KOREATECH
한국기술교육대학교

LSTM and Its Application

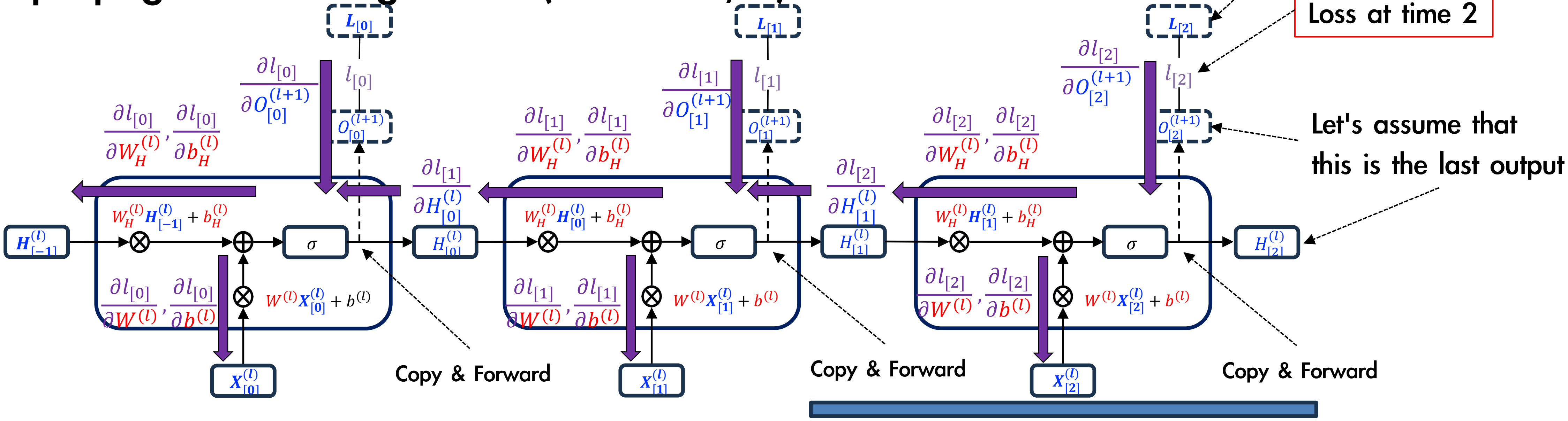
November 2023

http://link.koreatech.ac.kr

BPTT with RNN

RNN Backpropagation

◈Backpropagation Through Time (BPTT – 1/4)



$$\frac{\partial l_{[2]}}{\partial W_H^{(l)}} = \frac{\partial l_{[2]}}{\partial O_{[2]}^{(l+1)}} \cdot \frac{\partial O_{[2]}^{(l+1)}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z_{[2]}} \cdot \frac{\partial z_{[2]}}{\partial W_H^{(l)}}$$

$$\frac{\partial l_{[2]}}{\partial W^{(l)}} = \frac{\partial l_{[2]}}{\partial O_{[2]}^{(l+1)}} \cdot \frac{\partial O_{[2]}^{(l+1)}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z_{[2]}} \cdot \frac{\partial z_{[2]}}{\partial W^{(l)}}$$

where

$$z_{[2]} = W_H^{(l)} \boldsymbol{H}_{[1]}^{(l)} + b_H^{(l)} + W^{(l)} \boldsymbol{X}_{[2]}^{(l)} + b^{(l)}$$

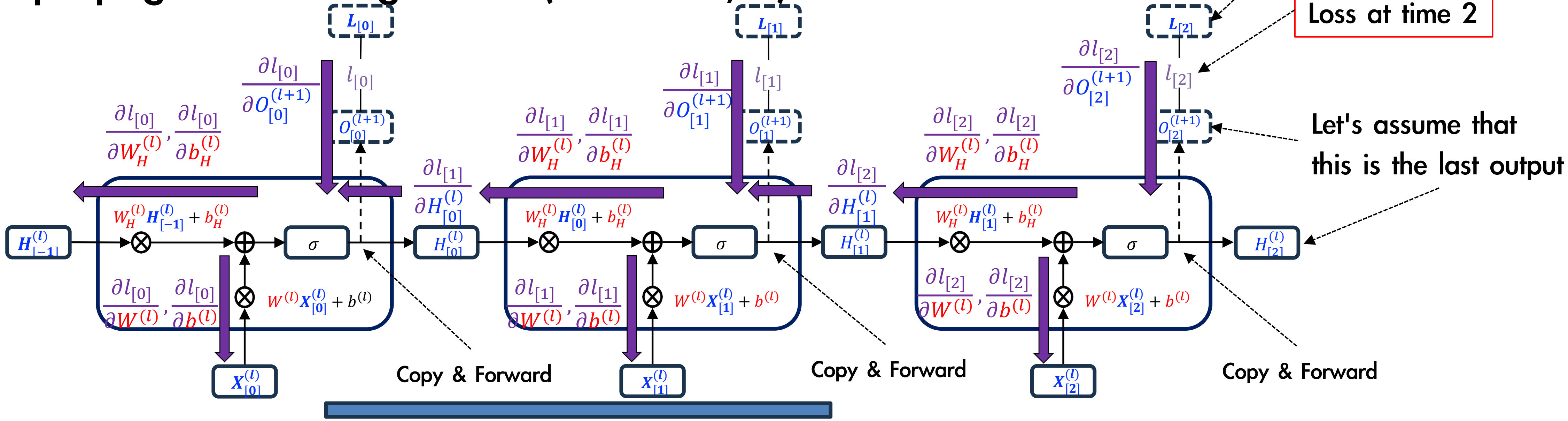
$$\frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}} = \frac{\partial l_{[2]}}{\partial O_{[2]}^{(l+1)}} \cdot \frac{\partial O_{[2]}^{(l+1)}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z_{[2]}} \cdot \frac{\partial z_{[2]}}{\partial H_{[1]}^{(l)}}$$

$$\frac{\partial l_{[2]}}{\partial b_H^{(l)}} = \frac{\partial l_{[2]}}{\partial O_{[2]}^{(l+1)}} \cdot \frac{\partial O_{[2]}^{(l+1)}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z_{[2]}} \cdot \frac{\partial z_{[2]}}{\partial b_H^{(l)}}$$

$$\frac{\partial l_{[2]}}{\partial b^{(l)}} = \frac{\partial l_{[2]}}{\partial O_{[2]}^{(l+1)}} \cdot \frac{\partial O_{[2]}^{(l+1)}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z_{[2]}} \cdot \frac{\partial z_{[2]}}{\partial b^{(l)}}$$

