Modeling Sequential Data Using Recurrent Neural Networks

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

Outline

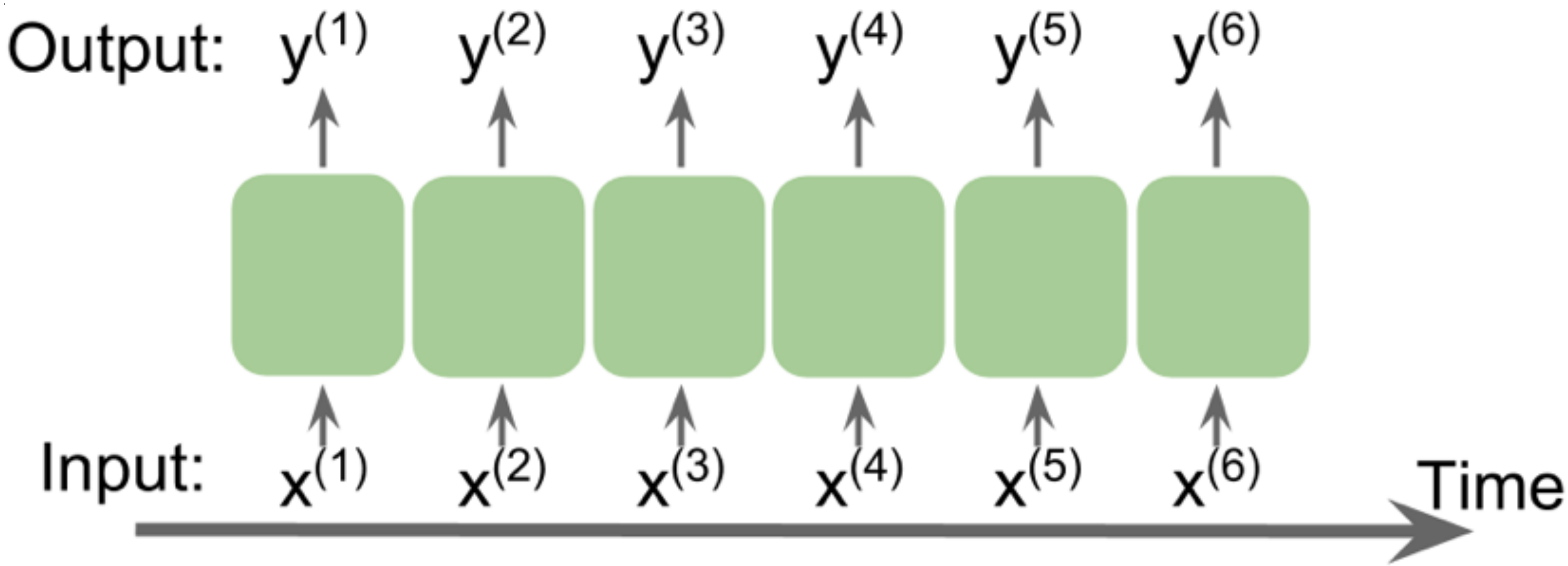
- Introducing Sequential Data
- Recurrent Neural Networks for Modeling Sequences
- Implementing a Multilayer RNN for Sequence Modeling in TensorFlow

Introducing Sequential Data

Modeling Sequential Data

- Elements in a sequence appear in a certain order, and are not independent of each other

$$\left(x^{(1)}, x^{(2)}, \ldots, x^{(T)}\right)$$

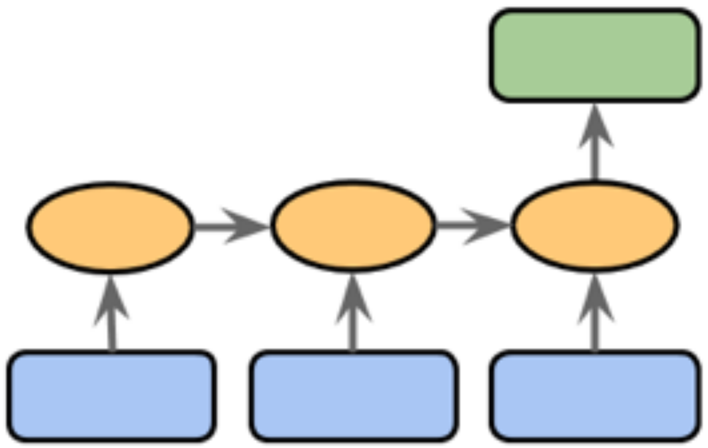


- RNN can remember past information and process new events accordingly

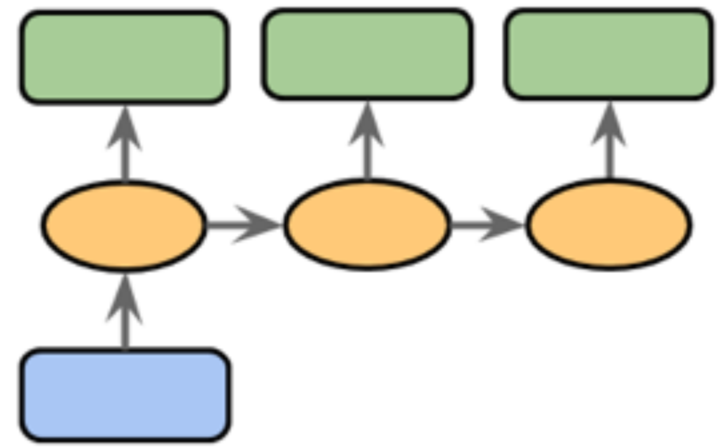
Different Categories of Sequence Modeling

- Application examples
  - language translation, image captioning, text generation
- If either input or output is a sequence, three different categories
  - Many-to-one: input:sequence, output: a fixed size vector.
    - Sentiment analysis: input:text-based, output: class label
  - One-to-many: input: standard format, output: sequence
    - Image captioning: input: image, output: an English phrase
  - Many-to-many: both input/output are sequences
    - **Synchronized** many-to-many: Video classification
    - **Delayed** many-to-many: Language translation

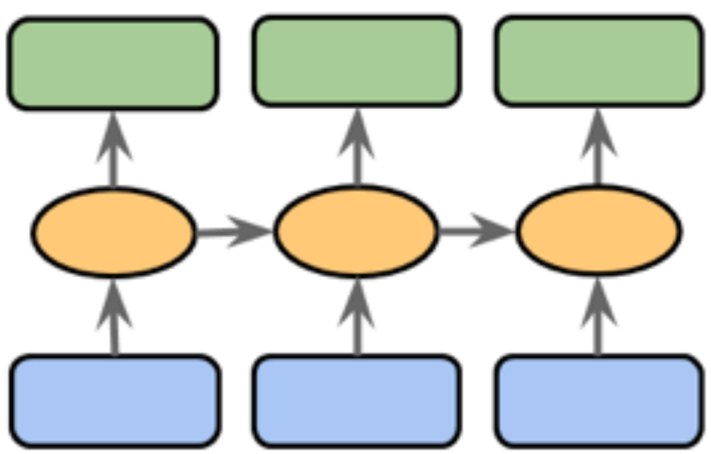
Different Categories of Sequence Modeling



