Tokenization

From bits to bytes to tokens

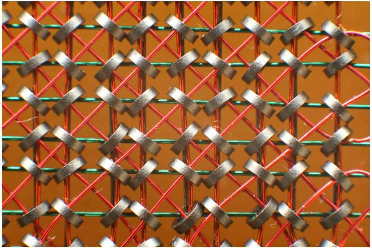
University of Maryland

Brief recap on how computers store stuff: bits

Setun

- Computers (except for some Soviet ones) are binary

Brief recap on how computers store stuff: bits



- Computers (except for some Soviet ones) are binary
- Electronic memory is either zero or one (bit)

Brief recap on how computers store stuff: bits

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

- Computers (except for some Soviet ones) are binary
- Electronic memory is either zero or one (bit)
- Those bits express numbers as sums of powers of two

$$\text{Value}(b_0 b_1 \ldots b_n) = \sum_{i=0}^{n} b_i \, 2^i \quad (1)$$

Brief recap on how computers store stuff: bits

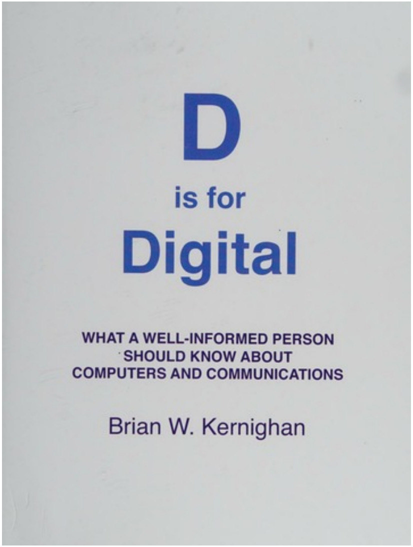
Binary	Decimal	Hex
0000	0	`0x0`
0001	1	`0x1`
0010	2	`0x2`
0011	3	`0x3`
0100	4	`0x4`
0101	5	`0x5`
0110	6	`0x6`
0111	7	`0x7`
1000	8	`0x8`
1001	9	`0x9`
1010	10	`0xA`
1011	11	`0xB`
1100	12	`0xC`
1101	13	`0xD`
1110	14	`0xE`
1111	15	`0xF`

- Computers (except for some Soviet ones) are binary
- Electronic memory is either zero or one (bit)
- Those bits express numbers as sums of powers of two

$$\text{Value}(b_0 b_1 \ldots b_n) = \sum_{i=0}^{n} b_i 2^i \tag{1}$$

- We often use hexadecimal notation because it's easier to convert to bit strings and binary is cumbersome

Brief recap on how computers store stuff: bits



- Computers (except for some Soviet ones) are binary
- Electronic memory is either zero or one (bit)
- Those bits express numbers as sums of powers of two

$$\text{Value}(b_0 b_1 \ldots b_n) = \sum_{i=0}^{n} b_i 2^i \tag{1}$$

- We often use hexadecimal notation because it's easier to convert to bit strings and binary is cumbersome

Bits to Characters

- Given a byte, how do you know what character to show on the screen
- If byte starts with 0, then ASCII character (e.g., 0x41 = "A")

Bits to Characters

- Given a byte, how do you know what character to show on the screen
- If byte starts with 0, then ASCII character (e.g., 0x41 = “A”)
- If byte starts with 1, then UTF-8 multicharacter
  - 110 for two bytes
  - 1110 for three
  - 11110 for four
  - e.g., E9 A9 AC starts with 1110, three bytes

Bits to Characters

- Given a byte, how do you know what character to show on the screen
- If byte starts with 0, then ASCII character (e.g., 0x41 = “A”)
- If byte starts with 1, then UTF-8 multicharacter
  - 110 for two bytes
  - 1110 for three
  - 11110 for four
  - e.g., E9 A9 AC starts with 1110, three bytes
- There are other encodings (e.g., UTF-16, UTF-32, GB2312)
- UTF-n means how many minimum bits needed for character
- Because ASCII is subset, UTF-8 is very efficient for Latin-script pages

Unicode is a Rabit Hole

Strings Need to Become Integers

- More on the details soon
- Sparse representations: a “word” becomes an integer

Maryland

$\downarrow$

[0, 0, … 1, … 0, 0]

Strings Need to Become Integers

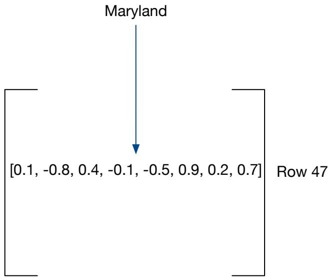
- More on the details soon
- Sparse representations: a "word" becomes an integer
- Dense representations: a "word" becomes a vector

Maryland

[0.1, -0.8, 0.4, -0.1, -0.5, 0.9, 0.2, 0.7]

Strings Need to Become Integers

- More on the details soon
- Sparse representations: a “word” becomes an integer
- But the dense representation is a lookup, that lookup is itself an integer!



