간단한 RNN을 구현해 보기

학습 목표

- 딥러닝 대표 알고리즘 RNN을 실습을 통해 빠르게 구현해 본다.

목차

01. RNN 기본 이해

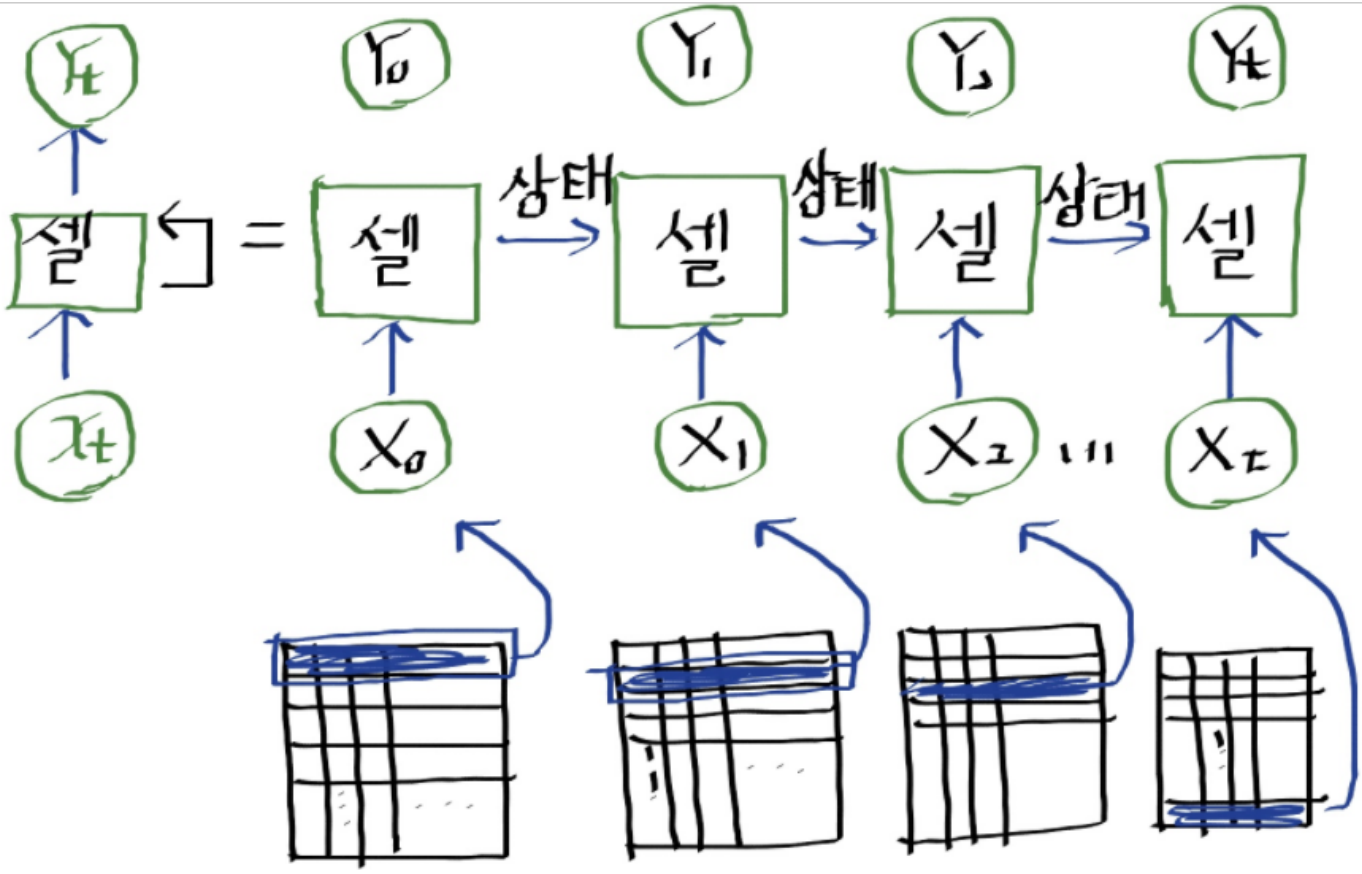
목차로 이동하기

RNN의 용어 이해

- 순환 신경망(Recurrent Neural Network)의 약자이다.
- hidden state : 이전의 정보를 기억하는 공간
- RNN은 상태가 고정된 데이터를 처리하는 다른 신경망과 달리 자연어 처리나 음성 인식처럼 **순서가 있는 데이터를 처리하는 데 강점**이 있다.
- 앞이나 뒤의 정보에 따라 전체의 의미가 달라질 때,
- **앞의 정보로 다음에 나오는 정보를 추측**하려고 할 때, RNN을 사용하면 좋은 프로그램을 만들 수 있다.
- 2016년 구글의 신경망 기반 기계 번역이 RNN을 이용하여 만든 서비스이다.

RNN의 구조 이해

- RNN은 셀을 여러개 중첩하여 심층 신경망을 만든다.
- 앞의 학습 결과를 다음 단계의 학습에 이용한다.
- 학습 데이터를 단계별로 구분하여 입력을 한다.
- MNIST를 RNN에 적용한다고 하면, 한 줄단위(28픽셀)을 한 단계의 입력값으로 한다. 총 28단계를 거쳐 입력받음.



RNN 작업 반복 과정

- 한 단계를 학습한 뒤, 상태를 저장한다.
- 그 상태를 다음 단계의 입력 상태로 하여 다시 학습한다.
- 주어진 단계만큼 반복하면서 상태를 전파하며 출력값을 만들어간다. RNN의 기본 구조

02. 라이브러리 준비

목차로 이동하기

In [1]:

```
import tensorflow as tf
import keras
```

In [2]:

```
print(tf.__version__)
print(keras.__version__)
```

```
2.9.1
2.9.0
```

In [3]:

```
import numpy as np
```

데이터 준비

In [4]:

```
for i in range(10):
    lst = list(range(i, i+5))
    print(lst)
```

```
[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, 10]
