Practical Machine Learning

Workshop 4.
Recurrent Neural Networks (RNN), LSTM, GRU, and Seq-to-seq & Attention Mechanism

Dr. Suyong Eum

Recurrent Neural Networks (RNN)

Some interesting applications

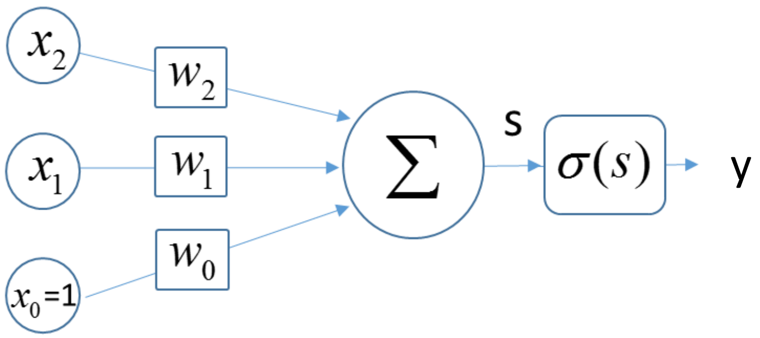
1. Music composition
   - http://people.idsia.ch/~juergen/blues/

2. Writing a poem
   - https://github.com/dvictor/lstm-poetry

A butterfly in the sun
Just because I know that I should leave this heart for you
You said I was falling apart
I wish I were you
I wanted you to know how I feel
I could have settled it all
It's time to go and do it big and you can be my side
I can't believe it when I see you
I'm lost in the world and I can't see you cry
I'm asking you to love me then let me go
I can't stop this way

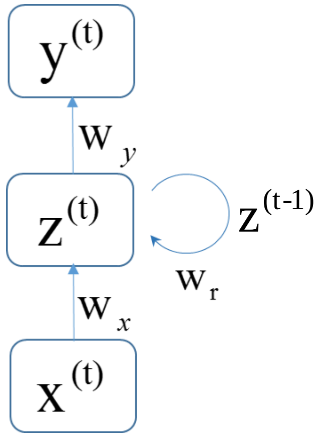
Recurrent Neural Network (RNN)

- Feed forward neural networks (e.g., NN)
  - Temporal independency
  - Fixed length input



$$y = \sigma(wx + b)$$

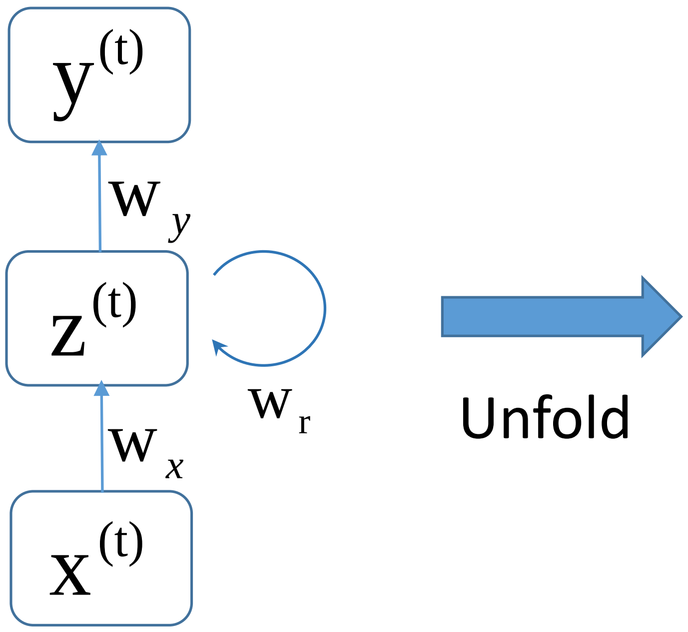
- Recurrent Neural Networks
  - Temporal dependencies
  - Variable sequence length



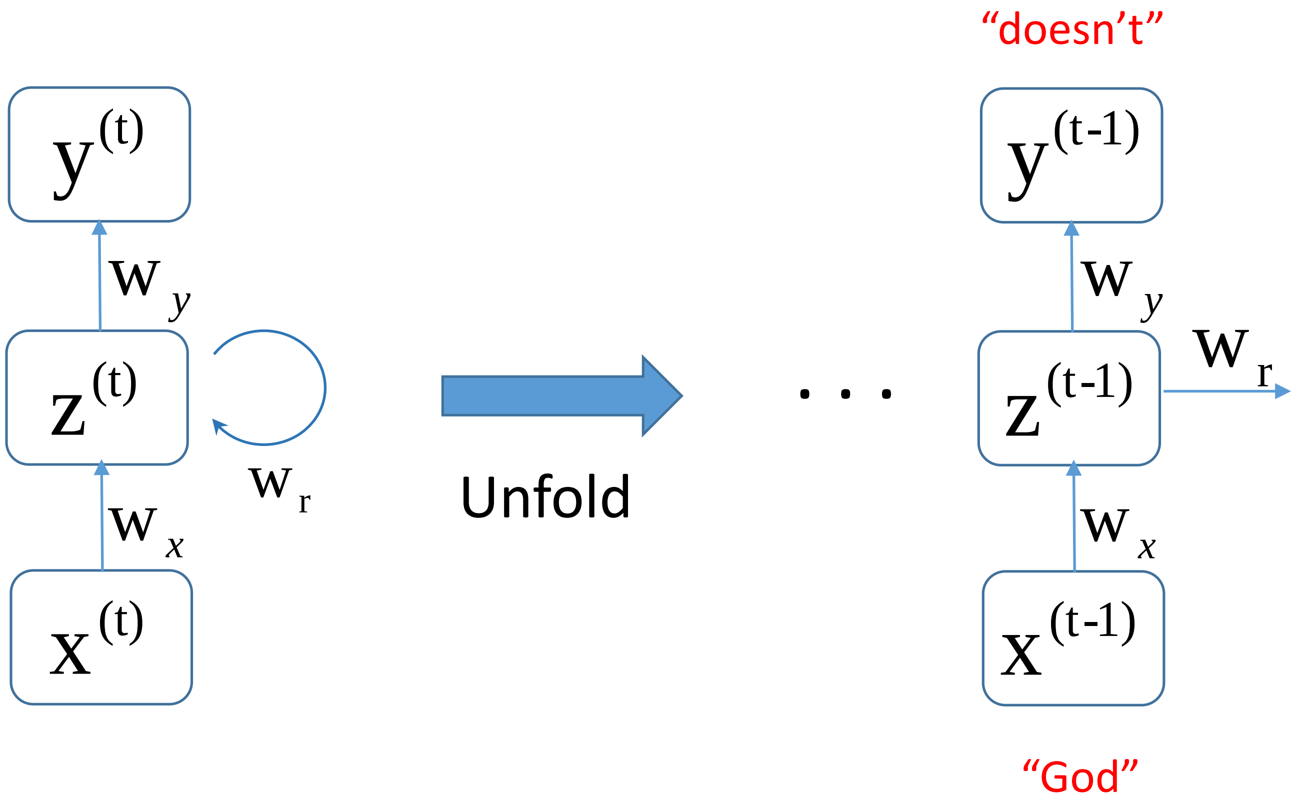
$$z^t = \sigma(w_x x^t + w_r z^{t-1} + b_z)$$

$$y^t = \sigma(w_y z^t + b_y)$$

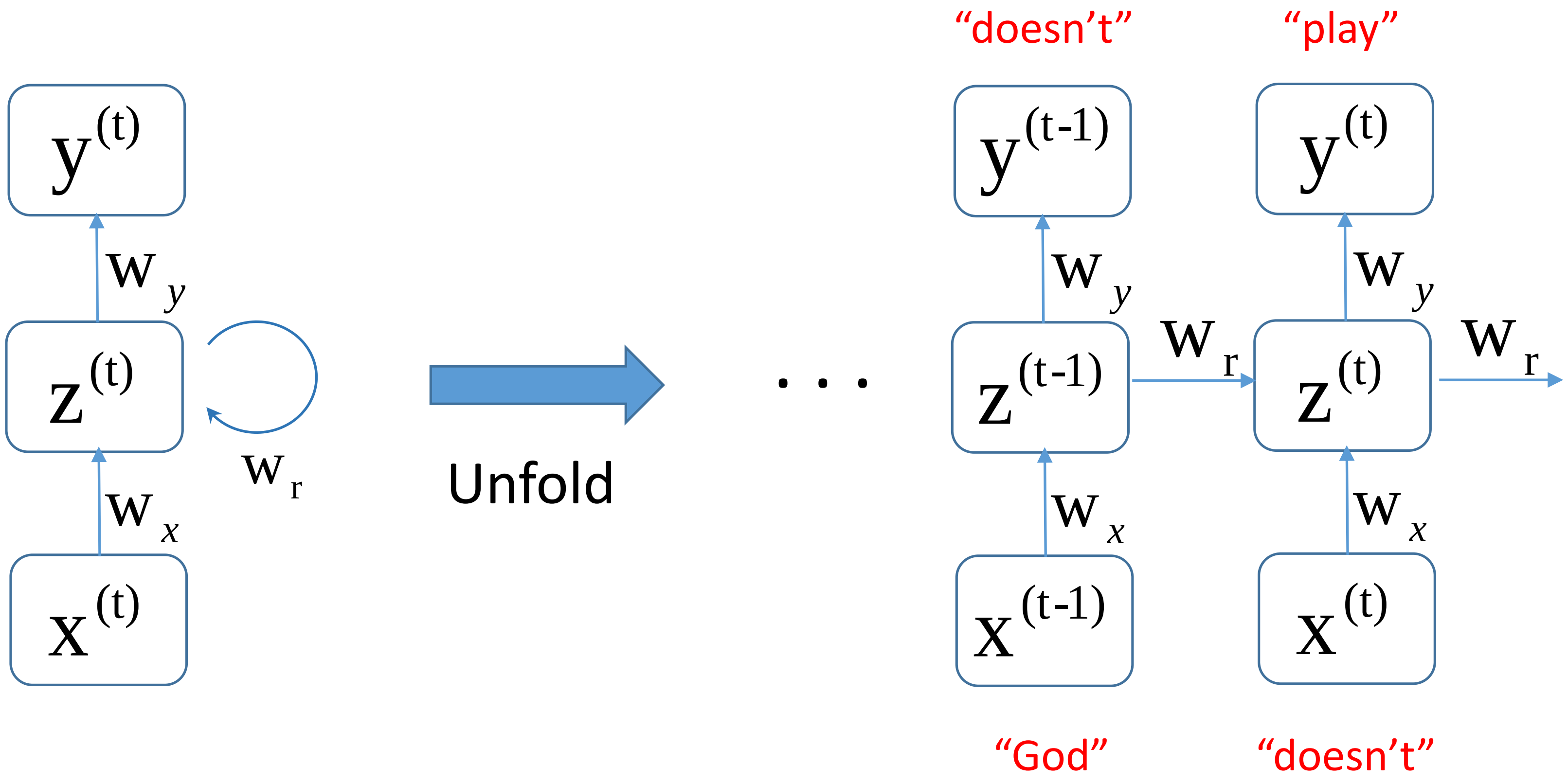
Recurrent Neural Network (RNN)



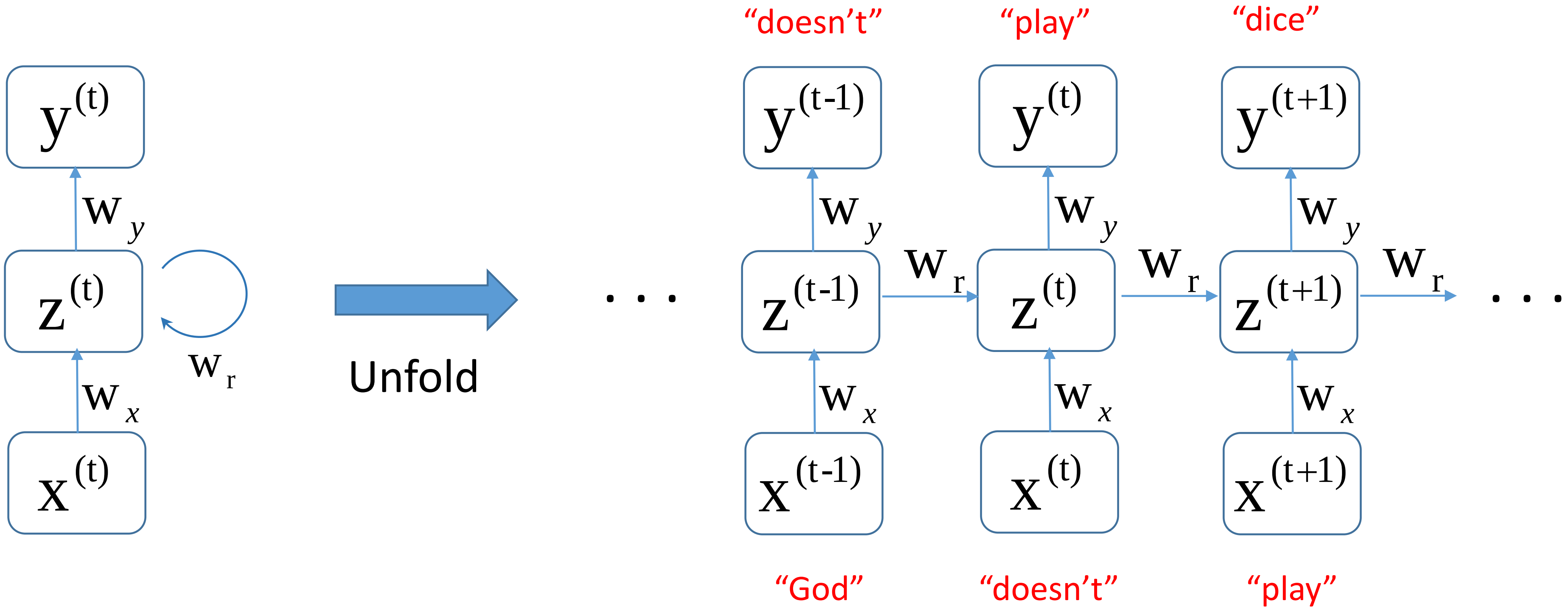
Recurrent Neural Network (RNN)



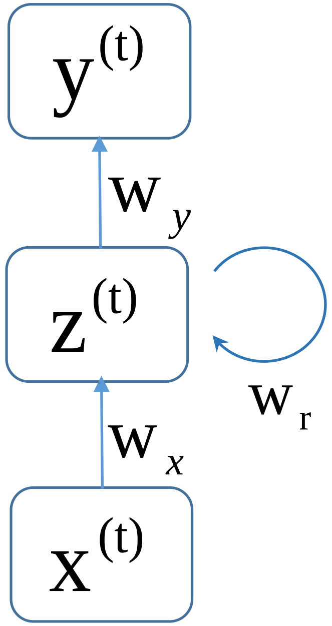
Recurrent Neural Network (RNN)



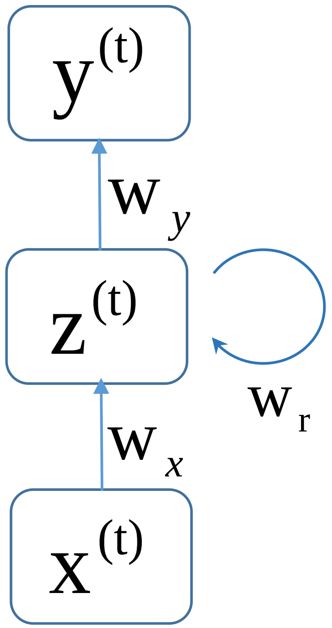
Recurrent Neural Network (RNN)

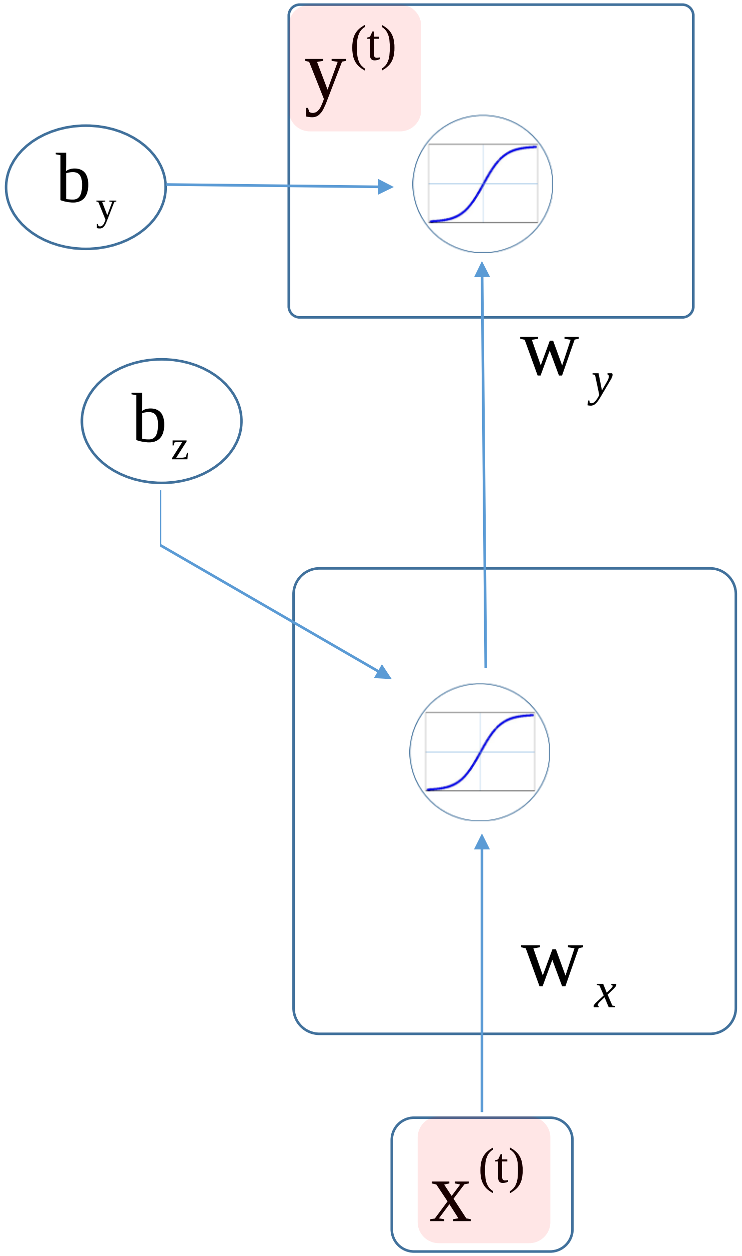


Recurrent Neural Network (RNN): inner structure

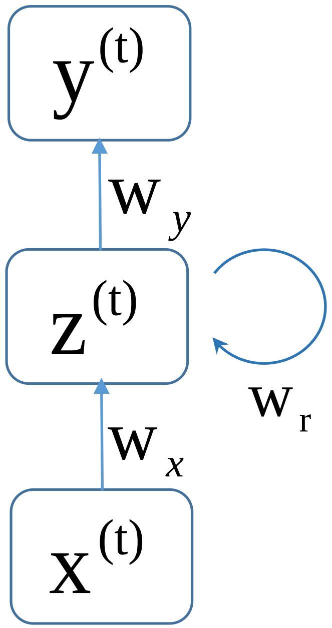


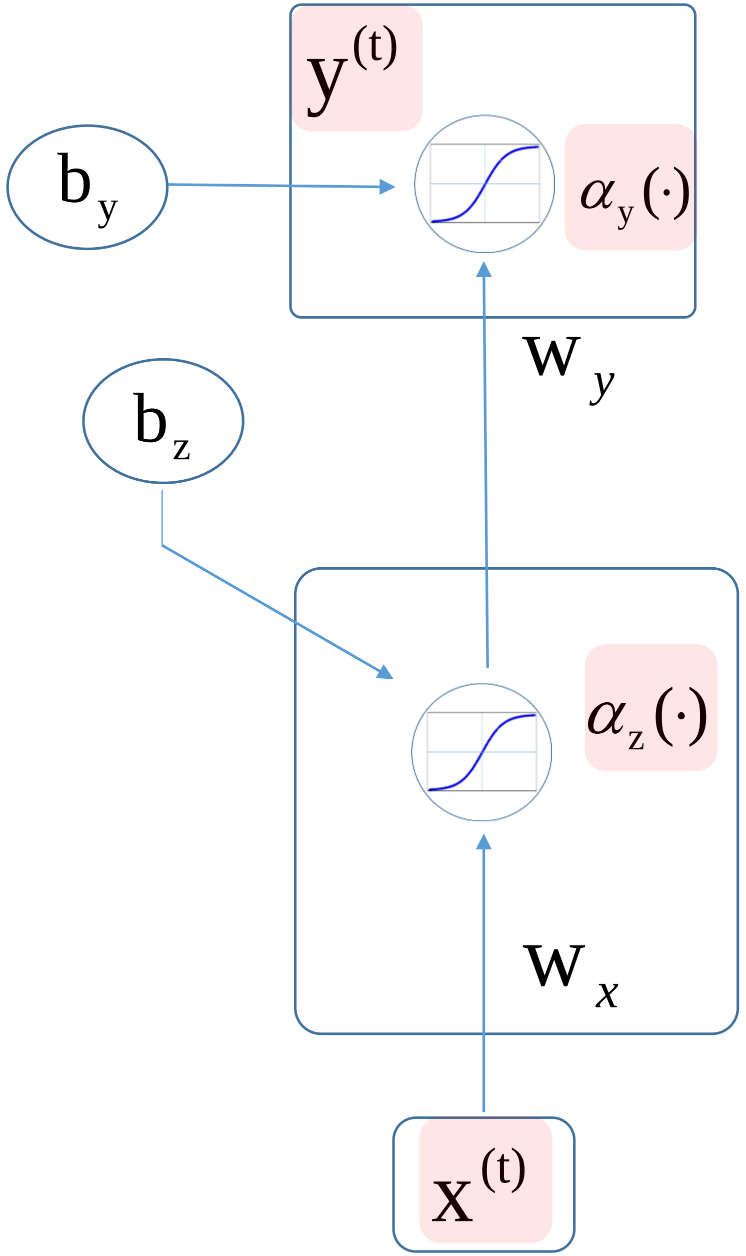
Recurrent Neural Network (RNN): inner structure





