Text classification using PyTorch

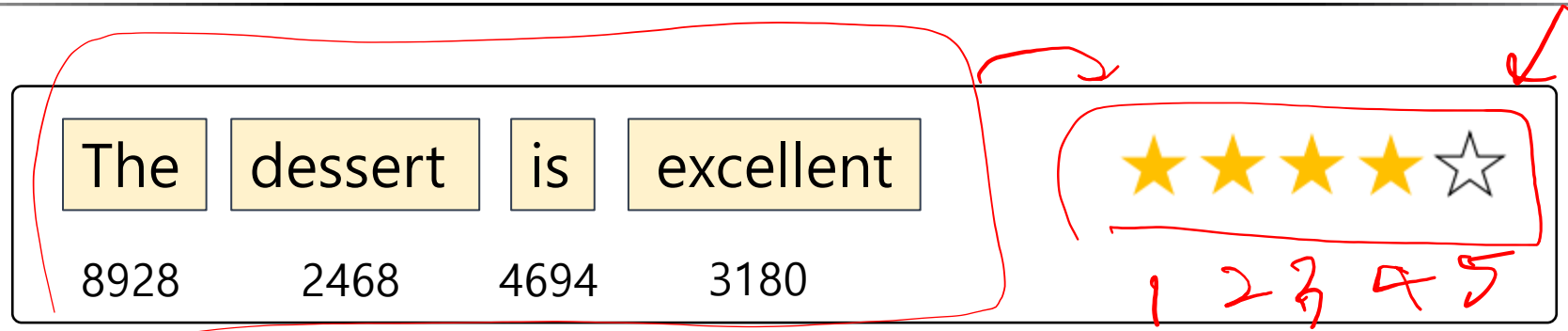
Deep Learning

Woohwan Jung

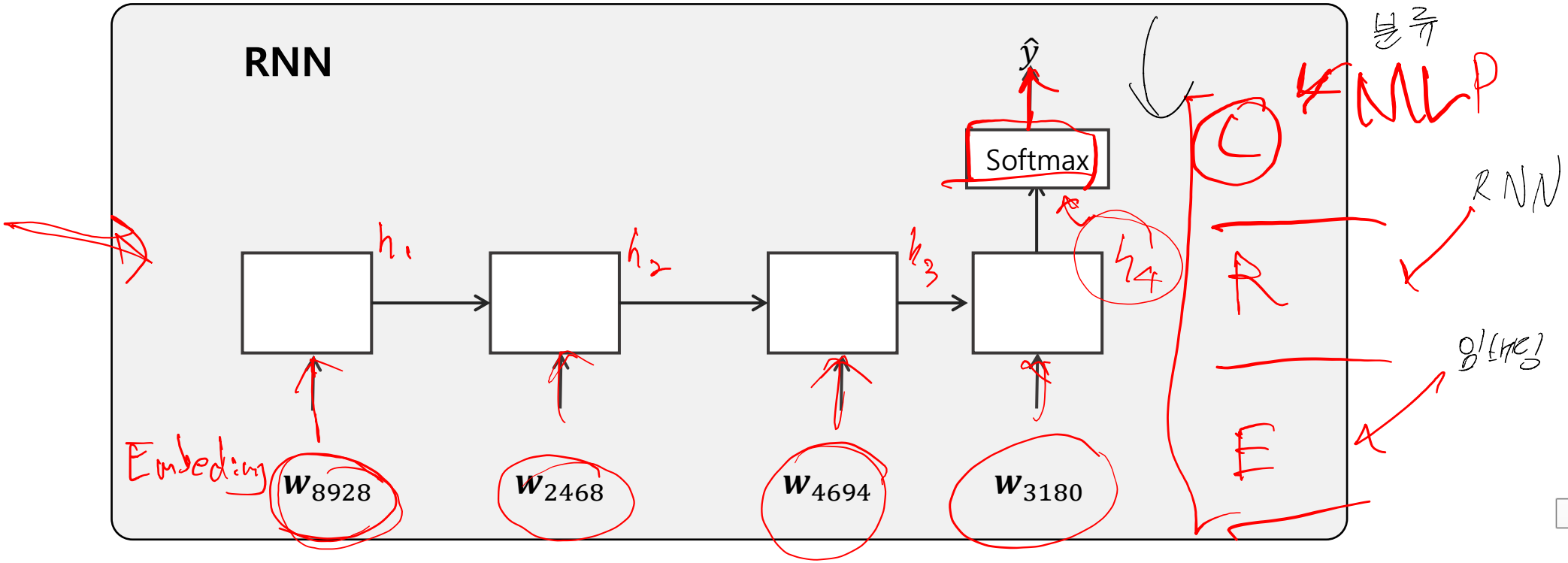
한양대학교
HANYANG UNIVERSITY

DSLAB
데이터사이언스 연

Sentiment Classification



감정분류 구성



RNN

torch.nn.RNN

CLASS `torch.nn.RNN(*args, **kwargs)` [SOURCE]

Applies a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then ReLU is used instead of $\tanh$.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

Examples:

Parameters

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

torch.nn.RNN

Inputs: input, h_0

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape (*num_layers * num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

Outputs: output, h_n

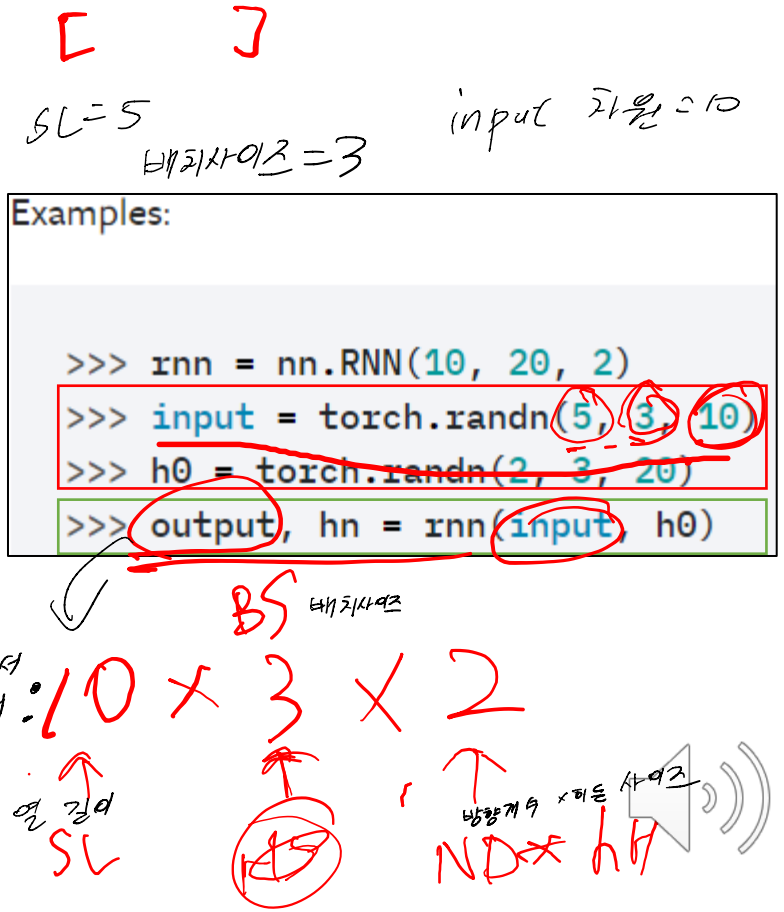
- **output** of shape (*seq_len, batch, num_directions * hidden_size*): tensor containing the output features (*h_t*) from the last layer of the RNN, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.

