Deep Learning

- Recurrent Neural Networks -

Eunbyung Park

Assistant Professor

School of Electronic and Electrical Engineering

Eunbyung Park (silverbottlep.github.io)

성균관대학교
SUNG KYUN KWAN UNIVERSITY

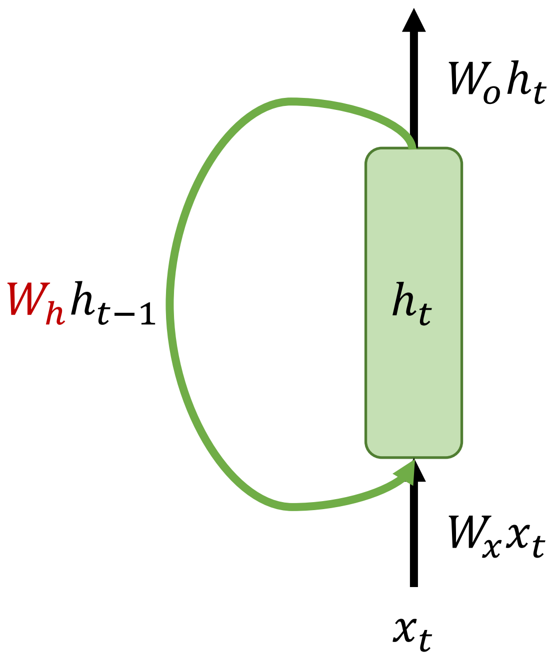
Recurrent Neural Networks

Recurrent Neural Network

- Variable input/output length (+)
- Memory functionality (+)
- Weight sharing (+)
- Sequential processing (-)
- Vanishing gradients (-)
- Hard to learn to preserve longer context in practice (-)

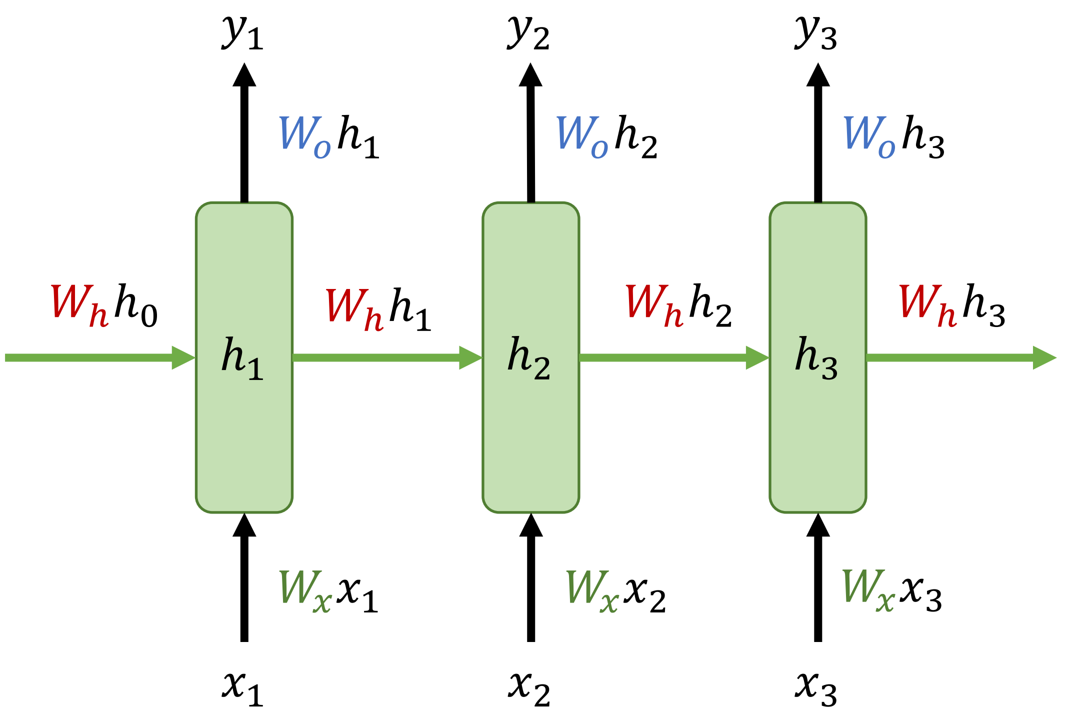
Recurrent Neural Network

- Internal state (memory, historical information)



$$h_t = f(x_t, h_{t-1})$$

$$e.g.\, \sigma(W_x x_t + W_h h_{t-1})$$



Recurrent Neural Networks

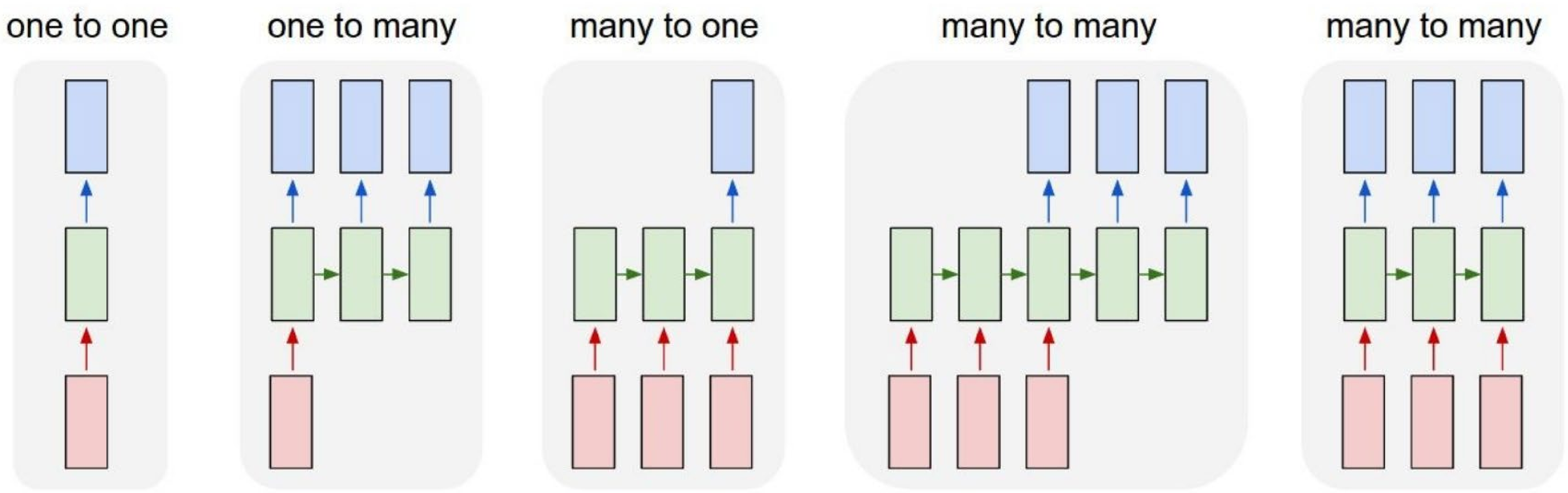


Image -> sequence of words

Video -> action class

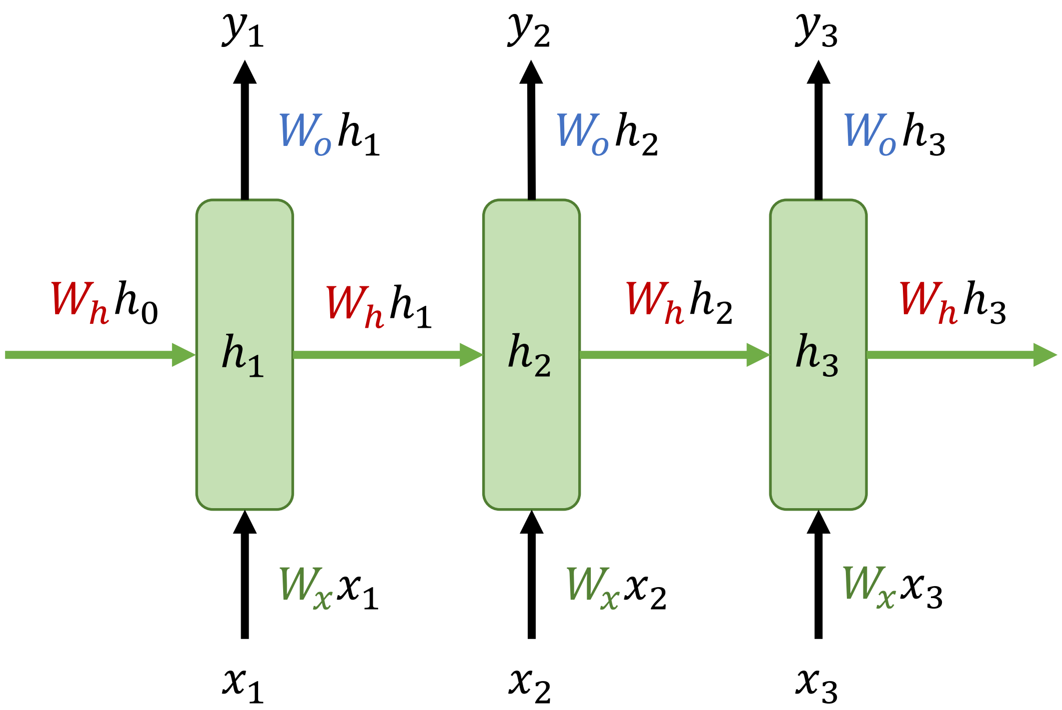
Video -> sequence of words

Video -> action class per each frames

lecture_10.pdf (stanford.edu)

