COMP 3361 Natural Language Processing

Lecture 6: Neural language models: Overview, tokenization

Spring 2024

Many materials from CSE447@UW and COS484@Princeton with special thanks!

Announcements

- TA will host an online virtual office hour @4-5 pm today!
  - Mainly about the assignment 1
  - Book a slot via the link on Slack
  - Also you can always ask questions on Slack!
- 新年快樂！ Happy Chinese New Year.
  - We will record a tutorial on PyTorch for the next Friday’s lecture and upload it to the course website.
  - So you don't need to attend in person on Feb 9
  - Unless most of you want to have class on the morning of New Year's Eve! :)

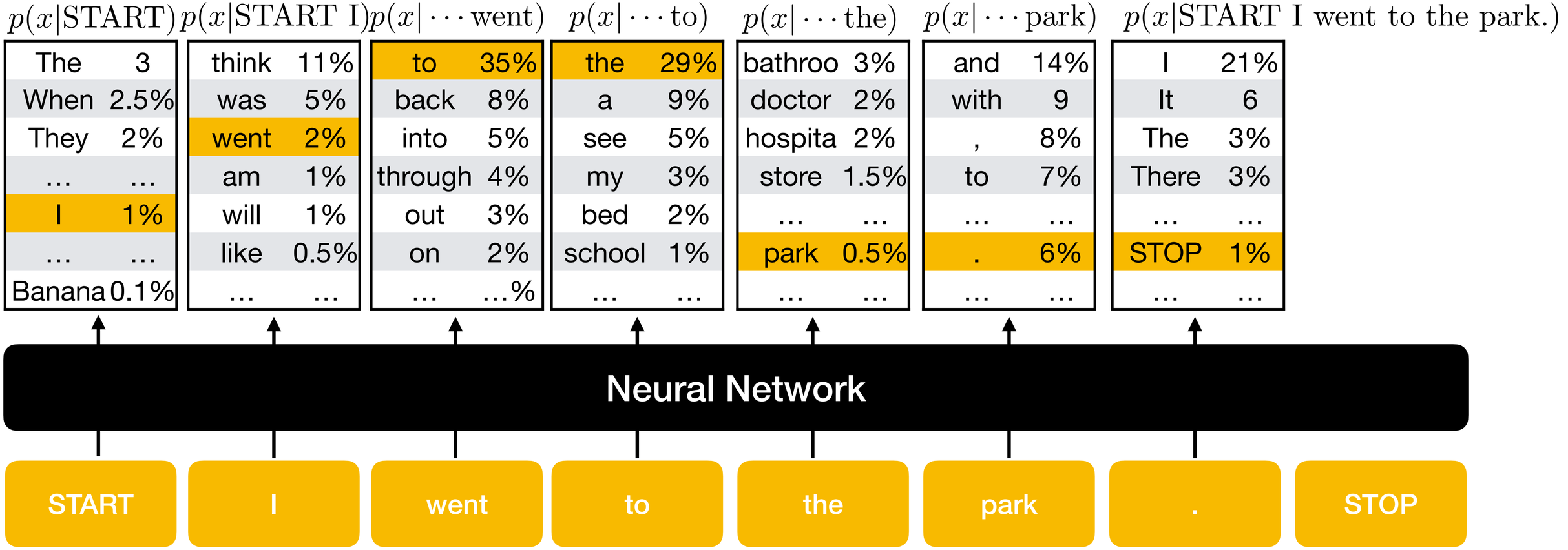
Lecture plan

- Neural language models: overview
- Running examples of neural language models
- Byte-pair encoding (BPE) tokenization
- Other tokenization variants

Neural language models: overview

Neural language models: inputs/outputs

- **Input:** sequences of words (or tokens)
- **Output:** probability distribution over the next word (token)



Neural language models

But neural networks take in real-valued vectors, not words…

- Use one-hot or learned embeddings to map from words to vectors!
  - Learned embeddings become part of parameters θ

Neural networks output vectors, not probability distributions…

- Apply the softmax to the outputs!
- What should the size of our output distribution be?
  - Same size as our vocabulary $|\mathcal{V}|$

Example: BERT for sentiment classification

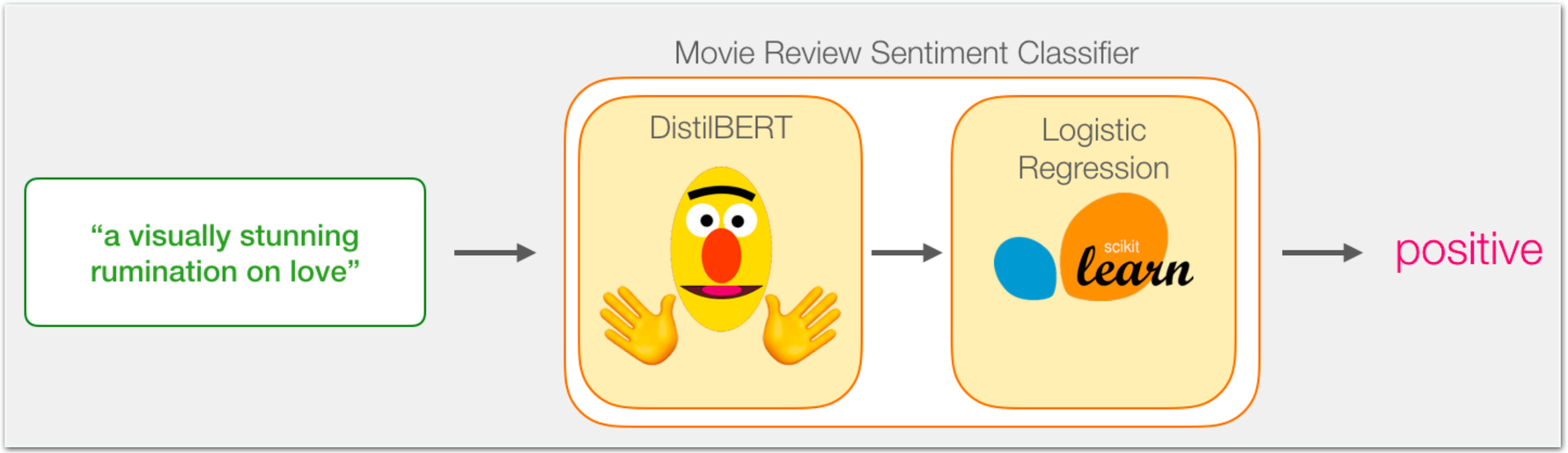
Task:

Data:

sentence	label
a stirring , funny and finally transporting re imagining of beauty and the beast and 1930s horror films	1
apparently reassembled from the cutting room floor of any given daytime soap	0
they presume their audience won't sit still for a sociology lesson	0
this is a visually stunning rumination on love , memory , history and the war between art and commerce	1
jonathan parker 's bartleby should have been the be all end all of the modern office anomie films	1

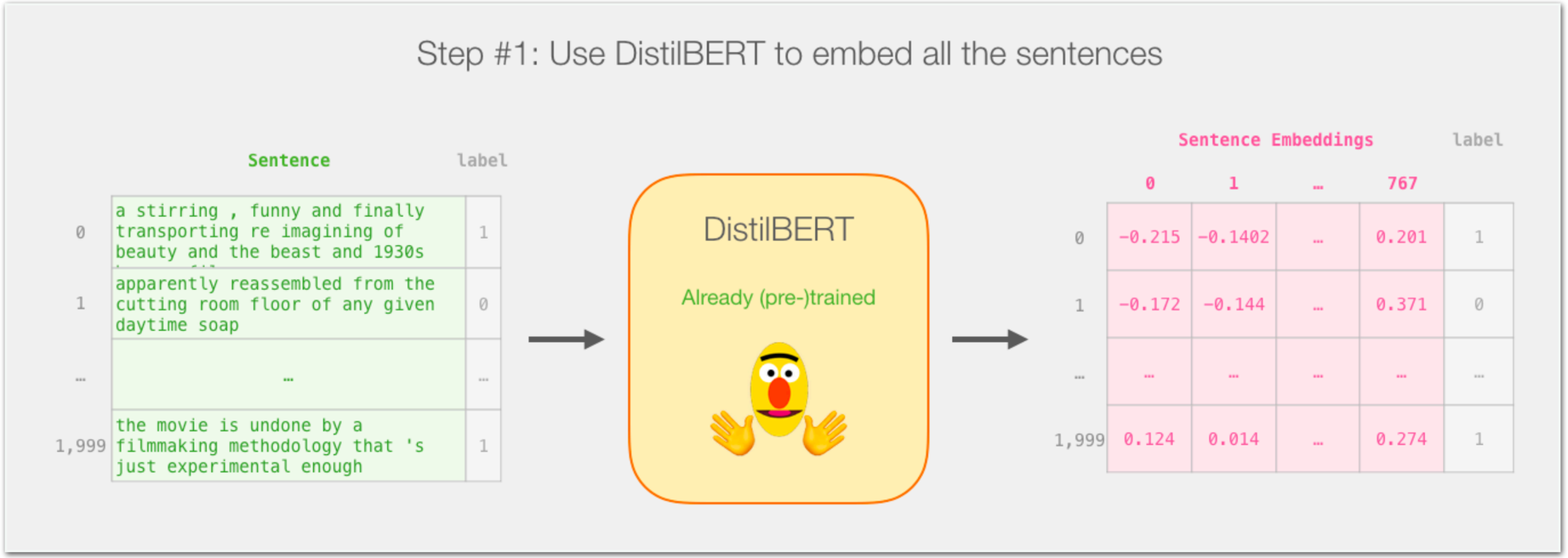
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

Example: BERT for sentiment classification



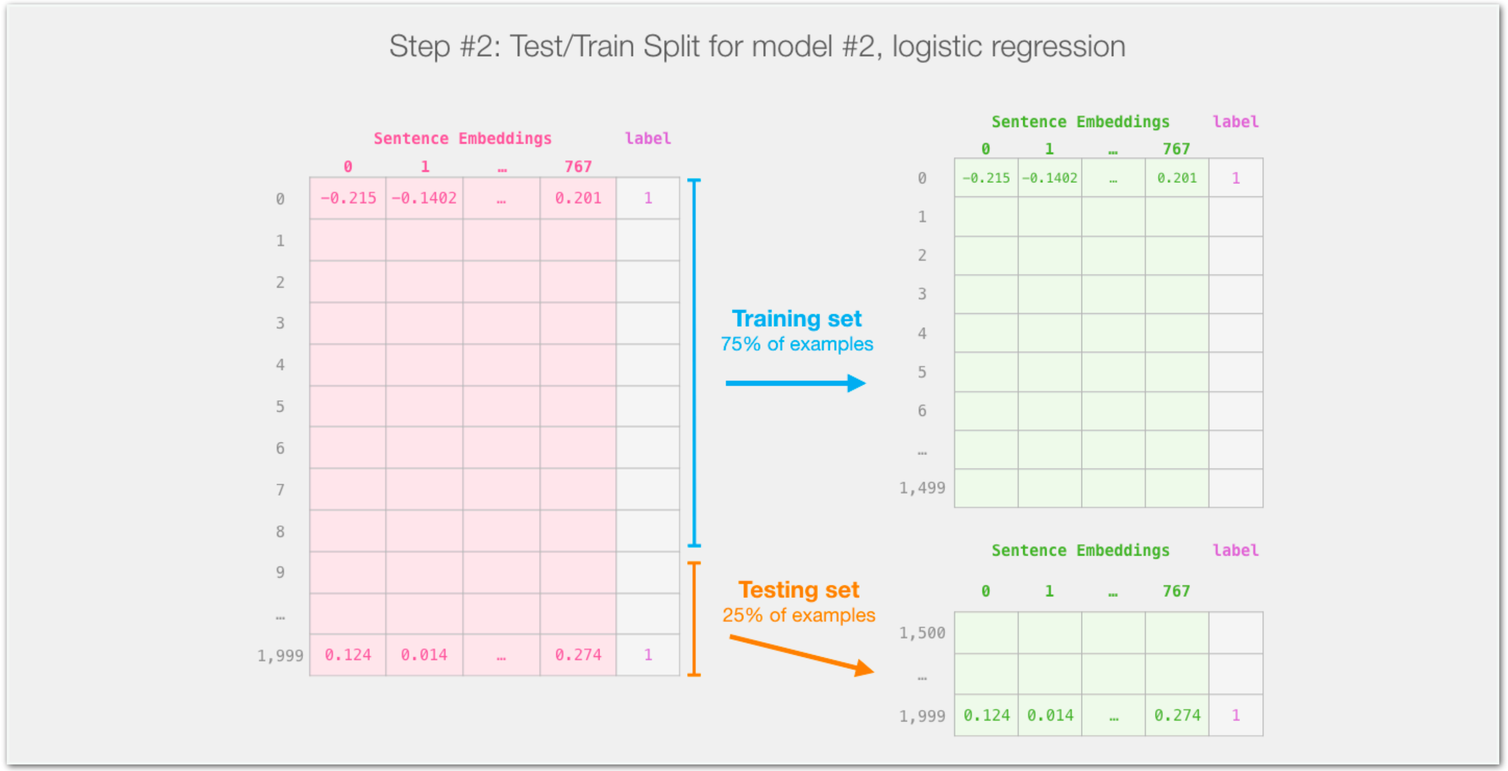
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