Recurrent Neural Network (RNN): inner structure

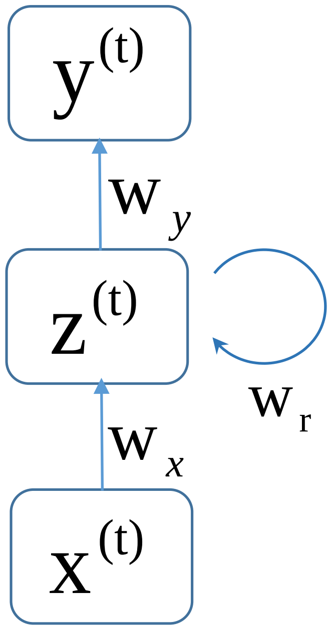


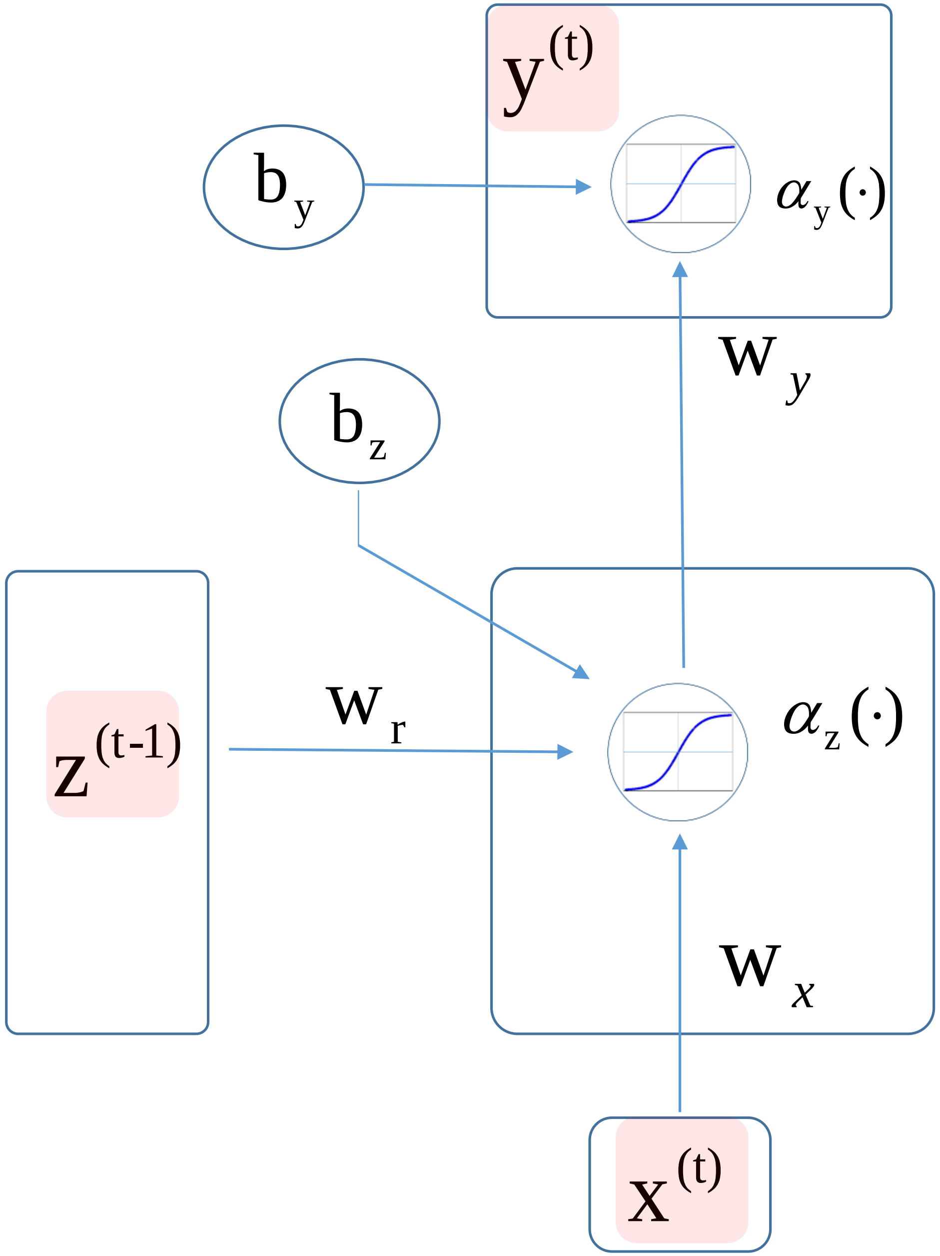


$\alpha_y(\cdot): soft\max$

$\alpha_z(\cdot): \tanh$

Recurrent Neural Network (RNN): inner structure

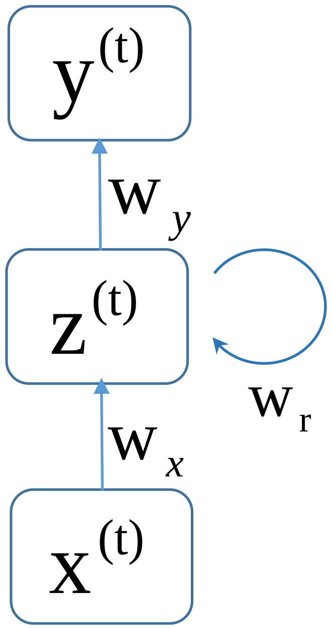


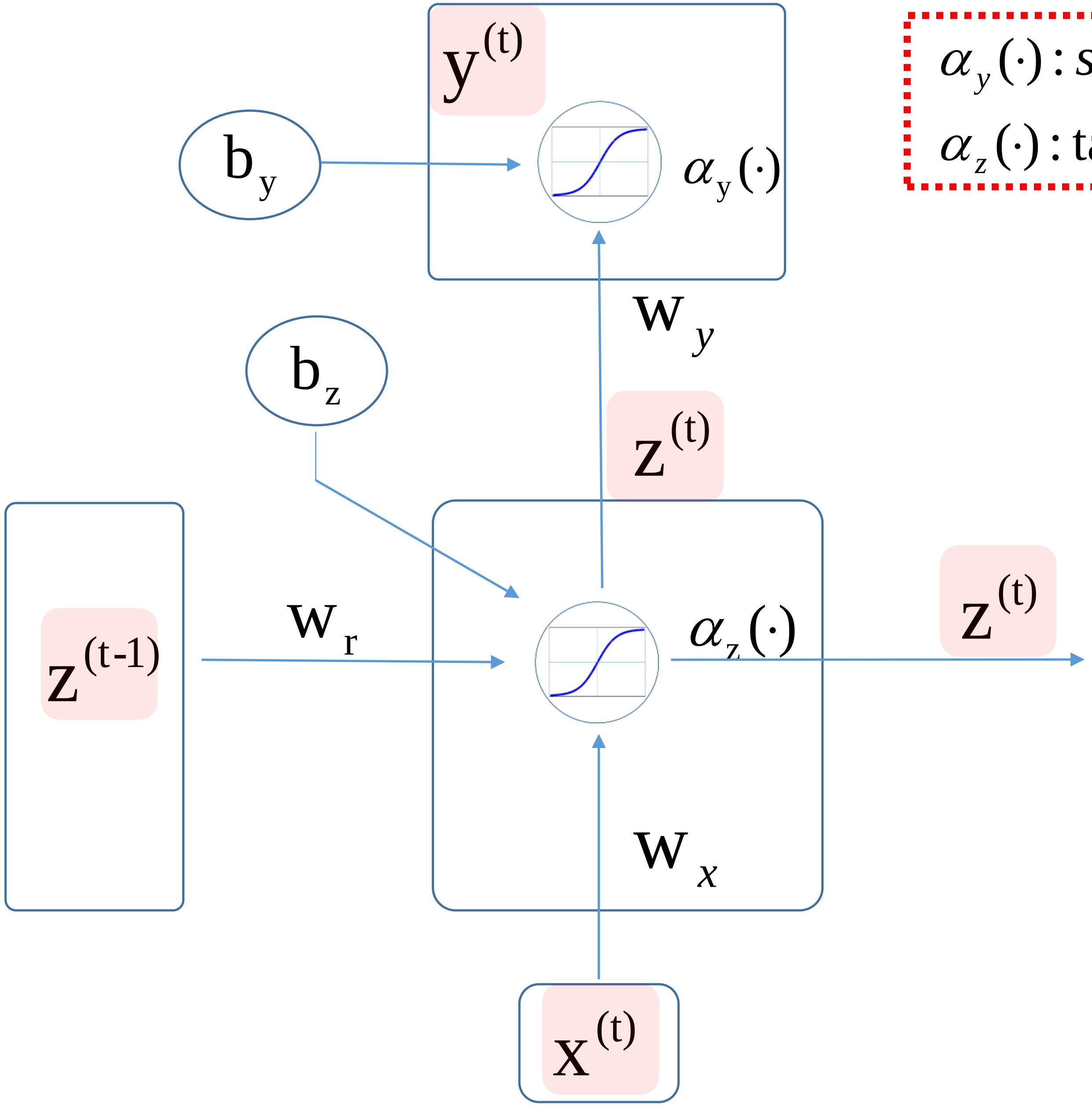


$\alpha_y(\cdot): soft\max$

$\alpha_z(\cdot): \tanh$

Recurrent Neural Network (RNN): inner structure

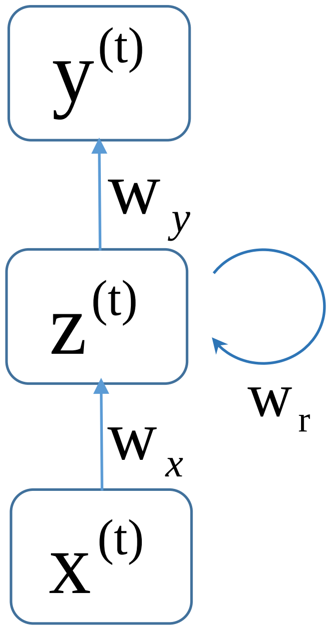


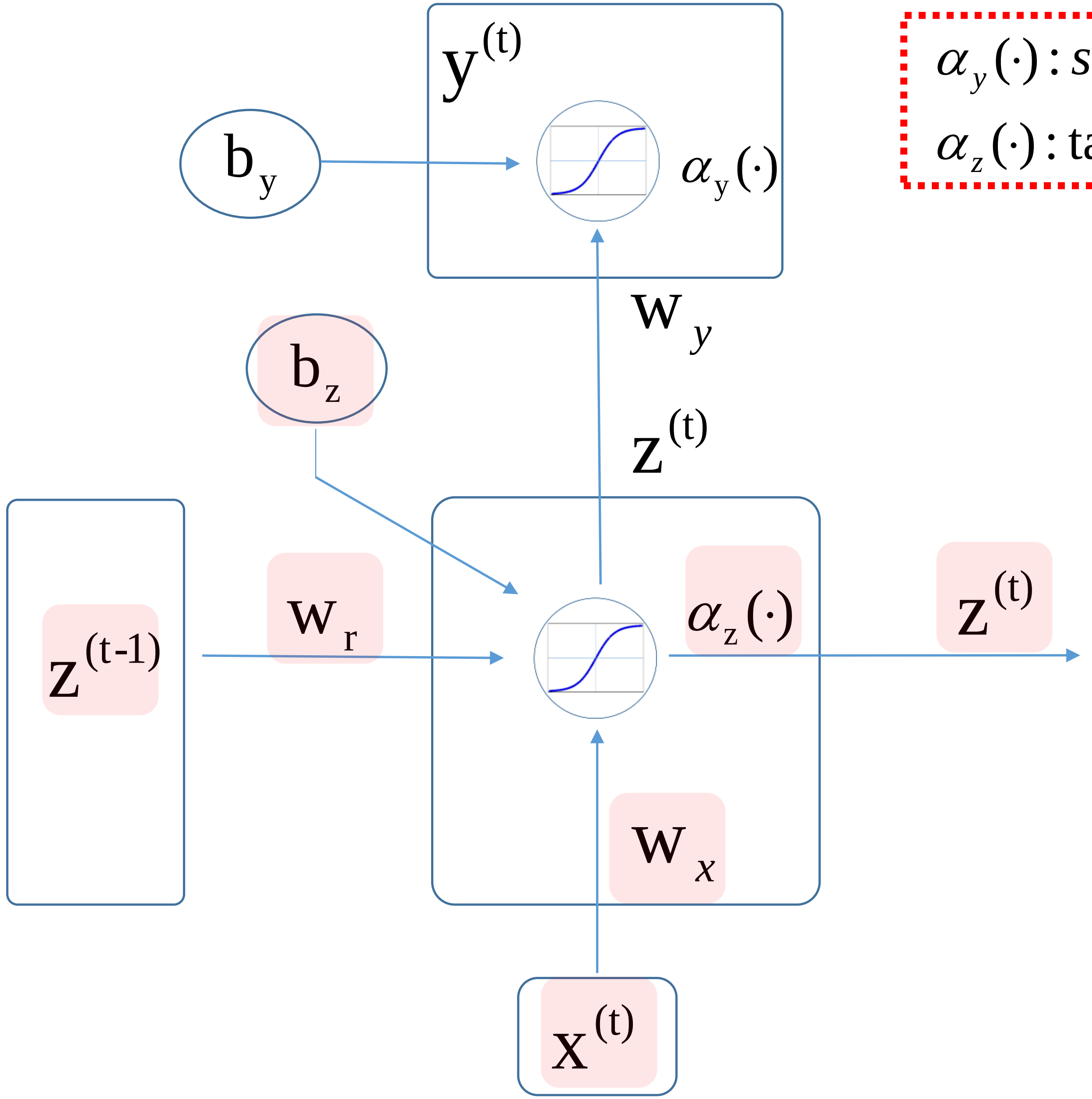


$\alpha_y(\cdot): soft\max$

$\alpha_z(\cdot): \tanh$

Recurrent Neural Network (RNN): inner structure



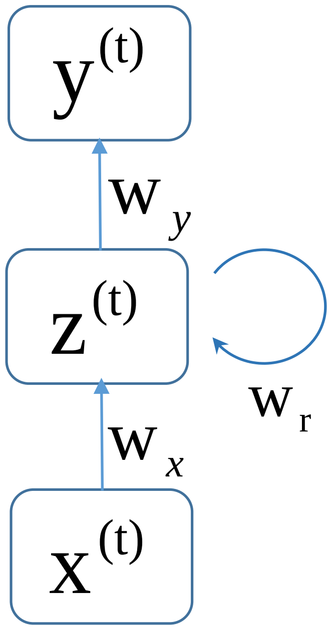


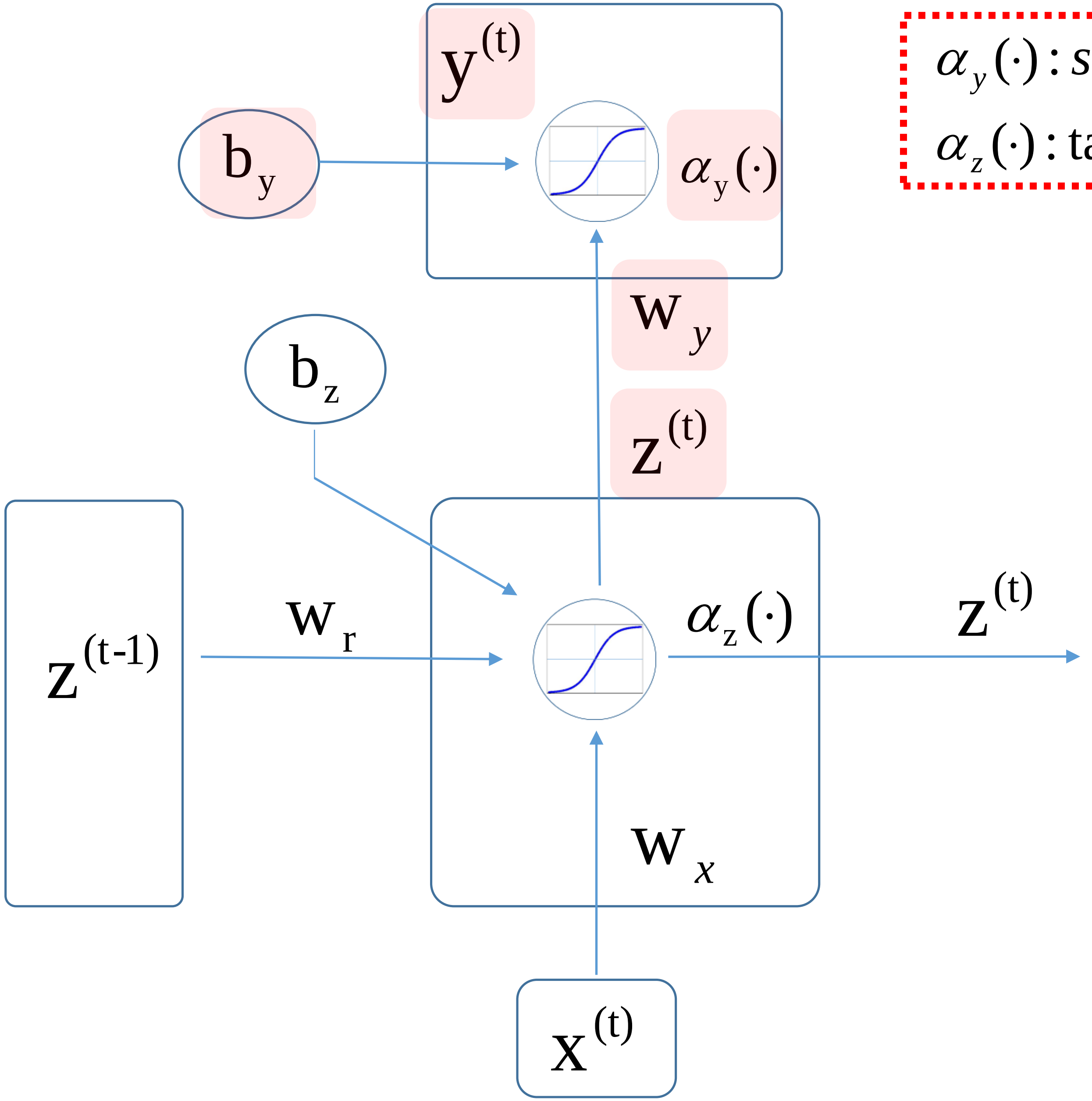
$\alpha_y(\cdot): soft\max$

$\alpha_z(\cdot): \tanh$

$$z^{(t)} = \alpha_z\left(w_x x^{(t)} + w_r z^{(t-1)} + b_z\right)$$

Recurrent Neural Network (RNN): inner structure





$\alpha_y(\cdot): soft\max$

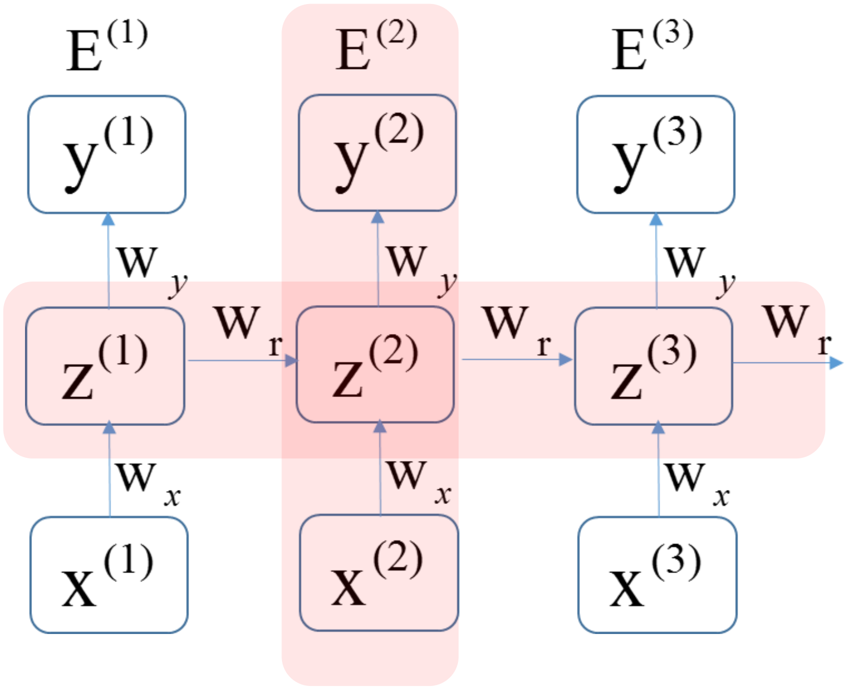
$\alpha_z(\cdot): \tanh$

$$z^{(t)} = \alpha_z\left(w_x x^{(t)} + w_r z^{(t-1)} + b_z\right)$$

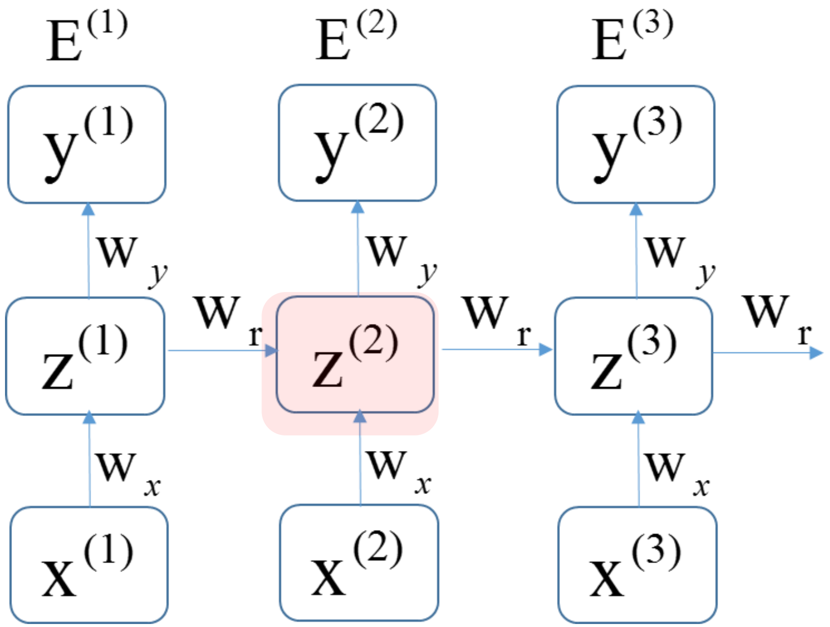
$$y^{(t)} = \alpha_y\left(w_y z^{(t)} + b_y\right)$$

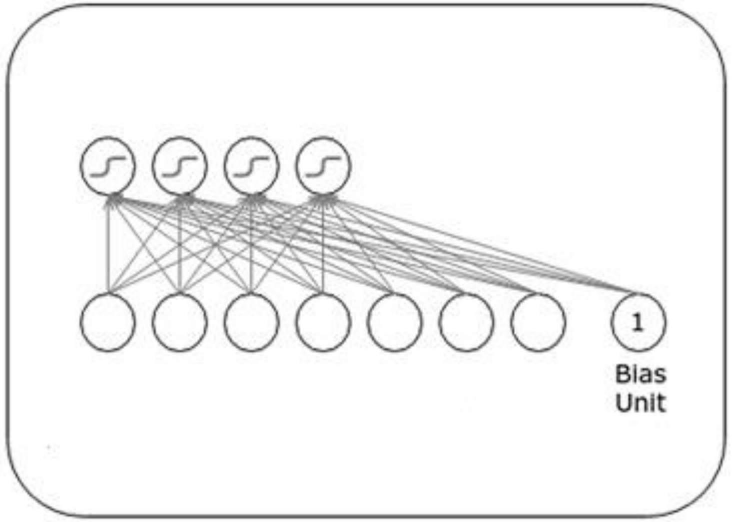
Connectivity of neurons in a vanilla RNN component

Operation of RNN: Connectivity of neurons in vanilla RNN component

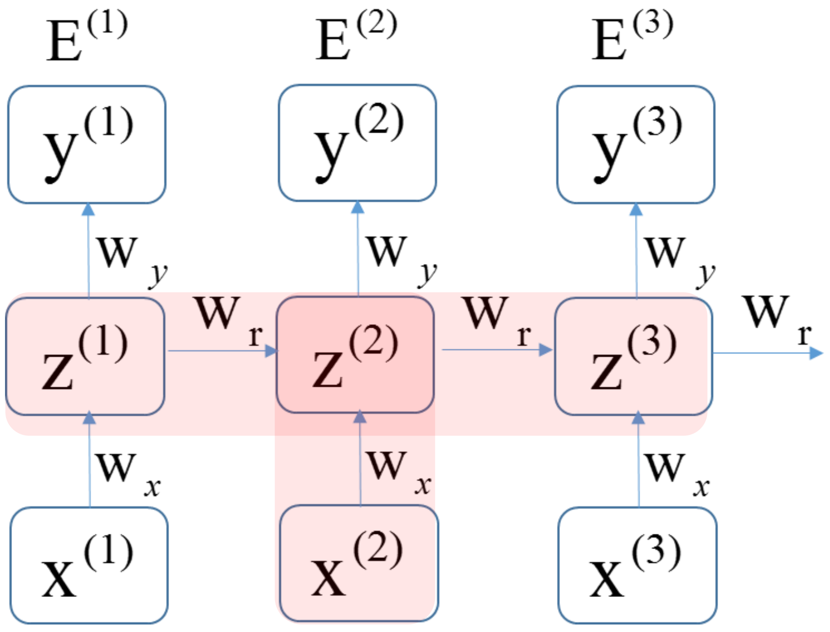


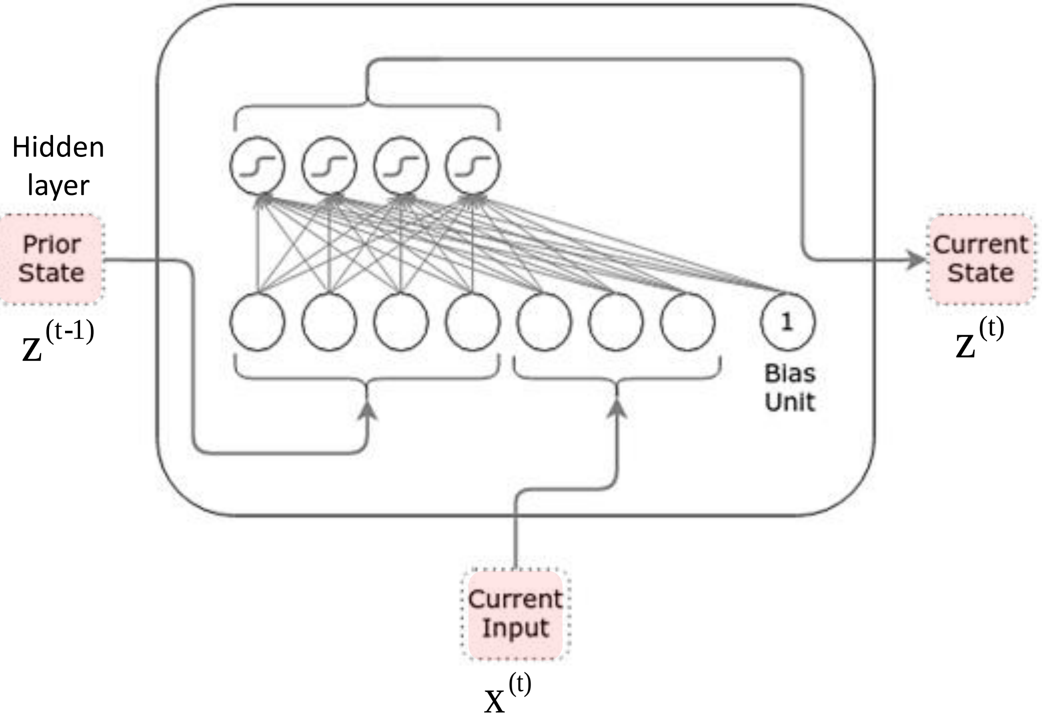
Operation of RNN: Connectivity of neurons in vanilla RNN component