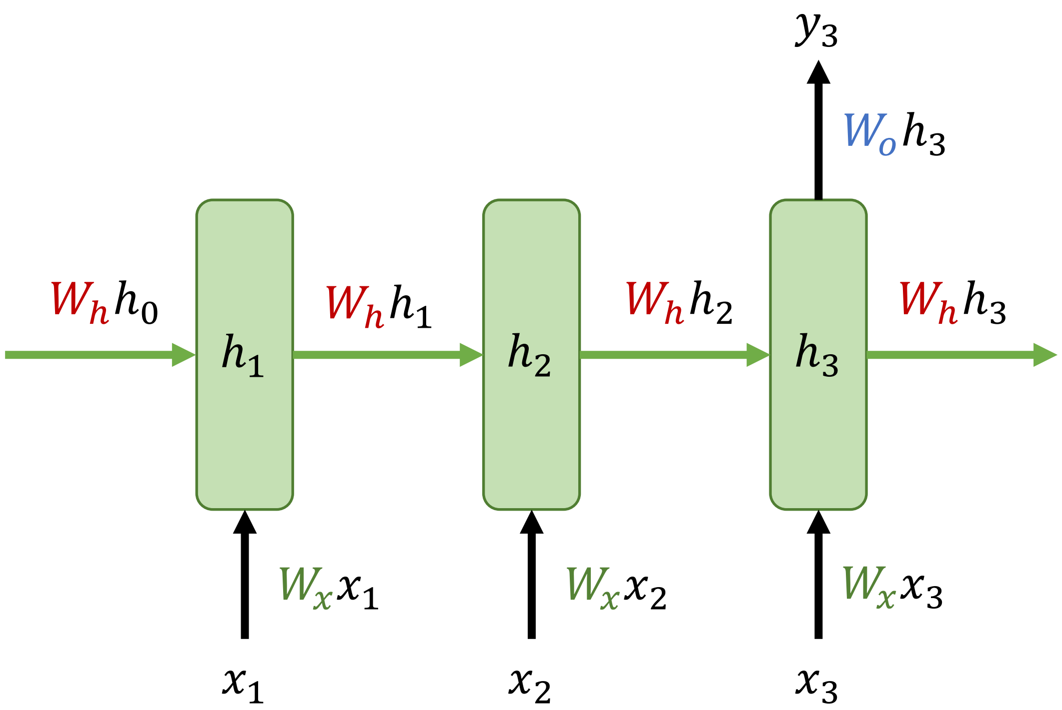
Recurrent Neural Network

- Many to many



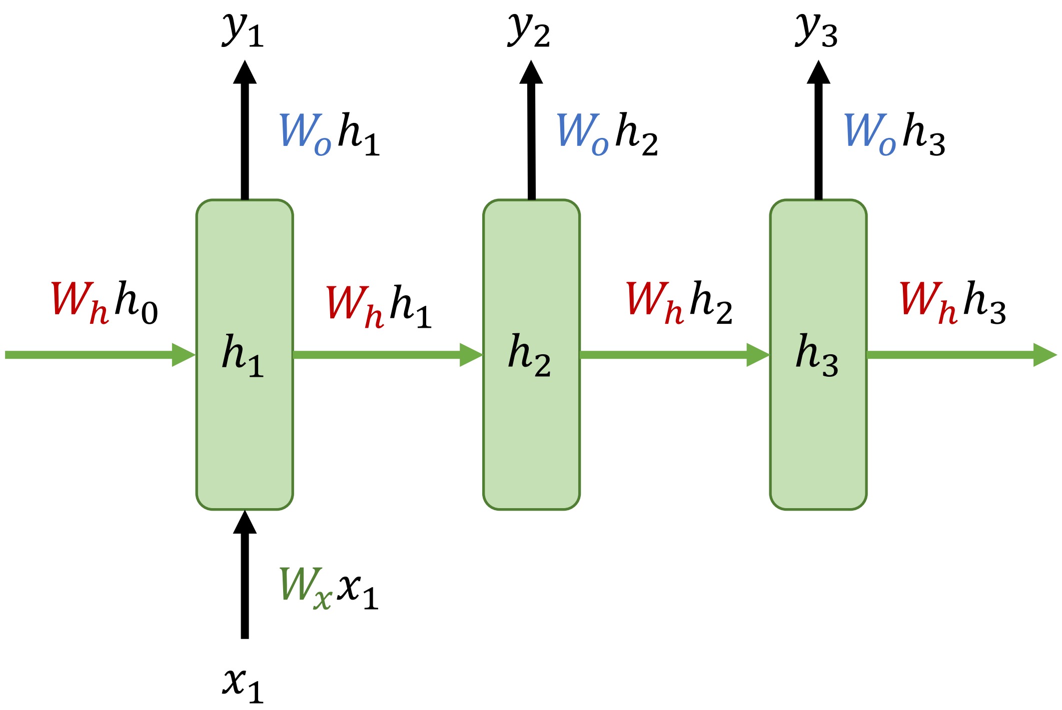
Recurrent Neural Network

- Many to one



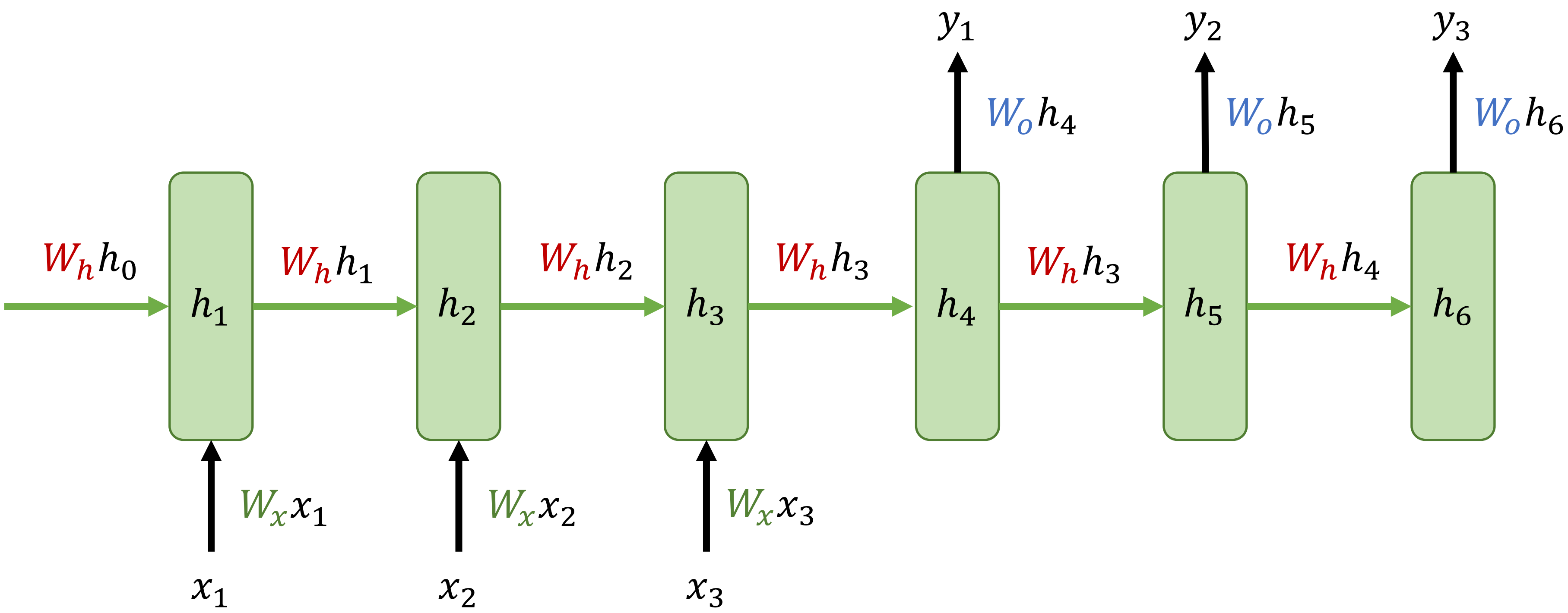
Recurrent Neural Network

- One to many



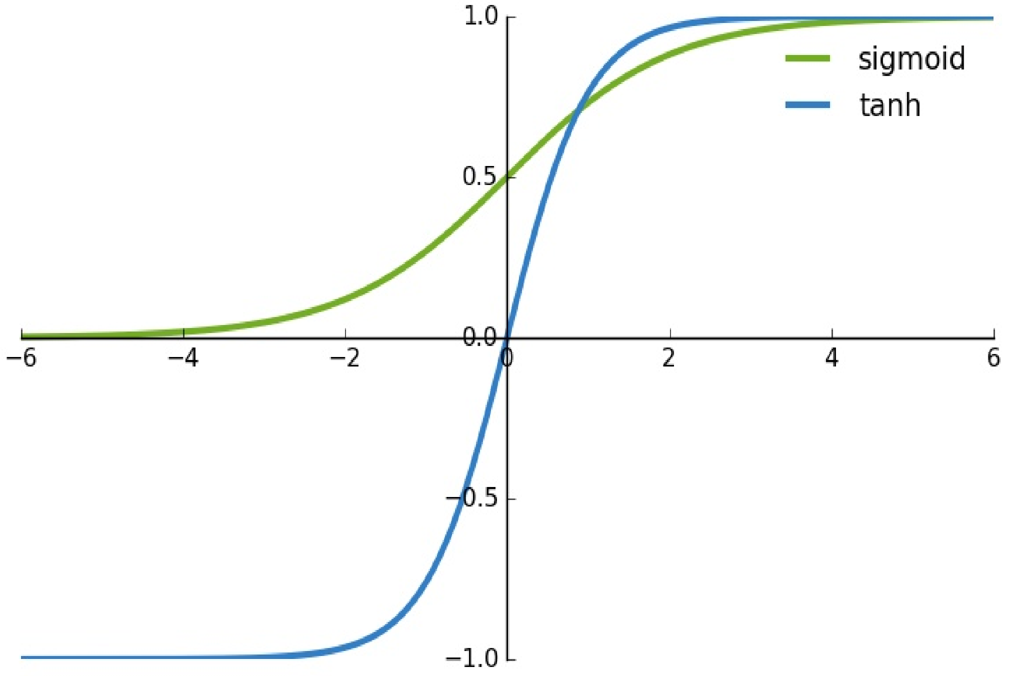
Recurrent Neural Network

- Many to many

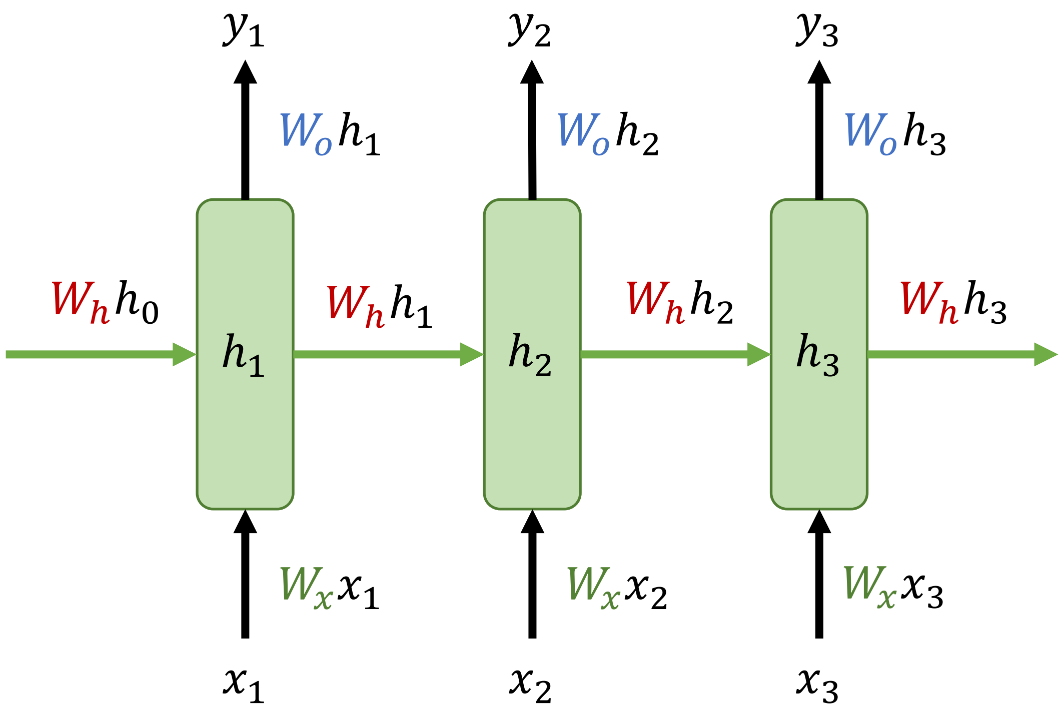


Activation Functions of RNNs

- Tanh is often used in RNNs to avoid gradient vanishing and gradient explosion
  - Shared weights multiplied repeatedly
  - May avoid exploding gradient
  - Zero mean activations may help faster convergence
  - Better empirical evidence



