June 21 2022

2022 Samsung AI 교육과정

Text Classification using RNN: Coding Exercise

Kyomin Jung

Department of Electrical and Computer Engineering
Seoul National University

About Today Class

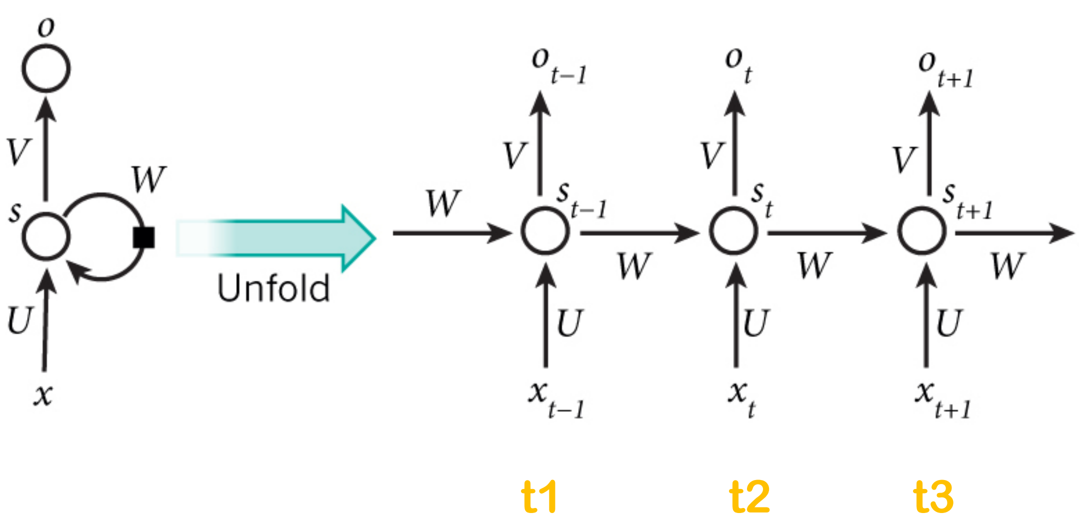
- **Today's TA**
  - 장윤아 **(vn2209@snu.ac.kr)**
  - 이동렬 **(drl123@snu.ac.kr)**

Contents

- Text classification using RNN
  - Preliminaries
    - RNNs
    - Word embedding
    - Attention
  - Practice
    - Load/preprocess data with PyTorch Torchtext library
    - Build model
    - Train model

Review: RNNs

- RNNs(Recurrent Neural Networks) are a very natural way to model sequential data
- Many applications in Natural Language Processing (NLP)
  - Text = Sequence of word
  - Classification: POS tagging, Sentiment Analysis …
  - Generation: Machine translation, Chatbot …



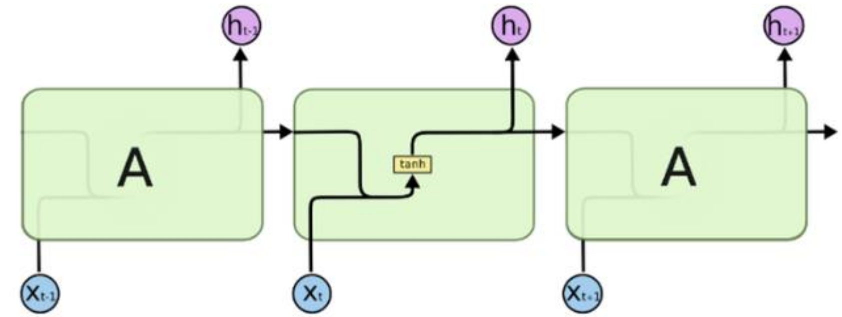
$$s_t = f(Ux_t + Ws_{t-1})$$

$$o_t = Vh_t$$

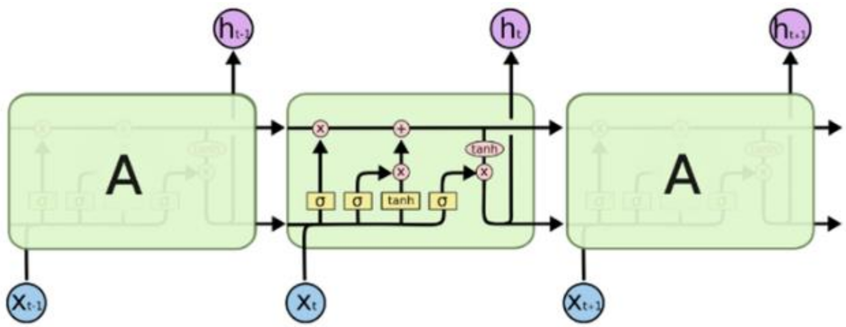
s_t : hidden state

f : activation function (Sigmoid, Tanh, ReLU)

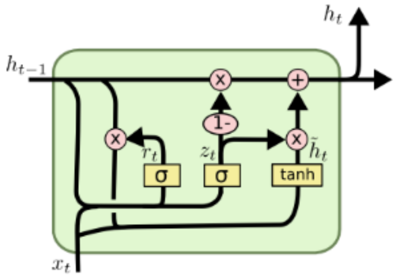
Review: RNNs



Vanilla RNN



LSTM
(Long Short Term Memory)



GRU(Gated Recurrent Unit)

RNNs: Process Sequences

Sentiment Classification
(seq of words → sentiment)