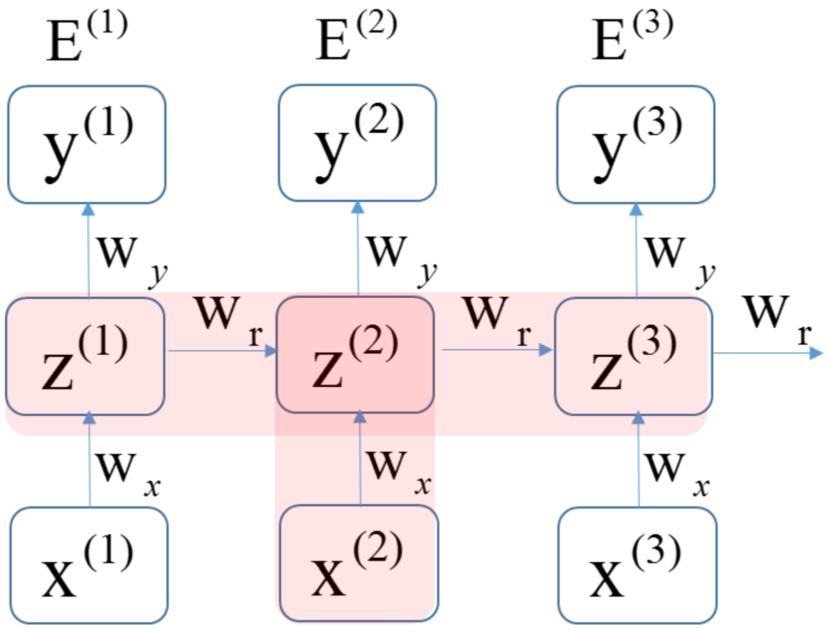


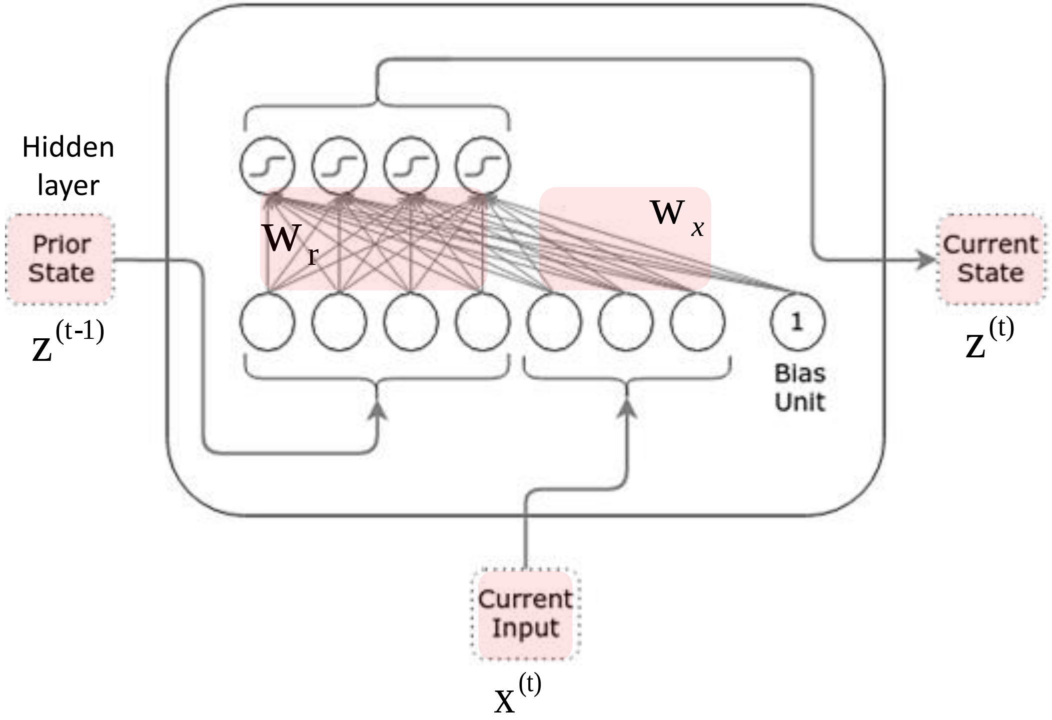
Operation of RNN: Connectivity of neurons in vanilla RNN component



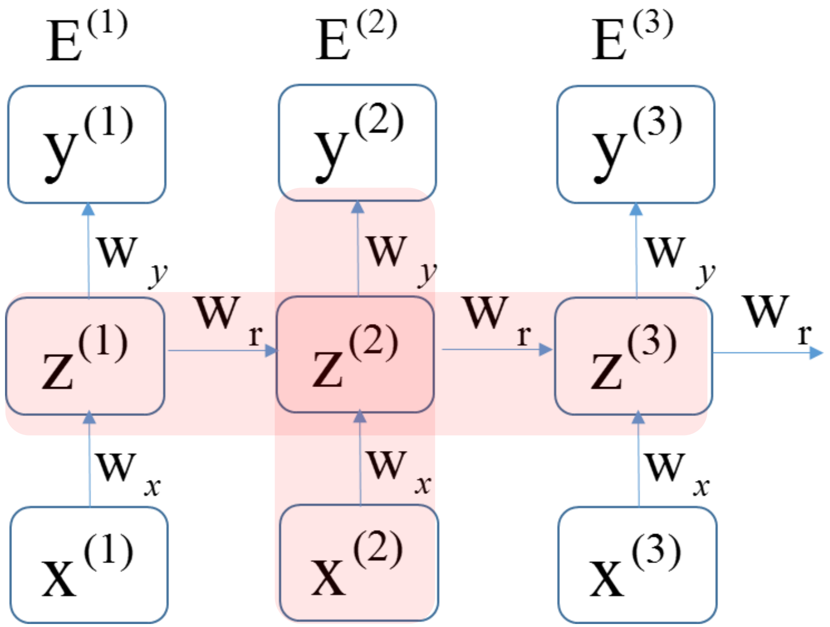


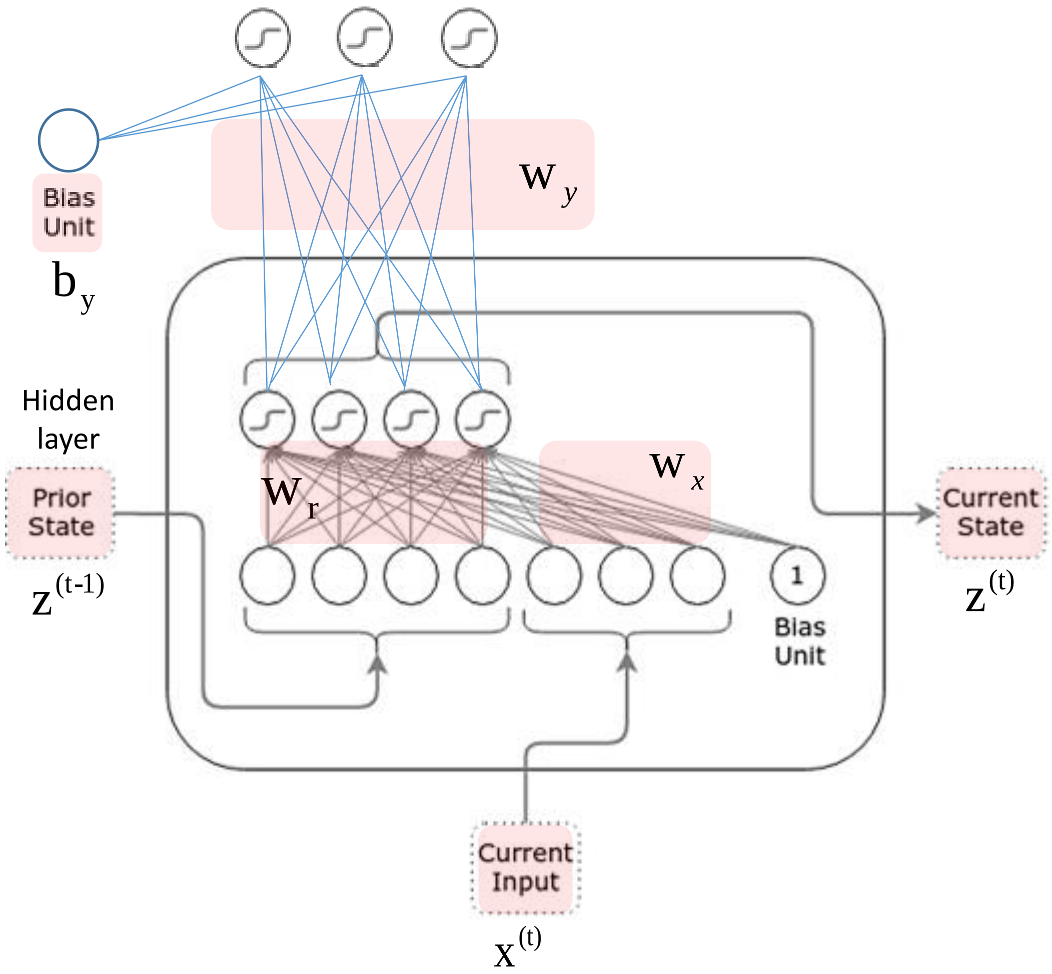
Operation of RNN: Connectivity of neurons in vanilla RNN component



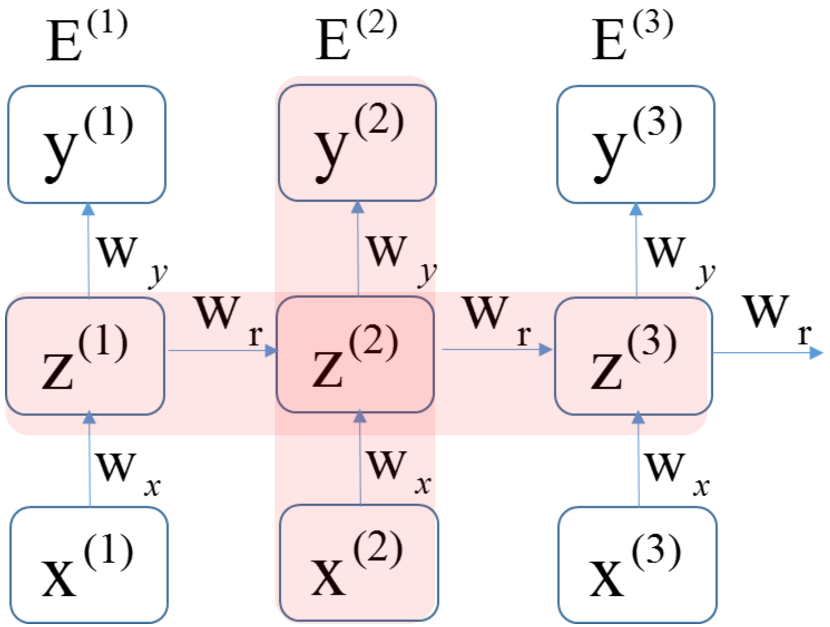


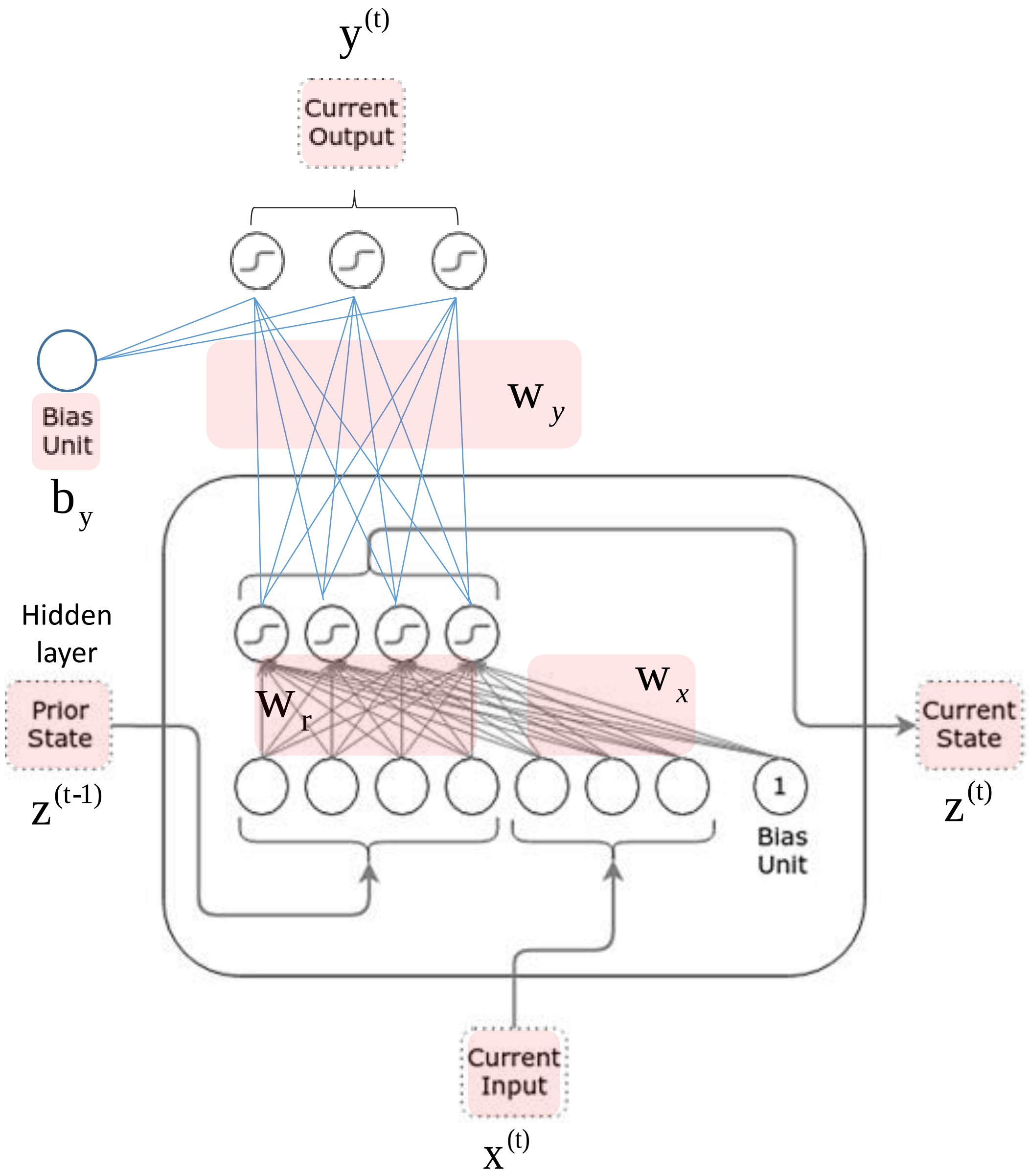
Operation of RNN: Connectivity of neurons in vanilla RNN component



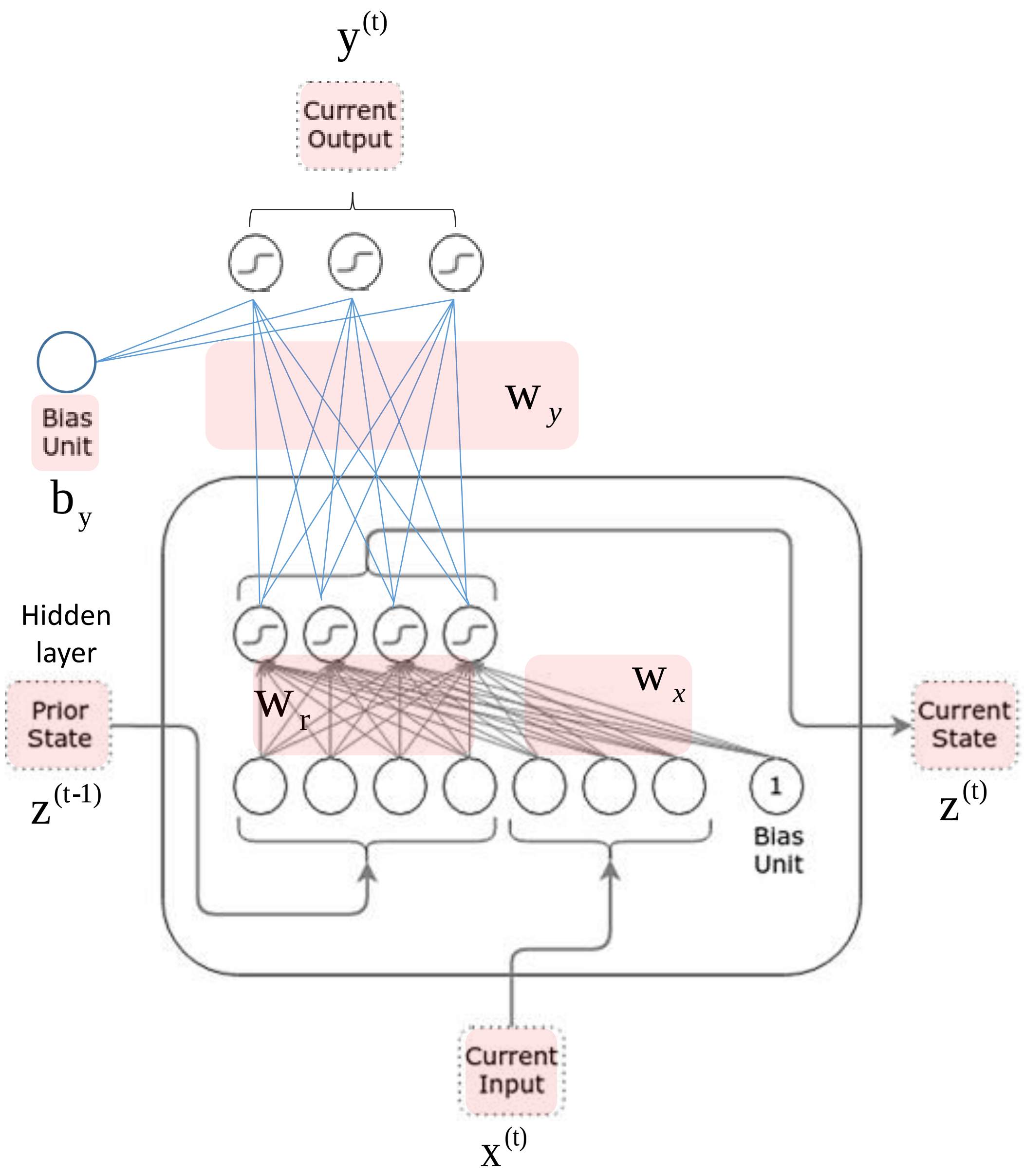


Operation of RNN: Connectivity of neurons in vanilla RNN component





Operation of RNN: Connectivity of neurons in vanilla RNN component



where:

$z^{(t-1)}, z^{(t)} \in R^n$ $\quad W_r \in R^{n\times n}$ $\quad b_z \in R^n$

$x^{(t)} \in R^m$ $\quad W_x \in R^{n\times m}$ $\quad b_y \in R^k$

$y^{(t)} \in R^k$ $\quad W_y \in R^{k\times n}$

$$z^{(t)} = \alpha_z\left(w_x x^{(t)} + w_r z^{(t-1)} + b_z\right)$$

(n x 1) = (n x m) (m x 1) + (n x n) (n x 1) + (n x 1)

$$y^{(t)} = \alpha_y\left(w_y z^{(t)} + b_y\right)$$

(k x 1) = (k x n) (n x 1) + (k x 1)

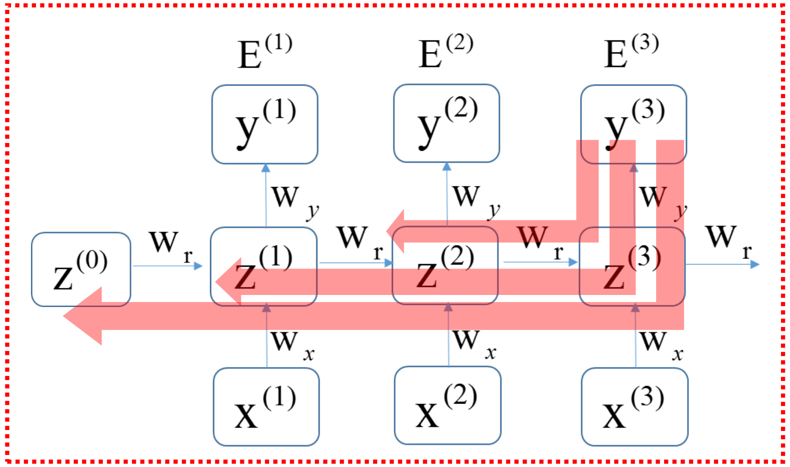
- n: hidden layer size
- m: encoding range (e.g., character level - ASCII: 256)
- k: output size

Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM) network

- *Long Short Term Memory (LSTM) architecture was motivated to overcome the problem: error is not back-propagated properly to the end of RNN architecture.*

RNN using BPTT: Vanishing and exploding problems



$$\sum_{t=1}^{3}\frac{\partial E^{(t)}}{\partial w_r}=\frac{\partial E^{(3)}}{\partial y^{(3)}}\frac{\partial y^{(3)}}{\partial s_y^{(3)}}\frac{\partial s_y^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial s_z^{(3)}}\frac{\partial s_z^{(3)}}{\partial w_r}$$