  For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (*num_layers * num_directions, batch, hidden_size*): tensor containing the hidden state for *t = seq_len*.

  Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)`.

Examples:

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```



torch.nn.LSTM

CLASS `torch.nn.LSTM(*args, **kwargs)`

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

Parameters

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

torch.nn.LSTM

Inputs: input, (h_0, c_0)

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. If the LSTM is bidirectional, num_directions should be 2, else it should be 1. If `proj_size > 0` was specified, the shape has to be (*num_layers*num_directions, batch, proj_size*).
- **c_0** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the initial cell state for each element in the batch.

  If (*h_0, c_0*) is not provided, both **h_0** and **c_0** default to zero.

Outputs: output, (h_n, c_n)

- **output** of shape (*seq_len, batch, num_directions*hidden_size*): tensor containing the output features (*h_t*) from the last layer of the LSTM, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence. If `proj_size > 0` was specified, output shape will be (*seq_len, batch, num_directions*proj_size*).

  For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the hidden state for *t = seq_len*. If `proj_size > 0` was specified, `h_n` shape will be (*num_layers*num_directions, batch, proj_size*). Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)` and similarly for *c_n*.
- **c_n** of shape (*num_layers*num_directions, batch, hidden_size*): tensor containing the cell state for *t = seq_len*.

계층 구성

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

은닉

→ short term state

셀 상태

→ long term state

Embedding layer

EMBEDDING

CLASS `torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None, device=None, dtype=None)` [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Parameters

- **num_embeddings** (*int*) – size of the dictionary of embeddings 사전 크기
- **embedding_dim** (*int*) – the size of each embedding vector 의 크기
- **padding_idx** (*int, optional*) – If specified, the entries at `padding_idx` do not contribute to the gradient; therefore, the embedding vector at `padding_idx` is not updated during training, i.e. it remains as a fixed "pad". For a newly constructed Embedding, the embedding vector at `padding_idx` will default to all zeros, but can be updated to another value to be used as the padding vector.

```
>>> # an Embedding module containing 10 tensors of siz
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)
tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

CLASSMETHOD from_pretrained(*embeddings*, *freeze=True*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*) [SOURCE]

Creates Embedding instance from given 2-dimensional FloatTensor.

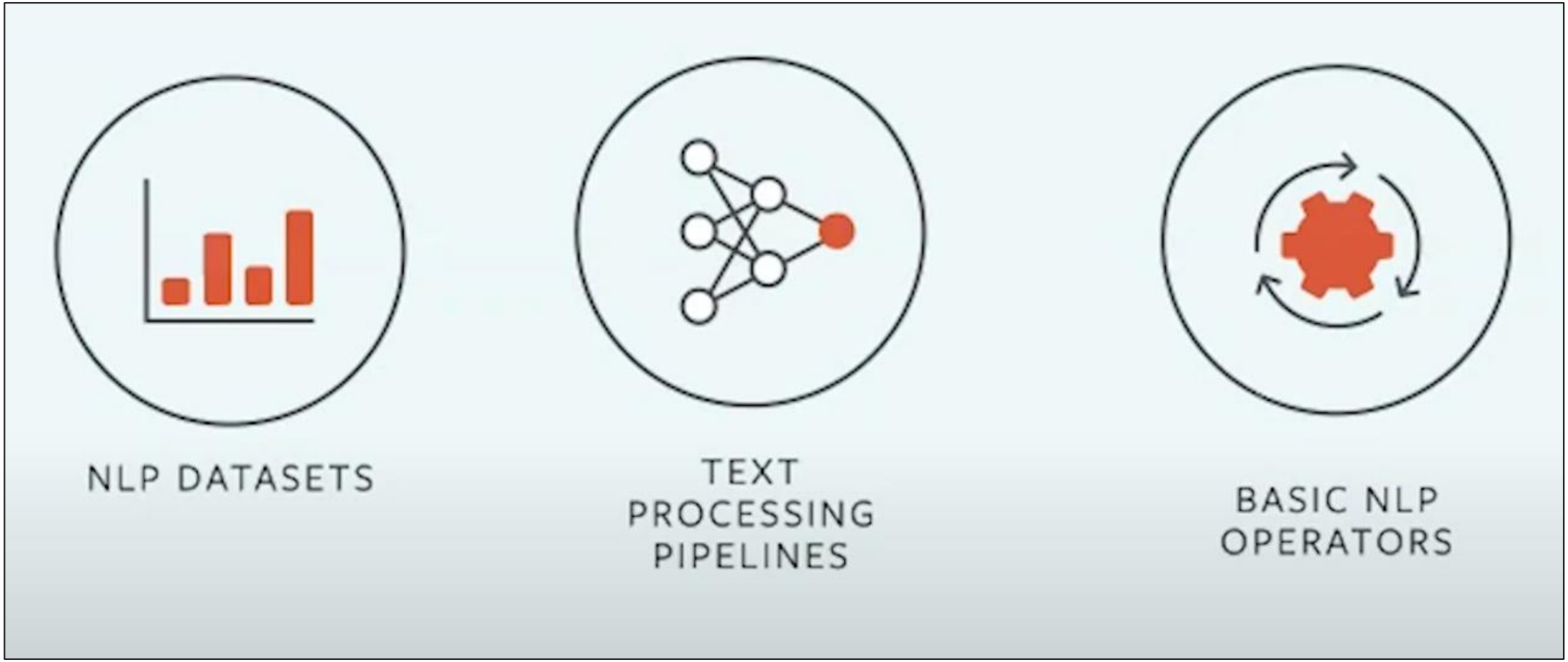
Parameters

- **embeddings** (*Tensor*) – FloatTensor containing weights for the Embedding. First dimension is being passed to Embedding as `num_embeddings`, second as `embedding_dim`.
- **freeze** (*boolean, optional*) – If `True`, the tensor does not get updated in the learning process. Equivalent to `embedding.weight.requires_grad = False`. Default: `True`