BERT for sentiment classification: overview



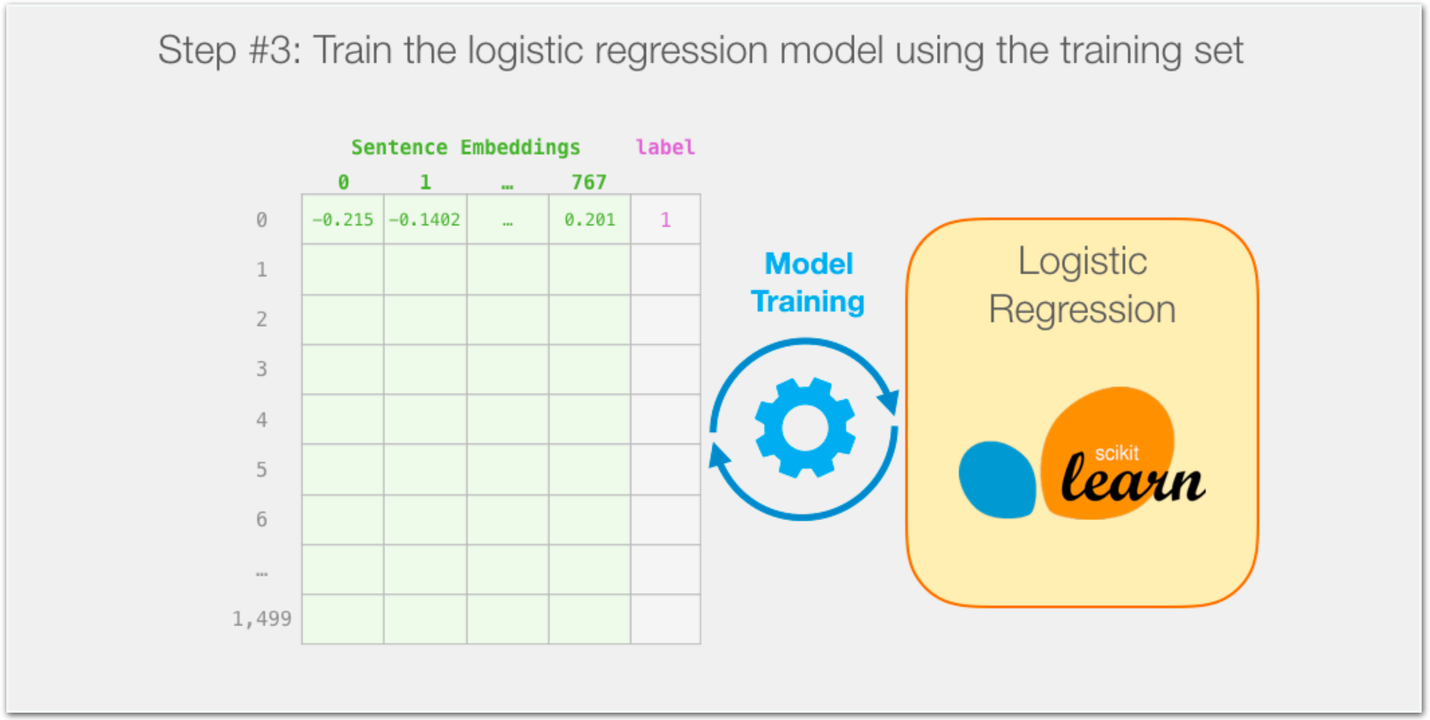
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

BERT for sentiment classification: overview



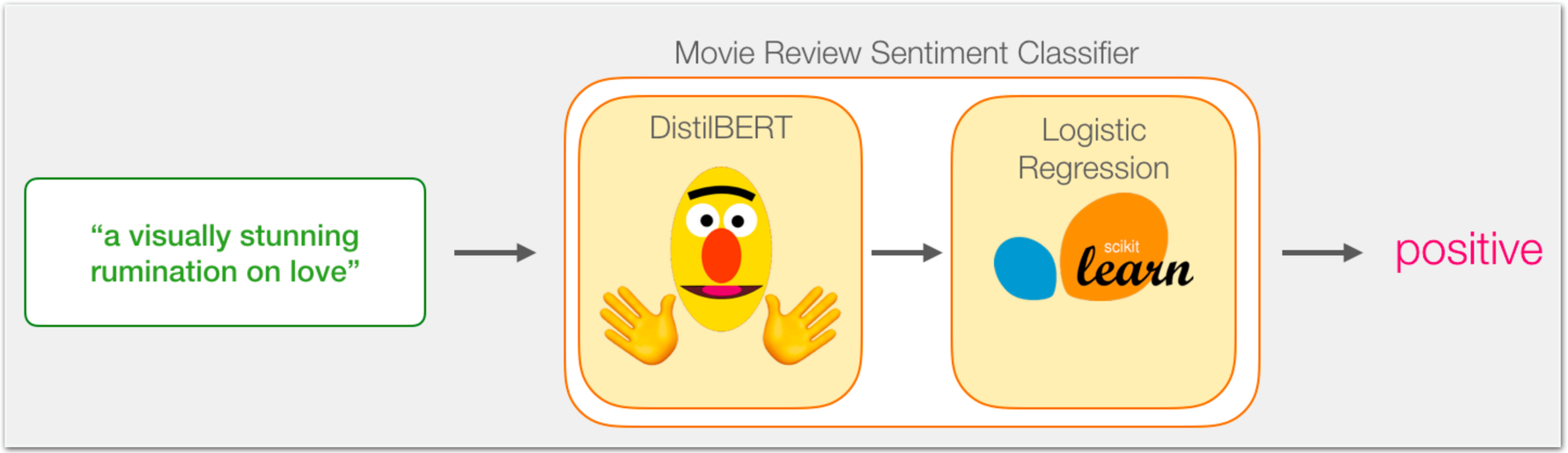
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

BERT for sentiment classification: overview



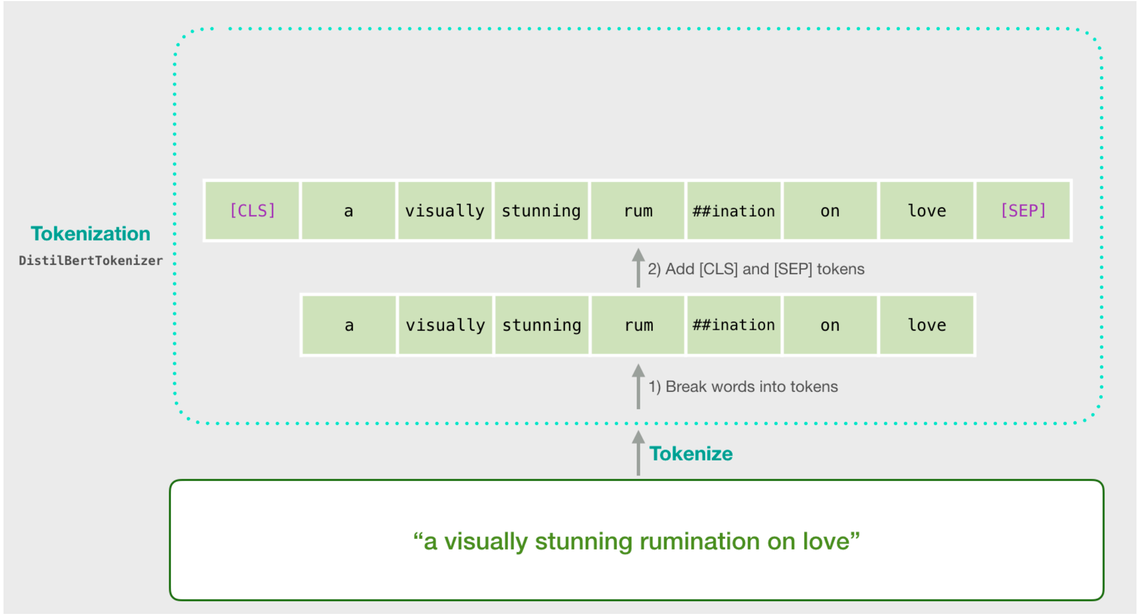
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

BERT for sentiment classification: prediction



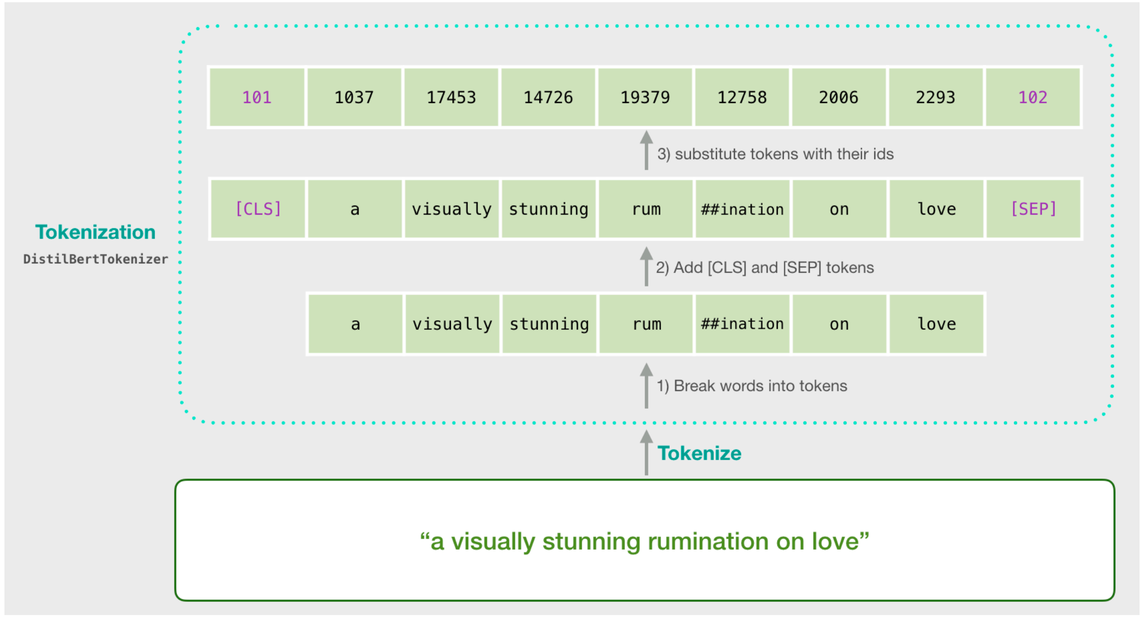
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

Prediction step 1: tokenization

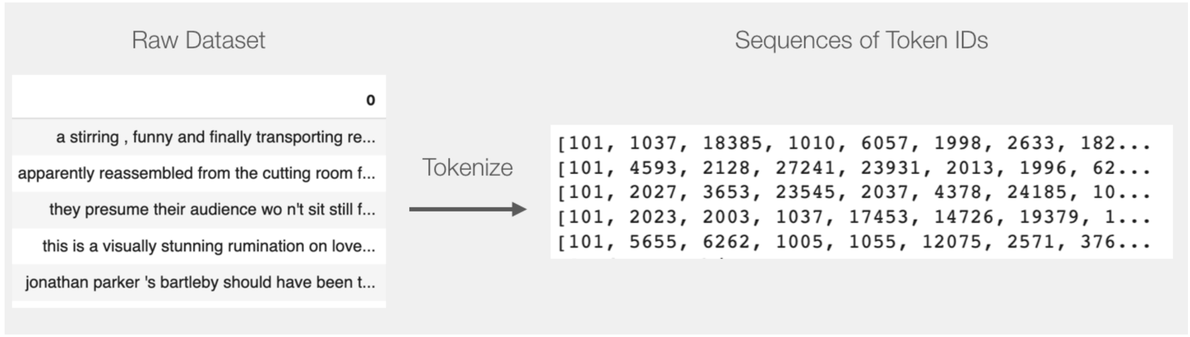


https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Prediction step 1: tokenization




```
tokenized = df[0].apply((lambda x: tokenizer.encode(x, add_special_tokens=True)))
```