Recurrent Neural Networks

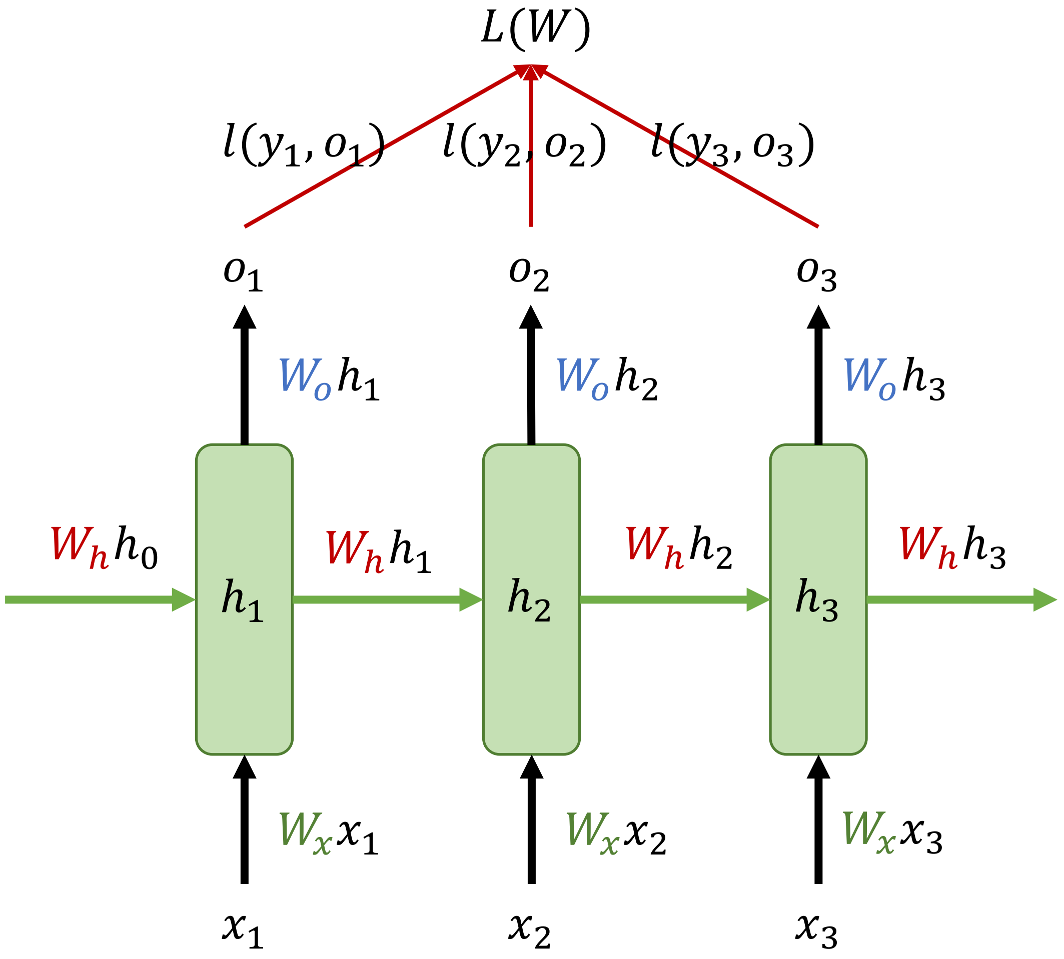


$$a_t = W_x x_t + W_h h_{t-1}$$

$$h_t = \tanh(a_t)$$

$$y_t = W_o h_t$$

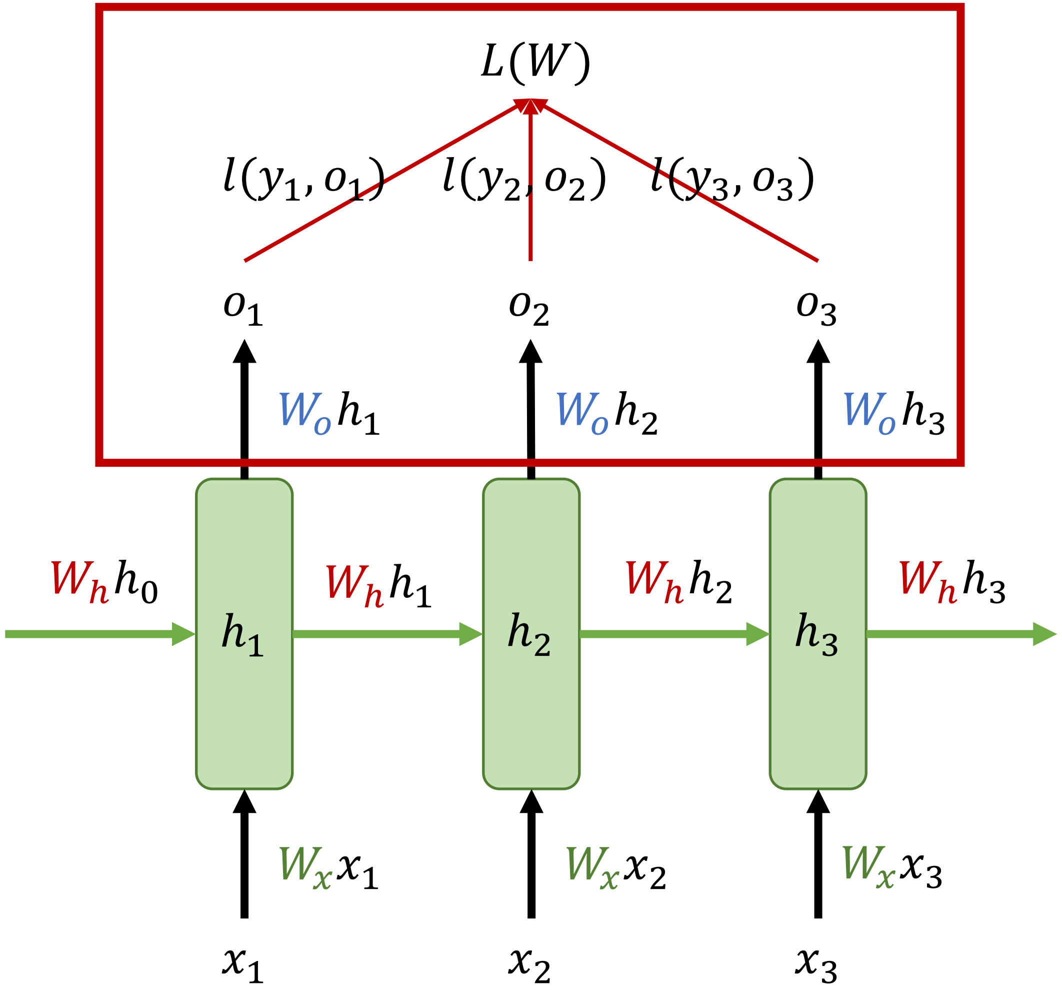
Backpropagation Through Time



Backpropagation Through Time

$$h_t = \tanh(W_x x_t + W_h h_{t-1})$$
$$o_t = W_o h_t$$
$$L(W_h, W_o, W_x) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t)$$

$$\frac{\partial L}{\partial W_o} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial o_t}{\partial W_o} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t} h_t^{\top}$$



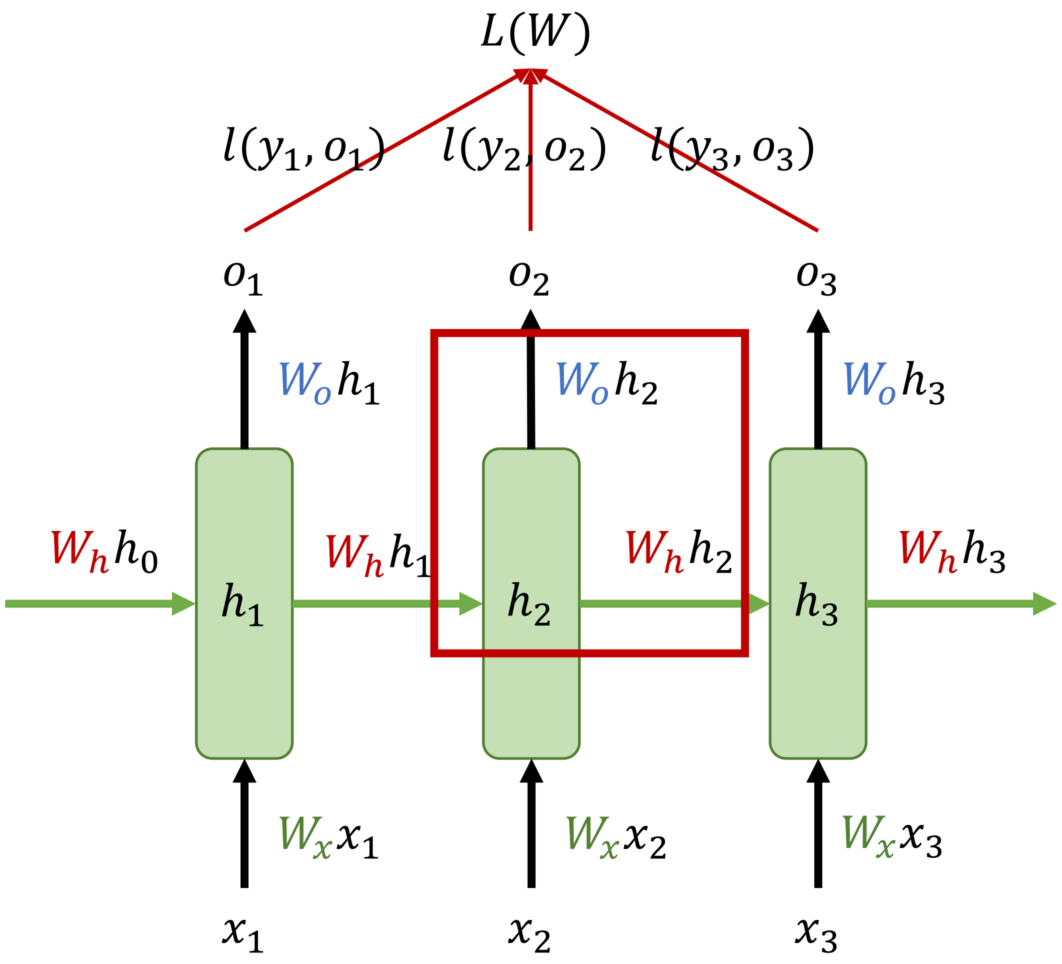
Backpropagation Through Time

$$h_t = \tanh(W_x x_t + W_h h_{t-1})$$
$$o_t = W_o h_t$$
$$L(W_h, W_o, W_x) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t)$$

$$\frac{\partial L}{\partial h_T} = \frac{\partial L}{\partial o_T}\frac{\partial o_T}{\partial h_T} = W_o \frac{\partial L}{\partial o_T}$$

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t} + \frac{\partial L}{\partial o_t}\frac{\partial o_t}{\partial h_t} = W_h^\top \frac{\partial L}{\partial h_{t+1}} + W_o^\top \frac{\partial L}{\partial o_t}$$

$$\frac{\partial L}{\partial h_t} = \sum_{i=t}^{T} \left(W_h^\top\right)^{T-i} W_o^\top \frac{\partial L}{\partial o_{T+t-i}}$$