```
>>> # FloatTensor containing pretrained weights
>>> weight = torch.FloatTensor([[1, 2.3, 3], [4, 5.1, 6.3]])
>>> embedding = nn.Embedding.from_pretrained(weight)
>>> # Get embeddings for index 1
>>> input = torch.LongTensor([1])
>>> embedding(input)
tensor([[ 4.0000,  5.1000,  6.3000]])
```

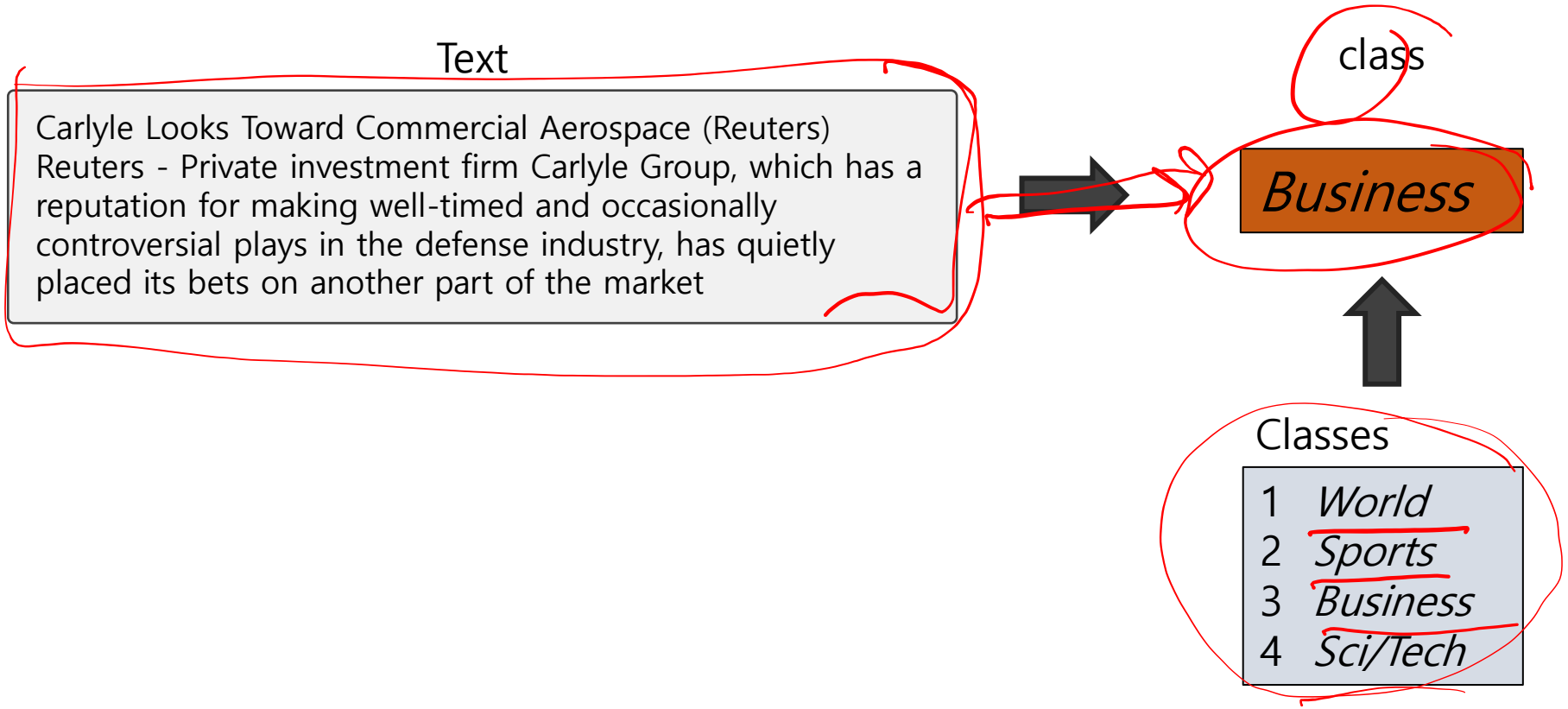
Text classification

torchtext ≈ torchvision 의 NLP 버전



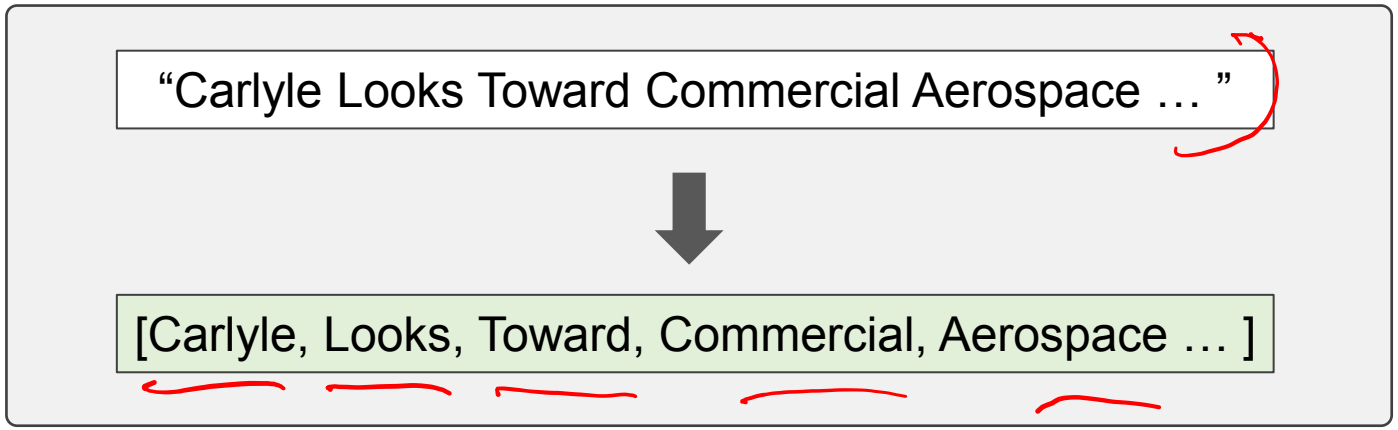
torchtext.datasets.AG_NEWS

- News classification dataset



Prepare Data Processing Piplelines

Tokenize



Build vocabulary

{<PAD>:0, <UNK>:1, '.': 2, 'the': 3, ...}

Prepare Data Processing Pipelines 데이터 준비

1

```
import torch
from torchtext.datasets import AG_NEWS
import random
from torchtext.vocab import Vectors
```

```
SEED = 5
random.seed(SEED)
torch.manual_seed(SEED)
```

```
<torch._C.Generator at 0x16aaad85390>
```

```
train_data, test_data = AG_NEWS()
```

2

```
from torchtext.data.utils import get_tokenizer
from collections import Counter
from torchtext.vocab import Vocab
```

```
tokenizer = get_tokenizer('basic_english')
```

```
tokenizer("A tokenizer is in charge of preparing the inputs for a model.")
```

```
['a',
 'tokenizer',
 'is',
 'in',
 'charge',
 'of',
 'preparing',
 'the',
 'inputs',
 'for',
 'a',
 'model',
 '.']
```

3

```
train_iter = AG_NEWS(split='train')
counter = Counter()
MAX_LEN = 0

labels =[]
for (label, line) in train_iter:
    tokens = tokenizer(line)
    counter.update(tokens)
    MAX_LEN = max(MAX_LEN, len(tokens))
    labels.append(label)

num_class = len(set(labels))
vocab = Vocab(counter, specials = ["<pad>","<unk>"],min_freq=10)
PAD = vocab["<pad>"]
print(len(vocab))
```

반복되지 않은 단어 제거

```
20645
```

4

```
# The vocabulary block converts a list of tokens into integers.
[vocab[token] for token in ['here', 'is', 'an', 'example']]
```

