할 땐 trailing 이동 평균을 적용했는데, 이건 미래의 값을 알 수 없기 때문

Windowed Dataset

Windowed Dataset

- “ 시계열 데이터를 다룰 때 다음 과 같은 처리 코드를 직접 구현하여 sequence를 만들어주어야 하는 번거로움이 있다.

```
size = 5
data = list(range(10))
print(data)    # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
X=[]
for i in range(len(data)):
    _X = data[i:size + i]
    X.insert(i,_X)
print(X)
```

출력결과

```
[[0, 1, 2, 3, 4],
 [1, 2, 3, 4, 5],
 [2, 3, 4, 5, 6],
 [3, 4, 5, 6, 7],
 [4, 5, 6, 7, 8],
 [5, 6, 7, 8, 9],
 [6, 7, 8, 9],
 [7, 8, 9],
 [8, 9],
 [9]]
```

Windowed Dataset

“ **tf.data.Dataset**을 사용하면 여러 줄로 구성되어 있는 앞의 코드가 단 하나의 함수로 해결됩니다.

출력결과

```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1)
for window_dataset in dataset:
    for val in window_dataset:
        print(val.numpy(), end=" ")
    print()
```

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
6 7 8 9
7 8 9
8 9
9
```


window()

“ window() 메서드

```
window(  
    size, shift=None, stride=1, drop_remainder=False  
)
```

입력 요소 (중첩)를 창의 데이터 세트 (중첩)로 결합합니다.

"창"은 크기의 평면 요소의 유한 데이터 세트입니다 `size` (또는 창을 채우고로 `drop_remainder` 평가할 입력 요소가 충분하지 않은 경우 더 적을 수 있음 `False`).

`shift` 인수는 입력 요소의 수를 결정하는 각각의 반복에 의해 이동 창. 창과 요소가 모두 0부터 번호가 매겨진 경우 창의 첫 번째 요소는 입력 데이터 세트의 `k` 요소 `k * shift` 가됩니다. 특히 첫 번째 창의 첫 번째 요소는 항상 입력 데이터 세트의 첫 번째 요소가 됩니다.

`stride` 인수는 입력 요소의 보폭을 결정하고, `shift` 인수는 윈도우의 시프트를 판정한다.

window()

window() 메서드 인자와 반환 값

Args	
size	<code>tf.int64</code> 스칼라 <code>tf.Tensor</code> 입력 데이터 세트의 요소의 개수를 나타내는 창에 결합한다. 긍정적이어야 합니다.
shift	(선택 사항) 각 반복에서 창이 이동하는 입력 요소의 수를 나타내는 <code>tf.int64</code> 스칼라 <code>tf.Tensor</code> 입니다. 기본값은 size. 긍정적이어야 합니다.
stride	(선택 사항) 슬라이딩 윈도우에서 입력 요소의 보폭을 나타내는 <code>tf.int64</code> 스칼라 <code>tf.Tensor</code> 입니다. 긍정적이어야 합니다. 기본값 1은 "모든 입력 요소 유지"를 의미합니다.
drop_remainder	(선택 사항) 크기가보다 작은 경우 마지막 창을 삭제해야하는지 여부를 나타내는 <code>tf.bool</code> 스칼라 입니다. <code>tf.TensorSize</code>
보고	
Dataset	Dataset입력 소자들 (의 동지)에서 생성 된 평면 엘리먼트의 유한 셋 - (동지의) 윈도우.

window()

window() 메서드 : **drop_remainder=True** 인 경우 size 크기보다 작은 창은 삭제된다

```
1 dataset = tf.data.Dataset.range(10)
2 dataset = dataset.window(5, shift=1, drop_remainder=True)
3 for window_dataset in dataset:
4     for val in window_dataset:
5         print(val.numpy(), end=" ")
6     print()
```

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

batch()

```
batch(  
    batch_size, drop_remainder=False, num_parallel_calls=None, deterministic=None  
)
```

batch() 메서드는 연속된 데이터 세트의 요소들을 batch로 나누어 결합한다

