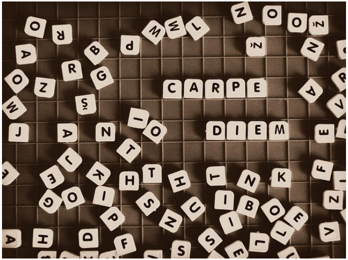
Deep Learning Basics

Revisiting words: Tokenization



Learning goals

- Understand the the process of text tokenization
- Learn the various types of text tokenization

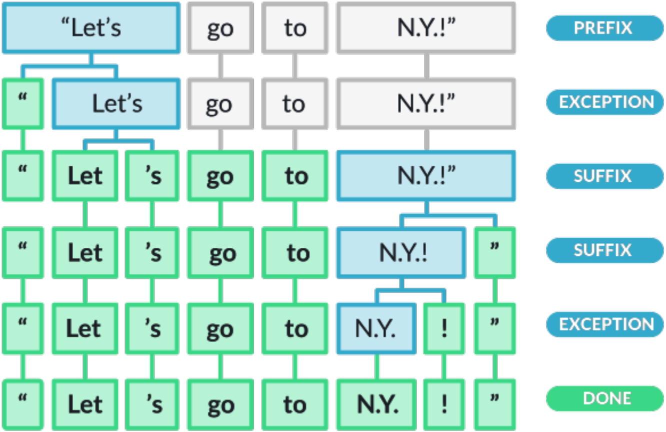
PROCESS OF TEXT TOKENIZATION

- Breaking text into smaller units called tokens
  - Tokens are discrete text units (letters, words, etc.)
  - They are the building blocks of natural language
- Encoding each token with unique IDs (numbers)
- Performed on the entire corpus of documents
  - Corpus vocabulary of unique tokens is obtained
- Mandatory preprocessing step for most of NLP tasks

WHY TOKENIZE?

- Computers must understand text
  - Text encoding is necessary
  - Encode small rather than large units
- Corpus documents can be large and hard to interpret
  - Working with tokens is easier
  - Building meaning in bottom-up fashion
- Text may contain extra whitespaces
  - Tokenization removes them

TOKENIZATION IN ACTION



Source: *spaCy*

TOKENIZATION TYPES

- Paragraph tokenization
  - Breaking doucments in paragraphs
  - Rarely used
- Sentence tokenization
  - Breaking text in sentences
- Word tokenization
  - Breaking text in words
  - The most common
- Subword tokenization
  - Breaking words in morphemes
- Character tokenization
  - Breaking text in individual characters
- Whitespace tokenization
  - Typical whitespaces: `" "`, `\t`, `\n`

WORD TOKENIZATION

- Most popular type of tokenization
  - Applied as preprocessing step in most NLP tasks
- Considers dictionary words and several delimiters
  - Accuracy depends on dictionary used for training
  - Tradeoff between accuracy and efficiency
- Whitespaces and punctuation symbols are used
  - They determine word boundaries
- Available in many NLP libraries

Example:

What is the tallest building? => 'What', 'is', 'the', 'tallest', 'building', '?'

SUBWORD TOKENIZATION

- Finer grained than word tokenization
  - Breaks text into words
  - Breaks words into smaller units (root, prefix, suffix, etc.)
  - Uses more complex linguistic rules
- More important for highly flective languages
  - Words have many forms
  - Prefixes and suffixes are added
  - Word meaning and function changes
- Helps to disambiguate meaning
- Helps to reduce out of vocabulary words

Example:

What is the tallest building? => ‘What’, ‘is’, ‘the’, ‘tall’, ‘est’, ‘build’, ‘ing’, ‘?’

CHARACTER TOKENIZATION

- Creates smaller vocabulary
  - Same as the number of lettters
- Helps with out of vocabulary words
  - Retains their character composition
- More complex process
  - The output becomes 5 or 6 times bigger

Example:

What is the tallest building? => ‘W’, ‘h’, ‘a’, ‘t’, ‘i’, ‘s’, ‘t’, ‘h’, ‘e’, ‘t’, ‘a’, ‘l’, ‘l’, ‘e’, ‘s’, ‘t’, ‘b’, ‘u’, ‘i’, ‘l’, ‘d’, ‘i’, ‘n’, ‘g’, ‘?’

REMEMBER FASTTEXT?

Assume, we want to represent the word *example*:

- Character n-grams (n = 3):

  <ex, exa, xam, amp, mpl, ple, le>, <example>

- In practice, we don't set $n = a$ but rather $a \leq n \leq b$
- Character n-grams ($2 \leq n \leq 4$):

  <e, ex, xa, am, mp, pl, le, e>,
  <ex, exa, xam, amp, mpl, ple, le>,
  <exa, exam, xamp, ampl, mple, ple>,
  <example>

- Note, that the 4-gram *exam* is different from the word $<$exam$>$.