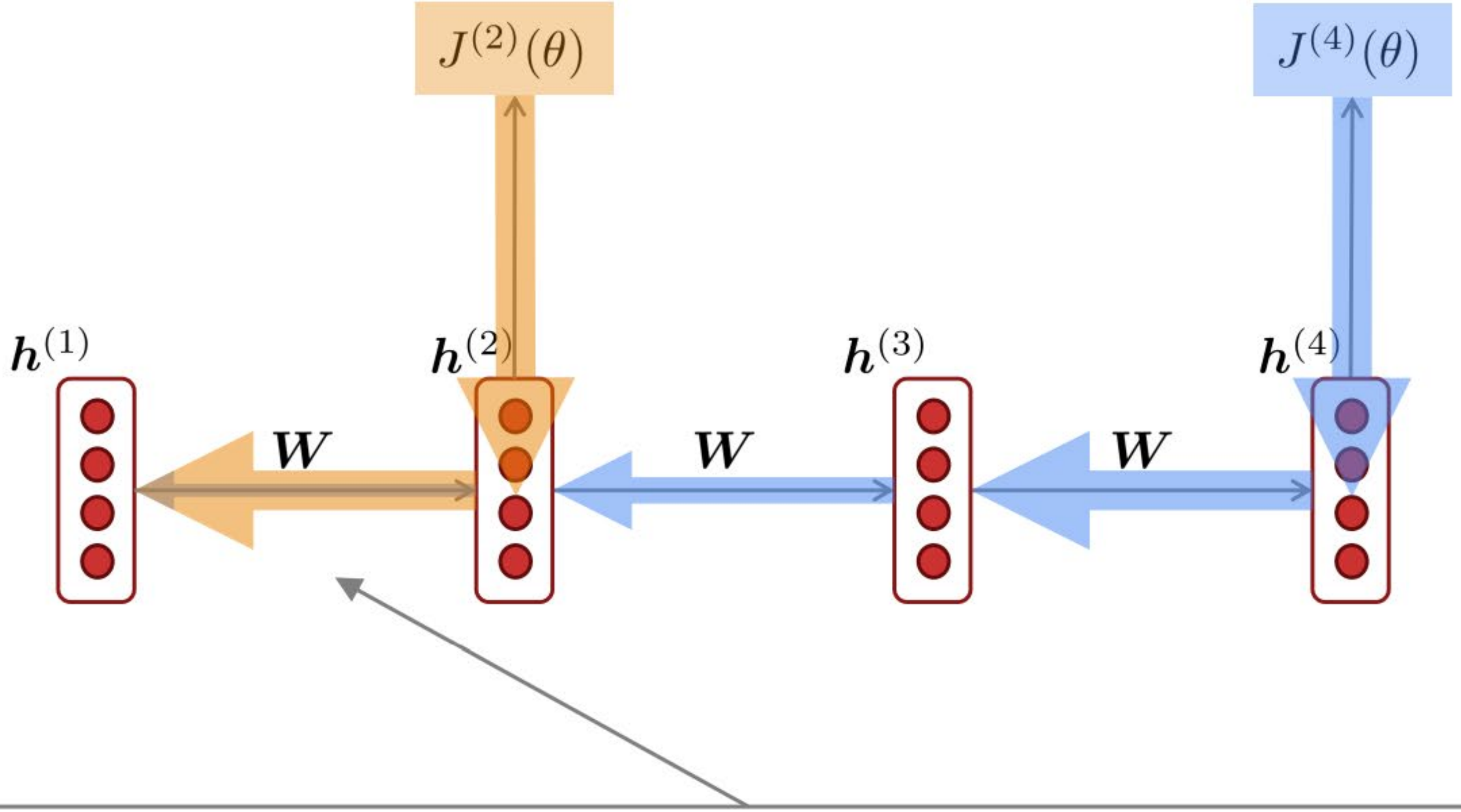
Vanishing or Exploding Gradient

- The largest eigenvalue is less than 1, then vanishing gradient
- The largest eigenvalue is greater than 1, then exploding gradient

$$\frac{\partial L}{\partial h_t} = \sum_{i=t}^{T} \left(W_h^\top\right)^{T-i} W_o^\top \frac{\partial L}{\partial o_{T+t-i}}$$

Why?

Vanishing or Exploding Gradient

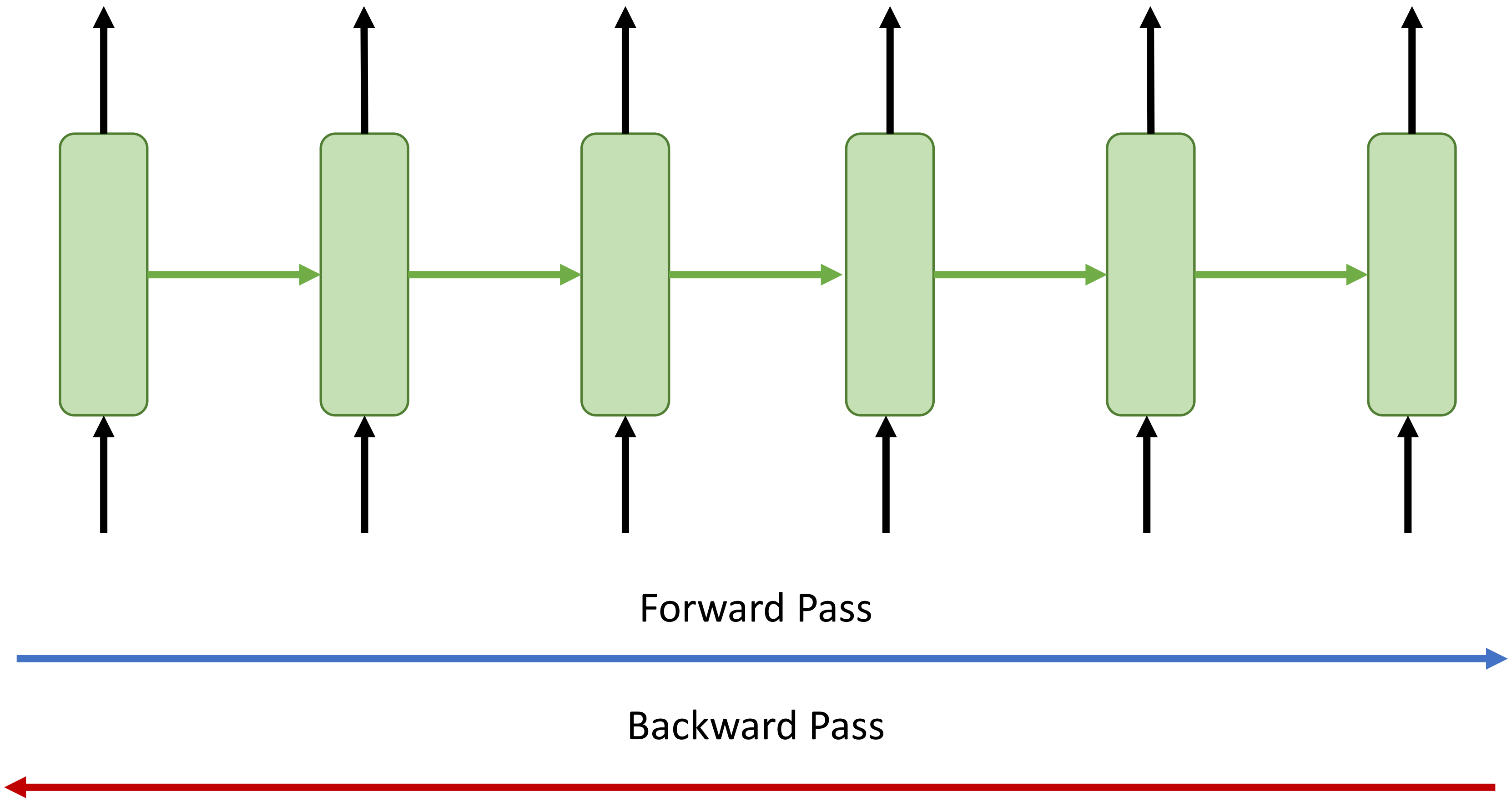


Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

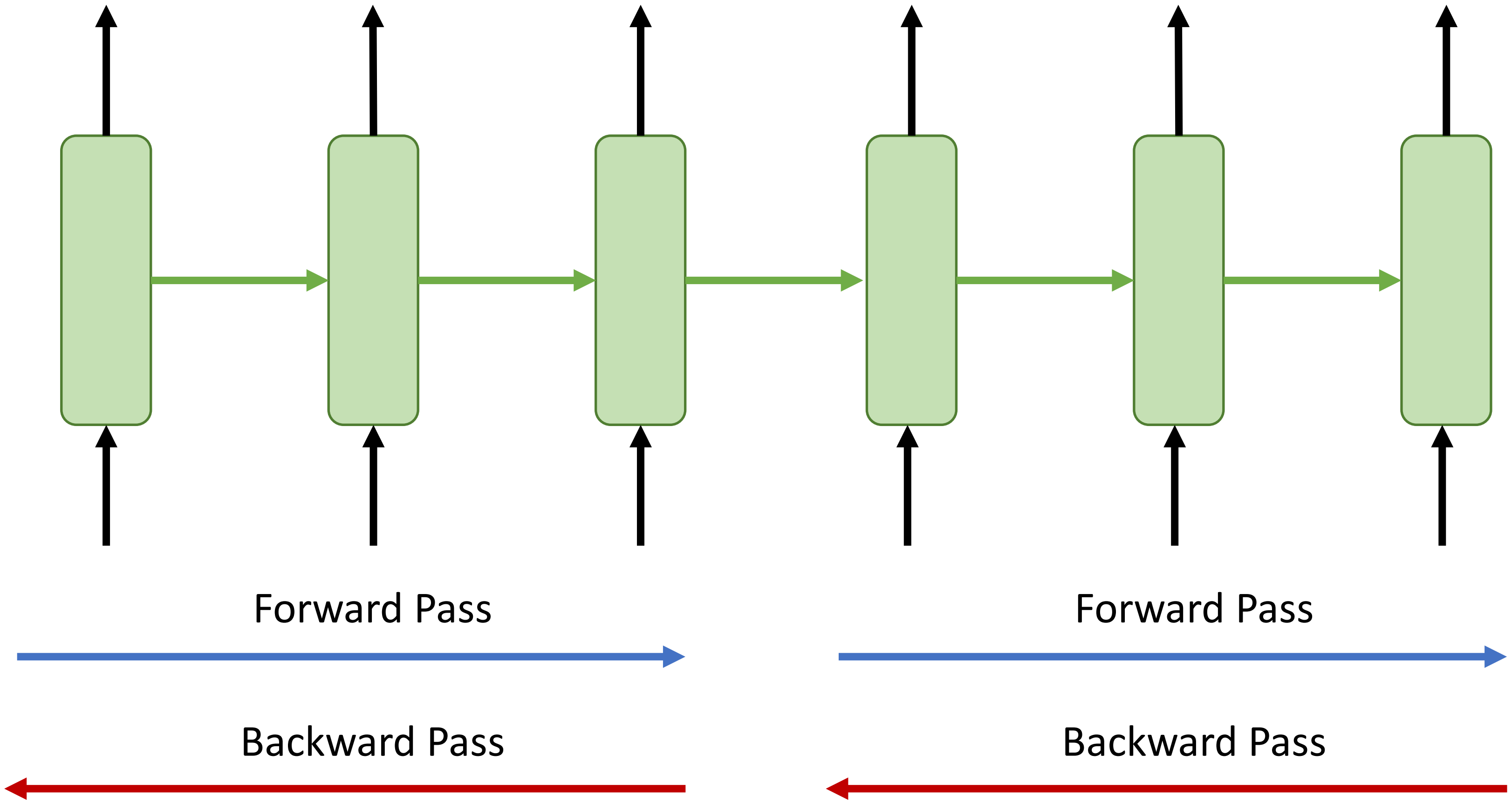
Backpropagation Through Time (BPTT)

- Spatial complexity increases proportional to the time steps



Truncated Backpropagation Through Time (TBPTT)

- Enabling training on longer sequence
  - But, Biased on short-term influence rather than long-term consequences
- Help avoiding vanishing or exploding gradients



Gradient Clipping

- Another trick to increase numerical stability

$$\nabla L \leftarrow \min(1, \frac{c}{||\nabla L||})\nabla L$$

Language Modeling

- Computing the joint probability of the sequence

$$p(x_1, x_2, \dots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, \dots, x_{t-1}) \approx p(x_t | h_{t-1})$$