```
[476, 22, 31, 5298]
```

```
text_pipeline = lambda x: [vocab[token] for token in tokenizer(x)]
label_pipeline = lambda x: int(x) - 1
```

```
text_pipeline('here is the an example')
```

```
[476, 22, 3, 31, 5298]
```

Generate data batch and iterator

```
from torch.utils.data import DataLoader
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
def collate_batch(batch):
    label_list, text_list= [], []

    max_len = 0
    for (_label, _text) in batch:
        label_list.append(label_pipeline(_label))
        token_ids = text_pipeline(_text)
        text_list.append(token_ids)
        max_len = max(max_len, len(token_ids))


    text_list = [tokens + [PAD] * (max_len-len(tokens)) for tokens in text_list]
    label_list = torch.tensor(label_list, dtype=torch.int64)
    text_list = torch.tensor(text_list, dtype=torch.int64)
    return label_list.to(device), text_list.to(device)
```

Define the Model

```python
from torch import nn
from torch.nn.utils.rnn import pack_padded_sequence

class TextClassificationModel(nn.Module):
    def __init__(self, vocab, num_class, embed_dim, hidden_dim, num_rnn_layers= 1):
        super(TextClassificationModel, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(vocab.vectors)

        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers = num_rnn_layers, batch_first = True)

        self.fc = nn.Linear(hidden_dim, num_class)
        #self.init_weights(vocab)

    def init_weights(self, vocab):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()

    def forward(self, input_text):
        embedded = self.embedding(input_text)
        #packed_input = pack_padded_sequence(embedded, input_lengths.tolist(), batch_first=True)
        output, _ = self.rnn(embedded)
        h_n = output[:,-1,:] # This line will be improved in the next class
        pred = self.fc(h_n)
        return pred
```

```python
vocab.load_vectors("glove.6B.100d")

emsize = 100

model = TextClassificationModel(vocab, num_class, emsize, 128, num_rnn_layers = 1)
```

Training

```python
import time

def train(dataloader, model, criterion, optimizer):
    model.train()
    total_acc, total_count = 0, 0
    log_interval = 500
    start_time = time.time()

    loss_cum = 0.0
    for idx, (label, text) in enumerate(dataloader):
        #input_lengths = torch.LongTensor([torch.max(text[i, :].nonzero())+1 for i in range(
        #input_lengths, sorted_idx = input_lengths.sort(0, descending=True)
        #text = text[sorted_idx]
        #label = label[sorted_idx]
        optimizer.zero_grad()
        predicted_label = model(text)

        loss = criterion(predicted_label, label)
        loss.backward()
        loss_cum += loss.item()

        optimizer.step()
        total_acc += (predicted_label.argmax(1) == label).sum().item()
        total_count += label.size(0)
        if idx % log_interval == 0 and idx > 0:
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches '
                  '| loss {:8.3f}| accuracy {:8.3f}'.format(epoch, idx, len(dataloader),
                                              loss_cum/total_count, total_acc/total_count))
            total_acc, total_count, loss_cum = 0, 0, 0.0
            start_time = time.time()
```

```python
def evaluate(dataloader, model):
    model.eval()
    total_acc, total_count = 0, 0

    with torch.no_grad():
        for idx, (label, text) in enumerate(dataloader):
            predited_label = model(text)
            loss = criterion(predited_label, label)
            total_acc += (predited_label.argmax(1) == label).sum().item()
            total_count += label.size(0)
    return total_acc/total_count
```

Training

```
LR = 1e-4
EPOCHS = 5
BATCH_SIZE = 8
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
```

```
from torch.utils.data.dataset import random_split
```

```
train_iter, test_iter = AG_NEWS()
train_dataset = list(train_iter)
test_dataset = list(test_iter)
num_train = int(len(train_dataset) * 0.95)
split_train_, split_valid_ = random_split(train_dataset, [num_train, len(train_dataset) - num_train])

train_dataloader = DataLoader(split_train_, batch_size=BATCH_SIZE,
                              shuffle=True, collate_fn=collate_batch)
valid_dataloader = DataLoader(split_valid_, batch_size=BATCH_SIZE,
                              shuffle=False, collate_fn=collate_batch)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE,
                             shuffle=False, collate_fn=collate_batch)
```

```
for epoch in range(1, EPOCHS + 1):
    epoch_start_time = time.time()
    train(train_dataloader, model, criterion, optimizer)
    accu_val = evaluate(valid_dataloader, model)

    print('-' * 59)
    print('| end of epoch {:3d} | time: {:5.2f}s | '
          'valid accuracy {:8.3f} '.format(epoch,
                                           time.time() - epoch_start_time,
                                           accu_val))
    print('-' * 59)
```

```
| epoch   1 |   500/14250 batches | loss    0.173| accuracy    0.262
| epoch   1 |  1000/14250 batches | loss    0.164| accuracy    0.368
| epoch   1 |  1500/14250 batches | loss    0.170| accuracy    0.308
| epoch   1 |  2000/14250 batches | loss    0.162| accuracy    0.364
| epoch   1 |  2500/14250 batches | loss    0.145| accuracy    0.446
| epoch   1 |  3000/14250 batches | loss    0.144| accuracy    0.430
| epoch   1 |  3500/14250 batches | loss    0.142| accuracy    0.461
| epoch   1 |  4000/14250 batches | loss    0.141| accuracy    0.448
| epoch   1 |  4500/14250 batches | loss    0.143| accuracy    0.469
| epoch   1 |  5000/14250 batches | loss    0.157| accuracy    0.398
| epoch   1 |  5500/14250 batches | loss    0.154| accuracy    0.415
| epoch   1 |  6000/14250 batches | loss    0.147| accuracy    0.441
| epoch   1 |  6500/14250 batches | loss    0.145| accuracy    0.430
| epoch   1 |  7000/14250 batches | loss    0.141| accuracy    0.450
| epoch   1 |  7500/14250 batches | loss    0.140| accuracy    0.478
| epoch   1 |  8000/14250 batches | loss    0.150| accuracy    0.414
| epoch   1 |  8500/14250 batches | loss    0.140| accuracy    0.453
| epoch   1 |  9000/14250 batches | loss    0.132| accuracy    0.471
| epoch   1 |  9500/14250 batches | loss    0.129| accuracy    0.503
| epoch   1 | 10000/14250 batches | loss    0.133| accuracy    0.488
| epoch   1 | 10500/14250 batches | loss    0.132| accuracy    0.482
| epoch   1 | 11000/14250 batches | loss    0.154| accuracy    0.392
| epoch   1 | 11500/14250 batches | loss    0.163| accuracy    0.378
| epoch   1 | 12000/14250 batches | loss    0.134| accuracy    0.480
| epoch   1 | 12500/14250 batches | loss    0.123| accuracy    0.492
| epoch   1 | 13000/14250 batches | loss    0.115| accuracy    0.584
| epoch   1 | 13500/14250 batches | loss    0.104| accuracy    0.638
| epoch   1 | 14000/14250 batches | loss    0.121| accuracy    0.543
-----------------------------------------------------------
| end of epoch   1 | time: 147.70s | valid accuracy    0.580
-----------------------------------------------------------
```