```
1 # batch(batch_size, drop_remainder=False, num_parallel_calls=None, deterministic=None)  
2 dataset = tf.data.Dataset.range(10)  
3 dataset = dataset.batch(3)  
4 list(dataset.as_numpy_iterator())  
  
[array([0, 1, 2], dtype=int64),  
 array([3, 4, 5], dtype=int64),  
 array([6, 7, 8], dtype=int64),  
 array([9], dtype=int64)]
```

flat_map()

```
flat_map(  
    map_func  
)
```

데이터를 map_func에 매핑시켜 실행시킨 결과를 다시 flatten 시킨다

```
1 # flat_map(map_func)  
2 dataset = tf.data.Dataset.range(10)  
3 dataset = dataset.window(5, shift=1, drop_remainder=True)  
4 print(dataset)  
5 dataset = dataset.flat_map(lambda window: window.batch(5))  
6 for window in dataset:  
7     print(window.numpy())
```

```
<WindowDataset shapes: DatasetSpec(TensorSpec(shape=(), dtype=tf.int64, name=None), TensorShape([])), type  
s: DatasetSpec(TensorSpec(shape=(), dtype=tf.int64, name=None), TensorShape([]))>
```

```
[0 1 2 3 4]  
[1 2 3 4 5]  
[2 3 4 5 6]  
[3 4 5 6 7]  
[4 5 6 7 8]  
[5 6 7 8 9]
```

map()

```
map(  
    map_func, num_parallel_calls=None, deterministic=None  
)
```

데이터의 요소를 map_func에 매핑시켜 실행한다

```
>>> dataset = Dataset.range(1, 6) # ==> [ 1, 2, 3, 4, 5 ]  
>>> dataset = dataset.map(lambda x: x + 1)  
>>> list(dataset.as_numpy_iterator())  
[2, 3, 4, 5, 6]
```

shuffle()

```
shuffle(  
    buffer_size, seed=None, reshuffle_each_iteration=None  
)
```

데이터의 요소를 무작위로 섞는다

```
1 # shuffle(buffer_size, seed=None, reshuffle_each_iteration=None)  
2 dataset = tf.data.Dataset.range(10)  
3 dataset = dataset.window(5, shift=1, drop_remainder=True)  
4 dataset = dataset.flat_map(lambda window: window.batch(5))  
5 dataset = dataset.map(lambda window: (window[:-1], window[-1:]))  
6 dataset = dataset.shuffle(buffer_size=10)  
7 for x,y in dataset:  
8     print(x.numpy(), y.numpy())
```

```
[0 1 2 3] [4]  
[5 6 7 8] [9]  
[2 3 4 5] [6]  
[1 2 3 4] [5]  
[3 4 5 6] [7]  
[4 5 6 7] [8]
```

prefetch()

```
prefetch(  
    buffer_size  
)
```

데이터로 부터 prefetch 하여 Dataset를 반환한다
데이터의 input pipeline을 구현하여 성능을 향상 시켜준다

아주 규모가 큰 데이터의 경우 텐서플로가 멀티쓰레딩, 큐, 배치, prefetch와 같은 상세한 사항을 모두 대신 처리해준다.

prefetch()

- 훈련 속도를 더 빠르게
- `prefetch(1)` 을 호출하면 데이터셋은 항상 한 배치가 미리 준비되도록 최선을 (=알고리즘이 한 배치로 작업하는 동안 이 데이터셋이 동시에 다음 배치를 준비)
- GPU에서 훈련하는 스텝을 수행하는 것보다 짧은 시간안에 한 배치 데이터를 준비할 수 있다. (=GPU 100%활용하는 방법)
- `interleave`와 `map`에 `num_parallel_calls` 을 함께 사용하면 데이터를 적재하고 전처리할때 CPU의 멀티코어를 사용해 더 빠르게 준비 가능
- `prefetch`는 일반적으로 하나도 충분, `tf.data.experimental.AUTOTUNE` 을 전달하면 텐서플로가 자동으로 결정 (하지만 아직 실험 단계)
- GPU에서 데이터를 바로 프리페치할 수 있는 `tf.data.experimental.prefetch_to_device()` 를 확인해보자

from_tensor_slices()