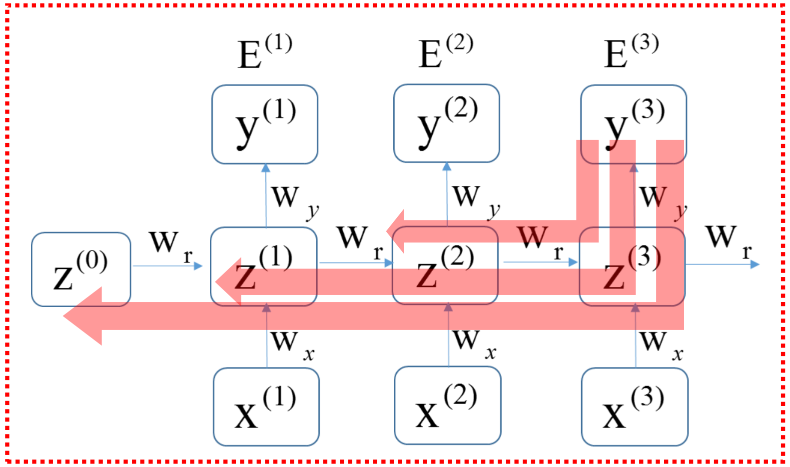
$$+\frac{\partial E^{(3)}}{\partial y^{(3)}}\frac{\partial y^{(3)}}{\partial s_y^{(3)}}\frac{\partial s_y^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial s_z^{(3)}}\frac{\partial s_z^{(3)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial s_z^{(2)}}\frac{\partial s_z^{(2)}}{\partial w_r}$$

$$+\frac{\partial E^{(3)}}{\partial y^{(3)}}\frac{\partial y^{(3)}}{\partial s_y^{(3)}}\frac{\partial s_y^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial s_z^{(3)}}\frac{\partial s_z^{(3)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial s_z^{(2)}}\frac{\partial s_z^{(2)}}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial s_z^{(1)}}\frac{\partial s_z^{(1)}}{\partial w_r}$$

http://www.suyongeum.com/ML/lectures/LectureW10_20180621_print.pdf

- Refer to Slide 12

RNN using BPTT: Vanishing and exploding problems

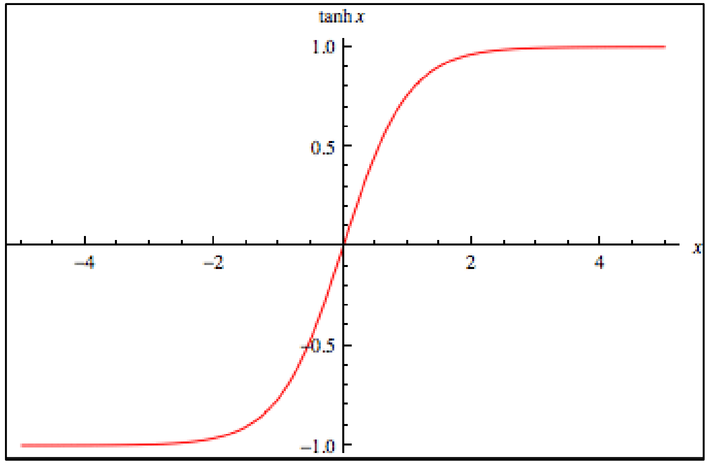


$$\sum_{t=1}^{3}\frac{\partial E^{(t)}}{\partial w_r} = \frac{\partial E^{(3)}}{\partial y^{(3)}}\frac{\partial y^{(3)}}{\partial s_y^{(3)}}\frac{\partial s_y^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial s_z^{(3)}}\frac{\partial s_z^{(3)}}{\partial w_r}$$

$$+\frac{\partial E^{(3)}}{\partial y^{(3)}}\frac{\partial y^{(3)}}{\partial s_y^{(3)}}\frac{\partial s_y^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial s_z^{(3)}}\frac{\partial s_z^{(3)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial s_z^{(2)}}\frac{\partial s_z^{(2)}}{\partial w_r}$$

$$+\frac{\partial E^{(3)}}{\partial y^{(3)}}\frac{\partial y^{(3)}}{\partial s_y^{(3)}}\overbrace{\frac{\partial s_y^{(3)}}{\partial z^{(3)}}}^{W_r}\boxed{\frac{\partial z^{(3)}}{\partial s_z^{(3)}}}\overbrace{\frac{\partial s_z^{(3)}}{\partial z^{(2)}}}^{W_r}\boxed{\frac{\partial z^{(2)}}{\partial s_z^{(2)}}}\overbrace{\frac{\partial s_z^{(2)}}{\partial z^{(1)}}}^{W_r}\boxed{\frac{\partial z^{(1)}}{\partial s_z^{(1)}}}\overbrace{\frac{\partial s_z^{(1)}}{\partial w_r}}^{z^{(0)}}$$

Activation function (tanh)



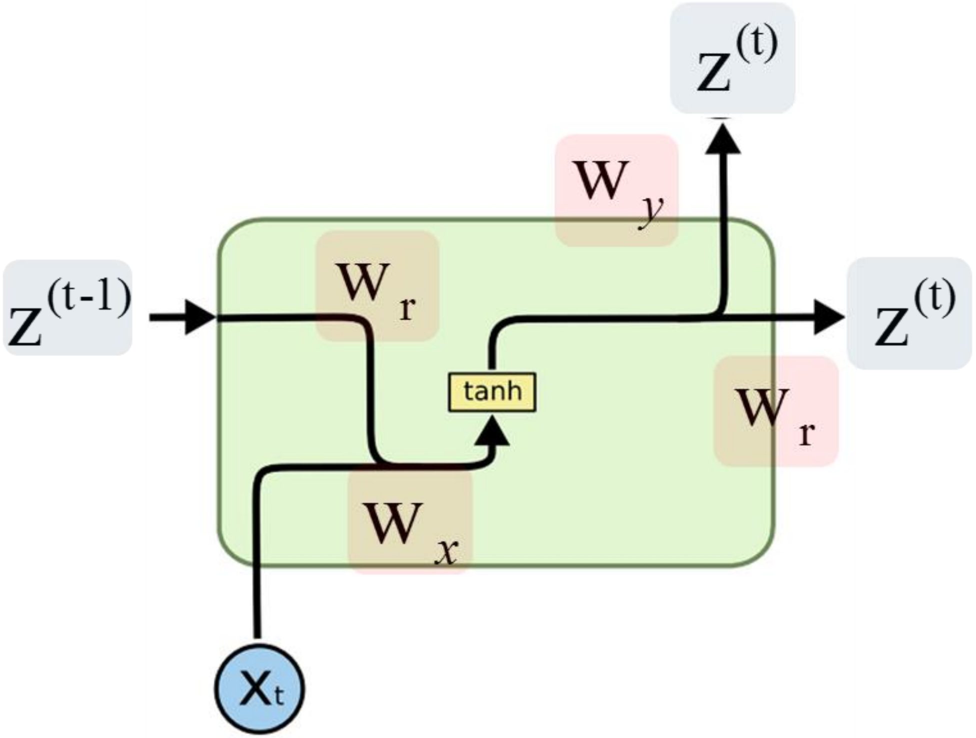
$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{(x)} - e^{(x)}}{e^{(x)} + e^{(x)}} => \sigma(x) \qquad \frac{d\sigma(x)}{dx} = 1-\sigma(x)^2$$

Long Short Term Memory (LSTM) network

- *Long Short Term Memory (LSTM) architecture was motivated to overcome the problem: error is not back-propagated properly to the end of RNN architecture.*

- *Hochreiter and Schmidhuber (1997) proposed the Long Short-Term Memory (LSTM) cell which includes "a memory unit":*
  1) *A cell with a number of components that together act similar to a memory cell.*
  2) *Inside one cell, multiple layers called "gates" are used.*
     1) *Forget gate*
     2) *Input gate*
     3) *Output gate*

Long Short Term Memory (LSTM) network

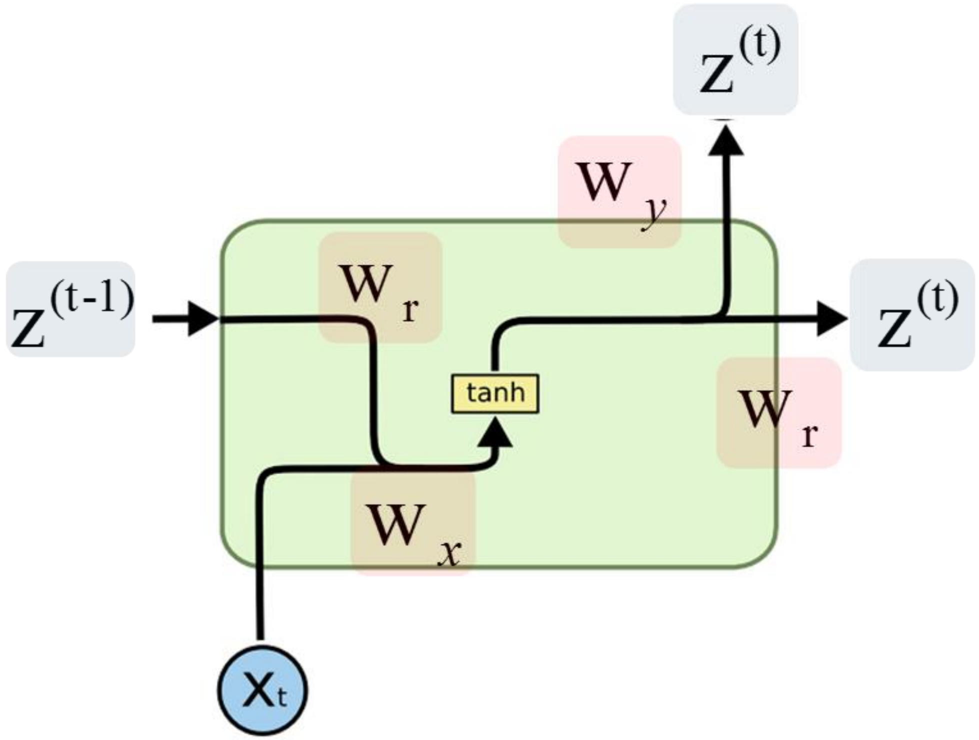


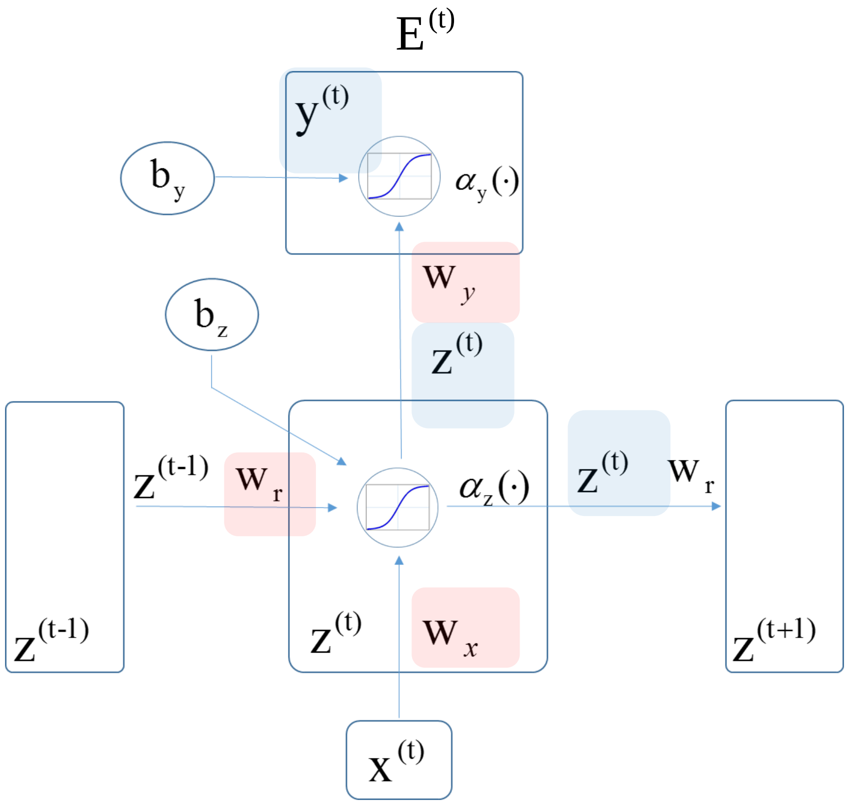
$$z^{(t)} = \alpha_z\left(w_x x^{(t)} + w_r z^{(t-1)} + b_z\right)$$

$$y^{(t)} = \alpha_y\left(w_y z^{(t)} + b_y\right)$$

RNN

Long Short Term Memory (LSTM) network



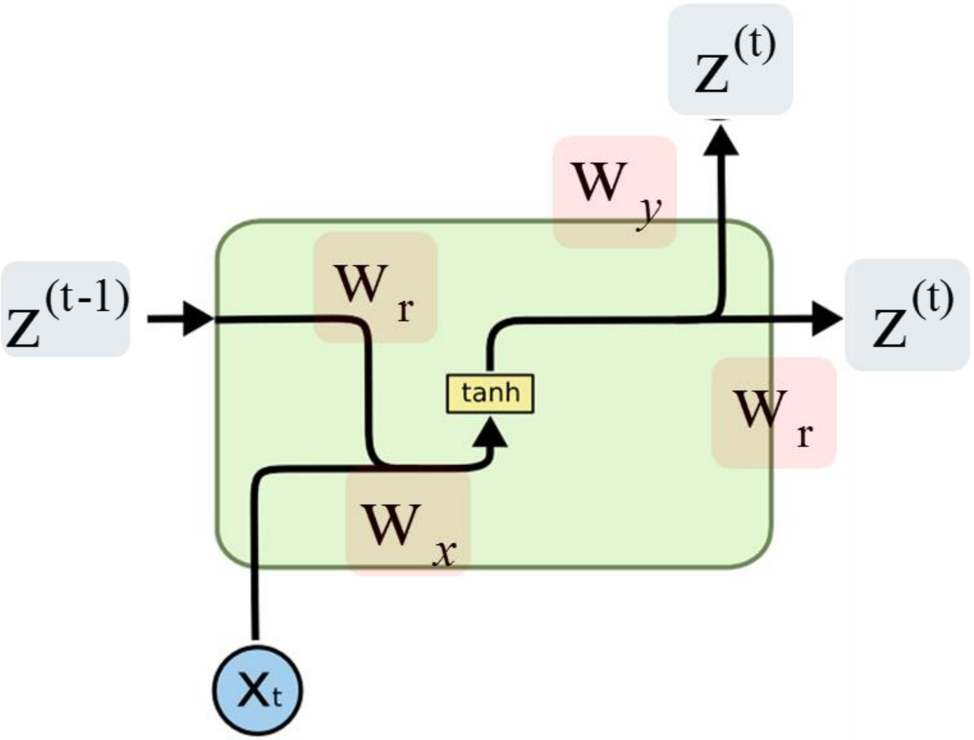


$$z^{(t)} = \alpha_z\left(w_x x^{(t)} + w_r z^{(t-1)} + b_z\right)$$

$$y^{(t)} = \alpha_y\left(w_y z^{(t)} + b_y\right)$$

RNN

Long Short Term Memory (LSTM) network

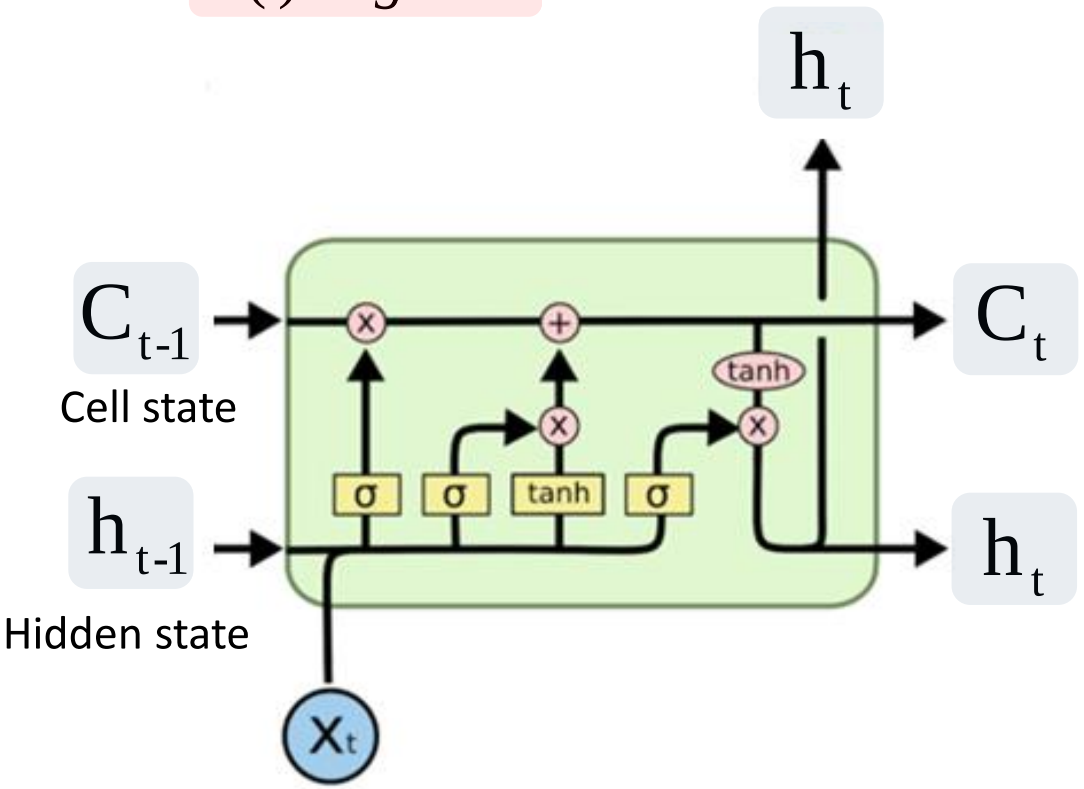


$$z^{(t)} = \alpha_z\left(w_x x^{(t)} + w_r z^{(t-1)} + b_z\right)$$

$$y^{(t)} = \alpha_y\left(w_y z^{(t)} + b_y\right)$$

RNN

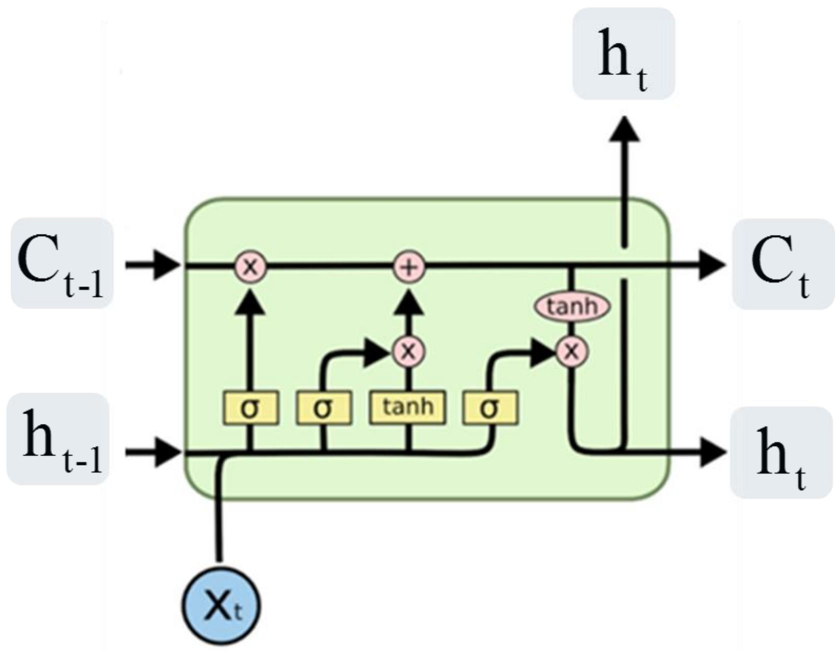
$\sigma(\cdot) : sigmoid$



- *Cell state: a memory of LSTM cell*
- *Hidden state: an output of this cell*

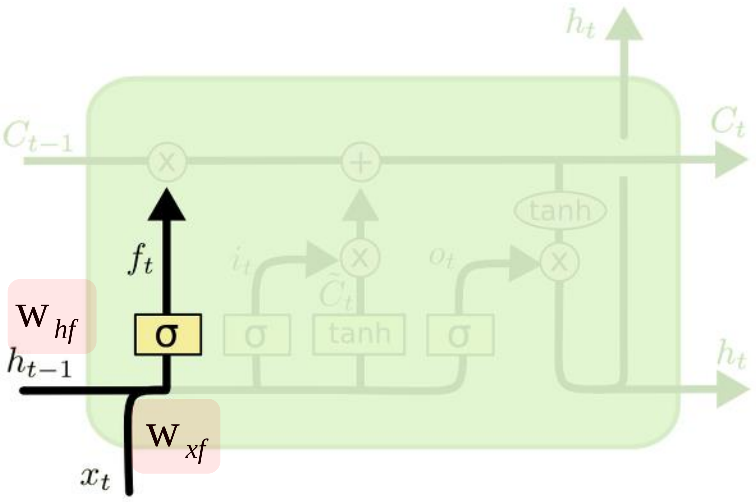
LSTM

Long Short Term Memory (LSTM) network: forget gate



LSTM

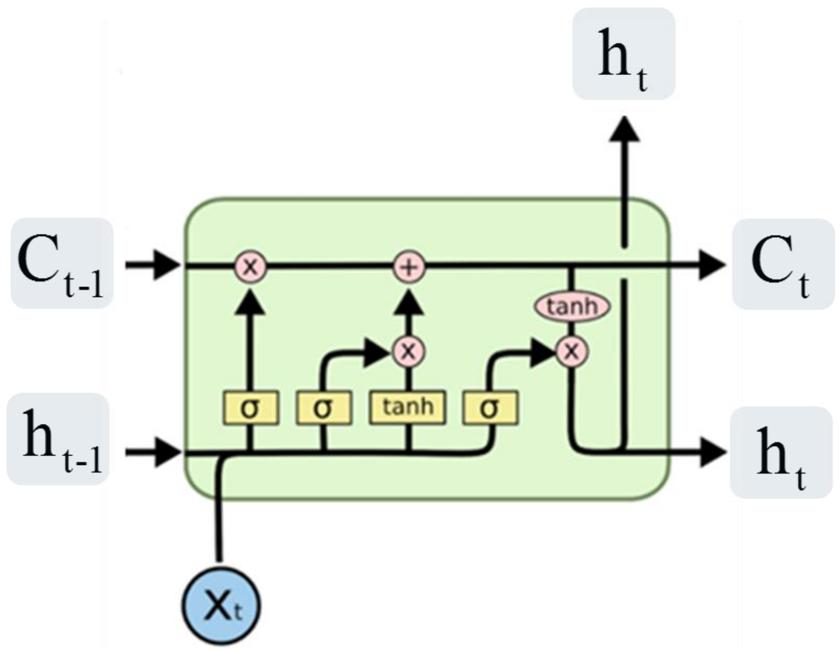
$\sigma(\cdot): sigmoid$



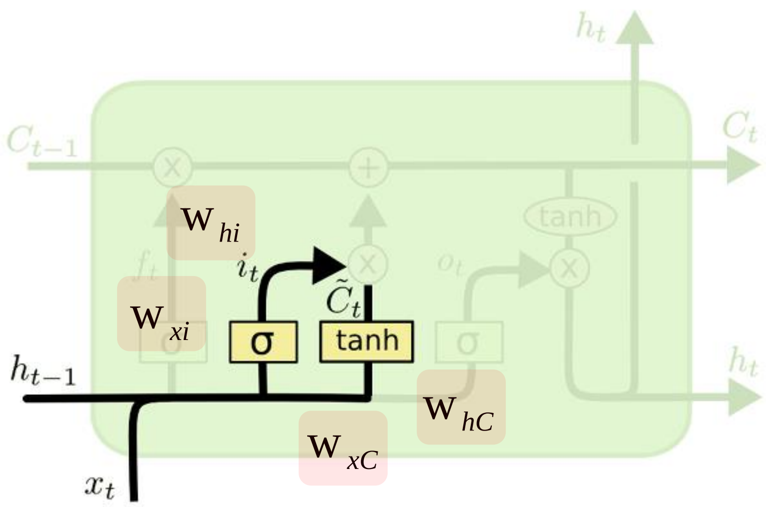
$$f_t = sigm(\mathrm{w}_{\mathrm{xf}}\mathrm{x}_{\mathrm{t}} + \mathrm{w}_{\mathrm{hf}}\mathrm{h}_{\mathrm{t\text{-}1}} + b_f)$$

- *Forget gate layer*
  - Decide how much "C_{t-1}" is forgotten?
  - If "f_t" is zero, forget "C_{t-1}" completely.
  - If "f_t" is one, do not forget "C_{t-1}" at all.