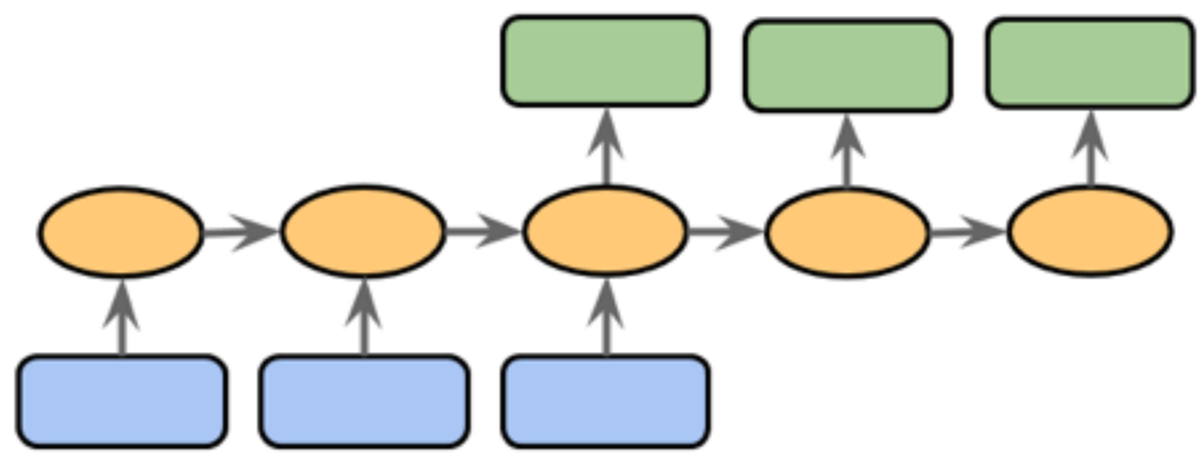
many-to-one



one-to-many



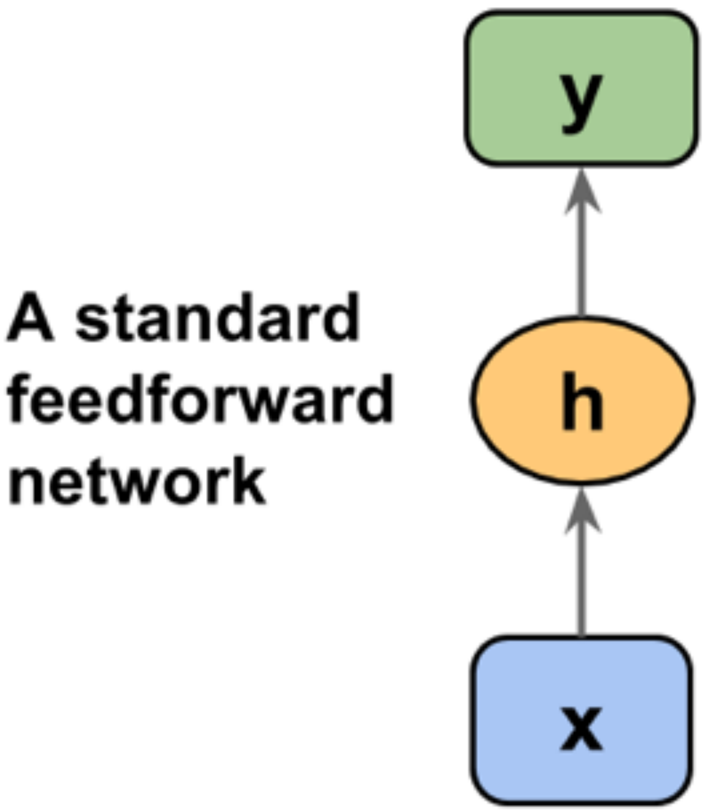
many-to-many

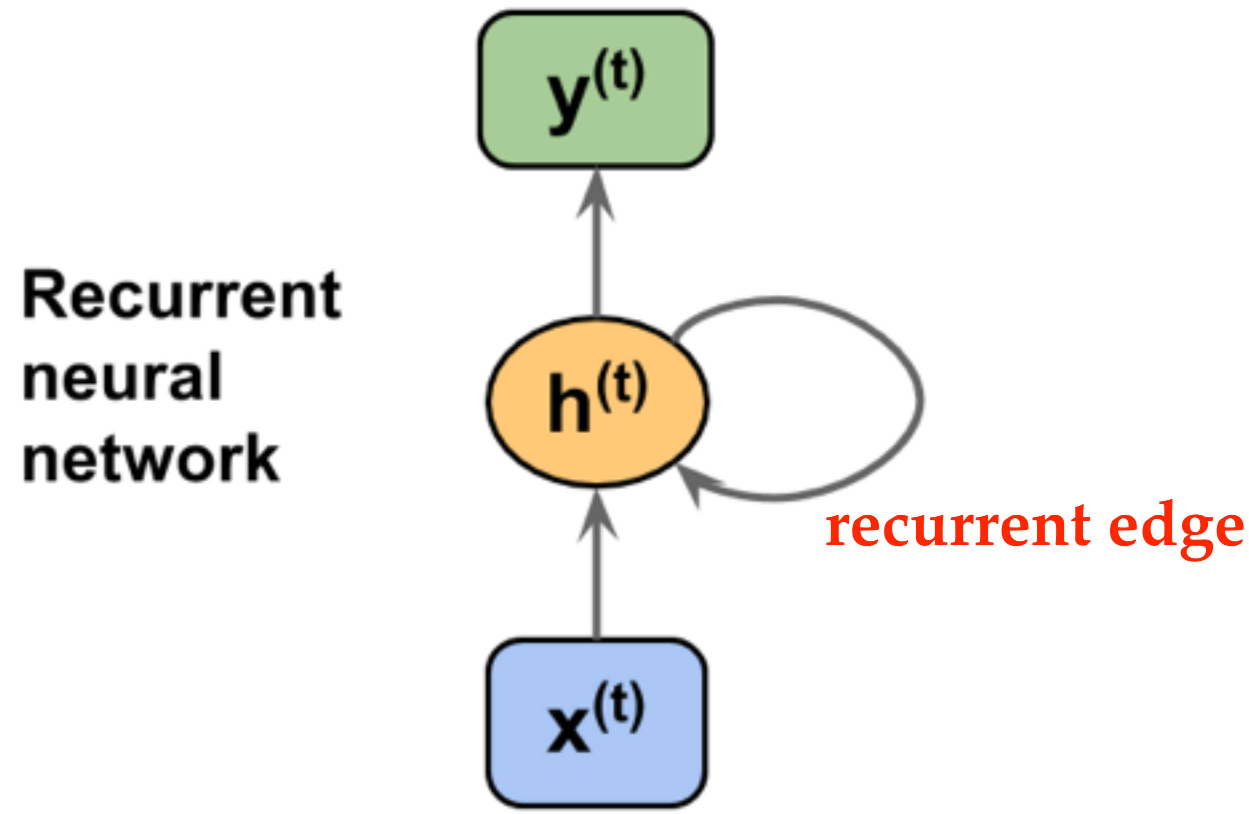


many-to-many

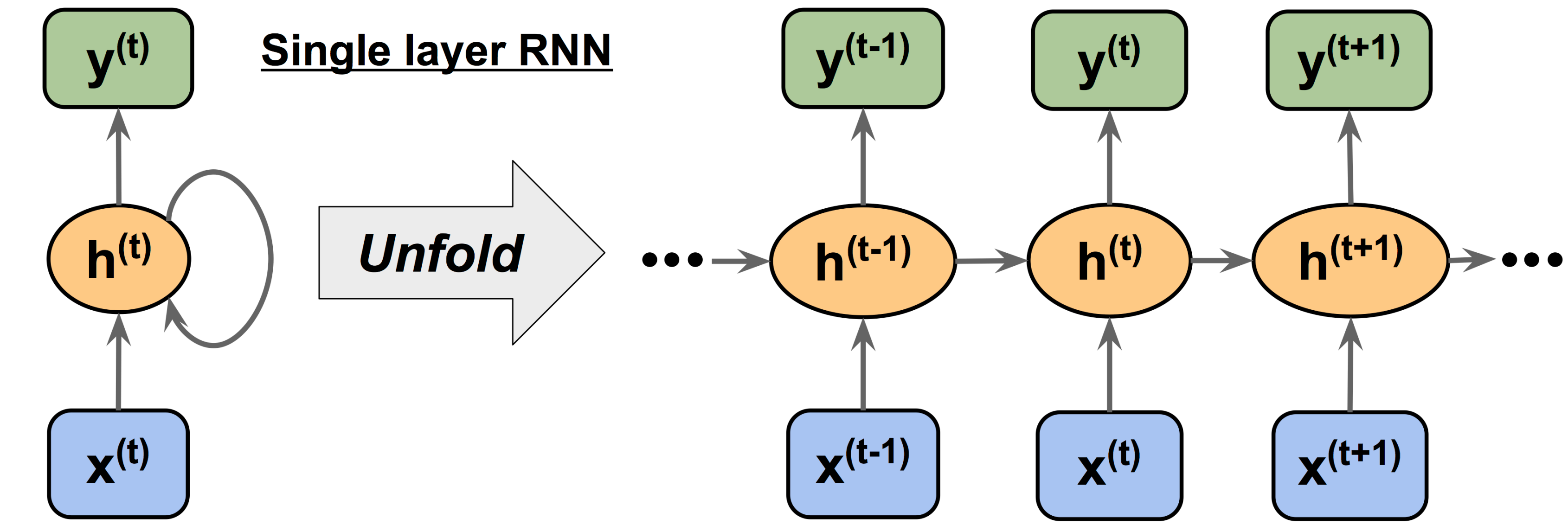
Recurrent Neural Networks for Modeling Sequences

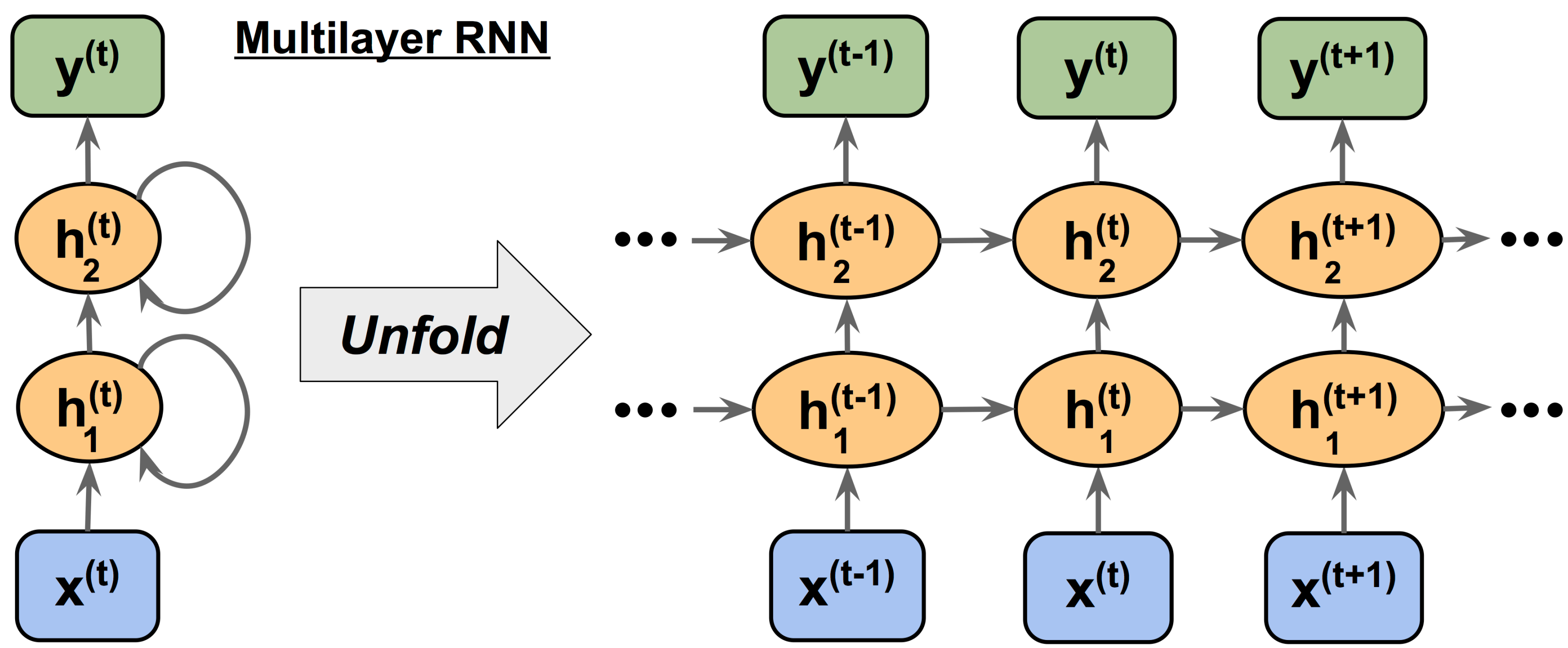
Comparison between Standard Feedforward NN and RNN