RNN Backpropagation

◈Backpropagation Through Time (BPTT – 2/4)



$$\frac{\partial l_{[1]}}{\partial W_H^{(l)}} = \left(\frac{\partial l_{[1]}}{\partial O_{[1]}^{(l+1)}} \cdot \frac{\partial O_{[1]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}} \cdot \frac{\partial H_{[1]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[1]}} \cdot \frac{\partial z_{[1]}}{\partial W_H^{(l)}}$$

$$\frac{\partial l_{[1]}}{\partial W^{(l)}} = \left(\frac{\partial l_{[1]}}{\partial O_{[1]}^{(l+1)}} \cdot \frac{\partial O_{[1]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}} \cdot \frac{\partial H_{[1]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[1]}} \cdot \frac{\partial z_{[1]}}{\partial W^{(l)}}$$

where

$$z_{[1]} = W_H^{(l)} H_{[0]}^{(l)} + b_H^{(l)} + W^{(l)} X_{[1]}^{(l)} + b^{(l)}$$

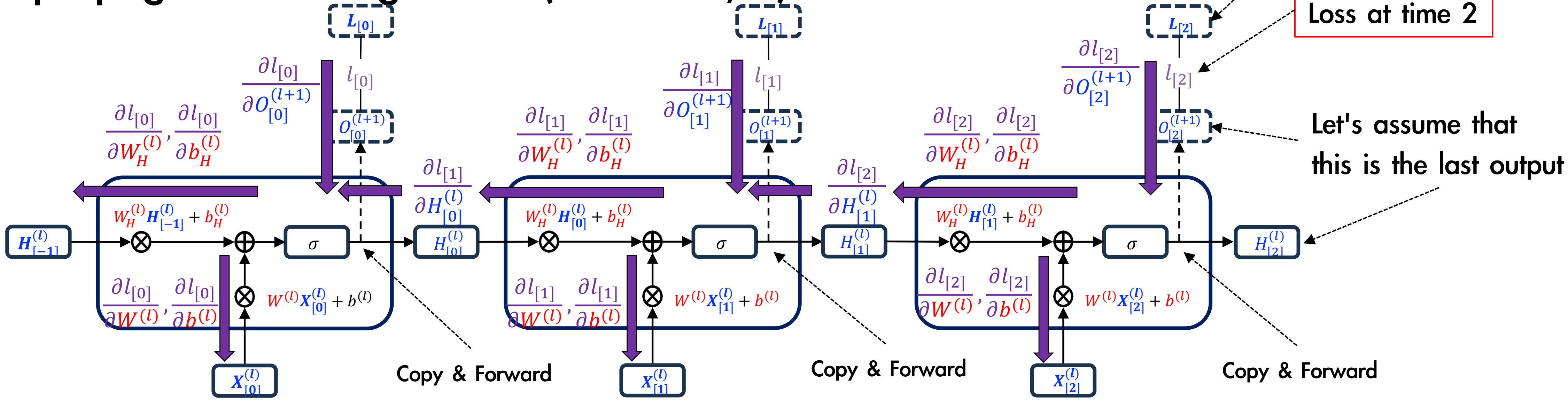
$$\frac{\partial l_{[1]}}{\partial b_H^{(l)}} = \left(\frac{\partial l_{[1]}}{\partial O_{[1]}^{(l+1)}} \cdot \frac{\partial O_{[1]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}} \cdot \frac{\partial H_{[1]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[1]}} \cdot \frac{\partial z_{[1]}}{\partial b_H^{(l)}}$$

$$\frac{\partial l_{[1]}}{\partial b^{(l)}} = \left(\frac{\partial l_{[1]}}{\partial O_{[1]}^{(l+1)}} \cdot \frac{\partial O_{[1]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}} \cdot \frac{\partial H_{[1]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[1]}} \cdot \frac{\partial z_{[1]}}{\partial b^{(l)}}$$

$$\frac{\partial l^{[1]}}{\partial H_{[0]}^{(l)}} = \left(\frac{\partial l_{[1]}}{\partial O_{[1]}^{(l+1)}} \cdot \frac{\partial O_{[1]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}} \cdot \frac{\partial H_{[1]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[1]}} \cdot \frac{\partial z_{[1]}}{\partial H_{[0]}^{(l)}}$$

RNN Backpropagation

◈Backpropagation Through Time (BPTT – 3/4)



$$\frac{\partial l_{[0]}}{\partial W_H^{(l)}} = \left(\frac{\partial l_{[0]}}{\partial O_{[0]}^{(l+1)}} \cdot \frac{\partial O_{[0]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[1]}}{\partial H_{[0]}^{(l)}} \cdot \frac{\partial H_{[0]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[0]}} \cdot \frac{\partial z_{[0]}}{\partial W_H^{(l)}}$$

$$\frac{\partial l_{[0]}}{\partial W^{(l)}} = \left(\frac{\partial l_{[0]}}{\partial O_{[0]}^{(l+1)}} \cdot \frac{\partial O_{[0]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[1]}}{\partial H_{[0]}^{(l)}} \cdot \frac{\partial H_{[0]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[0]}} \cdot \frac{\partial z_{[0]}}{\partial W^{(l)}}$$

$$\frac{\partial l_{[0]}}{\partial b_H^{(l)}} = \left(\frac{\partial l_{[0]}}{\partial O_{[0]}^{(l+1)}} \cdot \frac{\partial O_{[0]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[1]}}{\partial H_{[0]}^{(l)}} \cdot \frac{\partial H_{[0]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[0]}} \cdot \frac{\partial z_{[0]}}{\partial b_H^{(l)}}$$

