AI School 6기 10주차

RNN 기초2

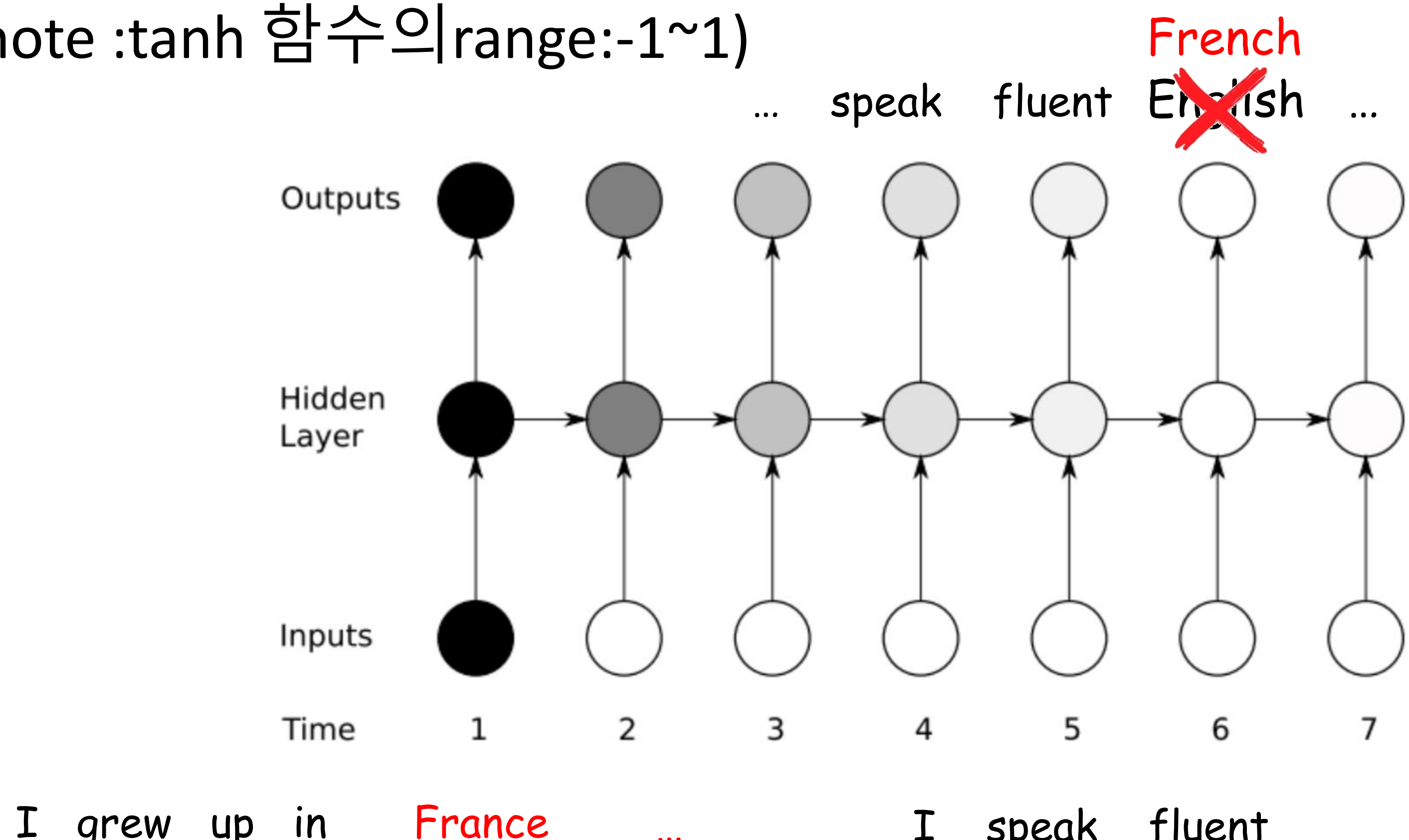
RNN 기반 텍스트 분류기

AI School 6기 10주차

RNN 기초 2

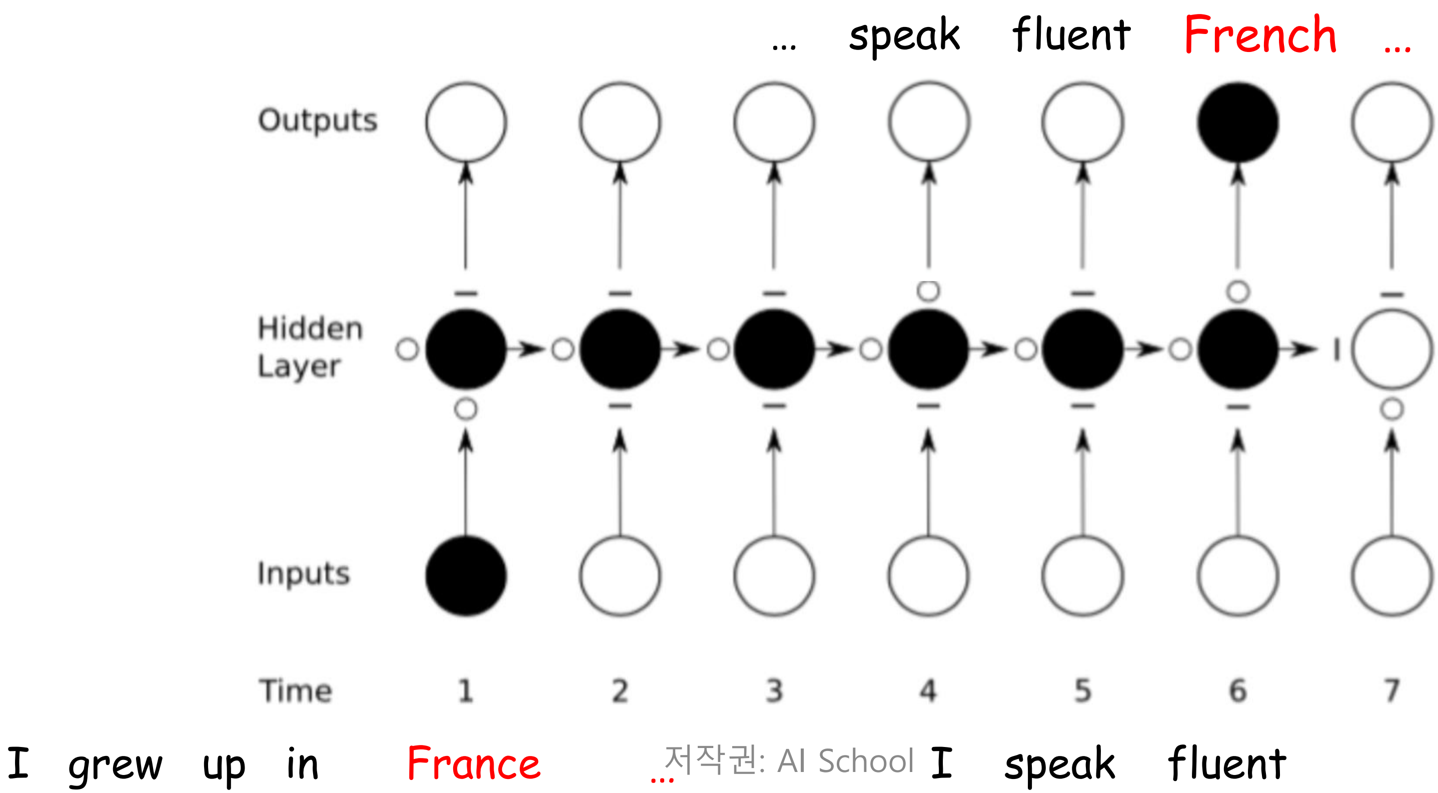
RNN의 한계

- 중요한 정보가 recurrent step이 계속됨에 따라 희석되는 문제 (long-term dependency)
- France라는 중요한 정보가 점차 희석됨
- (note :tanh 함수의range:-1~1)

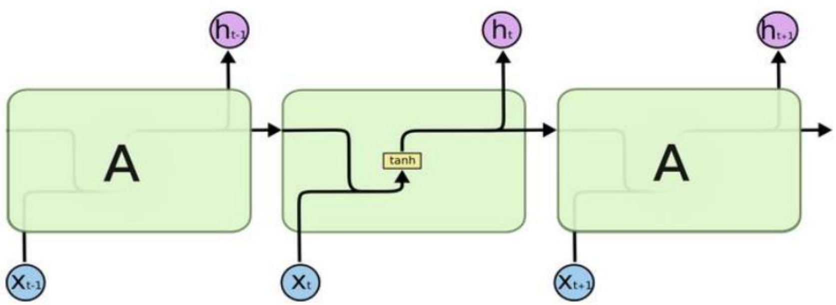


Long short-term memory (LSTM)

- 중요한 정보만 선택하여 이를 다음 state에 전달함으로써, long term dependency를 해결
- Cell에 정보를 저장하며, 정보들은 gate에 의해서 선택됨

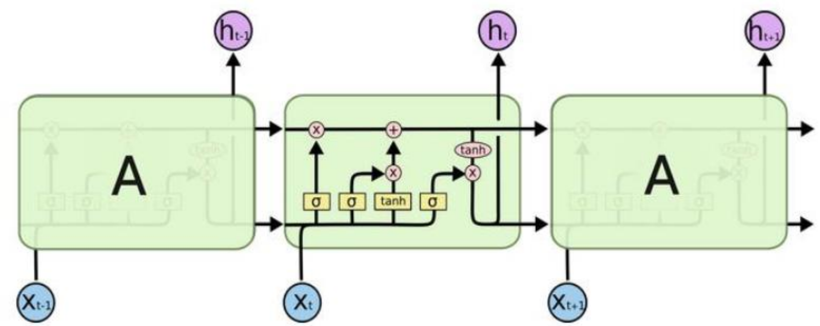


Long short-term memory (LSTM)



The repeating module in a standard RNN contains a single layer.

Basic RNN



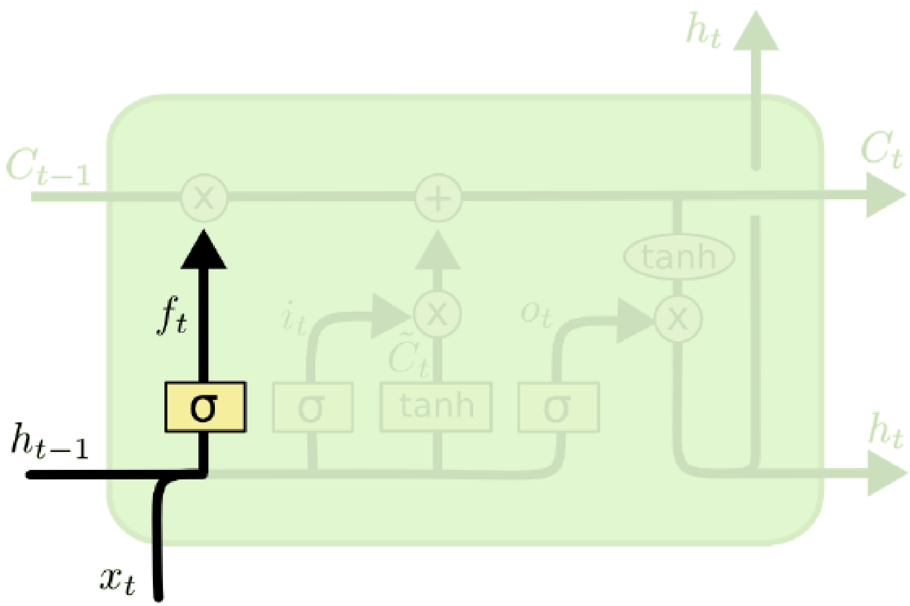
The repeating module in an LSTM contains four interacting layers.