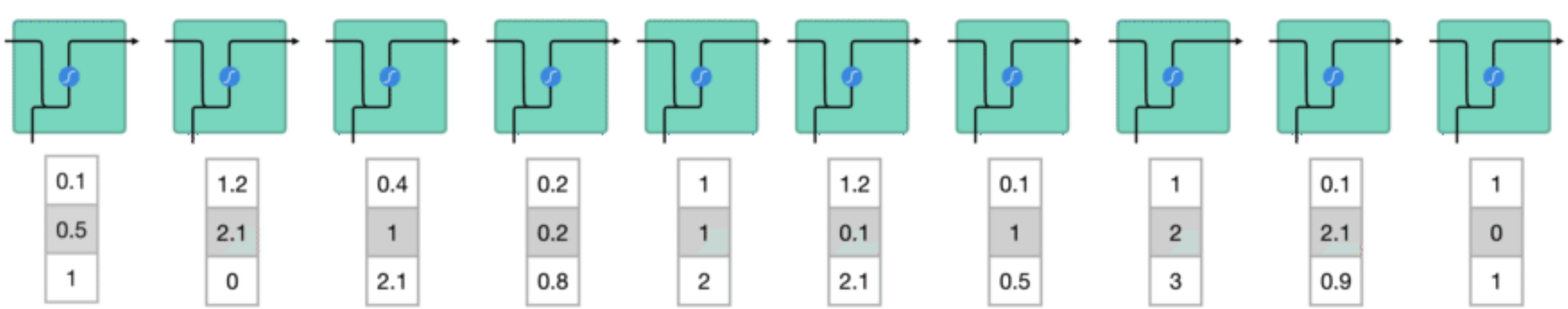




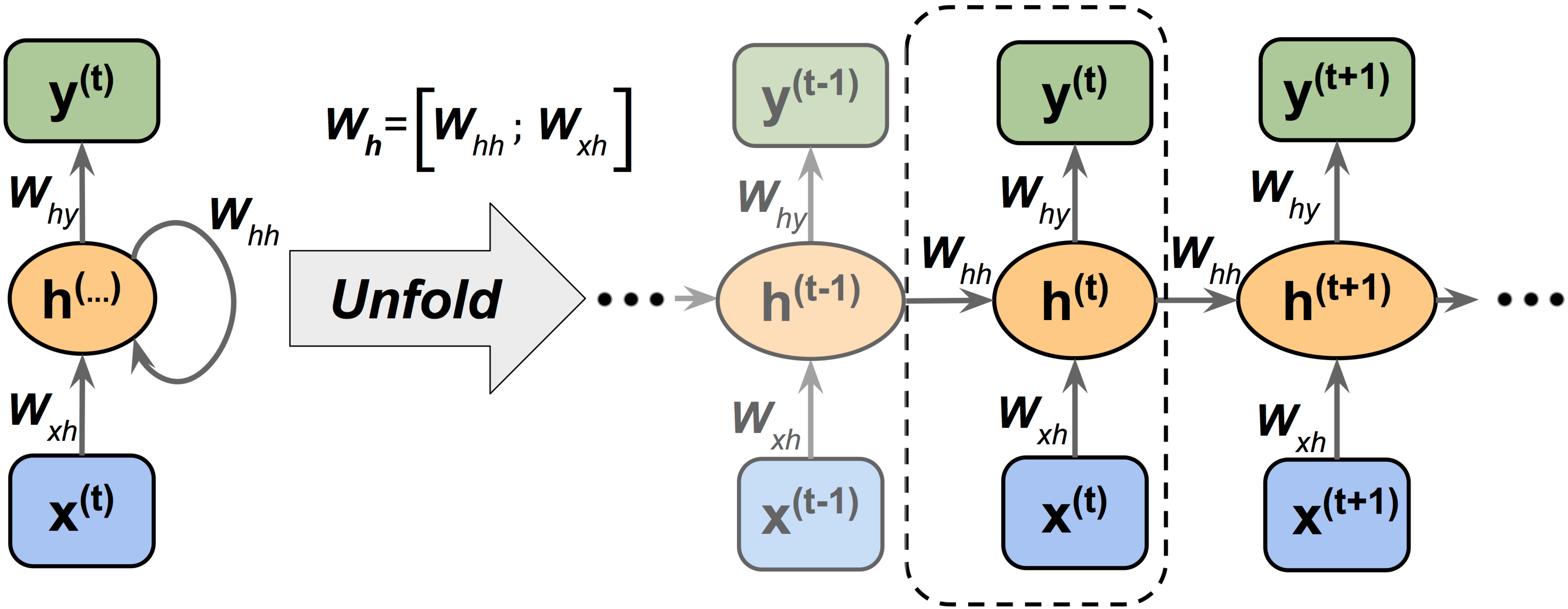
Unrolled RNNs





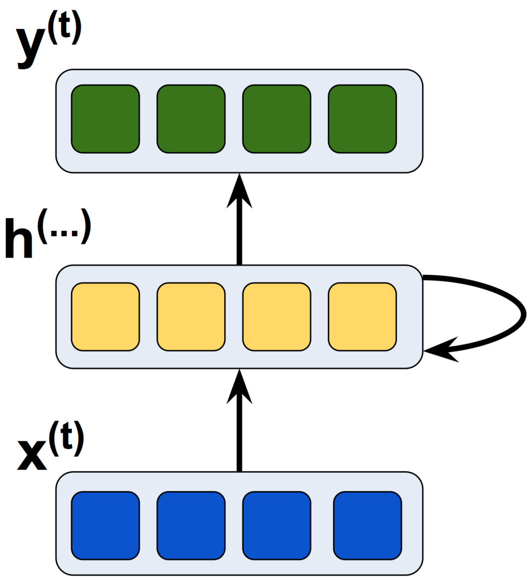


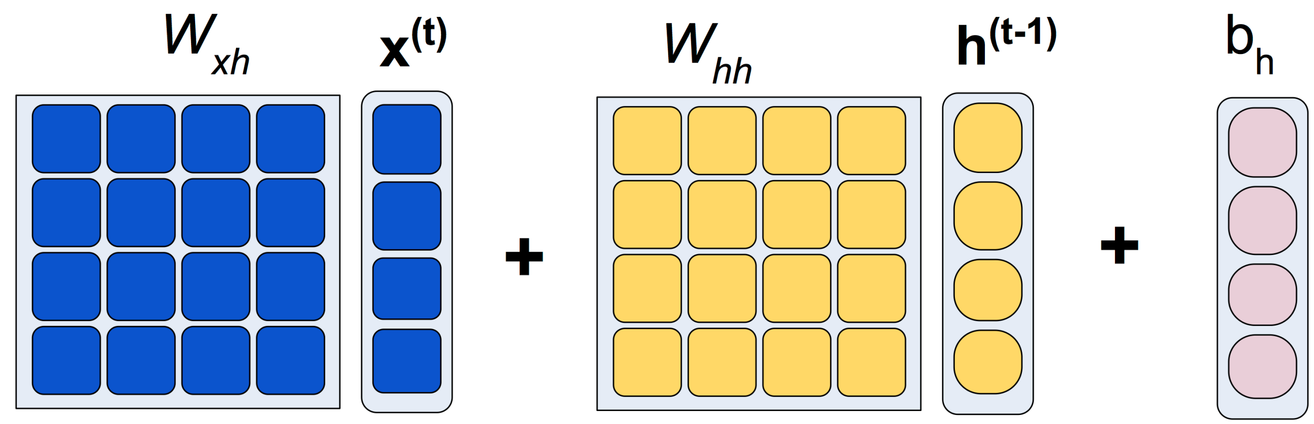
Computing Activations in RNNs



Computing Activations in RNNs

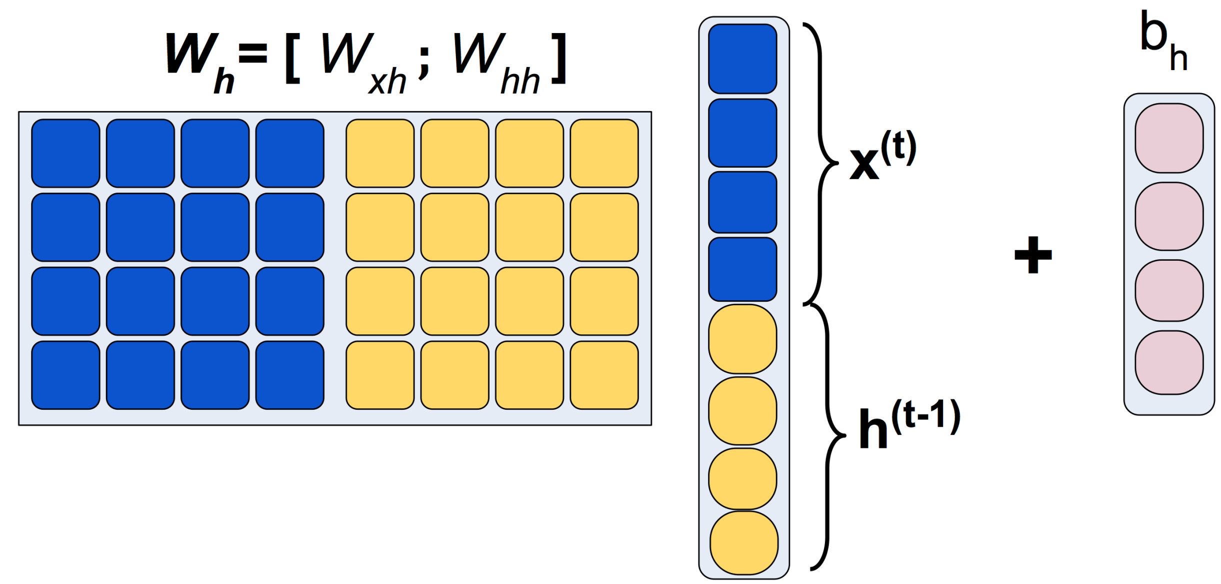
Formulation 1: $\mathbf{h}^{(t)} = \phi_h(W_{xh}\mathbf{x}^{(t)} + W_{hh}\mathbf{h}^{(t-1)} + b_h)$





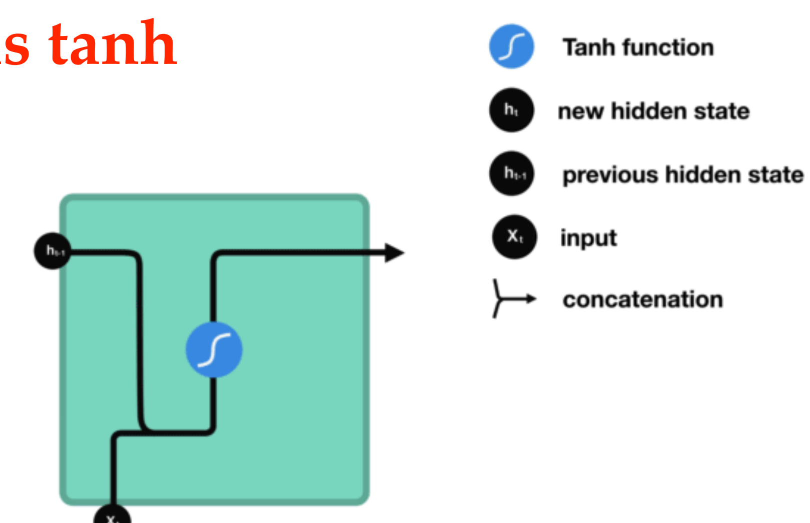
Formulation 2: $\mathbf{h}^{(t)} = \phi_h(\boldsymbol{W_h}\ [\mathbf{x}^{(t)}; \mathbf{h}^{(t-1)}]^T + b_h)$

$\boldsymbol{W_h} = [\ W_{xh}\ ;\ W_{hh}\ ]$



Final Output: $\mathbf{y}^{(t)} = \phi_y(W_{hy}\ \mathbf{h}^{(t)} + b_y)$

usually the activation function is tanh



Training RNNs Using BPTT

- **Backpropagation through time**
  - Overall loss

$$L = \sum_{t=1}^{T} L^{(t)}$$

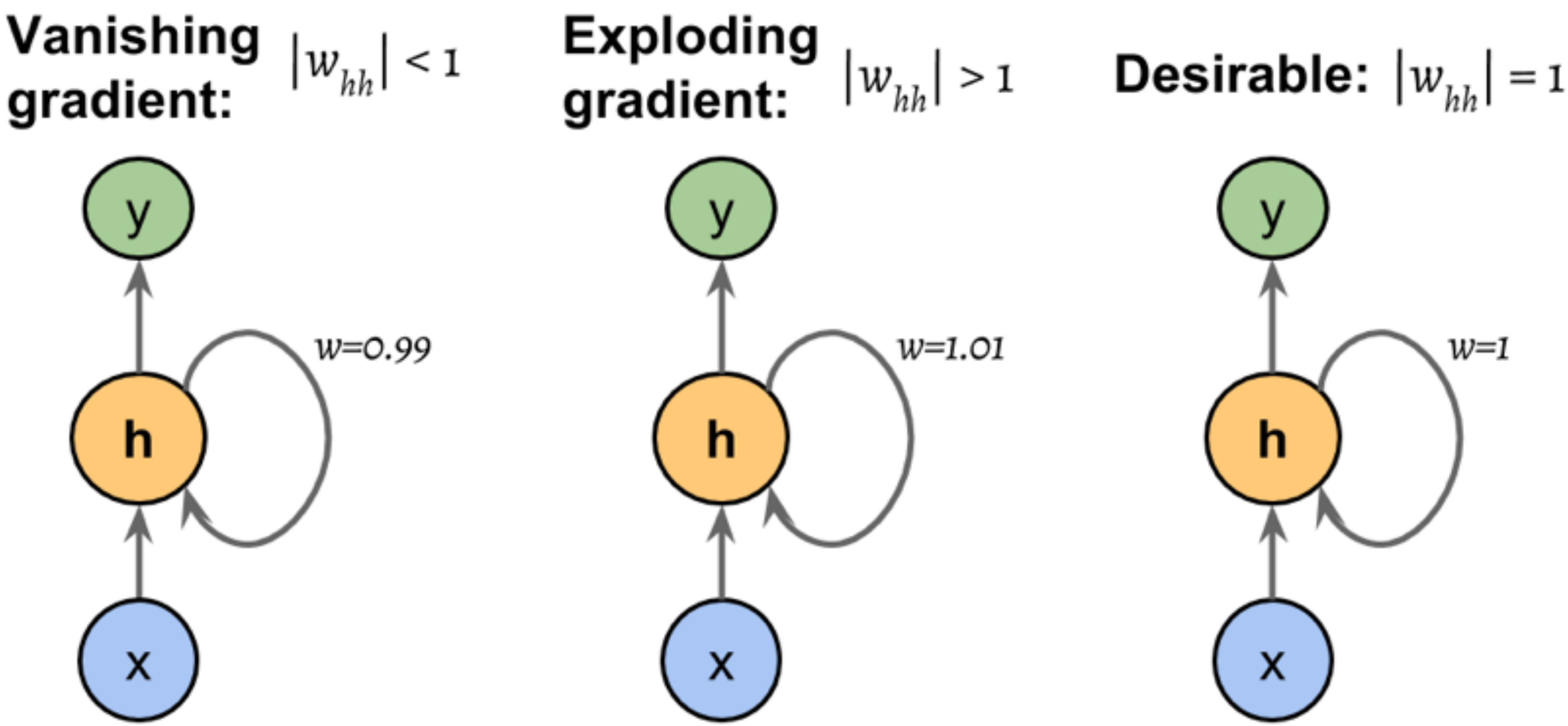
  - Derivation of the gradient

$$\frac{\partial L^{(t)}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \boldsymbol{y}^{(t)}} \times \frac{\partial \boldsymbol{y}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \times \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}_{hh}} \right)$$

  - $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ is computed as a multiplication of adjacent time steps $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$