```
@staticmethod
from_tensor_slices(
    tensors
)
```

인스턴스를 만들지 않아도 class의 메서드를 바로 실행할 수 있다
<https://wikidocs.net/21054>

주어진 텐서는 첫 번째 차원을 따라 슬라이스됩니다. 이 작업은 입력 텐서의 구조를 유지하여 각 텐서의 첫 번째 차원을 제거하고 이를 데이터 세트 차원으로 사용합니다. 모든 입력 텐서는 첫 번째 차원에서 동일한 크기를 가져야 합니다.

```
>>> # Slicing a tuple of 1D tensors produces tuple elements containing
>>> # scalar tensors.
>>> dataset = tf.data.Dataset.from_tensor_slices(([1, 2], [3, 4], [5, 6]))
>>> list(dataset.as_numpy_iterator())
[(1, 3, 5), (2, 4, 6)]
```

expand_dims()

```
tf.expand_dims(  
    input, axis, name=None  
)
```

axis값에 해당하는 축에 길이 1인 축을 추가로 삽입하여 차원을 늘린다

```
>>> image = tf.zeros([10,10,3])
```

```
>>> tf.expand_dims(image, axis=0).shape.as_list()  
[1, 10, 10, 3]
```

```
>>> tf.expand_dims(image, axis=1).shape.as_list()  
[10, 1, 10, 3]
```

```
>>> tf.expand_dims(image, -1).shape.as_list()  
