DeepLearning.AI

Introduction to NLP with PyTorch

Working with text using PyTorch

In Module 3 you'll dive into:

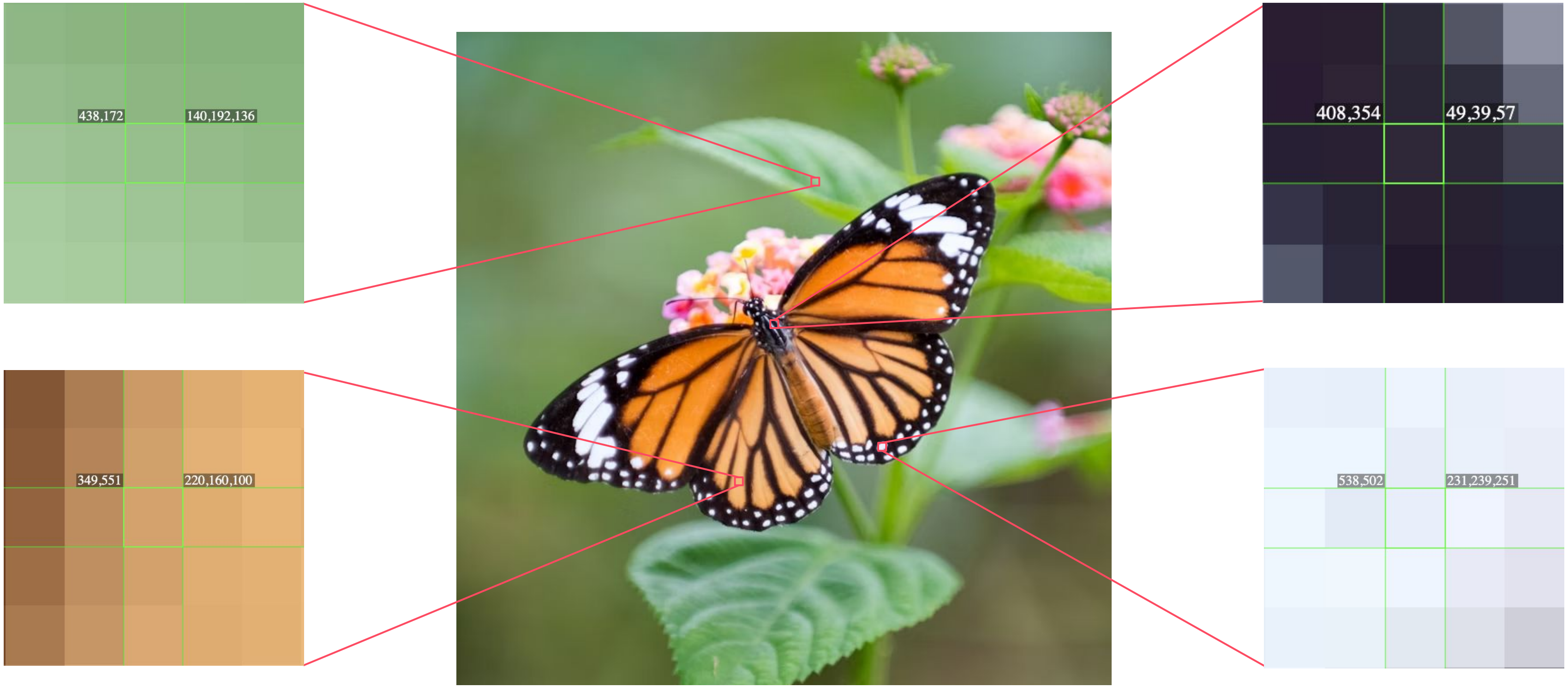
Optimization

Images

Text

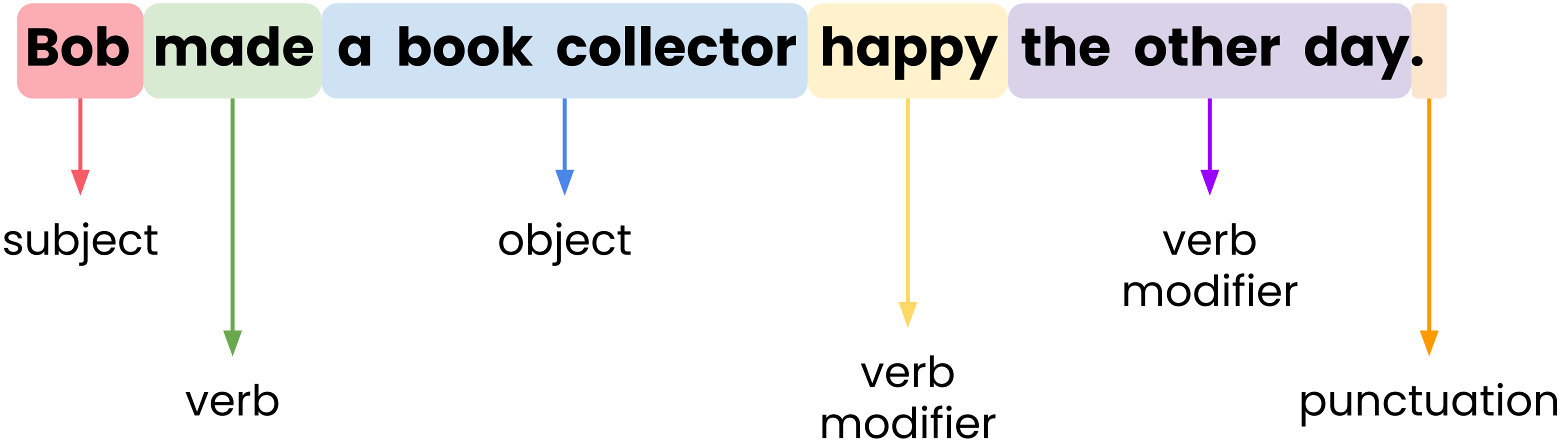
Efficiency

Images are represented as numerical values

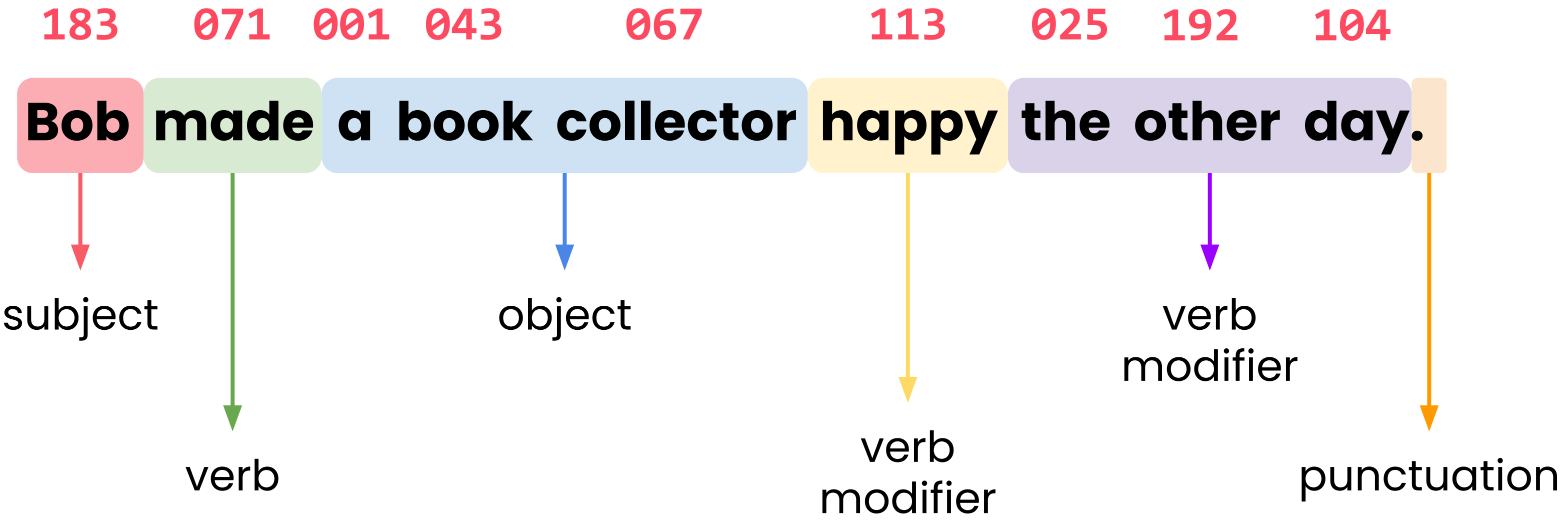


Laurence Moroney

Text has no intrinsic numeric meaning



Text has no intrinsic numeric meaning



Text has no intrinsic numeric meaning

Bob made a book collector happy the other day.

Last week, Bob made a book collector's day.

The other day, Bob delighted a book collector.

Bob recently brought delight to a book collector.

Text is sequential and contextual

Text is sequential and contextual

"A **bat** flew out of the cave"

Text is sequential and contextual

"A **bat** flew out of the cave"

"He swung the baseball **bat**"

NLP workflows require:

Specific steps

For text processing

NLP workflows require:

Specific steps

For text processing

Specialized models

To capture meaning and context

NLP workflows require:

Specific steps

For text processing

Specialized models

To capture meaning and context

Large datasets

To handle complexity

What makes text data challenging?

What makes text data challenging?

Sequence dependencies and context

What makes text data challenging?

Sequence dependencies and context

Variable length and structure

What makes text data challenging?

Sequence dependencies and context

Variable length and structure

Vocabulary and representation challenges

What makes text data challenging?

Sequence dependencies and context

Variable length and structure

Vocabulary and representation challenges

Ambiguity and polysemy

What makes text data challenging?