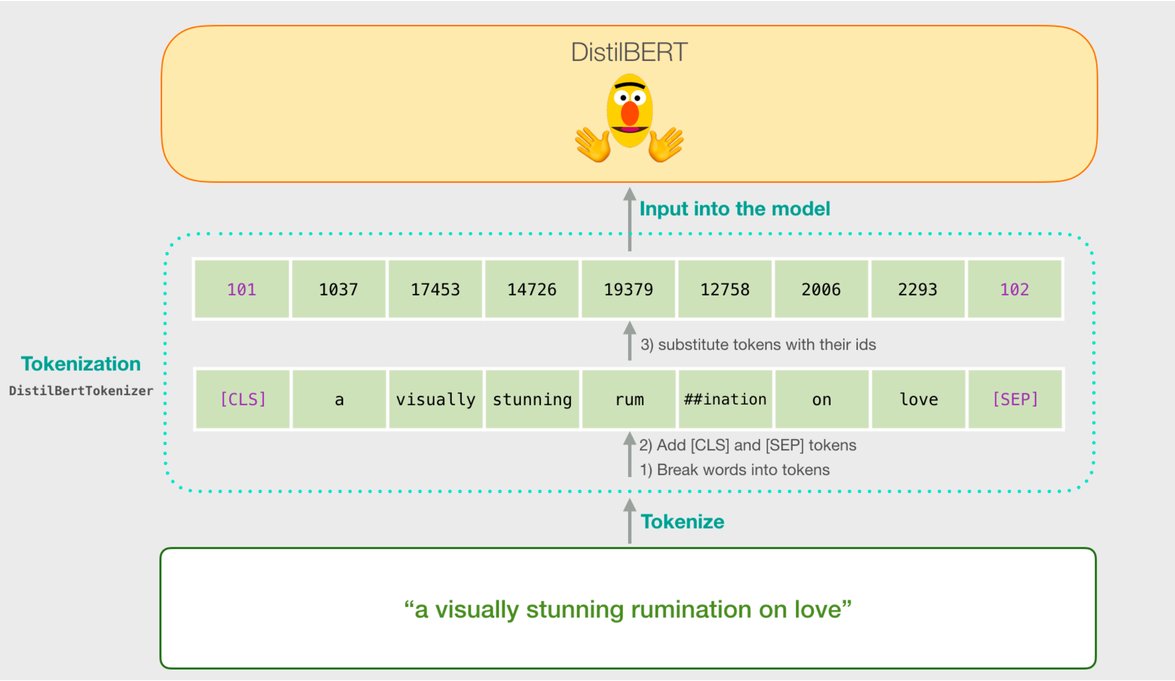


BERT/DistilBERT Input Tensor

		Tokens in each sequence			
		0	1	…	66
Input sequences (reviews)	0	101	1037	…	0
	1	101	2027	…	0
	…	…	…	…	
	1,999	101	1996	…	0

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Prediction step 2: input into BERT




```python
import numpy as np
import pandas as pd
import torch
import transformers as ppb # pytorch transformers
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
```

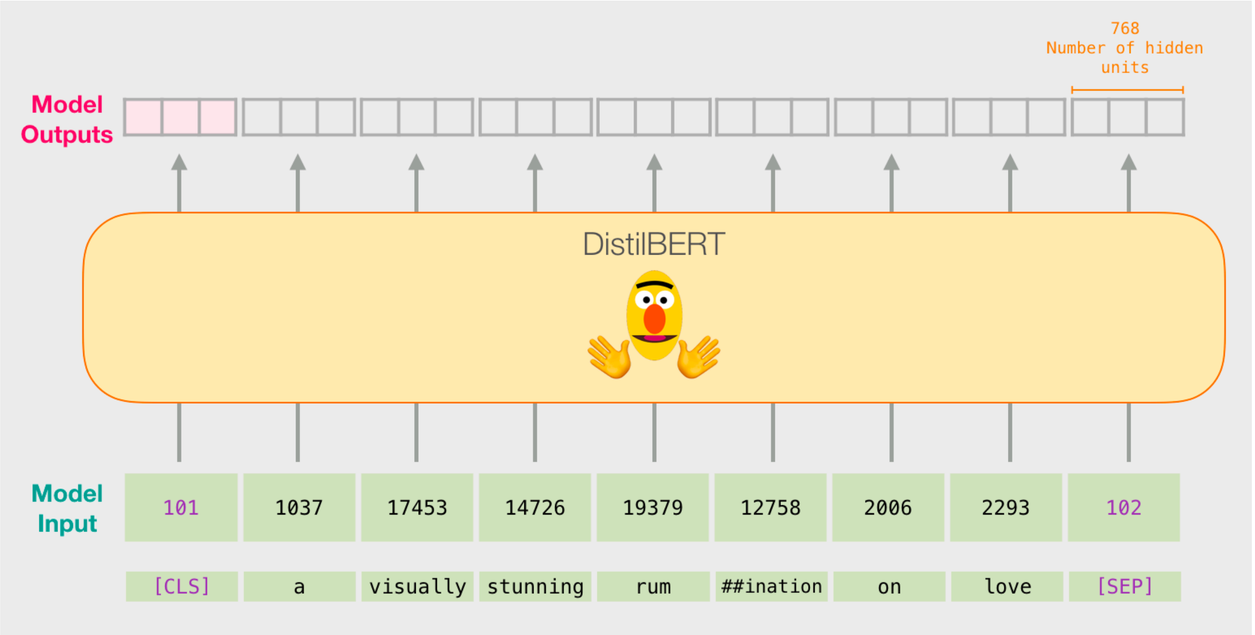
```python
model_class, tokenizer_class, pretrained_weights = (ppb.DistilBertModel, ppb.DistilBertTokenizer, 'distilbert-base-uncased')

## Want BERT instead of distilBERT? Uncomment the following line:
#model_class, tokenizer_class, pretrained_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')

# Load pretrained model/tokenizer
tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
model = model_class.from_pretrained(pretrained_weights)
```

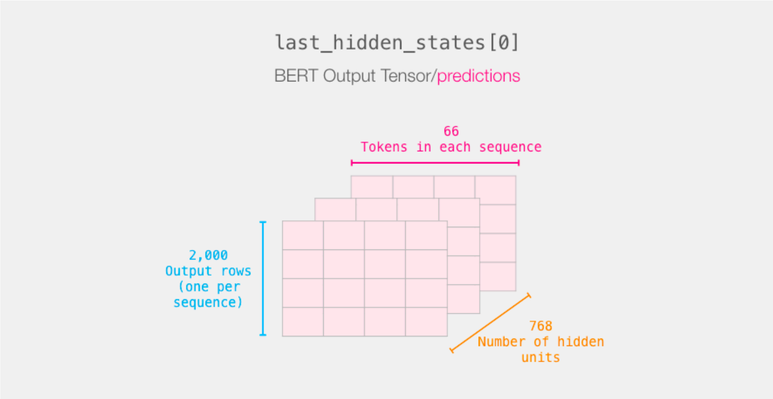
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Prediction step 3: run BERT to get outputs



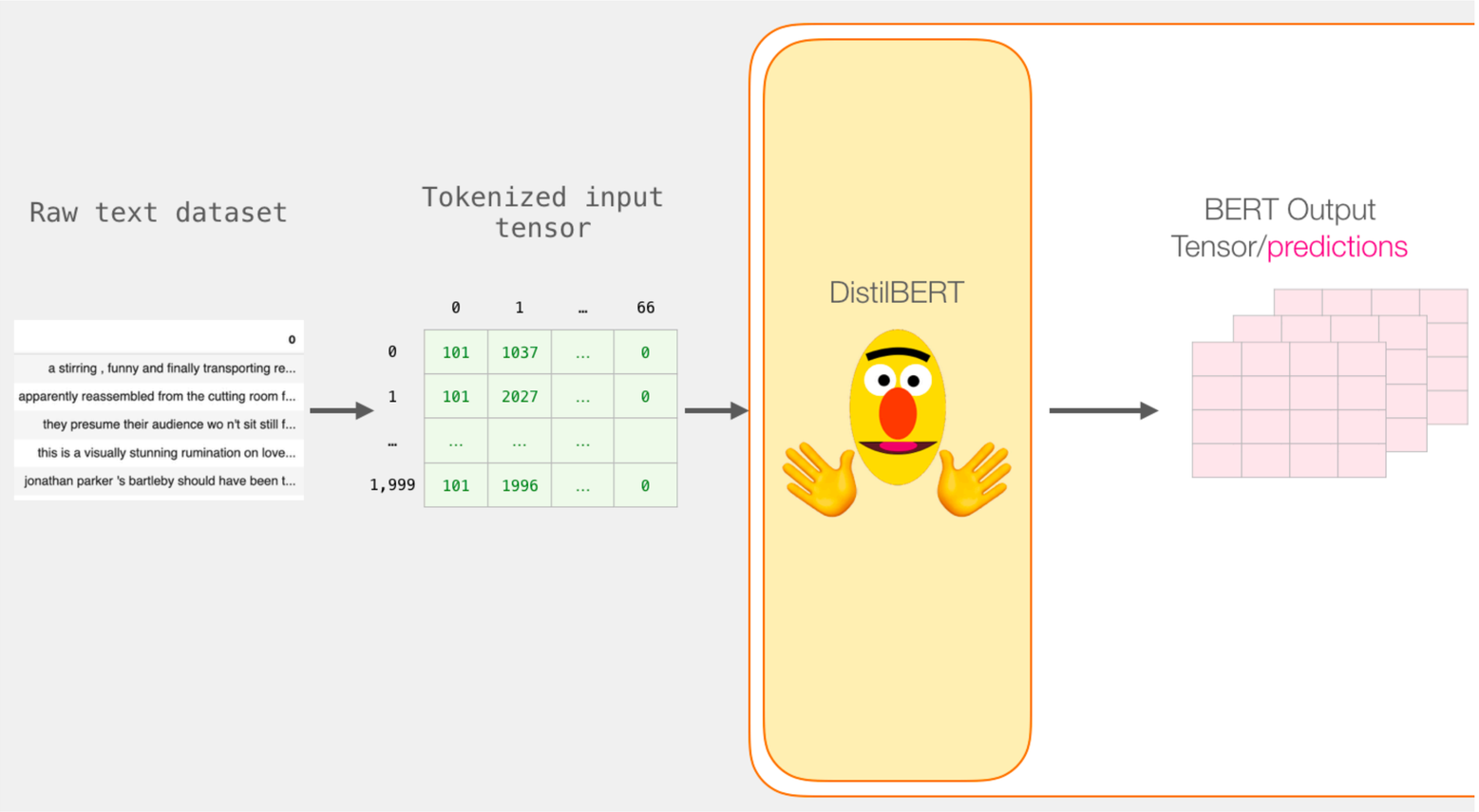

```
input_ids = torch.tensor(np.array(padded))

with torch.no_grad():
    last_hidden_states = model(input_ids)
```



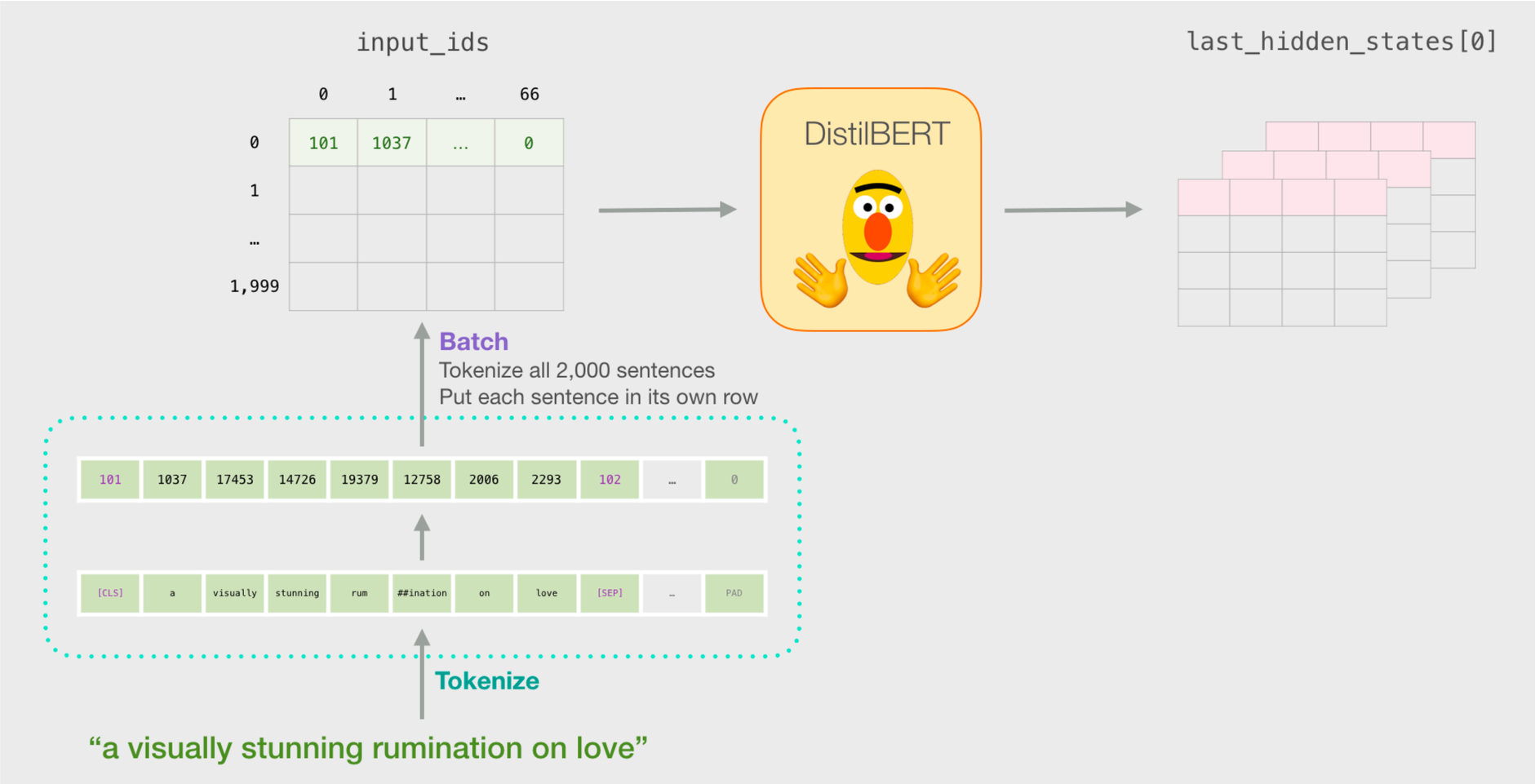
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Example overview so far