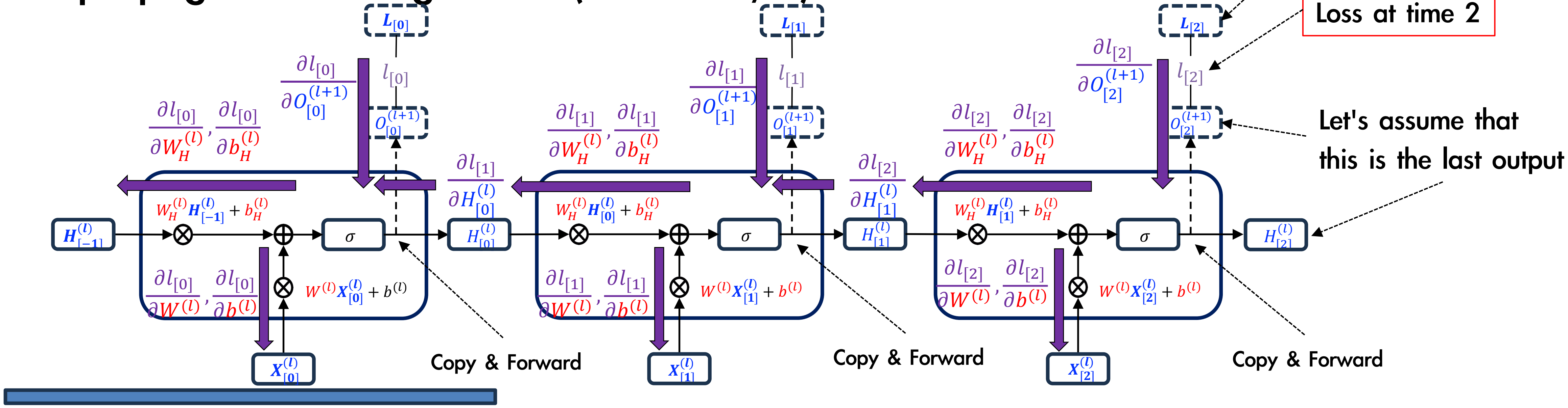
$$\frac{\partial l_{[0]}}{\partial b^{(l)}} = \left(\frac{\partial l_{[0]}}{\partial O_{[0]}^{(l+1)}} \cdot \frac{\partial O_{[0]}^{(l+1)}}{\partial \sigma} + \frac{\partial l_{[1]}}{\partial H_{[0]}^{(l)}} \cdot \frac{\partial H_{[0]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[0]}} \cdot \frac{\partial z_{[0]}}{\partial b^{(l)}}$$

where

$$z_{[0]} = W_H^{(l)} \boldsymbol{H}_{[-1]}^{(l)} + b_H^{(l)} + W^{(l)} \boldsymbol{X}_{[0]}^{(l)} + b^{(l)}$$

RNN Backpropagation

◈Backpropagation Through Time (BPTT – 4/4)



$$\frac{\partial l_{[0]}}{\partial W_H^{(l)}} = \left(\frac{\partial l_{[0]}}{\partial O_{[0]}^{(l+1)}} \cdot \frac{\partial O_{[0]}^{(l+1)}}{\partial \sigma} + \boxed{\frac{\partial l_{[1]}}{\partial H_{[0]}^{(l)}}} \cdot \frac{\partial H_{[0]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[0]}} \cdot \frac{\partial z_{[0]}}{\partial W_H^{(l)}}$$

$$\boxed{\frac{\partial l^{[1]}}{\partial H_{[0]}^{(l)}}} = \left(\frac{\partial l_{[1]}}{\partial O_{[1]}^{(l+1)}} \cdot \frac{\partial O_{[1]}^{(l+1)}}{\partial \sigma} + \boxed{\frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}}} \cdot \frac{\partial H_{[1]}^{(l)}}{\partial \sigma} \right) \cdot \frac{\partial \sigma}{\partial z_{[1]}} \cdot \frac{\partial z_{[1]}}{\partial H_{[0]}^{(l)}}$$

$$\boxed{\frac{\partial l_{[2]}}{\partial H_{[1]}^{(l)}}} = \frac{\partial l_{[2]}}{\partial O_{[2]}^{(l+1)}} \cdot \frac{\partial O_{[2]}^{(l+1)}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z_{[2]}} \cdot \frac{\partial z_{[2]}}{\partial H_{[1]}^{(l)}}$$

RNN Backpropagation

◈Backpropagation Through Time (BPTT)

– At each time step, we need to consider not only the gradient from the current time step but also the gradient that's being propagated backward from the future time steps

  • This is because the hidden state at any time step influences the outputs of all subsequent time steps