Video classification on frame level

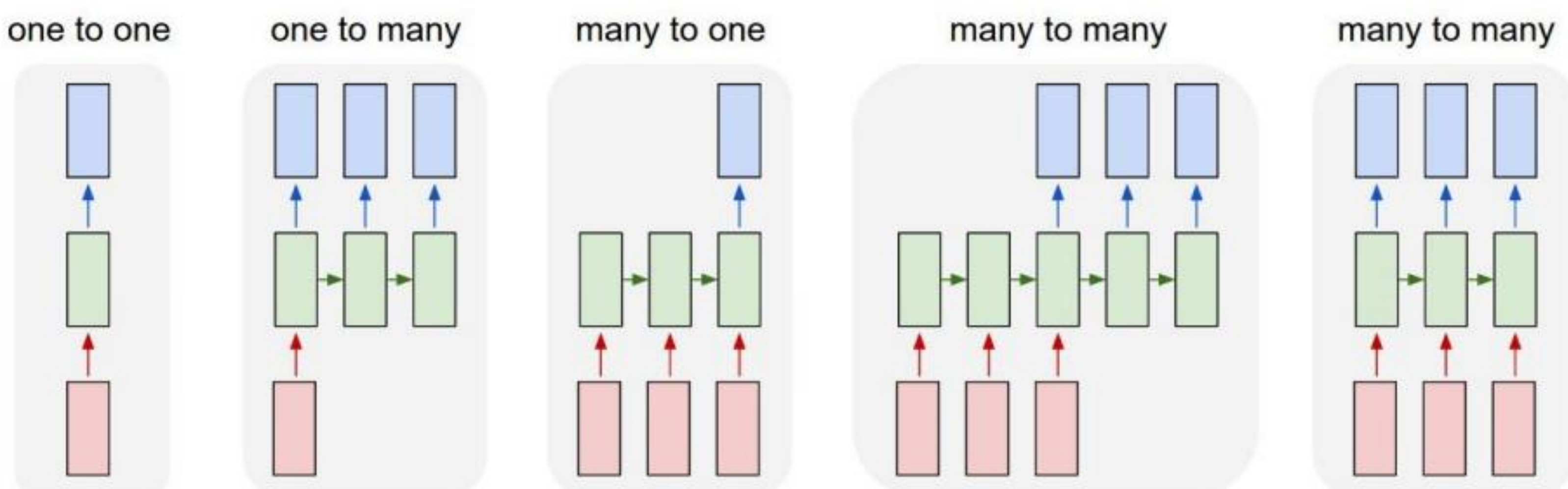
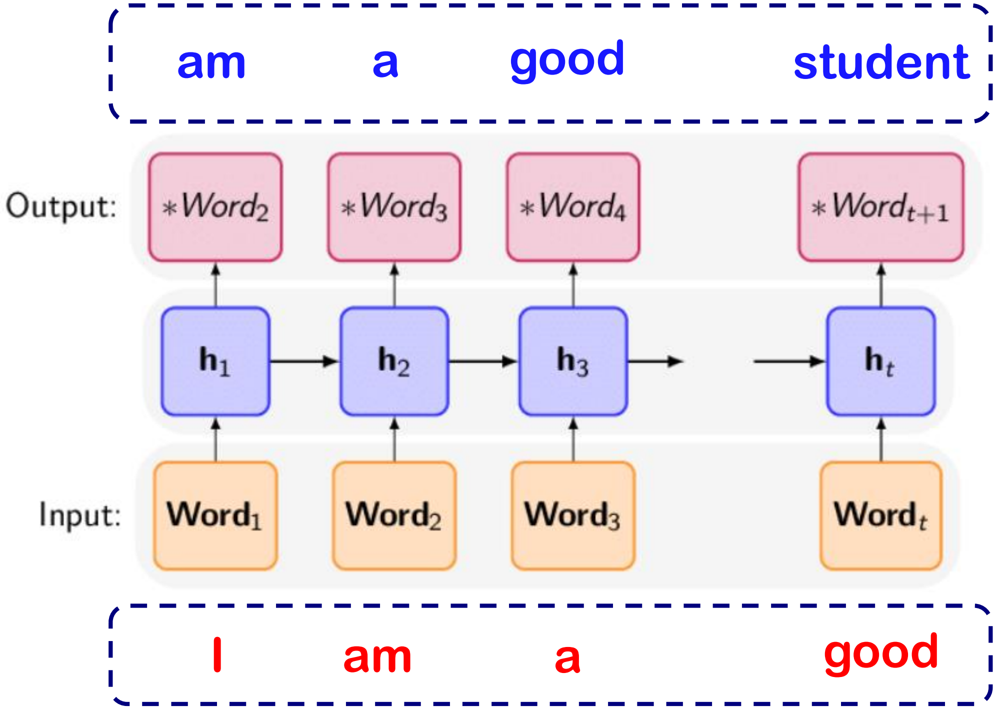


Image Captioning
(image → seq of words)

Machine Translation
(seq of words → seq of words)

Example: Language Model

- **Language Modeling**
  - Task of understanding the probability distribution over a sequence of words

