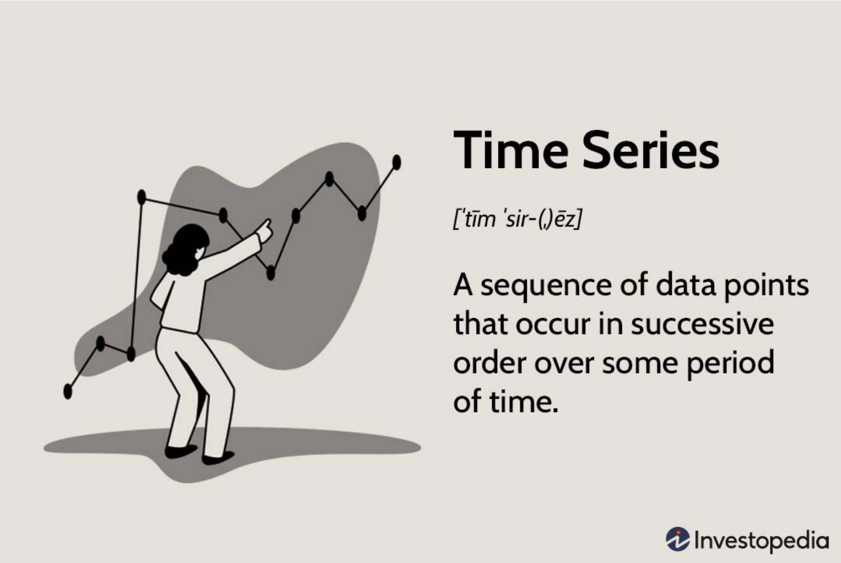
# Recurrent Neural Networks (RNNs)

November 2023

http://link.koreatech.ac.kr

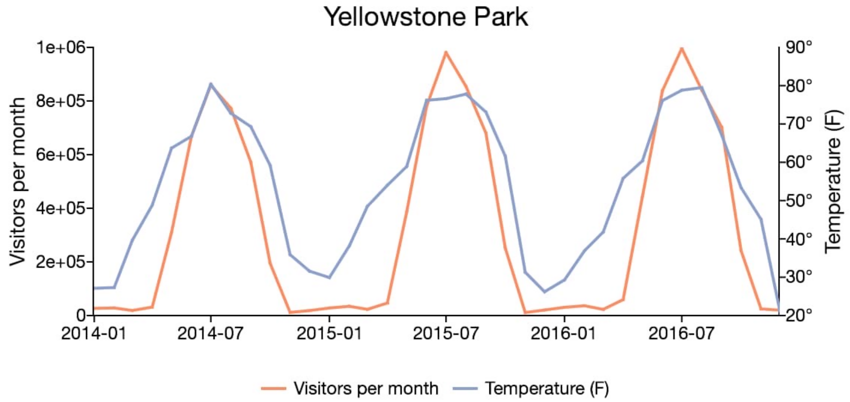
# Time Series Data

# 시계열 데이터




## ◈시계열 데이터 (Time Series Data)

- $d_t$: 순서가 있는 연속 데이터 (Sequenced Data)
- $t = 1, 2, 3, \ldots$
  - $t$가 실제로 일정 간격의 시각에 대응되는 경우가 많음
  - 하지만, 어떤 경우에서는 단순한 이벤트 기반 순서만 나타낼 수 있음
- 예: 음성, 동영상, 텍스트 등
- 연속렬 데이터의 길이 $T$는 고정 또는 가변
  - 종료 시점이 없을 수도 있음 → 무한 데이터

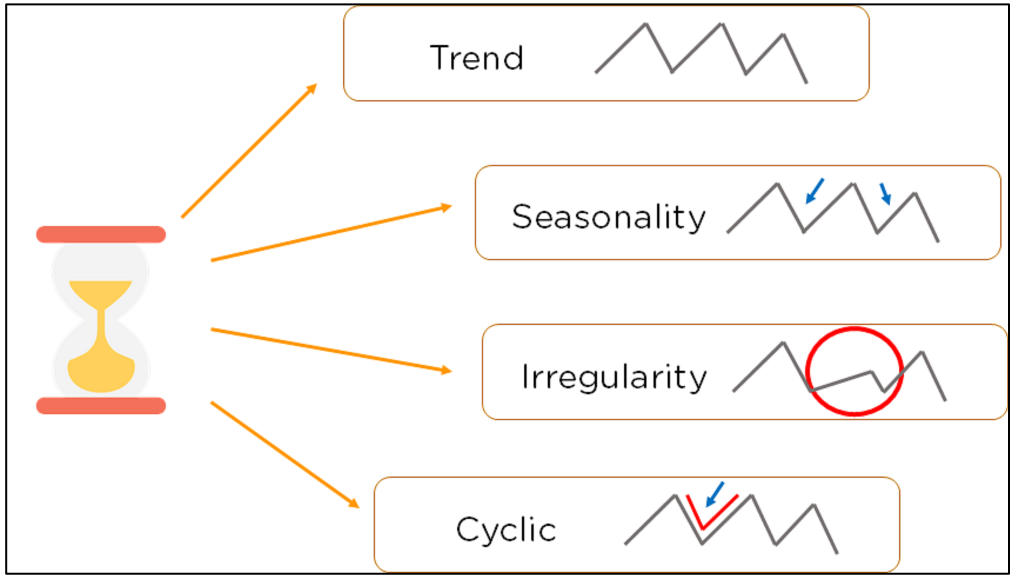



| Date | Ozone (μg/m$^3$) | Temperature (°C) | Relative humidity (%) | $n$ deaths |
|---|---|---|---|---|
| 1 Jan 2002 | 4.59 | −0.2 | 75.7 | 199 |
| 2 Jan 2002 | 4.88 | 0.1 | 77.5 | 231 |
| 3 Jan 2002 | 4.71 | 0.9 | 81.3 | 210 |
| 4 Jan 2002 | 4.14 | 0.5 | 85.4 | 203 |
| 5 Jan 2002 | 2.01 | 4.3 | 93.5 | 224 |
| 6 Jan 2002 | 2.4 | 7.1 | 96.4 | 198 |
| 7 Jan 2002 | 4.08 | 5.2 | 93.5 | 180 |
| 8 Jan 2002 | 3.13 | 3.5 | 81.5 | 188 |
| 9 Jan 2002 | 2.05 | 3.2 | 88.3 | 168 |
| 10 Jan 2002 | 5.19 | 5.3 | 85.4 | 194 |
| 11 Jan 2002 | 3.59 | 3.0 | 92.6 | 223 |
| 12 Jan 2002 | 12.87 | 4.8 | 94.2 | 201 |

# 시계열 데이터 활용

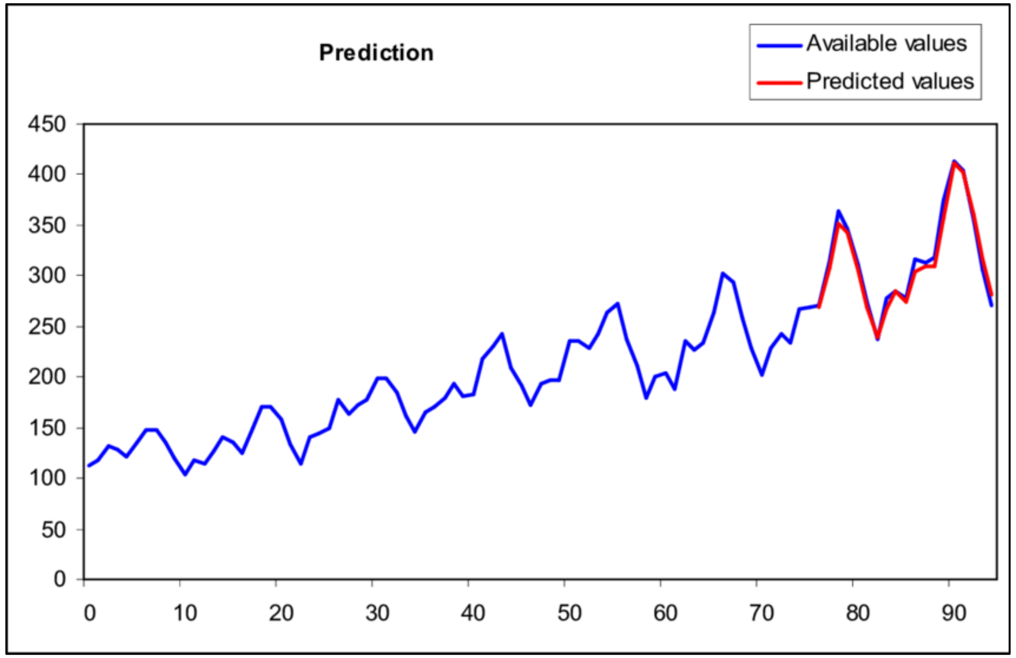
## ◈시계열 해석(Time Series Analysis)

– 시계열을 해석하고 이해하는 데 쓰이는 여러 가지 방법을 연구
  - 이런 시계열이 어떤 법칙에서 생성되어서 나오는가에 대한 기본적인 질문을 이해




## ◈시계열 예측(time series prediction)

– 주어진 시계열을 보고 모델을 만들어서 미래에 일어날 것들을 예측하는 것



# 시계열 예측 예

## ◈단어 예측 문제

- 입력 연속렬 데이터 $x^1, x^2, x^3, x^4, \ldots, x^t$로 부터 $y^t$를 예측하는 문제
- 전체적인 문맥을 학습하여 다음에 올 단어를 높은 정확도로 예측

to be or not to be that is the question

| word | to | be | or | … | is | the | ? |
|---|---|---|---|---|---|---|---|
| input | $x^1$ | $x^2$ | $x^3$ | | $x^{t-1}$ | $x^t$ | |
| output | | $y^1$ | $y^2$ | | $y^{t-2}$ | $y^{t-1}$ | $y^t$ |