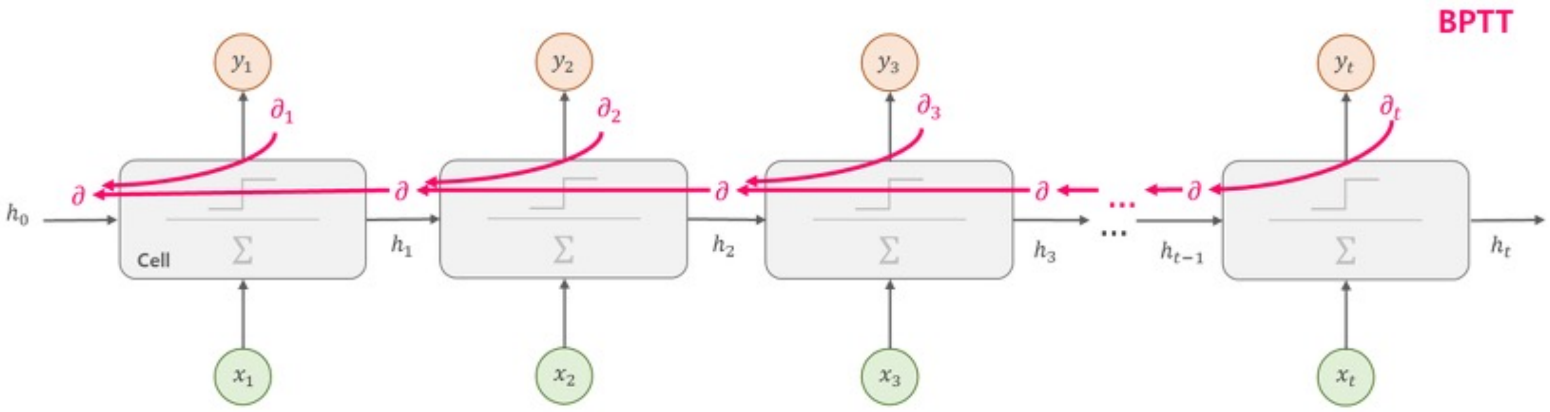
– Demerits

  • Exploding and Vanishing Gradients

    ➢ Not suitable for very long sequences

  • Computational and memory intensity

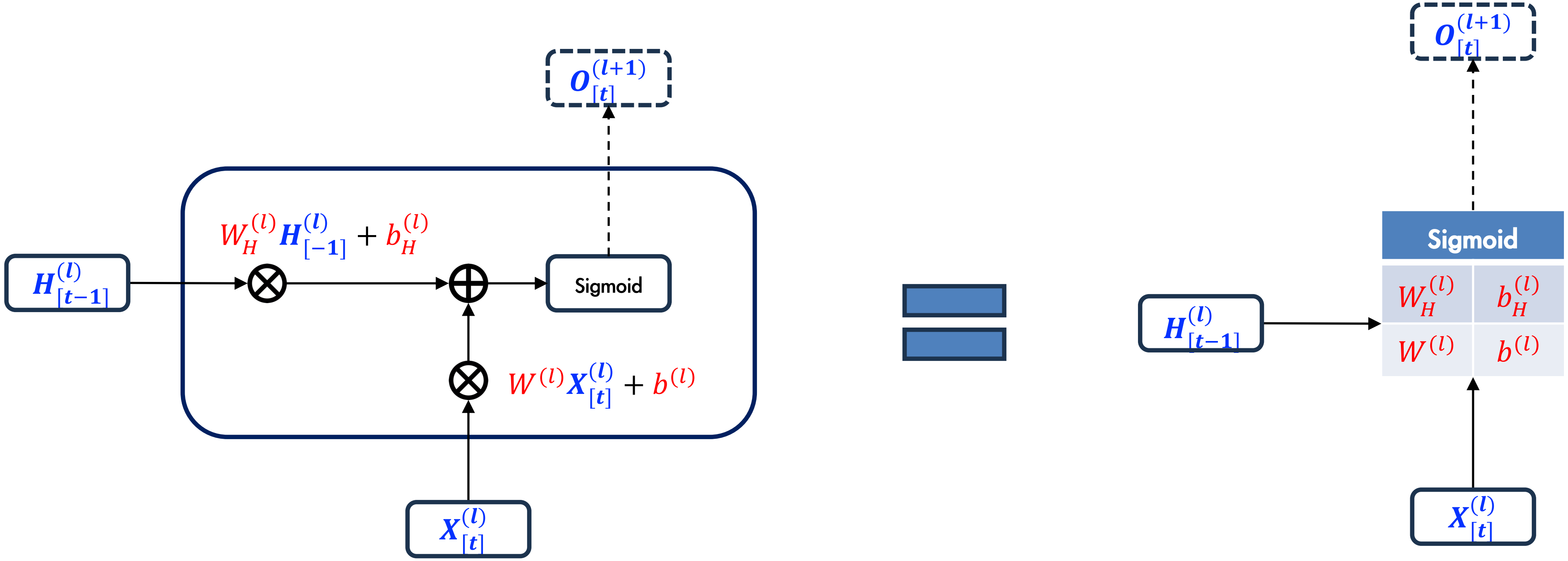
  • Difficulty in parallelization



LSTM
(Long-Short Term Memory)

LSTM Cell

◈A RNN Cell Description (표시)

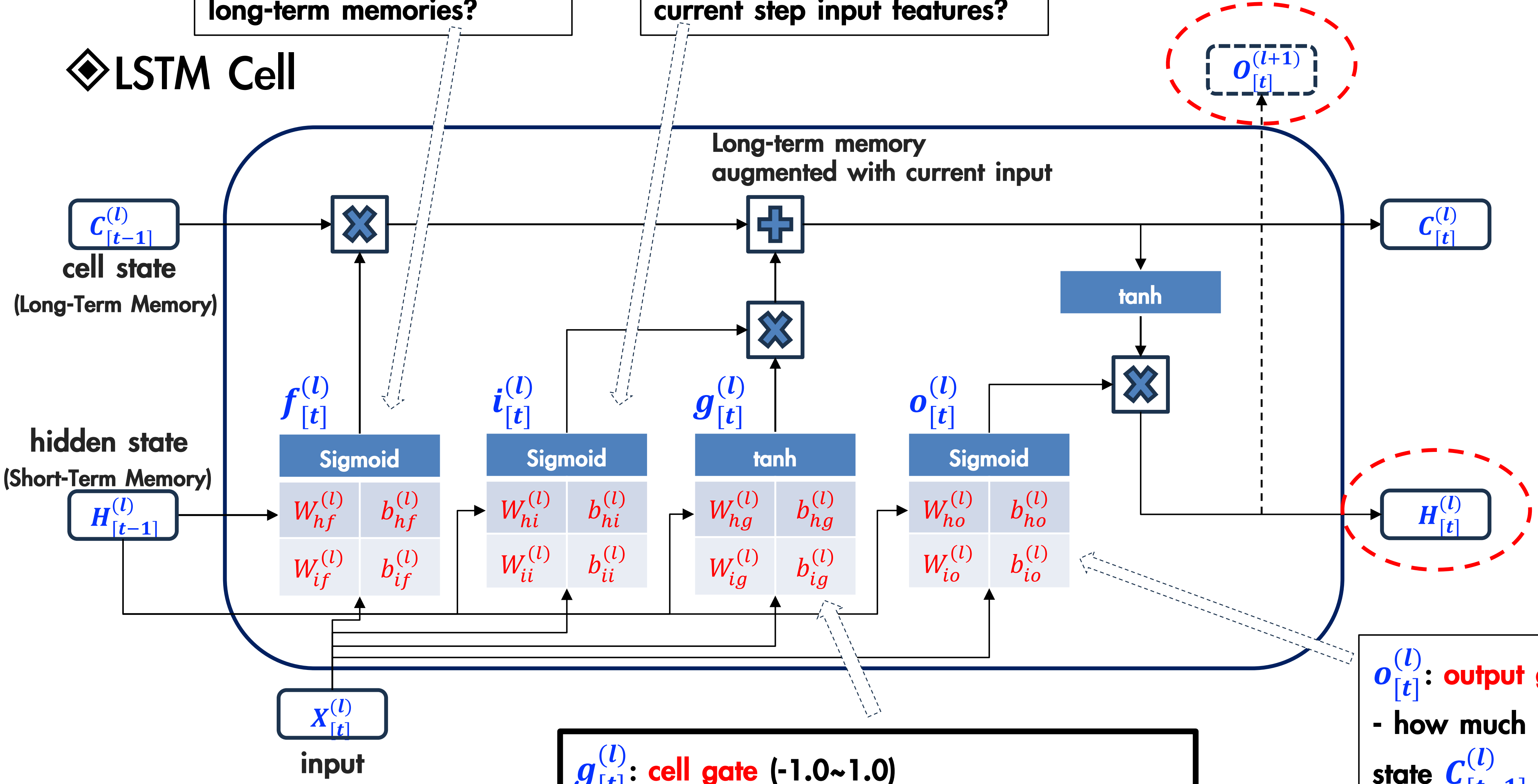


LSTM Cell

$f_{[t]}^{(l)}$: **forget gate** (0.0~1.0)
- how much it forgets its long-term memories?