LSTM

https://dgkim5360.tistory.com/entry/understanding-long-short-term-memory-lstm-kr

Long short-term memory (LSTM)

- forget gate : 중요하지 않은 과거의 정보를 삭제



Forget Gate

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

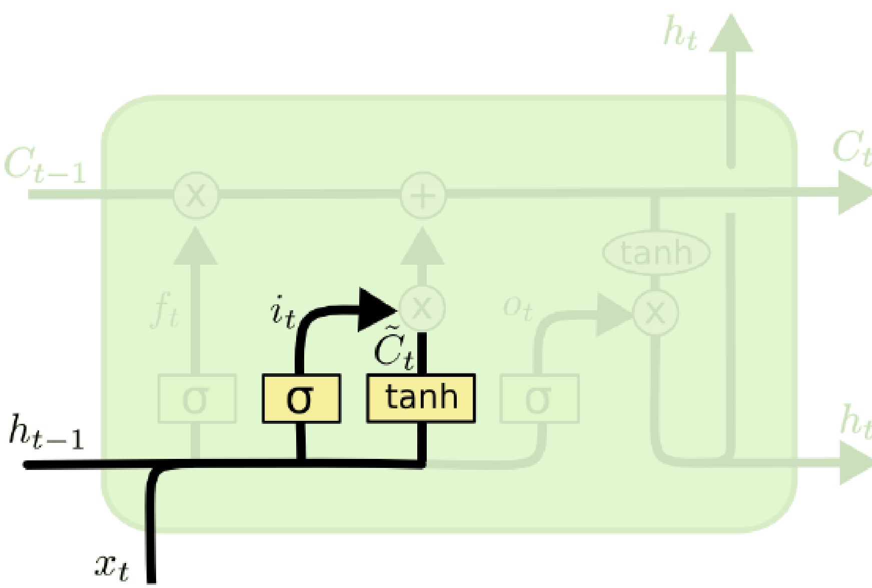
weight matrices for forget gate

$f_t \approx 1$: Register the previous state

$f_t \approx 0$: Forget the previous state

Long short-term memory (LSTM)

- input gate : 새로운 input에서 중요한 정보만 발췌
- candidate values : input을 과거의state의 정보를 고려하여 변형 (i.e. context에 맞게 word 변형)



Input Gates

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

weight matrices for input gate

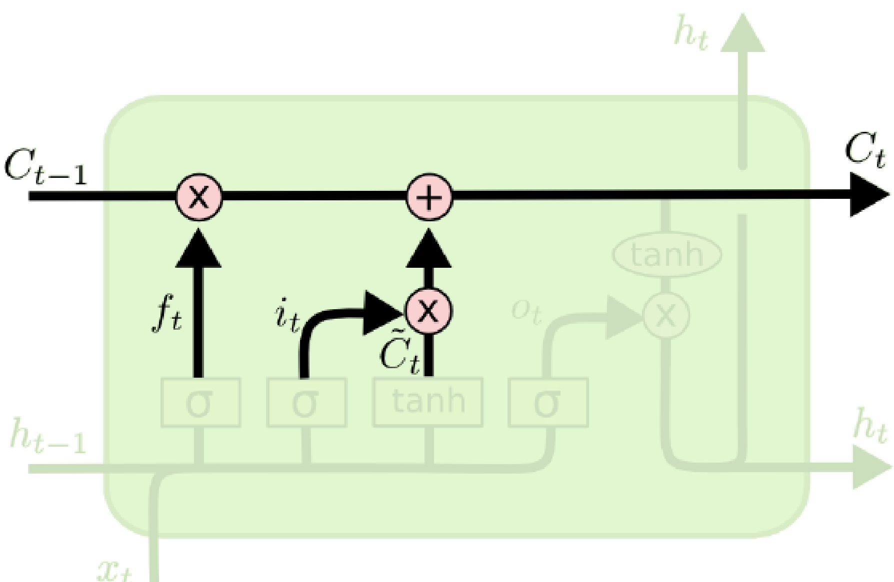
Candidate Values

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Generate values [-1, 1] to be added to cell state

Long short-term memory (LSTM)

- new cell state : forget gate와 input gate에 의한 정보 갱신



New Cell State

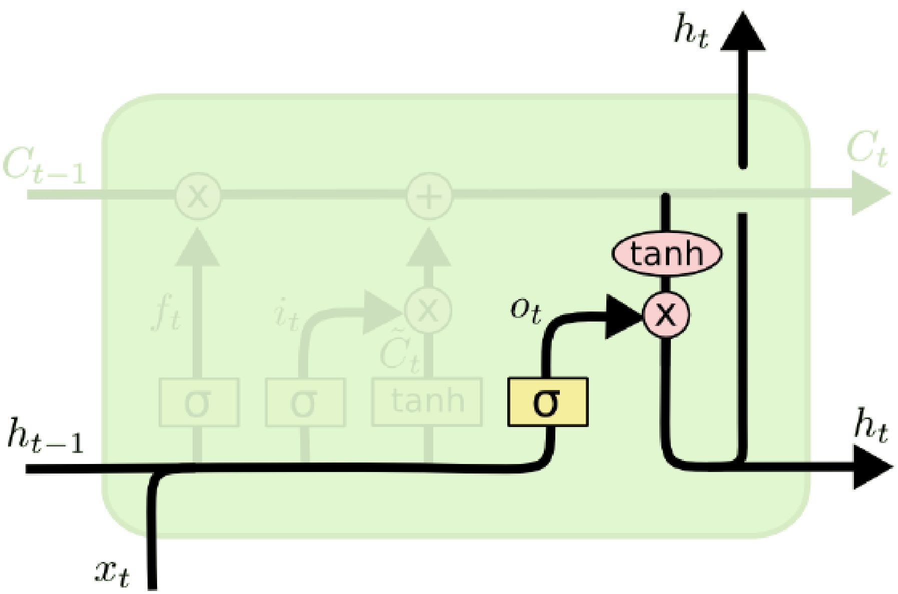
Candidate values

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

previous cell state

Long short-term memory (LSTM)

- output gate: cell state의 정보에서 추출해낼 정보의 양 선택
- hidden state : cell state와 output gate에 의해 결정



To decide which values of cell state to be used

$$o_t = \sigma(W_o\ [h_{t-1}, x_t]\ +\ b_o)$$

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

To scale the cell state within [-1, 1]

Long short-term memory (LSTM)

We have a sequence of inputs $\boldsymbol{x}^{(t)}$, and we will compute a sequence of hidden states $\boldsymbol{h}^{(t)}$ and cell states $\boldsymbol{c}^{(t)}$. On timestep *t*:

Sigmoid function: all gate values are between 0 and 1

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

$$\boldsymbol{f}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$

$$\boldsymbol{i}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$

$$\boldsymbol{o}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

Hidden state: read ("output") some content from the cell

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

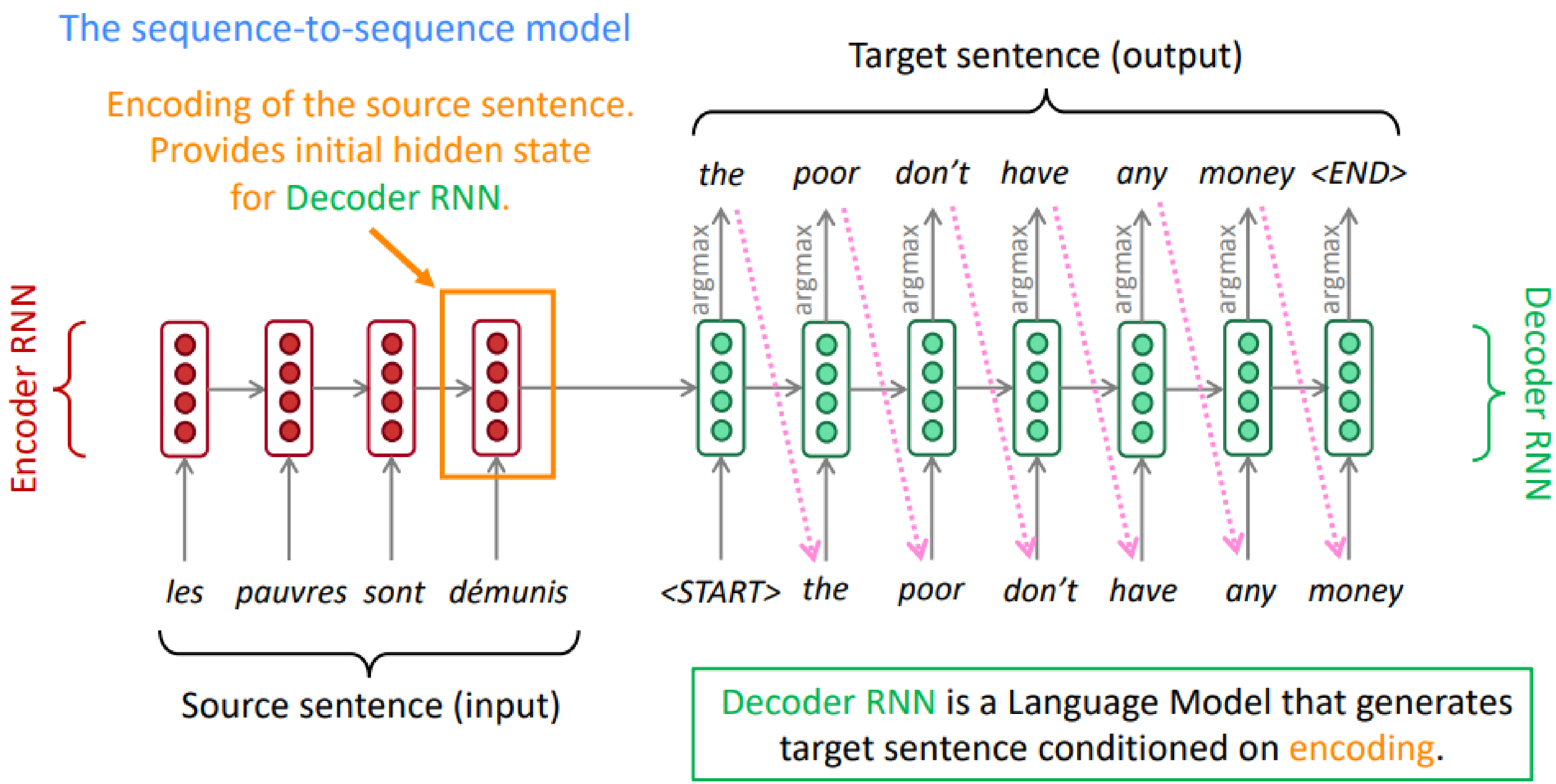
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