# RNN Cell & RNN Layer

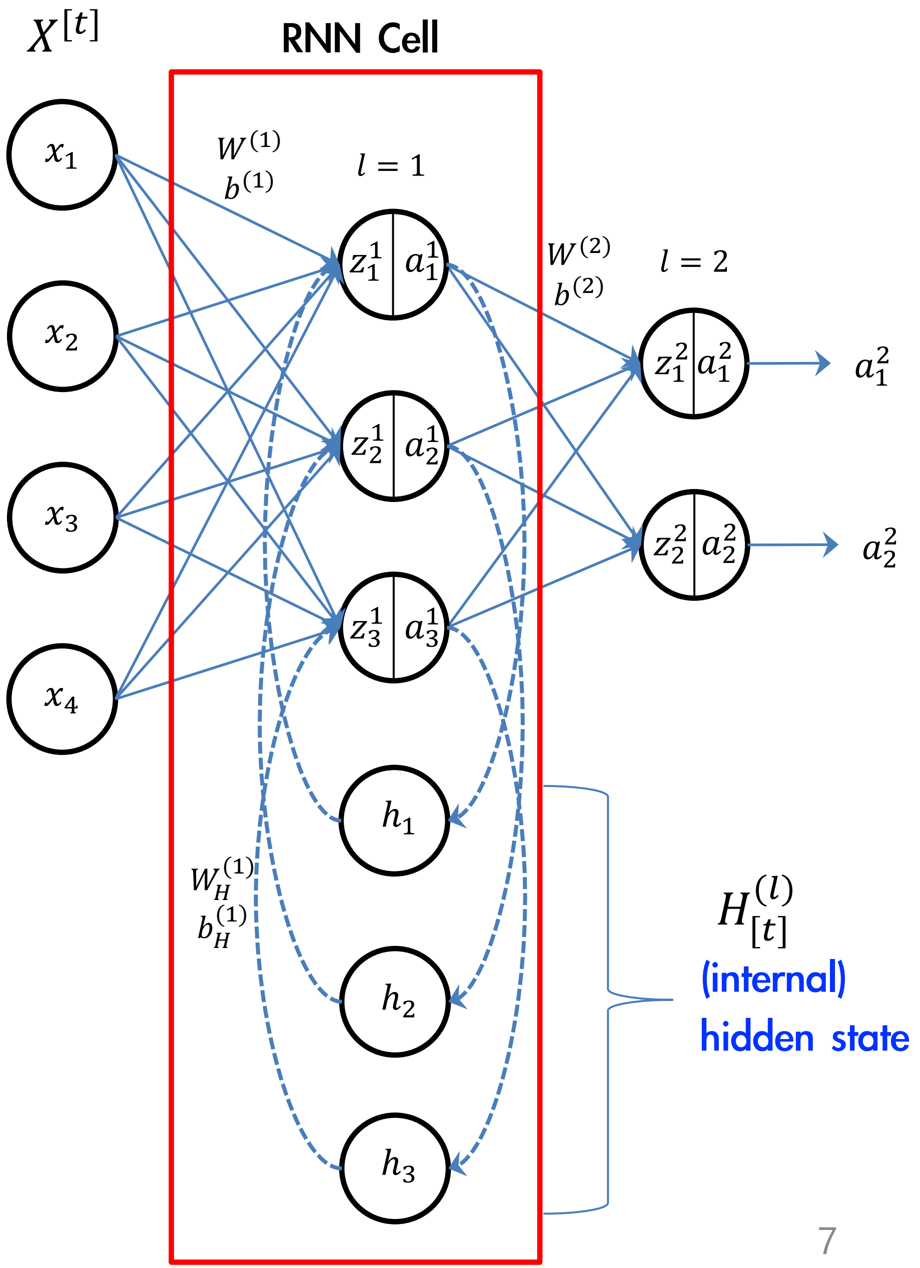
# RNN Cell

## ◈RNN Cell - Definition

– RNN cell includes the ability to maintain internal memory with feedback and therefore support temporal behavior

– RNN cell remains feed-forward still, but also maintains (internal) hidden state $H_{[t]}^{(l)} = (h_1, h_2, \dots, h_I)_{[t]}^{(l)}$ across each time step $t$

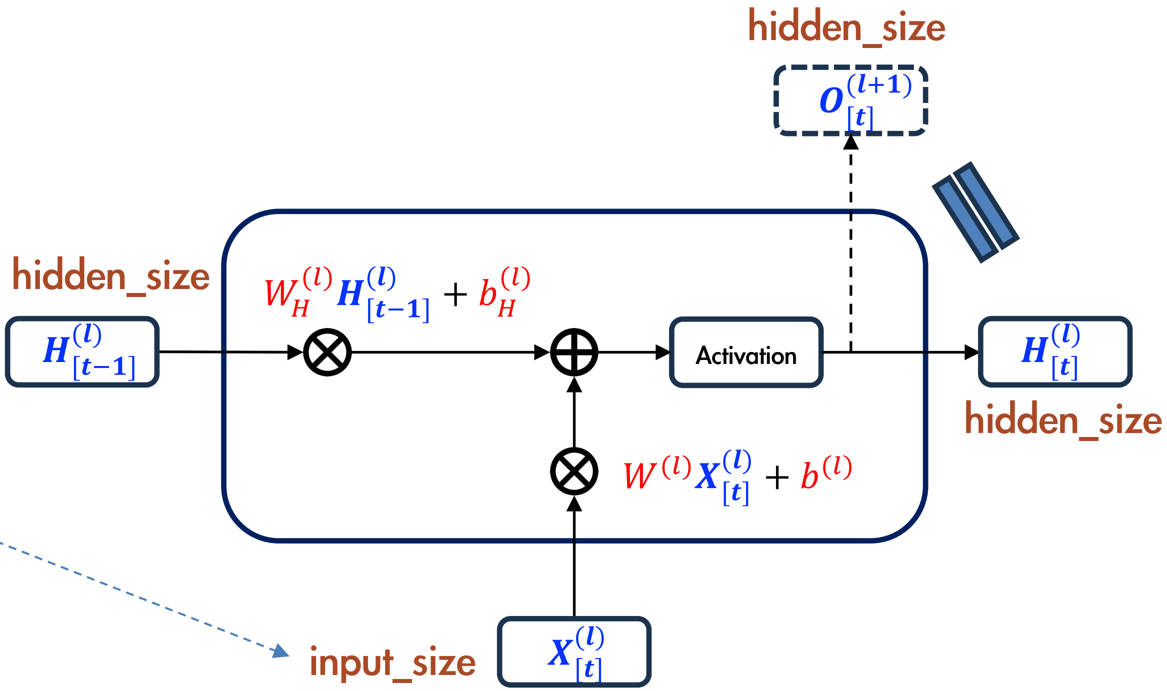
– The term "cell" captures the recurrent and sequential nature of the operation



# RNN Cell

## ◈RNN Cell

```python
rnn_cell = nn.RNNCell(
  input_size=3, hidden_size=4,
  bias=True, nonlinearity='tanh', device=None
)

for name, parameter in rnn_cell.named_parameters():
  print(name, parameter.shape)

# >>> weight_ih torch.Size([4, 3]): W^(l)
# >>> weight_hh torch.Size([4, 4]): W_H^(l)
# >>> bias_ih torch.Size([4]): b^(l)
# >>> bias_hh torch.Size([4]): b_H^(l)
```




$$H^{[t]} = Activation\left(\underbrace{W_H^{(l)} H^{(l)}_{[t-1]}}_{\text{matmul}} + b_H^{(l)} + \underbrace{W^{(l)} X^{(l)}_{[t]}}_{\text{matmul}} + b^{(l)}\right)$$

[4, 4]∘[4] + [4] + [4, 3]∘[3] + [4]

= [4] + [4] + [4] + [4] = [4]

Why the "tanh" as the default in RNNs over "ReLU"?

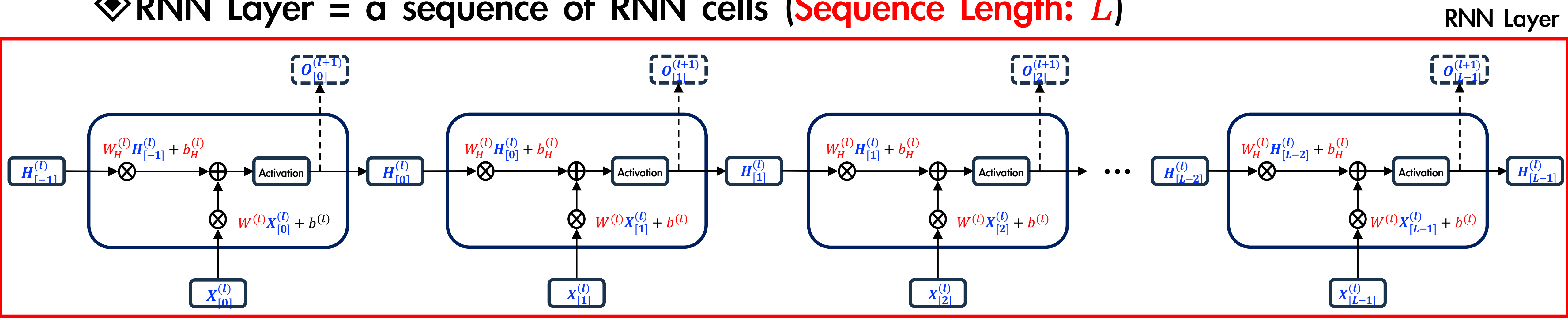
- Stability in Recurrent Connections

  : The bounded nature of "tanh" can lead to more stable behavior in the recurrent connections of an RNN.

  : ReLU's unbounded output can cause the activations to explode, especially in deep or long RNNs.

# RNN Layer

◈RNN Layer = a sequence of RNN cells (Sequence Length: $L$)

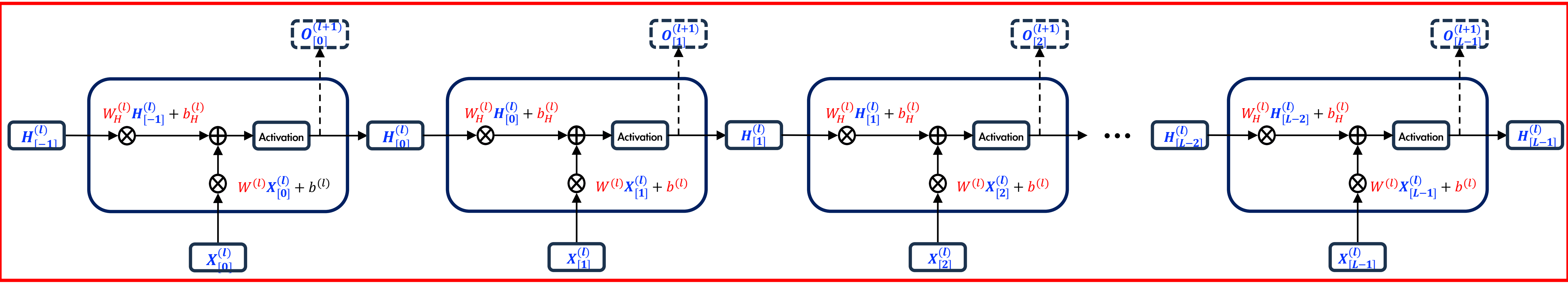



| | RNN Cell | RNN Layer |
|---|---|---|
| Definition | A single recurrent computation unit that processes one time step of the input sequence | A series of recurrent cells applied over the entire sequence |
| Units | Processes one time step where simple transformations are executed with weights and biases | Processes a sequence of time steps where simple transformations are executed with same weights and biases |
| Function | Produces an output and a hidden state | Processes the entire input sequence, producing a sequence of outputs (or hidden states) |
| Flow of Data | Processes input and the previous hidden state to produce a new hidden state and output | Data flows horizontally (from one time step to the next) and can also flow vertically in RNNs (from one layer to the next) |