$i_{[t]}^{(l)}$: **input gate** (0.0~1.0)
- How important is the current step input features?

◆LSTM Cell

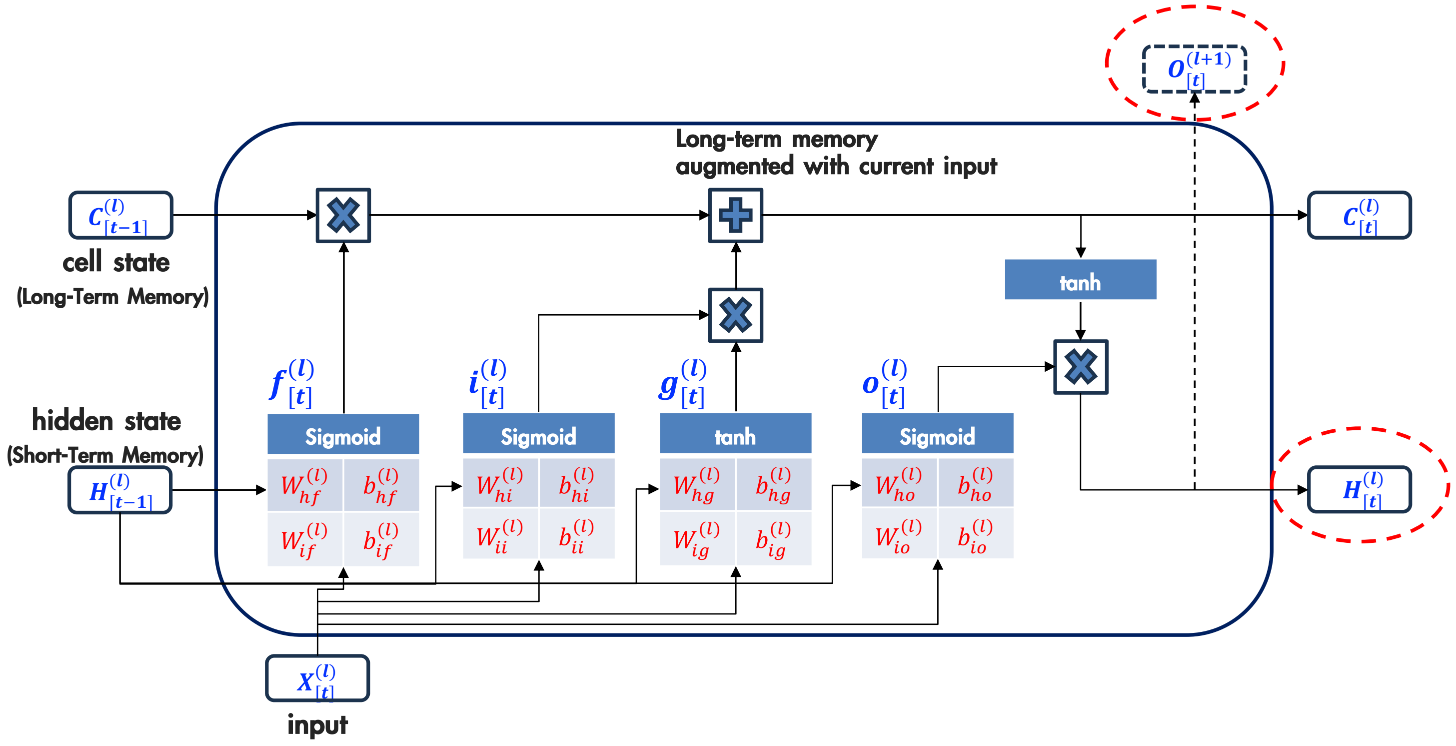


$g_{[t]}^{(l)}$: **cell gate** (-1.0~1.0)
- a non-linear transformation of the new input

$o_{[t]}^{(l)}$: **output gate** (0.0~1.0)
- how much of the current cell state $C_{[t-1]}^{(l)}$ should go to the new output and the new hidden state

LSTM Cell

◈LSTM Cell



$$f_{[t]}^{(l)} = sigmoid\left(W_{hf}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{hf}^{(l)} + W_{if}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{if}^{(l)}\right)$$

$$i_{[t]}^{(l)} = sigmoid\left(W_{hi}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{hi}^{(l)} + W_{ii}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{ii}^{(l)}\right)$$

$$g_{[t]}^{(l)} = tanh\left(W_{hg}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{hg}^{(l)} + W_{ig}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{ig}^{(l)}\right)$$

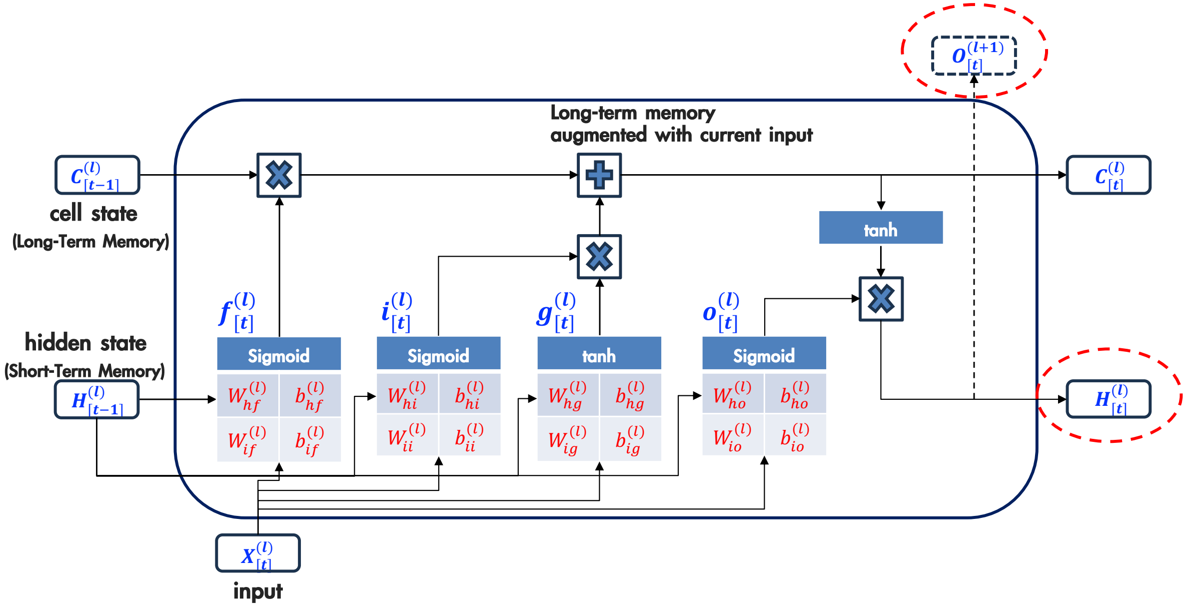
$$o_{[t]}^{(l)} = sigmoid\left(W_{ho}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{ho}^{(l)} + W_{io}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{io}^{(l)}\right)$$