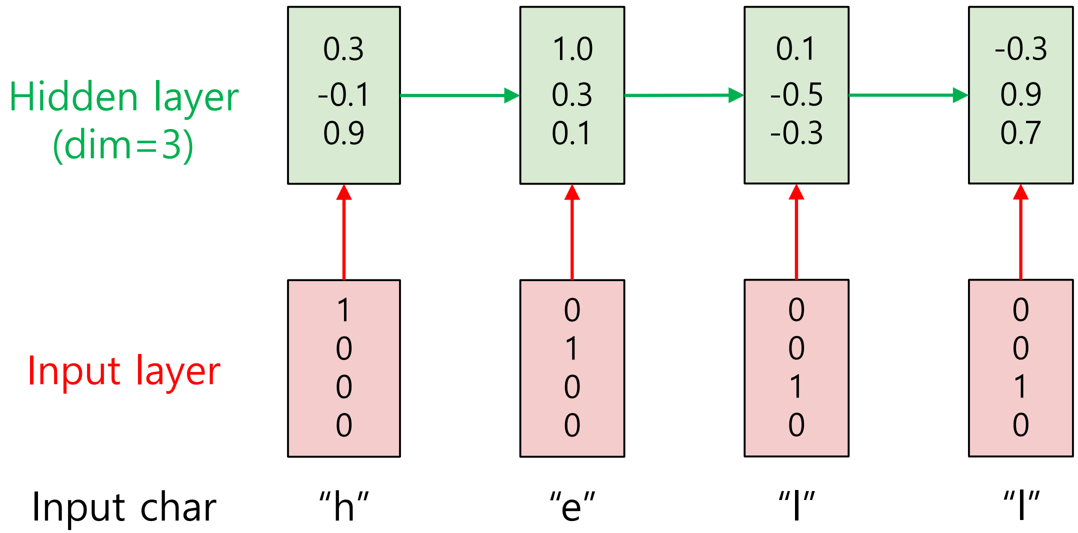


Example: Language Model

- **Ex) Character-level Language Model**
  - **Vocabulary = [h, e, l, o]**

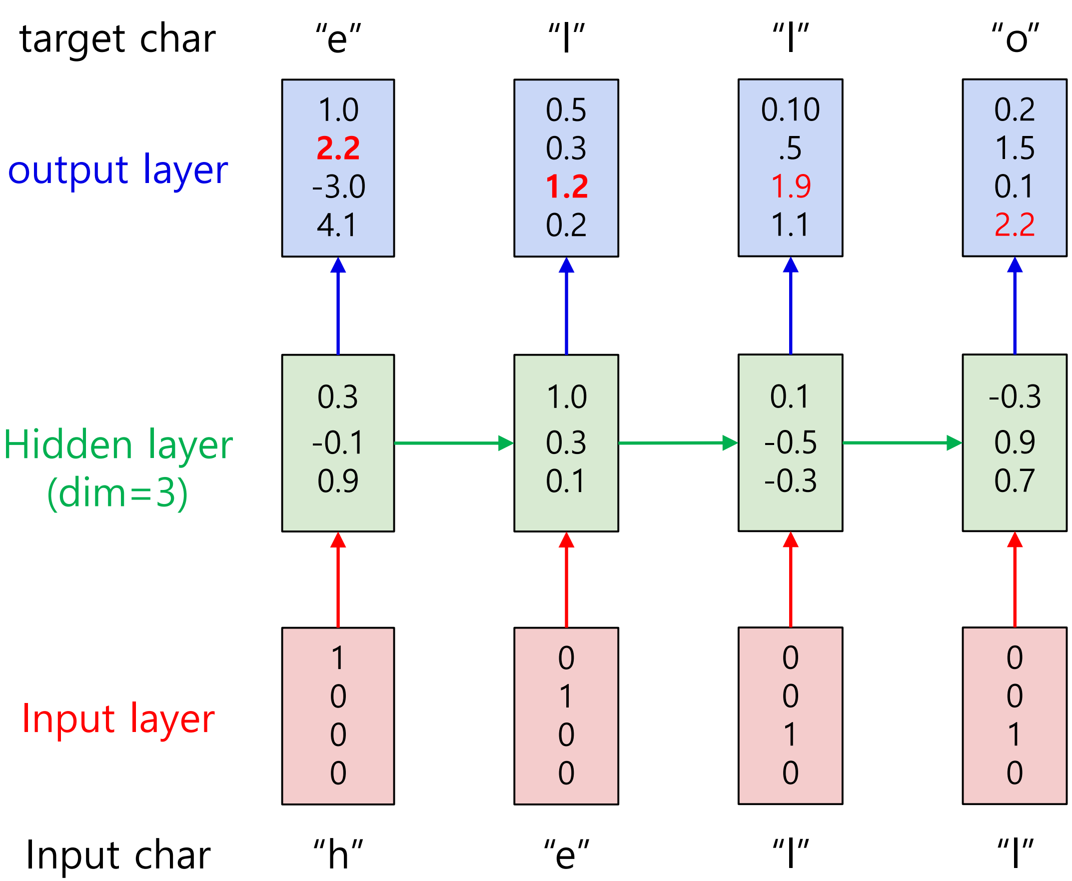
"h" = [1,0,0,0]
"e" = [0,1,0,0]
"l" = [0,0,1,0]
"o" = [0,0,0,1]