Gates are applied using element-wise product

All these are vectors of same length *n*

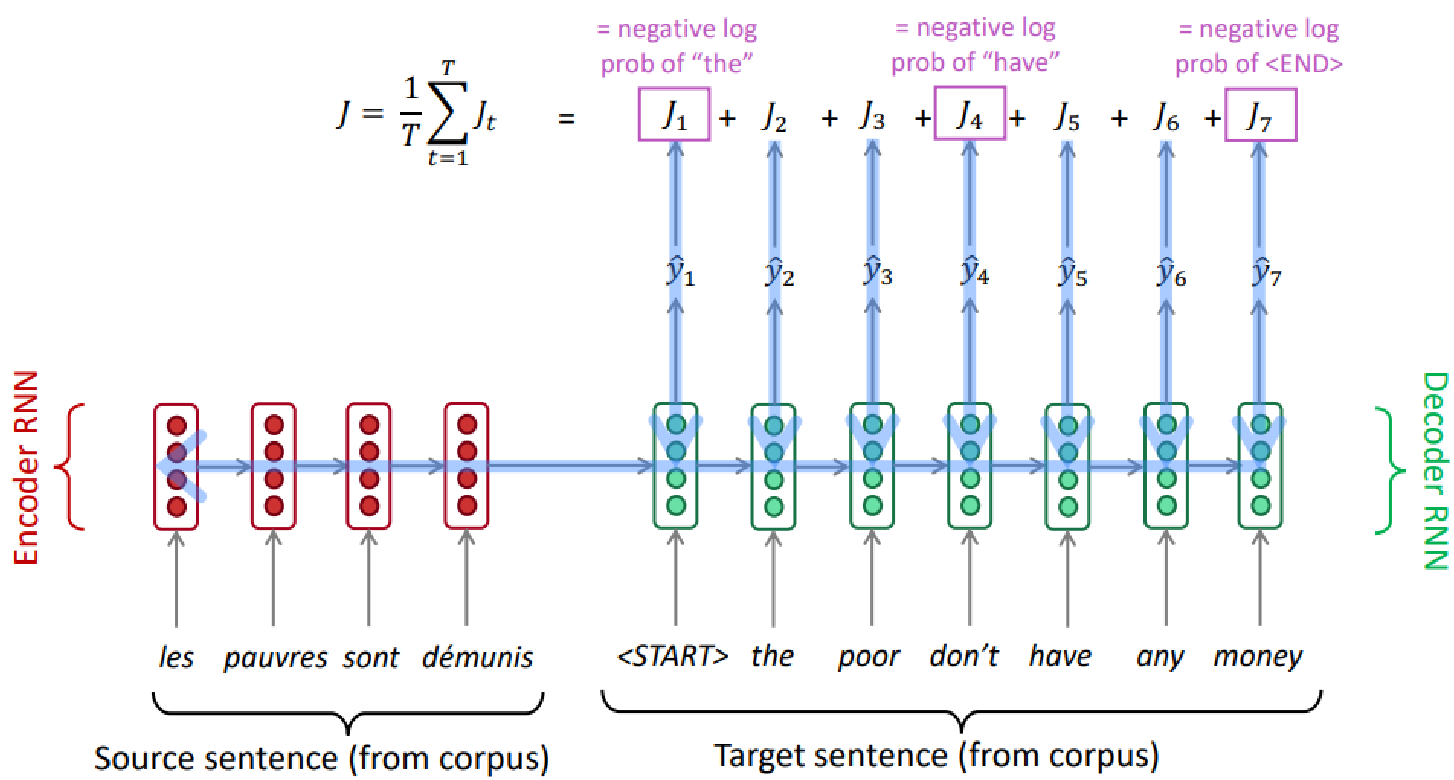
Basic neural machine translation (NMT)

- Source sentence(번역대상)를 RNN으로 encoding 후, 이를 바탕으로 target sentence(번역물) 생성

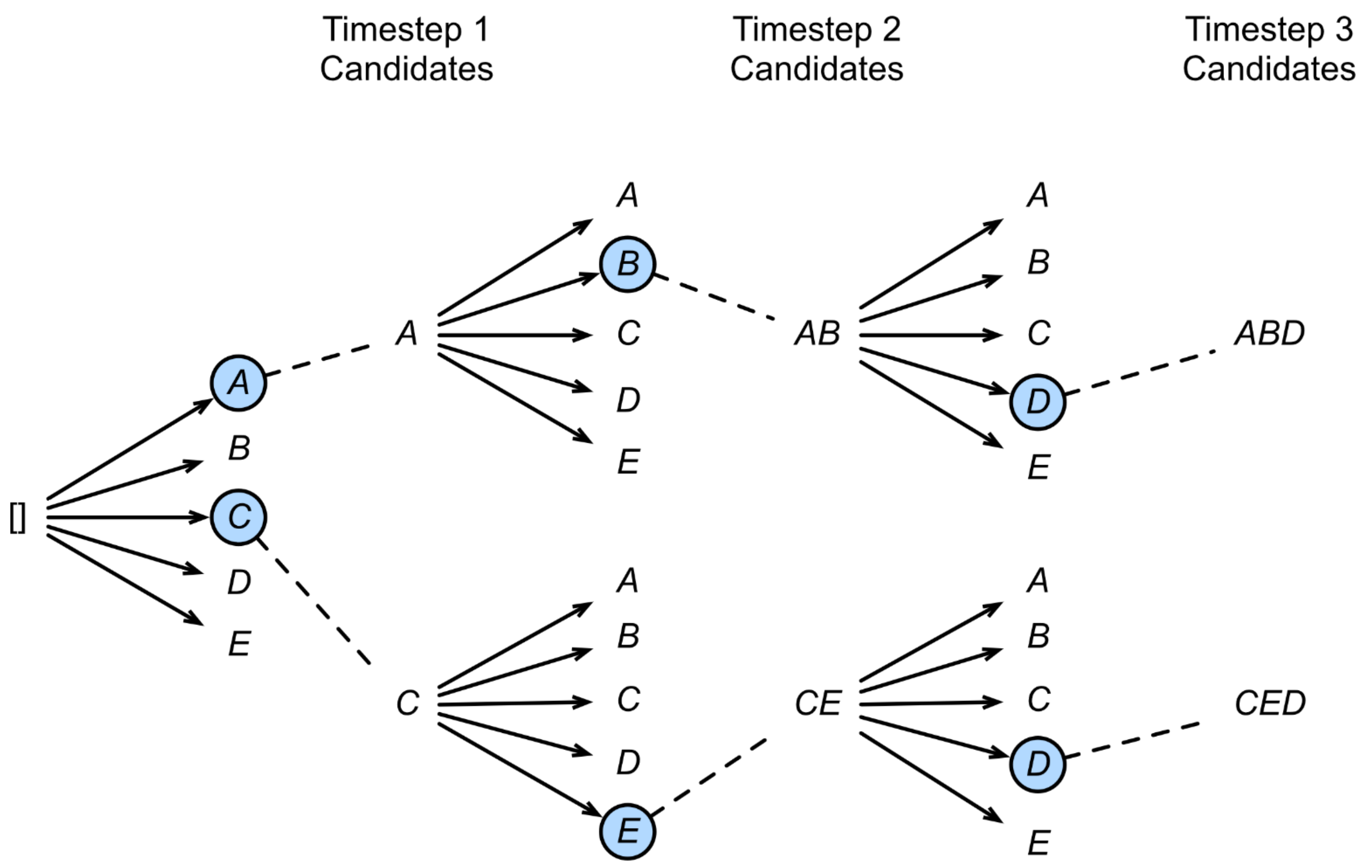


Basic NMT

- 기존의 방법론은, encoder의 마지막state를 통해task를 수행
- Attention mechanism은 마지막state가 아닌, 모든state들의 선형조합을 통해 new state를 생성

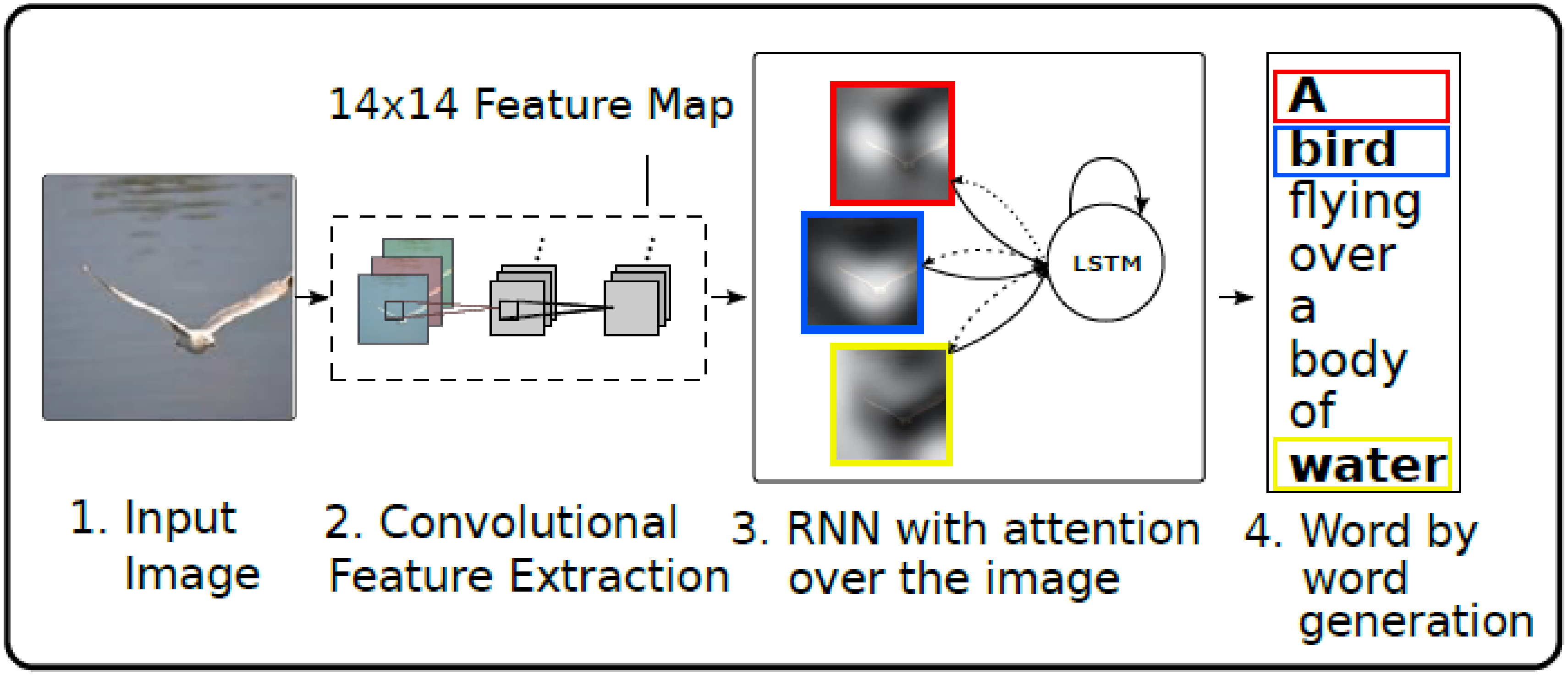


Beam Search



Timestep 1
Candidates
Timestep 2
Candidates
Timestep 3
Candidates
[]
A
B
C
D
E
A
C
A
B
C
D
E
A
B
C
D
E
AB
CE
A
B
C
D
E
A
B
C
D
E
ABD
CED