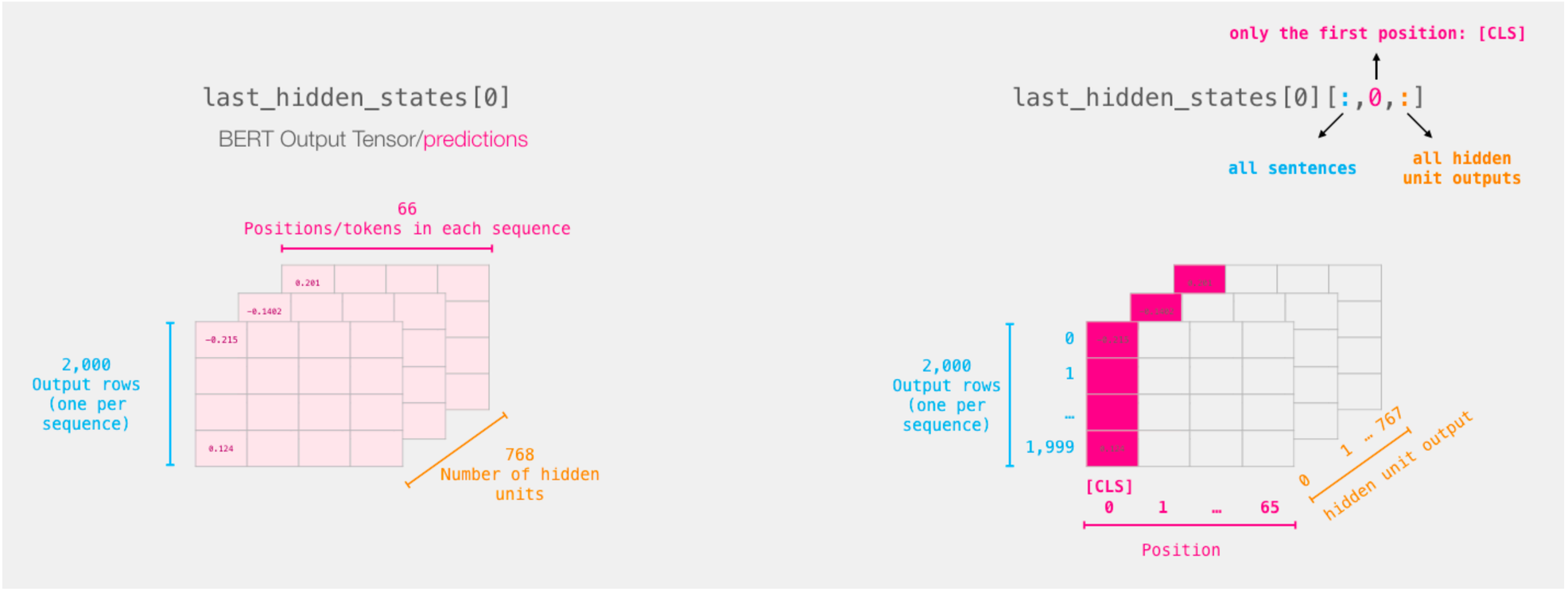
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Recapping a sentence's journey



https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

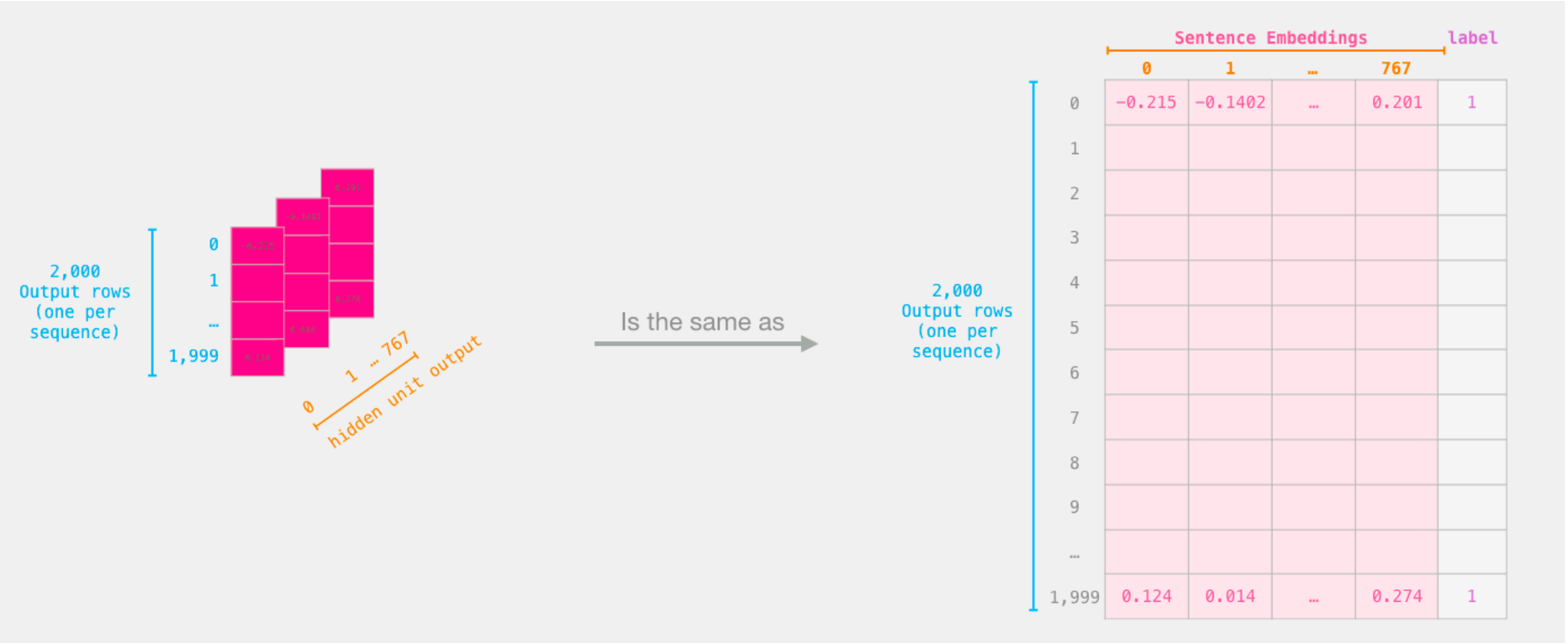
Slicing the important part




```
# Slice the output for the first position for all the sequences, take all hidden unit outputs
features = last_hidden_states[0][:,0,:].numpy()
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

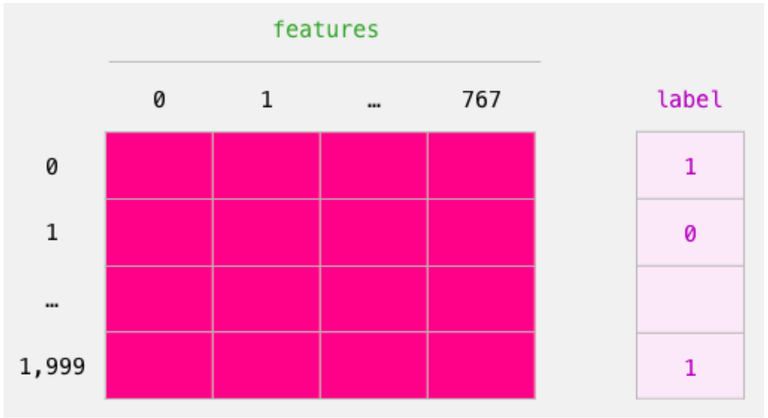
Final BERT output features

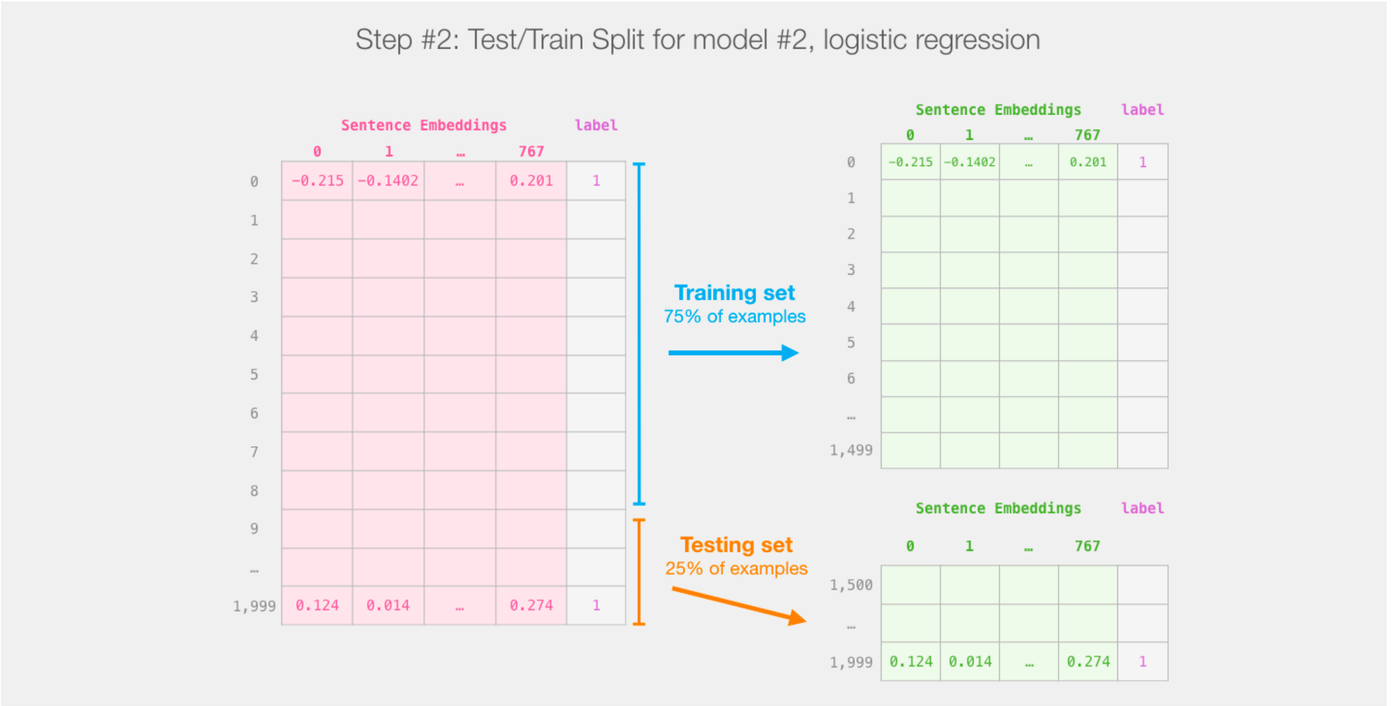



```
# Slice the output for the first position for all the sequences, take all hidden unit outputs
features = last_hidden_states[0][:,0,:].numpy()
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Dataset for logistic regression