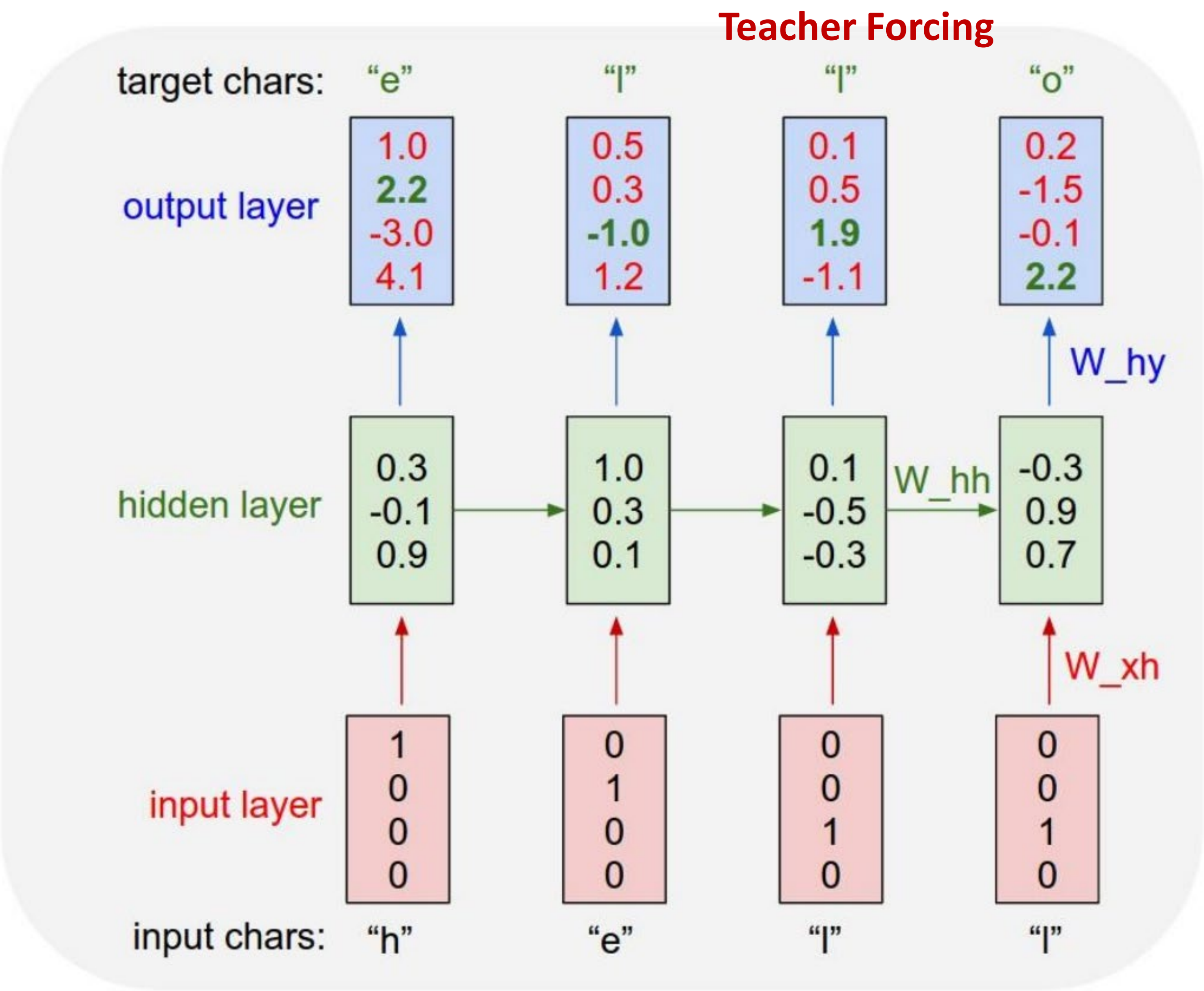
Stores the sequence information
up to time step t-1

- It is able to generate natural text, by drawing one token at a time

$$x_t \sim p(x_t | x_{t-1}, \dots, x_1)$$

Char-RNN (Character-Level Language Model)

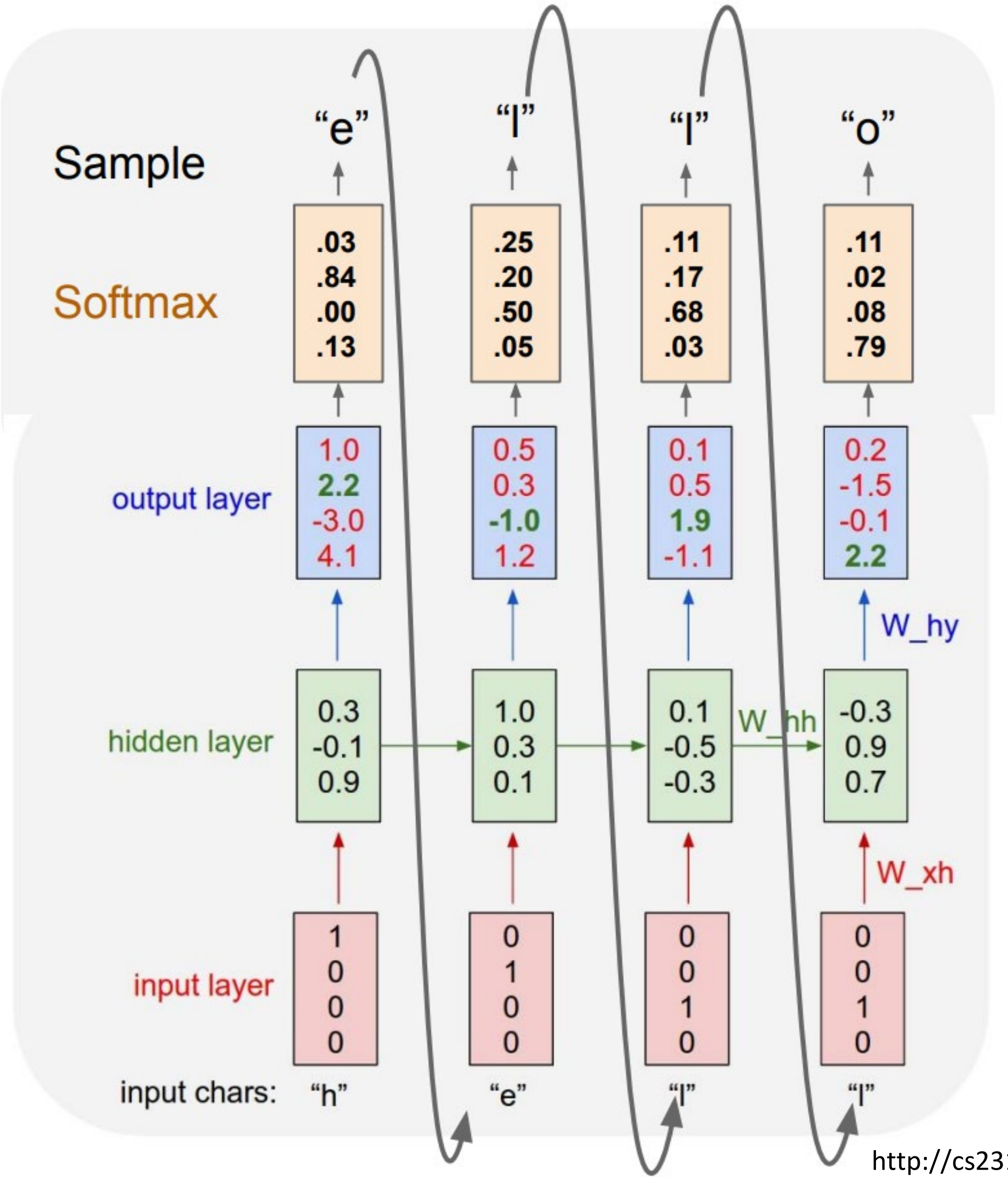
- Vocabulary – [h,e,l,o]
- Training sequence: “hello”



http://cs231n.stanford.edu/

Char-RNN (Character-Level Language Model)

- Autoregressive model
- One character at a time, feed back to model



http://cs231n.stanford.edu/

Example Implementation

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    # Hidden layer parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

[8.5. Implementation of Recurrent Neural Networks from Scratch — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]()

Example Implementation

```
def rnn(inputs, state, params):
    # Here `inputs` shape: (`num_steps`, `batch_size`, `vocab_size`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # Shape of `X`: (`batch_size`, `vocab_size`)
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

[8.5. Implementation of Recurrent Neural Networks from Scratch — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

Example Implementation

```python
class RNNModelScratch:  #@save
    """A RNN Model implemented from scratch."""
    def __init__(self, vocab_size, num_hiddens, device, get_params,
                 init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return self.init_state(batch_size, self.num_hiddens, device)
```

```python
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),)
```

8.5. Implementation of Recurrent Neural Networks from Scratch — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Example Implementation

```python
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """Generate new characters following the `prefix`."""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape(
        (1, 1))
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

[8.5. Implementation of Recurrent Neural Networks from Scratch — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