Attention mechanism for image captioning

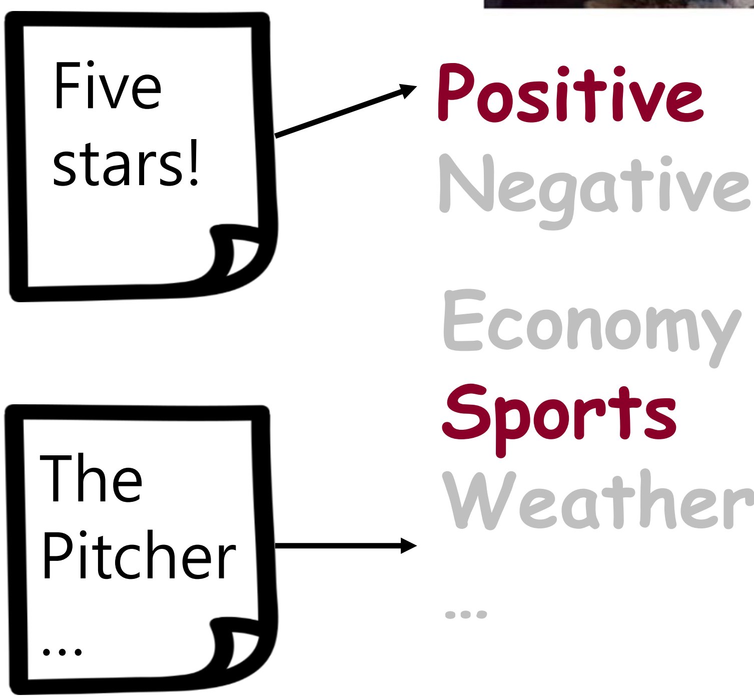


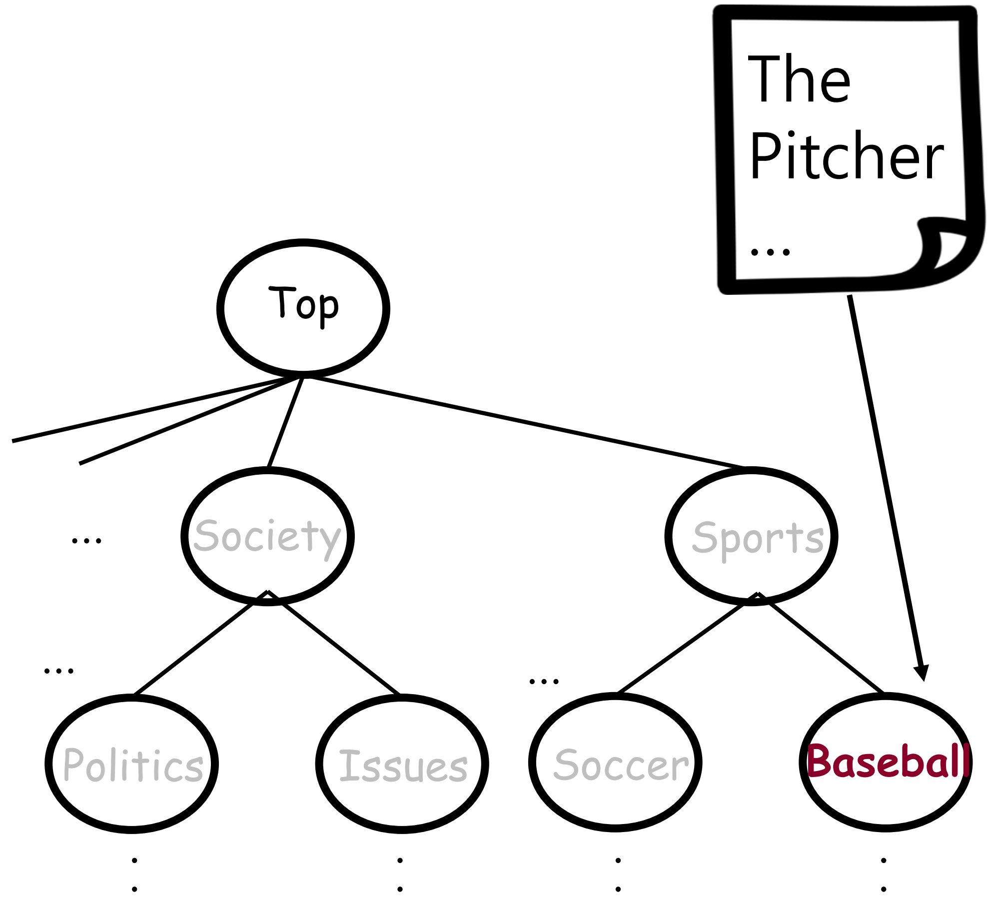
AI School 6기 10주차

CNN 기반 텍스트 분류기

CNN for text classification

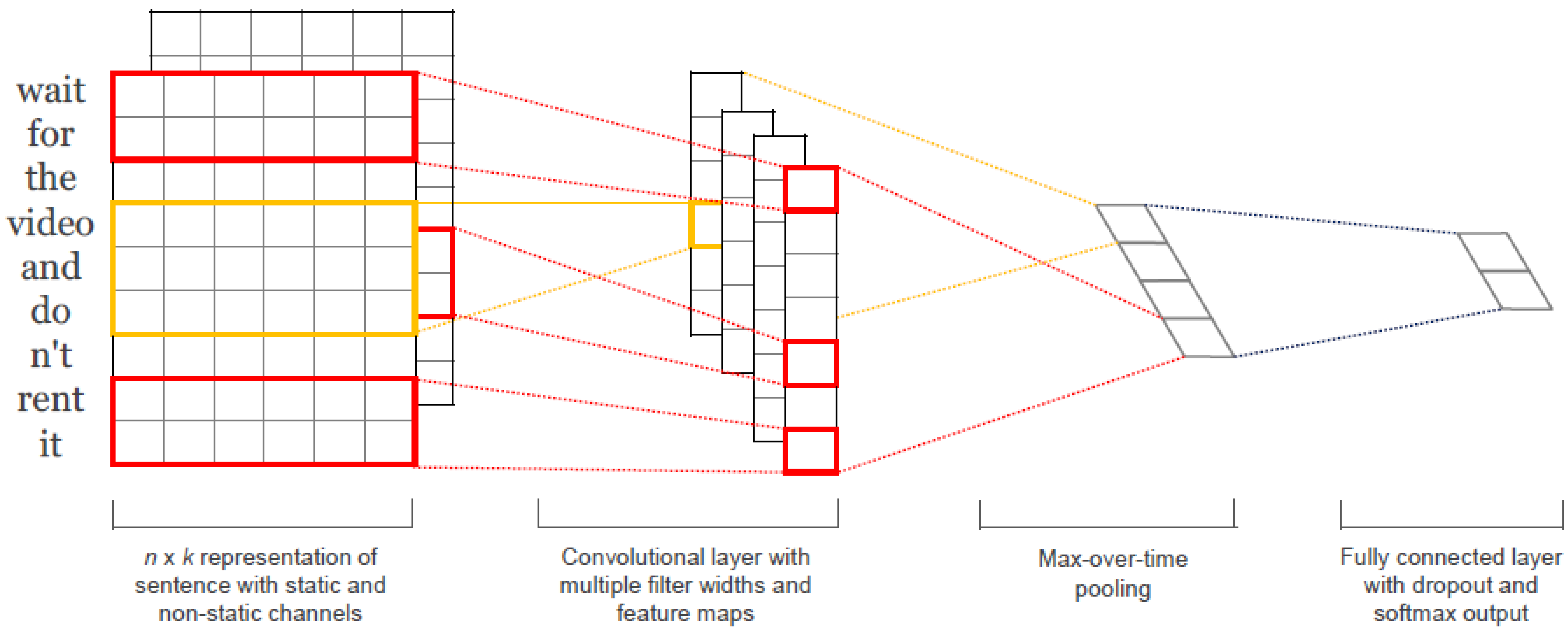
- CNN for sentence classification [Kim et al., 2014]





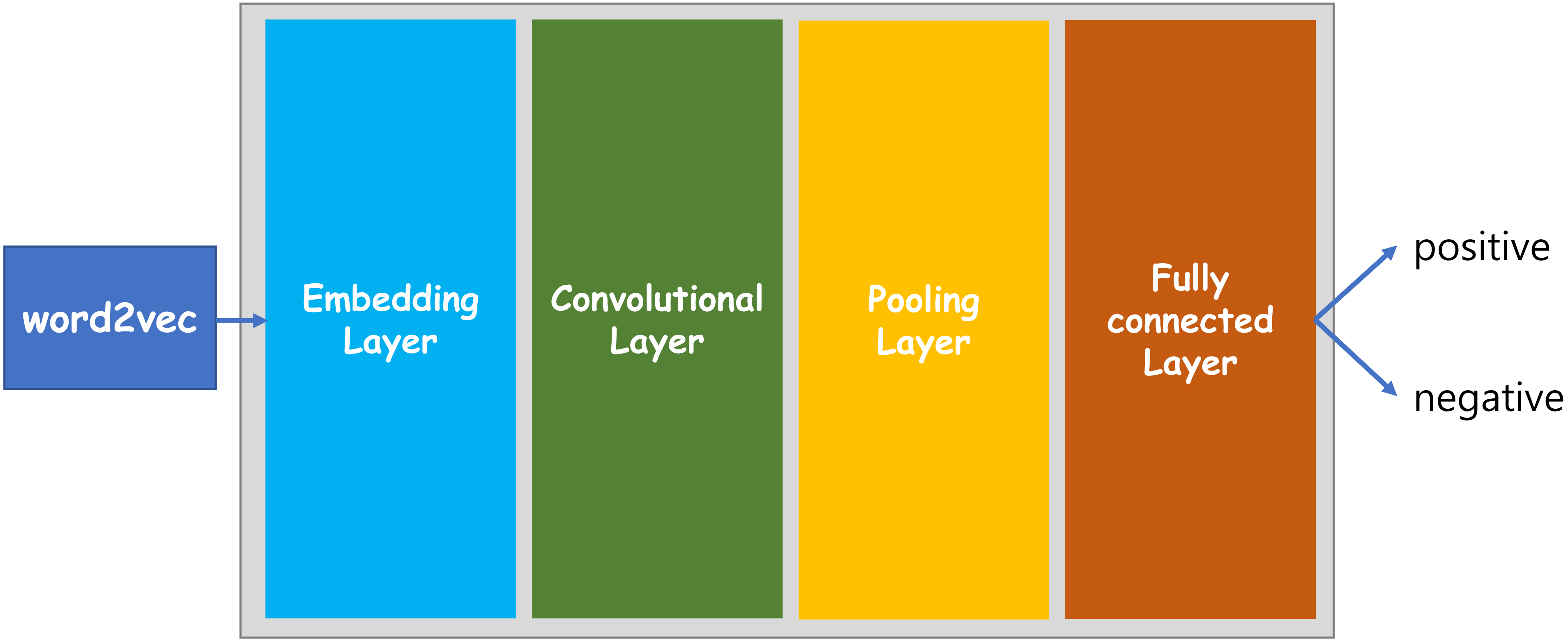
CNN for text classification

- Model overview



CNN for text classification

- Model overview



CNN for text classification

- Word2vec & Embedding layer

sentence

Five star! This movie... ➡

word list

["five", "star", "this", "movie"]