```
| epoch   3 | 12000/14250 batches | loss    0.118| accuracy    0.572
| epoch   3 | 12500/14250 batches | loss    0.103| accuracy    0.677
| epoch   3 | 13000/14250 batches | loss    0.104| accuracy    0.661
| epoch   3 | 13500/14250 batches | loss    0.143| accuracy    0.523
| epoch   3 | 14000/14250 batches | loss    0.126| accuracy    0.612
-----------------------------------------------------------
| end of epoch   3 | time: 162.01s | valid accuracy    0.646
-----------------------------------------------------------
```

```
| epoch   5 | 12500/14250 batches | loss    0.136| accuracy    0.507
| epoch   5 | 13000/14250 batches | loss    0.131| accuracy    0.553
| epoch   5 | 13500/14250 batches | loss    0.126| accuracy    0.563
| epoch   5 | 14000/14250 batches | loss    0.116| accuracy    0.603
-----------------------------------------------------------
| end of epoch   5 | time: 170.73s | valid accuracy    0.662
-----------------------------------------------------------
```

Testing

```
accu_val = evaluate(test_dataloader, model)
print(f"Test accuracy {accu_val:8.3f}")
```

```
Test accuracy    0.658
```

0.658 >> 0.25

보다 좋긴함

Question

마지막 토큰 가져오는 법

```
def forward(self, input_text):
    embedded = self.embedding(input_text)
    #packed_input = pack_padded_sequence(em
    output, _ = self.rnn(embedded)
    h_n = output[:,-1, :] # This line will
    pred = self.fc(h_n)
    return pred
```

Friendly	against	Scotland	at	Murray	.
Nadim	Ladki	<PAD>	<PAD>	<PAD>	<PAD>
AL-AIN	United	Arab	Emirates	<PAD>	<PAD>
ROME	1996-12	<PAD>	<PAD>	<PAD>	<PAD>
Two	goals	in	the	last	minutes

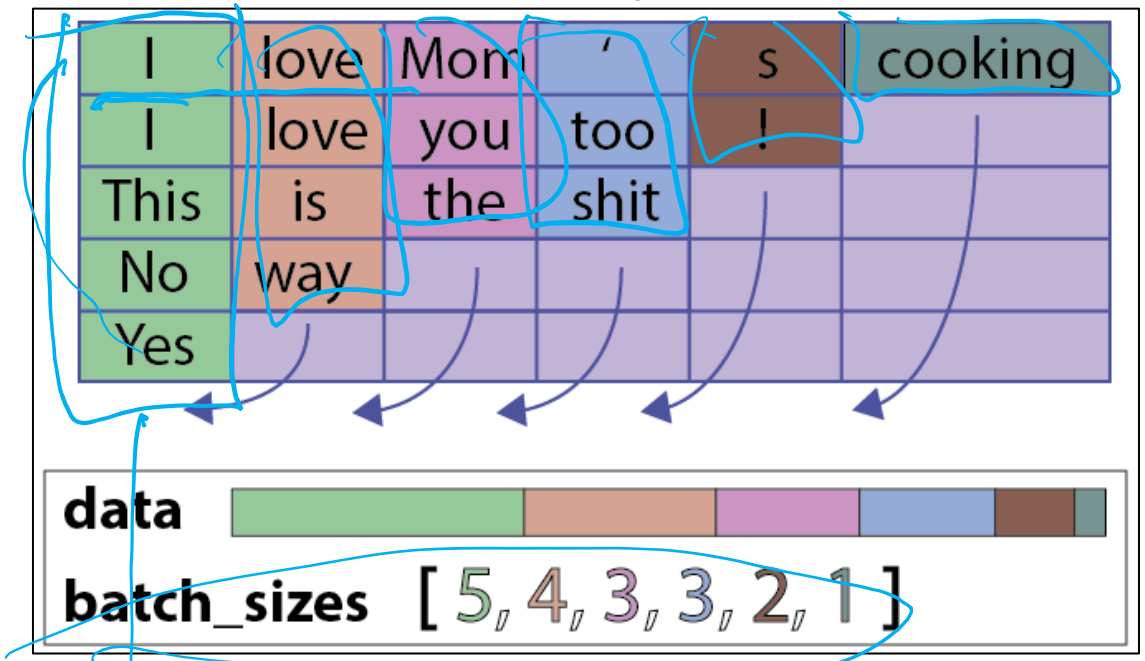
Packed Sequence

Packed Sequence

Normal tensor

I	love	Mom	'	s	cooking
I	love	you	too	!	
No	way				
This	is	the	shit		
Yes					

Packed sequence



Source: [Understanding emotions — from Keras to pyTorch]

Packed Sequence – Pytorch

Input sequences

```
input_seq2idx
================================================
tensor([[ 1, 16, 7, 11, 13, 2],
        [ 1, 16, 6, 15, 8, 0],
        [ 12, 9, 0, 0, 0, 0],
        [ 5, 14, 3, 17, 0, 0],
        [ 10, 0, 0, 0, 0, 0]])
```

Sort inputs by lengths

```
input_lengths = torch.LongTensor([torch.max(input_seq2idx[i, :].data.nonzero())+1 for i in range(input_seq2idx.size(0))])
input_lengths, sorted_idx = input_lengths.sort(0, descending=True)
input_seq2idx = input_seq2idx[sorted_idx]
```

Sorted inputs