# RNN Layer

◆RNN Layer = a sequence of RNN cells (Sequence Length: $L$)

RNN Layer

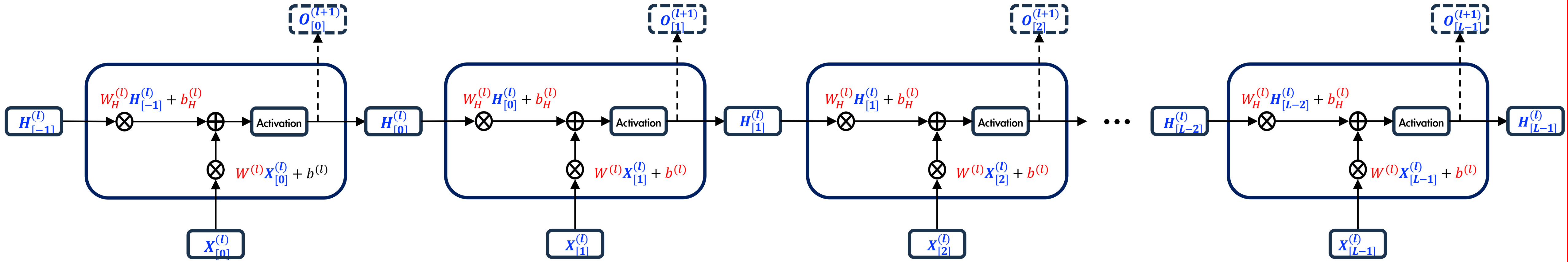



```python
rnn_cell = nn.RNNCell(input_size=3, hidden_size=4)

for name, parameter in rnn_cell.named_parameters():
  print(name, parameter.shape)

# >>> weight_ih torch.Size([4, 3]): W^(l)
# >>> weight_hh torch.Size([4, 4]): W_H^(l)
# >>> bias_ih torch.Size([4]): b^(l)
# >>> bias_hh torch.Size([4]): b_H^(l)
```

# RNN Layer

◈RNN Layer = a sequence of RNN cells (Sequence Length: $L$)

RNN Layer



```python
rnn_cell = nn.RNNCell(input_size=3, hidden_size=4)
# sequence size (L): 6, input size : 3
input = torch.randn(6, 3)

# hidden size: 4
hx = torch.randn(4)
output = []
for i in range(6):  # sequence size (N)
  hx = rnn_cell(input=input[i], hx=hx)
  output.append(hx)

for idx, out in enumerate(output):
  print(idx, output)
```

```
0 tensor([ 0.3005,  0.6700, -0.3974,  0.6741], grad_fn=<SqueezeBackward1>)
1 tensor([ 0.7542,  0.9379, -0.0069,  0.5461], grad_fn=<SqueezeBackward1>)
2 tensor([ 0.7395, -0.3393,  0.7367,  0.8133], grad_fn=<SqueezeBackward1>)
3 tensor([ 0.5698,  0.7913, -0.6019,  0.3906], grad_fn=<SqueezeBackward1>)
4 tensor([0.3682, 0.8590, 0.1441, 0.9075], grad_fn=<SqueezeBackward1>)
5 tensor([0.8370, 0.3408, 0.2225, 0.6136], grad_fn=<SqueezeBackward1>)
```

Sequence length ($L$) is not determined by the RNN cell construction

It will be determined by input data!!!

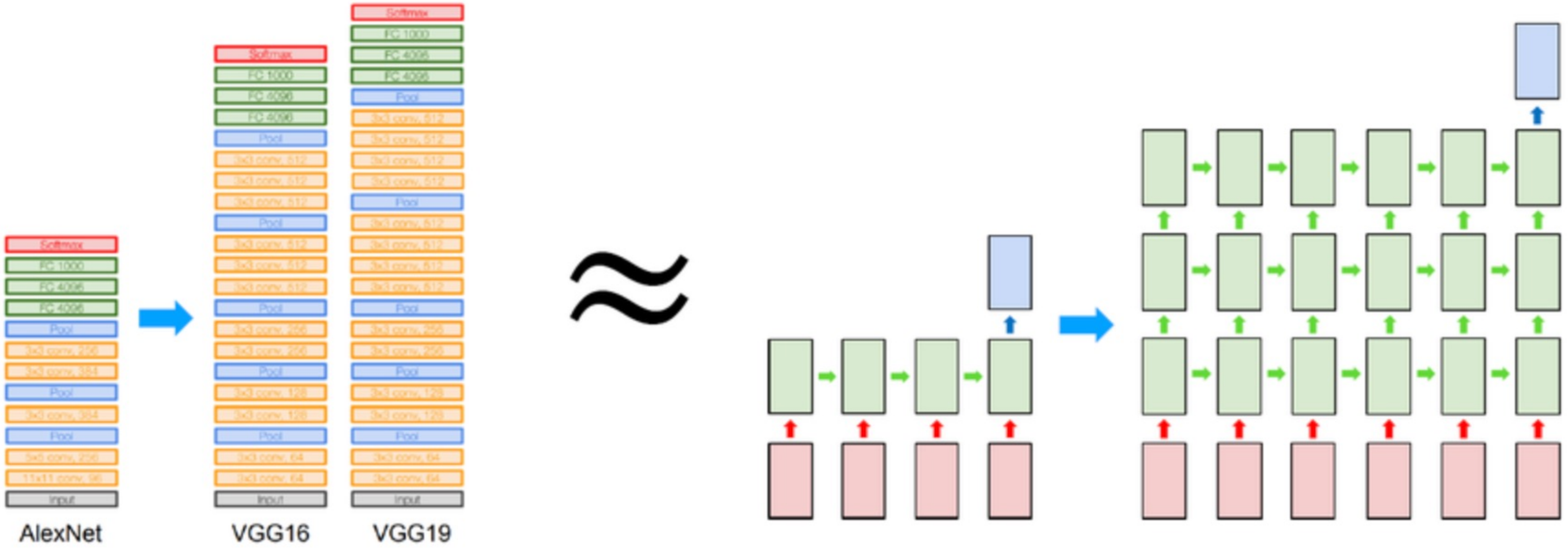
The RNN cell is rolled out based on the sequence length determined by the input data

# RNN Cells & RNN Layers → RNN

# RNN Stacking

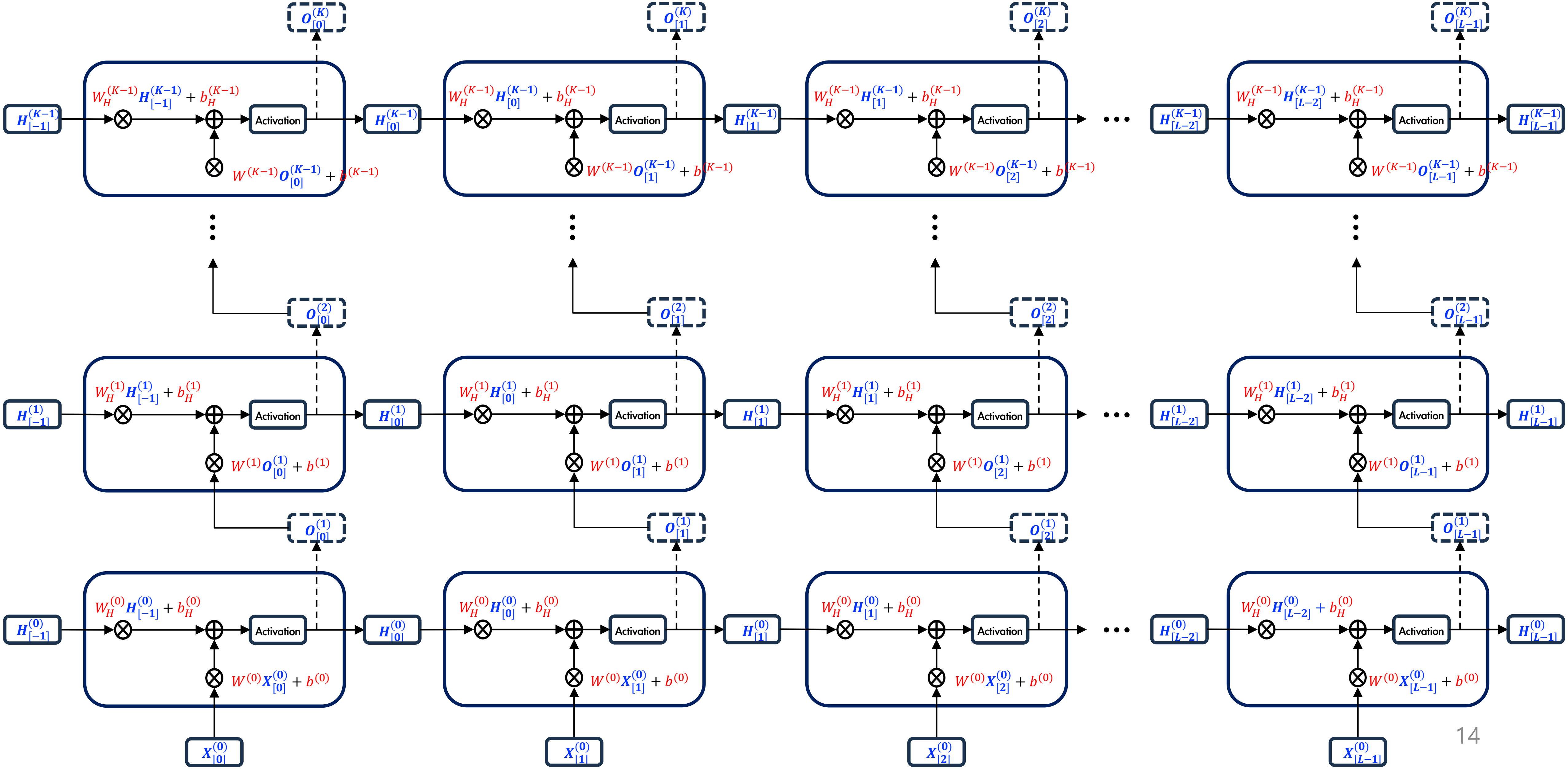
## ◈RNN with Many Layers

While it is not theoretically clear what is the additional power gained by the deeper architecture, it was observed empirically that deep RNNs work better than shallower ones on some tasks.