⬇

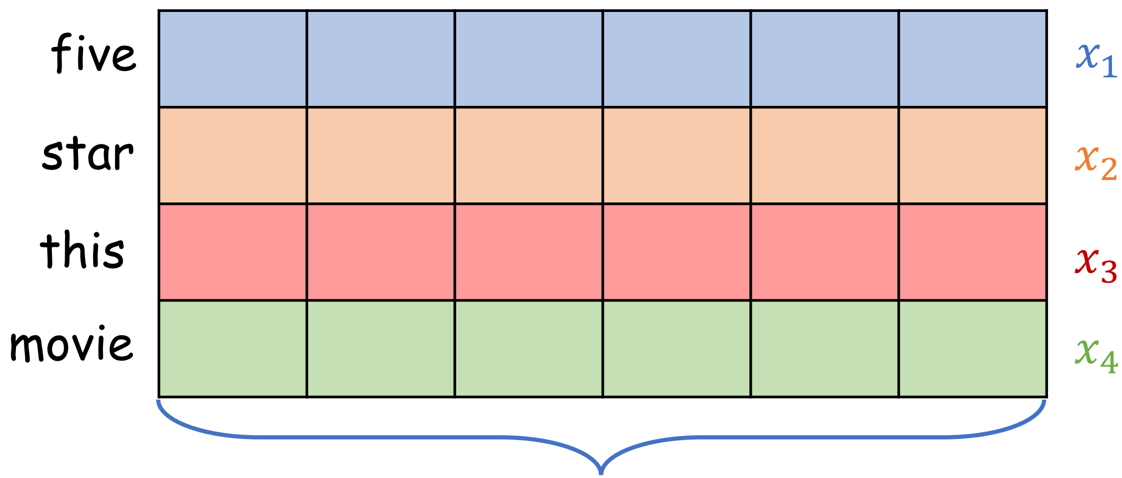
word vector list (word2vec)

$[x_1, x_2, x_3, x_4]$ ➡

sentence representation

$x_{1:4} = x_1 \oplus x_2 \oplus x_3 \oplus x_4$

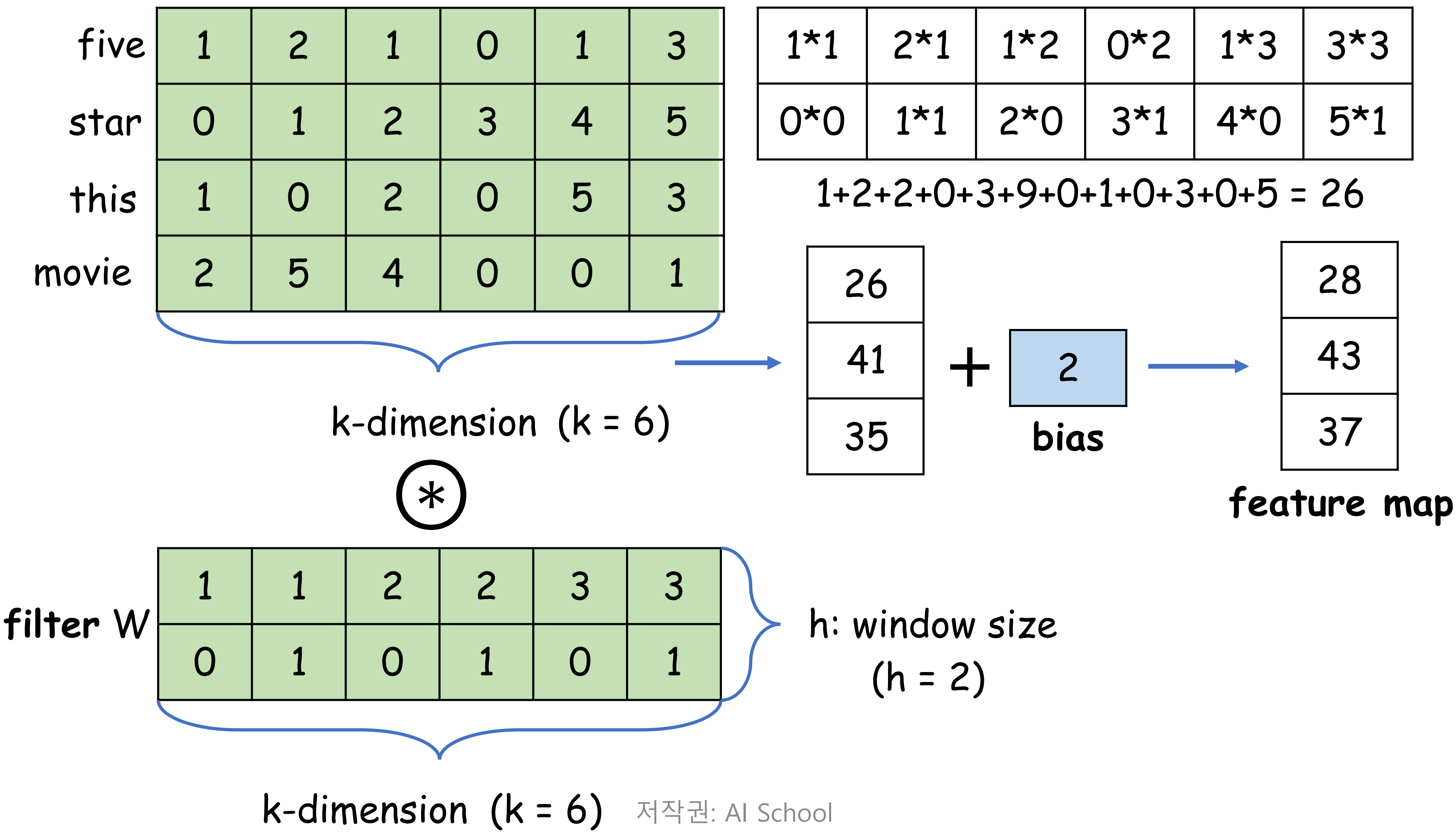
$\oplus$: concatenation operator



K-dimension (K = 6)