$$\boldsymbol{s_t = f(Ux_t + Ws_{t-1})}$$



Example: Language Model

- Ex) Character-level Language Model

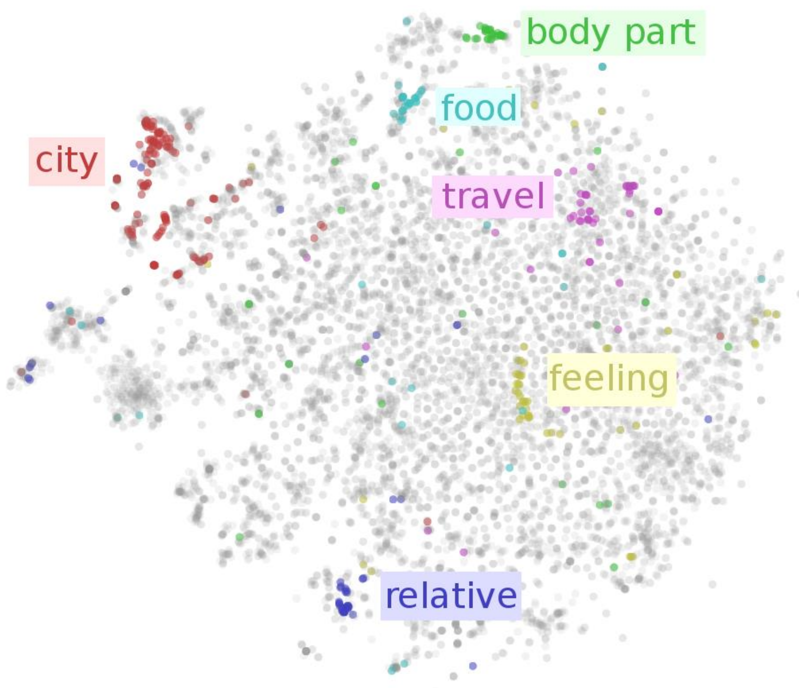


$$\boldsymbol{o_t = V h_t}$$

$$\boldsymbol{s_t = f(U x_t + W s_{t-1})}$$

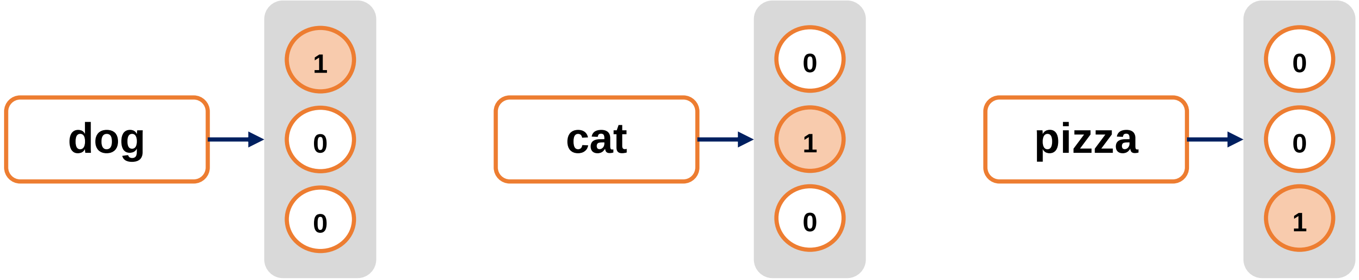
Word embedding

- Word Embedding
  - A representation that maps words to real-valued vectors



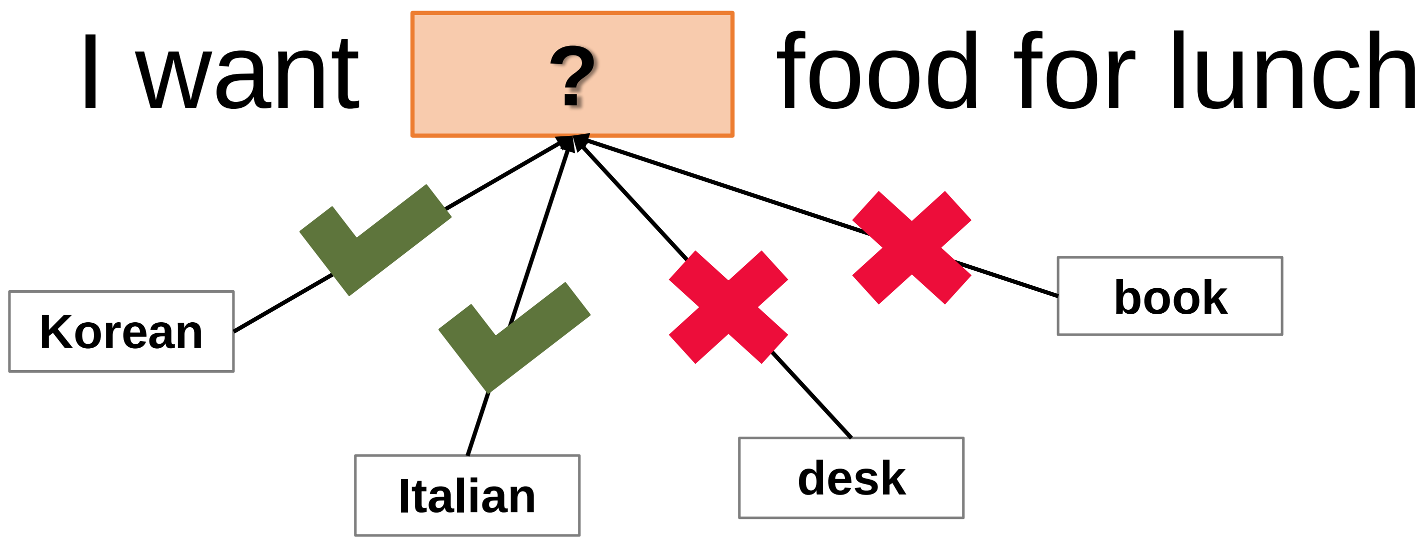
Word embedding

- Bag-of-Words(BoW)
  - A text is represented as the bag of its words
  - ex) One-hot encoding



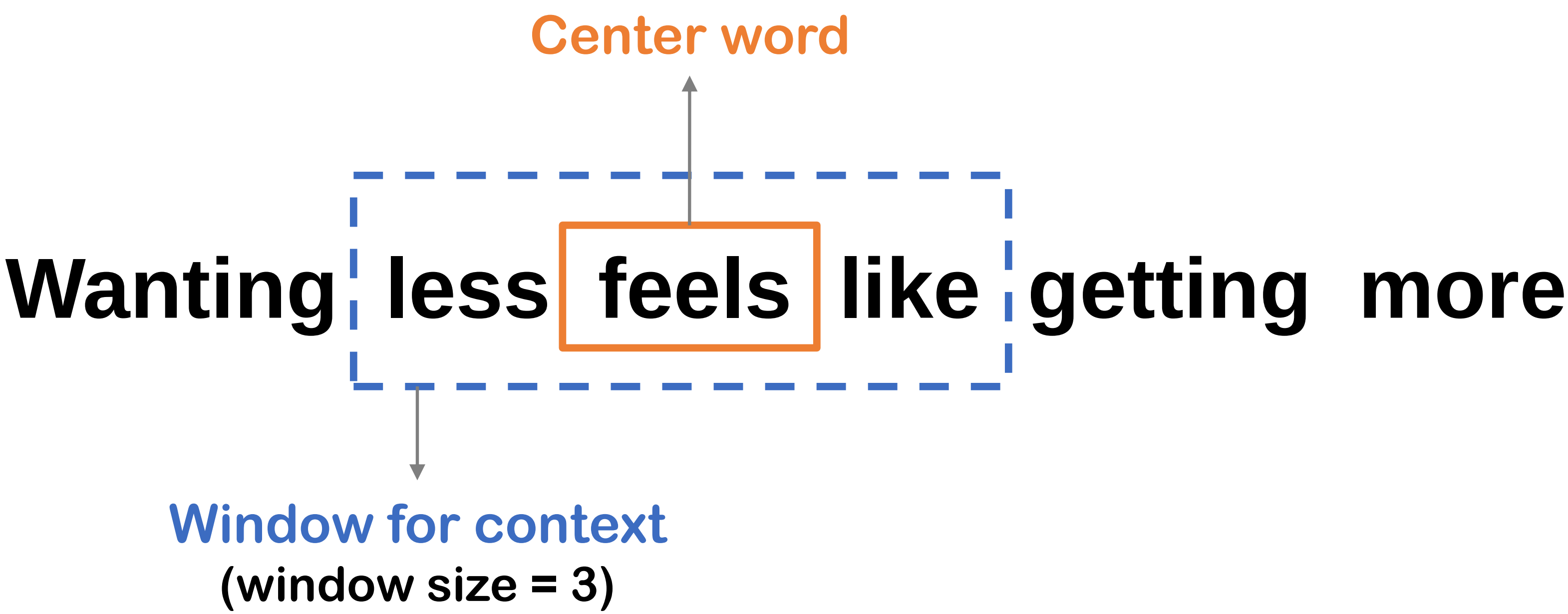
Word embedding

- Word2Vec
  - One of the most popular techniques (Tomas Mikolov, 2013)
  - Constructs word embeddings where words with similar context are embedded close to each other
  - CBOW and Skip-gram models



Word embedding

- How to generate word embedding in Word2Vec
  - Center word and its context words