[7, 8, 9, 10, 11]
[8, 9, 10, 11, 12]
[9, 10, 11, 12, 13]
```

데이터 준비

- X는 3D 텐서 - 5개의 값씩 한세트가 되어 10개의 세트가 있음.
- Y는 1D 텐서 - 0.5~1.4 까지 0.1씩 증가하면서 10개의 데이터\
- X가 0, 0.1, 0.2, 0.3, 0.4 라면 Y는 0.5
- X가 0.1, 0.2, 0.3, 0.4, 0.5 라면 Y는 0.6

In [5]:

```
X = []
Y = []
for i in range(10):
    lst = list(range(i, i+5))
    X.append( [ [c/10] for c in lst]  )  # 문제
    Y.append( (i+5)/10  )  #

X = np.array(X)
Y = np.array(Y)

print( X.shape, Y.shape ) # 10개의 샘플 (5,1), 다음 0.5
print( X[0], Y[0])
print()
print( X[1], Y[1])
print()
print(Y)
print(X)
```

```
(10, 5, 1) (10,)
[[0. ]
 [0.1]
 [0.2]
 [0.3]
 [0.4]] 0.5

[[0.1]
 [0.2]
 [0.3]
 [0.4]
 [0.5]] 0.6

[0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4]
[[[0. ]
  [0.1]
  [0.2]
  [0.3]
  [0.4]]

 [[0.1]
  [0.2]
  [0.3]
  [0.4]
  [0.5]]

 [[0.2]
  [0.3]
  [0.4]
  [0.5]
  [0.6]]

 [[0.3]
  [0.4]
  [0.5]
  [0.6]
  [0.7]]

 [[0.4]
  [0.5]
```

```
 [0.6]
 [0.7]
 [0.8]]

[[0.5]
 [0.6]
 [0.7]
 [0.8]
 [0.9]]

[[0.6]
 [0.7]
 [0.8]
 [0.9]
 [1. ]]

[[0.7]
 [0.8]
 [0.9]
 [1. ]
 [1.1]]

[[0.8]
 [0.9]
 [1. ]
 [1.1]
 [1.2]]

[[0.9]
 [1. ]
 [1.1]
 [1.2]
 [1.3]]]
```