http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture09.pdf

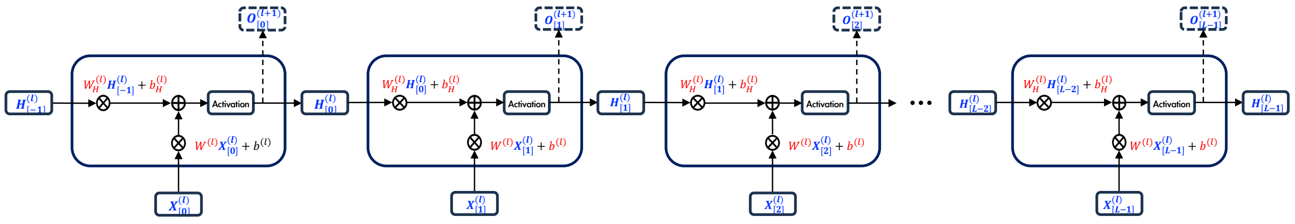
# ◈RNN = a multi-layered sequence of RNN cells (Num Layers: $K$, Sequence Length: $L$)



# RNN

## ◈RNN with One Layer

Num Layers: 1



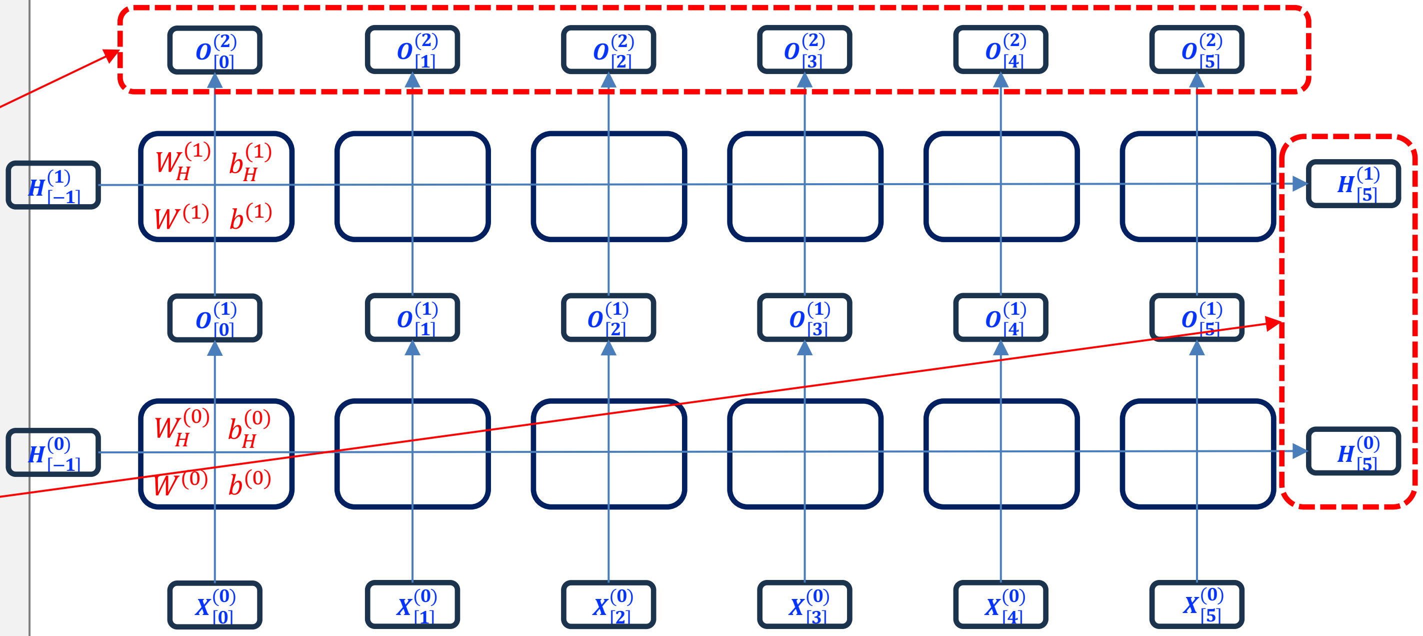
```
# sequence size (L): 6, batch size (N): ..., input size (F): 3
```

```
rnn1 = nn.RNN(
  input_size=3, hidden_size=4, num_layers=1
  nonlinearlity='tanh', bias=True,
  batch_first=False, dropout=0.0,
  bidirectional=False, device=None
)
for name, parameter in rnn1.named_parameters():
  print(name, parameter.shape)
# >>> weight_ih_l0 torch.Size([4, 3])
# >>> weight_hh_l0 torch.Size([4, 4])
# >>> bias_ih_l0 torch.Size([4])
# >>> bias_hh_l0 torch.Size([4])

input = torch.randn(6, 3)
output, hidden_state = rnn1(input)
for idx, out in enumerate(output):
  print(idx, out)
0 tensor([0.1388, 0.3009, 0.5137, 0.5397], grad_fn=<UnbindBackward0>)
1 tensor([0.0844, 0.0305, 0.1594, 0.7344], grad_fn=<UnbindBackward0>)
2 tensor([ 0.0070, -0.0689,  0.2532,  0.6872], grad_fn=<UnbindBackward0>)
3 tensor([-0.0699, -0.0744,  0.1923,  0.6173], grad_fn=<UnbindBackward0>)
4 tensor([-0.0539,  0.6625,  0.7565,  0.0793], grad_fn=<UnbindBackward0>)
5 tensor([0.3101, 0.5119, 0.4400, 0.6907], grad_fn=<UnbindBackward0>)

for idx, hidden in enumerate(hidden_state):
  print(idx, hidden)
0 tensor([0.3101, 0.5119, 0.4400, 0.6907], grad_fn=<UnbindBackward0>)
```

# RNN

## ◈RNN with Two Layers

```
rnn2 = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2
)

for name, parameter in rnn2.named_parameters():
  print(name, parameter.shape)

# sequence size (L): 6, input size (F): 3
input = torch.randn(6, 3)

output, hidden_state = rnn2(input)

for idx, out in enumerate(output):
  print(idx, out)

for idx, hidden in enumerate(hidden_state):
  print(idx, hidden)
```



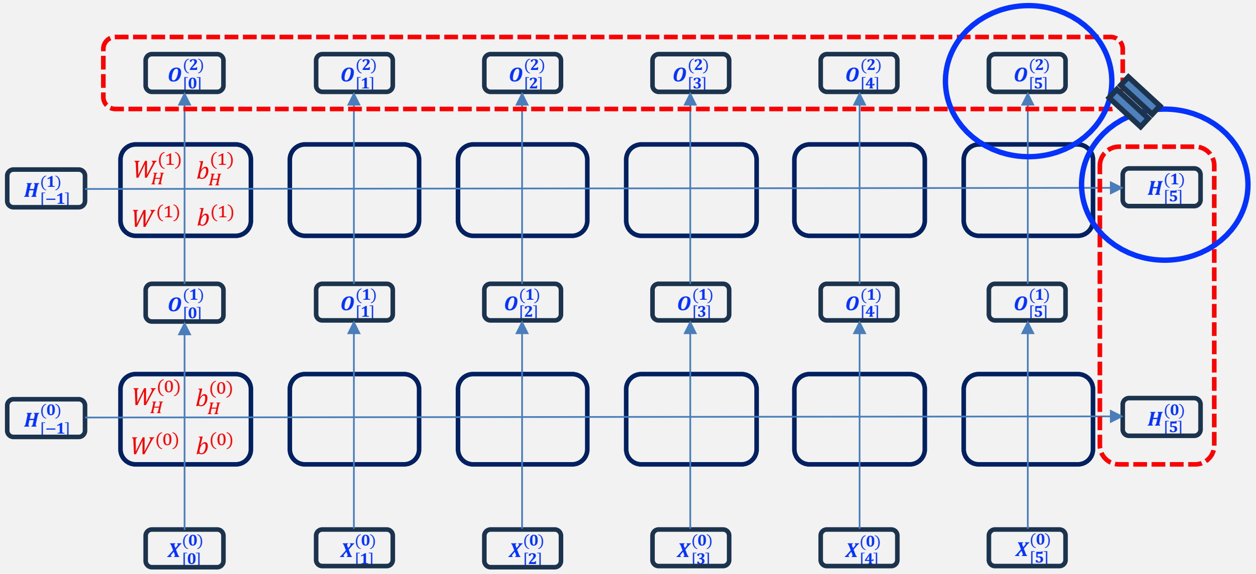

RNN knows how to handle the sequence of data, and rolls out the RNN cell automatically according to the sequence.

# RNN

## ◈RNN with Two Layers

```
rnn = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2
)

for name, parameter in rnn.named_parameters():
  print(name, parameter.shape)

# sequence size (L): 6, input size (F): 3
input = torch.randn(6, 3)

output, hidden_state = rnn(input)

for idx, out in enumerate(output):
  print(idx, out)    # shape: torch.Size([4])

for idx, hidden in enumerate(hidden_state):
  print(idx, hidden) # shape: torch.Size([4])
```

```
0 tensor([0.1444, 0.5225, 0.0172, 0.3065], grad_fn=<UnbindBackward0>)
1 tensor([-0.3921,  0.4679, -0.0863,  0.2366], grad_fn=<UnbindBackward0>)
2 tensor([-0.4235,  0.2706,  0.2536,  0.1714], grad_fn=<UnbindBackward0>)
3 tensor([0.3311, 0.5493, 0.0872, 0.3736], grad_fn=<UnbindBackward0>)
4 tensor([-0.7800,  0.0835,  0.0847, -0.1501], grad_fn=<UnbindBackward0>)
5 tensor([-0.1697, -0.0223,  0.4692,  0.1598], grad_fn=<UnbindBackward0>)

0 tensor([ 0.2373,  0.6753, -0.6874, -0.7837], grad_fn=<UnbindBackward0>)
1 tensor([-0.1697, -0.0223,  0.4692,  0.1598], grad_fn=<UnbindBackward0>)
```

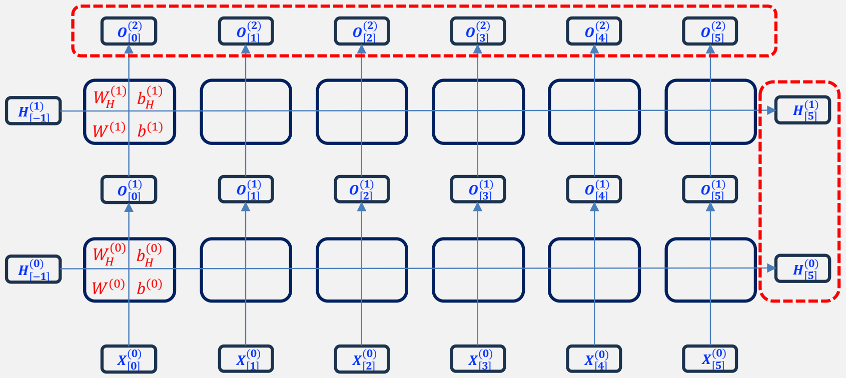


# RNN

## ◈Batch Inputs to RNN with Two Layers (Option 1) [Sequence, Batch, Input] $L \times N \times F$

```python
rnn = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2
)

# sequence size (L): 6, batch size (N): 10, input size (F): 3
batch_input = torch.randn(6, 10, 3)

batch_output, batch_hidden_state = rnn(batch_input)

print(batch_output.shape)   # >>> torch.Size([6, 10, 4])
for idx, out in enumerate(batch_output):
  print(idx, out.shape)     # >>> idx torch.Size([10, 4])

print()

print(batch_hidden_state.shape) # >>> torch.Size([2, 10, 4])
for idx, hidden in enumerate(batch_hidden_state):
  print(idx, hidden.shape)  # >>> idx torch.Size([10, 4])
```

```
torch.Size([6, 10, 4])
0 torch.Size([10, 4])
1 torch.Size([10, 4])
2 torch.Size([10, 4])
3 torch.Size([10, 4])
4 torch.Size([10, 4])
5 torch.Size([10, 4])

torch.Size([2, 10, 4])
0 torch.Size([10, 4])
1 torch.Size([10, 4])
```