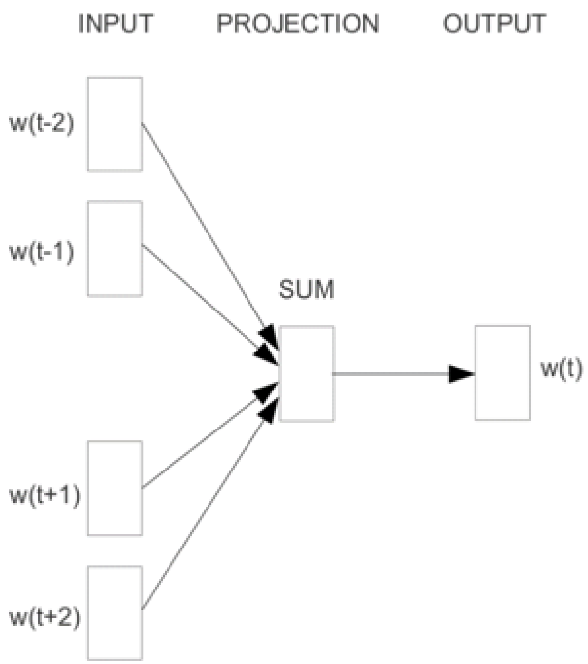
Word embedding

- Sliding window (window size = 5)

	w(t)	w(t+1)	w(t+2)			
#1	Wanting	Less	feels	like	getting	more
#2	Wanting	less	feels	like	getting	more
	w(t-2)	w(t-1)	w(t)	w(t+1)	w(t+2)	
#3	Wanting	less	feels	like	getting	more
#4	Wanting	less	feels	like	getting	more
#5	Wanting	less	feels	like	getting	more
				w(t-2)	w(t-1)	w(t)
#6	Wanting	less	feels	like	getting	more

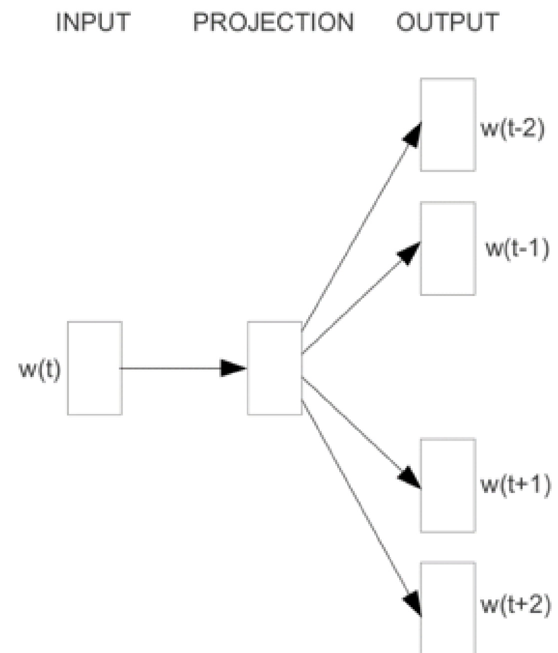
Word embedding

- **CBOW**(Continuous Bag-of-Words)