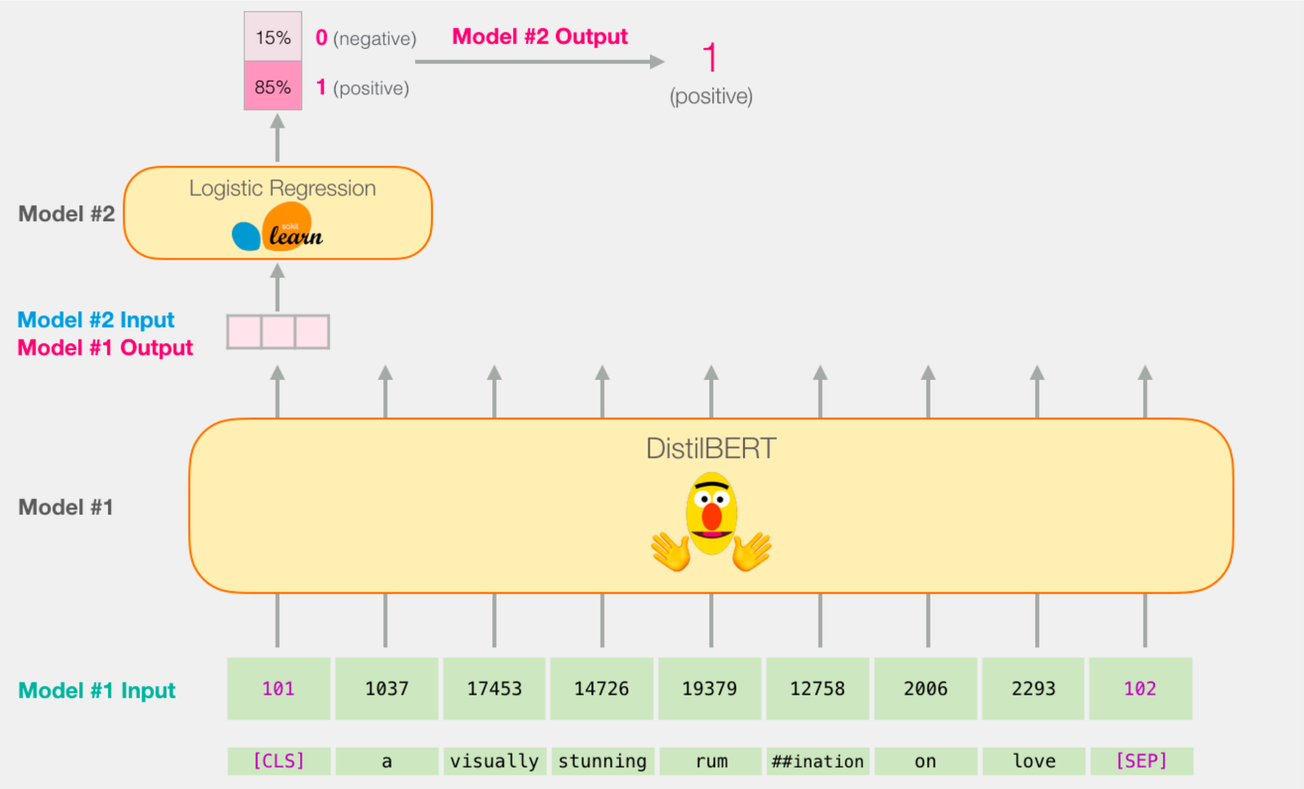





```
labels = df[1]
train_features, test_features, train_labels, test_labels = train_test_split(features, labels)
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Prediction step 4: get final predictions



Train a logistic regression classifier

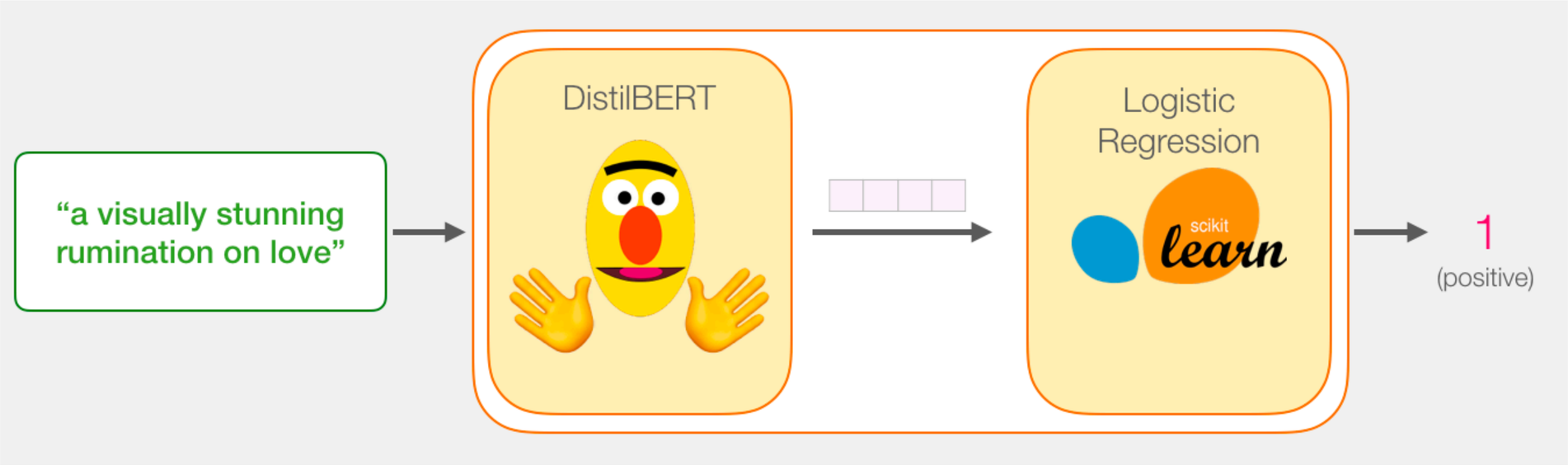
```
lr_clf = LogisticRegression()
lr_clf.fit(train_features, train_labels)
```

Run the trained logistic regression classifier

```
lr_clf.score(test_features, test_labels)
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

Example overview: BERT for sentiment classification



https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

BERT is a encoder-only language model

BERT (Bidirectional Encoder Representations from Transformers)

1. **Encoder-Only Model**: BERT is designed as an encoder-only model. In the context of the Transformer architecture, an encoder processes the input data (like text) to create a representation of it. BERT's architecture is composed entirely of such encoders.
2. **Cannot Generate Words**: BERT is not designed to generate text in the same way models like GPT-3 do. Instead, its strength lies in understanding the context and meaning of words in a sentence. This is why it excels in tasks like sentence classification, entity recognition, and question-answering, where understanding context is crucial.
3. **Working Mechanism**: BERT analyzes and encodes the input text, taking into account both the left and right context of each word in the input. This bidirectional understanding is a key feature that differentiates BERT from earlier models that could only analyze text in a single direction.

Credits: ChatGPT

How about BERT vs. GPT-3?

BERT (Bidirectional Encoder Representations from Transformers)

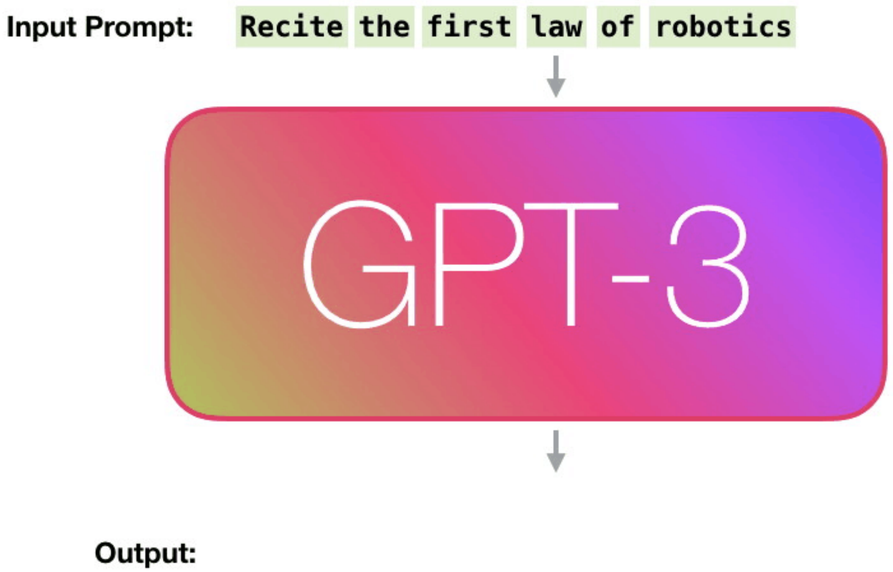
1. **Encoder-Only Model**: BERT is designed as an encoder-only model. In the context of the Transformer architecture, an encoder processes the input data (like text) to create a representation of it. BERT's architecture is composed entirely of such encoders.
2. **Cannot Generate Words**: BERT is not designed to generate text in the same way models like GPT-3 do. Instead, its strength lies in understanding the context and meaning of words in a sentence. This is why it excels in tasks like sentence classification, entity recognition, and question-answering, where understanding context is crucial.
3. **Working Mechanism**: BERT analyzes and encodes the input text, taking into account both the left and right context of each word in the input. This bidirectional understanding is a key feature that differentiates BERT from earlier models that could only analyze text in a single direction.

GPT-3 (Generative Pretrained Transformer 3)

1. **Decoder-Only Model**: GPT-3, on the other hand, is a decoder-only model. In the Transformer architecture, a decoder is designed to generate output based on the input it receives. While BERT focuses on understanding and encoding input, GPT-3 focuses on generating output.
2. **Generates Tokens**: GPT-3 is capable of generating text, making it suitable for tasks like text completion, creative writing, and dialogue generation. It generates one word (or token) at a time and can continue generating text based on the context provided by the previous text.
3. **Working Mechanism**: GPT-3 uses a unidirectional approach, meaning it only considers the context to the left (previous tokens) when generating a new token. This design is conducive to generating coherent and contextually relevant continuations of the input text.