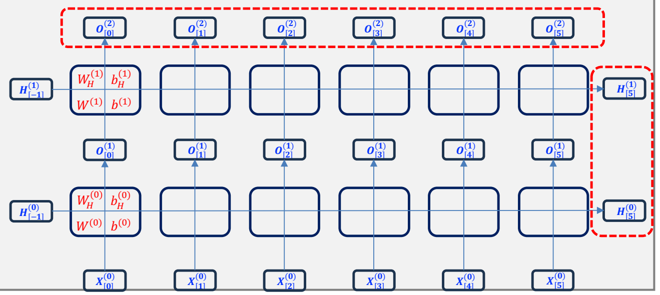


# RNN

## ◈Batch Inputs to RNN with Two Layers (Option 2) [Batch, Sequence, Input] $N \times L \times F$

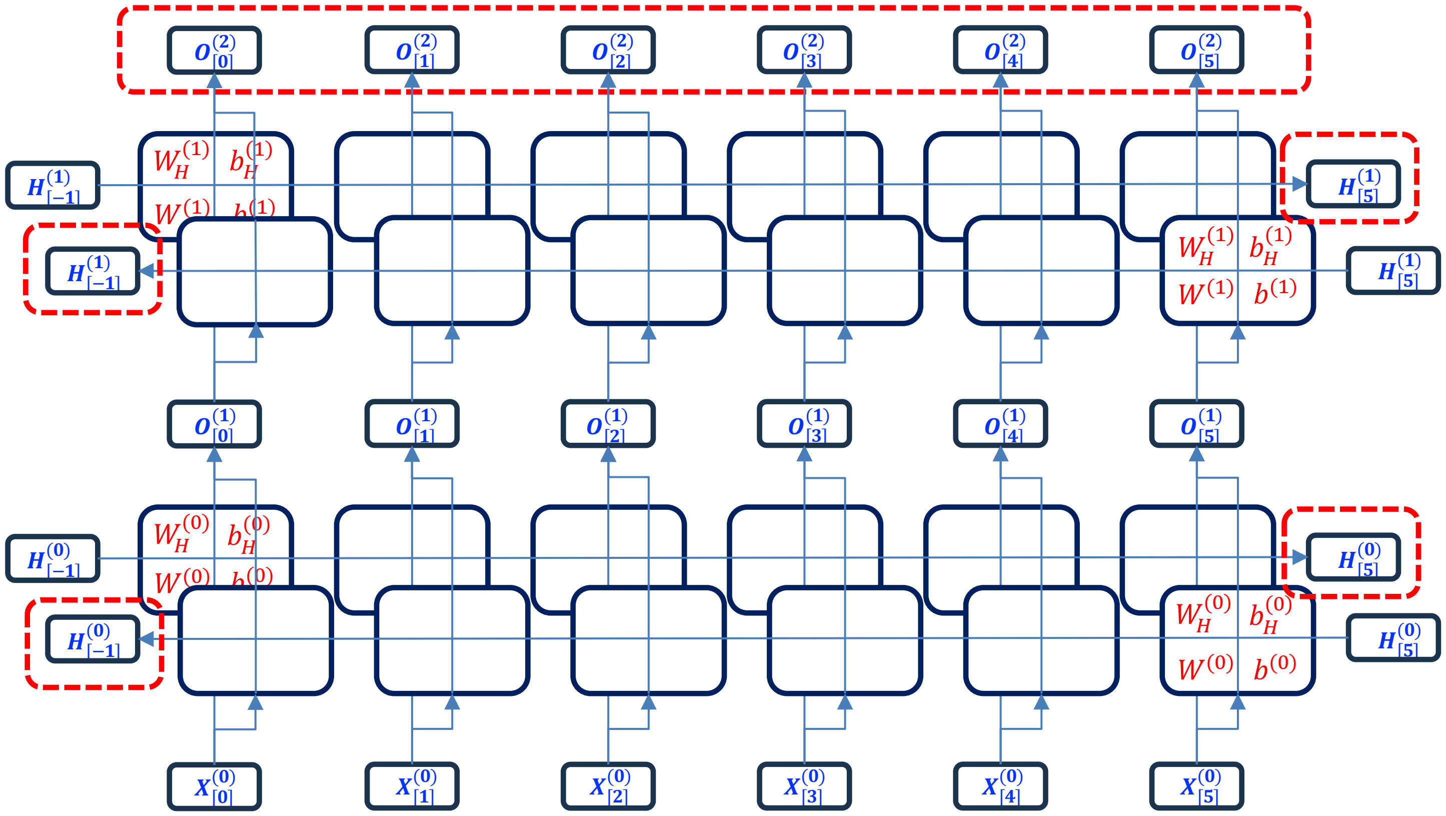
```
rnn = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2, batch_first=True
)

# batch size (N): 10, sequence size (L): 6, input size (F): 3
batch_input = torch.randn(10, 6, 3)

batch_output, batch_hidden_state = rnn(batch_input)

print(batch_output.shape)   # >>> torch.Size([10, 6, 4])
for idx, out in enumerate(batch_output):
  print(idx, out.shape)     # >>> idx torch.Size([6, 4])

print()

print(batch_hidden_state.shape) # >>> torch.Size([2, 10, 4])
for idx, hidden in enumerate(batch_hidden_state):
  print(idx, hidden.shape)  # >>> idx torch.Size([10, 4])
```

```
torch.Size([10, 6, 4])
0 torch.Size([6, 4])
1 torch.Size([6, 4])
2 torch.Size([6, 4])
3 torch.Size([6, 4])
4 torch.Size([6, 4])
5 torch.Size([6, 4])
6 torch.Size([6, 4])
7 torch.Size([6, 4])
8 torch.Size([6, 4])
9 torch.Size([6, 4])

torch.Size([2, 10, 4])
0 torch.Size([10, 4])
1 torch.Size([10, 4])
```



# Bidirectional RNN

## ◈Bidirectional RNN with Two Layers

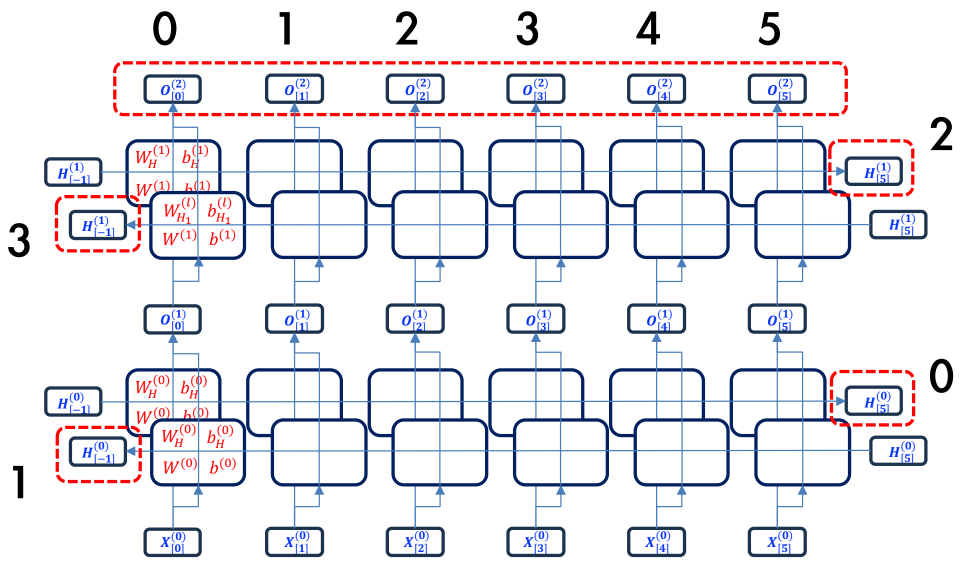



```python
rnn = nn.RNN(
  input_size=3, hidden_size=4,
  num_layers=2,
  bidirectional=True
)
for name, parameter in rnn.named_parameters():
  print(name, parameter.shape)

# >>> weight_ih_l0 torch.Size([4, 3])
# >>> weight_hh_l0 torch.Size([4, 4])
# >>> bias_ih_l0 torch.Size([4])
# >>> bias_hh_l0 torch.Size([4])
# >>> weight_ih_l0_reverse torch.Size([4, 3])
# >>> weight_hh_l0_reverse torch.Size([4, 4])
# >>> bias_ih_l0_reverse torch.Size([4])
# >>> bias_hh_l0_reverse torch.Size([4])
# >>> weight_ih_l1 torch.Size([4, 8])
# >>> weight_hh_l1 torch.Size([4, 4])
# >>> bias_ih_l1 torch.Size([4])
# >>> bias_hh_l1 torch.Size([4])
# >>> weight_ih_l1_reverse torch.Size([4, 8])
# >>> weight_hh_l1_reverse torch.Size([4, 4])
# >>> bias_ih_l1_reverse torch.Size([4])
# >>> bias_hh_l1_reverse torch.Size([4])
```

# Bidirectional RNN

## ◈Bidirectional RNN with Two Layers

```
rnn = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2,
  bidirectional=True
)

# sequence size (L): 6, input size (F): 3
input = torch.randn(6, 3)

output, hidden_state = rnn(input)

for idx, out in enumerate(output):
  print(idx, out)    # shape: torch.Size([4])

for idx, hidden in enumerate(hidden_state):
  print(idx, hidden) # shape: torch.Size([4])
```




```
0 tensor([-0.2724, -0.7689, -0.4468, -0.1966,  0.1103,  0.3610,  0.6847,  0.0536],
grad_fn=<UnbindBackward0>)
1 tensor([ 0.4187, -0.5284, -0.0518, -0.7682,  0.6282,  0.5067,  0.6140, -0.2717],
grad_fn=<UnbindBackward0>)
2 tensor([ 0.0062,  0.4153,  0.9006,  0.1024, -0.0874,  0.7522,  0.8035,  0.6523],
grad_fn=<UnbindBackward0>)
3 tensor([ 0.0162,  0.2409,  0.5205, -0.7116,  0.1083,  0.2039,  0.6002,  0.6289],
grad_fn=<UnbindBackward0>)
4 tensor([ 0.3742,  0.1368,  0.6304, -0.6833, -0.0602,  0.0595,  0.7382,  0.5925],
grad_fn=<UnbindBackward0>)
5 tensor([ 0.2831,  0.4516,  0.6628, -0.4940, -0.6240, -0.0086,  0.6180,  0.5068],
grad_fn=<UnbindBackward0>)

0 tensor([ 0.4010, -0.9737, -0.5143,  0.5953], grad_fn=<UnbindBackward0>)
1 tensor([-0.2416,  0.3461,  0.4018, -0.5601], grad_fn=<UnbindBackward0>)
2 tensor([ 0.2831,  0.4516,  0.6628, -0.4940], grad_fn=<UnbindBackward0>)
3 tensor([0.1103, 0.3610, 0.6847, 0.0536], grad_fn=<UnbindBackward0>)
```