BYTEPAIR ENCODING (BPE)

Data compression algorithm ▸ Gage (1994)

- Considering data on a *byte*-level
- Looking at pairs of bytes:
  1. Count the occurrences of all byte pairs
  2. Find the most frequent byte pair
  3. Replace it with an unused byte
- Repeat this process until no further compression is possible

BYTEPAIR ENCODING (BPE)

Open-vocabulary neural machine translation Sennrich et al. (2016)

- Instead of looking at bytes, look at characters
- Motivation: Translation as an open-vocabulary problem
- Word-level NMT models:
  - Handling out-of-vocabulary word by using back-off dictionaries
  - Unable to translate or generate previously unseen words
- Using BPE effectively *solves* this problem, except for ..
  - .. the occurence of unknown characters
  - .. when all occurences in the training set were merged into "larger" symbols (Example: *"safeguar"* and *"safeguard"*)

BYTEPAIR ENCODING (BPE)

Adapt BPE for word segmentation Sennrich et al. (2016)

- *Goal:* Represent an open vocabulary by a vocabulary of fixed size
  $\rightarrow$ Use variable-length character sequences
- Looking at pairs of characters:
  1. Initialize the the vocabulary with all characters plus end-of-word token
  2. Count occurrences and find the most frequent character pair, e.g. "A" and "B" (⚠ Word boundaries are **not** crossed)
     [Side effect: Can be run on a dictionary w/ frequency counts]
  3. Replace it with the new token "AB"
- Only one hyperparameter: Vocabulary size
  (Initial vocabulary + Specified no. of merge operations)
  $\rightarrow$ Repeat this process until given $|V|$ is reached

EXAMPLE – SETUP

```python
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out
```

EXAMPLE – MERGING

```
vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}

pairs = get_stats(vocab)
```

```
>>> print(pairs)
defaultdict(<class 'int'>, {
            ('l', 'o'): 7, ('o', 'w'): 7, ('w', '</w>'): 5,
            ('w', 'e'): 8, ('e', 'r'): 2, ('r', '</w>'): 2,
            ('n', 'e'): 6, ('e', 'w'): 6, ('e', 's'): 9,
            ('s', 't'): 9, ('t', '</w>'): 9, ('w', 'i'): 3,
            ('i', 'd'): 3, ('d', 'e'): 3
})
```

```
best = max(pairs, key=pairs.get)
vocab = merge_vocab(best, vocab)
```

```
>>> print(best)
('e', 's')
>>> print(vocab)
{'l o w </w>': 5, 'l o w e r </w>': 2,
 'n e w es t </w>': 6, 'w i d es t </w>': 3}
```

EXAMPLE – MERGING

```python
vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}

num_merges = 10

for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

```
('e', 's')
('es', 't')
('est', '</w>')
('l', 'o')
('lo', 'w')
('n', 'e')
('ne', 'w')
('new', 'est</w>')
('low', '</w>')
('w', 'i')
```

WORDPIECE

Voice Search for Japanese and Korean ‣ Schuster & Nakajima (2012)

- *Specific Problems:*
  - Asian languages have larger basic character inventories compared to Western languages
  - Concept of spaces between words does (partly) not exist
  - Many different pronounciations for each character

WORDPIECE

- *WordPieceModel:* Data-dependent + do not produce OOVs
  1. Initialize the the vocabulary with basic Unicode characters (22k for Japanese, 11k for Korean)
     ⚠ Spaces are indicated by an underscore attached before (of after) the respective basic unit or word (increases initial $|V|$ by up to factor 4)
  2. Build a language model using this vocabulary
  3. Merge word units that increase the likelihood on the training data the most, when added to the model
- Two possible stopping criteria:
  Vocabulary size *or* incremental increase of the likelihood

WORDPIECE

Use for neural machine translation Wu et al. (2016)

- *Adaptions:*
  - Application to Western languages leads to a lower number of basic units ($\sim$ 500)
  - Add space markers (underscores) *only* at the beginning of words
  - Final vocabulary sizes between 8k and 32k yield a good balance between accuracy and fast decoding speed (compared to around 200k from Schuster & Nakajima (2012))

Independent **vs.** *joint* **encodings for source & target language**

- Sennrich et al. (2016) report better results for joint BPE
- Wu et al. (2016) use shared WordPieceModel to guarantee identical segmentation in source & target language in order to facilitate copying rare entity names or numbers