Gradient Problems

- Vanishing or exploding gradient when *t-k* is large

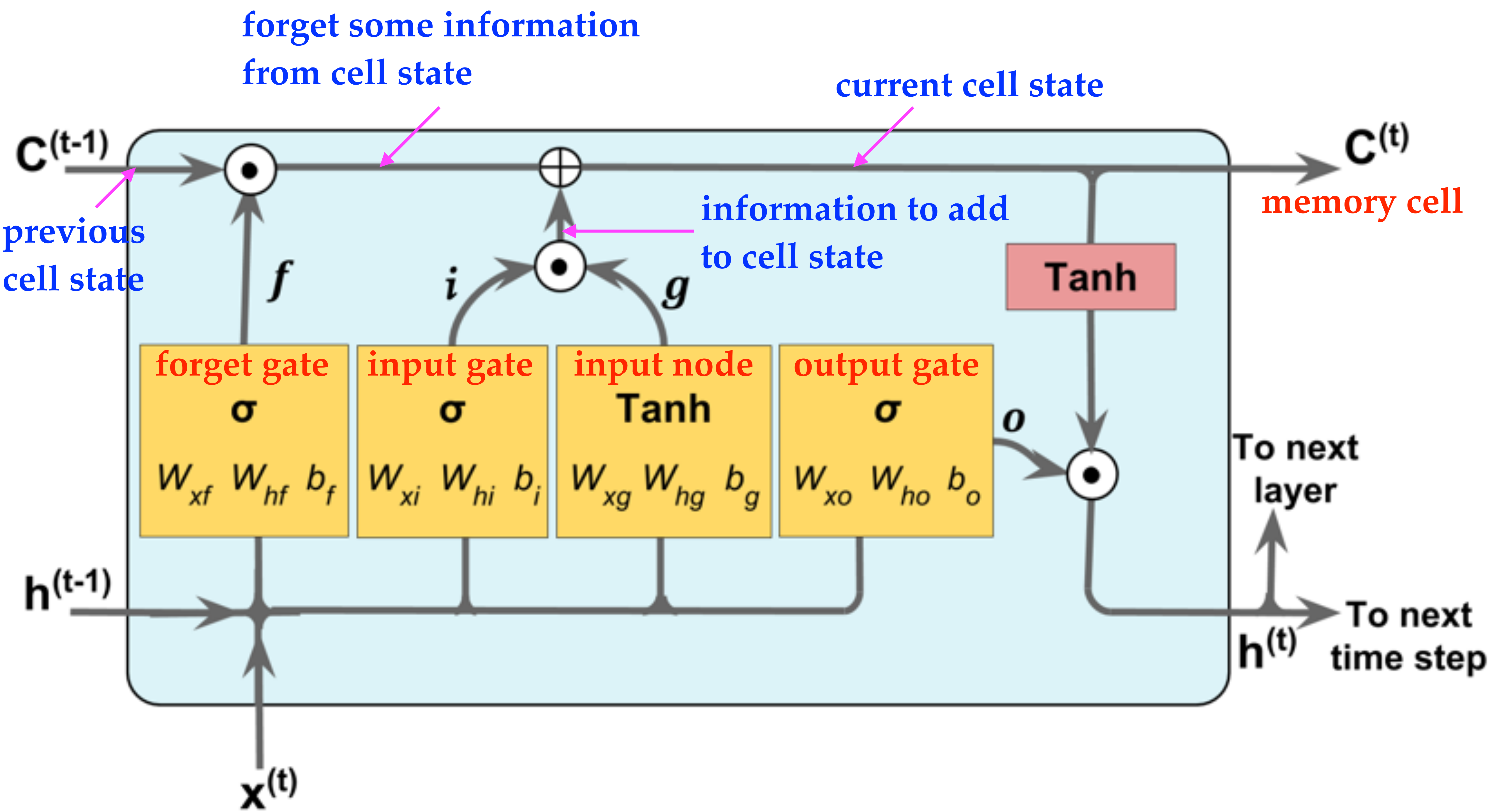


- Two practical solutions
  - Truncated back propagation through time (TBPTT)
  - Long short-term memory (LSTM)

Long Short-Term Memory (LSTM)

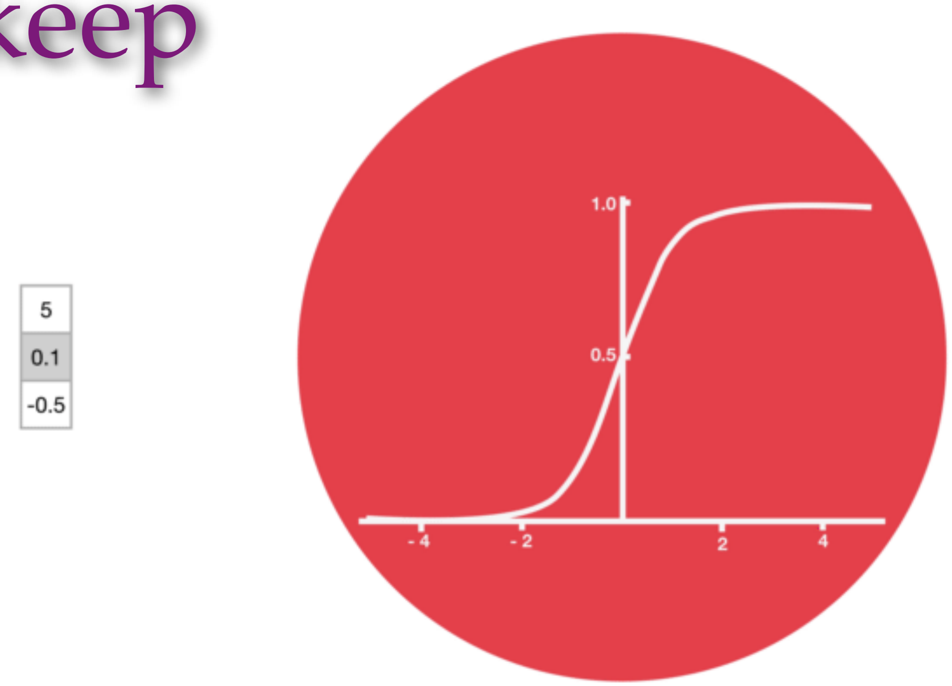
- Core concept
  - cell state + three gates (forget, input, output)
  - cell state: memory of the network
  - The forget gate decides what is relevant to keep from prior steps
  - The input gate decides what information is relevant to add from the current step
  - The output gate determines what the next hidden state should be

LSTM Units


forget some information
from cell state
current cell state
$C^{(t-1)}$
$C^{(t)}$
memory cell
previous
cell state
information to add
to cell state
f
i
g
Tanh
forget gate
σ
W_{xf} W_{hf} b_f
input gate
σ
W_{xi} W_{hi} b_i
input node
Tanh
W_{xg} W_{hg} b_g
output gate
σ
W_{xo} W_{ho} b_o
o
To next
layer
$h^{(t-1)}$
$h^{(t)}$
To next
time step
$x^{(t)}$

Sigmoid

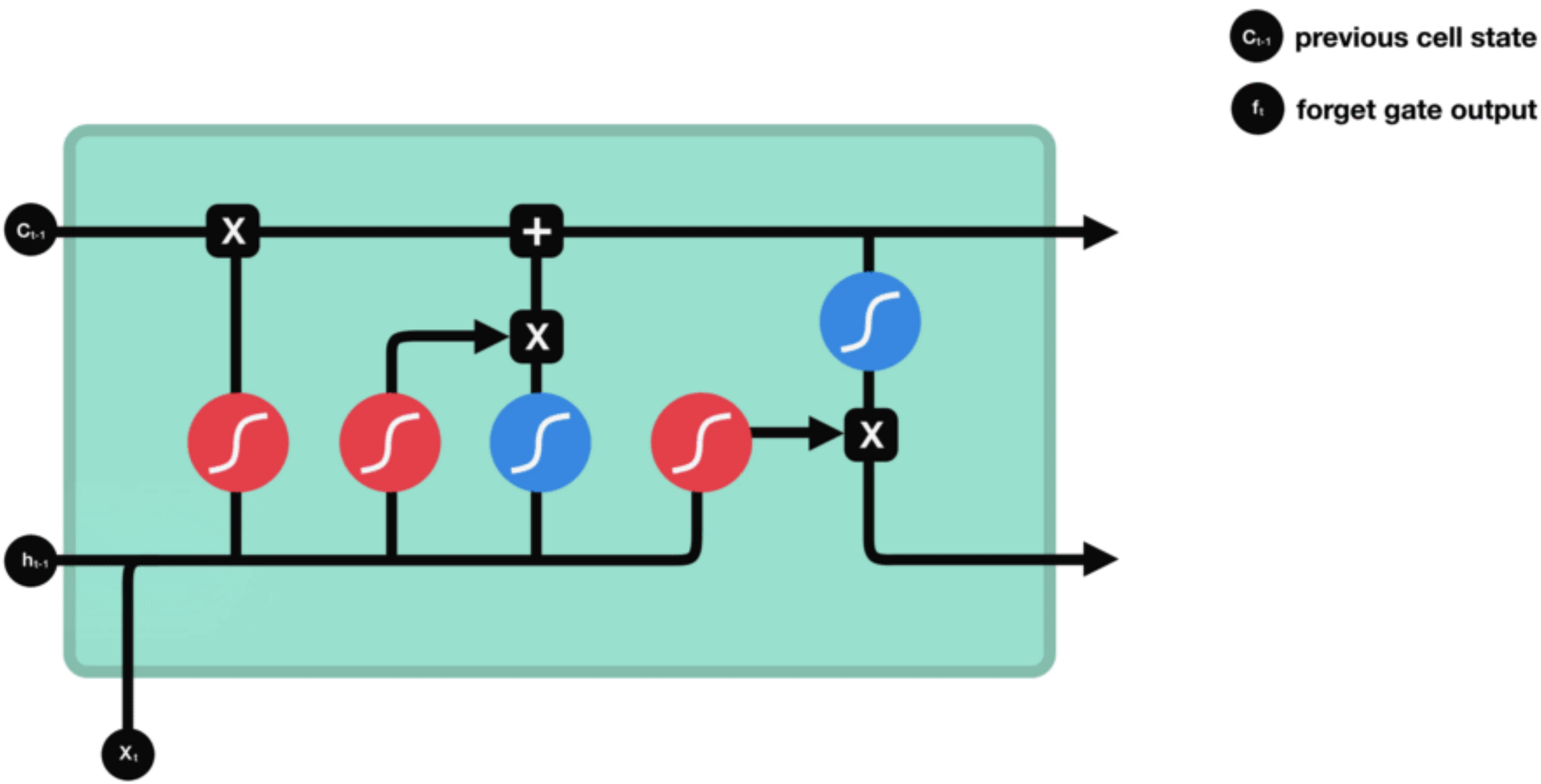
- Sigmoid activation can squash values between 0 and 1 to help to update or forget data
  - Data multiplied by 0 is 0: to be forgotten
  - Data multiplied by 1 is the same value: to be kept
- The network can learn which data is not important so can be forgotten or which data is important to keep