```
input_seq2idx, input_lengths
================================================
tensor([[ 1, 16, 7, 11, 13, 2],
        [ 1, 16, 6, 15, 8, 0],
        [ 5, 14, 3, 17, 0, 0],
        [ 12, 9, 0, 0, 0, 0],
        [ 10, 0, 0, 0, 0, 0]])
tensor([ 6, 5, 4, 2, 1])
```

https://simonjisu.github.io/nlp/2018/07/05/packedsequence.html

Packed Sequence with RNN

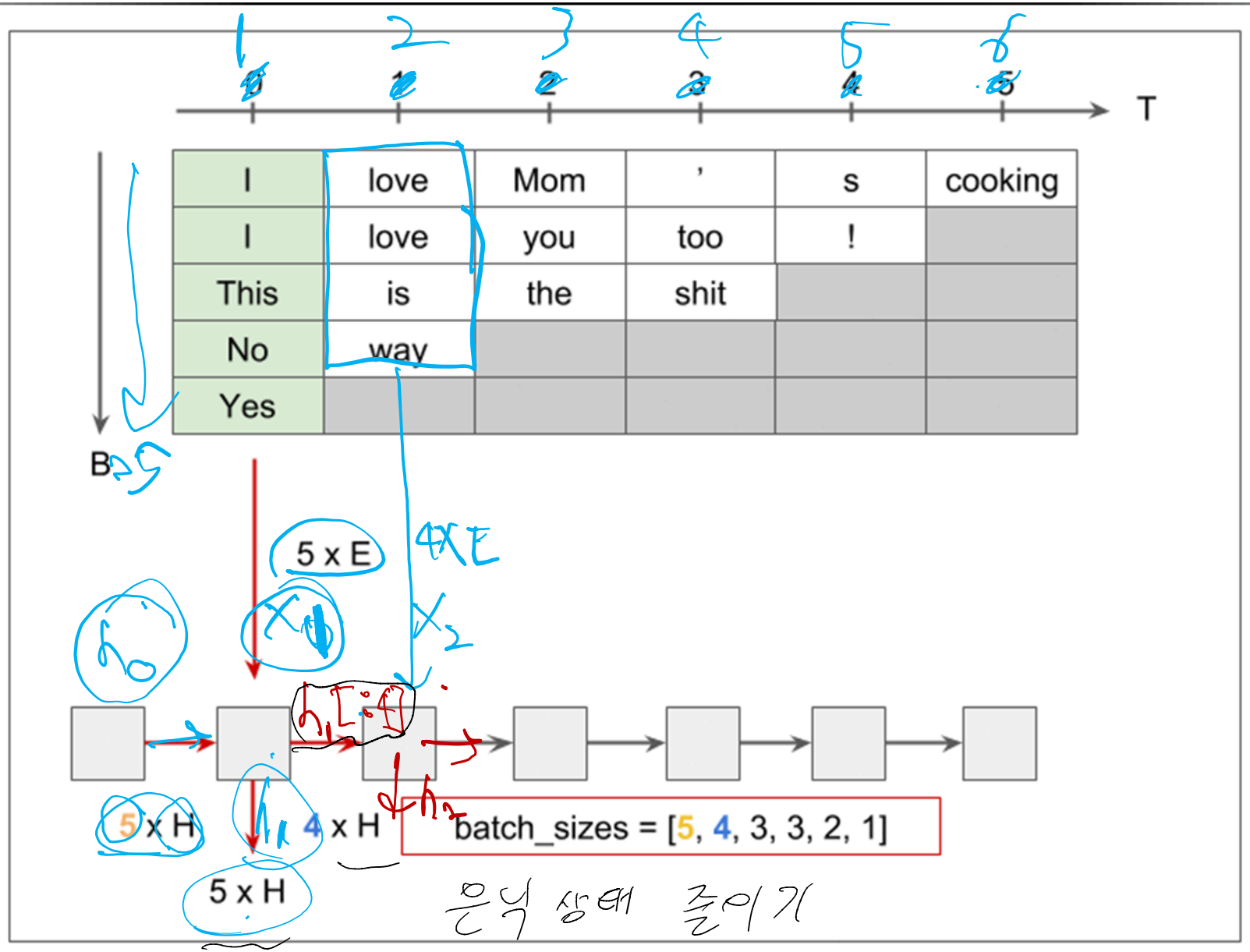
Define layers

```
embed = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
rnn = nn.RNN(input_size=embedding_size, hidden_size=hidden_size, batch_first=True)
```

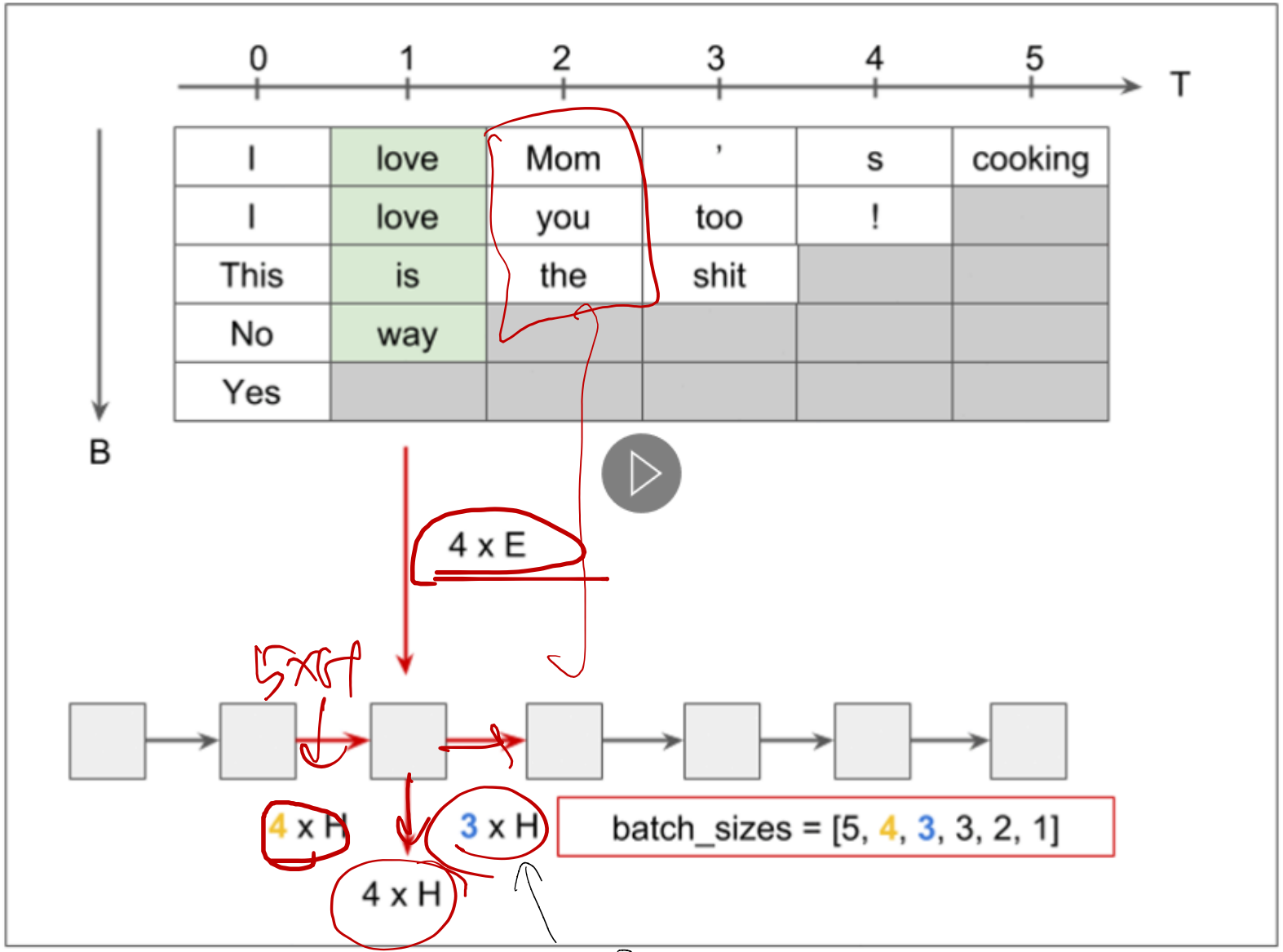
Using RNNs with packed sequences