In [6]:

```
# 전체 데이터 확인
for i in range(len(X)):
    print(X[i], Y[i])
print( X.shape, Y.shape )
```

```
[[0. ]
 [0.1]
 [0.2]
 [0.3]
 [0.4]] 0.5
[[0.1]
 [0.2]
 [0.3]
 [0.4]
 [0.5]] 0.6
[[0.2]
 [0.3]
 [0.4]
 [0.5]
 [0.6]] 0.7
[[0.3]
 [0.4]
 [0.5]
 [0.6]
 [0.7]] 0.8
[[0.4]
 [0.5]
 [0.6]
 [0.7]
 [0.8]] 0.9
[[0.5]
 [0.6]
 [0.7]
 [0.8]
 [0.9]] 1.0
[[0.6]
 [0.7]
 [0.8]
 [0.9]
 [1. ]] 1.1
[[0.7]
 [0.8]
 [0.9]
 [1. ]
 [1.1]] 1.2
[[0.8]
 [0.9]
 [1. ]
 [1.1]
 [1.2]] 1.3
[[0.9]
 [1. ]
 [1.1]
 [1.2]
 [1.3]] 1.4
(10, 5, 1) (10,)
```

03. 모델 구축 - SimpleRNN

목차로 이동하기

- SimpleRNN은 하나의 시퀀스가 아니라 시퀀스 배치를 처리한다는 것.
  - (timesteps, input_features) -> (batch_size, timesteps, input_features) 입력받음.
- SimpleRNN은 두 가지 실행 모드가 가능(return_sequences 매개변수로 선택 가능)
  - return_sequences= False
    - 입력 시퀀스에 대한 마지막 출력 반환 - (batch_size, output_features) - 2D 텐서
  - return_sequences= True
    - 각 타임스텝 출력을 모은 전체 시퀀스 반환 - (batch_size, timesteps, output_features) - 3D 텐서

- 매개변수
  - return_sequences : 기본값(False)
    - False : 마지막 상태만 출력
    - True : 모든 지점의 은닉 상태 출력
  - return_state : 기본값(False)
    - True : return_sequences의 값과 상관없이 마지막 은닉 상태를 출력

- 코드 구현

```
model = tf.keras.Sequential([
  tf.keras.layers.SimpleRNN(units=10, return_sequences=False, input_shape=[5,1]),
  tf.keras.layers.Dense(1)
])
```

In [7]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN
```

return_sequences가 True의 경우

In [8]:

```
model = Sequential()
model.add(SimpleRNN(10, return_sequences=True, input_shape=[5,1]))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 5, 10)             120

=================================================================
Total params: 120
Trainable params: 120
Non-trainable params: 0
_________________________________________________________________
```

- 출력이 3D텐서
  - batch_size, timesteps, output_features

return_sequences가 False의 경우

In [9]:

```
model = Sequential()
model.add(SimpleRNN(10, return_sequences=False, input_shape=[5,1]))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_1 (SimpleRNN)    (None, 10)                120

=================================================================
Total params: 120
Trainable params: 120
Non-trainable params: 0
_________________________________________________________________
```

- 출력이 2D텐서
  - batch_size, output_features

모델 compile()

In [10]:

```
model.compile(optimizer='adam', loss='mse')
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 simple_rnn_1 (SimpleRNN)     (None, 10)                120

=================================================================
Total params: 120
Trainable params: 120
Non-trainable params: 0
_________________________________________________________________
```

파라미터 개수

- model.add(SimpleRNN(10, return_sequences=False, input_shape=[5,1]))
  - hidden state의 Dh는 10
  - t=5 (RNN의 특성상 모든 시점의 히든 스테이트는 공유. time은 변수의 가중치와 관계없음.
  - input_dim(d) = 1
- of params = (Dh * Dh) + (Dh * input_dim) + (Dh)
  - (10 * 10) + (10 * 1) + (10) = 120

In [11]:

```
model.fit(X, Y, epochs=50, verbose=1)
```

```
Epoch 1/50
1/1 [==============================] - 1s 620ms/step - loss: 1.0241
Epoch 2/50
1/1 [==============================] - 0s 4ms/step - loss: 1.0171
Epoch 3/50
1/1 [==============================] - 0s 4ms/step - loss: 1.0103
Epoch 4/50
1/1 [==============================] - 0s 4ms/step - loss: 1.0036
Epoch 5/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9970
Epoch 6/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9906
Epoch 7/50
1/1 [==============================] - 0s 5ms/step - loss: 0.9842
Epoch 8/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9779
Epoch 9/50
1/1 [==============================] - 0s 5ms/step - loss: 0.9717
Epoch 10/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9657
Epoch 11/50
1/1 [==============================] - 0s 3ms/step - loss: 0.9598
Epoch 12/50
1/1 [==============================] - 0s 5ms/step - loss: 0.9539
Epoch 13/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9482
Epoch 14/50
1/1 [==============================] - 0s 3ms/step - loss: 0.9426
Epoch 15/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9371
Epoch 16/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9317
Epoch 17/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9264
Epoch 18/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9212
Epoch 19/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9161
Epoch 20/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9110
Epoch 21/50
1/1 [==============================] - 0s 5ms/step - loss: 0.9061
Epoch 22/50
1/1 [==============================] - 0s 4ms/step - loss: 0.9012
Epoch 23/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8964
Epoch 24/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8917
Epoch 25/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8870
Epoch 26/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8824
Epoch 27/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8778
Epoch 28/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8733
```

```
Epoch 29/50
1/1 [==============================] - 0s 3ms/step - loss: 0.8689
Epoch 30/50
1/1 [==============================] - 0s 3ms/step - loss: 0.8644
Epoch 31/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8600
Epoch 32/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8557
Epoch 33/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8513
Epoch 34/50
1/1 [==============================] - 0s 5ms/step - loss: 0.8470
Epoch 35/50
1/1 [==============================] - 0s 3ms/step - loss: 0.8427
Epoch 36/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8384
Epoch 37/50
1/1 [==============================] - 0s 6ms/step - loss: 0.8341
Epoch 38/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8298
Epoch 39/50
1/1 [==============================] - 0s 5ms/step - loss: 0.8256
Epoch 40/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8213
Epoch 41/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8170
Epoch 42/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8127
Epoch 43/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8085
Epoch 44/50
1/1 [==============================] - 0s 4ms/step - loss: 0.8042
Epoch 45/50
1/1 [==============================] - 0s 4ms/step - loss: 0.7999
Epoch 46/50
1/1 [==============================] - 0s 4ms/step - loss: 0.7956
Epoch 47/50
1/1 [==============================] - 0s 4ms/step - loss: 0.7912
Epoch 48/50
1/1 [==============================] - 0s 3ms/step - loss: 0.7869
Epoch 49/50
1/1 [==============================] - 0s 4ms/step - loss: 0.7825
Epoch 50/50
1/1 [==============================] - 0s 4ms/step - loss: 0.7781
```

Out[11]:

```
<keras.callbacks.History at 0x1dba66c8b80>
```

In [12]:

```
print(X.shape, X)
```

```
(10, 5, 1) [[[0. ]
  [0.1]
  [0.2]
  [0.3]
  [0.4]]

 [[0.1]
  [0.2]
  [0.3]
  [0.4]
  [0.5]]

 [[0.2]
  [0.3]
  [0.4]
  [0.5]
  [0.6]]

 [[0.3]
  [0.4]
  [0.5]
  [0.6]
  [0.7]]

 [[0.4]
  [0.5]
  [0.6]
  [0.7]
  [0.8]]

 [[0.5]
  [0.6]
  [0.7]
  [0.8]
  [0.9]]

 [[0.6]
  [0.7]
  [0.8]
  [0.9]
  [1. ]]

 [[0.7]
  [0.8]
  [0.9]
  [1. ]
  [1.1]]

 [[0.8]
  [0.9]
  [1. ]
  [1.1]
  [1.2]]

 [[0.9]
  [1. ]
```

```
  [1.1]
  [1.2]
  [1.3]]]
```

In [13]:

```
print(Y) # 실제값
pred = model.predict(X)
np.max(pred, axis=0)
```

```
[0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4]
1/1 [==============================] - 0s 136ms/step
```

Out[13]:

```
array([ 0.76799405, -0.19018568,  0.7338462 ,  0.35154802,  0.35796756,
       -0.18646191,  0.24571645,  0.44652814,  0.7476799 ,  0.92708737],
      dtype=float32)
```