Forget Gate

- The gate decides what information should be forgotten or kept

$$f_t = \sigma\left(W_{xf} x^{(t)} + W_{hf} h^{(t-1)} + b_f\right)$$

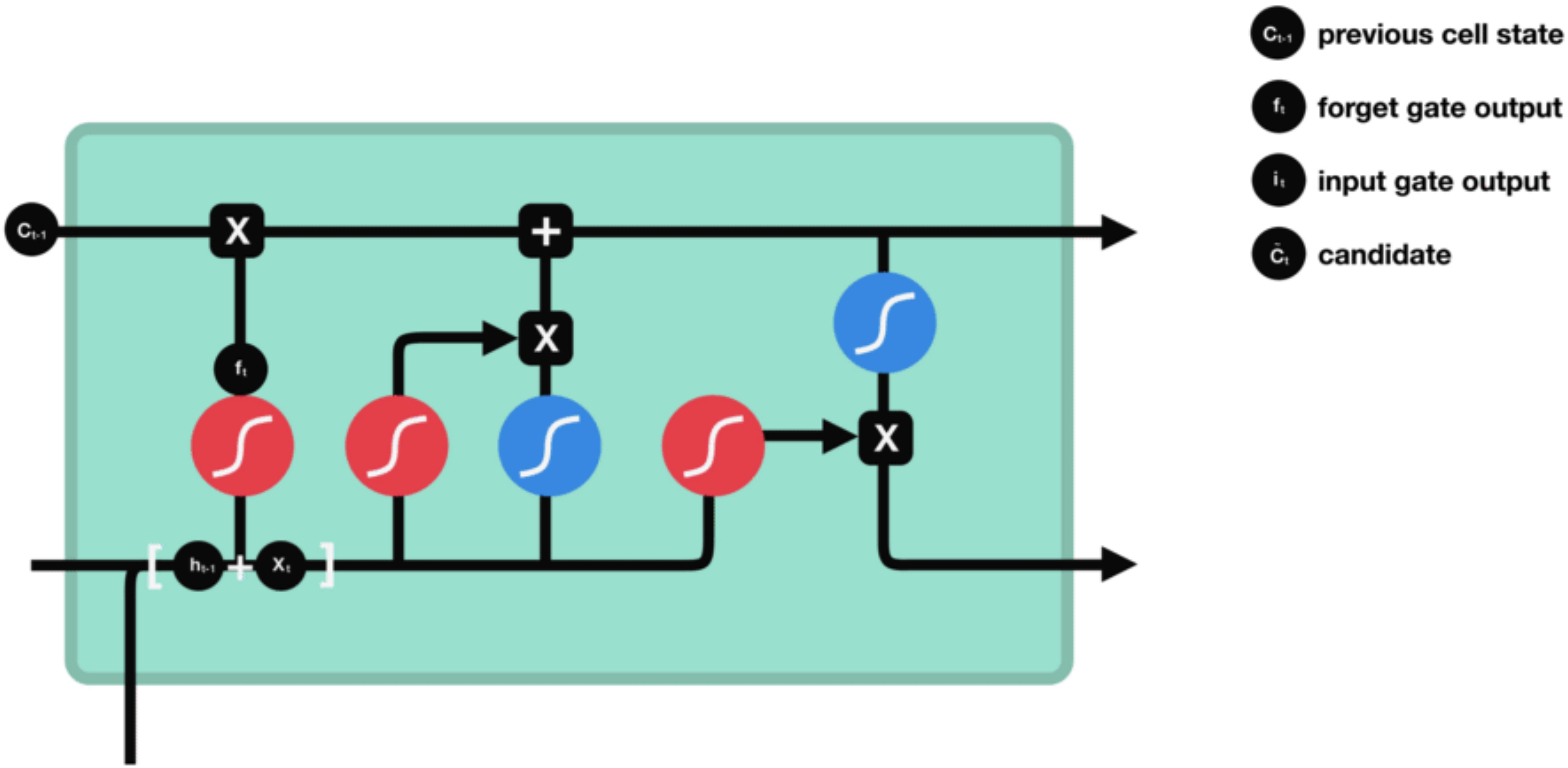


Input Gate

- The input gate updates the cell state

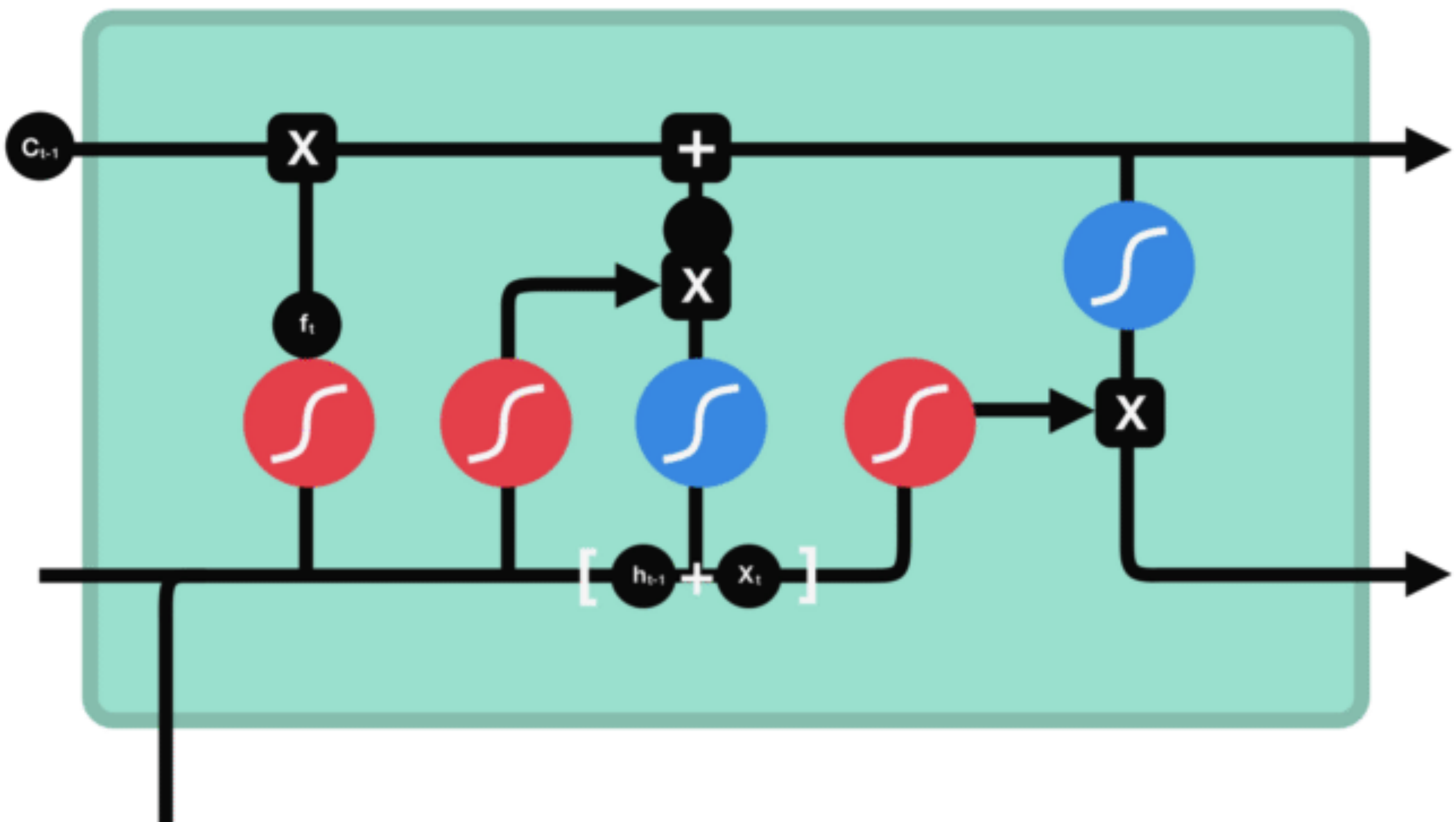
$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right)$$ decide which values to be update

$$\boldsymbol{g}_t = \tanh\left(\boldsymbol{W}_{xg}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hg}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_g\right)$$ candidate values to be added to the state



Cell State

$$C^{(t)} = \left(C^{(t-1)} \odot f_t \right) \oplus \left(i_t \odot g_t \right)$$



- C_{t-1} previous cell state
- f_t forget gate output
- i_t input gate output
- $\tilde{C}_t$ candidate
- C_t new cell state

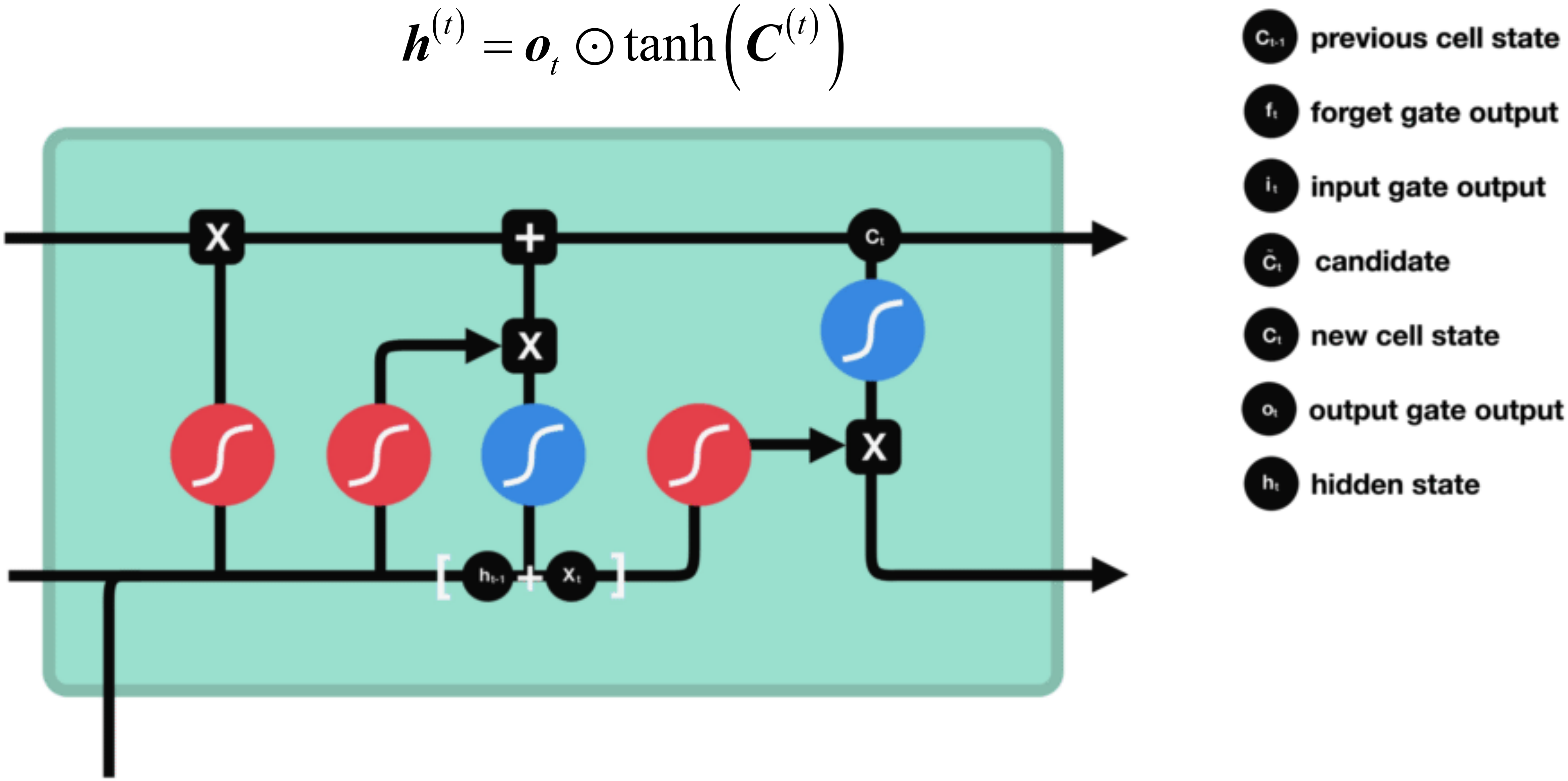
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output Gate

- The gate decides what the next hidden state should be

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right)$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}^{(t)}\right)$$



Implementing a Multilayer RNN for Sequence Modeling in TensorFlow

- Sentiment Analysis
- Language Modeling

Sentimental Analysis

Preparing the Data (IMDb)

- A multilayer RNN with many-to-one architecture
  - Encode the '*review*' input data into numerical values
    - Find unique words in the entire dataset (Counter)
    - Create a dictionary to map each unique word into a unique integer number
  - To confirm all sequences have the same length, define a hyperparameter ***sequence_length***, and fill the index of words in each sequence from the right-hand side of the matrix (others fill with zeros)

Preparing the Data

Text Corpus

Extract indices of unique words

21, 88, 19, 26
14, 56, 4, 6, 2, 11, 10, 33, 27, 38, 70, 76
39, 29, 28, 24, 11, 5, 78, 39
77, 63, 22, 78, 34, 25, 67, 4, 83, 17, 23
19, 14, 8, 61, 23, 24, 4
23, 42, 18
4, 8, 11, 25, 23, 42, 84, 76, 45, 24
45, 13, 68, 92, 33, 15, 16, 76, 25, 33, 89, 40, 16

A matrix of all zeros

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Filling sequences from the right side

	0	0	0	0	0	0	**21**	**88**	**19**	**26**
~~14, 56,~~	**4**	**6**	**2**	**11**	**10**	**33**	**27**	**38**	**70**	**76**
	0	**0**	**39**	**29**	**28**	**24**	**11**	**5**	**78**	**39**
~~77,~~	**63**	**22**	**78**	**34**	**25**	**67**	**4**	**83**	**17**	**23**
	0	0	0	**19**	**14**	**8**	**61**	**23**	**24**	**4**
	0	0	0	0	0	0	0	**23**	**42**	**18**
	4	**8**	**11**	**25**	**23**	**42**	**84**	**76**	**45**	**24**
~~45, 13, 68,~~	**92**	**33**	**15**	**16**	**76**	**25**	**33**	**89**	**40**	**16**

Read in the IMDb Data

```
import pyprind
import pandas as pd
from string import punctuation
import re
import numpy as np


df = pd.read_csv('movie_data.csv', encoding='utf-8')
print(df.head(3))
```

```
                                              review  sentiment
0  In 1974, the teenager Martha Moxley (Maggie Gr...          1
1  OK... so... I really like Kris Kristofferson a...          0
2  ***SPOILER*** Do not read this, if you think a...          0
```

Count the Unique Word in the Dataset

```
## Preprocessing the data:
## Separate words and
## count each word's occurrence


from collections import Counter


counts = Counter()
pbar = pyprind.ProgBar(len(df['review']),
                       title='Counting words occurences')
for i,review in enumerate(df['review']):
    text = ''.join([c if c not in punctuation else ' '+c+' ' \
                    for c in review]).lower()
    df.loc[i,'review'] = text
    pbar.update()
    counts.update(text.split())
```

```
Counting words occurences
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:03:19
```

Create the Word to Integer Mapping

```python
## Create a mapping:
## Map each unique word to an integer

word_counts = sorted(counts, key=counts.get, reverse=True)
print(word_counts[:5])
word_to_int = {word: ii for ii, word in enumerate(word_counts, 1)}


mapped_reviews = []
pbar = pyprind.ProgBar(len(df['review']),
                       title='Map reviews to ints')
for review in df['review']:
    mapped_reviews.append([word_to_int[word] for word in review.split()])
    pbar.update()
```

```
Map reviews to ints
```

```
['the', '.', ',', 'and', 'a']
```

```
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:00:03
```