$$E = -\log p(w_t | w_{t-c}, \ldots w_{t+c})$$

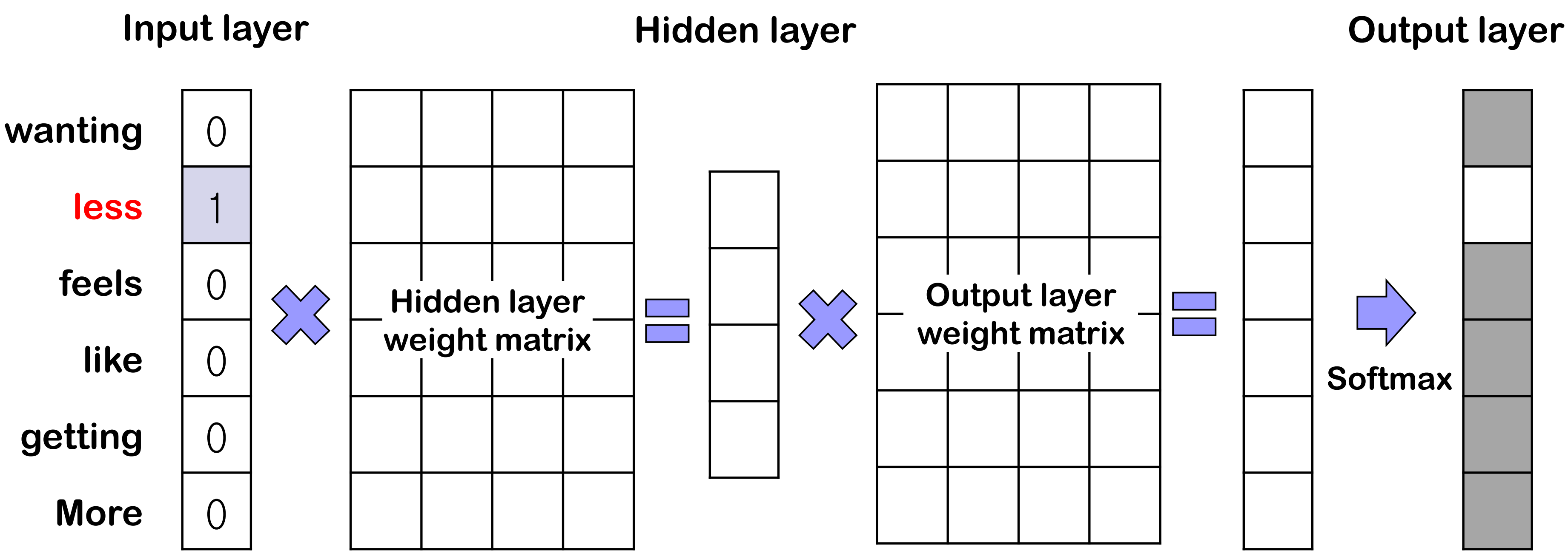
- **Skip-gram**



$$E = -\log p(w_{t-c}, \ldots w_{t+c} | w_t)$$

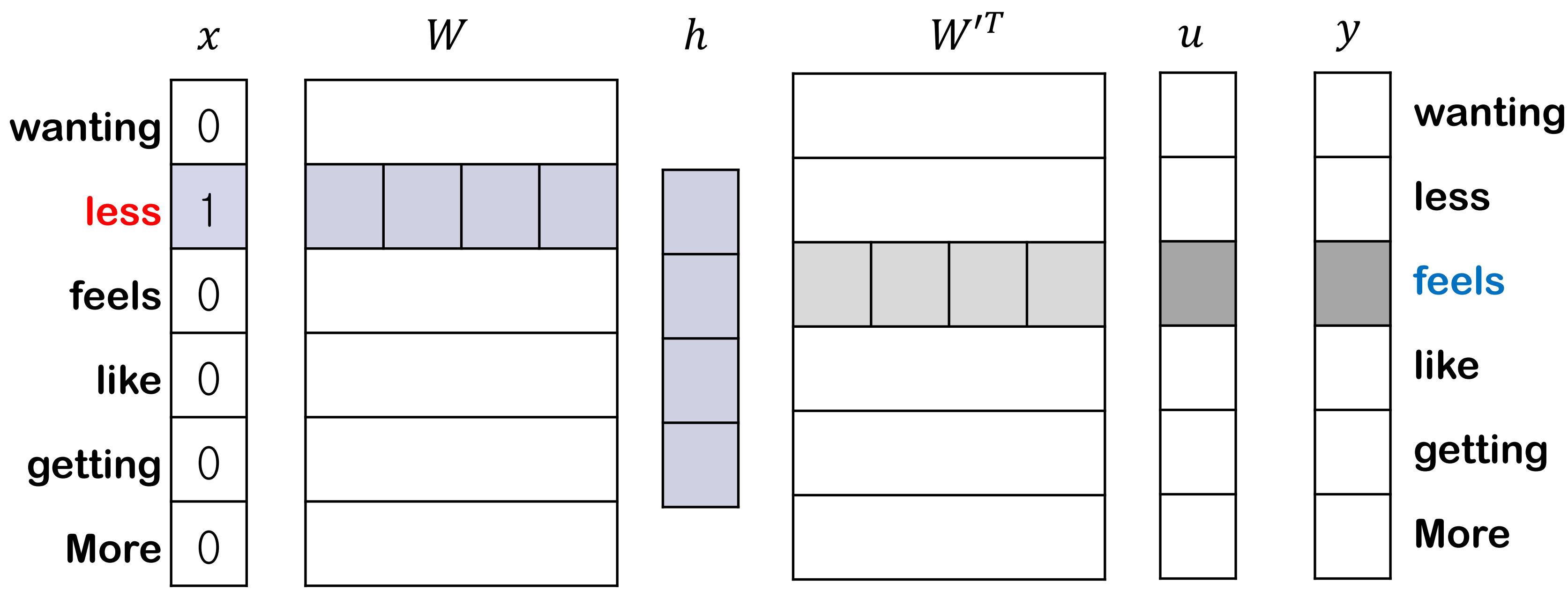
Word embedding

- Skip-gram model
  - Output represents the probability of being the context word, given the input center word



Word embedding

- Skip-gram model
  - $y_j = p(w_j | w_i)$ is the probability that w_j is the context word, given the input w_i



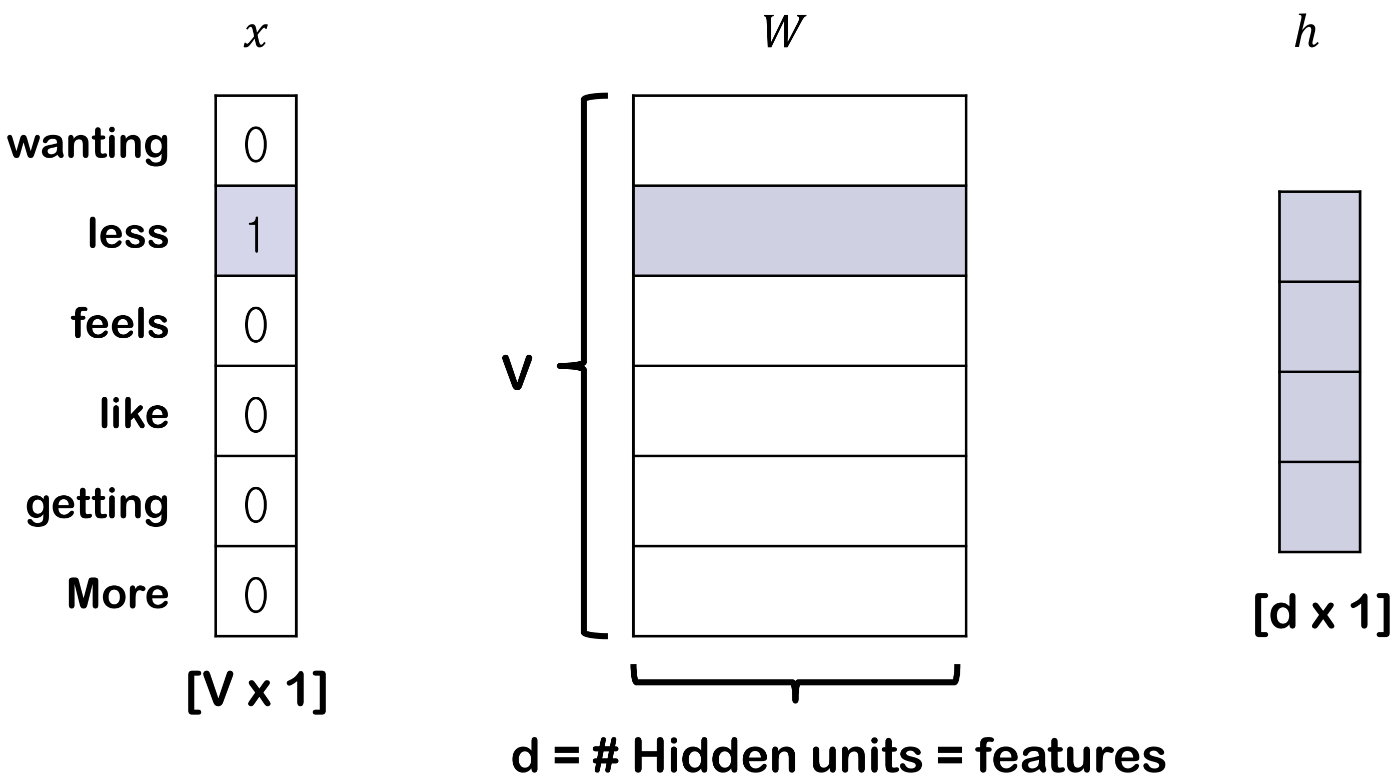
1 $h = W^T x$

2 $u = W'^T h$

3 $y_j = \dfrac{e^{u_j}}{\sum_{j'=1}^{V} e^{u_{j'}}}$

Word embedding

- Skip-gram model
  - Rows in hidden layer weight matrix become word vectors (word vector look-up table)