[10, 10, 3, 1]
```

정리

`windowed_dataset`은 Time Series 데이터셋을 생성할때 매우 유용합니다.

`window` : 그룹화 할 윈도우의 크기(갯수)

`drop_remainder` : 남은 부분을 버릴지 살릴지

`shift` : 1 iteration 당 몇 개씩 이동할 것인지

`flat_map` : 데이터셋에 함수를 apply 해주고 결과를 flatten하게 펼쳐 줍니다.

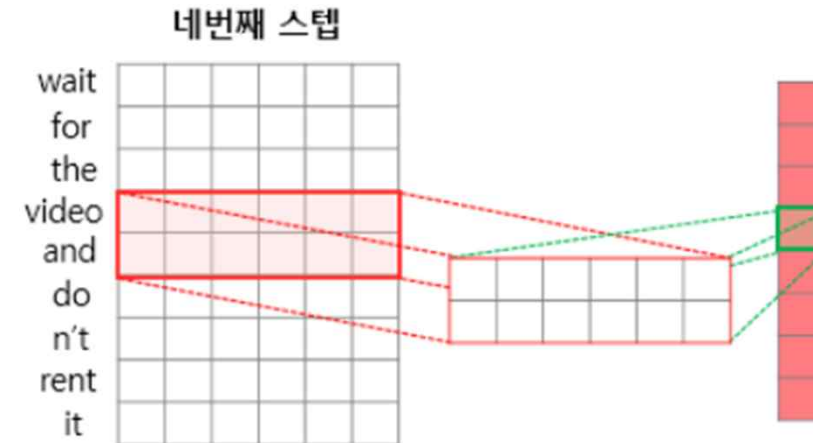
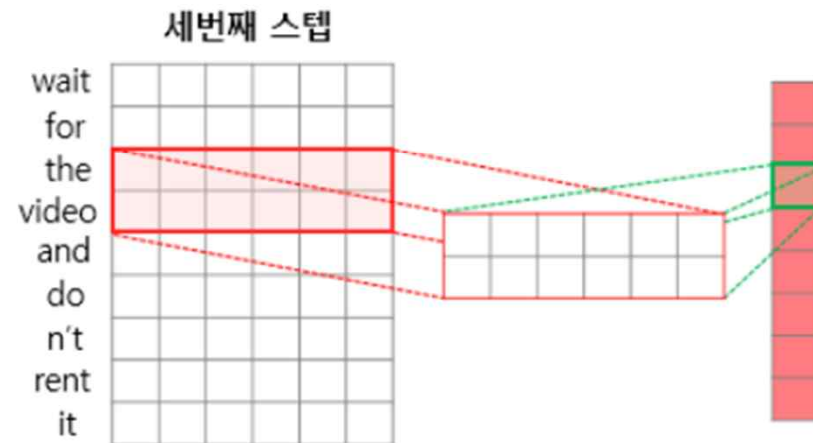
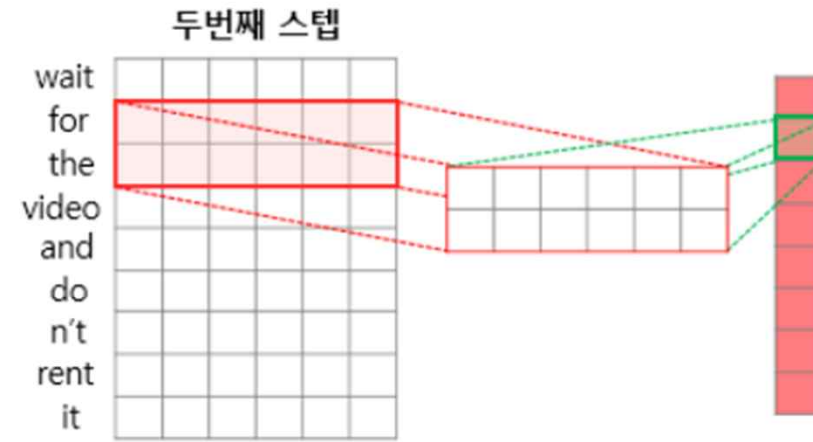
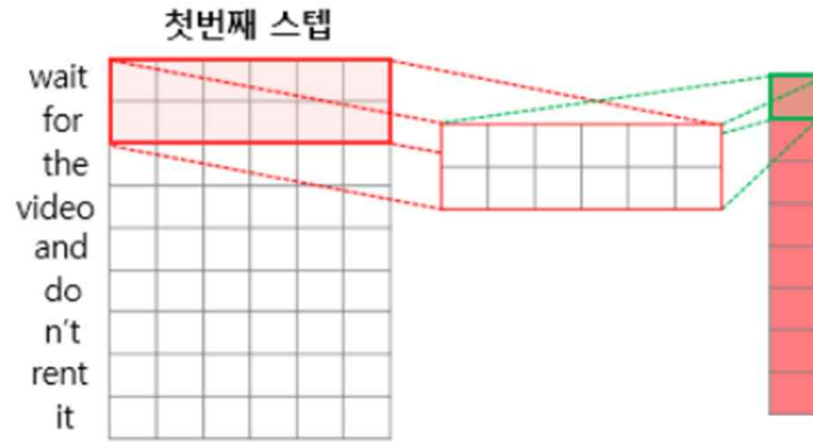
`shuffle` : 데이터 셋을 섞어 줍니다.

CNN for Sequence Model

시퀀스 데이터에 합성곱(Convolution) 적용하기

- “ 합성곱 신경망(CNN)의 convolution 연산을 보면, 입력 데이터의 각 위치에 커널(kernel)을 슬라이딩 해가며 곱하고, 그 곱한 값을 summation하여 단일 값들로 요약 표현 했다.
- “ 이것은 데이터에 내재된 local한 특징을 추출(extraction)하는 역할을 한다고 볼 수 있다.
- “ CNN이 널리 적용되는 2차원 이미지 데이터에 대해서 뿐만 아니라 1차원 시퀀스 데이터인 Text 데이터에도 동일한 목적으로 적용될 수 있다.
text 데이터에 적용되는 convoltion은 아래와 같다.
필터를 이동시킬수 있는 축(axis)이 순서에 관한 축 1개 밖에 없기 때문에 1D convolution이라고 부른다.