PyTorch Library

```python
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """Generate new characters following the `prefix`."""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape(
        (1, 1))
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

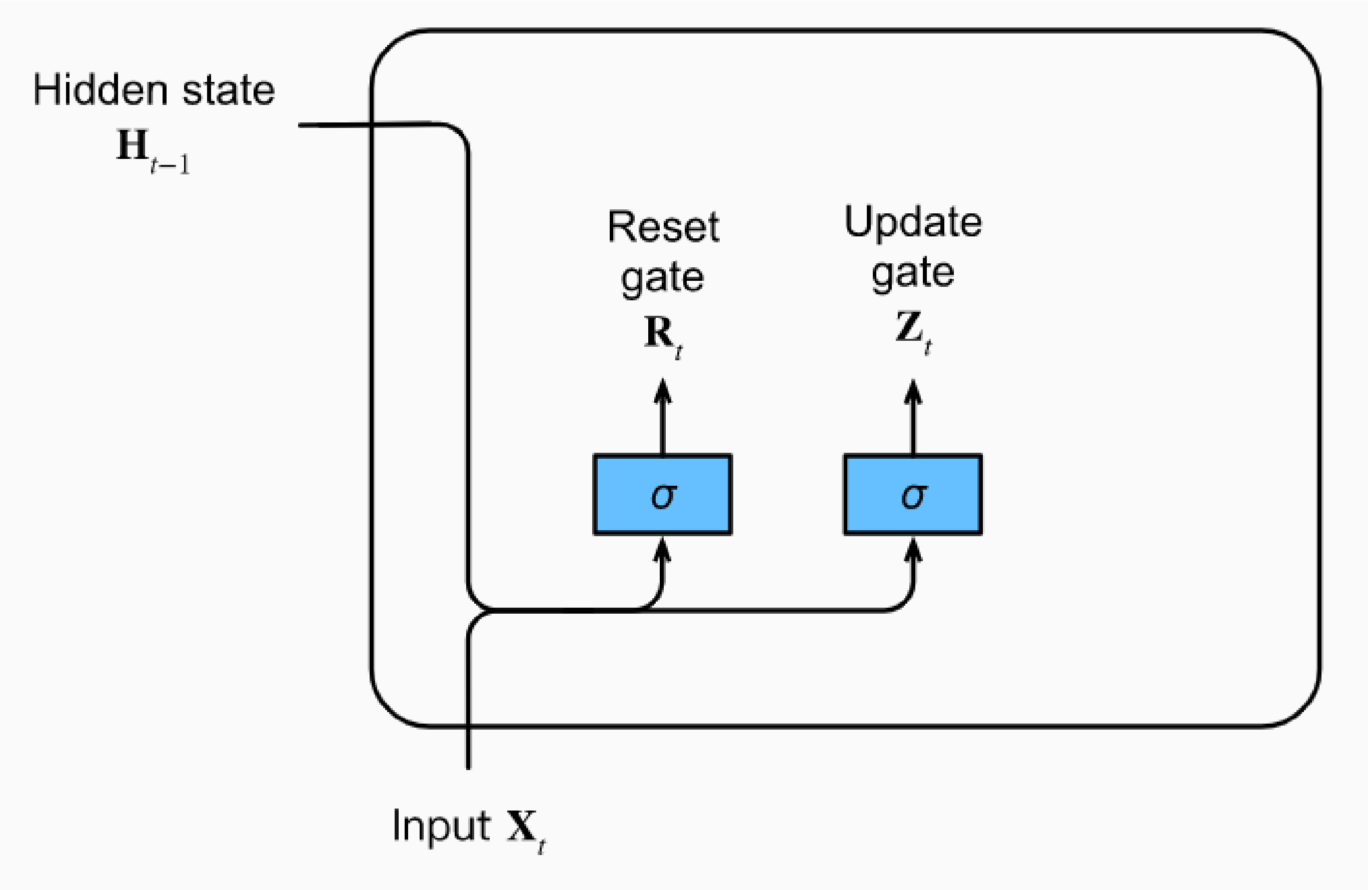
8.5. Implementation of Recurrent Neural Networks from Scratch — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Modern RNNs

Gated Recurrent Units (GRU)

- Gating the information from the past
- An early observation is sometimes very crucial, but sometimes has no relevant information to current time steps.
- We might want to control whether a hidden state should be updated, or when a hidden state should be reset

Gated Recurrent Units (GRU)

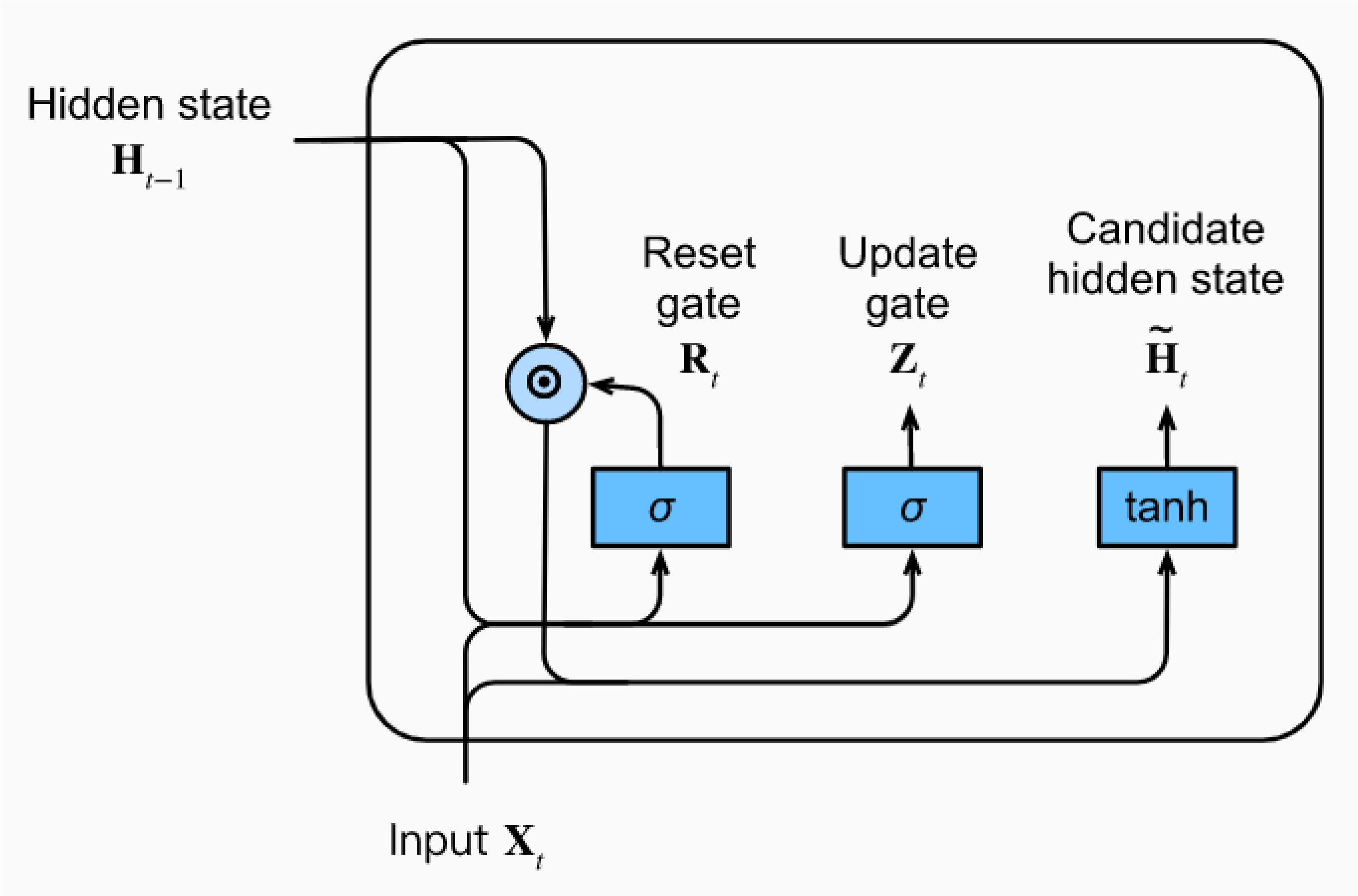


$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1})$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1})$$

[9.1. Gated Recurrent Units (GRU) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

Gated Recurrent Units (GRU)



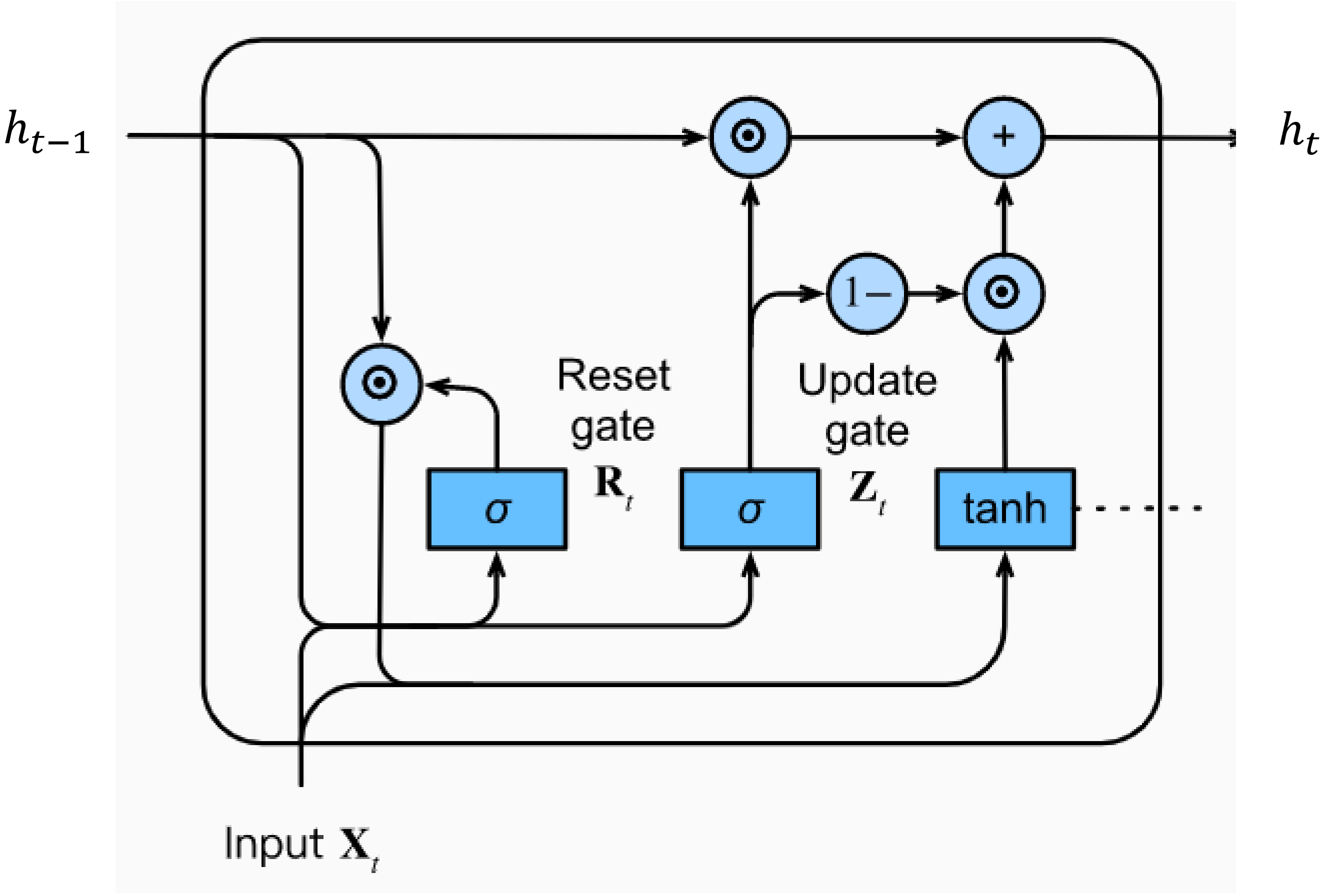
$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1})$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1})$$

$$\widehat{h_t} = \tanh\left(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1})\right)$$

9.1. Gated Recurrent Units (GRU) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Gated Recurrent Units (GRU)



$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1})$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1})$$

$$\hat{h}_t = \tanh\left(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1})\right)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t)\hat{h}_t$$

9.1. Gated Recurrent Units (GRU) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Implementation

```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid((X @ W_xz) + (H @ W_hz) + b_z)
        R = torch.sigmoid((X @ W_xr) + (H @ W_hr) + b_r)
        H_tilda = torch.tanh((X @ W_xh) + ((R * H) @ W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = H @ W_hq + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

9.1. Gated Recurrent Units (GRU) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

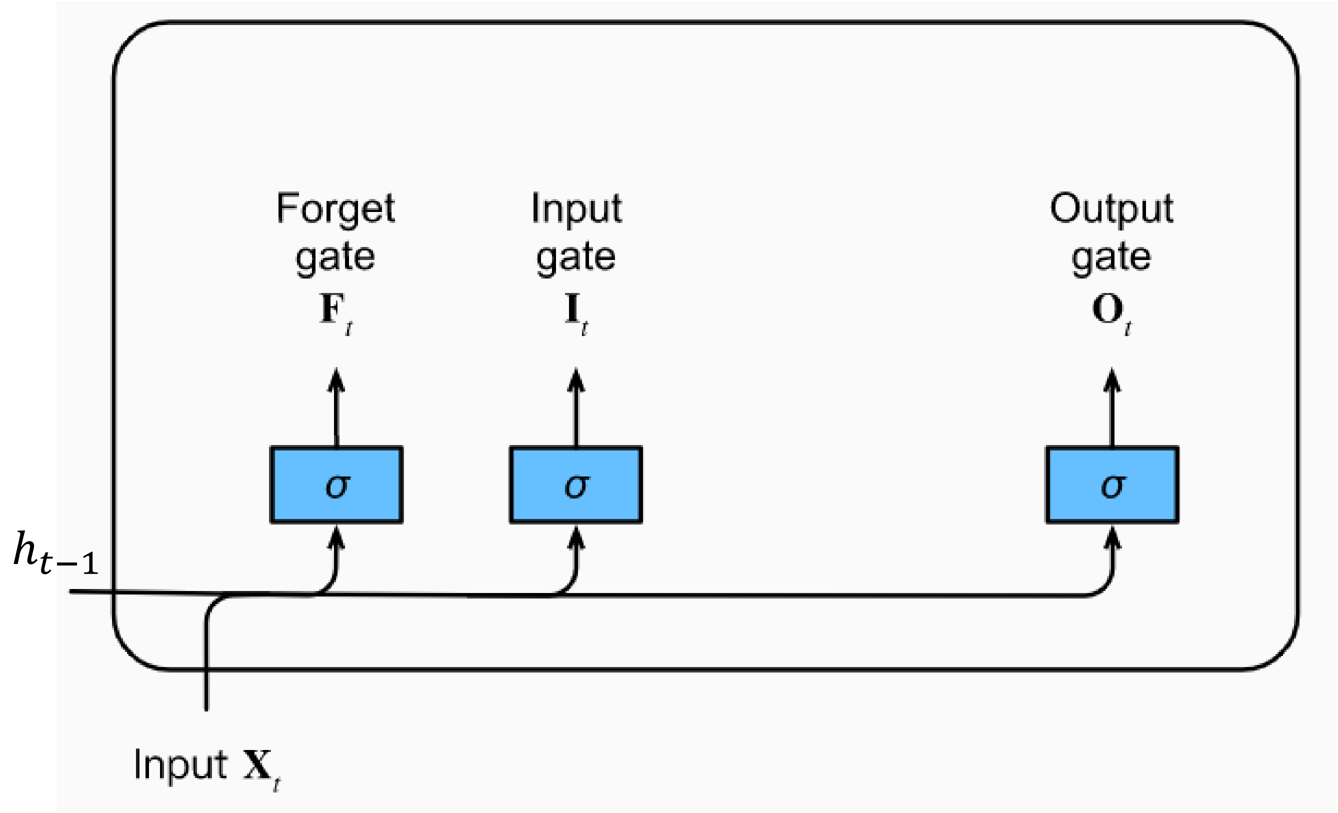
Long Short-Term Memory (LSTM)