$$C_{[t]}^{(l)} = f_{[t]}^{(l)} \cdot C_{[t-1]}^{(l)} + i_{[t]}^{(l)} \cdot g_{[t]}^{(l)}$$

$$H_{[t]}^{(l)} = o_{[t]}^{(l)} \cdot tanh\left(C_{[t]}^{(l)}\right)$$

LSTM Cell

◈LSTM Cell overcomes the BPTT problems



– Avoid the vanishing and exploding gradient problems faced by traditional RNNs

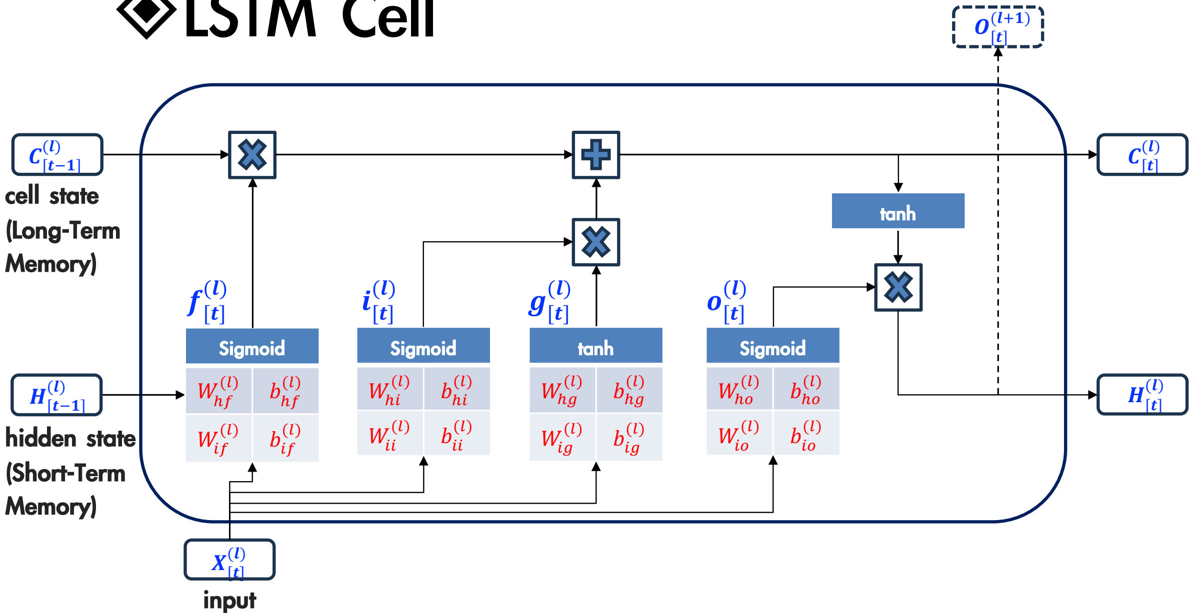
- This is achieved because the cell state has multiple gates but avoids non-linear transformations like tanh or sigmoid when transferring information from one step to the next
- This design ensures that the gradient can remain constant and doesn't vanish during long sequences

– Instead of directly making the cell state like traditional RNNs, LSTMs decide which cell state values to update, and by how much, through a gating mechanism

– LSTM can maintain and propagate relevant information through long sequences without the gradients vanishing, enabling them to learn long-term dependencies effectively

LSTM Cell

◈LSTM Cell



$$f_{[t]}^{(l)} = sigmoid\left(W_{hf}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{hf}^{(l)} + W_{if}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{if}^{(l)}\right)$$

$$i_{[t]}^{(l)} = sigmoid\left(W_{hi}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{hi}^{(l)} + W_{ii}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{ii}^{(l)}\right)$$

$$g_{[t]}^{(l)} = tanh\left(W_{hg}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{hg}^{(l)} + W_{ig}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{ig}^{(l)}\right)$$

$$o_{[t]}^{(l)} = sigmoid\left(W_{ho}^{(l)} \boldsymbol{H}_{[t-1]}^{(l)} + b_{ho}^{(l)} + W_{io}^{(l)} \boldsymbol{X}_{[t]}^{(l)} + b_{io}^{(l)}\right)$$

$$C_{[t]}^{(l)} = f_{[t]}^{(l)} \cdot C_{[t-1]}^{(l)} + i_{[t]}^{(l)} \cdot g_{[t]}^{(l)}$$

$$H_{[t]}^{(l)} = o_{[t]}^{(l)} \cdot tanh\left(C_{[t]}^{(l)}\right)$$

```python
lstm_cell = nn.LSTMCell(input_size=3, hidden_size=4)

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

# >>> weight_ih torch.Size([16, 3]):W_if,W_ii,W_ig,W_io

# >>> weight_hh torch.Size([16, 4]):W_hf,W_hi,W_hg,W_ho

# >>> bias_ih torch.Size([16]):b_if,b_ii,b_ig,b_io

# >>> bias_hh torch.Size([16]):b_hf,b_hi,b_hg,b_ho
```

LSTM Layer

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

```
lstm_cell = nn.LSTMCell(input_size=3, hidden_size=4)
# sequence size (L): 6, input size (F): 3
input = torch.randn(6, 3)

# hidden size: 4, cell size: 4
hx = (torch.randn(4), torch.randn(4))
           Hidden          Cell
output = []
for i in range(6):
  hx = lstm_cell(input[i], hx)
  output.append(hx)

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

```
0 (tensor([-0.2236, -0.3512,  0.0760,  0.0570], grad_fn=<SqueezeBackward1>),   Hidden
   tensor([-0.4575, -0.9113,  0.2719,  0.1065], grad_fn=<SqueezeBackward1>))   Cell