1D Convolution



1D Convolution

- 1D 합성곱 층에 적절한 수의 필터를 설정하면, 각 필터마다 시퀀스위를 슬라이딩하며 feature map으로써 길이가 줄어든 새로운 시퀀스가 필터 개수만큼 생성된다.
- 2D 합성곱과 마찬가지로 stride와 padding, dilate 등을 적용할 수 있으며, 그에 따라 output의 길이/차원을 조절, 특징 추출 간격을 통제할 수 있다.
- 1D Convolution layer와 Pooling layer, recurrent layer를 모두 섞어 신경망을 구성할 수 있다!!

“

1D Convolution

- 아래는 20개의 크기 4인 필터들을 사용하여 stride는 2로 설정하고 padding은 따로 적용하지 않는 1D 합성곱 층을 RNN기반 모델의 첫번째 층으로 사용한 예시이다.

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

이 예에서 Conv1D층은 stride를 2로 설정하여 입력 시퀀스를 **1/2배로 다운샘플링**한다.

이를 통해 모델은 입력되는 시퀀스에서 중요하지 않은 세부사항은 버리고 유용한 정보 위주로 보존하도록 학습할 것이라 기대할 수 있다.

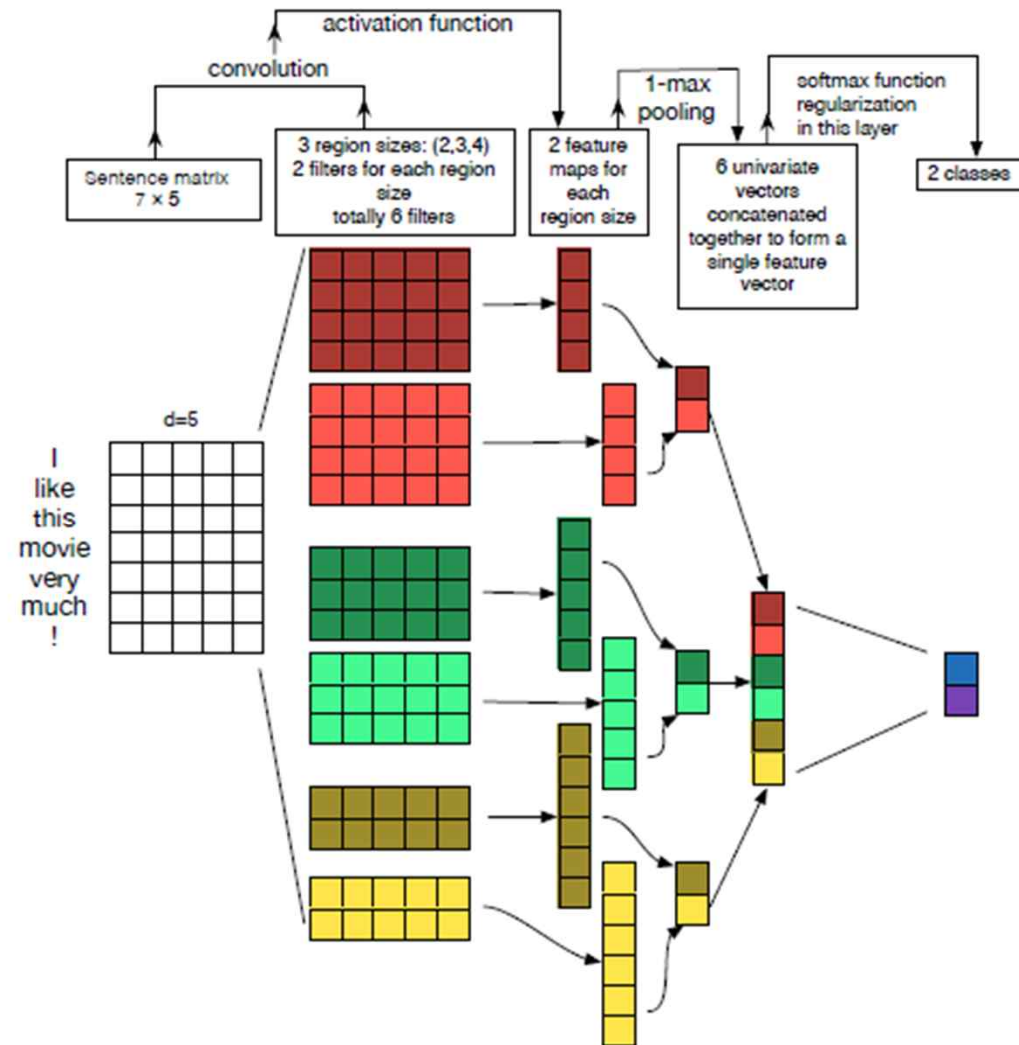
CNN for sequence modeling

Recurrent한 유닛을 사용하지 않고 오직 합성곱층만을 사용한 CNN으로도 시퀀스 모델링이 가능하다.

Yoon Kim (2014)

<https://arxiv.org/abs/1408.5882>

가 처음으로 발표



Keras Conv1D 레이어