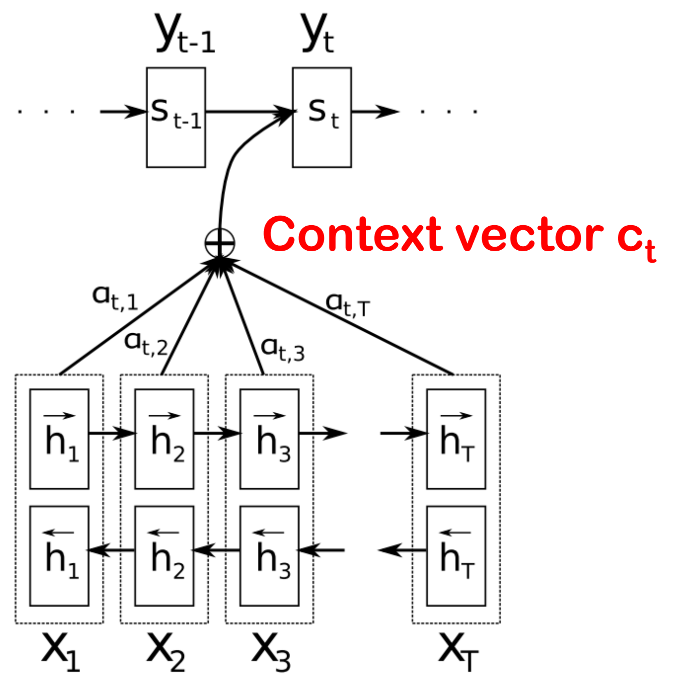
Review: Attention Mechanism

- **RNN hidden state of the decoder at i:** $s_i = f(s_{i-1}, y_{i-1}, c_i)$

- **The context vector c_i is computed as a weighted sum of annotations $(h_1, \dots, h_T)$:**

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j$$

- **How to get attention weight α_{ij}:**
  Alignment score function



Attention Mechanism - Scoring

- Alignment score function $e_{ij} = \mathrm{score}(s_{i-1}, h_j)$
  where s_{i-1} is the RNN hidden state just before emitting i th word, and h_j is the j th RNN hidden state of the input sentence.

- It scores how well the inputs around position j and the output at position i match.

- $$\mathrm{score}(s_{i-1}, h_j) = \begin{cases} s_{i-1}{}^{\top} h_j \\ s_{i-1}{}^{\top} W_a h_j \\ v_a{}^{\top} \tanh(W_a [s_{i-1}; h_j]) \end{cases}$$

  single hidden layer network

Attention Mechanism – Normalization

- Let α_{ij} be the probability that the target word y_i is aligned to (or translated from) a source word x_j.

- α_{ij} is computed by normalizing the probabilities with a softmax:

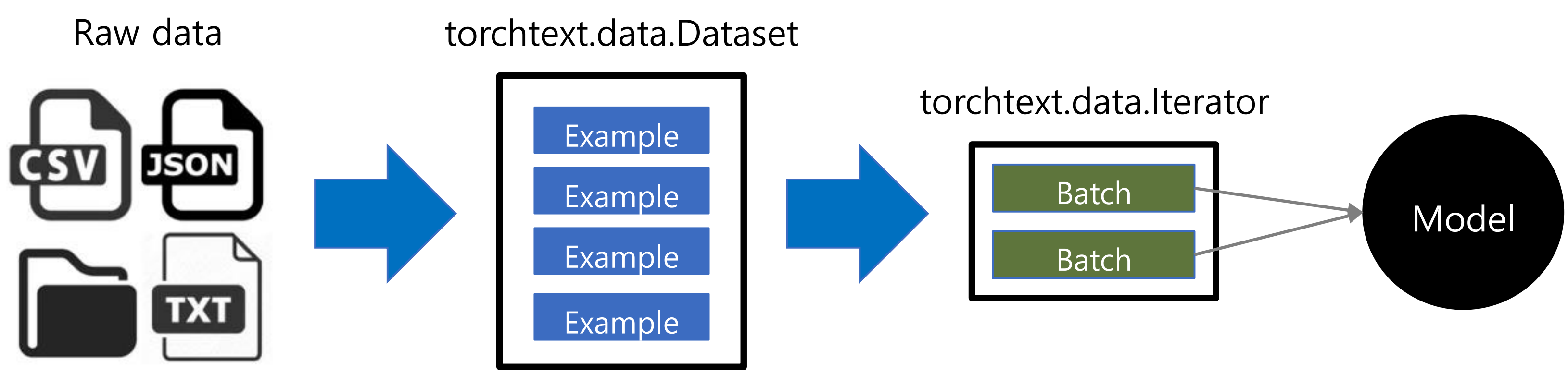
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

What we will do today

- Load data
  - Download dataset and load into Python
  - Divide train/val/test sets
- Preprocess data
  - Parse plain text into tokens (indices)
  - Build vocab and batching (with padding)
  - Use pretrained word embeddings
- Build model
  - Define RNN/LSTM model
  - Select hyperparameters
- Train model

Data Processing using Torchtext

- **TorchText**
  - **A PyTorch library providing utilities to process text and popular NLP datasets**



Handles:
- Reading data into memory
- Define a preprocessing pipeline
- Numericalizing
- Building a vocabulary object

Handles:
- Batching / Padding
- Moving data to the GPUs

Data Processing

- IMDB Movie Review Dataset
  - 50k movie reviews
  - Binary sentiment classification: positive/negative label

Review Text	Label
The movie has pointless story and worst music of all.	Negative
The movie is amazing because the fact that the romantic scenes are best.	Positive

- Simple way of loading from PyTorch (using torchtext library):

```
import torch
from torchtext import data, datasets

TEXT = data.Field(tokenize='spacy', include_lengths=True)
LABEL = data.LabelField(dtype=torch.float)

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
```

Field class is used for data processing, including tokenizer and numberzation. To check out the dataset, users need to first set up the TEXT/LABEL fields.