1 (tensor([-0.0392, -0.2142,  0.1468, -0.0885], grad_fn=<SqueezeBackward1>),
   tensor([-0.1755, -0.7796,  0.4526, -0.1401], grad_fn=<SqueezeBackward1>))

2 (tensor([ 0.1761, -0.3698,  0.1268, -0.0105], grad_fn=<SqueezeBackward1>),
   tensor([ 0.3203, -0.5426,  0.2170, -0.0517], grad_fn=<SqueezeBackward1>))

3 (tensor([-0.0341, -0.3021, -0.0076, -0.1843], grad_fn=<SqueezeBackward1>),
   tensor([-0.0629, -0.7238, -0.0288, -0.3589], grad_fn=<SqueezeBackward1>))

4 (tensor([-0.0351, -0.3190,  0.0126, -0.2346], grad_fn=<SqueezeBackward1>),
   tensor([-0.0945, -0.8489,  0.0474, -0.4533], grad_fn=<SqueezeBackward1>))

5 (tensor([ 0.0726, -0.4724, -0.0524, -0.0963], grad_fn=<SqueezeBackward1>),
   tensor([ 0.1181, -0.7899, -0.1151, -0.4005], grad_fn=<SqueezeBackward1>))
```

LSTM

LSTM

◈**LSTM** = a multi-layered sequence of LSTM cells (Num Layers: K, Sequence Length: L)

```
lstm = nn.LSTM(input_size=3, hidden_size=4, num_layers=2)

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

# >>> weight_ih_l0 torch.Size([16, 3])
# >>> weight_hh_l0 torch.Size([16, 4])
# >>> bias_ih_l0 torch.Size([16])
# >>> bias_hh_l0 torch.Size([16])
# >>> weight_ih_l1 torch.Size([16, 4])
# >>> weight_hh_l1 torch.Size([16, 4])
# >>> bias_ih_l1 torch.Size([16])
# >>> bias_hh_l1 torch.Size([16])
```

LSTM

- **LSTM** = a multi-layered sequence of LSTM cells (Num Layers: K, Sequence

```python
lstm = nn.LSTM(
  input_size=3, hidden_size=4, num_layers=2
)

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

output, (hidden_state, cell_state) = lstm(input)

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

print()

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

```
0 tensor([-0.0247, -0.0655,  0.0380, -0.0376], grad_fn=<UnbindBackward0>)
1 tensor([-0.0515, -0.1203,  0.0627, -0.0655], grad_fn=<UnbindBackward0>)
2 tensor([-0.0570, -0.1428,  0.0708, -0.0681], grad_fn=<UnbindBackward0>)
3 tensor([-0.0470, -0.1604,  0.0810, -0.0916], grad_fn=<UnbindBackward0>)
4 tensor([-0.0503, -0.1819,  0.0907, -0.1150], grad_fn=<UnbindBackward0>)
5 tensor([-0.0514, -0.1795,  0.0911, -0.0987], grad_fn=<UnbindBackward0>)

0 tensor([ 0.0203,  0.0033, -0.2166, -0.0344], grad_fn=<UnbindBackward0>)
tensor([ 0.0256,  0.0067, -0.4132, -0.1036], grad_fn=<UnbindBackward0>)
1 tensor([-0.0514, -0.1795,  0.0911, -0.0987], grad_fn=<UnbindBackward0>)
tensor([-0.0944, -0.3841,  0.2684, -0.1816], grad_fn=<UnbindBackward0>)
```

LSTM

◈LSTM with Batched Input: Option 1. [Sequence, Batch, Input] $L \times N \times F$

```
lstm = nn.LSTM(
  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, _) = lstm(batch_input)

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

print()

print(batch_hidden_state.shape)
for idx, hidden in enumerate(batch_hidden_state):
  print(idx, hidden.shape)  # shape: 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])
```

LSTM

◈LSTM with Batched Input: Option 2. [Batch, Sequence, Input] *N × L × F*

```python
lstm = nn.LSTM(
  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, _) = lstm(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 LSTM

◈Bidirectional LSTM

= a multi-layered sequence of bidirectional LSTM cells

(Num Layers: K, Sequence Length: L, Bidirectional: True)

```python
rnn = nn.LSTM(
  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([16, 3])
# >>> weight_hh_l0 torch.Size([16, 4])
# >>> bias_ih_l0 torch.Size([16])
# >>> bias_hh_l0 torch.Size([16])
# >>> weight_ih_l0_reverse torch.Size([16, 3])
# >>> weight_hh_l0_reverse torch.Size([16, 4])
# >>> bias_ih_l0_reverse torch.Size([16])
# >>> bias_hh_l0_reverse torch.Size([16])
# >>> weight_ih_l1 torch.Size([16, 8])
# >>> weight_hh_l1 torch.Size([16, 4])
# >>> bias_ih_l1 torch.Size([16])
# >>> bias_hh_l1 torch.Size([16])
# >>> weight_ih_l1_reverse torch.Size([16, 8])
# >>> weight_hh_l1_reverse torch.Size([16, 4])
# >>> bias_ih_l1_reverse torch.Size([16])
# >>> bias_hh_l1_reverse torch.Size([16])
```

Bidirectional LSTM

◈Bidirectional LSTM

```
lstm = nn.LSTM(
  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, _) = lstm(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([-0.0077, -0.0905,  0.1366, -0.1025, -0.1140, -0.1305,
0.1807, -0.1883], grad_fn=<UnbindBackward0>)
1 tensor([-0.0062, -0.1749,  0.2050, -0.1524, -0.1216, -0.1336,
0.1727, -0.1992], grad_fn=<UnbindBackward0>)
2 tensor([-0.0205, -0.2297,  0.2082, -0.1296, -0.1233, -0.1144,
0.1761, -0.1912], grad_fn=<UnbindBackward0>)
3 tensor([-0.0419, -0.2385,  0.2063, -0.0739, -0.1092, -0.0920,
0.1816, -0.1681], grad_fn=<UnbindBackward0>)
4 tensor([-0.0339, -0.2420,  0.2295, -0.0684, -0.0883, -0.0779,
0.1613, -0.1432], grad_fn=<UnbindBackward0>)
5 tensor([-0.0311, -0.2600,  0.2044, -0.0561, -0.0740, -0.0696,
0.1292, -0.0991], grad_fn=<UnbindBackward0>)