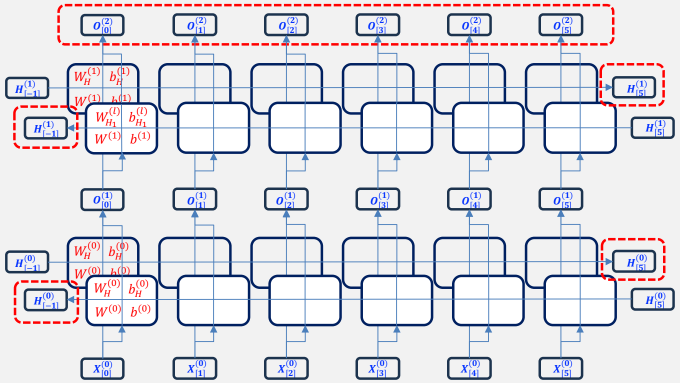
# Bidirectional RNN

## ◇Batch Inputs to Bidirectional RNN with Two Layers (Option 1) [Sequence, Batch, Input]

$L \times N \times F$

```python
rnn = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2, bidirectional=True
)

# sequence size (L): 6, batch size (N): 10, input size (F): 3
batch_input = torch.randn(6, 10, 3)

batch_output, batch_hidden_state = rnn(batch_input)

print(batch_output.shape)   # >>> torch.Size([6, 10, 4])
for idx, out in enumerate(batch_output):
  print(idx, out.shape)     # >>> idx torch.Size([10, 4])

print()

print(batch_hidden_state.shape) # >>> torch.Size([2, 10, 4])
for idx, hidden in enumerate(batch_hidden_state):
  print(idx, hidden.shape)  # >>> idx torch.Size([10, 4])
```

```
torch.Size([6, 10, 8])
0 torch.Size([10, 8])
1 torch.Size([10, 8])
2 torch.Size([10, 8])
3 torch.Size([10, 8])
4 torch.Size([10, 8])
5 torch.Size([10, 8])

torch.Size([4, 10, 4])
0 torch.Size([10, 4])
1 torch.Size([10, 4])
2 torch.Size([10, 4])
3 torch.Size([10, 4])
```



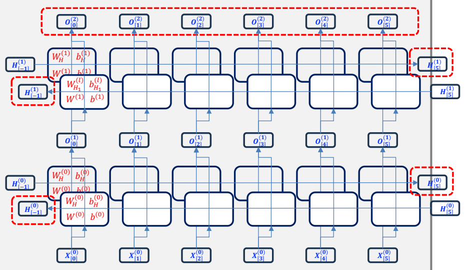
# Bidirectional RNN

## ◈Batch Inputs to Bidirectional RNN with Two Layers (Option 2) [Batch, Sequence, Input]

$N \times L \times F$

```python
rnn = nn.RNN(
  input_size=3, hidden_size=4, num_layers=2, batch_first=True,
  bidirectional=True
)

# batch size (N): 10, sequence size (L): 6, input size (F): 3
batch_input = torch.randn(10, 6, 3)

batch_output, batch_hidden_state = rnn(batch_input)

print(batch_output.shape)   # >>> torch.Size([10, 6, 4])
for idx, out in enumerate(batch_output):
  print(idx, out.shape)     # >>> idx torch.Size([6, 4])

print()

print(batch_hidden_state.shape) # >>> torch.Size([2, 10, 4])
for idx, hidden in enumerate(batch_hidden_state):
  print(idx, hidden.shape)  # >>> idx torch.Size([10, 4])
```

```
torch.Size([10, 6, 8])
0 torch.Size([6, 8])
1 torch.Size([6, 8])
2 torch.Size([6, 8])
3 torch.Size([6, 8])
4 torch.Size([6, 8])
5 torch.Size([6, 8])
6 torch.Size([6, 8])
7 torch.Size([6, 8])
8 torch.Size([6, 8])
9 torch.Size([6, 8])

torch.Size([4, 10, 4])
0 torch.Size([10, 4])
1 torch.Size([10, 4])
2 torch.Size([10, 4])
3 torch.Size([10, 4])
```



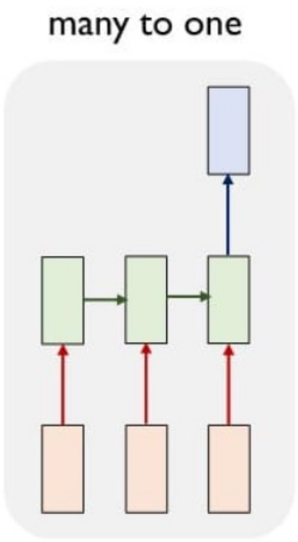
# RNN Operation Mode

# Operation Mode for an RNN

## ◈Modes of operation for an RNN

– Many-to-One (1/2)

- a sequence of inputs generates a single output
- This is often used in sentiment analysis, where a sequence of words is classified as expressing a positive or negative sentiment
- Time series prediction such as "bikes rental count prediction" or "bitcoin price prediction"




- An example dataset

Input dim

Sequence

| | Open | High | Low | Volume | Close |
|---|---|---|---|---|---|
| 0 | 828.659973 | 833.450012 | 828.349976 | 1247700 | 831.659973 |
| 1 | 823.020020 | 828.070007 | 821.655029 | 1597800 | 828.070007 |
| 2 | 819.929993 | 824.400024 | 818.979980 | 1281700 | 824.159973 |
| 3 | 819.359985 | 823.000000 | 818.469971 | 1304000 | 818.979980 |
| 4 | 819.000000 | 823.000000 | 816.000000 | 1053600 | 820.450012 |
| 5 | 816.000000 | 820.958984 | 815.489990 | 1198100 | 819.239990 |
| 6 | 811.700012 | 815.250000 | 809.780029 | 1129100 | 813.669983 |
| 7 | 809.510010 | 810.659973 | 804.539978 | 989700 | 809.559998 |
| 8 | 807.000000 | 811.840027 | 803.190002 | 1155300 | 808.380005 |
| 9 | 803.989990 | 810.500000 | 801.780029 | 1235200 | 806.969971 |

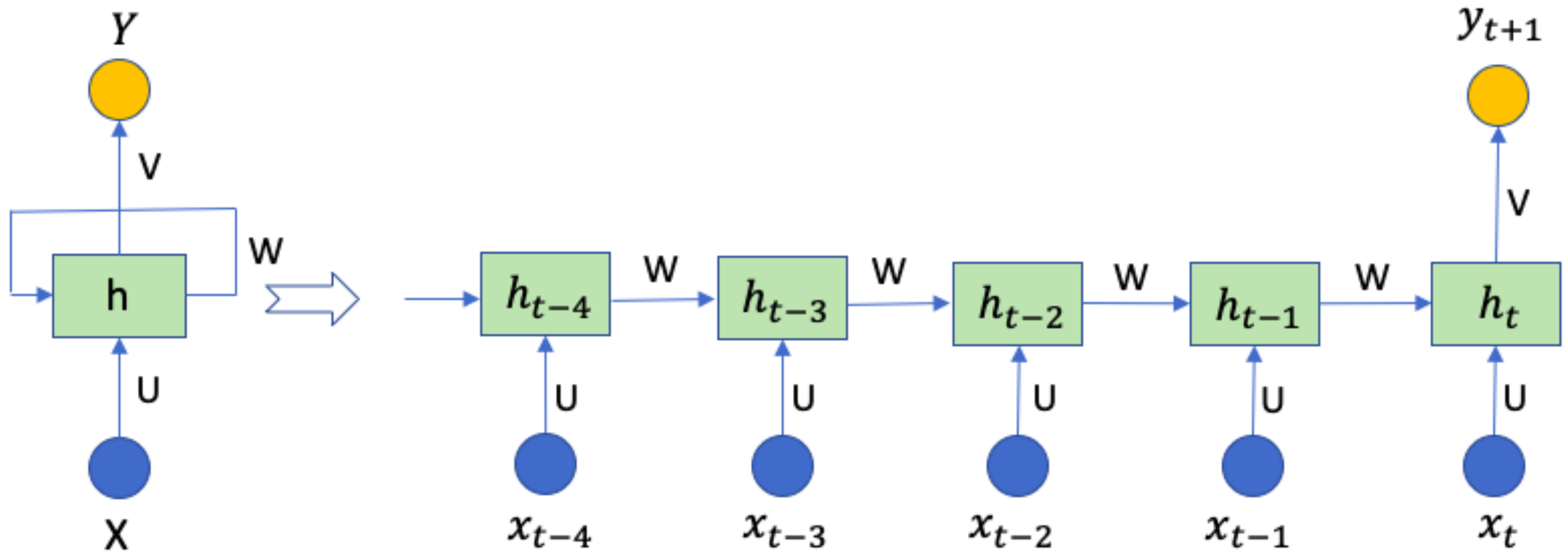
Output dim

# Operation Mode for an RNN

## ◈Modes of operation for an RNN

– Many-to-One (2/2)

- Model Construction and Usage

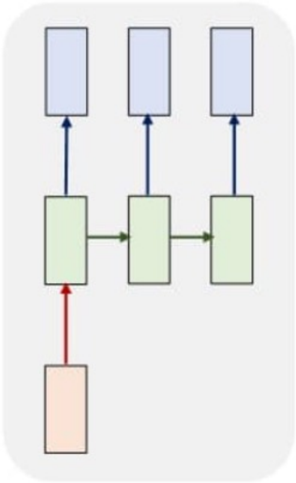


# Operation Mode for an RNN

## ◈Modes of operation for an RNN

– One-to-Many

- This mode is used in applications where one input may correspond to a sequence of outputs, such as image captioning, where an image input generates a sequence of words as an output




- An example dataset

```
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43]
[[2, 3], [5, 6], [8, 9], [11, 12], [14, 15], [17, 18], [20, 21], [23, 24], [26, 27], [29, 30], [32, 33], [35, 36], [38, 39], [41, 42], [44, 45]]
```