```
embedded = embed(input_seq2idx)
packed_input = pack_padded_sequence(embedded, input_lengths.tolist(), batch_first=True)
packed_output, hidden = rnn(packed_input)
```

Packed Sequence with RNN



Packed Sequence with RNN



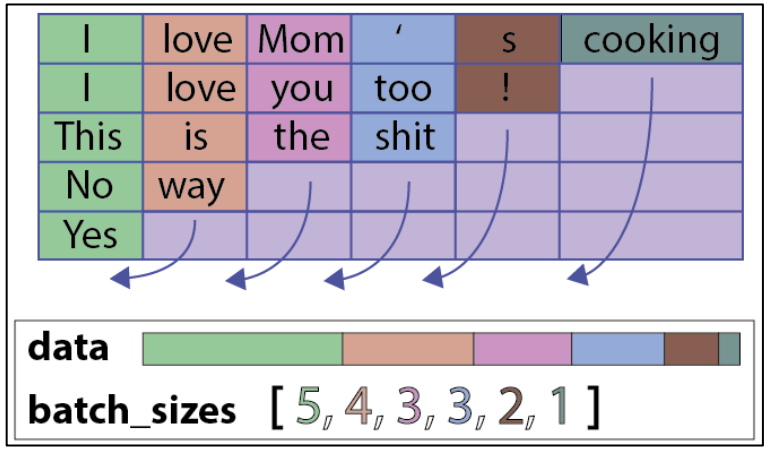
Text classification

With packed sequences

Define the Model

```python
from torch import nn
from torch.nn.utils.rnn import pack_padded_sequence

class TextClassificationModel(nn.Module):
    def __init__(self, vocab, num_class, embed_dim, hidden_dim, num_rnn_layers= 1):
        super(TextClassificationModel, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(vocab.vectors)

        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers = num_rnn_layers, batch_first = True)

        self.fc = nn.Linear(hidden_dim, num_class)
        #self.init_weights(vocab)

    def init_weights(self, vocab):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()

    def forward(self, input_text, input_lengths):
        embedded = self.embedding(input_text)
        packed_input = pack_padded_sequence(embedded, input_lengths.tolist(), batch_first=True)

        packed_output, hidden = self.rnn(packed_input)
        #output, _ = self.rnn(embedded)
        #h_n = output[:,-1, :] # this line will be improved in the next class
        h_n = hidden[0]
        pred = self.fc(h_n)
        return pred
```

Last hidden states



왜 사용하기

output[:, -1]

같은값?

Training

```
import time

def train(dataloader, model, criterion, optimizer):
    model.train()
    total_acc, total_count = 0, 0
    log_interval = 100
    start_time = time.time()

    loss_cum = 0.0
    for idx, (label, text) in enumerate(dataloader):
        input_lengths = torch.LongTensor([torch.max(text[i, :].nonzero())+1 for i in range(text.size(0))])
        input_lengths, sorted_idx = input_lengths.sort(0, descending=True)
        text = text[sorted_idx]
        label = label[sorted_idx]

        optimizer.zero_grad()
        predicted_label = model(text, input_lengths)


        loss = criterion(predicted_label, label)
        loss.backward()
        loss_cum += loss.item()

        optimizer.step()
        total_acc += (predicted_label.argmax(1) == label).sum().item()
        total_count += label.size(0)
        if idx % log_interval == 0 and idx > 0:
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches '
                  '| loss {:8.3f}| accuracy {:8.3f}'.format(epoch, idx, len(dataloader),
                                              loss_cum/total_count, total_acc/total_count))
            total_acc, total_count, loss_cum = 0, 0, 0.0
            start_time = time.time()
```

Batchsize 4

Packed sequence

```
| epoch   1 |   500/14250 batches | loss    0.139| accuracy    0.539
| epoch   1 |  1000/14250 batches | loss    0.094| accuracy    0.698
| epoch   1 |  1500/14250 batches | loss    0.077| accuracy    0.761
| epoch   1 |  2000/14250 batches | loss    0.073| accuracy    0.785
| epoch   1 |  2500/14250 batches | loss    0.064| accuracy    0.825
| epoch   1 |  3000/14250 batches | loss    0.063| accuracy    0.840
| epoch   1 |  3500/14250 batches | loss    0.054| accuracy    0.864
| epoch   1 |  4000/14250 batches | loss    0.067| accuracy    0.816
| epoch   1 |  4500/14250 batches | loss    0.058| accuracy    0.857
| epoch   1 |  5000/14250 batches | loss    0.055| accuracy    0.861
| epoch   1 |  5500/14250 batches | loss    0.058| accuracy    0.849
| epoch   1 |  6000/14250 batches | loss    0.055| accuracy    0.859
| epoch   1 |  6500/14250 batches | loss    0.055| accuracy    0.868
| epoch   1 |  7000/14250 batches | loss    0.056| accuracy    0.855
| epoch   1 |  7500/14250 batches | loss    0.057| accuracy    0.856
| epoch   1 |  8000/14250 batches | loss    0.052| accuracy    0.870
| epoch   1 |  8500/14250 batches | loss    0.055| accuracy    0.858
| epoch   1 |  9000/14250 batches | loss    0.054| accuracy    0.863
| epoch   1 |  9500/14250 batches | loss    0.076| accuracy    0.746
| epoch   1 | 10000/14250 batches | loss    0.080| accuracy    0.766
| epoch   1 | 10500/14250 batches | loss    0.062| accuracy    0.831
| epoch   1 | 11000/14250 batches | loss    0.055| accuracy    0.857
| epoch   1 | 11500/14250 batches | loss    0.055| accuracy    0.857
| epoch   1 | 12000/14250 batches | loss    0.055| accuracy    0.854
| epoch   1 | 12500/14250 batches | loss    0.052| accuracy    0.867
| epoch   1 | 13000/14250 batches | loss    0.054| accuracy    0.866
| epoch   1 | 13500/14250 batches | loss    0.052| accuracy    0.864
| epoch   1 | 14000/14250 batches | loss    0.052| accuracy    0.869
-----------------------------------------------------------
| end of epoch   1 | time: 165.30s | valid accuracy    0.861
-----------------------------------------------------------
```

Padded Seq

```
def forward(self, input_text):
    embedded = self.embedding(input_text)
    #packed_input = pack_padded_sequence(em
    output, _ = self.rnn(embedded)
    h_n = output[:,-1, :] # This line will
    pred = self.fc(h_n)
    return pred
```