Long Short Term Memory (LSTM) network: input gate



LSTM



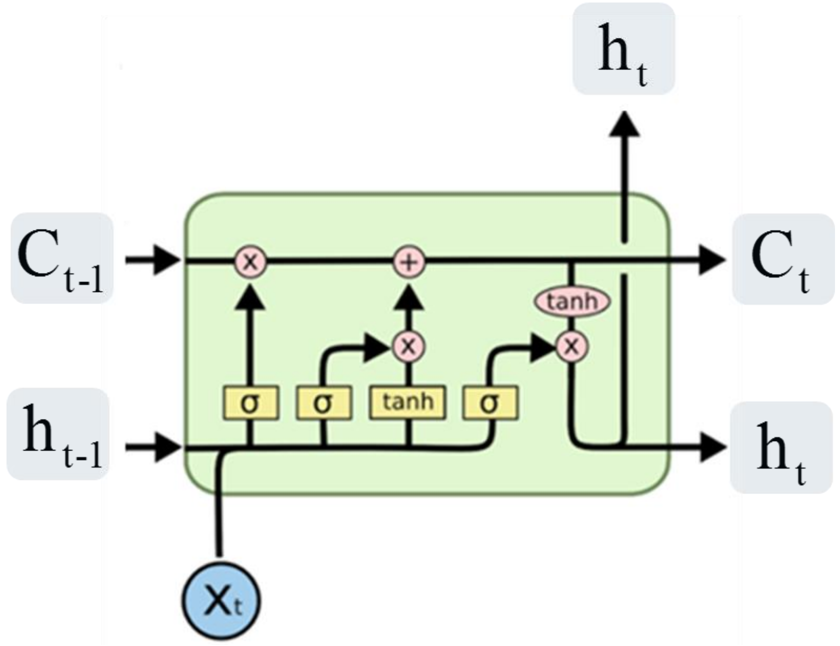
$\sigma(\cdot): sigmoid$

$$i_t = sigm(\mathrm{w}_{\mathrm{xi}}\mathrm{x}_{\mathrm{t}} + \mathrm{w}_{\mathrm{hi}}\mathrm{h}_{\mathrm{t\text{-}1}} + b_i)$$

$$\tilde{C}_t = \tanh(\mathrm{w}_{\mathrm{xC}}\mathrm{x}_{\mathrm{t}} + \mathrm{w}_{\mathrm{hC}}\mathrm{h}_{\mathrm{t\text{-}1}} + b_C)$$

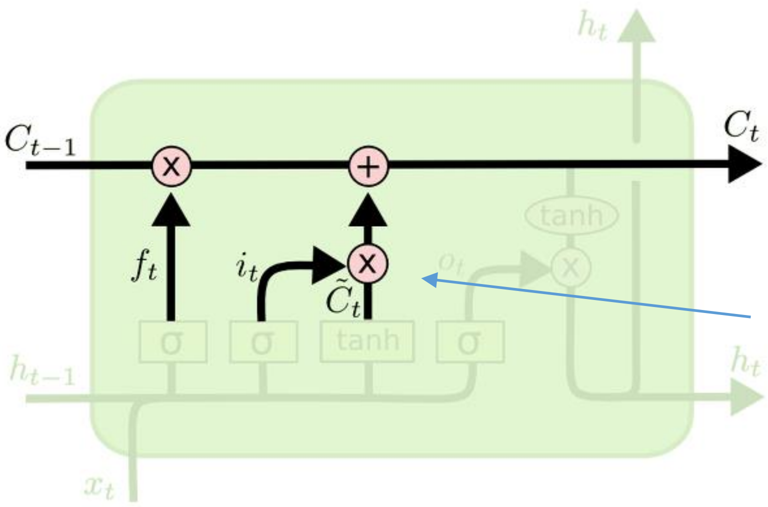
- ❑ *Input gate layer*
  - decide how much "$\tilde{C}_t$" is forgotten?
- ❑ *tanh layer:*
  - *decide which value is updated.*

Long Short Term Memory (LSTM) network: update cell



LSTM

$\sigma(\cdot): sigmoid$



Hadamard product

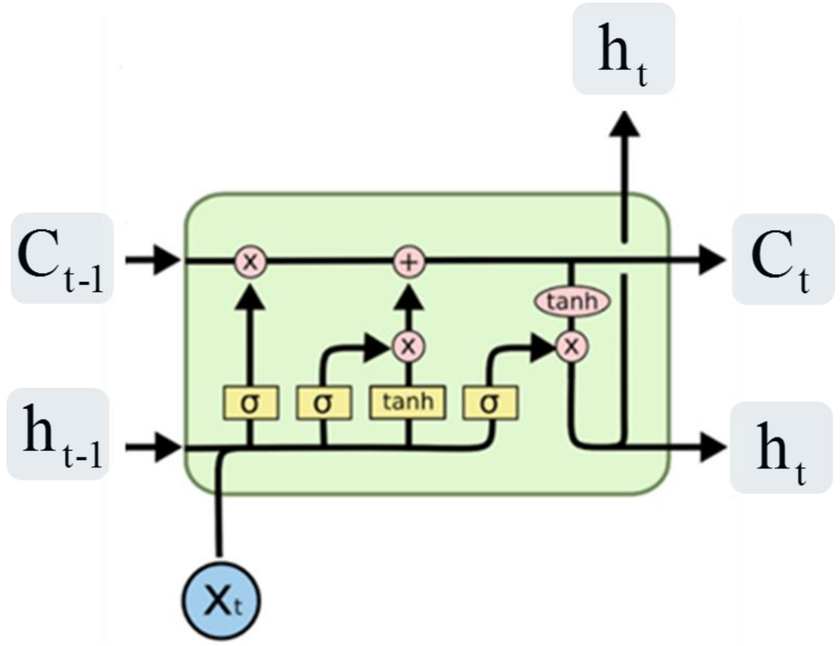
$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} * \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \end{bmatrix}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

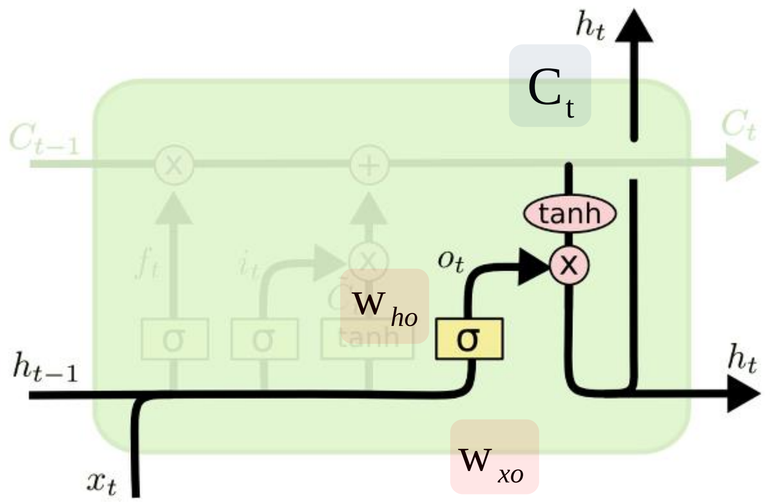
past memory　　present memory

- *Update the output cell state of "C_t" by adding the past cell state of "C_{t-1}" to the present cell state of "$\tilde{C}_t$"*

Long Short Term Memory (LSTM) network: output gate



LSTM



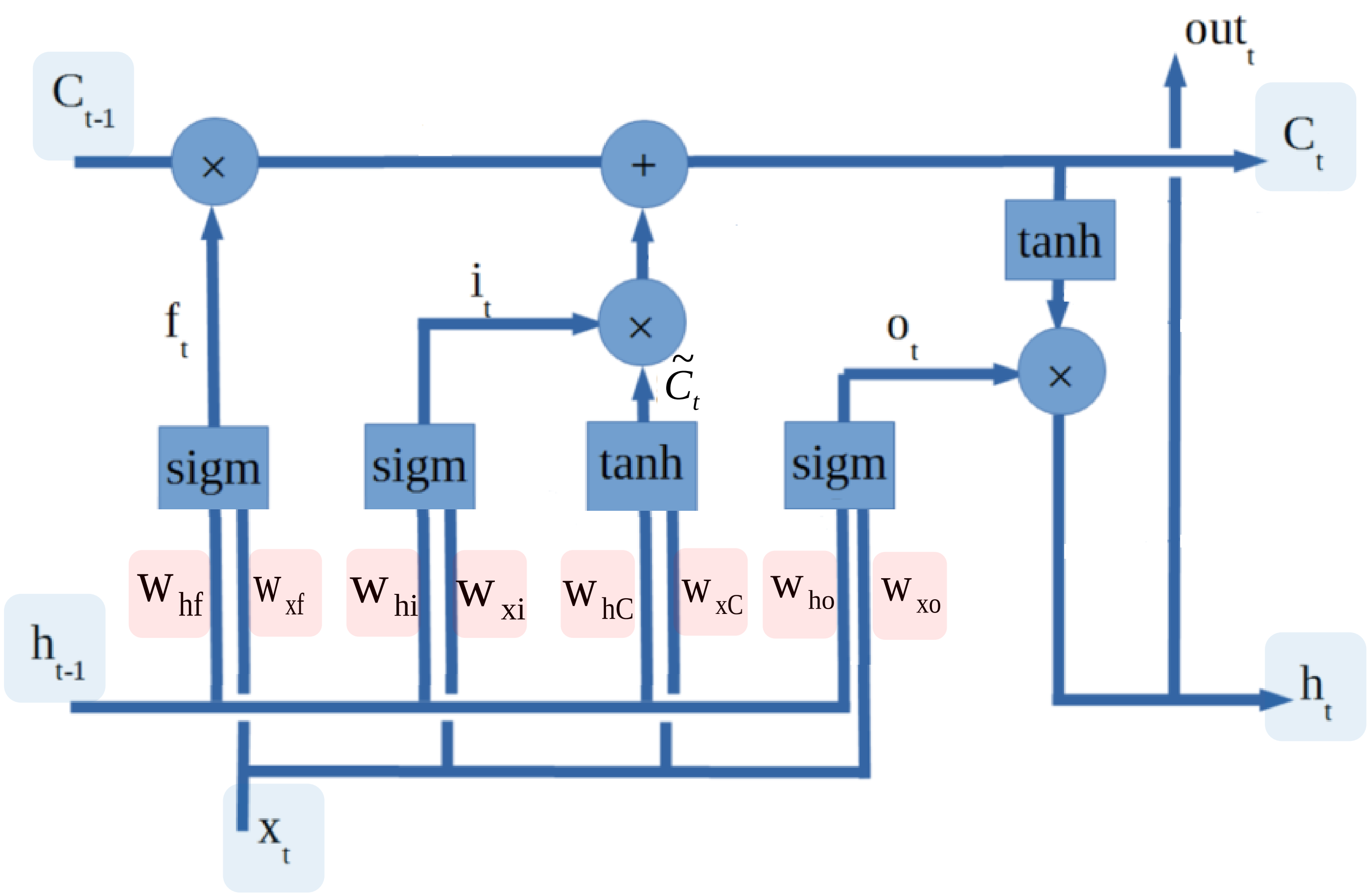
$\sigma(\cdot): sigmoid$

$$o_t = sigm(\mathrm{w}_{\mathrm{xo}}\mathrm{x}_{\mathrm{t}} + \mathrm{w}_{\mathrm{ho}}\mathrm{h}_{\mathrm{t\text{-}1}} + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

- *Output gate layer*
  - *The output cell state is put through tanh() and rescaled by the output of the sigmoid function.*

Long Short Term Memory (LSTM) network: summary



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

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

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

$$\tilde{C}_t = \tanh(W_{xC} x_t + W_{hC} h_{t-1} + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$h_t = o_t * \tanh(C_t)$$

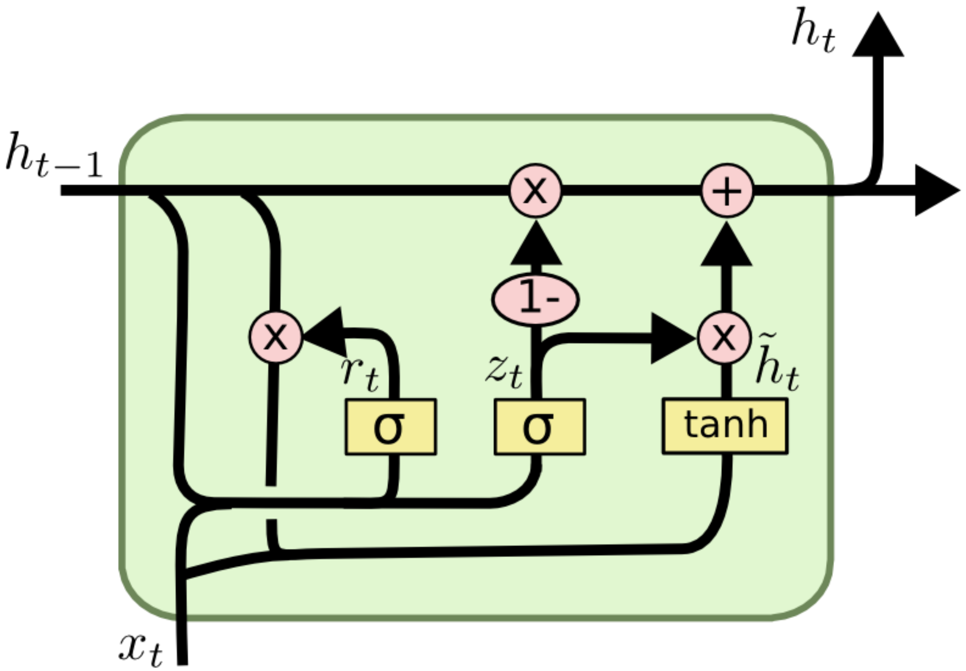
http://www.suyongeum.com/ML/lectures/LectureW10_20180621_print.pdf

- Refer to Slide 28 for LSTM backpropagation

Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU)

- Simpler than LSTM and so training is faster,
- Cell state (C) is replaced by hidden state (h),
- GRU has two gates: update gate (z), reset gate (r).



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

Hand-on Experience

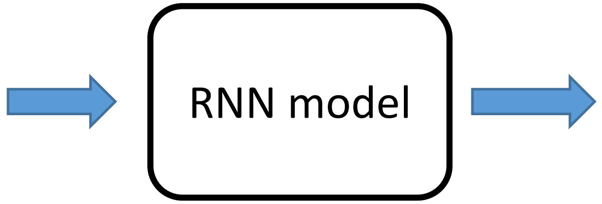
Character level language model using RNN

- The implementation is purely based on numpy only
  - https://gist.github.com/karpathy/d4dee566867f8291f086

shakespeare.txt

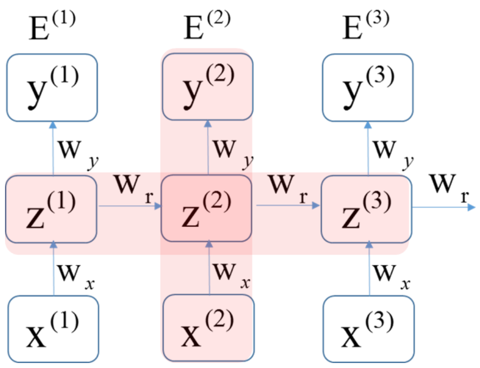
- 1115390 characters
- 65 unique characters

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?
```

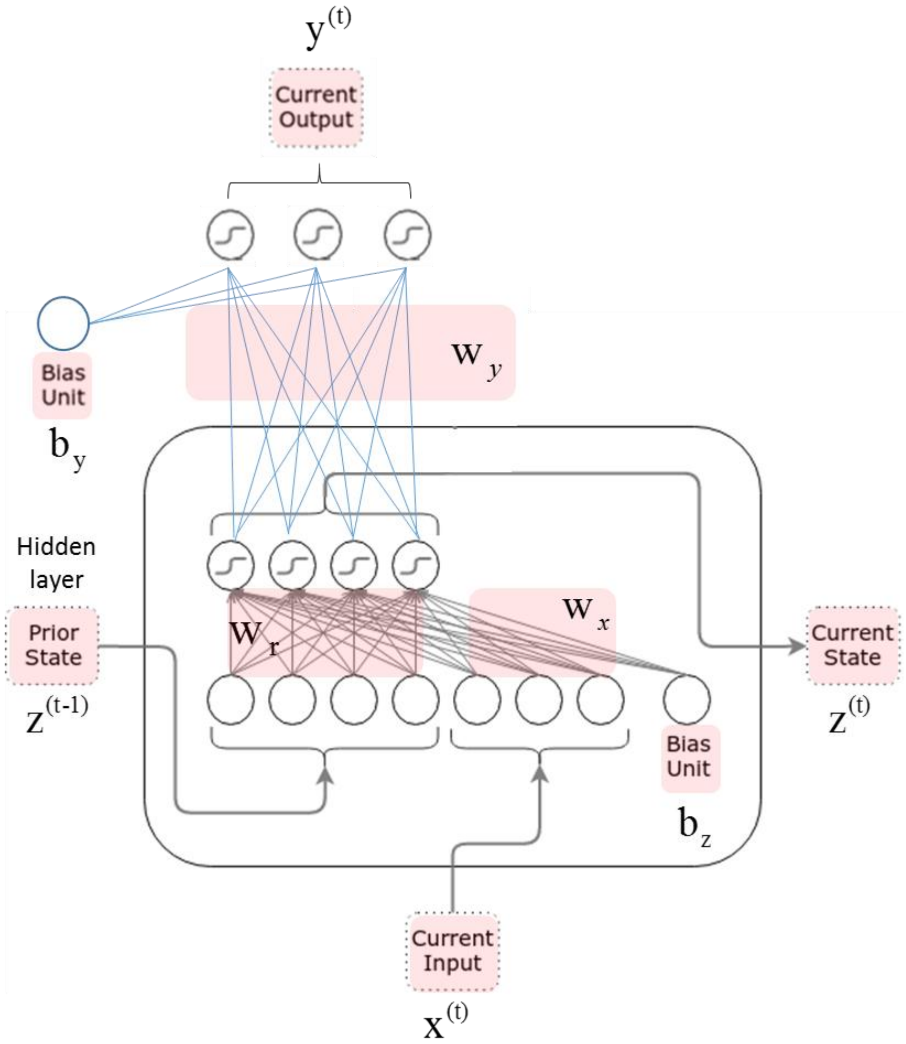


Newly generated text

KING HENRY VI I shall you sir;
When princes but friend ....

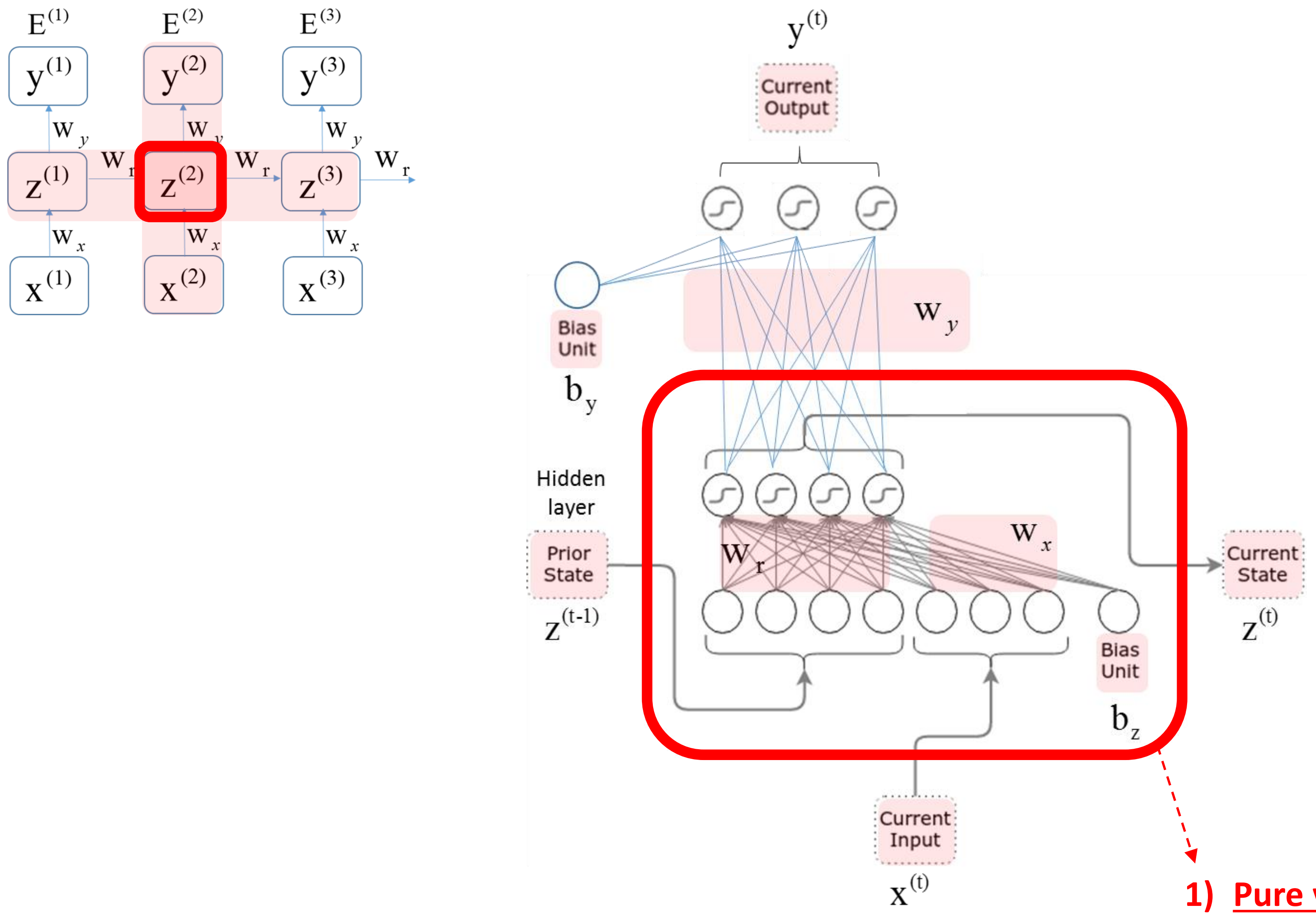


$E^{(1)}$
$E^{(2)}$
$E^{(3)}$
$y^{(1)}$
$y^{(2)}$
$y^{(3)}$
W_y
W_r
$z^{(1)}$
$z^{(2)}$
$z^{(3)}$
W_x
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$

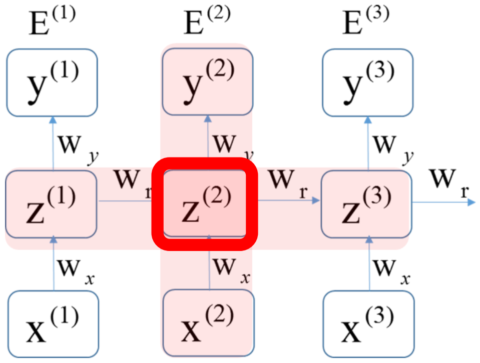


$y^{(t)}$
Current Output
W_y
Bias Unit
b_y
Hidden layer
Prior State
$z^{(t-1)}$
W_r
W_x
Current State
$z^{(t)}$
Bias Unit
b_z
Current Input
$x^{(t)}$

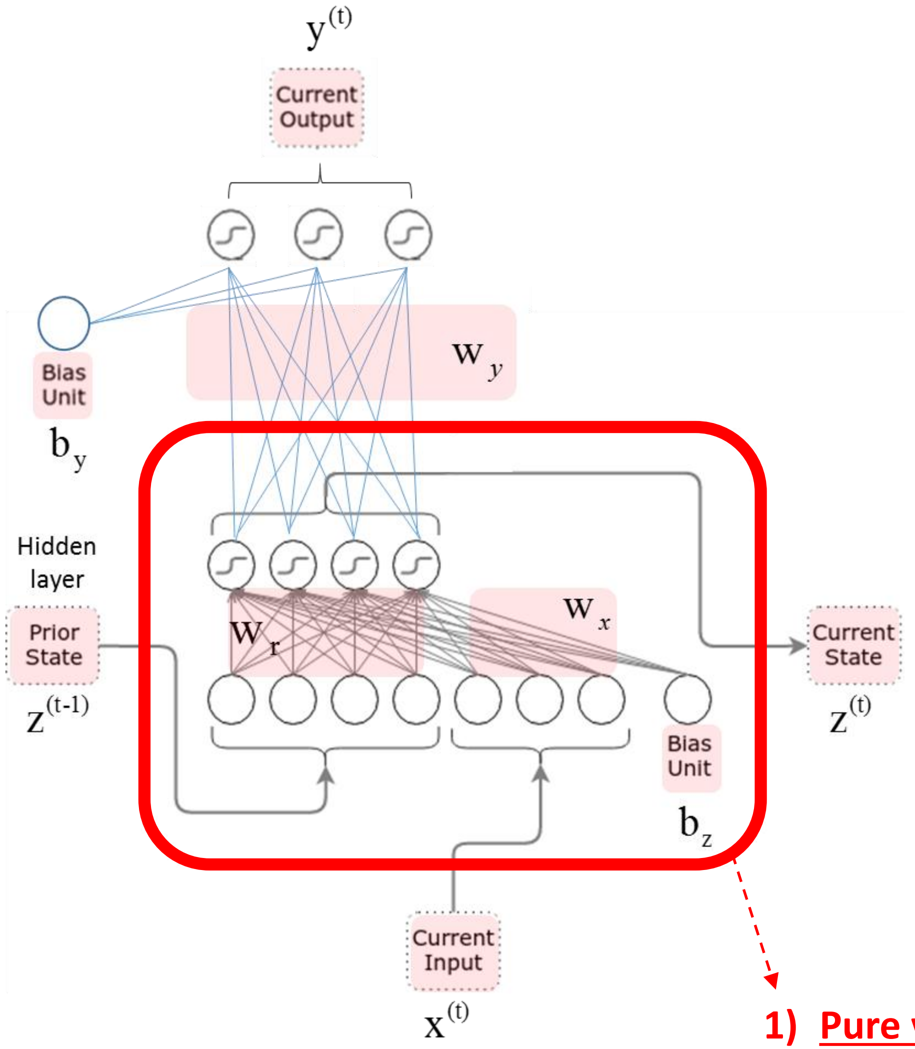
RNN review: terminology



1) <u>Pure vanilla RNN cell</u>

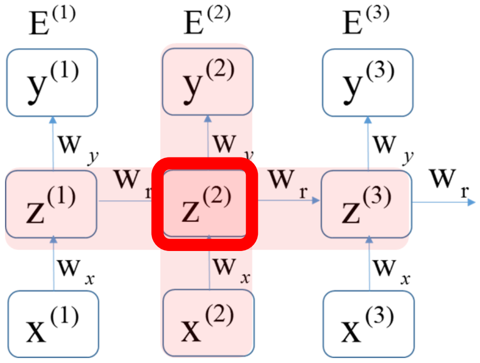


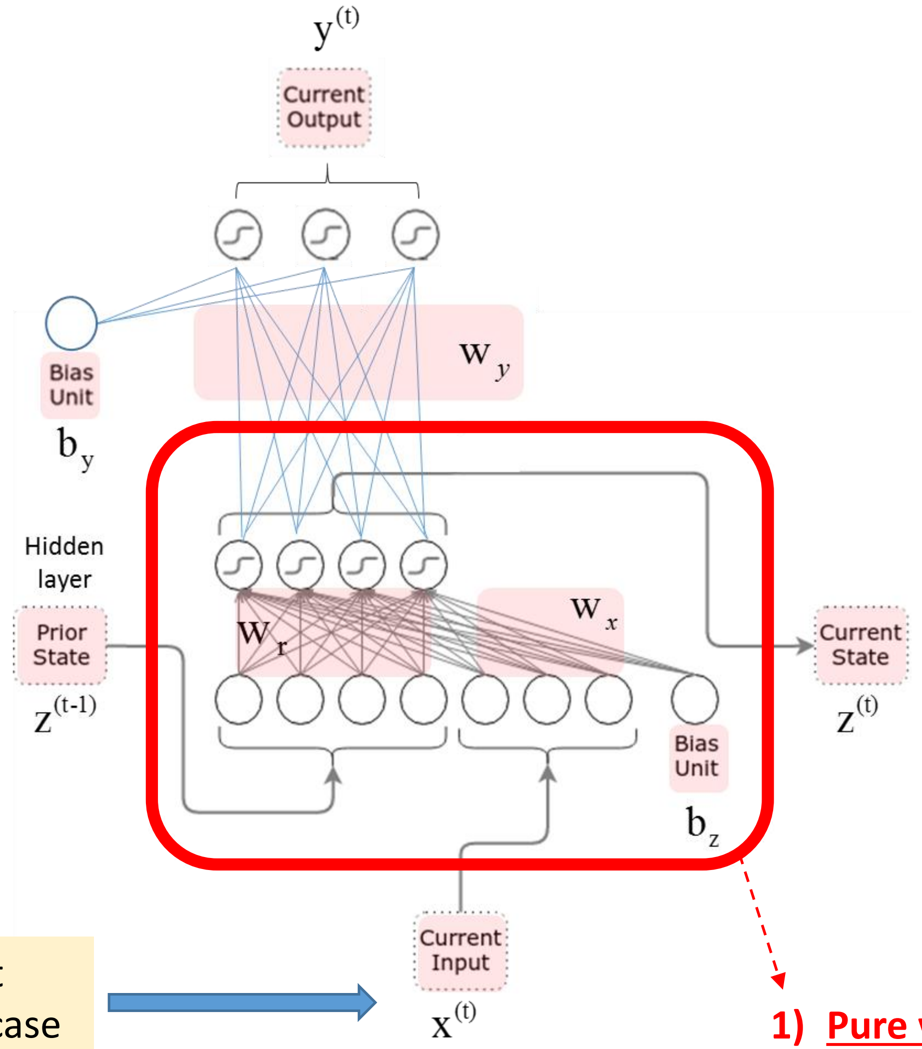
- ❑ Number of neurons in hidden layer
- hidden_size: **100**



1) **Pure vanilla RNN cell**

RNN review: terminology





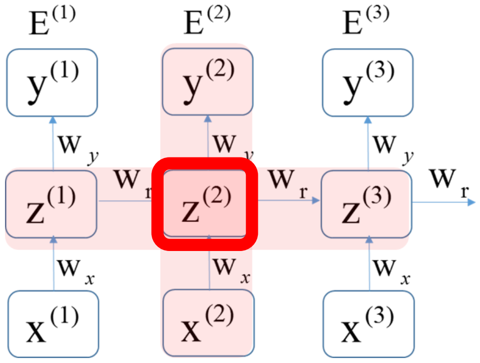
- ❑ Number of neurons in hidden layer
- \- hidden_size: **100**

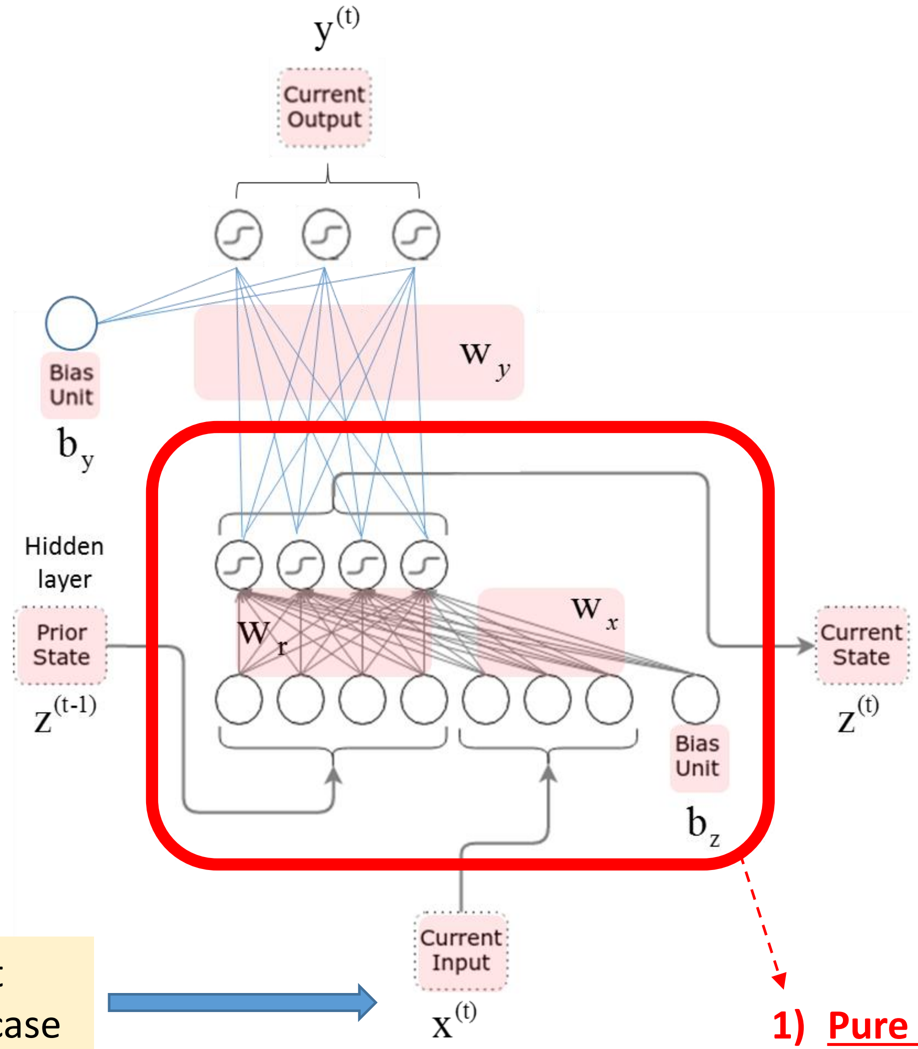
- ❑ Input dimension and Output dimension are same in this case

1) **Pure vanilla RNN cell**

- There are 65 unique characters including white space.
- One hot encoding : [0, 0, 0, …, 1, …, 0]

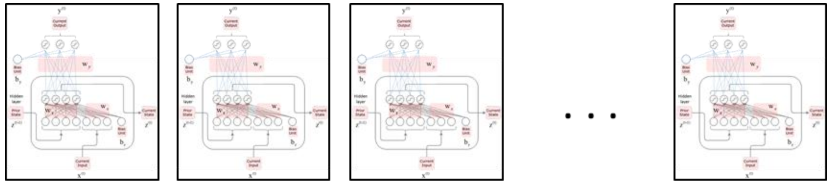
RNN review: terminology





1) **Pure vanilla RNN cell**

- ❑ Number of steps
- - seq_length: **25**



. . .

- ❑ Number of neurons in hidden layer
- - hidden_size: **100**

- ❑ Input dimension and Output dimension are same in this case

- There are **65** unique characters including white space.
- One hot encoding : [0, 0, 0, …, 1, …, 0]

Data loading

1) Data loading