Understanding Bytes in Text Tokenization

- Text is converted into byte arrays
- Each byte is an integer
- Example Python to convert text to bytes

```
text = "This is some text"
byte_ary = bytearray(text, "utf-8")
print(byte_ary)
```

- List conversion shows integer byte values for each character.

Understanding Bytes in Text Tokenization

- Text is converted into byte arrays
- Each byte is an integer
- Example Python to convert text to bytes

  ```
  text = "This is some text"
  byte_ary = bytearray(text, "utf-8")
  print(byte_ary)
  ```

- List conversion shows integer byte values for each character.
- Output: `[84, 104, 105, 115, 32, 105, ...]`

Understanding Bytes in Text Tokenization

- Text is converted into byte arrays
- Each byte is an integer
- Example Python to convert text to bytes
```
text = "This is some text"
byte_ary = bytearray(text, "utf-8")
print(byte_ary)
```
- List conversion shows integer byte values for each character.
- Output: `[84, 104, 105, 115, 32, 105, ...]`
- We want to have integers represent “words”, not characters

Why not use whitespace?

- Convert bytearray to list of integers for token IDs.

```
punctuation = "Listen, when 'writing', we puncuate!"
tokens = punctuation.split()
```

Why not use whitespace?

- Convert bytearray to list of integers for token IDs.
```
punctuation = "Listen, when 'writing', we puncuate!"
tokens = punctuation.split()
```
- Problem: Punctuation gets lumped into tokens
- `['Listen,', 'when', "'writing',", 'we', 'puncuate!']`

Why not use simple regexp?

- Define regexp to handle all characters together

```
import re
contraction = "We can't ignore contractions, right?"
re.split(r"(\w+|#\d|\?|!)", contraction)
```

Why not use simple regexp?

- Define regexp to handle all characters together

```
import re
contraction = "We can't ignore contractions, right?"
re.split(r"(\w+|#\d|\?|!)", contraction)
```

- Problem: Doesn't deal with contractions
- `['', 'We', ' ', 'can', "'", 't', ' ', 'ignore', ' ', 'contractions', ', ', 'right', '', '?', '']`
- Also, things like C++ and "Mr." cause problems

Why not complex regexp?

- Later, we'll talk about the Penn Treebank
- Syntactic parses also need to turn trees into non-terminals
- Very complicated regexp that handles titles, money, contractions

```
from nltk.tokenize.treebank import TreebankWordTokeniz
s = '''We cannot buy my boss, Dr. Zwicker, the cheap
       muffins that cost $2.50, okay?  Thanks!'''
t = TreebankWordTokenizer()
toks = t.tokenize(s)
```

Why not complex regexp?

- Later, we'll talk about the Penn Treebank
- Syntactic parses also need to turn trees into non-terminals
- Very complicated regexp that handles titles, money, contractions

```
from nltk.tokenize.treebank import TreebankWordTokeniz
s = '''We cannot buy my boss, Dr. Zwicker, the cheap
       muffins that cost $2.50, okay?  Thanks!'''
t = TreebankWordTokenizer()
toks = t.tokenize(s)
```

- Problem: Doesn't deal with contractions
- `['We', 'can', 'not', 'buy', 'my', 'boss', ',', 'Dr.', 'Zwicker', ',', 'the', 'cheap', 'muffins', 'that', 'cost', '$', '2.50', ',', 'okay', '?', 'Thanks', '!']`

What about new words?

Algorithm for Learning Mapping

1. Read through dataset, extract all words
2. Take the N most frequent, order them as $\mathscr{V}$
3. This becomes your vocabulary

Algorithm for Test-Time Mapping

1. Given token, see if it's in $\mathscr{V}$
2. If so, call it the index of token in $\mathscr{V}$
3. If not, then call it OOV

- Don't put <u>everything</u> into vocabulary
- Need to learn how to generalize
- Waste of memory

What about new words?

Algorithm for Learning Mapping

1. Read through dataset, extract all words
2. Take the N most frequent, order them as $\mathscr{V}$
3. This becomes your vocabulary

Algorithm for Test-Time Mapping

1. Given token, see if it's in $\mathscr{V}$
2. If so, call it the index of token in $\mathscr{V}$
3. If not, then call it OOV

- Don't put <u>everything</u> into vocabulary
- Need to learn how to generalize
- Waste of memory
- Subwords have structure
- Generative models need flexibility