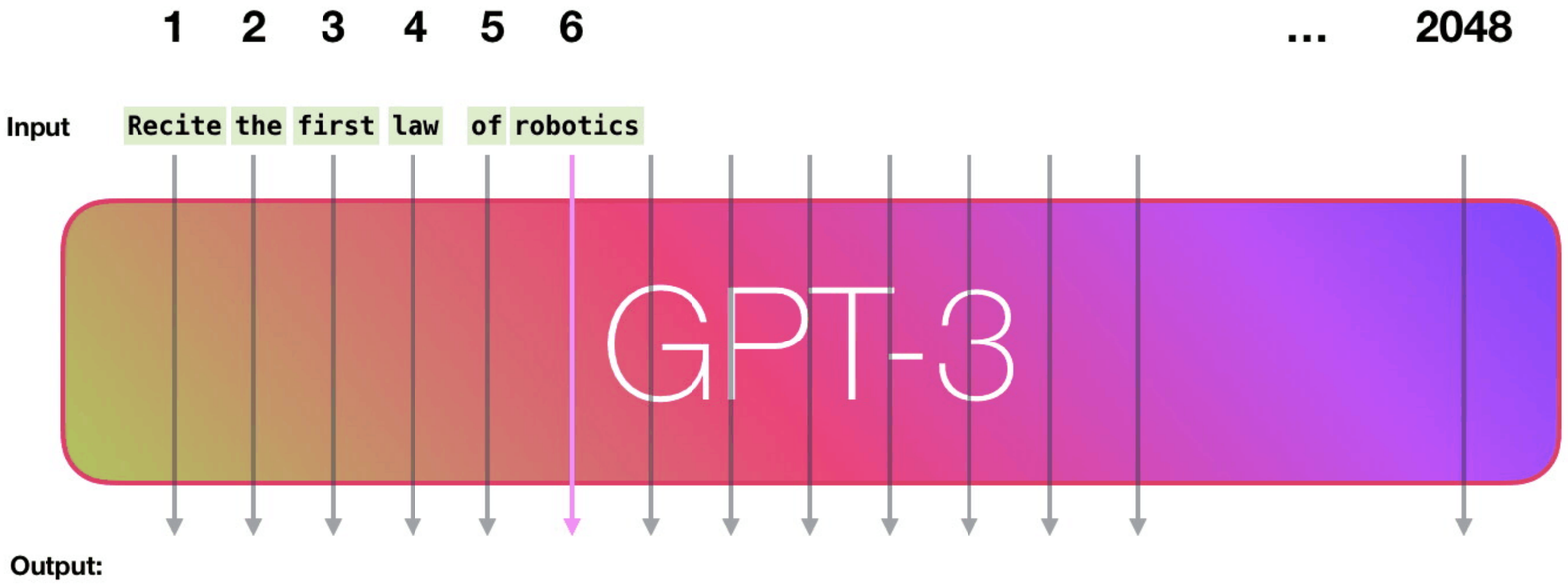
Credits: ChatGPT

Usage example: GPT-3



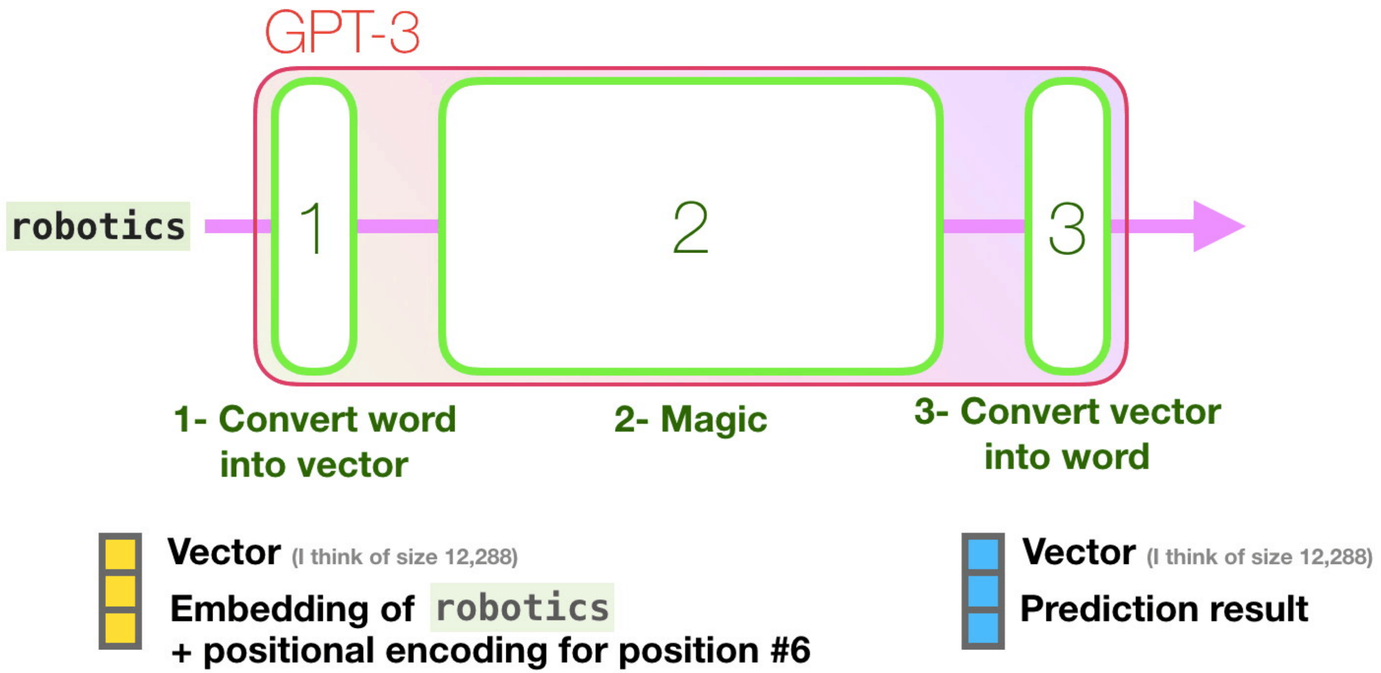
https://jalammar.github.io/how-gpt3-works-visualizations-animations/

Usage example: GPT-3



https://jalammar.github.io/how-gpt3-works-visualizations-animations/

Usage example: GPT-3

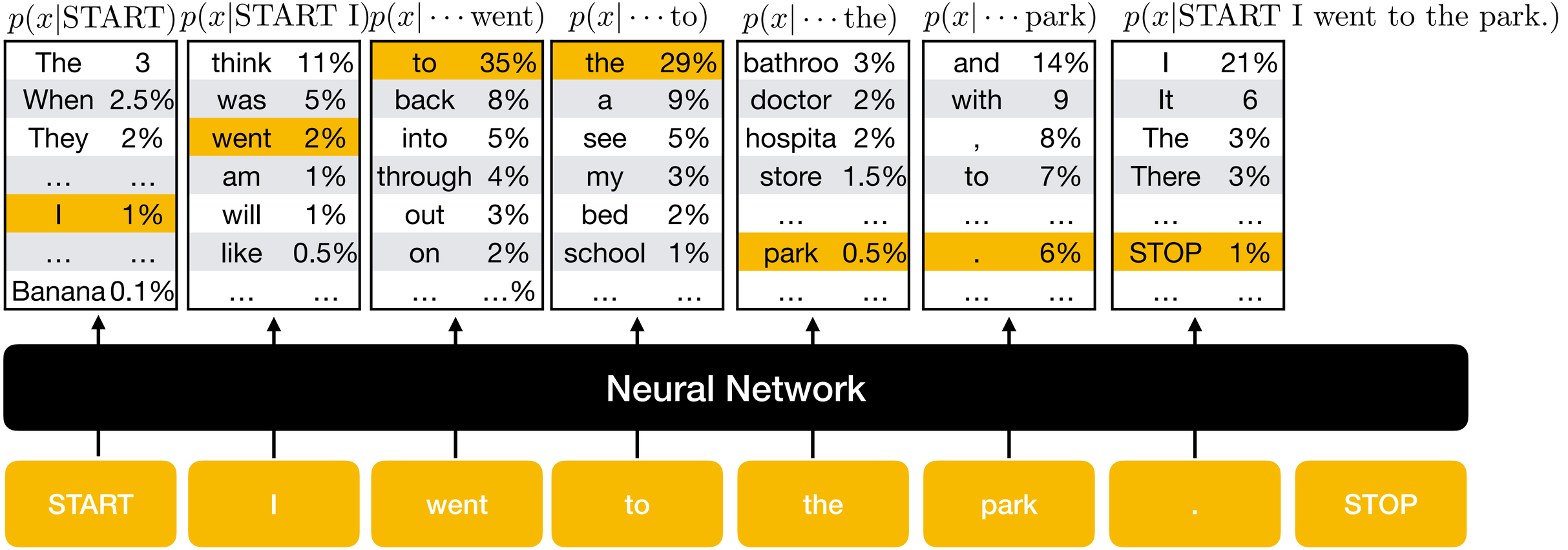


https://jalammar.github.io/how-gpt3-works-visualizations-animations/

Neural language models: tokenization

Neural language models: inputs/outputs

- **Input:** sequences of words (or tokens)
- **Output:** probability distribution over the next word (token)



Neural language models: input vectors

But neural networks take in real-valued vectors, not words…

- Use one-hot or learned embeddings to map from words to vectors!
  - Learned embeddings become part of parameters θ

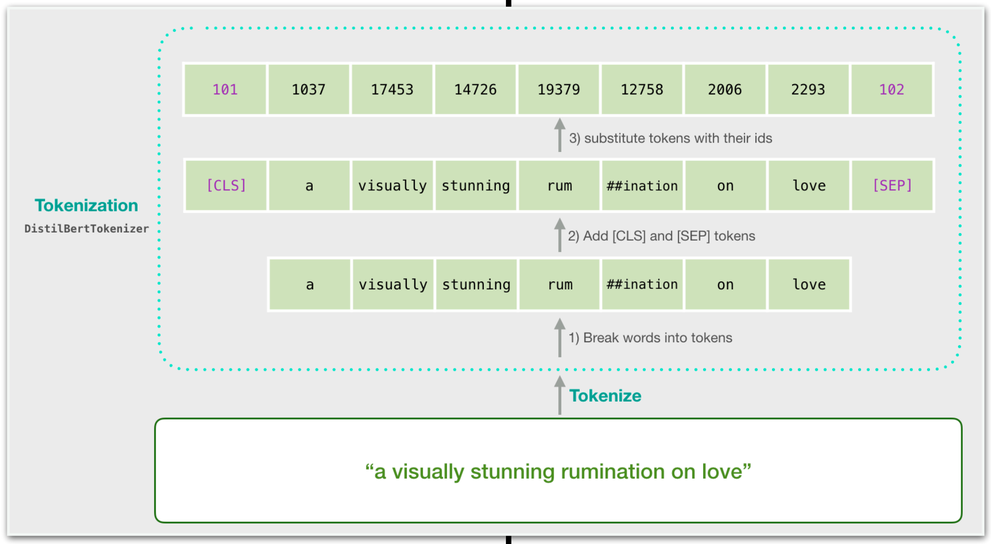
Tokenization to input vectors

$p(x|\text{START})\ p(x|\text{START I})\ p(x|\cdots\text{went})\quad p(x|\cdots\text{to})\quad p(x|\cdots\text{the})\quad p(x|\cdots\text{park})\quad p(x|\text{START I went to the park.})$

Neural Network

Mapping each tokenized id into its corresponding embeddings

Tokenization:



START | I | went | to | the | park | . | STOP

ChatGPT tokenization example

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people’s hats off—then, I account it high time tozz get to sea as soon as I can. This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

TEXT TOKEN IDS

Tokens 239

Characters 1109

[7368, 757, 57704, 1764, 301, 13, 4427, 1667, 4227, 2345, 37593, 4059, 1268, 1317, 24559, 2345, 69666, 2697, 477, 912, 3300, 304, 856, 53101, 11, 323, 4400, 4040, 311, 2802, 757, 389, 31284, 11, 358, 3463, 358, 1053, 30503, 922, 264, 2697, 323, 1518, 279, 30125, 727, 961, 315, 279, 1917, 13, 1102, 374, 264, 1648, 358, 617, 315, 10043, 1022, 279, 87450, 268, 323, 58499, 279, 35855, 13, 43633, 358, 1505, 7182, 7982, 44517, 922, 279, 11013, 26, 15716, 433, 374, 264, 41369, 11, 1377, 73825, 6841, 304, 856, 13836, 26, 15716, 358, 1505, 7182, 4457, 3935, 6751, 7251, 985, 1603, 78766, 83273, 11, 323, 12967, 709, 279, 14981, 315, 1475, 32079, 358, 3449, 26, 323, 5423, 15716, 856, 6409, 981, 636, 1778, 459, 8582, 1450, 315, 757, 11, 430, 433, 7612, 264, 3831, 16033, 17966, 311, 5471, 757, 505, 36192, 36567, 1139, 279, 8761, 11, 323, 1749, 2740, 50244, 1274, 753, 45526, 1022, 2345, 3473, 11, 358, 2759, 433, 1579, 892, 311, 10616, 636, 311, 9581, 439, 5246, 439, 358, 649, 13, 1115, 374, 856, 28779, 369, 40536, 323, 5041, 13, 3161, 264, 41903, 67784, 356, 4428, 3872, 5678, 5304, 813, 20827, 26, 358, 30666, 1935, 311, 279, 8448, 13, 2684, 374, 4400, 15206, 304, 420, 13, 1442, 814, 719, 7020, 433, 11, 4661, 682, 3026, 304, 872, 8547, 11, 1063, 892, 477, 1023, 11, 87785, 1890, 16024, 7119, 279, 18435, 449, 757, 13]

TEXT TOKEN IDS

https://platform.openai.com/tokenizer

Vocabulary: word-level

- For the n-gram model, our vocabulary $\mathcal{V}$ was comprised of all of the words in a language
- Some problems with this:
  - $|\mathcal{V}|$ **can be quite large** - ~470,000 words Webster's English Dictionary (3rd edition)
  - **Language is changing all of the time** - 690 words were added to Merriam Webster's in September 2023 ("rizz", "goated", "mid")
  - **Long tail of infrequent words**. Many words just occur a few times
  - **Some words may not appear** in a training set of documents
  - **No modeled relationship between words** - e.g., "run", "ran", "runs", "runner" are all separate entries despite being linked in meaning

Character-level?

What about representing text with characters?

- $V = \{a, b, c, \ldots, z\}$
  - (Maybe add capital letters, punctuation, spaces, …)
- Pros:
  - Small vocabulary size ($|V| = 26$ for English)
  - Complete coverage (unseen words are represented by letters)
- Cons:
  - Encoding becomes very long - # chars instead of # words
  - Poor inductive bias for learning

~~Word~~ ~~Character~~ Subword tokenization!

How can we combine the high coverage of character-level representation with the efficiency of word-level representation?

Subword tokenization! (e.g., Byte-Pair Encoding)

- Start with character-level representations
- Build up representations from there

Original BPE Paper (Sennrich et al., 2016)

https://arxiv.org/abs/1508.07909

Byte-pair encoding: ChatGPT example

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off—then, I account it high time tozz get to sea as soon as I can. This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

TEXT TOKEN IDS

Tokens 239

Characters 1109

[7368, 757, 57704, 1764, 301, 13, 4427, 1667, 4227, 2345, 37593, 4059, 1268, 1317, 24559, 2345, 69666, 2697, 477, 912, 3300, 304, 856, 53101, 11, 323, 4400, 4040, 311, 2802, 757, 389, 31284, 11, 358, 3463, 358, 1053, 30503, 922, 264, 2697, 323, 1518, 279, 30125, 727, 961, 315, 279, 1917, 13, 1102, 374, 264, 1648, 358, 617, 315, 10043, 1022, 279, 87450, 268, 323, 58499, 279, 35855, 13, 43633, 358, 1505, 7182, 7982, 44517, 922, 279, 11013, 26, 15716, 433, 374, 264, 41369, 11, 1377, 73825, 6841, 304, 856, 13836, 26, 15716, 358, 1505, 7182, 4457, 3935, 6751, 7251, 985, 1603, 78766, 83273, 11, 323, 12967, 709, 279, 14981, 315, 1475, 32079, 358, 3449, 26, 323, 5423, 15716, 856, 6409, 981, 636, 1778, 459, 8582, 1450, 315, 757, 11, 430, 433, 7612, 264, 3831, 16033, 17966, 311, 5471, 757, 505, 36192, 36567, 1139, 279, 8761, 11, 323, 1749, 2740, 50244, 1274, 753, 45526, 1022, 2345, 3473, 11, 358, 2759, 433, 1579, 892, 311, 10616, 636, 311, 9581, 439, 5246, 439, 358, 649, 13, 1115, 374, 856, 28779, 369, 40536, 323, 5041, 13, 3161, 264, 41903, 67784, 356, 4428, 3872, 5678, 5304, 813, 20827, 26, 358, 30666, 1935, 311, 279, 8448, 13, 2684, 374, 4400, 15206, 304, 420, 13, 1442, 814, 719, 7020, 433, 11, 4661, 682, 3026, 304, 872, 8547, 11, 1063, 892, 477, 1023, 11, 87785, 1890, 16024, 7119, 279, 18435, 449, 757, 13]