```
tf.keras.layers.Conv1D(  
    filters, kernel_size, strides=1, padding='valid',  
    data_format='channels_last', dilation_rate=1, groups=1,  
    activation=None, use_bias=True, kernel_initializer='glorot_uniform',  
    bias_initializer='zeros', kernel_regularizer=None,  
    bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,  
    bias_constraint=None, **kwargs  
)
```

"valid", "same" 또는 "causal" (대소 문자 구분 안 함) 중 하나입니다. "valid" 패딩이 없음을 의미합니다. "same" 출력이 입력과 동일한 높이 / 너비 치수를 갖도록 입력의 왼쪽 / 오른쪽 또는 위 / 아래에 균등하게 0이 채워집니다. "causal" 인과 적 (확장 된) 컨볼 루션을 생성합니다. 예를 들어 output[t] 의존하지 않습니다 input[t+1:]. 모델이 시간 순서를 위반하지 않아야 하는 시간 데이터를 모델링 할 때 유용합니다. WaveNet: 원시 오디오를 위한 생성 모델, 섹션 2.1을 참조하십시오.

```
tf.keras.layers.Conv1D(filters=60, kernel_size=5,  
                        strides=1, padding="causal",  
                        activation="relu",  
                        input_shape=[None, 1]),  
tf.keras.layers.LSTM(60, return_sequences=True),  
tf.keras.layers.LSTM(60, return_sequences=True),  
tf.keras.layers.Dense(30, activation="relu"),  
tf.keras.layers.Dense(10, activation="relu"),  
tf.keras.layers.Dense(1),  
])
```

LearningRateScheduler

- “ 딥러닝 모델을 학습할 때 학습율(Learning Rate)을 감소시키는 방법이 자주 사용된다
- “ callback 함수에 호출될 함수를 등록하여 사용한다.

At the beginning of every epoch, this callback gets the updated learning rate value from `schedule` function provided at `__init__`, with the current epoch and current learning rate, and applies the updated learning rate on the optimizer.

Args

<code>schedule</code>	a function that takes an epoch index (integer, indexed from 0) and current learning rate (float) as inputs and returns a new learning rate as output (float).
<code>verbose</code>	int. 0: quiet, 1: update messages.

LearningRateScheduler

```
>>> # This function keeps the initial learning rate for the first ten epochs
>>> # and decreases it exponentially after that.
>>> def scheduler(epoch, lr):
...     if epoch < 10:
...         return lr
...     else:
...         return lr * tf.math.exp(-0.1)
>>>
>>> model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
>>> model.compile(tf.keras.optimizers.SGD(), loss='mse')
>>> round(model.optimizer.lr.numpy(), 5)
0.01
```

```
>>> callback = tf.keras.callbacks.LearningRateScheduler(scheduler)
>>> history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),
...                     epochs=15, callbacks=[callback], verbose=0)
>>> round(model.optimizer.lr.numpy(), 5)
0.00607
```

³⁷ https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler

LearningRateScheduler