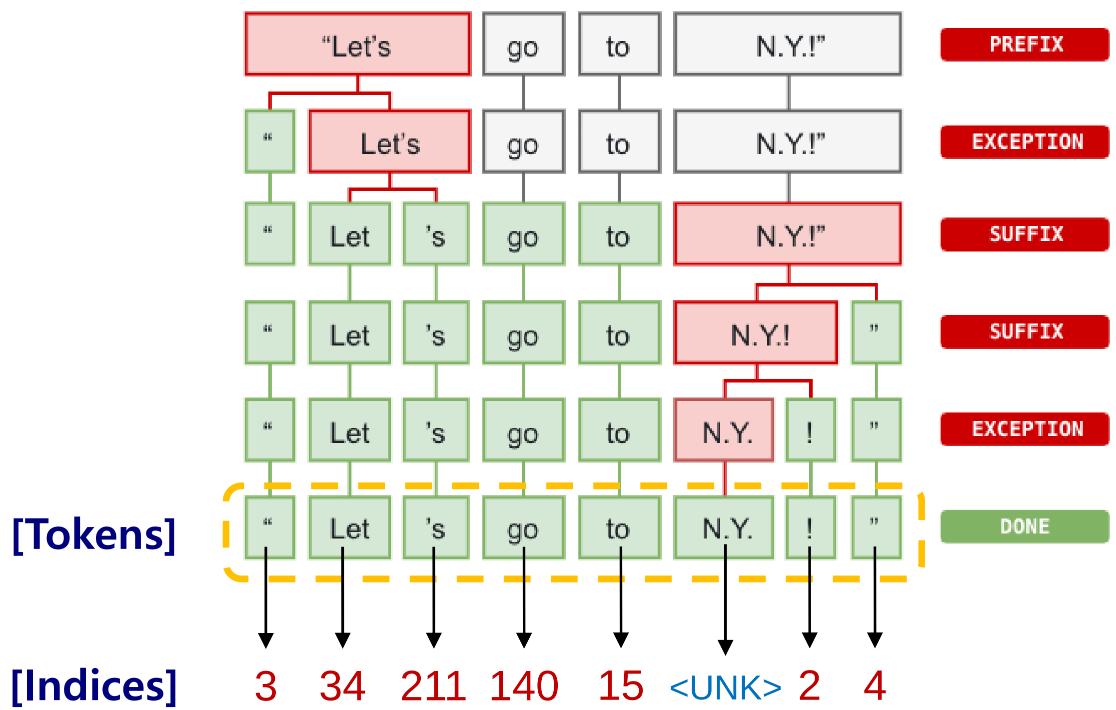
```
from torchtext.datasets import IMDB
train_iter, test_iter = IMDB(split=('train', 'test'))
```

```
next(train_iter)
```

- Returns the train/test dataset split directly without the preprocessing information
- Each split is an iterator which yields the raw texts and labels line-by-line.

Preprocessing data

- Tokenizer
  - Parse plain text into tokens
  - When vocabulary size is too big:
    -> Switch rare words with <UNK> token



Preprocessing data

- Batching and Padding

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

Batch size : [5, 4, 3, 3, 2, 1]

Padding

Building Vocabulary

- Users can build the vocabulary directly with the Vocab class
  - Min_freq

- Or we can use pre-trained word embeddings
  - GloVe (algorithm to calculate word vectors)
  - 6B: corpus of 6 billion words
  - Dim: dimensions of word vectors

    (50, 100, 200, 300)
    (42B, 840B – 300dim only)

Building Vocabulary

- Build Vocab
  - `From torchtext.vocab import GloVe, vocab`
  - **Vocab object matches the tokens with integer indices**
  - <unk> → **0** , <BOS> → **1,** <EOS> → **2,** <PAD> → **3**

```
from torchtext.vocab import GloVe, vocab
unk_index = 0
bos_index = 1
eos_index = 2
pad_index = 3

glove_vectors = GloVe(name='6B', dim=100)
glove_vocab = vocab(glove_vectors.stoi)
glove_vocab.insert_token("<unk>",unk_index)
glove_vocab.insert_token("<BOS>",bos_index)
glove_vocab.insert_token("<EOS>",eos_index)
glove_vocab.insert_token("<PAD>",pad_index)
glove_vocab.set_default_index(unk_index)

vocab = glove_vocab.get_stoi()
```

Add special tokens

<unk> as default_index

Word	Index	Embedding vector			
<unk>	0	0.5	2.1	1.9	1.5
<pad>	1	0.8	1.2	2.8	1.8
a	2	0.1	0.8	1.2	0.9
to	3	2.1	1.8	1.5	1.7
		…			
great	29,999	1.2	0.7	1.9	1.5

Build Basic RNN model

- Create subclass of torch.nn.Module

- Three components
  - 1. Learned word embedding
    - Convert word index to dense embedding vector
  - 2. RNN/LSTM module
    - Process word embedding sequentially to learn hidden state
  - 3. Fully connected layer
    - Nonlinear transformation of hidden state to prediction vector of dimension C, where C = # of classes

Reference: Modules

- torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, ...)
  - num_embeddings = size of the dictionary
  - embedding_dim = size of each embedding vector
  - padding_idx = padding index (initialized to zeros)

- input = [seq_len, batch_size]
- output = [seq_len, batch_size, embedding_dim]

Reference: Modules

- torch.nn.LSTM(input_size, hidden_size, num_layers, bidirectional, dropout, …)
  - input_size = the number of expected features in the input
  - hidden_size = the number of features in the hidden state
  - num_layers = number of recurrent layers (default=1)
  - dropout = if non-zero, introduces a Dropout layer on the outputs of each LSTM layer (except the last layer)
  - bidirectional = if true, becomes a bidirectional LSTM

- input = [seq_len, batch_size, input_size]

- output = [seq_len, batch_size, num_directions x hidden_size]

h_n = [num_layers x num_directions, batch_size, hidden_size)

c_n = [num_layers x num_directions, batch_size, hidden_size)

Reference: Modules

- torch.nn.Linear(in_features, out_features)
  - in_features = size of each input sample
  - out_features = size of each output sample

- input = [batch_size, *, in_features]
- output = [batch_size, *, out_features]

- torch.nn.Dropout(p)
  - p = probability of an element to be zeroed (default=0.5)

- input = any shape
- output = the same shape as input

Reference: Modules

- **torch.nn.BCEwithLogitsLoss()**

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right]$$

N = batch_size

- input(prediction) = [batch_size, *]
  target = [batch_size, *]

- output = [batch_size, *] (same shape as input)

Train model

- PyTorch training flow: for each batch
  - Reset gradient
    - `optimizer.zero_grad()`
  - Forward-pass
    - `predictions = model(batch.text).squeeze(1)`
    - `loss = criterion(predictions, batch.label)`
  - Backward-pass
    - `loss.backward()`
  - Backprop
    - `optimizer.step()`