TEXT TOKEN IDS

https://platform.openai.com/tokenizer

Byte-pair encoding: usage

- Basically state of the art in tokenization
- Used in all modern left-to-right large language models (LLMs), including ChatGPT

Model/Tokenizer	Vocabulary Size
GPT-3.5/GPT-4/ChatGPT	100k
GPT-2/GPT-3	50k
Llama2	32k
Falcon	65k

Byte-pair encoding (BPE): algorithm

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than characters in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$ → $\mathcal{D} = \{\text{“i hug pugs”}, \text{“hugging pugs is fun”}, \text{“i make puns”}\}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$) → $N = 20$

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation) → $\mathcal{D} = \{\text{“i”}, \text{“ hug”}, \text{“ pugs”}, \text{“hugging”}, \text{“ pugs”}, \text{“ is”}, \text{“ fun”}, \text{“i”}, \text{“ make”}, \text{“ puns”}\}$
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$ → $\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$
- Convert $\mathcal{D}$ into a list of tokens (characters) → $\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

Example inspired by: https://huggingface.co/docs/transformers/tokenizer_summary

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{V} = \{1 : \text{`\ '}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'}, 8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'}\}$$

Implementation aside: We normally store $\mathcal{D}$ with the token indices instead of the text itself!

$$\mathcal{D} = \{ [7], [1, 6, 13, 5], [1, 11, 13, 5, 12], [6, 13, 5, 5, 7, 10, 5], [1, 11, 13, 5, 12], [1, 7, 12], [1, 4, 13, 10], [7], [1, 9, 2, 8, 3], [1, 11, 13, 10, 12]\}$$

For legibility of the example, we will show the text corresponding to each token

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}] \}$$

Bigram	Count
‘u’,’g’	4
‘p’, ‘u’	3
‘ ‘, ‘p’	3
‘h’, ‘u’	2
…	…

$$v_{14} := \text{concat}(\text{‘u’}, \text{‘g’}) = \text{‘ug’}$$

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$v_{14} := \text{concat}(\text{‘u’}, \text{‘g’}) = \text{‘ug’}$$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’},$$
$$\text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}\}, |\mathcal{V}| = 14$$

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}] \}$$

Bigram	Count
‘ ‘, ‘p’	3
‘p’, ‘ug’	2
‘ug’, ’s'	2
‘u’, ’n'	2
…	…

$$v_{15} := \text{concat}(\text{‘ ’}, \text{‘p’}) = \text{‘ p’}$$

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$v_{15} := \text{concat}(\text{‘ ’}, \text{‘p’}) = \text{‘ p’}$$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’},$$
$$\text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}, \text{‘ p’}\}, |\mathcal{V}| = 15$$

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

Repeat until $|\mathcal{V}| = N$...

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ hug’}], [\text{‘ pugs’}], [\text{‘hug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ pugs’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘un’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘un’}, \text{‘s’}] \}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}, \text{‘ p’}, \text{‘hug’}, \text{‘ pug’}, \text{‘ pugs’}, \text{‘un’}, \text{‘ hug’}\},$$

$$|\mathcal{V}| = 20$$

CHANGES FROM START

Byte-pair encoding: example

CHANGES FROM START

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ hug’}], [\text{‘ pugs’}],$$
$$[\text{‘hug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ pugs’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘un’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘un’}, \text{‘s’}]\}$$

Questions to think about:

- Is every token we made used in the corpus? Why or why not?
- How much memory (#tokens) have we saved for each document?
- What would happen if you kept adding vocabulary until you couldn’t anymore?

$$\mathcal{D} = \{ [7], [20], [18],$$
$$[16, 5, 7, 10, 5], [18],$$
$$[1, 7, 12], [1, 4, 19], [7],$$
$$[1, 9, 2, 8, 3], [15, 19, 12]\}$$

(as tokens indices)

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

Byte-pair encoding: tokenization/encoding

With this vocabulary, can you represent (or, tokenize/encode):

- “apple”?
  - No, there is no ‘l’ in the vocabulary
- “huge”?
  - Yes - [16, 4]
- “ huge”?
  - Yes - [18, 4]
- “ hugest”?
  - No, there is no ‘t’ in the vocabulary
- “unassumingness”?
  - Yes - [19, 2, 12, 12, 13, 9, 7, 10, 5, 10, 3, 12, 12]

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

Byte-pair encoding: tokenization/encoding

$$\mathcal{V} = \{1 : \text{`}\ \text{'}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'},$$
$$8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'},$$
$$14 : \text{`ug'}, 15 : \text{`}\ \text{p'}, 16 : \text{`hug'}, 17 : \text{`}\ \text{pug'}, 18 : \text{`}\ \text{pugs'},$$
$$19 : \text{`un'}, 20 : \text{`}\ \text{hug'}\}$$

- Sometimes, there may be more than one way to represent a word with the vocabulary…
  - E.g., “ hugs” = [20, 12] = [1, 16, 12] = [1, 6, 14, 12] = [1, 6, 13, 5, 13]
    - Which is the best representation? Why?

Byte-pair encoding: tokenization/encoding

$$\mathcal{V} = \{1 : \text{`` ''}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'},$$
$$8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'},$$
$$14 : \text{`ug'}, 15 : \text{`\ p'}, 16 : \text{`hug'}, 17 : \text{`\ pug'}, 18 : \text{`\ pugs'},$$
$$19 : \text{`un'}, 20 : \text{`\ hug'}\}$$

Encoding algorithm

Given string s and (ordered) vocab $\mathcal{V}$,

- Pretokenize $\mathcal{D}$ in same way as before
- Tokenize $\mathcal{D}$ into characters
- Perform merge rules in same order as in training until no more merges may be done

Byte-pair encoding: tokenization/encoding

$$\mathcal{V} = \{1 : \text{`` ''}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'},$$
$$8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'},$$
$$14 : \text{`ug'}, 15 : \text{`\ p'}, 16 : \text{`hug'}, 17 : \text{`\ pug'}, 18 : \text{`\ pugs'},$$
$$19 : \text{`un'}, 20 : \text{`\ hug'}\}$$

Encoding algorithm

Given string $\mathcal{S}$ and (ordered) vocab $\mathcal{V}$,

- Pretokenize $\mathcal{D}$ in same way as before
- Tokenize $\mathcal{D}$ into characters
- Perform merge rules in same order as in training until no more merges may be done

$$\text{Encode}(\text{`` hugs''}) = [20, 12]$$
$$\text{Encode}(\text{``misshapenness''}) = [9, 7, 12, 12, 6, 2,$$
$$11, 3, 10, 10, 3, 12, 12]$$

Byte-pair encoding: decoding

$$\mathcal{V} = \{1 : \text{`` ''}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'},$$
$$8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'},$$
$$14 : \text{`ug'}, 15 : \text{`\ p'}, 16 : \text{`hug'}, 17 : \text{`\ pug'}, 18 : \text{`\ pugs'},$$
$$19 : \text{`un'}, 20 : \text{`\ hug'}\}$$

Decoding algorithm

Given list of tokens T:

- Initialize string $s := \text{“”}$
- Keep popping off tokens from the front of T and appending the corresponding string to s

$$\text{Encode}(\text{“ hugs”}) = [20, 12]$$
$$\text{Encode}(\text{“misshapenness”}) = [9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]$$

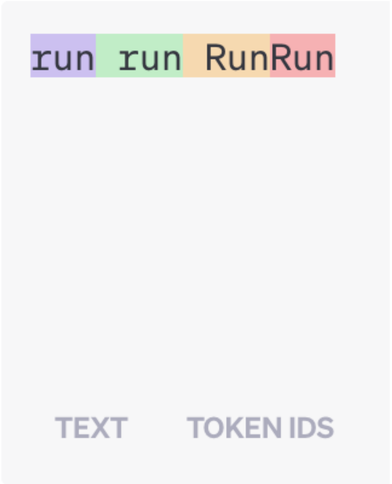
$$\text{Decode}([20, 12]) = \text{“ hugs”}$$
$$\text{Decode}([9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]) = \text{“misshapenness”}$$

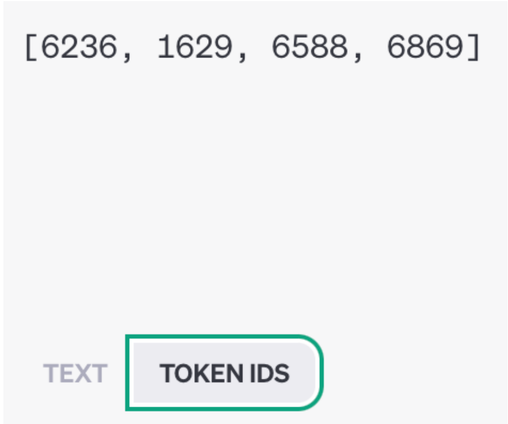
Byte-pair encoding: properties

- Efficient to run (greedy vs. global optimization)
- Lossless compression
- Potentially some shared representations - e.g., the token “hug” could be used both in “hug” and “hugging”

Weird properties of tokenizers

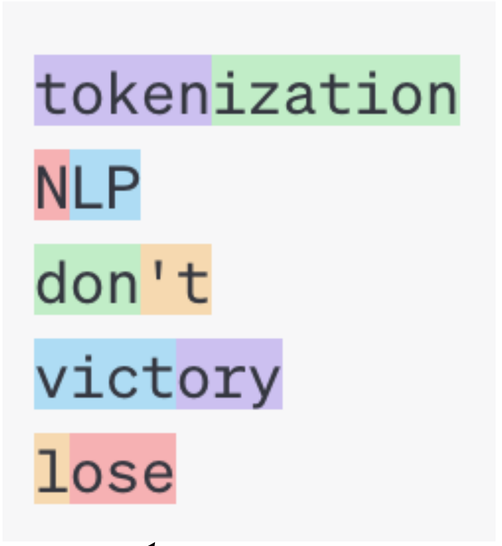
- Token != word
- Spaces are part of token
  - “run” is a different token than “ run”
- Not invariant to case changes
  - “Run” is a different token than “run”

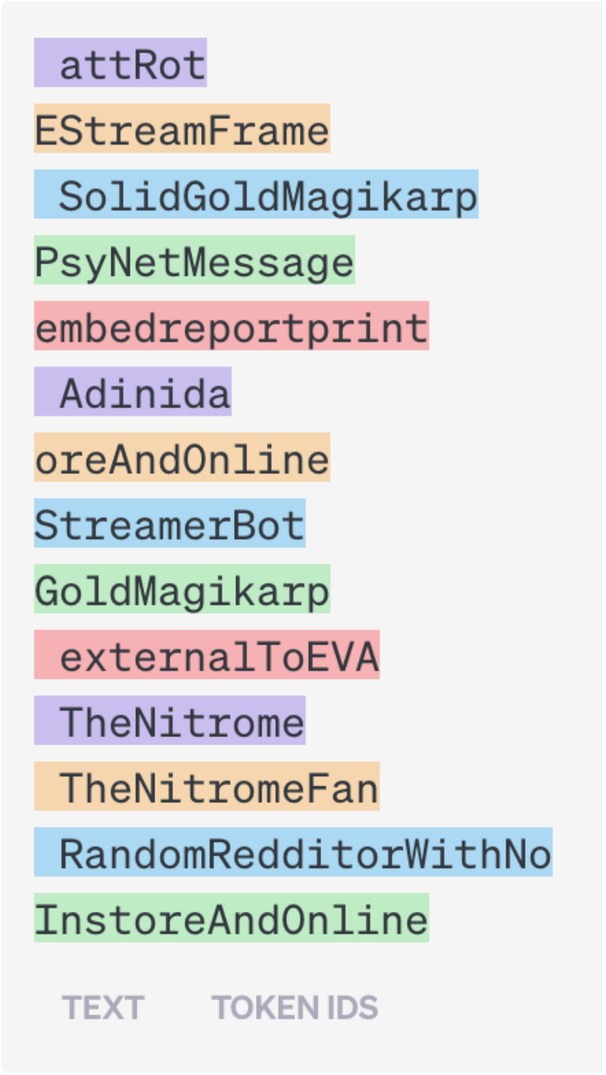




Weird properties of tokenizers

- Token != word
- Spaces are part of token
  - “run” is a different token than “ run”
- Not invariant to case changes
  - “Run” is a different token than “run”
- Tokenization fits statistics of your data
  - e.g., while these words are multiple tokens…
  - These words are all 1 token in GPT-3’s tokenizer!
  - *Why?*
    - Reddit usernames and certain code attributes appeared enough in the corpus to surface as its own token!