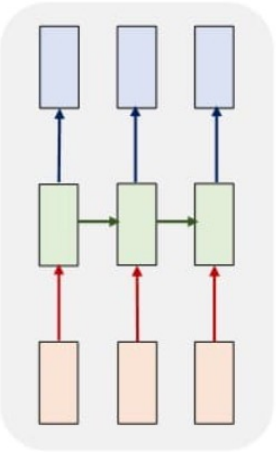
# Operation Mode for an RNN

## ◈Modes of operation for an RNN

– Many-to-Many Synchronous (Sequence-to-Sequence or Seq2Seq Model) (1/2)

- a sequence of inputs is mapped to a sequence of outputs, used for tasks such as video classification, where every frame of the video is labeled
- An example dataset

many to many



Raw dataset

| Feature 1 | Feature 2 | Target (Y) |
|---|---|---|
| 1 | 2 | 10 |
| 3 | 4 | 20 |
| 5 | 6 | 30 |
| 7 | 8 | 40 |
| 9 | 10 | 50 |
| 11 | 12 | 60 |
| 13 | 14 | 70 |
| 15 | 16 | 80 |
| 17 | 18 | 90 |
| 19 | 20 | 100 |
| 21 | 22 | 110 |
| 23 | 24 | 120 |
| 25 | 26 | 130 |
| 27 | 28 | 140 |
| 29 | 30 | 150 |
| ... | ... | ... |

Sequence data

| Feature 1 | Feature 2 | Target (Y) |
|---|---|---|
| 1 | 2 | 50 |
| 3 | 4 | 60 |
| 5 | 6 | 70 |
| 7 | 8 | 80 |
| 9 | 10 | 90 |

| Feature 1 | Feature 2 | Target (Y) |
|---|---|---|
| 3 | 4 | 60 |
| 5 | 6 | 70 |
| 7 | 8 | 80 |
| 9 | 10 | 90 |
| 11 | 12 | 100 |

| Feature 1 | Feature 2 | Target (Y) |
|---|---|---|
| 5 | 6 | 70 |
| 7 | 8 | 80 |
| 9 | 10 | 90 |
| 11 | 12 | 100 |
| 13 | 14 | 110 |

...

Sequential dataset

| Feature 1 | Feature 2 | Target (Y) |
|---|---|---|
| 5 | 6 | 70 |
| 7 | 8 | 80 |
| 9 | 10 | 90 |
| 11 | 12 | 100 |
| 13 | 14 | 110 |

...

Source: Author

[Batch, Sequence, Input]

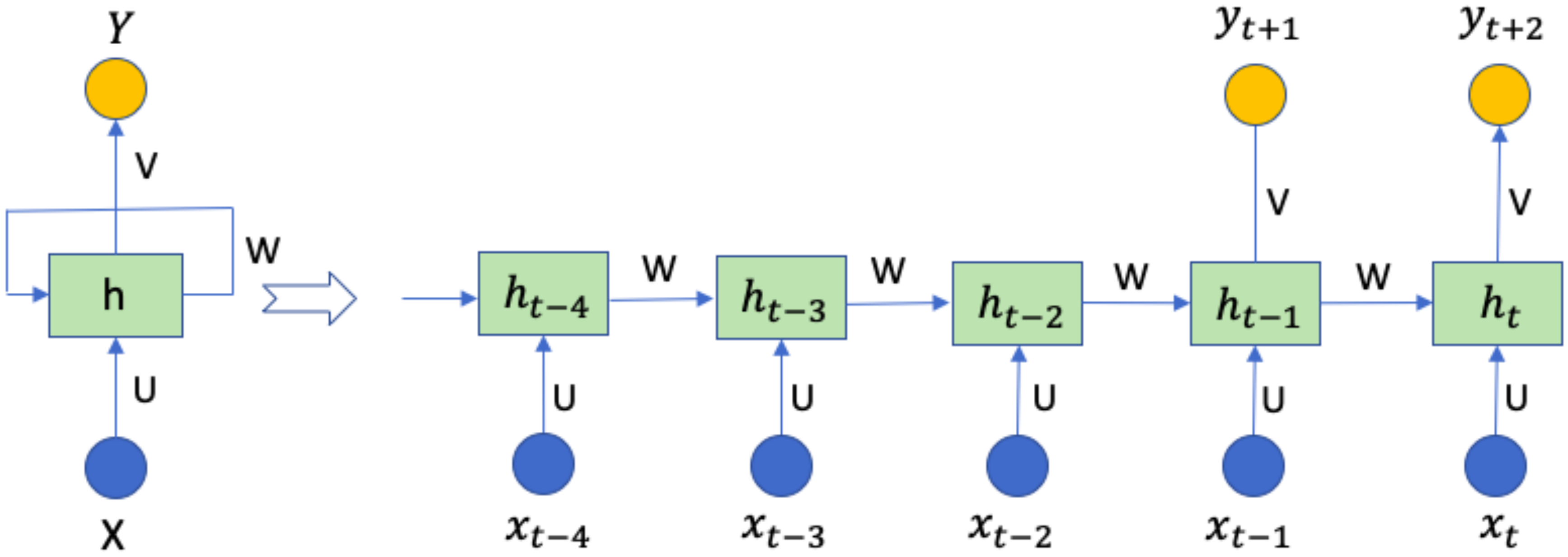
$N \times L \times F = (3, 5, 2)$

# Operation Mode for an RNN

## ◈Modes of operation for an RNN

– Many-to-Many Synchronous (or Sequence-to-Sequence or Seq2Seq Model) (2/2)

  - Model Construction and Usage

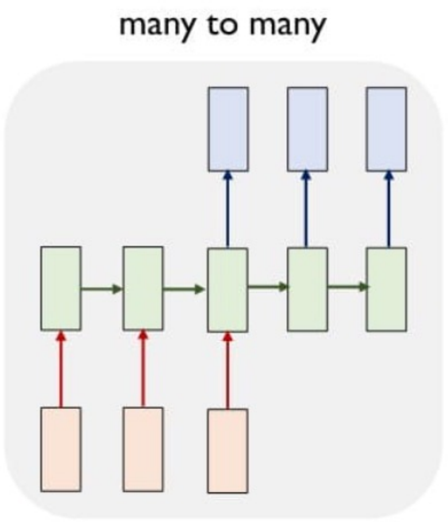


# Operation Mode for an RNN

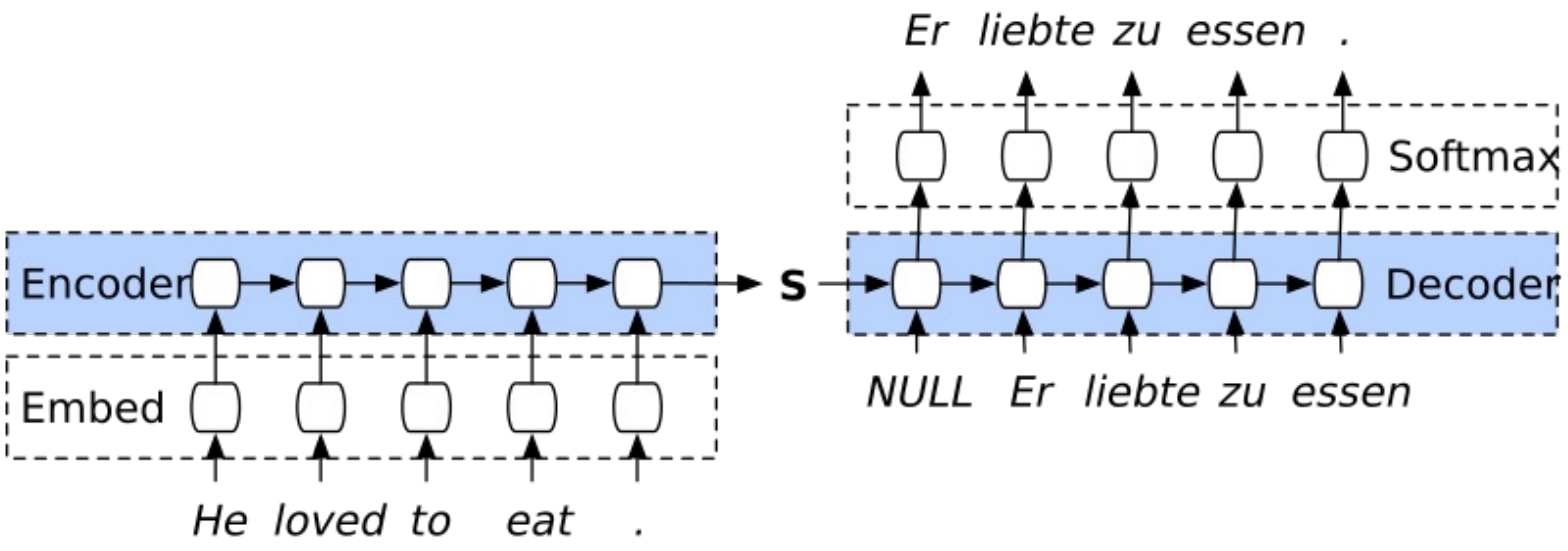
## ❖Modes of operation for an RNN

– Many-to-Many Asynchronous (or Encoder-Decoder Model)

- It is used for tasks like machine translation, where an input sentence in one language is translated into a sentence in another language with potentially different word order




- An example



# RNN Best Practice

## - Hourly Bikes Sharing -

# RNN with PyTorch

## ◈Regression Trainer - init

```
from datetime import datetime
import torch
from torch import nn

from _01_code._06_fcn_best_practice.c_trainer import EarlyStopping
from _01_code._99_common_utils.utils import strfdelta

class CustomRegressionTrainer:
  def __init__(
    self, project_name, model, optimizer, train_data_loader, validation_data_loader, transforms,
    run_time_str, wandb, device, checkpoint_file_path
  ):
    self.project_name = project_name
    self.model = model
    self.optimizer = optimizer
    self.train_data_loader = train_data_loader
    self.validation_data_loader = validation_data_loader
    self.transforms = transforms
```

# RNN with PyTorch

## ◈Regression Trainer - init

```
class CustomRegressionTrainer:
  def __init__(
    self, project_name, model, optimizer, train_data_loader, validation_data_loader, transforms,
    run_time_str, wandb, device, checkpoint_file_path
  ):
    ...

    self.run_time_str = run_time_str
    self.wandb = wandb
    self.device = device
    self.checkpoint_file_path = checkpoint_file_path

    # Use a built-in loss function
    self.loss_fn = nn.MSELoss()
```

# RNN with PyTorch

## ◈Regression Trainer - do_train

```
class CustomRegressionTrainer:
  ...
  def do_train(self):
    self.model.train()  # Explained at 'Diverse Techniques' section

    loss_train = 0.0
    num_trains = 0

    for train_batch in self.train_data_loader:
      input_train, target_train = train_batch
      input_train = input_train.to(device=self.device)
      target_train = target_train.to(device=self.device)

      if self.transforms:
        input_train = self.transforms(input_train)

      output_train = self.model(input_train)
```