```
| epoch   1 |   500/14250 batches | loss    0.173| accuracy    0.262
| epoch   1 |  1000/14250 batches | loss    0.164| accuracy    0.368
| epoch   1 |  1500/14250 batches | loss    0.170| accuracy    0.308
| epoch   1 |  2000/14250 batches | loss    0.162| accuracy    0.364
| epoch   1 |  2500/14250 batches | loss    0.145| accuracy    0.446
| epoch   1 |  3000/14250 batches | loss    0.144| accuracy    0.430
| epoch   1 |  3500/14250 batches | loss    0.142| accuracy    0.461
| epoch   1 |  4000/14250 batches | loss    0.141| accuracy    0.448
| epoch   1 |  4500/14250 batches | loss    0.143| accuracy    0.469
| epoch   1 |  5000/14250 batches | loss    0.157| accuracy    0.398
| epoch   1 |  5500/14250 batches | loss    0.154| accuracy    0.415
| epoch   1 |  6000/14250 batches | loss    0.147| accuracy    0.441
| epoch   1 |  6500/14250 batches | loss    0.145| accuracy    0.430
| epoch   1 |  7000/14250 batches | loss    0.141| accuracy    0.450
| epoch   1 |  7500/14250 batches | loss    0.140| accuracy    0.478
| epoch   1 |  8000/14250 batches | loss    0.150| accuracy    0.414
| epoch   1 |  8500/14250 batches | loss    0.140| accuracy    0.453
| epoch   1 |  9000/14250 batches | loss    0.132| accuracy    0.471
| epoch   1 |  9500/14250 batches | loss    0.129| accuracy    0.503
| epoch   1 | 10000/14250 batches | loss    0.133| accuracy    0.488
| epoch   1 | 10500/14250 batches | loss    0.132| accuracy    0.482
| epoch   1 | 11000/14250 batches | loss    0.154| accuracy    0.392
| epoch   1 | 11500/14250 batches | loss    0.163| accuracy    0.378
| epoch   1 | 12000/14250 batches | loss    0.134| accuracy    0.480
| epoch   1 | 12500/14250 batches | loss    0.123| accuracy    0.492
| epoch   1 | 13000/14250 batches | loss    0.115| accuracy    0.584
| epoch   1 | 13500/14250 batches | loss    0.104| accuracy    0.638
| epoch   1 | 14000/14250 batches | loss    0.121| accuracy    0.543
-----------------------------------------------------------
| end of epoch   1 | time: 147.70s | valid accuracy    0.580
-----------------------------------------------------------
```

성능 증가

Batchsize 32

정확성 수준을 올려야 함

Packed sequence

```
def forward(self, input_text):
    embedded = self.embedding(input_text)
    #packed_input = pack_padded_sequence(em
    output, _ = self.rnn(embedded)
    h_n = output[:,-1,:] # This line will
    pred = self.fc(h_n)
    return pred
```

```
| epoch   1 |   900/ 3563 batches | loss    0.015| accuracy    0.832
| epoch   1 |  1000/ 3563 batches | loss    0.014| accuracy    0.849
| epoch   1 |  1100/ 3563 batches | loss    0.014| accuracy    0.863
| epoch   1 |  1200/ 3563 batches | loss    0.014| accuracy    0.854
| epoch   1 |  1300/ 3563 batches | loss    0.013| accuracy    0.864
| epoch   1 |  1400/ 3563 batches | loss    0.014| accuracy    0.860
| epoch   1 |  1500/ 3563 batches | loss    0.012| accuracy    0.870
| epoch   1 |  1600/ 3563 batches | loss    0.013| accuracy    0.878
| epoch   1 |  1700/ 3563 batches | loss    0.013| accuracy    0.870
| epoch   1 |  1800/ 3563 batches | loss    0.014| accuracy    0.858
| epoch   1 |  1900/ 3563 batches | loss    0.013| accuracy    0.866
| epoch   1 |  2000/ 3563 batches | loss    0.012| accuracy    0.876
| epoch   1 |  2100/ 3563 batches | loss    0.013| accuracy    0.871
| epoch   1 |  2200/ 3563 batches | loss    0.013| accuracy    0.869
| epoch   1 |  2300/ 3563 batches | loss    0.013| accuracy    0.868
| epoch   1 |  2400/ 3563 batches | loss    0.012| accuracy    0.876
| epoch   1 |  2500/ 3563 batches | loss    0.013| accuracy    0.864
| epoch   1 |  2600/ 3563 batches | loss    0.012| accuracy    0.875
| epoch   1 |  2700/ 3563 batches | loss    0.013| accuracy    0.863
| epoch   1 |  2800/ 3563 batches | loss    0.012| accuracy    0.871
| epoch   1 |  2900/ 3563 batches | loss    0.013| accuracy    0.871
| epoch   1 |  3000/ 3563 batches | loss    0.012| accuracy    0.874
| epoch   1 |  3100/ 3563 batches | loss    0.012| accuracy    0.885
| epoch   1 |  3200/ 3563 batches | loss    0.012| accuracy    0.880
| epoch   1 |  3300/ 3563 batches | loss    0.012| accuracy    0.870
| epoch   1 |  3400/ 3563 batches | loss    0.012| accuracy    0.873
| epoch   1 |  3500/ 3563 batches | loss    0.012| accuracy    0.880
-----------------------------------------------------------
| end of epoch   1 | time: 77.12s | valid accuracy    0.871 |
-----------------------------------------------------------
```

>>>

```
| epoch   1 |   900/ 3563 batches | loss    0.037| accuracy    0.458
| epoch   1 |  1000/ 3563 batches | loss    0.035| accuracy    0.459
| epoch   1 |  1100/ 3563 batches | loss    0.039| accuracy    0.403
| epoch   1 |  1200/ 3563 batches | loss    0.039| accuracy    0.419
| epoch   1 |  1300/ 3563 batches | loss    0.037| accuracy    0.441
| epoch   1 |  1400/ 3563 batches | loss    0.044| accuracy    0.273
| epoch   1 |  1500/ 3563 batches | loss    0.040| accuracy    0.393
| epoch   1 |  1600/ 3563 batches | loss    0.035| accuracy    0.475
| epoch   1 |  1700/ 3563 batches | loss    0.034| accuracy    0.464
| epoch   1 |  1800/ 3563 batches | loss    0.037| accuracy    0.431
| epoch   1 |  1900/ 3563 batches | loss    0.036| accuracy    0.450
| epoch   1 |  2000/ 3563 batches | loss    0.044| accuracy    0.316
| epoch   1 |  2100/ 3563 batches | loss    0.043| accuracy    0.280
| epoch   1 |  2200/ 3563 batches | loss    0.043| accuracy    0.284
| epoch   1 |  2300/ 3563 batches | loss    0.043| accuracy    0.264
| epoch   1 |  2400/ 3563 batches | loss    0.043| accuracy    0.291
| epoch   1 |  2500/ 3563 batches | loss    0.042| accuracy    0.287
| epoch   1 |  2600/ 3563 batches | loss    0.042| accuracy    0.316
| epoch   1 |  2700/ 3563 batches | loss    0.042| accuracy    0.317
| epoch   1 |  2800/ 3563 batches | loss    0.042| accuracy    0.318
| epoch   1 |  2900/ 3563 batches | loss    0.042| accuracy    0.306
| epoch   1 |  3000/ 3563 batches | loss    0.042| accuracy    0.316
| epoch   1 |  3100/ 3563 batches | loss    0.042| accuracy    0.325
| epoch   1 |  3200/ 3563 batches | loss    0.041| accuracy    0.318
| epoch   1 |  3300/ 3563 batches | loss    0.038| accuracy    0.394
| epoch   1 |  3400/ 3563 batches | loss    0.038| accuracy    0.389
| epoch   1 |  3500/ 3563 batches | loss    0.038| accuracy    0.396
-----------------------------------------------------------
| end of epoch   1 | time: 71.57s | valid accuracy    0.394 |
-----------------------------------------------------------
```

References

- https://github.com/bentrevett/pytorch-sentiment-analysis