CNN for text classification

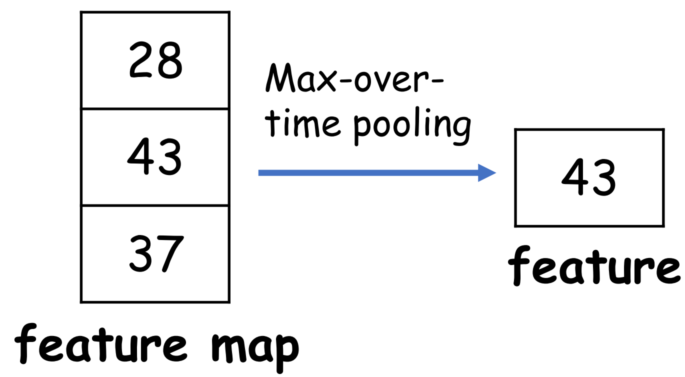
- Convolutional layer



CNN for text classification

- Pooling layer

five	1	2	1	0	1	3
star	0	1	2	3	4	5
this	1	0	2	0	5	3
movie	2	5	4	0	0	1

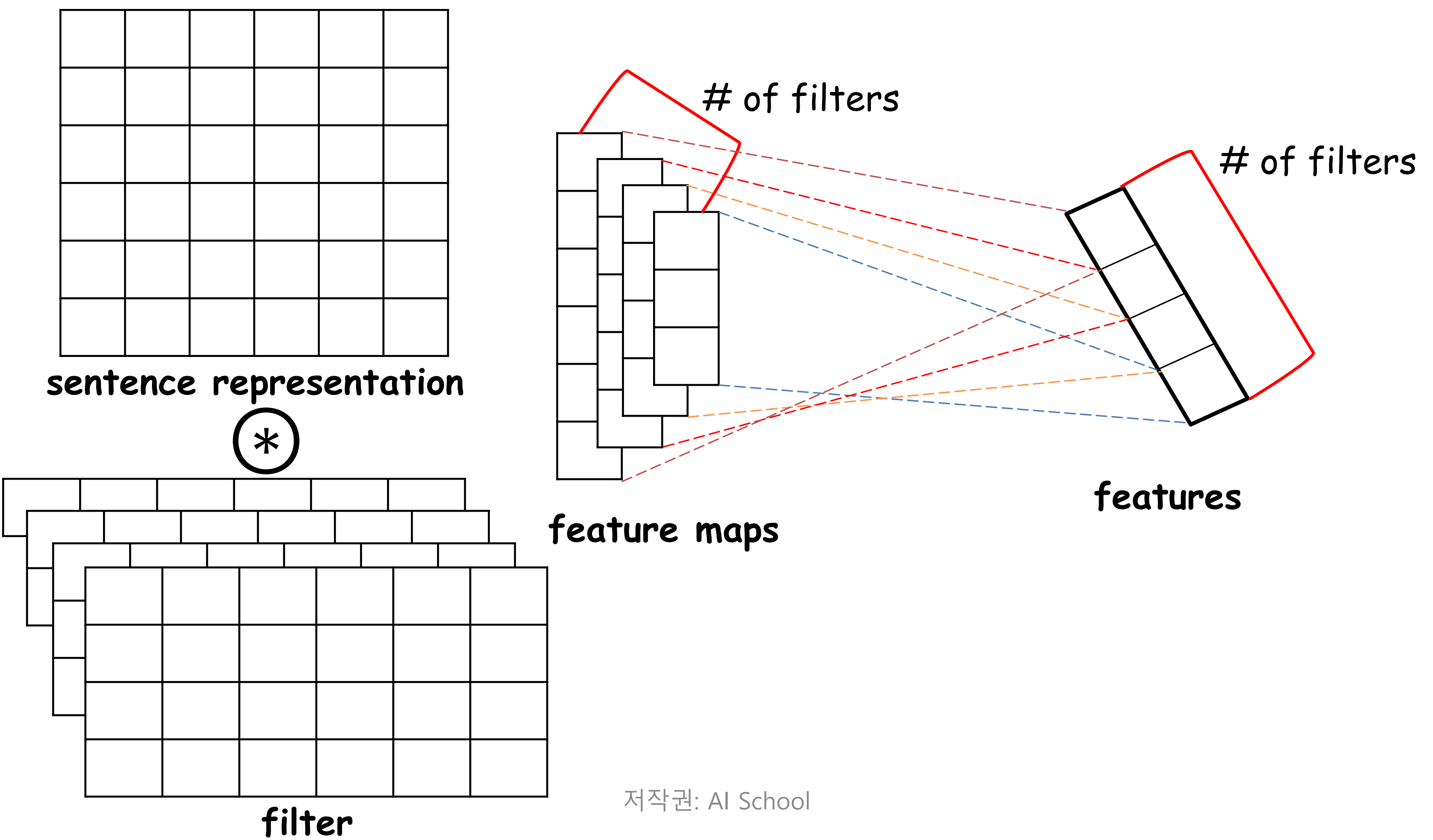


filter W	1	1	2	2	3	3
	0	1	0	1	0	1

One filter -> **One** feature
Multiple filter -> **Multiple** feature

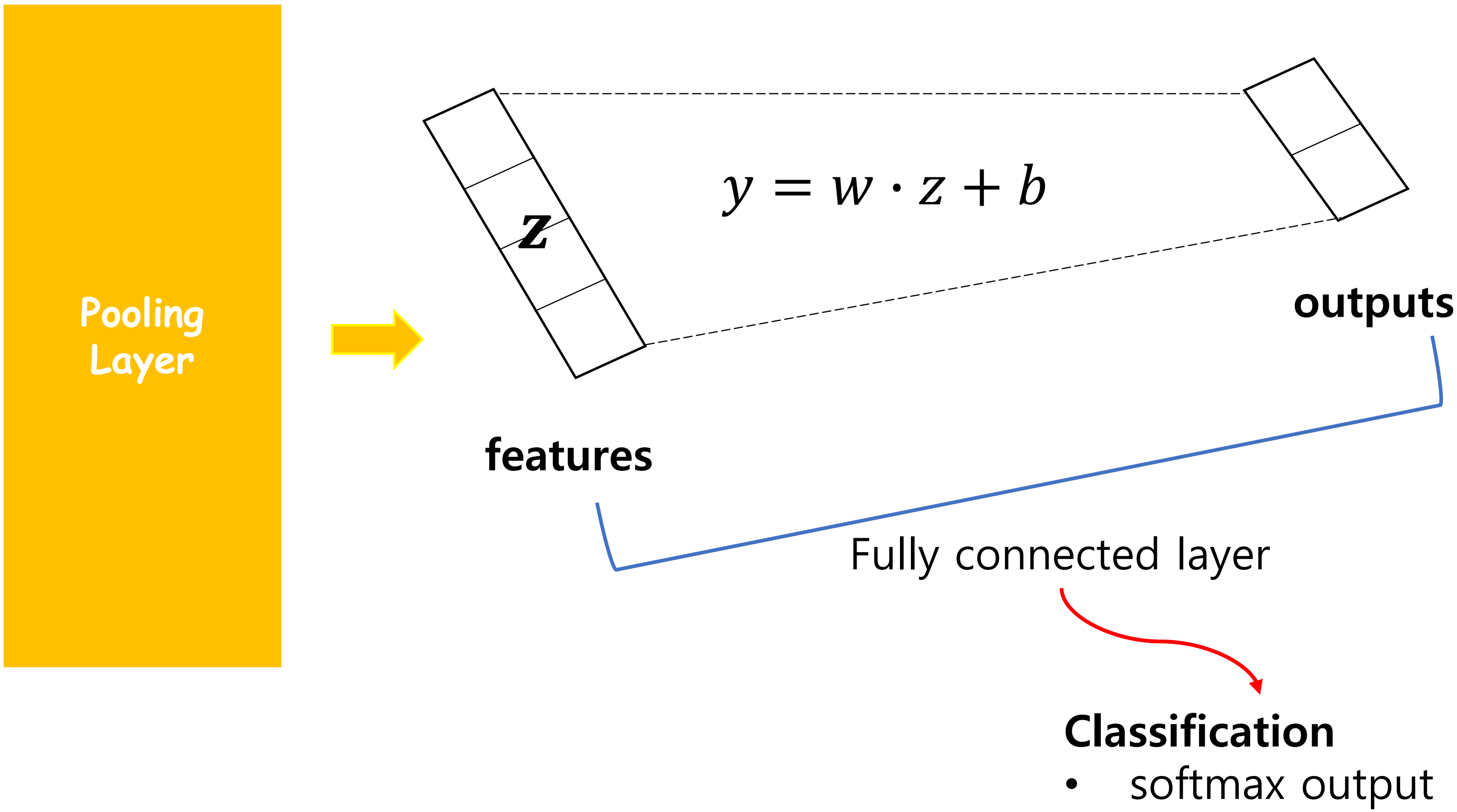
CNN for text classification

- Pooling layer



CNN for text classification

- Fully connected layer

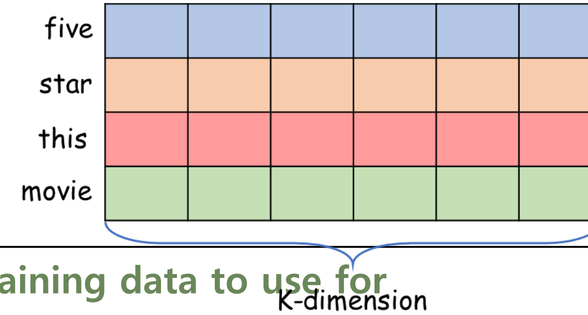


CNN 분류기 학습

- Import

```
import tensorflow as tf
import numpy as np
import os
import time
import datetime
import re
import smart_open
import pickle
import text_classification_master.data_helpers as dh
from text_classification_master.text_cnn import TextCNN
from gensim.models.keyedvectors import KeyedVectors
```

Hyperparameters



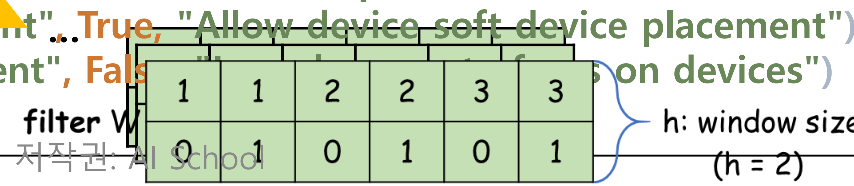
- train.py

```
tf.flags.DEFINE_float("dev_sample_percentage", .1, "Percentage of the training data to use for validation")
tf.flags.DEFINE_string("x_train_file", "./data/train/x_TrecTrain.txt", "Data source for the training")
tf.flags.DEFINE_string("t_train_file", "./data/train/t_TrecTrain.txt", "Data source for the training")
tf.flags.DEFINE_string("word2vec", "./data/GoogleNews-vectors-negative300.bin", "Word2vec file")
tf.flags.DEFINE_integer("vocab_size", 30000, "Vocabulary size (defualt: 0)")
tf.flags.DEFINE_integer("num_classes", 0, "The number of labels (defualt: 0)")
tf.flags.DEFINE_integer("max_length", 0, "max sequence length (defualt: 0)")
tf.flags.DEFINE_integer("embedding_dim", 300, "Dimensionality of character embedding (default: 128)")
tf.flags.DEFINE_string("filter_sizes", "3,4,5", "Comma-separated filter sizes (default: '3,4,5')")
tf.flags.DEFINE_integer("num_filters", 100, "Number of filters per filter size (default: 128)")
tf.flags.DEFINE_float("dropout_keep_prob", 0.5, "Dropout keep probability (default: 0.5)")
tf.flags.DEFINE_float("l2_reg_lambda", 0.001, "L2 regularization lambda (default: 0.0)")
tf.flags.DEFINE_float("lr_decay", 0.9, "Learning rate decay rate (default: 0.98)")
tf.flags.DEFINE_float("lr", 1e-3, "Learning rate(default: 0.01)")
tf.flags.DEFINE_integer("batch_size", 50, "Batch Size (default: 64)")
tf.flags.DEFINE_integer("num_epochs", 200, "Number of training epochs (default: 200)")
tf.flags.DEFINE_integer("evaluate_every", 100, "Evaluate model on dev set after this many steps (default: 100)")
tf.flags.DEFINE_integer("checkpoint_every", 100, "Save model after this many steps (default: 100)")
tf.flags.DEFINE_integer("num_checkpoints", 3, "Number of checkpoints to store (default: 5)")
tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow device soft device placement")
tf.flags.DEFINE_boolean("log_device_placement", False, "Log placement of ops on devices")

FLAGS = tf.flags.FLAGS
```

단어를 표현하는 벡터의 크기

Filter의 높이
or
Window size
or
N-gram

Filter 종류별 개수



Data loading & preprocessing

- train.py

```
print("Loading data...")
x_text, y, _ = dh.load_data(FLAGS.x_train_file, FLAGS.t_train_file)

# Build vocabulary
word_id_dict, _ = dh.buildVocab(x_text, FLAGS.vocab_size)
print(word_id_dict)
FLAGS.vocab_size = len(word_id_dict) + 4
print("vocabulary size: ", FLAGS.vocab_size)

for word_id in word_id_dict.keys():
    word_id_dict[word_id] += 4  # 0: <pad>, 1: <unk>, 2: <s>, 3: </s>
word_id_dict['<pad>'] = 0
word_id_dict['<unk>'] = 1
word_id_dict['<s>'] = 2
word_id_dict['</s>'] = 3
```

data loading

- data_helpers.py

```
def load_data(x_file, t_file):
    # Load data from files

    t_large = []

    x_text = list(open(x_file, "r", encoding='UTF8').readlines())
    x_text = [s.strip() for s in x_text]
    x_text = np.array([clean_str(sent) for sent in x_text])

    lengths = np.array(list(map(len, [sent.split(" ") for sent in x_text])))

    t_text_temp = np.array(list(open(t_file, "r", encoding='UTF8').readlines()))

    maxLabel = t_text_temp.astype(np.int)
    print(maxLabel)
    maxLabel = np.max(maxLabel) + 1
    print("max label: "+str(maxLabel))
    for i, s in enumerate(t_text_temp):
        t = np.zeros(maxLabel)
        t[int(s)] = 1.0
        t_large.append(t)

    t_large = np.array(t_large)

    return [x_text, t_large, lengths]
```

Build Vocabulary

- train.py

```python
def buildVocab(sentences, vocab_size):
    # Build vocabulary
    words = []
    for sentence in sentences: words.extend(sentence.split())
    print("The number of words: ", len(words))
    word_counts = collections.Counter(words)
    # Mapping from index to word
    vocabulary_inv = [x[0] for x in word_counts.most_common(vocab_size)]
    # Mapping from word to index
    vocabulary = {x: i for i, x in enumerate(vocabulary_inv)}
    return [vocabulary, vocabulary_inv]
```

Text to indices, indices to tensor

- train.py

```
x = dh.text_to_index(x_text, word_id_dict, max(list(map(int, FLAGS.filter_sizes.split(",")))) - 1)
x, FLAGS.max_length = dh.train_tensor(x)

# Randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
# TODO: This is very crude, should use cross-validation
dev_sample_index = -1 * int(FLAGS.dev_sample_percentage * float(len(y)))
x_train, x_dev = x_shuffled[:dev_sample_index], x_shuffled[dev_sample_index:]
y_train, y_dev = y_shuffled[:dev_sample_index], y_shuffled[dev_sample_index:]

FLAGS.num_classes = y_train.shape[1]

del x, x_text, y, x_shuffled, y_shuffled
print(x_train)
print(y_train)

print("Train/Dev split: {:d}/{:d}".format(len(y_train), len(y_dev)))
return x_train, y_train, word_id_dict, x_dev, y_dev
```

Text to indices, indices to tensor

- train.py

```
x = dh.text_to_index(x_text, word_id_dict, max(list(map(int, FLAGS.filter_sizes.split(",")))) - 1)
x, FLAGS.max_length = dh.train_tensor(x)

# Randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
# TODO: This is very crude, should use cross-validation
dev_sample_index = -1 * int(FLAGS.dev_sample_percentage * float(len(y)))
x_train, x_dev = x_shuffled[:dev_sample_index], x_shuffled[dev_sample_index:]
y_train, y_dev = y_shuffled[:dev_sample_index], y_shuffled[dev_sample_index:]

FLAGS.num_classes = y_train.shape[1]

del x, x_text, y, x_shuffled, y_shuffled
print(x_train)
print(y_train)

print("Train/Dev split: {:d}/{:d}".format(len(y_train), len(y_dev)))
return x_train, y_train, word_id_dict, x_dev, y_dev
```