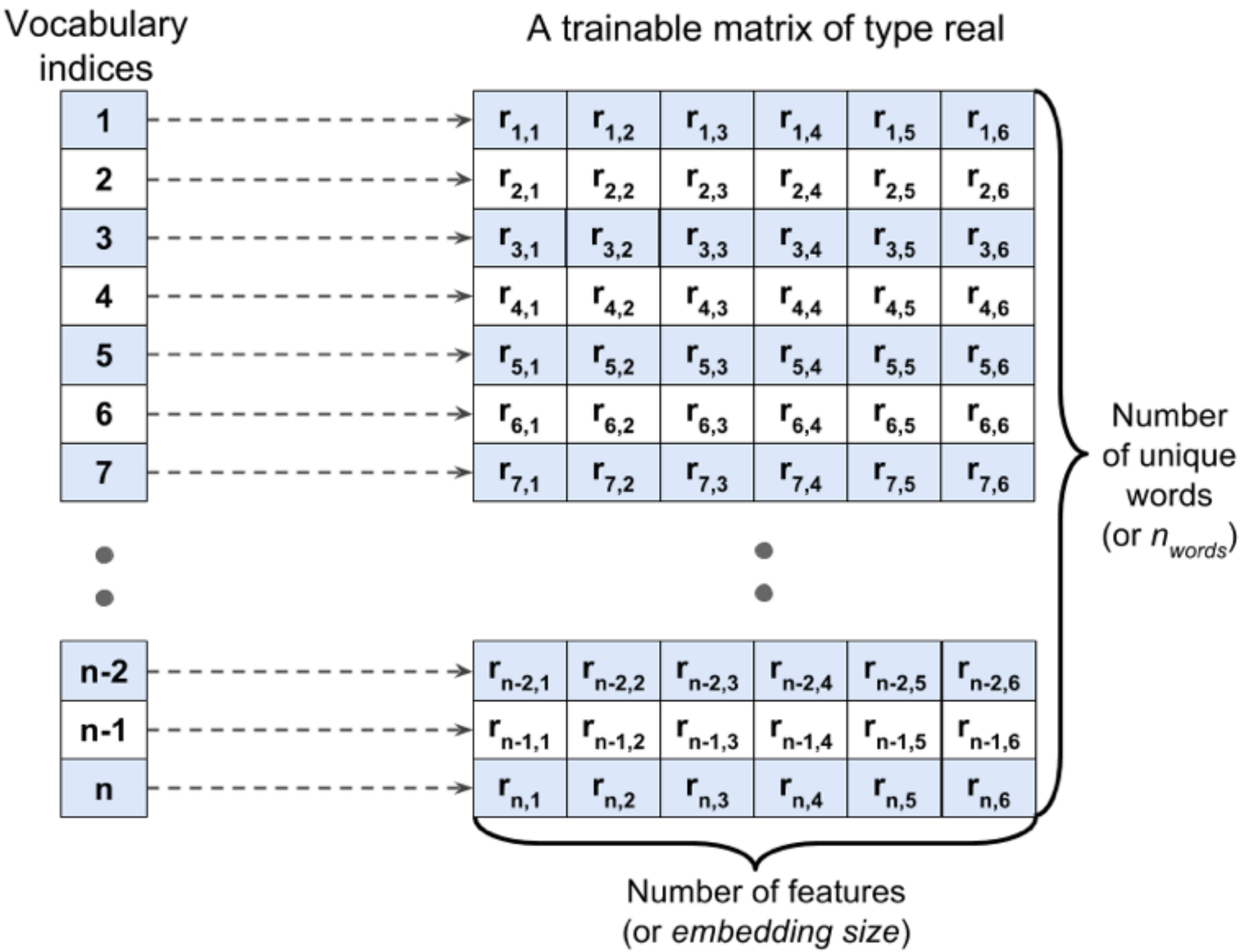
Prepare Fixed-Length Sequences

```
sequence_length = 200  ## sequence length (or T in our formulas)
sequences = np.zeros((len(mapped_reviews), sequence_length), dtype=int)
for i, row in enumerate(mapped_reviews):
    review_arr = np.array(row)
    sequences[i, -len(row):] = review_arr[-sequence_length:]

X_train = sequences[:25000, :]
y_train = df.loc[:25000, 'sentiment'].values
X_test = sequences[25000:, :]
y_test = df.loc[25000:, 'sentiment'].values



np.random.seed(123) # for reproducibility

## Function to generate minibatches:
def create_batch_generator(x, y=None, batch_size=64):
    n_batches = len(x)//batch_size
    x= x[:n_batches*batch_size]
    if y is not None:
        y = y[:n_batches*batch_size]
    for ii in range(0, len(x), batch_size):
        if y is not None:
            yield x[ii:ii+batch_size], y[ii:ii+batch_size]
        else:
            yield x[ii:ii+batch_size]
```

Embedding (Input Feature Encoding)

- The word indices to be converted into input features
  - One-hot encoding (too many features may suffer from curse of dimensionality, very sparse)
  - Embedding: use finite-sized vectors to represent an infinite number of real numbers
    - A reduction in the dimensionality of the feature space to decrease the effect of the curse of dimensionality
    - The extraction of salient features since the embedding layer in a neural network is trainable

Embedding


Vocabulary indices
A trainable matrix of type real
1
2
3
4
5
6
7
n-2
n-1
n
$r_{1,1}$ $r_{1,2}$ $r_{1,3}$ $r_{1,4}$ $r_{1,5}$ $r_{1,6}$
$r_{2,1}$ $r_{2,2}$ $r_{2,3}$ $r_{2,4}$ $r_{2,5}$ $r_{2,6}$
$r_{3,1}$ $r_{3,2}$ $r_{3,3}$ $r_{3,4}$ $r_{3,5}$ $r_{3,6}$
$r_{4,1}$ $r_{4,2}$ $r_{4,3}$ $r_{4,4}$ $r_{4,5}$ $r_{4,6}$
$r_{5,1}$ $r_{5,2}$ $r_{5,3}$ $r_{5,4}$ $r_{5,5}$ $r_{5,6}$
$r_{6,1}$ $r_{6,2}$ $r_{6,3}$ $r_{6,4}$ $r_{6,5}$ $r_{6,6}$
$r_{7,1}$ $r_{7,2}$ $r_{7,3}$ $r_{7,4}$ $r_{7,5}$ $r_{7,6}$
$r_{n-2,1}$ $r_{n-2,2}$ $r_{n-2,3}$ $r_{n-2,4}$ $r_{n-2,5}$ $r_{n-2,6}$
$r_{n-1,1}$ $r_{n-1,2}$ $r_{n-1,3}$ $r_{n-1,4}$ $r_{n-1,5}$ $r_{n-1,6}$
$r_{n,1}$ $r_{n,2}$ $r_{n,3}$ $r_{n,4}$ $r_{n,5}$ $r_{n,6}$
Number of unique words (or n_{words})
Number of features (or *embedding size*)

Create an Embedded Layer

- Create an embedded layer with input layer tf_x
  - Create a matrix of size [**n_words** x **n_embedding_size**] as a tensor variable (*embedding*) and initialize its elements randomly with floats between [-1,1]

```
embedding = tf.Variable(
                tf.random_uniform(
                    shape=(n_words, embedding_size),
                    minval=-1, maxval=1)
            )
```

  - Use *tf.nn.embedding_lookup* function to look up the row in the embedded matrix associated with each element of *tf_x*

```
embed_x = tf.nn.embedding_lookup(embedding, tf_x)
```

Building an RNN Model

- SentimentRNN class
  - A ***constructor*** to set all the model parameters, create a computation graph and call the self.build to build the multilayer RNN
  - ***build***: Declare 3 placeholders (input data, input labels, and the keep-probability for the dropout configuration of the hidden layer), create an embedded layer and build the RNN using the embedded representation as input.
  - ***train***: Create a TensorFlow session and save the model after 10 epochs for checkpointing
  - ***predict***: Create a new session, restore the last checkpoint and carry out the predictions for the test data

SentimentRNN: the constructor

```
import tensorflow as tf


class SentimentRNN(object):
    def __init__(self, n_words, seq_len=200,
                 lstm_size=256, num_layers=1, batch_size=64,
                 learning_rate=0.0001, embed_size=200):
        self.n_words = n_words
        self.seq_len = seq_len
        self.lstm_size = lstm_size   ## number of hidden units
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.embed_size = embed_size

        self.g = tf.Graph()
        with self.g.as_default():
            tf.set_random_seed(123)
            self.build()
            self.saver = tf.train.Saver()
            self.init_op = tf.global_variables_initializer()
```

SentimentRNN: build() (1/4)

```
    def build(self):
        ## Define the placeholders
        tf_x = tf.placeholder(tf.int32,
                    shape=(self.batch_size, self.seq_len),
                    name='tf_x')
        tf_y = tf.placeholder(tf.float32,
                    shape=(self.batch_size),
                    name='tf_y')
        tf_keepprob = tf.placeholder(tf.float32,
                    name='tf_keepprob')
        ## Create the embedding layer
        embedding = tf.Variable(
                    tf.random_uniform(
                        (self.n_words, self.embed_size),
                        minval=-1, maxval=1),
                    name='embedding')
        embed_x = tf.nn.embedding_lookup(
                    embedding, tf_x,
                    name='embeded_x')
```

SentimentRNN: build() (2/4)

```
## Define LSTM cell and stack them together
cells = tf.contrib.rnn.MultiRNNCell(      3. Make a list of such cells
        [tf.contrib.rnn.DropoutWrapper(      2. Apply the dropout to the RNN cells
           tf.contrib.rnn.BasicLSTMCell(self.lstm_size),      1. create RNN cells
           output_keep_prob=tf_keepprob)
         for i in range(self.num_layers)])

## Define the initial state:
self.initial_state = cells.zero_state(
         self.batch_size, tf.float32)
print('  << initial state >> ', self.initial_state)
# Create RNN using the RNN cells and their states
lstm_outputs, self.final_state = tf.nn.dynamic_rnn(
         cells, embed_x,
         initial_state=self.initial_state)
## Note: lstm_outputs shape:
##  [batch_size, max_time, cells.output_size]
print('\n  << lstm_output   >> ', lstm_outputs)
print('\n  << final state   >> ', self.final_state)
```

SentimentRNN: build() (3/4)

```
## Apply a FC layer after on top of RNN output:
logits = tf.layers.dense(
         inputs=lstm_outputs[:, -1],
         units=1, activation=None,
         name='logits')

logits = tf.squeeze(logits, name='logits_squeezed')
print ('\n  << logits        >> ', logits)

y_proba = tf.nn.sigmoid(logits, name='probabilities')
predictions = {
    'probabilities': y_proba,
    'labels' : tf.cast(tf.round(y_proba), tf.int32,
         name='labels')
}
print('\n  << predictions   >> ', predictions)
```

SentimentRNN: build() (4/4)

```
        ## Define the cost function
        cost = tf.reduce_mean(
                 tf.nn.sigmoid_cross_entropy_with_logits(
                 labels=tf_y, logits=logits),
                 name='cost')

        ## Define the optimizer
        optimizer = tf.train.AdamOptimizer(self.learning_rate)
        train_op = optimizer.minimize(cost, name='train_op')
```

SentimentRNN: train()

```
    def train(self, X_train, y_train, num_epochs):
        with tf.Session(graph=self.g) as sess:
            sess.run(self.init_op)
            iteration = 1
            for epoch in range(num_epochs):
                state = sess.run(self.initial_state)

                for batch_x, batch_y in create_batch_generator(
                            X_train, y_train, self.batch_size):
                    feed = {'tf_x:0': batch_x,
                            'tf_y:0': batch_y,
                            'tf_keepprob:0': 0.5,
                            self.initial_state : state}
                    loss, _, state = sess.run(
                            ['cost:0', 'train_op',
                             self.final_state],
                            feed_dict=feed)

                    if iteration % 20 == 0:
                        print("Epoch: %d/%d Iteration: %d "
                              "| Train loss: %.5f" % (
                               epoch + 1, num_epochs,
                               iteration, loss))

                    iteration +=1
                if (epoch+1)%10 == 0:
                    self.saver.save(sess,
                        "model/sentiment-%d.ckpt" % epoch)
```

SentimentRNN: predict()

```
    def predict(self, X_data, return_proba=False):
        preds = []
        with tf.Session(graph = self.g) as sess:
            self.saver.restore(
                sess, tf.train.latest_checkpoint('model/'))
            test_state = sess.run(self.initial_state)
            for ii, batch_x in enumerate(
                create_batch_generator(
                    X_data, None, batch_size=self.batch_size), 1):
                feed = {'tf_x:0' : batch_x,
                        'tf_keepprob:0': 1.0,
                        self.initial_state : test_state}
                if return_proba:
                    pred, test_state = sess.run(
                        ['probabilities:0', self.final_state],
                        feed_dict=feed)
                else:
                    pred, test_state = sess.run(
                        ['labels:0', self.final_state],
                        feed_dict=feed)

                preds.append(pred)

        return np.concatenate(preds)
```