Example from https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation

Other tokenization variants

Variants: no spaces in tokens

- The way we presented BPE, we included whitespace with the following word. (E.g., “ pug”)
  - This is most common in modern LMs
- However, in another BPE variant, you instead strip whitespace (e.g., “pug”) and add spaces between words at decoding time
  - This was the original BPE paper’s implementation!
- Example:
  - [“I”, “hug”, “pugs”] -> “I hug pugs” (w/out whitespace)
  - [“I”, “ hug”, “ pugs”] -> “I hug pugs” (w/ whitespace)

space

no space

Original (w/ whitespace)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (**split before** whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

Updated (w/out whitespace)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

+ Pre-tokenize $\mathcal{D}$ by splitting into words (**removing** whitespace)

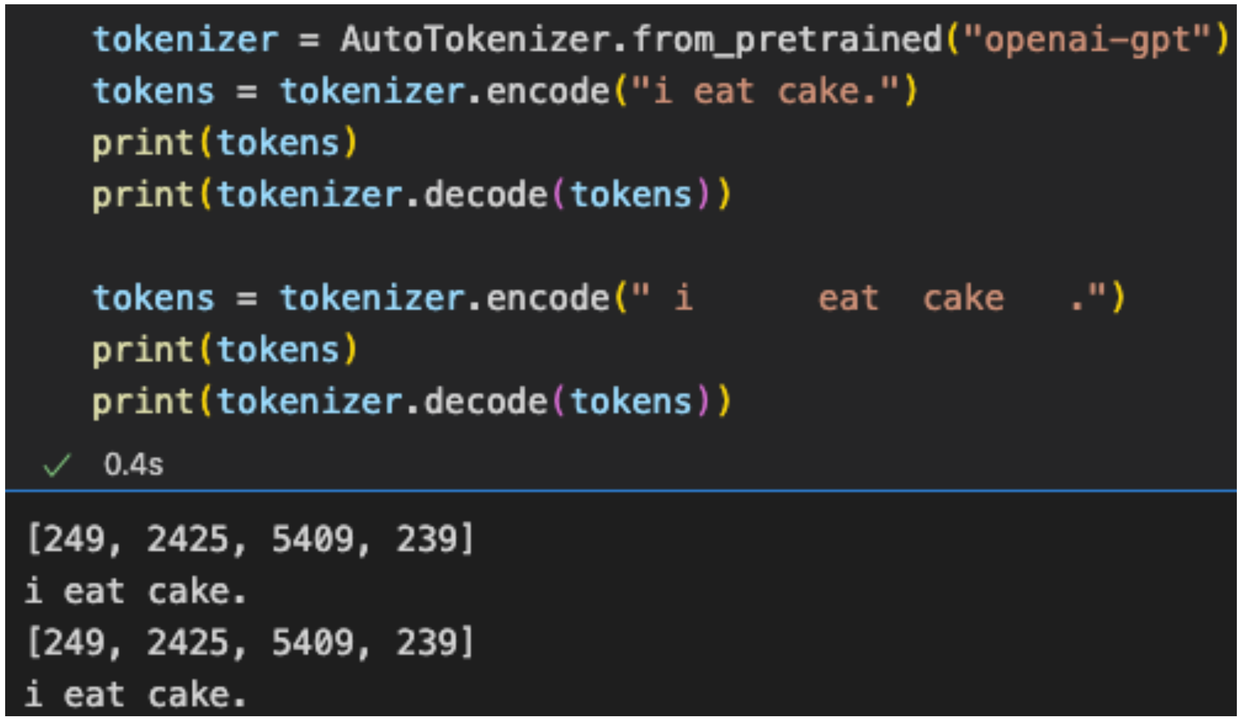
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

Variants: no spaces in tokens

- For sub-word tokens, need to add “continue word” special character
  - E.g., for the word “Tokenization”, if the subword tokens are “Token” and “ization”,
    - W/out special character: [“Token”, “ization”] -> “Token ization”
    - W/ special character #: [“Token”, “#ization”] -> Tokenization”
  - When decoding, if does not have special character add a space
- Example:
  - [“I”, “li”, “#ke”, “to”, “hug”, “pug”, “#s”] -> “I like to hug pugs”

Variants: no spaces in tokens

- Loses some whitespace information (lossy compression!)
  - E.g., Tokenize("I eat cake.") == Tokenize(" I eat cake .")
  - Especially problematic for code (e.g., Python) - why?



(Example using GPT's tokenizer, which does not include spaces in the token)

Variants: no pre-tokenization

- In the variant we proposed, we start by splitting into words
  - This guarantees that each token will be no longer than one word
  - However, this does not work so well for character-based languages. *Why?*

Variants: no pre-tokenization

- Instead, we could *not* pre-tokenize, and treat the entire document or sentence as a single list of tokens
  - Allows for tokens to span multiple words/characters
- Sometimes called SentencePiece tokenization* (Kudo, 2018)

* (not to be confused with the SentencePiece library, which is an implementation of *many* kinds of tokenization)

Paper: https://arxiv.org/abs/1808.06226
Library: https://github.com/google/sentencepiece

Original (w/ pre-tokenization)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- **Pre-tokenize** $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

Updated (w/out pre-tokenization)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

+ **Do not pre-tokenize** $\mathcal{D}$

- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)

Variants: no pre-tokenization

- Allows sequences of words/characters to become tokens

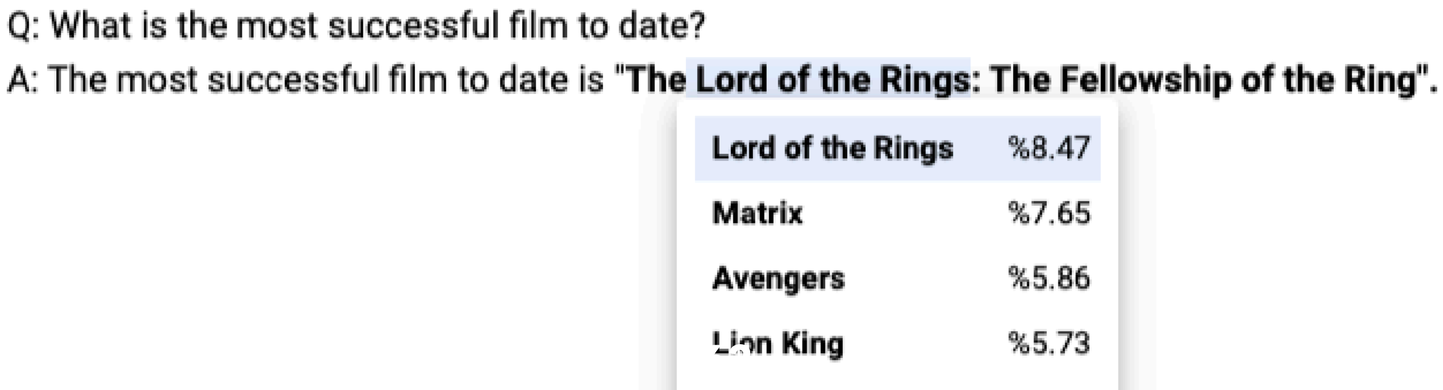
SentencePiece paper example in Japanese:

https://arxiv.org/pdf/1808.06226.pdf

- **Raw text:** [こんにちは世界。] (*Hello world.*)
- **Tokenized:** [こんにちは] [世界] [。]

Jurassic-1 model example in English:

https://uploads-ssl.webflow.com/60fd4503684b466578c0d307/61138924626a6981ee09caf6_jurassic_tech_paper.pdf



Variants: byte-based

- Originally, we presented BPE as dealing with characters as the smallest unit
  - However, there are *many* characters - especially if you want to support:
    - character-based languages (e.g., Chinese has >100k characters!)
    - non-alphanumeric characters like emojis (Unicode 15 has ~150k characters!)
- Instead, can initialize tokens as set of bytes! (e.g., with UTF-8*)

*Only 256 bytes! Each Unicode char is 1-4 bytes

Original (w/ characters)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)

- Initialize $\mathcal{V}$ as the set of **characters** in $\mathcal{D}$

- Convert $\mathcal{D}$ into a list of tokens (**characters**)

- While $|\mathcal{V}| < N$:

Modified (w/ bytes)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)

+ Initialize $\mathcal{V}$ as the set of **bytes** in $\mathcal{D}$

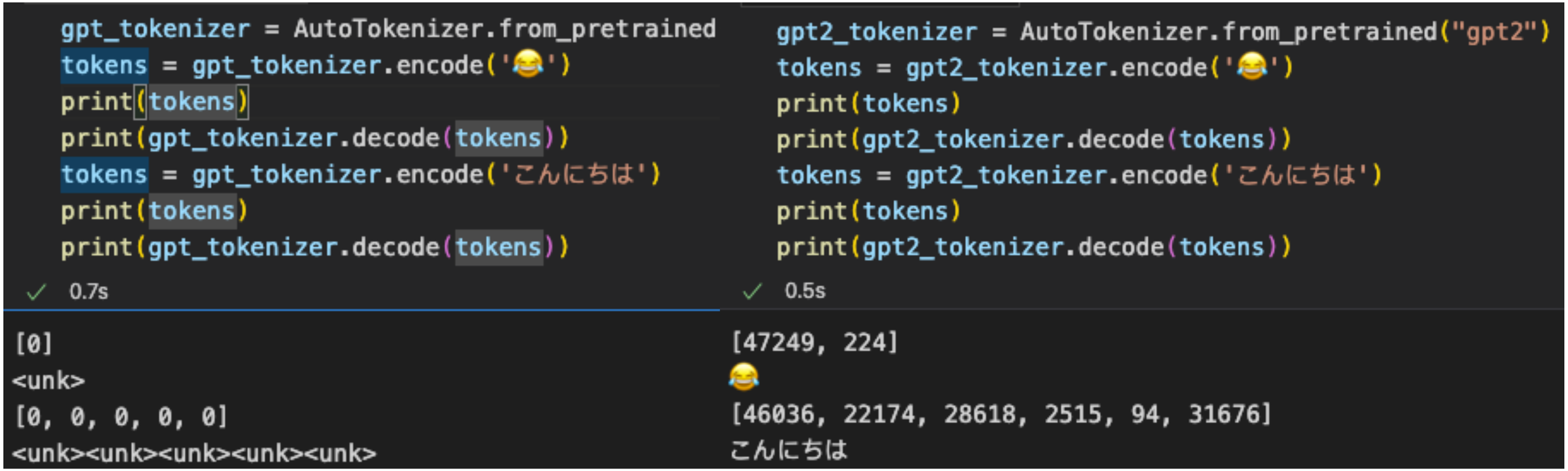
+ Convert $\mathcal{D}$ into a list of tokens (**bytes**)

- While $|\mathcal{V}| < N$:

Variants: byte-based

While character-based GPT tokenizer fails on emojis and Japanese…

The Byte-based GPT-2 tokenizer succeeds!



Variants: WordPiece objective

- To merge, we selected the bigram with highest frequency
  - This is the same as bigram with highest probability!
- Instead, we could choose the bigram which would maximize the likelihood of the data after the merge is made (also called WordPiece!)

Original (BPE)

…

- For the most frequent bigram v_i, v_j (breaking ties arbitrarily)

(Sam as bigram which maximizes - $p(v_i, v_j)$)

Modified (Word Piece)

…

+ For the bigram that would maximize likelihood of the training data once the change is made v_i, v_j (breaking ties arbitrarily)

(Same as bigram which maximizes $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$)

Variants: WordPiece objective

- BPE: the bigram with highest frequency/highest probability $p(v_i, v_j)$
- WordPiece: bigram which maximizes the likelihood of the data after the merge is made $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$
  - Maximizes the probability of the bigram, normalized by the probability of the unigrams

Variants: WordPiece encoding

At inference time, instead of applying the merge rules in order, tokens are selected left-to-right greedily:

Encoding algorithm

Given string s and (unordered) vocab $\mathcal{V}$,

- Initialize list of tokens $T := []$
- While $len(s) > 0$:
  - Find longest token t_i that matches the beginning of s
  - Let $T := T + [t_i]$
  - Pop corresponding vocab v_i off of front of s
- Return T

Variants: unigram objective

- BPE starts with a small vocabulary (characters) and builds up until the desired vocabulary size N
- The Unigram tokenization algorithm starts with a large vocabulary (all sub-word substrings) and throws away tokens until we reach size N

Examples of LLMs and their tokenizers

Model/Tokenizer	Objective	Spaces part of token?	Pre-tokenization	Smallest unit
GPT	BPE	No	Yes	Character-level
GPT-2/3/4, ChatGPT, Llama(2), Falcon, …	BPE	Yes	Yes	Byte-level
Jurassic	BPE	Yes	No. “SentencePiece” - treat whitespace like char	Byte-level
Bert, DistilBert, Electra	WordPiece	No	Yes	Character-level
T5, ALBERT, XLNet, Marian	Unigram	Yes	No. “SentencePiece” - treat whitespace like char*	Character-level

*For non-English languages

Next lecture: Recurrent neural networks