Text to indices, indices to tensor

- data_helpers.py

```
def text_to_index(text_list, word_to_id, nb_pad):
    text_indices = []
    for text in text_list:
        words = text.split(" ")
        pad = [0 for _ in range(nb_pad) ]
        ids = []
        for word in words:
            if word in word_to_id:
                word_id = word_to_id[word]
            else:
                word_id = 1
            ids.append(word_id)
        ids = pad + ids
        text_indices.append(ids)
    return text_indices

def train_tensor(batches):
    max_length = max([len(batch) for batch in batches])
    tensor = np.zeros((len(batches), max_length), dtype=np.int64)
    for i, indices in enumerate(batches):
        tensor[i, :len(indices)] = np.asarray(indices, dtype=np.int64)

    return tensor, max_length
```

TextCNN class & input

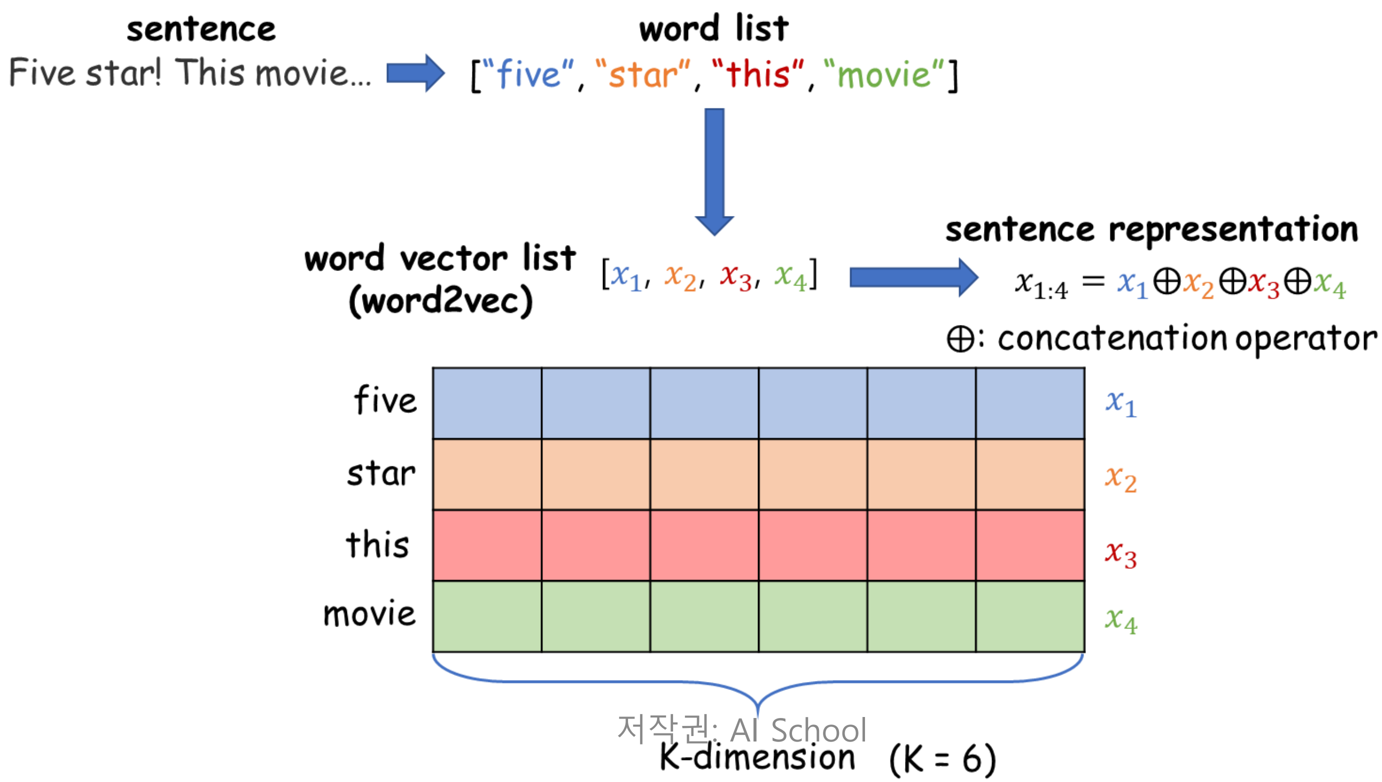
- text_cnn.py

```
import tensorflow as tf
import numpy as np


class TextCNN(object):

    def __init__(self, config):
        self.num_classes = config["num_classes"]
        self.vocab_size = config["vocab_size"]
        self.embedding_size = config["embedding_dim"]
        self.filter_sizes = list(map(int, config["filter_sizes"].split(",")))
        self.num_filters = config["num_filters"]
        self.l2_reg_lambda = config["l2_reg_lambda"]
        self.max_length = config["max_length"]

        # Placeholders for input, output and dropout
        self.input_x = tf.placeholder(tf.int32, [None, self.max_length], name="input_x")
        self.input_y = tf.placeholder(tf.float32, [None, self.num_classes], name="input_y")
        self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")
```

Embedding layer

- text_cnn.py

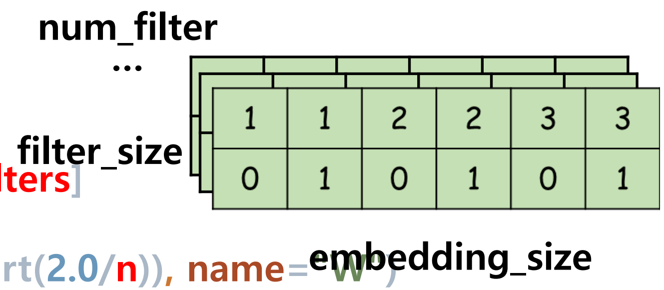
```
# Embedding layer
with tf.device('/gpu:0'), tf.name_scope("embedding"):
    self.W = tf.Variable(
        tf.random_uniform([self.vocab_size, self.embedding_size], -1.0, 1.0), trainable=True,
        name="W")
    self.embedded_chars = tf.nn.embedding_lookup(self.W, self.input_x)
    self.embedded_chars_expanded = tf.expand_dims(self.embedded_chars, -1)
```

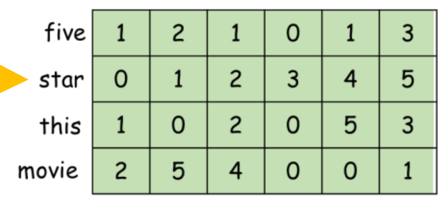


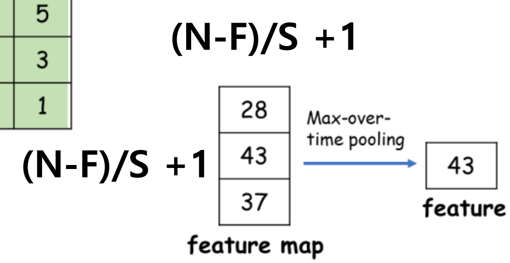
Convolutional layer

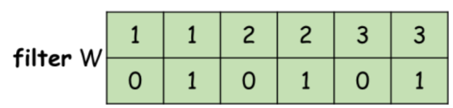
- text_cnn.py

```python
pooled_outputs = []
for i, filter_size in enumerate(self.filter_sizes):
    with tf.name_scope("conv-maxpool-%s" % filter_size):
        # Convolution Layer
        filter_shape = [filter_size, self.embedding_size, 1, self.num_filters]
        n = filter_size * self.embedding_size * self.num_filters
        W = tf.Variable(tf.random_normal(filter_shape, stddev=np.sqrt(2.0/n)), name="W")
        b = tf.Variable(tf.constant(0.1, shape=[self.num_filters]), name="b")
        conv = tf.nn.conv2d(
            self.embedded_chars_expanded,
            W,
            strides=[1, 1, 1, 1],
            padding="VALID",
            name="conv")
        # Apply nonlinearity
        h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
        # Maxpooling over the outputs
        pooled = tf.nn.max_pool(
            h,
            ksize=[1, self.max_length - filter_size + 1, 1, 1],
            strides=[1, 1, 1, 1],
            padding='VALID',
            name="pool")
        pooled_outputs.append(pooled)
```







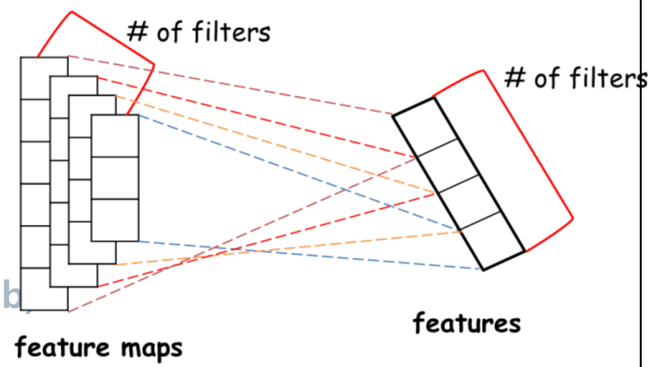


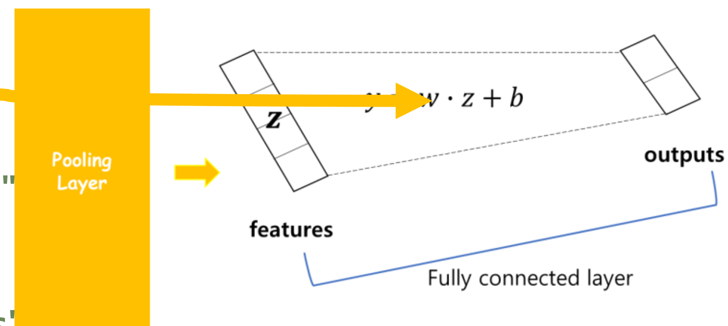
One filter -> **One** feature
Multiple filter -> **Multiple** feature

Fully connected layer

- text_cnn.py

```
# Combine all the pooled features
num_filters_total = num_filters * len(filter_sizes)
self.h_pool = tf.concat(pooled_outputs, 3)
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])

with tf.name_scope("dropout"):
    self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob

with tf.name_scope("output"):
    W = tf.get_variable(
        "W",
        shape=[num_filters_total, num_classes],
        initializer=tf.contrib.layers.xavier_initializer())
    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="
    l2_loss += tf.nn.l2_loss(W)
    l2_loss += tf.nn.l2_loss(b)
    self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
    self.predictions = tf.argmax(self.scores, 1, name="predictions")
```





Fully connected layer

- text_cnn.py

```
costs = []
for var in tf.trainable_variables():
    costs.append(tf.nn.l2_loss(var))
l2_loss = tf.add_n(costs)

# Calculate mean cross-entropy loss
with tf.name_scope("loss"):
    losses = tf.nn.softmax_cross_entropy_with_logits(logits=self.scores, labels=self.input_y)
    self.loss = tf.reduce_mean(losses) + self.l2_reg_lambda * l2_loss

# Accuracy
with tf.name_scope("accuracy"):
    correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
    self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```

Optimizer

- train.py

```
with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        cnn = TextCNN(FLAGS.flag_values_dict())

        # Define Training procedure
        global_step = tf.Variable(0, name="global_step", trainable=False)
        decayed_lr = tf.train.exponential_decay(FLAGS.lr, global_step, 1000, FLAGS.lr_decay,
staircase=True)
        optimizer = tf.train.AdamOptimizer(decayed_lr)
        grads_and_vars = optimizer.compute_gradients(cnn.loss)
        train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)
```

Save vocabulary and FLAGS

- train.py