Instantiate the SentimentRNN Class

```
## Train:

n_words = max(list(word_to_int.values())) + 1

rnn = SentimentRNN(n_words=n_words,
                   seq_len=sequence_length,
                   embed_size=256,
                   lstm_size=128,
                   num_layers=1,
                   batch_size=100,
                   learning_rate=0.001)
```

Training the SentimentRNN Model

```
rnn.train(X_train, y_train, num_epochs=40)
```

```
Epoch: 1/40 Iteration: 20 | Train loss: 0.70637
Epoch: 1/40 Iteration: 40 | Train loss: 0.60539
Epoch: 1/40 Iteration: 60 | Train loss: 0.66977
Epoch: 1/40 Iteration: 80 | Train loss: 0.51997
Epoch: 1/40 Iteration: 100 | Train loss: 0.53567
Epoch: 1/40 Iteration: 120 | Train loss: 0.59073
Epoch: 1/40 Iteration: 140 | Train loss: 0.45970
Epoch: 1/40 Iteration: 160 | Train loss: 0.43817
Epoch: 1/40 Iteration: 180 | Train loss: 0.45852
Epoch: 1/40 Iteration: 200 | Train loss: 0.45753
Epoch: 1/40 Iteration: 220 | Train loss: 0.42869
Epoch: 1/40 Iteration: 240 | Train loss: 0.48586
Epoch: 2/40 Iteration: 260 | Train loss: 0.39664
Epoch: 2/40 Iteration: 280 | Train loss: 0.30718
Epoch: 2/40 Iteration: 300 | Train loss: 0.31172
```

Test and Optimizing the Model

```
## Test:
preds = rnn.predict(X_test)
y_true = y_test[:len(preds)]
print('Test Acc.: %.3f' % (
      np.sum(preds == y_true) / len(y_true)))
```

```
INFO:tensorflow:Restoring parameters from model/sentiment-39.ckpt
Test Acc.: 0.860
```

```
## Get probabilities:
proba = rnn.predict(X_test, return_proba=True)
```

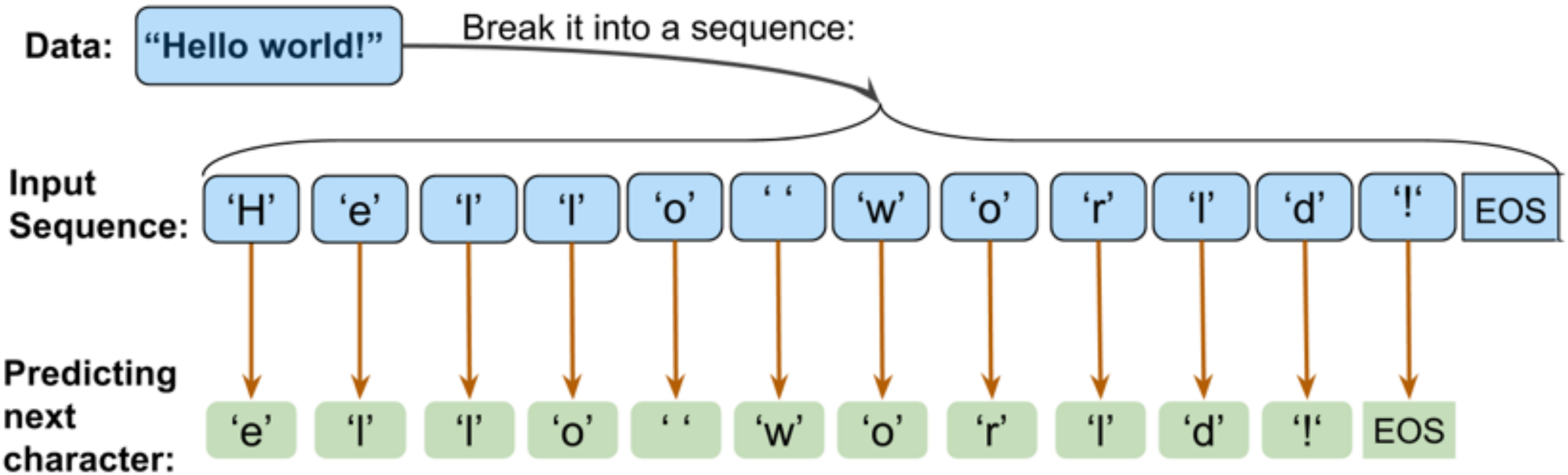
```
INFO:tensorflow:Restoring parameters from model/sentiment-39.ckpt
```

Character-Level Language Modeling

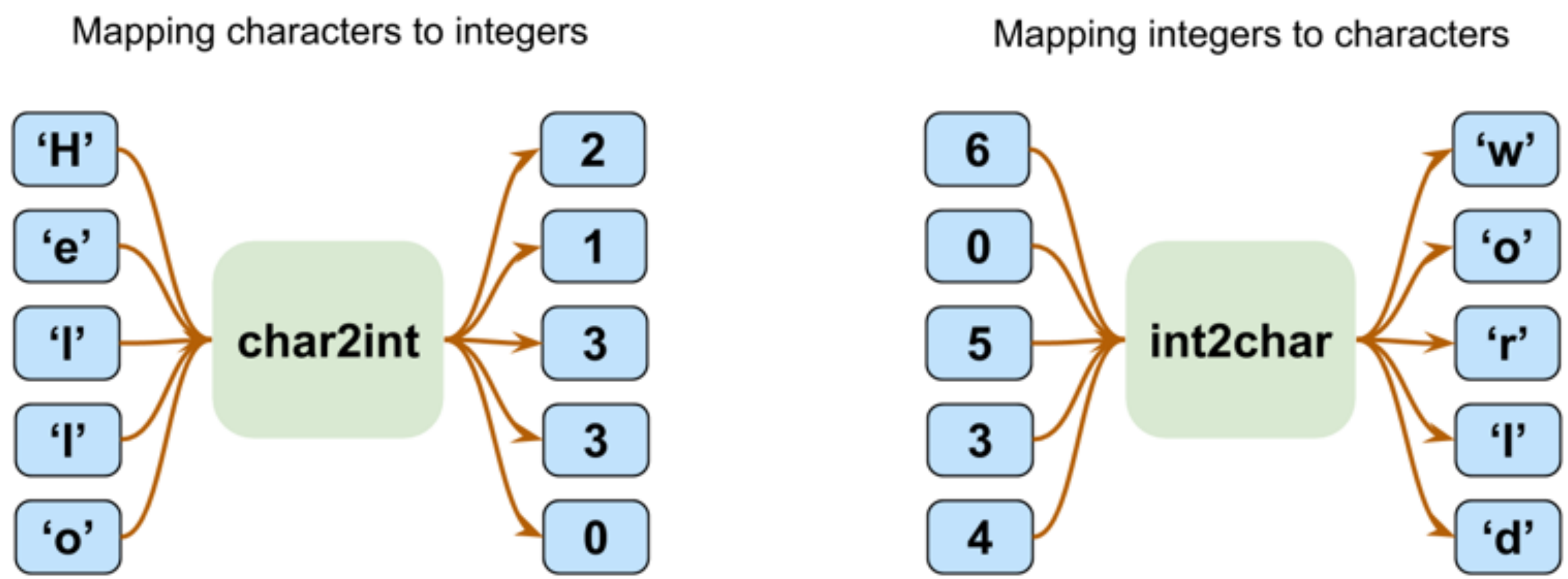
NTHU EE

Laboratory for Reliable Computing

Character-Level Language Modeling



Preparing the Data




```python
import numpy as np


## Reading and processing text
with open('pg2265.txt', 'r', encoding='utf-8') as f:
    text=f.read()

text = text[15858:]
chars = set(text)
char2int = {ch:i for i,ch in enumerate(chars)}
int2char = dict(enumerate(chars))
text_ints = np.array([char2int[ch] for ch in text],
                     dtype=np.int32)
```

Reshape the Data into Batches of Sequences (1/3)

Text Corpus

Convert text into a long sequence of integers

49, 29, 29, 29, 5, 19, 27, 0, 7, 3, 36, 65, 27, 41, 31, 0, 26, 4, 31, 27, 86, 10, 27, 3, 84, 67, 12, 0, 80, 31, 27, 58, 31, 0, 36, 28, 0, 75, 19, 22, . . . , 52, 84, 19, 31, 0, 22

Create sequences x and y

Sequence x:

49, 29, 29, 29, 5, 19, 27, 0, 7, 3, 36, 65, 27, 41, 31, 0, 26, 4, 31, 27, 86, 10, 27, 3, 84, 67, 12, 0, 80, 31, 27, 58, 31, 0, 36, 28, 0, 75, 19, 22, . . . , 52, 84, 19, 31, 0, 22

Sequence y:

49, 29, 29, 29, 5, 19, 27, 0, 7, 3, 36, 65, 27, 41, 31, 0, 26, 4, 31, 27, 86, 10, 27, 3, 84, 67, 12, 0, 80, 31, 27, 58, 31, 0, 36, 28, 0, 75, 19, 22, . . . , 52, 84, 19, 31, 0, 22

Training data array x:

49, 29, 29, 29, 5, 19, . . . , 41, 31
73, 11, 56, 0, 36, 28, . . . , 72, 45
19, 22, 31, 67, 12, 0, . . . , 12, 0
. . .
22, 51, 51, 0, 51, 52, . . . , 86, 42

Batch size

Training data array y:

29, 29, 29, 5, 19, 27, . . . , 31, 0
11, 56, 0, 36, 28, 0, . . . , 45, 4
22, 31, 67, 12, 0, 48, . . . , 0, 86
. . .
51, 51, 0, 51, 52, 4, . . . , 42, 0

Batch size

Number of batches × Number of steps

Reshape the Data into Batches of Sequences (2/3)

```python
def reshape_data(sequence, batch_size, num_steps):
    tot_batch_length = batch_size * num_steps
    num_batches = int(len(sequence) / tot_batch_length)
    if num_batches*tot_batch_length + 1 > len(sequence):
        num_batches = num_batches - 1
    ## Truncate the sequence at the end to get rid of
    ## remaining charcaters that do not make a full batch
    x = sequence[0 : num_batches*tot_batch_length]
    y = sequence[1 : num_batches*tot_batch_length + 1]
    ## Split x & y into a list batches of sequences:
    x_batch_splits = np.split(x, batch_size)
    y_batch_splits = np.split(y, batch_size)
    ## Stack the batches together
    ## batch_size x tot_batch_length
    x = np.stack(x_batch_splits)
    y = np.stack(y_batch_splits)

    return x, y
```

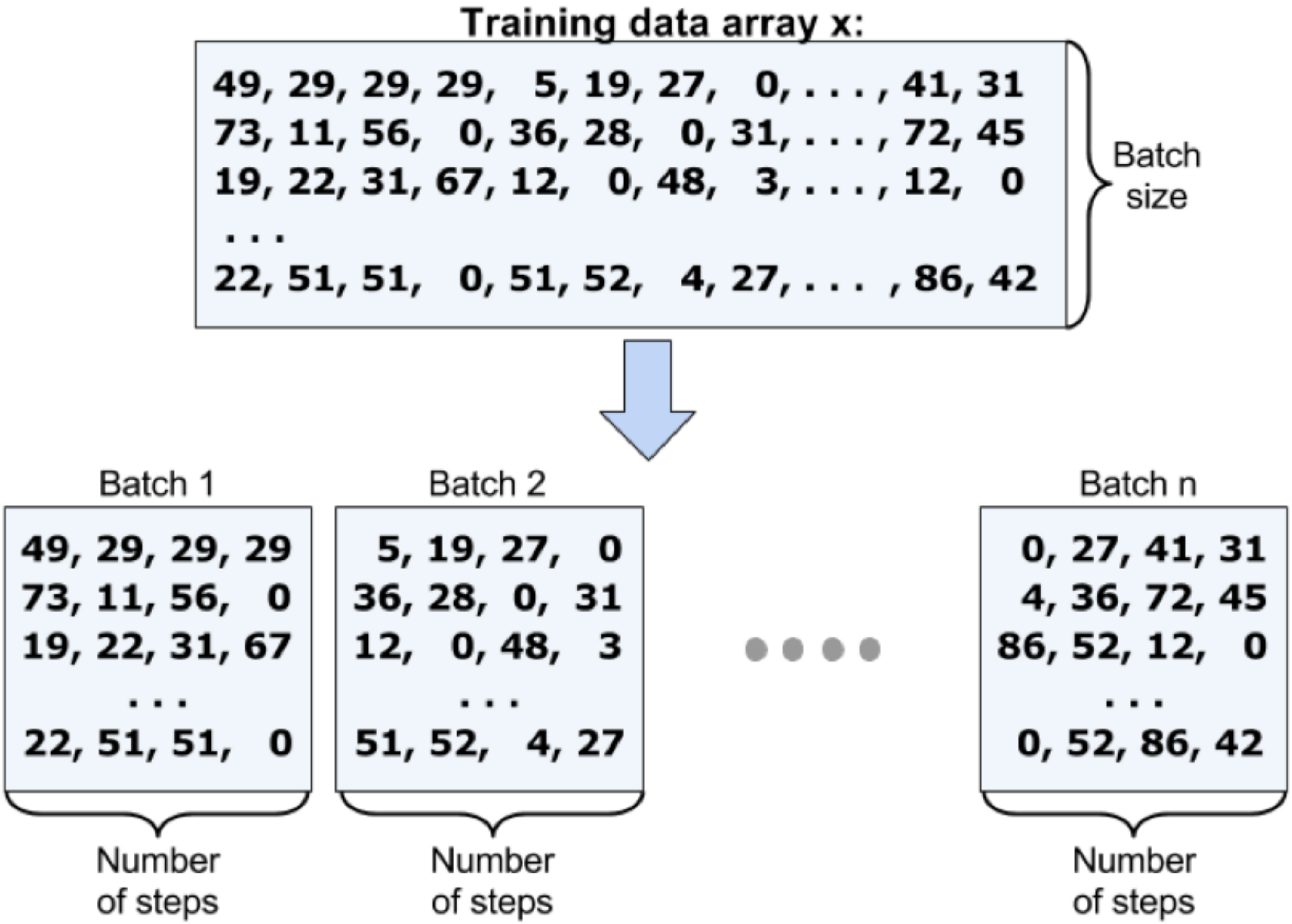
Reshape the Data into Batches of Sequences (3/3)