- Introducing “memory cell”
  - Hidden states
- Output gate – to read from the memory cell
- Input gate – to write to the memory cell
- Forget gate – to reset the memory cell

[9.2. Long Short-Term Memory (LSTM) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

Long Short-Term Memory (LSTM)

- Gates



$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1})$$

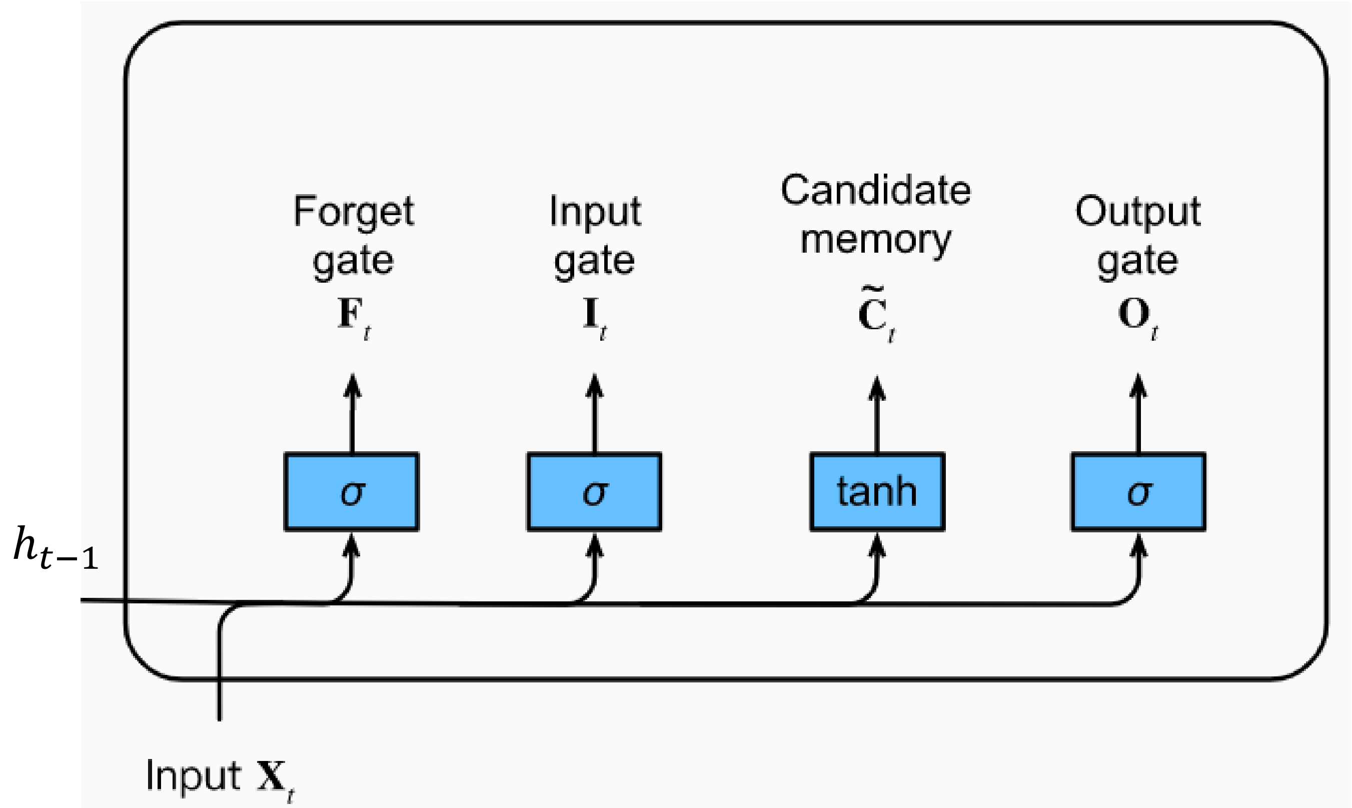
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1})$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1})$$

9.2. Long Short-Term Memory (LSTM) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Long Short-Term Memory (LSTM)

- Candidate memory cell



$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1})$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1}\right)$$

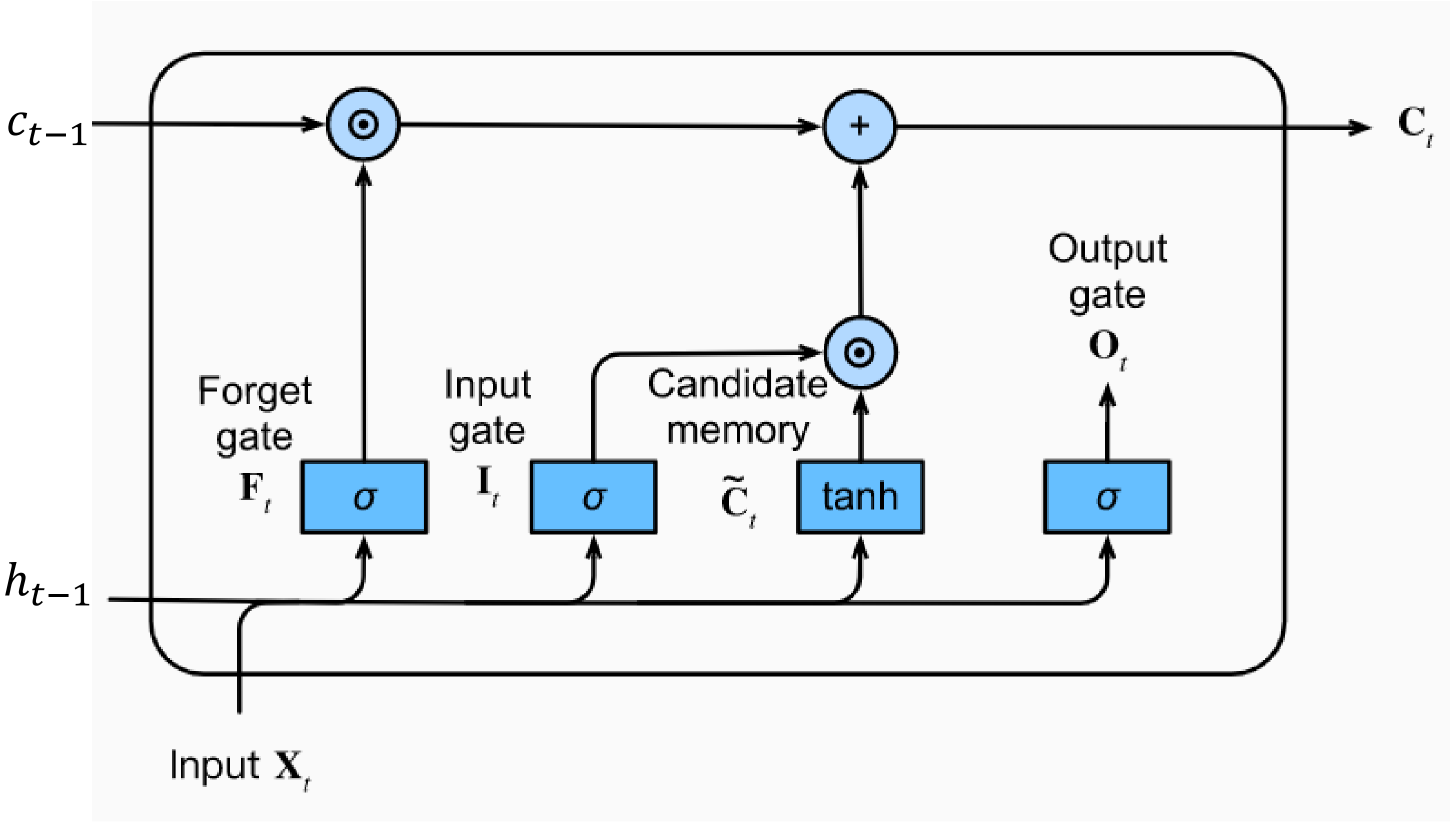
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1})$$

$$\hat{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1})$$

9.2. Long Short-Term Memory (LSTM) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Long Short-Term Memory (LSTM)

- Memory cell
  - If the forget gate is 1 and input gate is 0, then the memory will be saved over time
  - May partly resolve vanishing gradient problem



$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1})$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1}\right)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1})$$

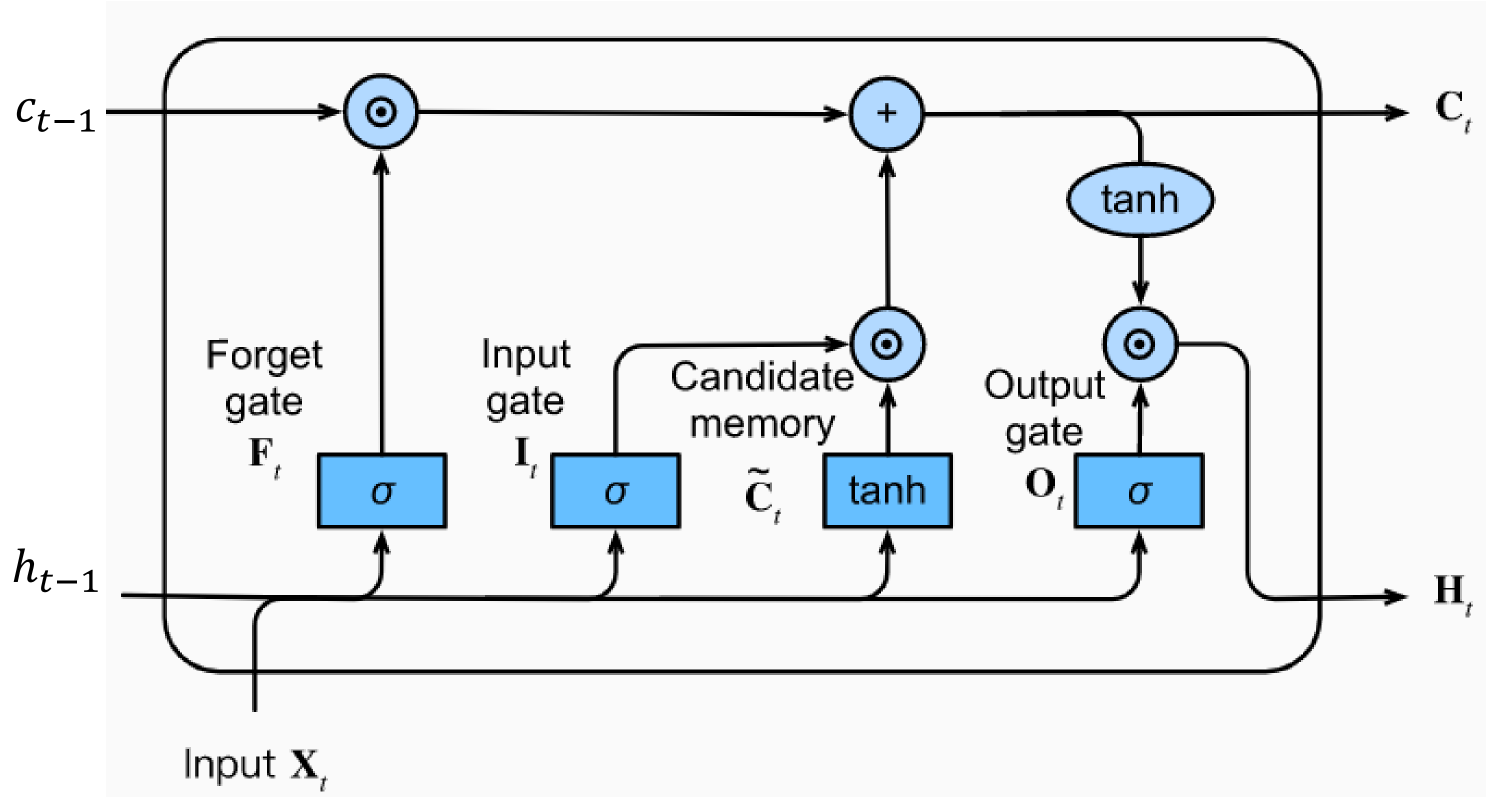
$$\hat{c_t} = \tanh(W_{xc}x_t + W_{hc}h_{t-1})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c_t}$$