```python
# data I/O
data = open(input_file, 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size, vocab_size))

char_to_ix = { ch:i for i,ch in enumerate(chars) } # make a dictionary format. See below how it looks like
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```

```
>>> x = [1, 1, 2, 2, 2, 2, 2, 3, 3]
>>> set(x)
set([1, 2, 3])
```

```
data has 1115390 characters, 65 unique.
{'i': 0, 'D': 1, 'W': 2, '&': 3, 'l': 4, 'H': 5, 'y': 6, 'A': 7, 'I': 8, 'b': 9, 'j': 10, 'G': 11, '3': 12,
{0: 'i', 1: 'D', 2: 'W', 3: '&', 4: 'l', 5: 'H', 6: 'y', 7: 'A', 8: 'I', 9: 'b', 10: 'j', 11: 'G', 12: '3',
```

1) Data loading

```
# data I/O
data = open(input_file, 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size, vocab_size))

char_to_ix = { ch:i for i,ch in enumerate(chars) } # make a dictionary format. See below how it looks like
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```

```
data has 1115390 characters, 65 unique.
{'i': 0, 'D': 1, 'W': 2, '&': 3, 'l': 4, 'H': 5, 'y': 6, 'A': 7, 'I': 8, 'b': 9, 'j': 10, 'G': 11, '3': 12,
{0: 'i', 1: 'D', 2: 'W', 3: '&', 4: 'l', 5: 'H', 6: 'y', 7: 'A', 8: 'I', 9: 'b', 10: 'j', 11: 'G', 12: '3',
```

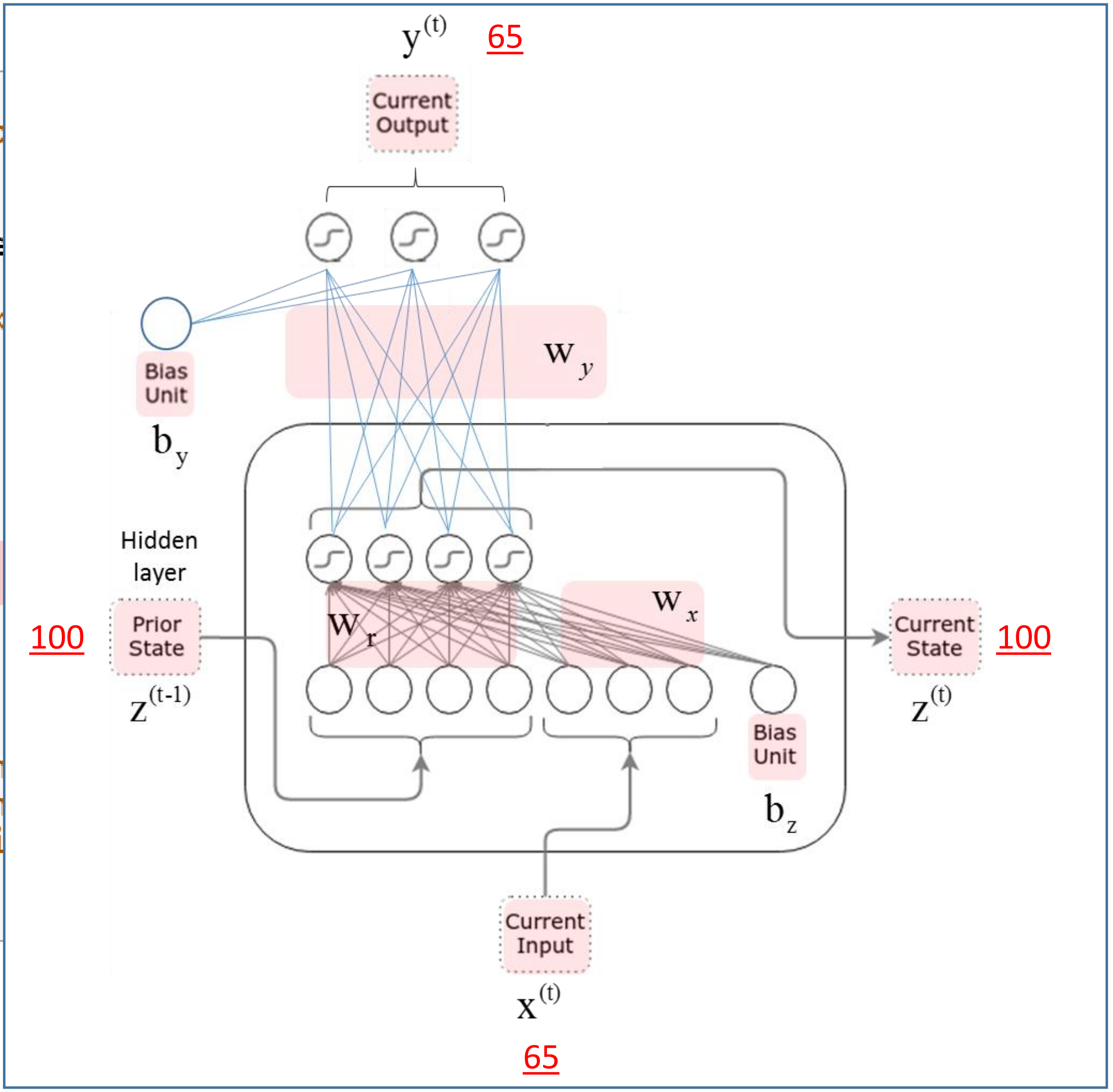
1) Data loading

```
# data I/O
data = open(input_file, 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size, vocab_size))

char_to_ix = { ch:i for i,ch in enumerate(chars) } # make a dictionary format. See below how it looks like
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```

```
data has 1115390 characters, 65 unique.
{'i': 0, 'D': 1, 'W': 2, '&': 3, 'l': 4, 'H': 5, 'y': 6, 'A': 7, 'I': 8, 'b': 9, 'j': 10, 'G': 11, '3': 12,
{0: 'i', 1: 'D', 2: 'W', 3: '&', 4: 'l', 5: 'H', 6: 'y', 7: 'A', 8: 'I', 9: 'b', 10: 'j', 11: 'G', 12: '3',
```

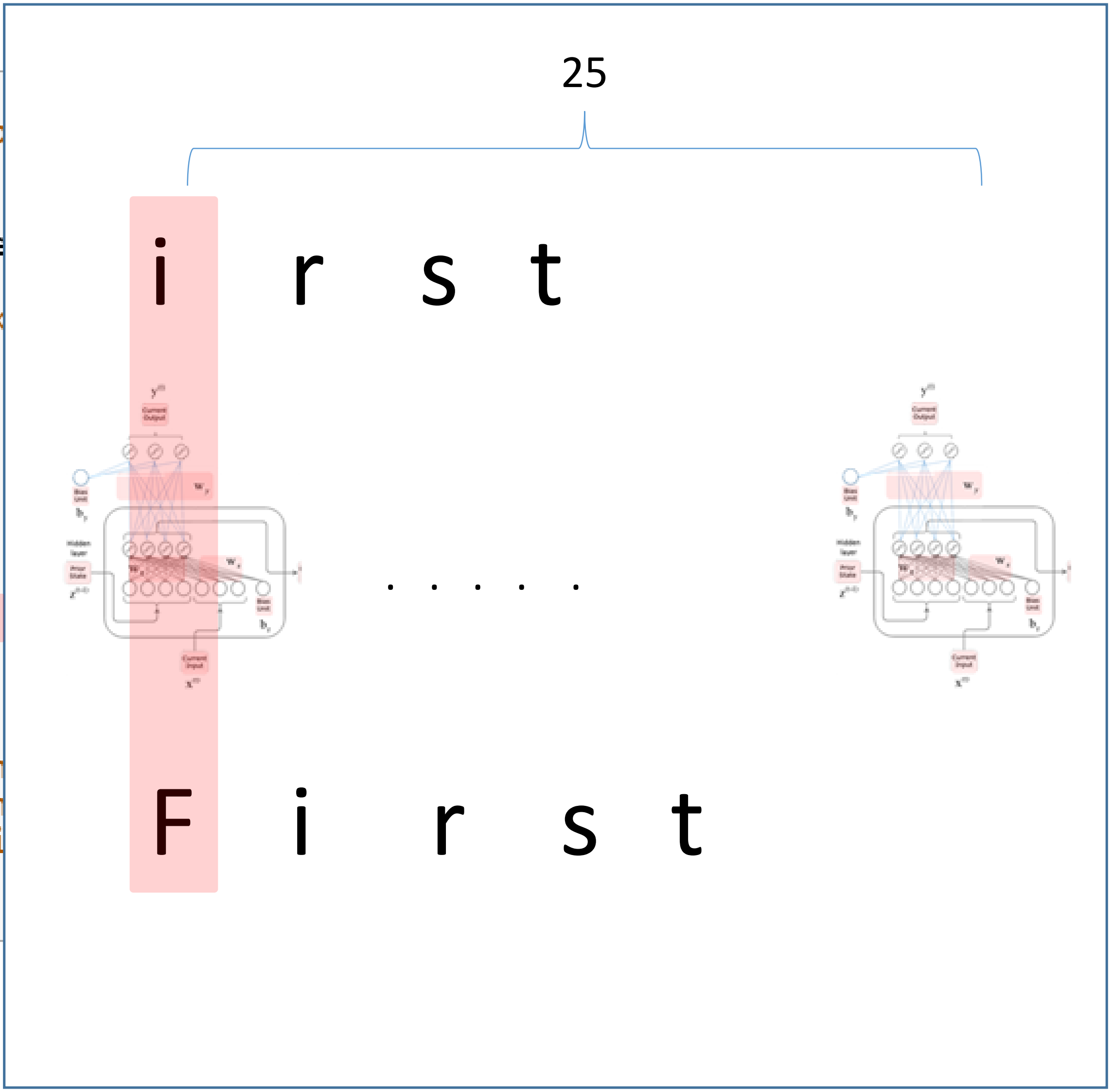
1) Data loading

```
# data I/O
data = open(input_file, 'r').read() # should be simple p
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size

char_to_ix = { ch:i for i,ch in enumerate(chars) } # mak
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # in
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # h
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hi
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```



1) Data loading

```
# data I/O
data = open(input_file, 'r').read() # should be simple p
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size

char_to_ix = { ch:i for i,ch in enumerate(chars) } # mak
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # in
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # h
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hi
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```



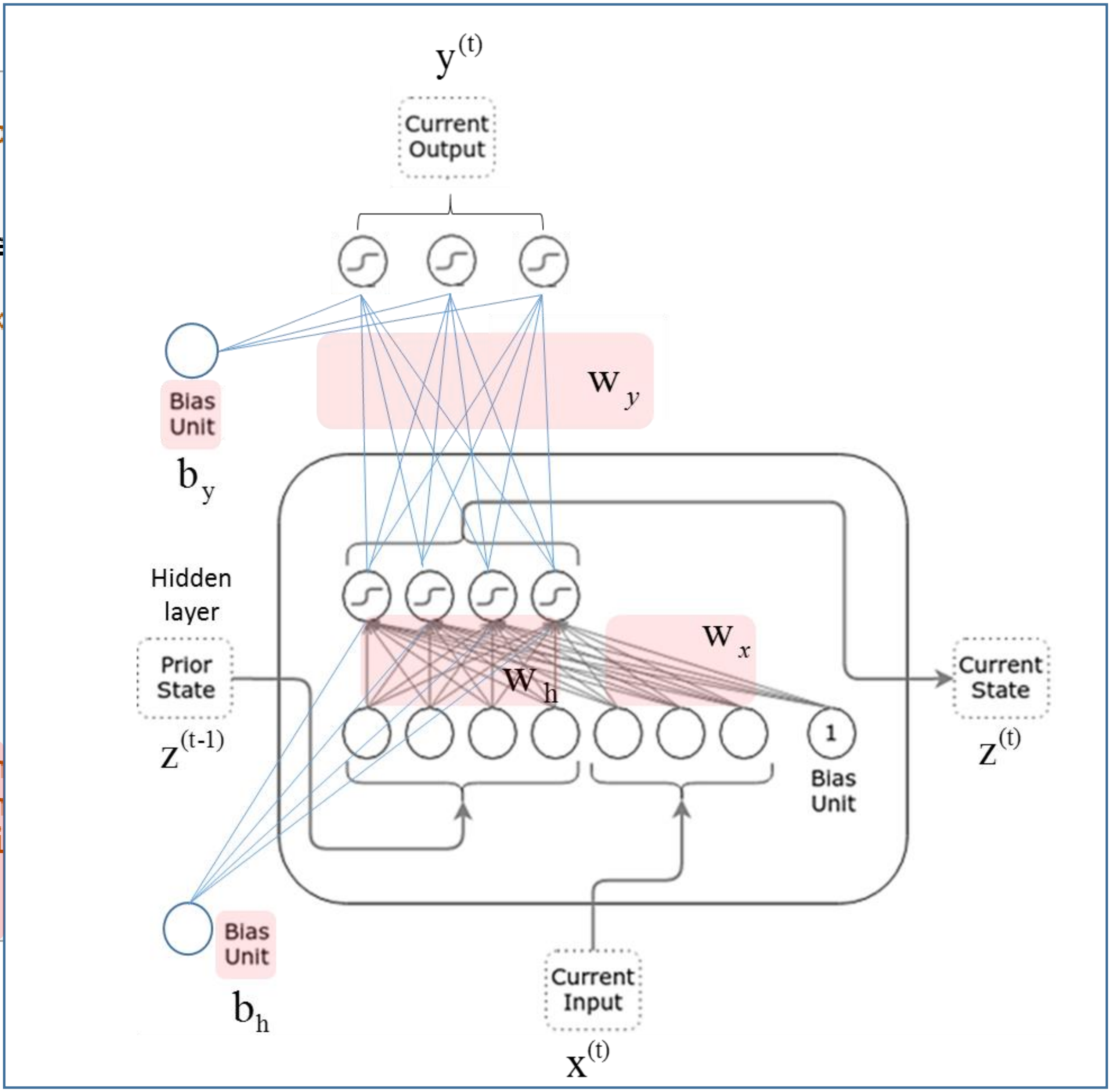
1) Data loading

```
# data I/O
data = open(input_file, 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size, vocab_size))

char_to_ix = { ch:i for i,ch in enumerate(chars) } # make a dictionary format. See below how it looks like
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```

```
data has 1115390 characters, 65 unique.
{'i': 0, 'D': 1, 'W': 2, '&': 3, 'l': 4, 'H': 5, 'y': 6, 'A': 7, 'I': 8, 'b': 9, 'j': 10, 'G': 11, '3': 12,
{0: 'i', 1: 'D', 2: 'W', 3: '&', 4: 'l', 5: 'H', 6: 'y', 7: 'A', 8: 'I', 9: 'b', 10: 'j', 11: 'G', 12: '3',
```

1) Data loading

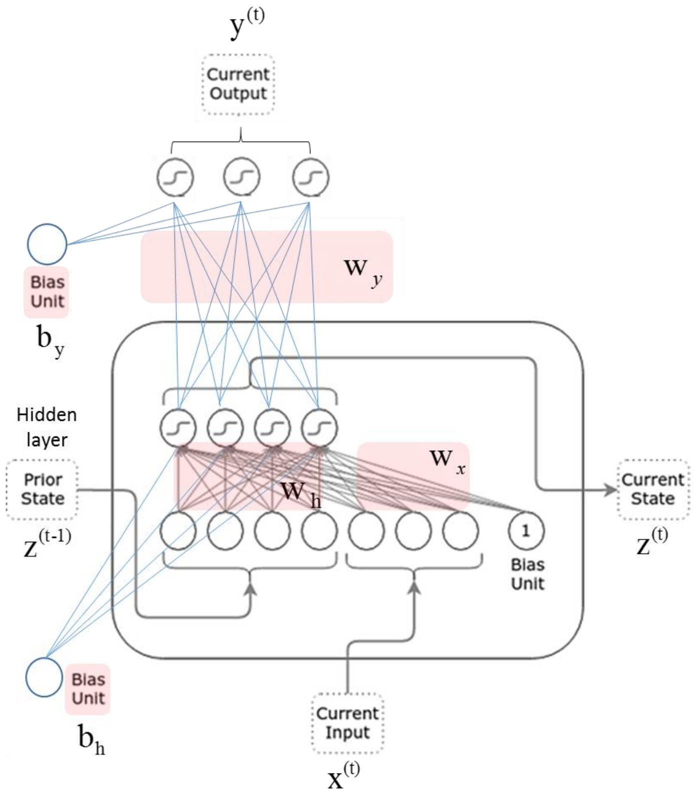
```
# data I/O
data = open(input_file, 'r').read() # should be simple p
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size

char_to_ix = { ch:i for i,ch in enumerate(chars) } # mak
ix_to_char = { i:ch for i,ch in enumerate(chars) }

print (char_to_ix)
print (ix_to_char)