- Test

```
## Testing:
train_x, train_y = reshape_data(text_ints, 64, 10)
print(train_x.shape)
print(train_x[0, :10])
print(train_y[0, :10])
print(''.join(int2char[i] for i in train_x[0, :50]))
```

```
(64, 2540)
[ 8  5 41  2  8 39 19 57 41 55]
[ 5 41  2  8 39 19 57 41 55 47]
The Tragedie of Hamlet

Actus Primus. Scoena Prima
```

Split x and y into Mini-Batches (1/2)



Split x and y into Mini-Batches (2/2)

```
np.random.seed(123)

def create_batch_generator(data_x, data_y, num_steps):
    batch_size, tot_batch_length = data_x.shape
    num_batches = int(tot_batch_length/num_steps)
    for b in range(num_batches):
        yield (data_x[:, b*num_steps: (b+1)*num_steps],
               data_y[:, b*num_steps: (b+1)*num_steps])

bgen = create_batch_generator(train_x[:,:100], train_y[:,:100], 15)
for b in bgen:
    print(b[0].shape, b[1].shape, end='  ')
    print(''.join(int2char[i] for i in b[0][0,:]).replace('\n', '*'), '     ',
          ''.join(int2char[i] for i in b[1][0,:]).replace('\n', '*'))
```

```
(64, 15) (64, 15)  The Tragedie of      he Tragedie of
(64, 15) (64, 15)   Hamlet**Actus       Hamlet**Actus P
(64, 15) (64, 15)  Primus. Scoena       rimus. Scoena P
(64, 15) (64, 15)  Prima.**Enter B      rima.**Enter Ba
(64, 15) (64, 15)  arnardo and Fra      rnardo and Fran
(64, 15) (64, 15)  ncisco two Cent      cisco two Centi
```

Building a Character-Level RNN Model

- CharRNN to predict the next character
  - A constructor: To set the parameters, create the computation graph, call *build* method to build RNN
  - ***build***: Define the placeholders for feeding the data, construct RNN using LSTM cells, define the output of the network, cost function, optimizer
  - ***train***: To iterate through mini-batches and train the network for the specified number of epochs
  - ***sample***: To start from a given string, calculate the probabilities for the next character, and choose the character accordingly. This process will be repeated, and the sampled characters will be concatenated together to form a string. Once the size of this string reaches specified length, it will return the string

CharRNN: The constructor

```
import tensorflow as tf
import os

class CharRNN(object):
    def __init__(self, num_classes, batch_size=64,
                 num_steps=100, lstm_size=128,
                 num_layers=1, learning_rate=0.001,
                 keep_prob=0.5, grad_clip=5,
                 sampling=False):
        self.num_classes = num_classes
        self.batch_size = batch_size
        self.num_steps = num_steps
        self.lstm_size = lstm_size
        self.num_layers = num_layers
        self.learning_rate = learning_rate
        self.keep_prob = keep_prob
        self.grad_clip = grad_clip

        self.g = tf.Graph()
        with self.g.as_default():
            tf.set_random_seed(123)

            self.build(sampling=sampling)
            self.saver = tf.train.Saver()
            self.init_op = tf.global_variables_initializer()
```

CharRNN: build() (1/4)

```python
    def build(self, sampling):
        if sampling == True:
            batch_size, num_steps = 1, 1
        else:
            batch_size = self.batch_size
            num_steps = self.num_steps

        tf_x = tf.placeholder(tf.int32,
                              shape=[batch_size, num_steps],
                              name='tf_x')
        tf_y = tf.placeholder(tf.int32,
                              shape=[batch_size, num_steps],
                              name='tf_y')
        tf_keepprob = tf.placeholder(tf.float32,
                              name='tf_keepprob')

        # One-hot encoding:
        x_onehot = tf.one_hot(tf_x, depth=self.num_classes)
        y_onehot = tf.one_hot(tf_y, depth=self.num_classes)
```

$$\text{in sampling mode}: \begin{cases} batch\_size = 1 \\ num\_steps = 1 \end{cases}$$

$$\text{in training mode}: \begin{cases} batch\_size = self.batch\_size \\ num\_steps = self.num\_steps \end{cases}$$

CharRNN: build() (2/4)

```
### Build the multi-layer RNN cells
cells = tf.contrib.rnn.MultiRNNCell(
    [tf.contrib.rnn.DropoutWrapper(
        tf.contrib.rnn.BasicLSTMCell(self.lstm_size),
        output_keep_prob=tf_keepprob)
    for _ in range(self.num_layers)])

## Define the initial state
self.initial_state = cells.zero_state(
            batch_size, tf.float32)

## Run each sequence step through the RNN
lstm_outputs, self.final_state = tf.nn.dynamic_rnn(
            cells, x_onehot,
            initial_state=self.initial_state)

print('  << lstm_outputs  >>', lstm_outputs)

seq_output_reshaped = tf.reshape(
            lstm_outputs,
            shape=[-1, self.lstm_size],
            name='seq_output_reshaped')
```

CharRNN: build() (3/4)

```
        logits = tf.layers.dense(
                    inputs=seq_output_reshaped,
                    units=self.num_classes,
                    activation=None,
                    name='logits')

        proba = tf.nn.softmax(
                    logits,
                    name='probabilities')
        print(proba)

        y_reshaped = tf.reshape(
                    y_onehot,
                    shape=[-1, self.num_classes],
                    name='y_reshaped')
        cost = tf.reduce_mean(
                    tf.nn.softmax_cross_entropy_with_logits(
                        logits=logits,
                        labels=y_reshaped),
                    name='cost')
```

CharRNN: build() (4/4)

```
# Gradient clipping to avoid "exploding gradients"
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(
            tf.gradients(cost, tvars),
            self.grad_clip)
optimizer = tf.train.AdamOptimizer(self.learning_rate)
train_op = optimizer.apply_gradients(
            zip(grads, tvars),
            name='train_op')
```

CharRNN: train() (1/3)

```python
    def train(self, train_x, train_y,
              num_epochs, ckpt_dir='./model/'):
        ## Create the checkpoint directory
        ## if does not exists
        if not os.path.exists(ckpt_dir):
            os.mkdir(ckpt_dir)

        with tf.Session(graph=self.g) as sess:
            sess.run(self.init_op)

            n_batches = int(train_x.shape[1]/self.num_steps)
            iterations = n_batches * num_epochs
            for epoch in range(num_epochs):
```

CharRNN: train() (2/3)

```
# Train network
new_state = sess.run(self.initial_state)
loss = 0
## Minibatch generator:
bgen = create_batch_generator(
        train_x, train_y, self.num_steps)
for b, (batch_x, batch_y) in enumerate(bgen, 1):
    iteration = epoch*n_batches + b

    feed = {'tf_x:0': batch_x,
            'tf_y:0': batch_y,
            'tf_keepprob:0': self.keep_prob,
            self.initial_state : new_state}
    batch_cost, _, new_state = sess.run(
            ['cost:0', 'train_op',
                self.final_state],
            feed_dict=feed)
    if iteration % 10 == 0:
        print('Epoch %d/%d Iteration %d'
              '| Training loss: %.4f' % (
              epoch + 1, num_epochs,
              iteration, batch_cost))
```

CharRNN: train() (3/3)

```
## Save the trained model
self.saver.save(
        sess, os.path.join(
            ckpt_dir, 'language_modeling.ckpt'))
```

CharRNN: sample() (1/2)

```
def sample(self, output_length,
           ckpt_dir, starter_seq="The "):
    observed_seq = [ch for ch in starter_seq]
    with tf.Session(graph=self.g) as sess:
        self.saver.restore(
            sess,
            tf.train.latest_checkpoint(ckpt_dir))
        ## 1: run the model using the starter sequence
        new_state = sess.run(self.initial_state)
        for ch in starter_seq:
            x = np.zeros((1, 1))
            x[0,0] = char2int[ch]
            feed = {'tf_x:0': x,
                    'tf_keepprob:0': 1.0,
                    self.initial_state: new_state}
            proba, new_state = sess.run(
                    ['probabilities:0', self.final_state],
                    feed_dict=feed)

        ch_id = get_top_char(proba, len(chars))
        observed_seq.append(int2char[ch_id])
```

CharRNN: sample() (2/2)

```
        ## 2: run the model using the updated observed_seq
        for i in range(output_length):
            x[0,0] = ch_id
            feed = {'tf_x:0': x,
                    'tf_keepprob:0': 1.0,
                    self.initial_state: new_state}
            proba, new_state = sess.run(
                    ['probabilities:0', self.final_state],
                    feed_dict=feed)

            ch_id = get_top_char(proba, len(chars))
            observed_seq.append(int2char[ch_id])

    return ''.join(observed_seq)
```

get_top_char()

```
def get_top_char(probas, char_size, top_n=5):
    p = np.squeeze(probas)
    p[np.argsort(p)[:-top_n]] = 0.0
    p = p / np.sum(p)
    ch_id = np.random.choice(char_size, 1, p=p)[0]
    return ch_id
```

Creating and Training the CharRNN Model

```
batch_size = 64
num_steps = 100
train_x, train_y = reshape_data(text_ints,
                                batch_size,
                                num_steps)

rnn = CharRNN(num_classes=len(chars), batch_size=batch_size)
rnn.train(train_x, train_y,
          num_epochs=100,
          ckpt_dir='./model-100/')
```

```
    << lstm_outputs  >> Tensor("rnn/transpose:0", shape=(64, 100, 128), dtype=float32)
  Tensor("probabilities:0", shape=(6400, 65), dtype=float32)
  Epoch 1/100 Iteration 10| Training loss: 3.7960
  Epoch 1/100 Iteration 20| Training loss: 3.3718
  Epoch 2/100 Iteration 30| Training loss: 3.2945
  Epoch 2/100 Iteration 40| Training loss: 3.2526
  Epoch 2/100 Iteration 50| Training loss: 3.2370
  Epoch 3/100 Iteration 60| Training loss: 3.2187
  Epoch 3/100 Iteration 70| Training loss: 3.1814
  Epoch 4/100 Iteration 80| Training loss: 3.1635
  Epoch 4/100 Iteration 90| Training loss: 3.1449
  Epoch 4/100 Iteration 100| Training loss: 3.1177
```

CharRNN Model in Sampling Mode

```
np.random.seed(123)
rnn = CharRNN(len(chars), sampling=True)

print(rnn.sample(ckpt_dir='./model-100/',
                 output_length=500))
```

```
The stall soues tay and the hates,
The perse in there is that so the meanes this made there

   Ham. Ile teath thes are this makere of a driane,
Why shis mestend the Casst of is singe,
In this to this, to mers it is for marth,
Ase hinees sim thig tald ow a tore andere,
In histhene tistere shere this wile and my Lord:
And tit mighes the secleer allost heruen, and that hash to sall and hears,
If you his moses tonger and mout ofr mesting a forte tis at

   Pomin. Where in you dist and sintere shan shall
```