[9.2. Long Short-Term Memory (LSTM) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

Long Short-Term Memory (LSTM)

- Hidden states



$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1})$$

$$f_t = \sigma\big(W_{xf}x_t + W_{hf}h_{t-1}\big)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1})$$

$$\hat{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

$$h_t = o_t \odot \tanh(c_t)$$

[9.2. Long Short-Term Memory (LSTM) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

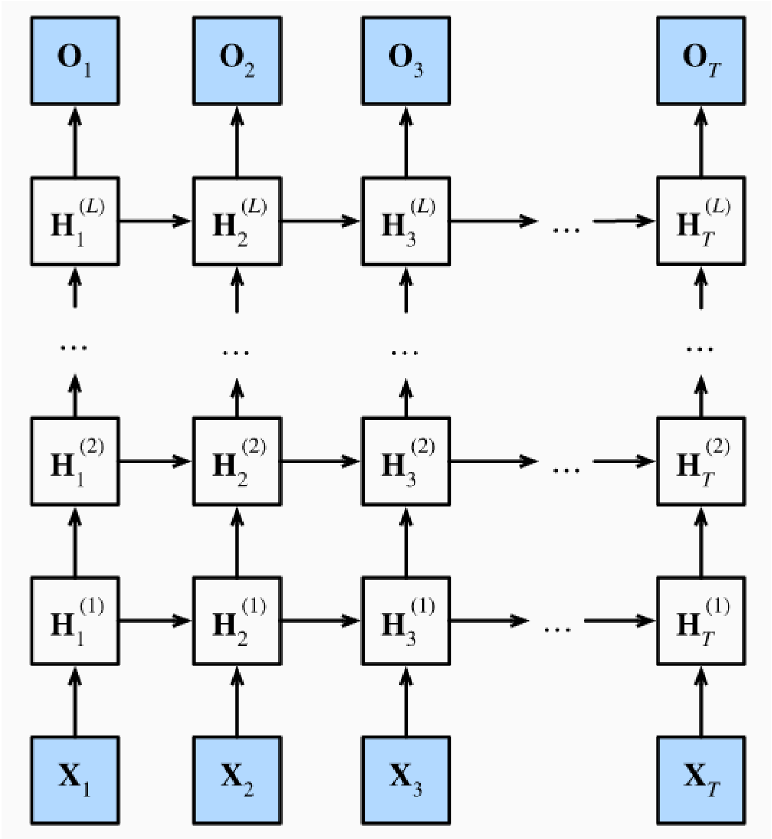
Implementation

```
def lstm(inputs, state, params):
    [
        W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
        W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid((X @ W_xi) + (H @ W_hi) + b_i)
        F = torch.sigmoid((X @ W_xf) + (H @ W_hf) + b_f)
        O = torch.sigmoid((X @ W_xo) + (H @ W_ho) + b_o)
        C_tilda = torch.tanh((X @ W_xc) + (H @ W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * torch.tanh(C)
        Y = (H @ W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H, C)
```

9.2. Long Short-Term Memory (LSTM) — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

Deep RNNs

- Stacking up multiple rnn units



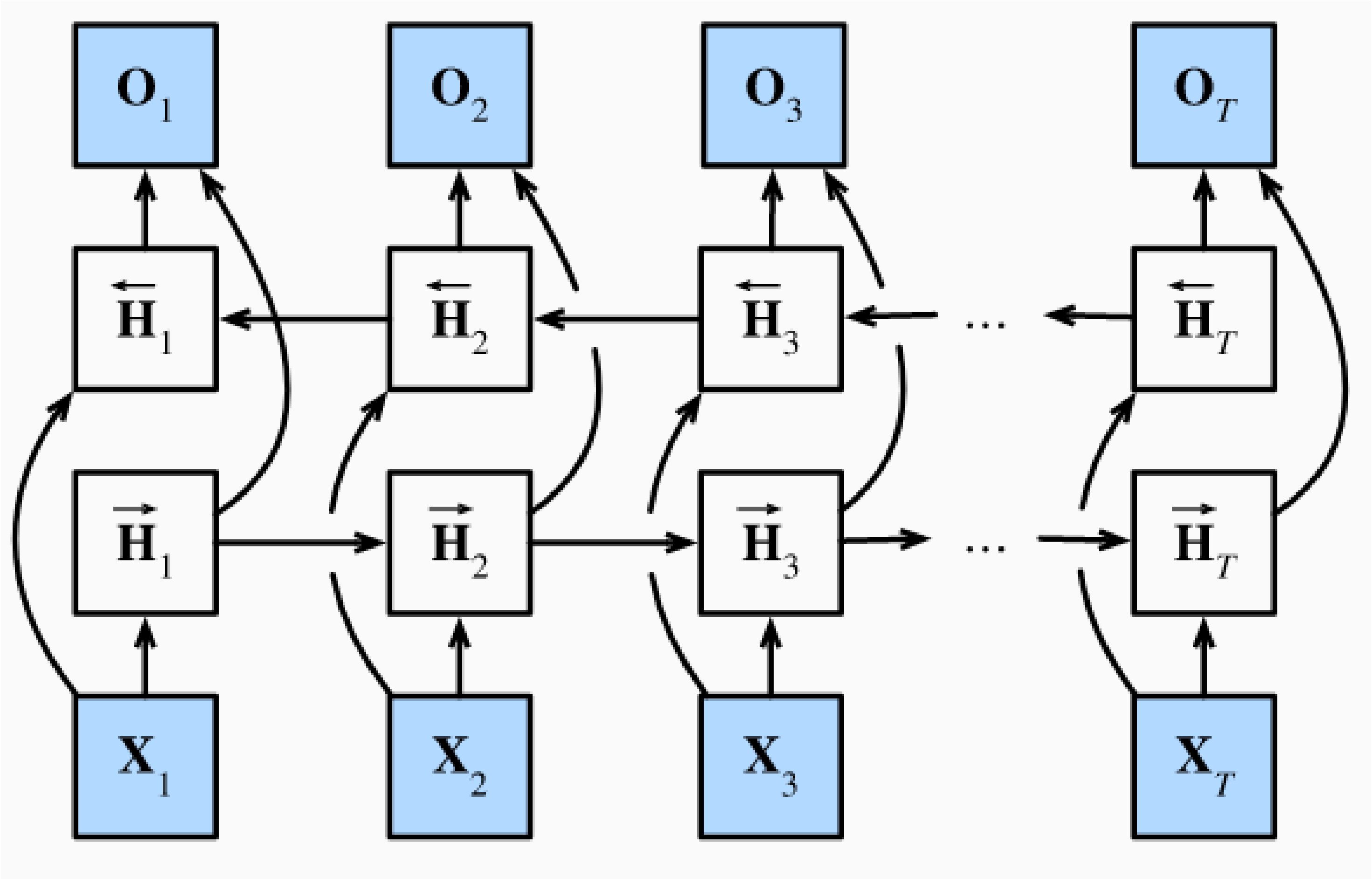
[9.3. Deep Recurrent Neural Networks — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

Bidirectional RNNs

- We might need both past and future information to predict at current time steps
  - E.g. I am _____ hungry and I can eat half a pig

9.4. Bidirectional Recurrent Neural Networks — Dive into Deep Learning 0.17.0 documentation (d2l.ai)

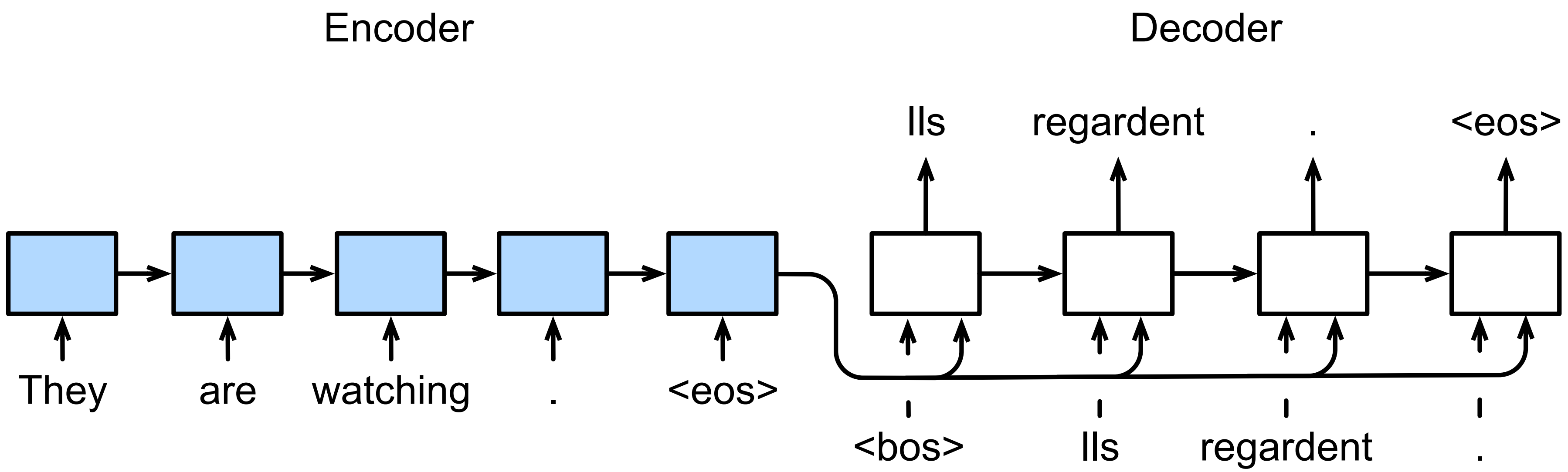
Bidirectional RNNs



[9.4. Bidirectional Recurrent Neural Networks — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]

Sequence to Sequence

- Variable-length input sequence and variable-length output sequence
  - E.g. Machine Translation
- Encoder-decoder based architecture



[9.4. Bidirectional Recurrent Neural Networks — Dive into Deep Learning 0.17.0 documentation (d2l.ai)]