# hyperparameters
hidden_size = 100 # hidden_size: # of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # in
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # h
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hi
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```



Evaluation

2) Evaluation: loss calculation

```python
def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in range(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
```



- ps[t]: predicted label which is the output from the previous layer
  - e.g., 0.345
- targets[t]: index of the corresponding character. Thus, [target[t], 0] always returns 1

example

ps	target	Cross Entropy (error)
0.1, 0.2, 0.7	0, 0, 1	-ln(0.1)*0-ln(0.2)*0-ln(0.7)*1 = 0.357
0.1, 0.6, 0.3	0, 1, 0	-ln(0.1)*0-ln(0.6)*1-ln(0.3)*0 = 0.511
0.3, 0.3, 0.4	1, 0, 0	-ln(0.3)*1-ln(0.3)*0-ln(0.4)*0 = 1.204

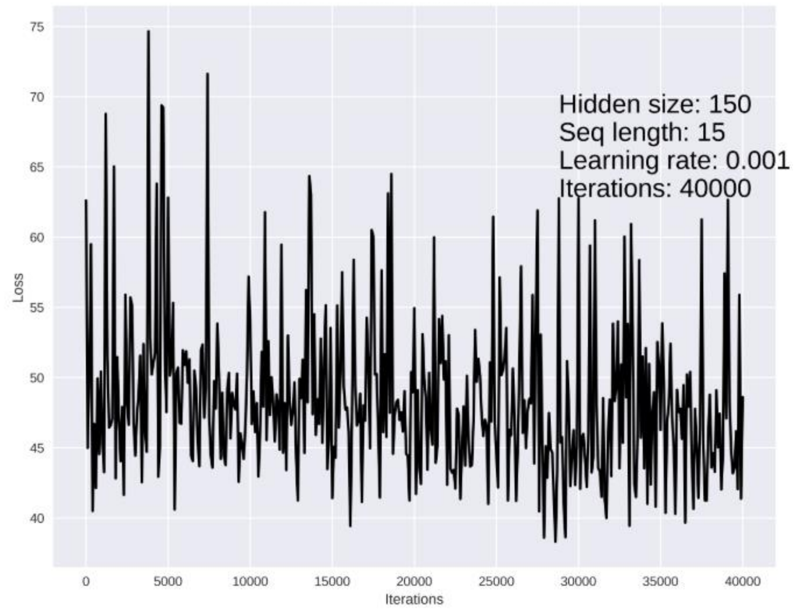
2) Evaluation: loss update

```python
# forward seq_length characters through the net and fetch gradient
loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
smooth_loss = smooth_loss * 0.999 + loss * 0.001 #
if n % 1000 == 0: print ('iter %d, loss: %f \n\n' % (n, smooth_loss, loss)) # print progress
```

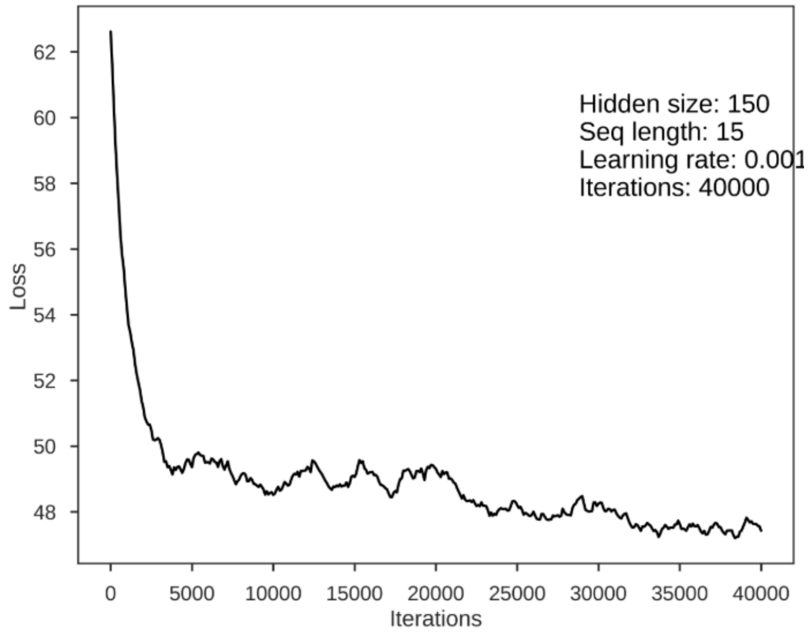
$$SL = SL * (1-\alpha) + L * \alpha$$

$$= SL - (SL - L) * \alpha$$

Loss



Smooth Loss

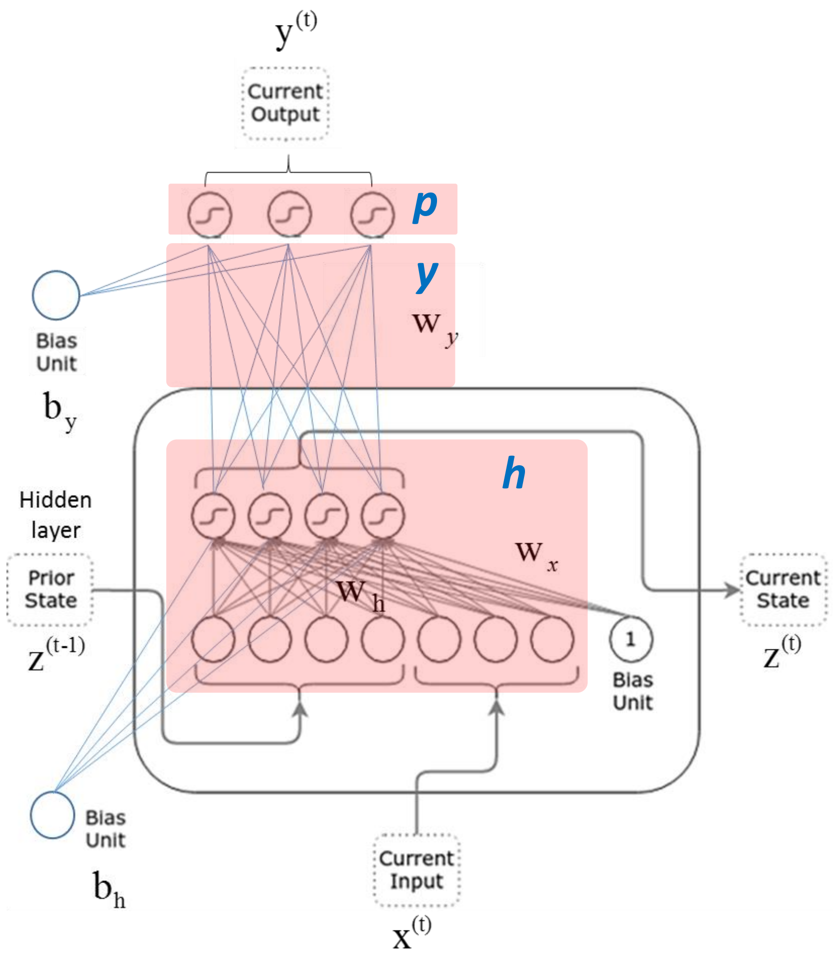


https://gist.github.com/karpathy/d4dee566867f8291f086 (sleebapaul commented on July 27)

Sampling

```python
def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in range(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    # This part is interesting to think why random not max?
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes
```

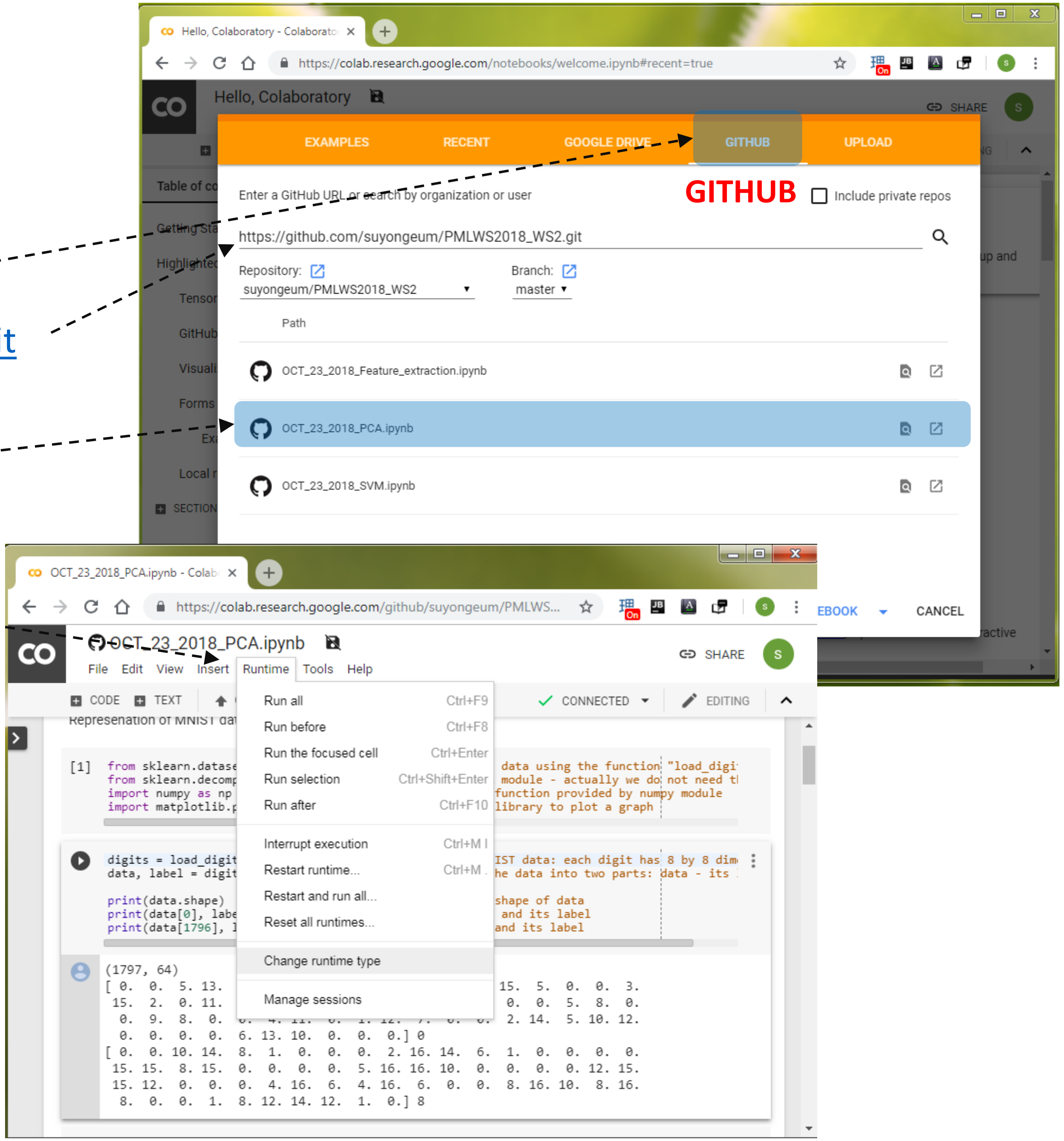
- ❑ Generating currently 200 characters
- ❑ softmax
- ❑ Selection of a character based on its probability



- h: trained parameters
- seed_ix: random selection of a character
- n: how many characters you want to generate

Colab: Character level language model using RNN

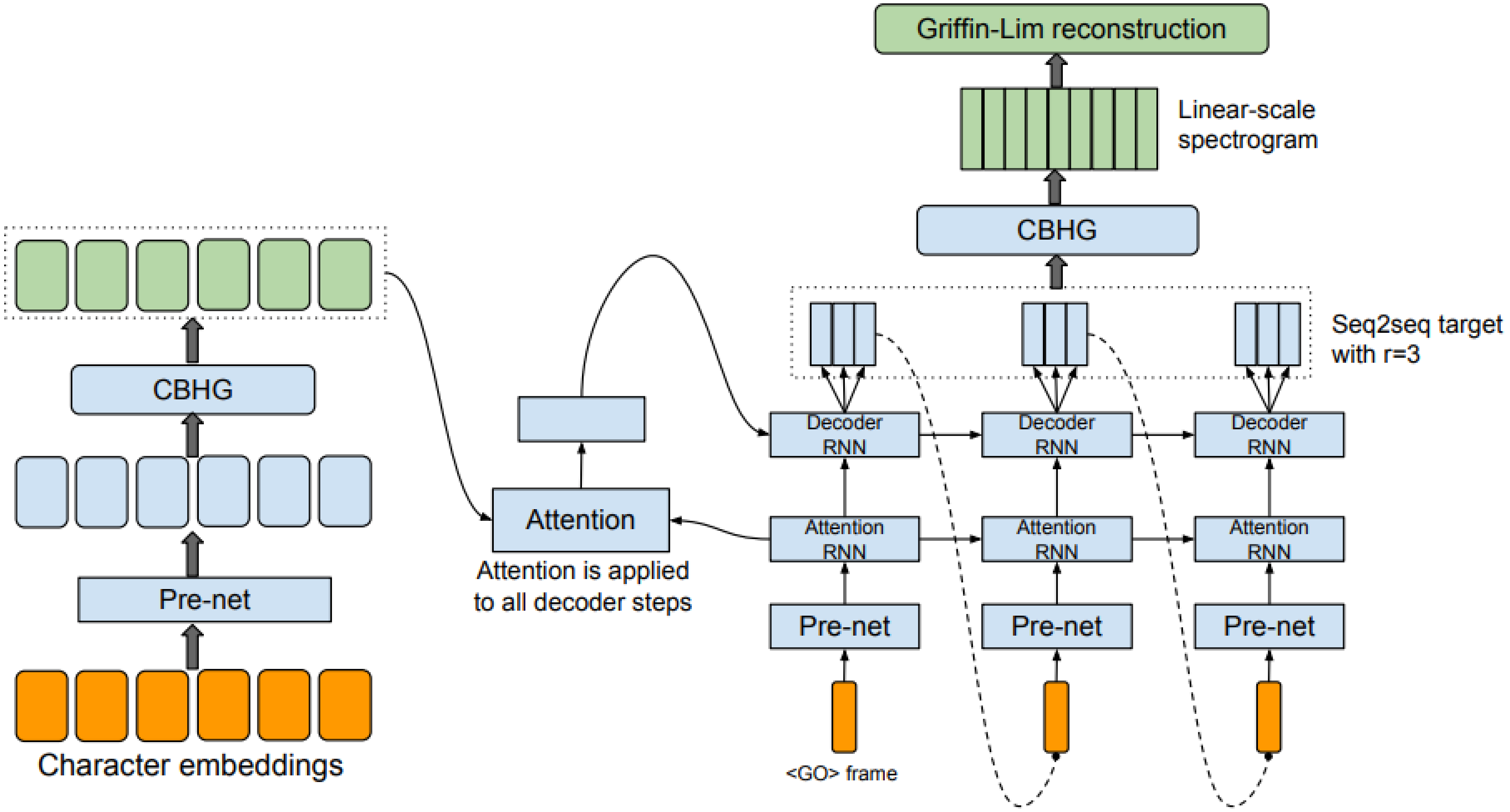
1) Go to the Colab
   - https://colab.research.google.com
2) Select “GITHUB” and copy the link below into
   - https://github.com/suyongeum/PMLWS2018_WS4.git
3) Select the notebook in the list
   - Nov_20_2018_RNN.ipynb
4) Go to “Runtime” – “Change runtime type”
   - Python 3
   - GPU
5) Save it into your gdrive
   - “File” - “Save a copy in Drive …”



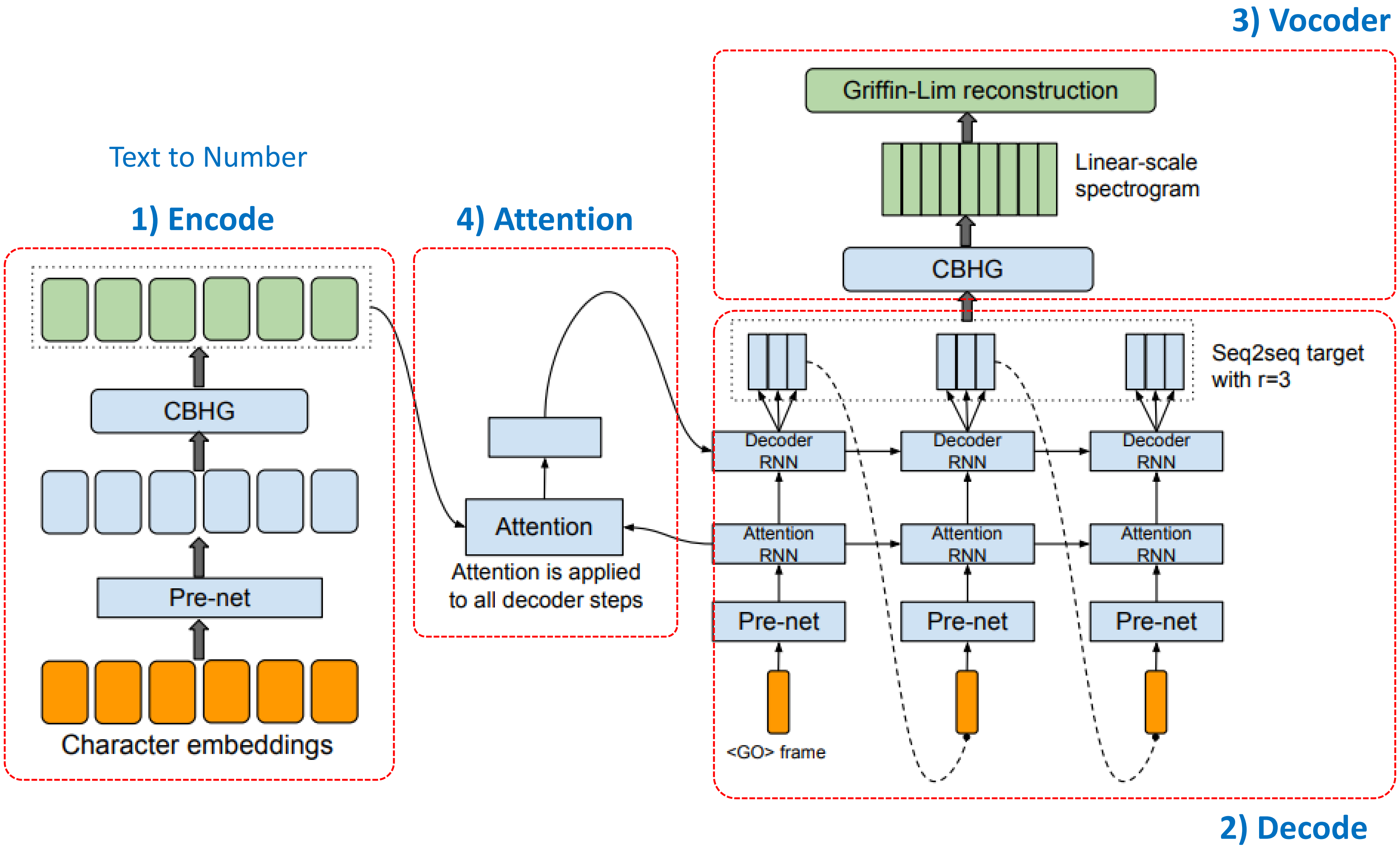
Backup slides

Tacotron: Towards End-to-End Speech synthesis

Model Architecture



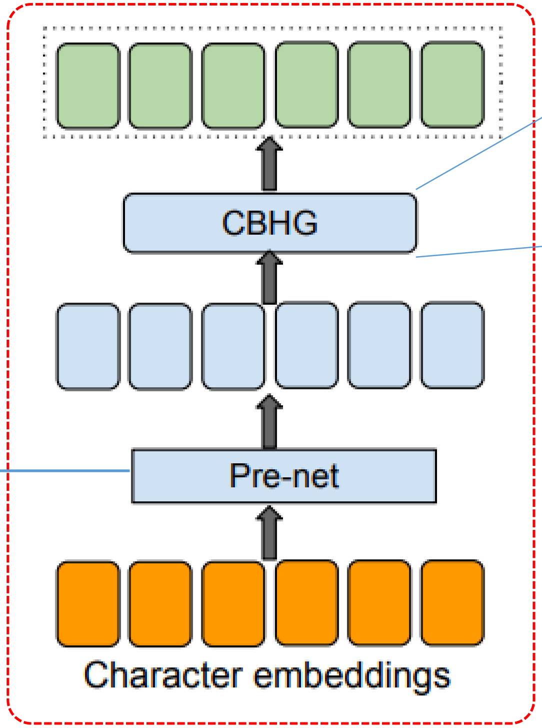
Model Architecture



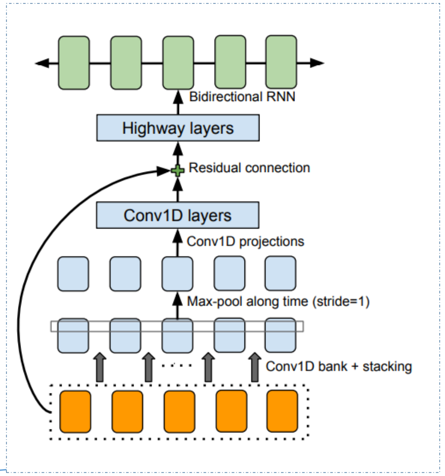
Model Architecture: 1) Encode

Text to Number

1) Encode



FC-ReLU-Dropout x 2



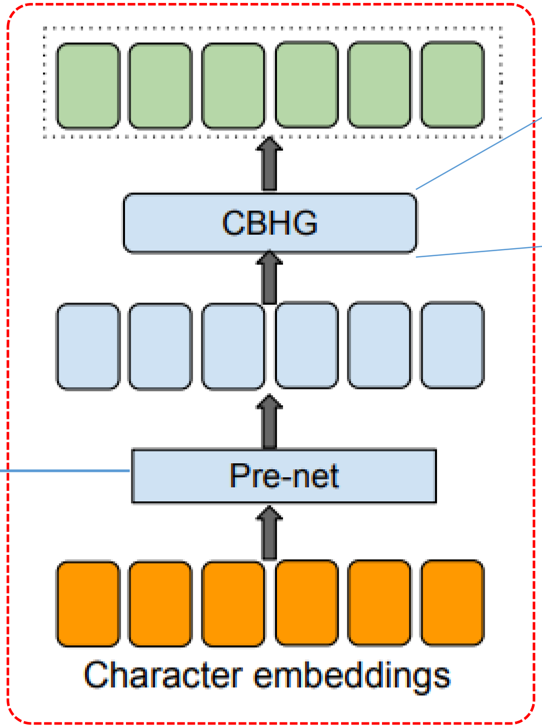
1-Convolution bank + Highway network + Bidirectional GRU

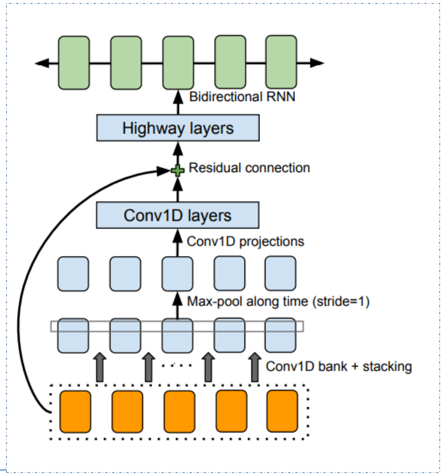
Model Architecture: 1) Encode

Text to Number

1) Encode

FC-ReLU-Dropout x 2

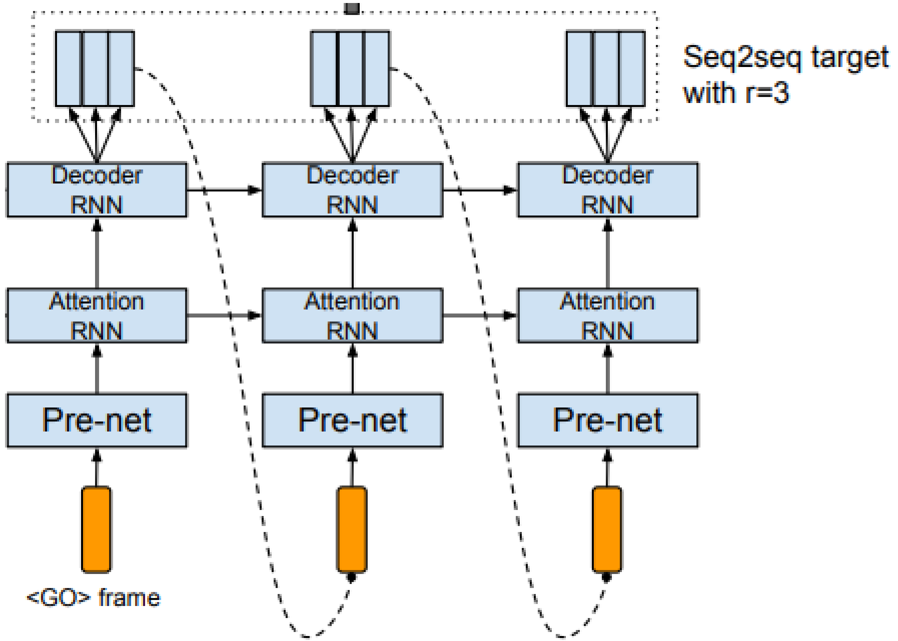




1-Convolution Bank + Highway network + Bidirectional GRU

- Convolution bank:
  - Normal convolution layer operation with some filters (filter bank)
- Highway network:
  - Learning how much input data passes through each layer.
  - https://arxiv.org/pdf/1505.00387.pdf
- Bidirectional GRU:
  - Estimating a present data point based on not only “past one” but also “future one”.
  - E.g., I like to ?? a soccer.