```
# Write vocabulary
with smart_open.smart_open(os.path.join(out_dir, "vocab"), 'wb') as f:
    pickle.dump(word_id_dict, f)
with smart_open.smart_open(os.path.join(out_dir, "config"), 'wb') as f:
    pickle.dump(FLAGS.flag_values_dict(), f)
```

CNN 분류기 평가

- cnn_eval.py

```
tf.flags.DEFINE_string("x_test_file", "./data/test/x_Trec_test.txt", "Data source for the ODP training")
tf.flags.DEFINE_string("t_test_file", "./data/test/t_Trec_test.txt", "Data source for the ODP training")
# Eval Parameters
tf.flags.DEFINE_string("dir", "./runs/1585383108", "Checkpoint directory from training run")
# Misc Parameters
tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow device soft device placement")
tf.flags.DEFINE_boolean("log_device_placement", False, "Log placement of ops on devices")
FLAGS = tf.flags.FLAGS

x_raw, y_test, _ = dh.load_data(FLAGS.x_test_file, FLAGS.t_test_file)
y_test = np.argmax(y_test, axis=1)

# Map data into vocabulary
with smart_open.smart_open(os.path.join(FLAGS.dir, "vocab"), 'rb') as f:
    word_id_dict = pickle.load(f)
with smart_open.smart_open(os.path.join(FLAGS.dir, "config"), 'rb') as f:
    config = pickle.load(f)

    x_test = dh.text_to_index(x_raw, word_id_dict, max(list(map(int, config["filter_sizes"].split(",")))) - 1)
    x_test = dh.test_tensor(x_test, config["max_length"])

print("₩nEvaluating...₩n")
```

CNN 분류기 평가

- cnn_eval.py

```
checkpoint_file = tf.train.latest_checkpoint(os.path.join(FLAGS.dir, "checkpoints"))
graph = tf.Graph()
with graph.as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        cnn = TextCNN(config)
        sess.run(tf.global_variables_initializer())
        saver = tf.train.Saver(tf.global_variables())
        saver.restore(sess, checkpoint_file)

        # Generate batches for one epoch
        batches = dh.batch_iter(list(x_test), config["batch_size"], 1, shuffle=False)

        all_predictions = []
        for x_test_batch in batches:
            batch_predictions = sess.run(cnn.predictions, {cnn.input_x: x_test_batch,
cnn.dropout_keep_prob: 1.0})
            all_predictions = np.concatenate([all_predictions, batch_predictions])
```

AI School 6기 10주차

RNN 기반 텍스트 분류기

Hyperparameters

- train_rnn.py

```
tf.flags.DEFINE_float("val_sample_percentage", .1, "Percentage of the training data to use for validation")
tf.flags.DEFINE_string("x_train_file", "./data/train/x_agnewsTrain.txt", "Data source for the training")
tf.flags.DEFINE_string("t_train_file", "./data/train/t_agnewsTrain.txt", "Data source for the training")
tf.flags.DEFINE_string("word2vec", None, "Word2vec file with pre-trained embeddings (default: None)")

tf.flags.DEFINE_integer("embedding_dim", 100, "Dimensionality of word embedding (default: 128)")
tf.flags.DEFINE_string("model", "LSTM-pool", "Type of classifiers. You have three choices: [LSTM, BiLSTM, LSTM-pool, BiLSTM-pool, ATT-LSTM, ATT-BiLSTM] (default: LSTM)")
tf.flags.DEFINE_integer("hidden_layer_num", 1, "LSTM hidden layer num (default: 1)")
tf.flags.DEFINE_integer("hidden_neural_size", 100, "LSTM hidden neural size (default: 128)")
tf.flags.DEFINE_integer("attention_size", 200, "LSTM hidden neural size (default: 128)")

tf.flags.DEFINE_float("lr", 0.001, "learning rate (default=0.001)")
tf.flags.DEFINE_float("lr_decay", 0.9, "Learning rate decay rate (default: 0.98)")
tf.flags.DEFINE_float("dropout_keep_prob", 0.5, "Dropout keep probability (default: 0.5)") #살리는 확률
tf.flags.DEFINE_float("l2_reg_lambda", 1.0e-4, "L2 regularization lambda (default: 0.0)")
tf.flags.DEFINE_integer("vocab_size", 30000, "Vocabulary size (defualt: 0)")
tf.flags.DEFINE_integer("num_classes", 0, "Number of classes (defualt: 0)")

tf.flags.DEFINE_integer("batch_size", 50, "Batch Size (default: 64)")
tf.flags.DEFINE_integer("num_epochs", 200, "Number of training epochs (default: 200)")
tf.flags.DEFINE_integer("evaluate_every", 100, "Evaluate model on dev set after this many steps(literations) (default: 100)")
tf.flags.DEFINE_integer("checkpoint_every", 100, "Save model after this many steps (default: 100)")
tf.flags.DEFINE_integer("num_checkpoints", 3, "Number of checkpoints to store (default: 5)")
```

Data loading & preprocessing

- train.py

```
print("Loading data...")
x_text, y, lengths = dh.load_data(FLAGS.x_train_file, FLAGS.t_train_file)

print("Build vocabulary...")
# Build vocabulary
word_id_dict, _ = dh.buildVocab(x_text, FLAGS.vocab_size)
print(word_id_dict)
FLAGS.vocab_size = len(word_id_dict) + 4
print("vocabulary size: ", FLAGS.vocab_size)

for word_id in word_id_dict.keys():
    word_id_dict[word_id] += 4  # 0: <pad>, 1: <unk>, 2: <s>
word_id_dict['<pad>'] = 0
word_id_dict['<unk>'] = 1
word_id_dict['<s>'] = 2
word_id_dict['</s>'] = 3
```

Data loading & preprocessing

- train.py

```
print("Loading data...")
x_text, y, lengths = dh.load_data(FLAGS.x_train_file, FLAGS.t_train_file)

print("Build vocabulary...")
# Build vocabulary
word_id_dict, _ = dh.buildVocab(x_text, FLAGS.vocab_size)
print(word_id_dict)
FLAGS.vocab_size = len(word_id_dict) + 4
print("vocabulary size: ", FLAGS.vocab_size)

for word_id in word_id_dict.keys():
    word_id_dict[word_id] += 4  # 0: <pad>, 1: <unk>, 2: <s>
word_id_dict['<pad>'] = 0
word_id_dict['<unk>'] = 1
word_id_dict['<s>'] = 2
word_id_dict['</s>'] = 3
```

Data loading & preprocessing

- train.py

```
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_text = x_text[shuffle_indices]
print("Split train/validation set...")
val_sample_index = -1 * int(FLAGS.val_sample_percentage * float(len(y)))
x_train, x_val = x_text[:val_sample_index], x_text[val_sample_index:]

x_train = dh.text_to_index(x_train, word_id_dict, 0)
x_val = dh.text_to_index(x_val, word_id_dict, 0)

FLAGS.num_classes = y.shape[1]

y = y[shuffle_indices]
lengths = lengths[shuffle_indices]

y_train, y_val = y[:val_sample_index], y[val_sample_index:]
lengths, lengths_val = lengths[:val_sample_index], lengths[val_sample_index:]

print("Vocabulary Size: {:d}".format(FLAGS.vocab_size))
print("Train/Val split: {:d}/{:d}".format(len(y_train), len(y_val)))
return x_train, y_train, lengths, word_id_dict, x_val, y_val, lengths_val
```

TextRNN class & input

- text_rnn.py

```
class TextRNN(object):

    def __init__(self, config):
        self.num_classes = config["num_classes"] # e.g., positive, negatie - 2
        self.vocab_size = config["vocab_size"]
        self.hidden_size = config["hidden_neural_size"]
        self.attention_size = config["attention_size"]
        self.embedding_dim = config["embedding_dim"] # word vector size
        self.num_layers = config["hidden_layer_num"] #
        self.l2_reg_lambda = config["l2_reg_lambda"]

        self.batch_size = tf.placeholder(tf.int32, shape=(), name="batch_size")
        self.input_x = tf.placeholder(tf.int32, [None, None], name="input_x")
        self.input_y = tf.placeholder(tf.float32, [None, self.num_classes], name="input_y")
        self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")
        self.sequence_length = tf.placeholder(tf.int32, [None], name="sequence_length")

        self.l2_loss = tf.constant(0.0)
```

Embedding layer

- text_rnn.py

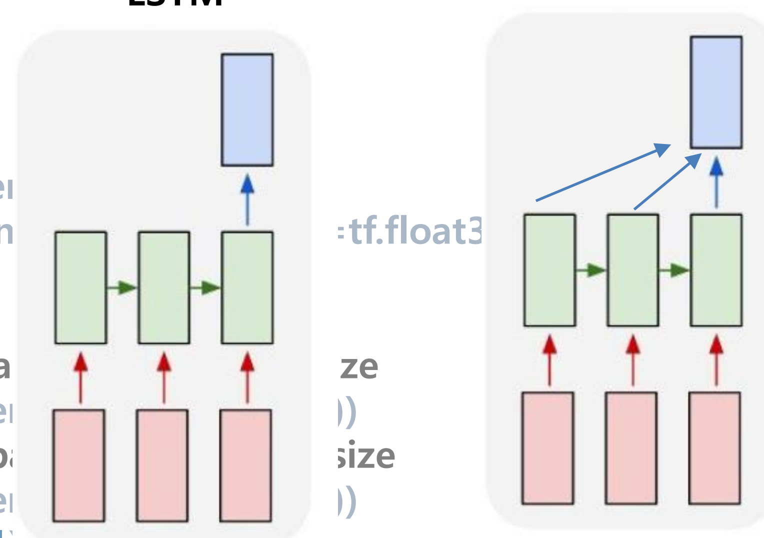
```
# Embedding layer
with tf.device('/gpu:0'), tf.name_scope("embedding"):
    self.W = tf.Variable(tf.random_uniform([self.vocab_size, self.embedding_dim], -1.0, 1.0),
trainable=True, name="W")
    self.inputs = tf.nn.embedding_lookup(self.W, self.input_x)
```

Embedding layer

- text_rnn.py

```
if config["model"] == "LSTM":
    _, self.final_state = self.normal_lstm()
elif config["model"] == "LSTM-pool":
    output, _ = self.normal_lstm()
    masks = tf.sequence_mask(lengths=self.sequence_length,
                             maxlen=tf.reduce_max(self.sequence_length), dtype=tf.float32, name='masks')
    output = output * tf.expand_dims(masks, -1)
    self.final_state = tf.div(tf.reduce_sum(output, 1), tf.expand_dims(tf.cast(self.sequence_length,
tf.float32), 1))
elif config["model"] == "BiLSTM":
    _, self.final_state = self.bi_lstm()
elif config["model"] == "BiLSTM-pool":
    output, _ = self.bi_lstm()
    masks = tf.sequence_mask(lengths=self.sequence_le
                             maxlen=tf.reduce_max(self.sequen                =tf.float3
    output_fw = output[0] * tf.expand_dims(masks, -1)
    output_bw = output[1] * tf.expand_dims(masks, -1)
    output_fw = tf.div(tf.reduce_sum(output_fw, 1), # ba                    ze
                       tf.expand_dims(tf.cast(self.sequence_le               ))
    output_bw = tf.div(tf.reduce_sum(output_bw, 1), # b                     size
                       tf.expand_dims(tf.cast(self.sequence_le               ))
    self.final_state = tf.concat([output_fw, output_bw], 1)
```

LSTM LSTM-pool



Q&A