# RNN with PyTorch

## ◈Regression Trainer - do_train

```
class CustomRegressionTrainer:
  ...
  def do_train(self):
    ...
    for train_batch in self.train_data_loader:
      ...
      loss = self.loss_fn(output_train.squeeze(dim=-1), target_train)
      loss_train += loss.item()

      num_trains += 1

      self.optimizer.zero_grad()
      loss.backward()
      self.optimizer.step()

    train_loss = loss_train / num_trains

    return train_loss
```

# RNN with PyTorch

## ◈Regression Trainer - do_validation

```
class CustomRegressionTrainer:
  ...
  def do_validation(self):
    self.model.eval()   # Explained at 'Diverse Techniques' section

    loss_validation = 0.0
    num_validations = 0

    with torch.no_grad():
      for validation_batch in self.validation_data_loader:
        input_validation, target_validation = validation_batch
        input_validation = input_validation.to(device=self.device)
        target_validation = target_validation.to(device=self.device)

        if self.transforms:
          input_validation = self.transforms(input_validation)
```

# RNN with PyTorch

## ◈Regression Trainer - do_validation

```
class CustomRegressionTrainer:
  ...
  def do_validation(self):
    ...
    with torch.no_grad():
      for validation_batch in self.validation_data_loader:
        ...

        output_validation = self.model(input_validation)
        loss_validation += self.loss_fn(output_validation.squeeze(dim=-1), target_validation).item()

        num_validations += 1

    validation_loss = loss_validation / num_validations

    return validation_loss
```

# RNN with PyTorch

## ◈Regression Trainer - train_loop

```
class CustomRegressionTrainer:
  ...
  def train_loop(self):
    early_stopping = EarlyStopping(
      patience=self.wandb.config.early_stop_patience, delta=self.wandb.config.early_stop_delta,
      project_name=self.project_name, checkpoint_file_path=self.checkpoint_file_path,
      run_time_str=self.run_time_str
    )
    n_epochs = self.wandb.config.epochs
    training_start_time = datetime.now()

    for epoch in range(1, n_epochs + 1):
      train_loss = self.do_train()

      if epoch == 1 or epoch % self.wandb.config.validation_intervals == 0:
        validation_loss = self.do_validation()
        elapsed_time = datetime.now() - training_start_time
        epoch_per_second = 1000 * epoch / elapsed_time.microseconds
```

# RNN with PyTorch

## ◈Regression Trainer - train_loop

```
class CustomRegressionTrainer:
  ...
  def train_loop(self):
    ...
    for epoch in range(1, n_epochs + 1):
      ...
      if epoch == 1 or epoch % self.wandb.config.validation_intervals == 0:
        ...
        message, early_stop = early_stopping.check_and_save(validation_loss, self.model)

        print(
          f"[Epoch {epoch:>3}] ”
          f"T_loss: {train_loss:6.4f}, ”
          f"V_loss: {validation_loss:6.4f}, ”
          f"{message} | ”
          f"T_time: {strfdelta(elapsed_time, '%H:%M:%S')}, ”
          f"T_speed: {epoch_per_second:4.3f}”
        )
```

# RNN with PyTorch

## ◈Regression Trainer - train_loop

```
class CustomRegressionTrainer:
  ...
  def train_loop(self):
    ...
    for epoch in range(1, n_epochs + 1):
      ...
      if epoch == 1 or epoch % self.wandb.config.validation_intervals == 0:
        ...
        self.wandb.log({
          "Epoch": epoch, "Training loss": train_loss, "Validation loss": validation_loss,
          "Training speed (epochs/sec.)": epoch_per_second,
        })

        if early_stop:
          break

    elapsed_time = datetime.now() - training_start_time
    print(f"Final training time: {strfdelta(elapsed_time, '%H:%M:%S')}")
```

# RNN with PyTorch

## ◈RNN with Bikes Dataset

```
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, random_split
from datetime import datetime
import os
import wandb
from pathlib import Path

BASE_PATH = str(Path(__file__).resolve().parent.parent.parent) # BASE_PATH: /Users/yhhan/git/link_dl
import sys
sys.path.append(BASE_PATH)

CURRENT_FILE_PATH = os.path.dirname(os.path.abspath(__file__))
CHECKPOINT_FILE_PATH = os.path.join(CURRENT_FILE_PATH, "checkpoints")
if not os.path.isdir(CHECKPOINT_FILE_PATH):
  os.makedirs(os.path.join(CURRENT_FILE_PATH, "checkpoints"))

from _01_code._03_real_world_data_to_tensors.o_hourly_bikes_sharing_dataset_dataloader import
get_hourly_bikes_data, HourlyBikesDataset
from _01_code._10_rnn.g_rnn_trainer import RegressionTrainer
from _01_code._10_rnn.f_arg_parser import get_parser
```

link_dl/_01_code/_10_rnn/h_bikes_train_rnn.py

# RNN with PyTorch

## ◈RNN with Bikes Dataset

```
def get_train_bikes_data():
  X_train, X_validation, X_test, y_train, y_validation, y_test = get_hourly_bikes_data(
      sequence_size=24, validation_size=96, test_size=24, y_normalizer=100
  )

  train_hourly_bikes_dataset = HourlyBikesDataset(X=X_train, y=y_train)
  validation_hourly_bikes_dataset = HourlyBikesDataset(X=X_validation, y=y_validation)

  train_data_loader = DataLoader(
    dataset=train_hourly_bikes_dataset, batch_size=wandb.config.batch_size, shuffle=True
  )

  validation_data_loader = DataLoader(
    dataset=validation_hourly_bikes_dataset, batch_size=wandb.config.batch_size, shuffle=True
  )

  return train_data_loader, validation_data_loader
```

# RNN with PyTorch

## ◈RNN with Bikes Dataset

```
def get_model():
  class MyModel(nn.Module):
    def __init__(self, n_input, n_output):
      super().__init__()

      self.rnn = nn.RNN(
        input_size=n_input, hidden_size=128, num_layers=2, batch_first=True
      )
      self.fcn = nn.Linear(in_features=128, out_features=n_output)

    def forward(self, x):
      x, hidden = self.rnn(x)
      x = x[:, -1, :]  # x.shape: [32, 128]
      x = self.fcn(x)
      return x

  my_model = MyModel(n_input=18, n_output=1)
  return my_model
```

# RNN with PyTorch

## ◈RNN with Bikes Dataset

```
def main(args):
  run_time_str = datetime.now().astimezone().strftime('%Y-%m-%d_%H-%M-%S’)
  config = {
    'epochs': args.epochs, 'batch_size': args.batch_size,
    'validation_intervals': args.validation_intervals,
    'learning_rate': args.learning_rate,
    'early_stop_patience': args.early_stop_patience
    'early_stop_delta': args.early_stop_delta,
  }
  project_name = "rnn_bikes”
  wandb.init(
    mode="online" if args.wandb else "disabled",
    project=project_name,    notes="bikes experiment with rnn",
    tags=["rnn", "bikes"],   name=run_time_str,
    config=config
  )
  print(args)
  print(wandb.config)
```

# RNN with PyTorch

## ◈RNN with Bikes Dataset

```
def main(args):
  ...
  train_data_loader, validation_data_loader = get_train_bikes_data()
  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
  print(f"Training on device {device}.")
  model = get_model()
  model.to(device)
  wandb.watch(model)

  optimizer = optim.Adam(model.parameters(), lr=wandb.config.learning_rate)

  classification_trainer = CustomRegressionTrainer(
    project_name, model, optimizer, train_data_loader, validation_data_loader, None,
    run_time_str, wandb, device, CHECKPOINT_FILE_PATH
  )

  classification_trainer.train_loop()
  wandb.finish()
```

# RNN with PyTorch

## ◈RNN with Bikes Dataset

```
if __name__ == "__main__":
  parser = get_parser()
  args = parser.parse_args()
  main(args)
  # python _01_code/_10_rnn/h_bikes_train_rnn.py -v 100
```

# RNN Test with PyTorch

## ◈RNN Test with Test Bikes Dataset

```
def test_main(test_model):
  _, _, X_test, _, _, y_test = get_hourly_bikes_data(
    sequence_size=24, validation_size=96, test_size=24, y_normalizer=100
  )

  test_hourly_bikes_dataset = HourlyBikesDataset(X=X_test, y=y_test)

  test_data_loader = DataLoader(
    dataset=test_hourly_bikes_dataset, batch_size=len(test_hourly_bikes_dataset)
  )

  test_model.eval()

  y_normalizer = 100

  print("[TEST DATA]")
```

# RNN Test with PyTorch

## ◈RNN Test with Test Bikes Dataset

```
def test_main(test_model):
  ...
  with torch.no_grad():
    for test_batch in test_data_loader:
      input_test, target_test = test_batch

      output_test = test_model(input_test)

    for idx, (output, target) in enumerate(zip(output_test, target_test)):
      output = round(output.item() * y_normalizer)
      target = target.item() * y_normalizer

      print("{0:2}: {1:6,.2f} <--> {2:6,.2f} (Loss: {3:>13,.2f})".format(
        idx, output, target, abs(output - target)
      ))
```

# RNN Test with PyTorch

## ◈RNN Test with Test Bikes Dataset

```
def predict_all(test_model):
  y_normalizer = 100
  X_train, X_validation, X_test, y_train, y_validation, y_test = get_hourly_bikes_data(
      sequence_size=24, validation_size=96, test_size=24, y_normalizer=100]
  )
  train_hourly_bikes_dataset = HourlyBikesDataset(X=X_train, y=y_train)
  validation_hourly_bikes_dataset = HourlyBikesDataset(X=X_validation, y=y_validation)
  test_hourly_bikes_dataset = HourlyBikesDataset(X=X_test, y=y_test)

  dataset_list = [
    train_hourly_bikes_dataset, validation_hourly_bikes_dataset, test_hourly_bikes_dataset
  ]
  dataset_labels = [
    "train", "validation", "test”
  ]

  num = 0
  fig, axs = plt.subplots(3, 1, figsize=(6, 9))
```

# RNN Test with PyTorch

## ◈RNN Test with Test Bikes Dataset

```
def predict_all(test_model):
  for i in range(3):
    X = []
    TARGET_Y = []
    PREDICTION_Y = []
    for data in dataset_list[i]:
      input, target = data
      prediction = test_model(input.unsqueeze(0)).squeeze(-1).squeeze(-1)
      X.append(num)
      TARGET_Y.append(target.item() * y_normalizer)
      PREDICTION_Y.append(prediction.item() * y_normalizer)
      num += 1
    axs[i].plot(X, TARGET_Y, label='target’)
    axs[i].plot(X, PREDICTION_Y, label='prediction’)
    axs[i].set_title(dataset_labels[i])
    axs[i].legend()
  plt.tight_layout()
  plt.show()
```