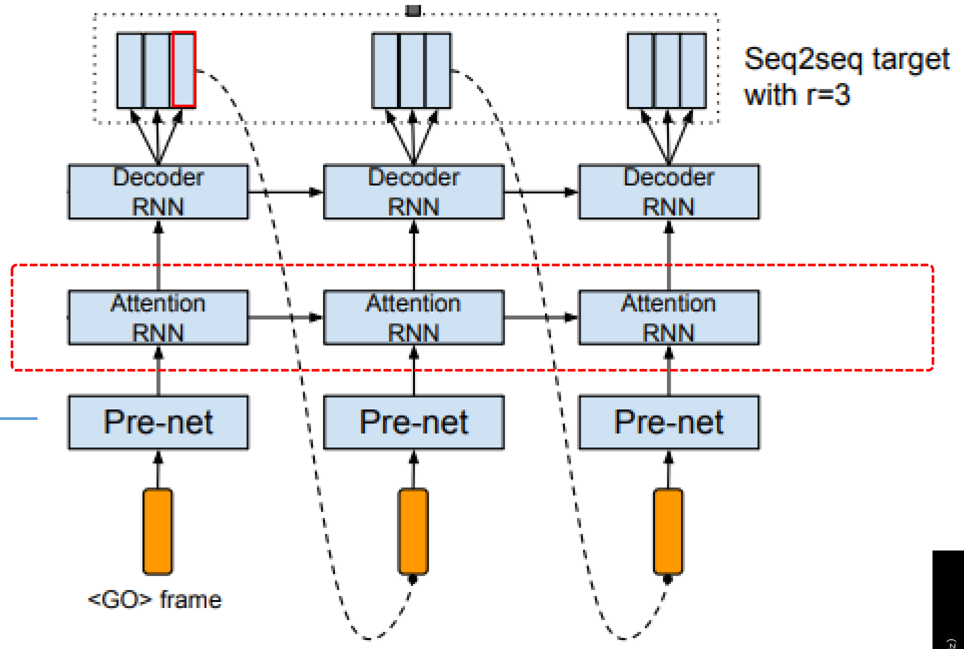
Model Architecture: 2) Decode



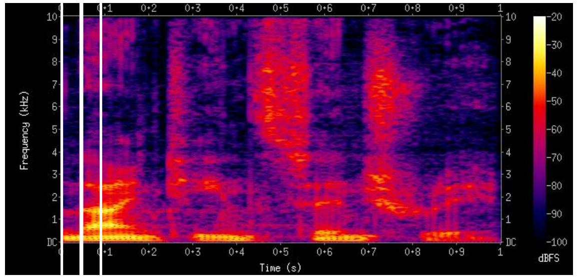
Model Architecture: 2) Decode

n spectrogram
estimation

FC-ReLU-Dropout x 2

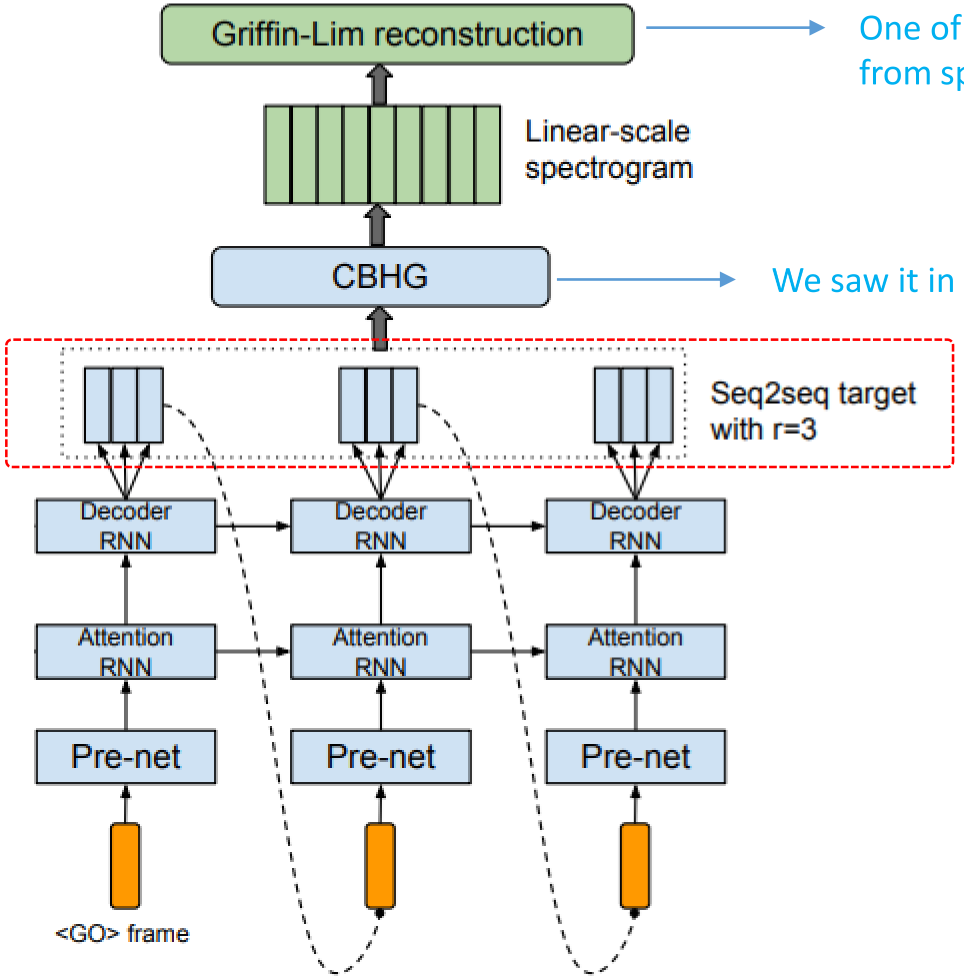


- Recurrent Neural Network
- Input: spectrogram
- Output: spectrogram



Visual representation of the spectrum of frequencies of sound or other signal as they vary with time.

Model Architecture: 3) Vocoder

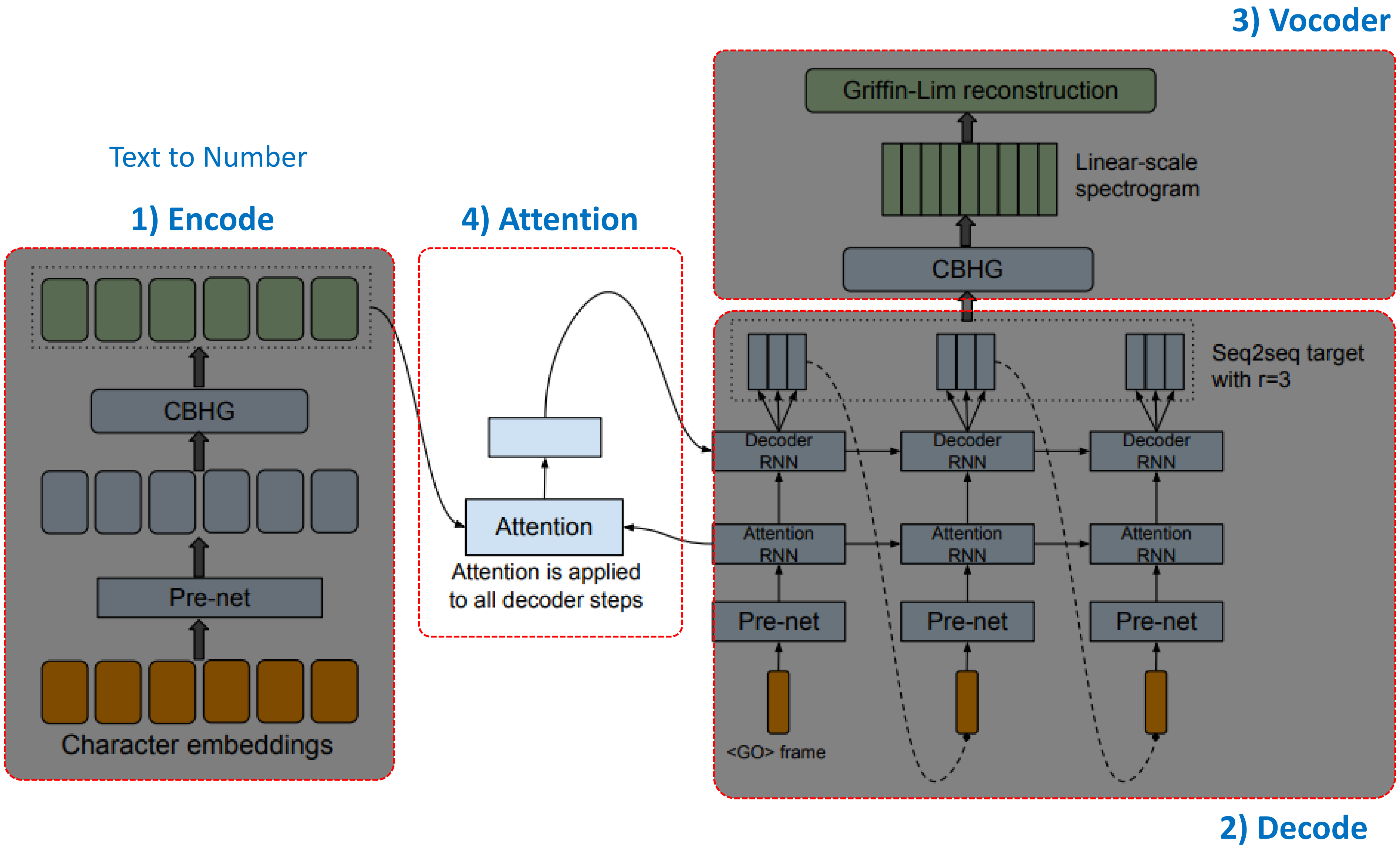


One of algorithms which convert from spectrogram to audio.

We saw it in the encode part

A set of spectrograms are estimated from decoder

Model Architecture: 4) Attention



Attention mechanism: sequence-to-sequence model

- Sequence-to-sequence is a description of a problem where your input is a sequence and your output is also sequence.
  - Machine translation
  - Question answering
  - Transcription of a photo, a video, or a summary of a document.
- RNN and LSTM are neural network models which address the sequence-to-sequence problem.

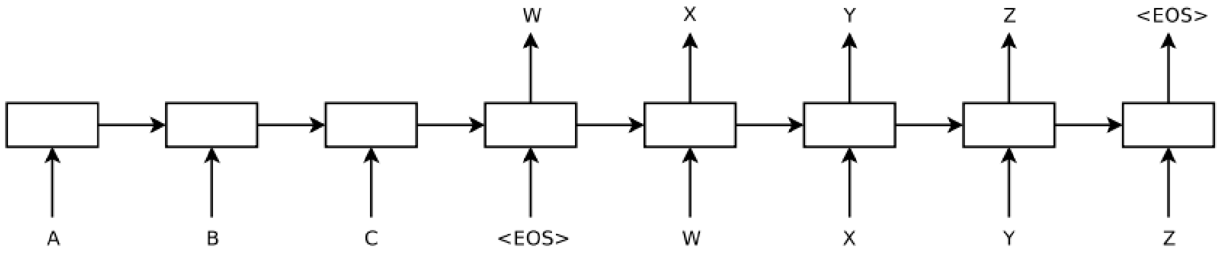


Figure 1: Our model reads an input sentence “ABC” and produces “WXYZ” as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

Sutskever et al., 2014, “Sequence to Sequence Learning with Neural Networks”

Attention mechanism: sequence-to-sequence model

- Sequence-to-sequence is a description of a problem where your input is a sequence and your output is also sequence.
  - Machine translation
  - Question answering
  - Transcription of a photo, a video, or a summary of a document.
- RNN and LSTM are neural network models which address the sequence-to-sequence problem.

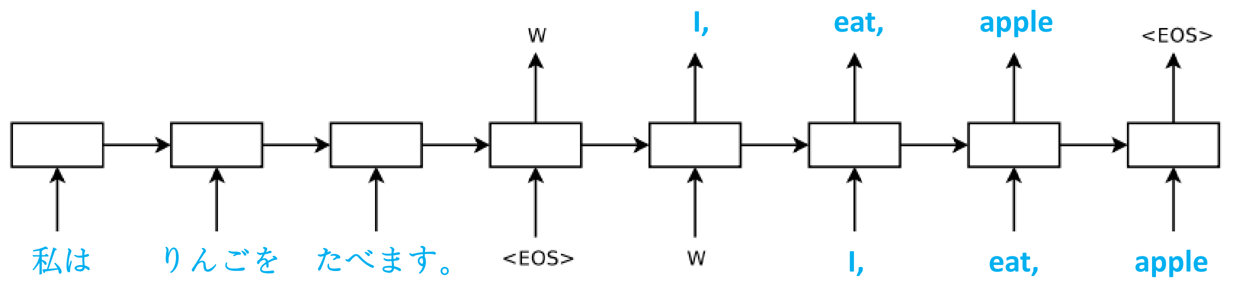
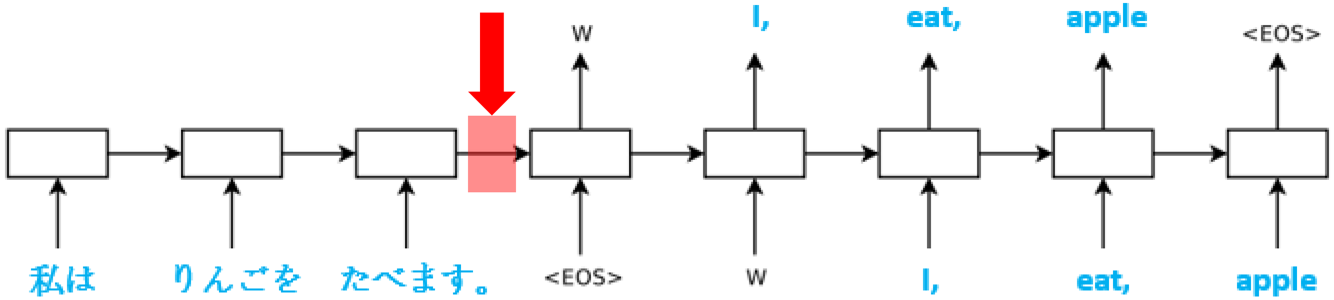


Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

Sutskever et al., 2014, "Sequence to Sequence Learning with Neural Networks"

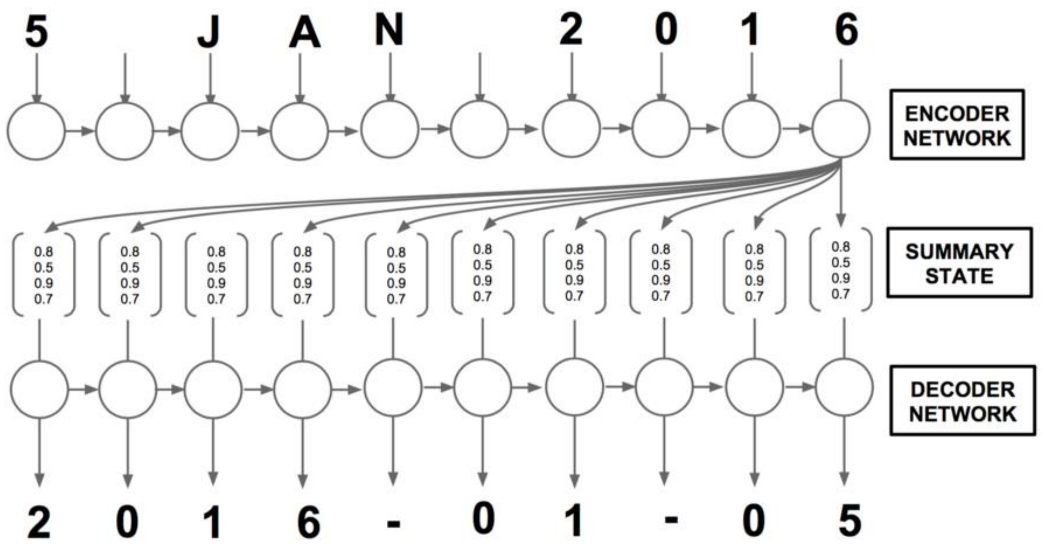
Attention mechanism: sequence-to-sequence model



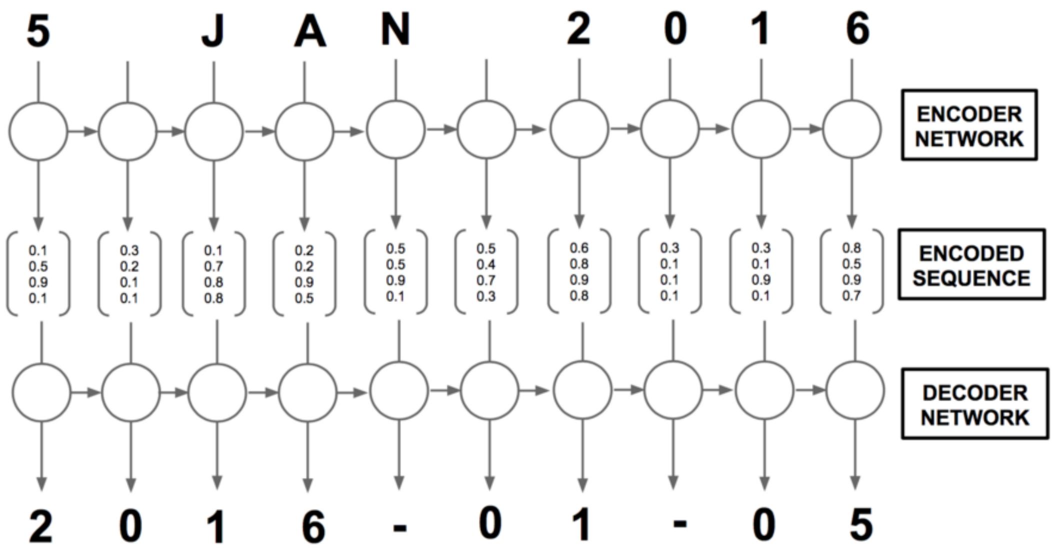
- Seq-to-seq model assumes that an input sequence is encoded into the final vector and the final vector well represents the whole input sequence.
  - However, using other encoded vectors seems to be more reasonable when decoding each part of the sentence.
    - E.g. to decode “I” we may pay more *attention* to the encode after “私は”

Example: original architecture vs with attention mechanism

- ❑ Original architecture: input data is encoded and represented as single unique code
- ❑ Architecture with an attention mechanism: individual input data are encoded and represented as multiple codes.



Original architecture



An architecture with attention mechanism

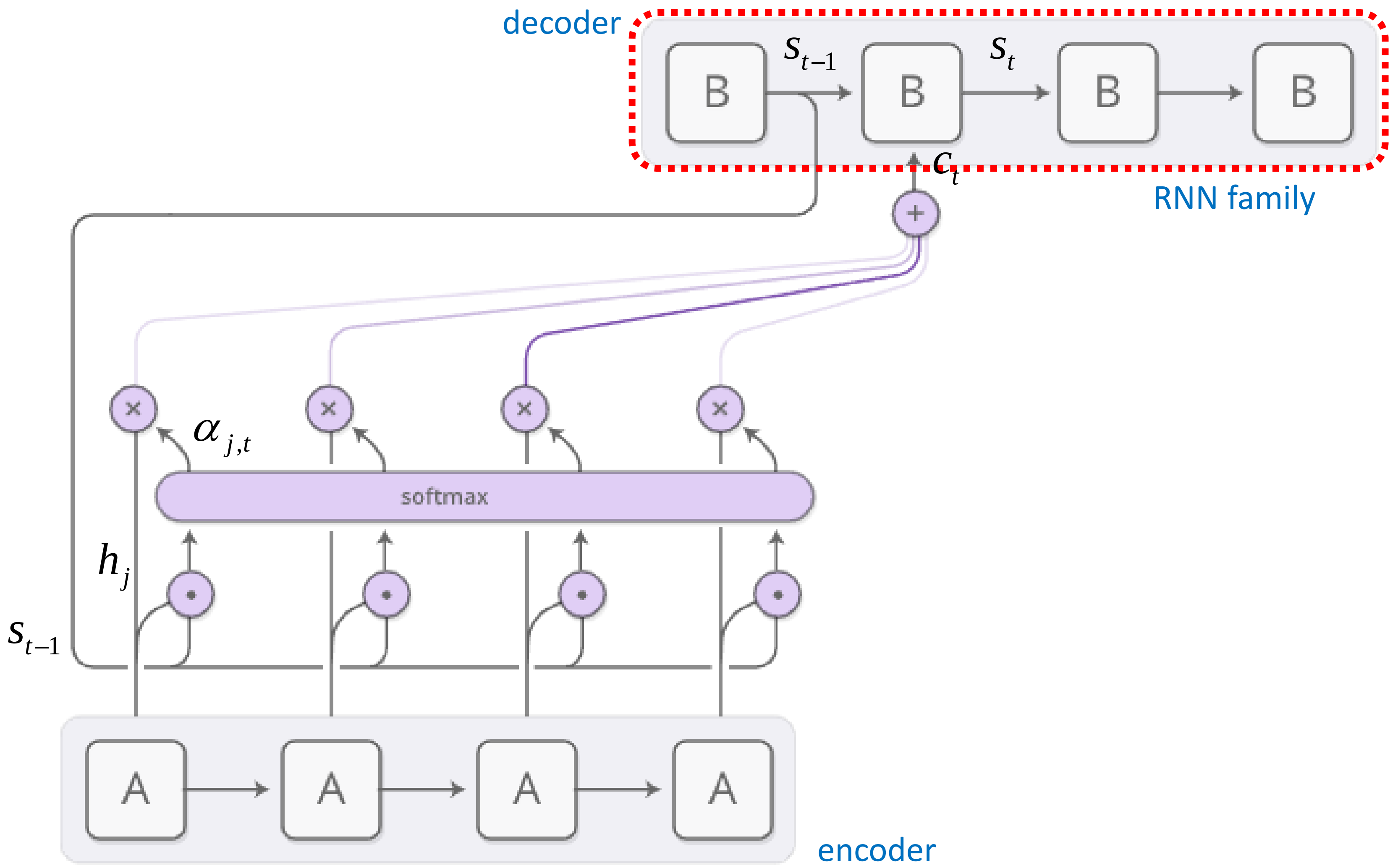
https://medium.com/datalogue/attention-in-keras-1892773a4f22

Architecture with an attention mechanism

$$e_{j,t} = V_a \cdot \tanh(W_a s_{t-1} + U_a h_j)$$

$$\alpha_{j,t} = \frac{\exp(e_j)}{\sum_{k=1}^{T} \exp(e_k)}$$

$$c_t = \sum_{k=1}^{T} \alpha_{k,t} h_k$$



https://arxiv.org/pdf/1409.0473.pdf

https://distill.pub/2016/augmented-rnns/