0 tensor([ 0.3271, -0.0983, -0.0019,  0.1823],
grad_fn=<UnbindBackward0>)
1 tensor([-0.1731, -0.0902,  0.0813,  0.2303],
grad_fn=<UnbindBackward0>)
2 tensor([-0.0311, -0.2600,  0.2044, -0.0561],
grad_fn=<UnbindBackward0>)
3 tensor([-0.1140, -0.1305,  0.1807, -0.1883],
grad_fn=<UnbindBackward0>)
```

Bidirectional LSTM

◈LSTM with Batched Input: Option 1. [Sequence, Batch, Input] *L × N × F*

```python
lstm = nn.LSTM(
  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, _) = lstm(batch_input)

print(batch_output.shape)   # >>> torch.Size([6, 10, 8])
for idx, out in enumerate(batch_output):
  print(idx, out.shape)     # >>> idx torch.Size([10, 8])

print()

print(batch_hidden_state.shape) # >>> torch.Size([4, 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 LSTM

◈LSTM with Batched Input: Option 2. [Batch, Sequence, Input] $N \times L \times F$

```python
lstm = nn.LSTM(
  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, _) = lstm(batch_input)

print(batch_output.shape)   # >>> torch.Size([10, 6, 8])
for idx, out in enumerate(batch_output):
  print(idx, out.shape)     # >>> idx torch.Size([6, 8])

print()

print(batch_hidden_state.shape) # >>> torch.Size([4, 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])
```

LSTM Best Practice
- Cryptocurrency Data -

LSTM with PyTorch

◈LSTM with Cryptocurrency Dataset (BTC-WON)

```python
def get_btc_krw_data(sequence_size=10, validation_size=100, test_size=10, is_regression=True):
  X_train, X_validation, X_test, y_train, y_validation, y_test, y_train_date, y_validation_date, y_test_date \
    = get_cryptocurrency_data(
      sequence_size=sequence_size, validation_size=validation_size, test_size=test_size,
      target_column='Close', y_normalizer=1.0e7, is_regression=is_regression
  )
  train_crypto_currency_dataset = CryptoCurrencyDataset(X=X_train, y=y_train)
  validation_crypto_currency_dataset = CryptoCurrencyDataset(X=X_validation, y=y_validation)
  test_crypto_currency_dataset = CryptoCurrencyDataset(X=X_test, y=y_test)

  train_data_loader = DataLoader(
    dataset=train_crypto_currency_dataset, batch_size=wandb.config.batch_size, shuffle=True
  )
  validation_data_loader = DataLoader(
    dataset=validation_crypto_currency_dataset, batch_size=wandb.config.batch_size, shuffle=True
  )
  test_data_loader = DataLoader(
    dataset=test_crypto_currency_dataset, batch_size=len(test_crypto_currency_dataset), shuffle=True
  )

  return train_data_loader, validation_data_loader, test_data_loader
```

LSTM with PyTorch

◈LSTM with Cryptocurrency Dataset (BTC-WON)

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

      self.lstm = nn.LSTM(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 (self, x):
      x, hidden = self.lstm(x)
      x = x[:, -1, :]  # x.shape: [32, 128]
      x = self.fcn(x)
      return x

  my_model = MyModel(n_input=5, n_output=1)

  return my_model
```

LSTM with PyTorch

◈LSTM with Cryptocurrency Dataset (BTC-WON)

```
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,
    'weight_decay': args.weight_decay
  }

  project_name = "lstm_regression_btc_krw”
  wandb.init(
    mode="online" if args.wandb else "disabled",
    project=project_name,
    notes="btc_krw experiment with lstm",
    tags=["lstm", "regression", "btc_krw"],
    name=run_time_str,
    config=config
  )
```

LSTM with PyTorch

◈LSTM with Cryptocurrency Dataset (BTC-WON)

```
def main(args):
  ...
  train_data_loader, validation_data_loader, _ = get_btc_krw_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 , weight_decay=wandb.config.weight_decay
  )
  regression_trainer = RegressionTrainer(
    project_name, model, optimizer, train_data_loader, validation_data_loader, None,
    run_time_str, wandb, device, CHECKPOINT_FILE_PATH
  )
  regression_trainer.train_loop()
  wandb.finish()
```

LSTM Test with PyTorch

◈LSTM Test with Cryptocurrency Dataset (BTC-WON)

```
def test_main(test_model):
  _, _, test_data_loader = get_btc_krw_data()

  test_model.eval()

  loss_fn = nn.MSELoss()

  loss_test = 0.0
  y_normalizer = 100

  print("[TEST DATA]")
    with torch.no_grad():
      for test_batch in test_data_loader:
        input_test, target_test = test_batch

        output_test = test_model(input_test)
```

LSTM Test with PyTorch

◈LSTM Test with Cryptocurrency Dataset (BTC-WON)

```
def test_main(test_model):
  ...

  with torch.no_grad():
    ...

    for idx, (output, target) in enumerate(zip(output_test, target_test)):
      print("{0:2}: {1:6,.2f} <--> {2:6,.2f} (Loss: {3:>13,.2f})".format(
        idx,
        output.item() * y_normalizer,
        target.item() * y_normalizer,
        abs(output.squeeze(dim=-1).item() - target.item()) * y_normalizer
      ))
```

LSTM Test with PyTorch

◈LSTM Test with Cryptocurrency Dataset (BTC-WON)

```
def predict_all(test_model):
  y_normalizer = 100

  X_train, X_validation, X_test, y_train, y_validation, y_test, y_train_date, y_validation_date, y_test_date \
    = get_cryptocurrency_data(
      sequence_size=10, validation_size=100, test_size=10,
      target_column='Close', y_normalizer=1.0e7, is_regression=True
  )

  train_crypto_currency_dataset = CryptoCurrencyDataset(X=X_train, y=y_train)
  validation_crypto_currency_dataset = CryptoCurrencyDataset(X=X_validation, y=y_validation)
  test_crypto_currency_dataset = CryptoCurrencyDataset(X=X_test, y=y_test)

  dataset_list = [
    train_crypto_currency_dataset, validation_crypto_currency_dataset, test_crypto_currency_dataset
  ]
  dataset_labels = [
    "train", "validation", "test"
  ]
  num = 0
  fig, axs = plt.subplots(3, 1, figsize=(6, 9))
```

LSTM Test with PyTorch

◈LSTM Test with Cryptocurrency Dataset (BTC-WON)

```
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()
```