```
lr_schedule = tf.keras.callbacks.LearningRateScheduler(  
    lambda epoch: 1e-8 * 10**(epoch / 20))  
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-8, momentum=0.9)  
model.compile(loss=tf.keras.losses.Huber(),  
              optimizer=optimizer,  
              metrics=["mae"])  
history = model.fit(train_set, epochs=100, callbacks=[lr_schedule])
```

학습율 감소는 지수적으로 하는 경우도 있고 계단 모양으로 똑똑 떨어뜨리는 경우도 있다

<https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=nostresss12&logNo=221544987534>

SGD

```
tf.keras.optimizers.SGD(  
    learning_rate=0.01, momentum=0.0, nesterov=False, name='SGD', **kwargs  
)
```

learning_rate	A Tensor, floating point value, or a schedule that is a <code>tf.keras.optimizers.schedules.LearningRateSchedule</code> , or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.01.
momentum	float hyperparameter ≥ 0 that accelerates gradient descent in the relevant direction and dampens oscillations. Defaults to 0, i.e., vanilla gradient descent.
nesterov	boolean. Whether to apply Nesterov momentum. Defaults to <code>False</code> .
name	Optional name prefix for the operations created when applying gradients. Defaults to "SGD".
**kwargs	Keyword arguments. Allowed to be one of "clipnorm" or "clipvalue". "clipnorm" (float) clips gradients by norm; "clipvalue" (float) clips gradients by value.

SGD - momentum

Update rule for parameter `w` with gradient `g` when `momentum` is 0:

```
w = w - learning_rate * g
```

Update rule when `momentum` is larger than 0:

```
velocity = momentum * velocity - learning_rate * g  
w = w + velocity
```

When `nesterov=True`, this rule becomes:

```
velocity = momentum * velocity - learning_rate * g  
w = w + momentum * velocity - learning_rate * g
```


L1,L2 Regularization

Regularization

λ = regularization strength
(hyperparameter)

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Max norm regularization (might see later)

Dropout (will see later)

Fancier: Batch normalization, stochastic depth

L1,L2 Regularization

“가중치(W)가 클수록 Loss를 줄여나가는데 더 큰 페널티가 있음을 알 수 있다.

이는 특정 변수의 가중치가 과도하게 커지는 것을 방지하는 역할을 하게 된다.(가중치가 커지면 Loss도 그만큼 증가하므로)

“이렇게 W 를 억제하는 방식의 정규화를 Weight decay라고 한다

L1,L2 Regularization

- “ λ (람다)는 Regularization term의 계수로서, 우리가 적절하게 설정해야하는 hyper parameter이다.
- “ λ 의 크기에 따라 regularization의 정도가 정해지므로 regularization strength라고도 한다.
- “ λ (람다)가 너무 클 때
: Loss를 최소화하기 위해 가중치(W)가 0에 수렴하게 된다
(과도한 일반화)
- “ λ (람다)가 너무 작을 때
: regularization의 효과가 없다.

L1 Regularizer

```
>>> @tf.keras.utils.register_keras_serializable(package='Custom', name='l1')
... def l1_reg(weight_matrix):
...     return 0.01 * tf.math.reduce_sum(tf.math.abs(weight_matrix))
...
>>> layer = tf.keras.layers.Dense(5, input_dim=5,
...     kernel_initializer='ones', kernel_regularizer=l1_reg)
>>> tensor = tf.ones(shape=(5, 5))
>>> out = layer(tensor)
>>> layer.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=0.25>]
```

L2 Regularizer

```
@tf.keras.utils.register_keras_serializable(package='Custom', name='l2')
class L2Regularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, l2=0.):
        self.l2 = l2

    def __call__(self, x):
        return self.l2 * tf.math.reduce_sum(tf.math.square(x))

    def get_config(self):
        return {'l2': float(self.l2)}

layer = tf.keras.layers.Dense(
    5, input_dim=5, kernel_initializer='ones',
    kernel_regularizer=L2Regularizer(l2=0.5))
```

The End