Sequence dependencies and context

Variable length and structure

Vocabulary and representation challenges

Ambiguity and polysemy

NLP applications: Classification

NLP applications: Classification

Sentiment analysis

NLP applications: Classification

- Sentiment analysis
- Spam detection

NLP applications: Classification

- Sentiment analysis
- Spam detection
- Intent classification

NLP applications: Classification

- Sentiment analysis
- Spam detection
- Intent classification
- Topic categorization

NLP applications: Labeling

Named entity recognition (NER)

- Search engines
- Virtual assistants
- Medical record analyzers

NLP applications: Labeling

Named entity recognition (NER)

- Search engines
- Virtual assistants
- Medical record analyzers

Alice works at DeepLearning.AI in Mountain View.

Alice → `person`

DeepLearning.AI → `organization`

Mountain View → `location`

NLP applications: Labeling

Named entity recognition (NER)

- Search engines
- Virtual assistants
- Medical record analyzers

Part-of-speech tagging

- Grammar checkers
- Speech recognition

NLP applications: Generative tasks

NLP applications: Generative tasks

Summarization

NLP applications: Generative tasks

- Summarization
- Dialogue generation

NLP applications: Generative tasks

- Summarization
- Dialogue generation
- Machine translation

NLP applications: Generative tasks

- Summarization
- Dialogue generation
- Machine translation
- Creative writing

NLP applications: Generative tasks

- Summarization
- Dialogue generation
- Machine translation
- Creative writing

NLP applications: Generative tasks

- Summarization
- Dialogue generation
- Machine translation
- Creative writing

PyTorch

Laurence Moroney

PyTorch is a popular framework for NLP

Preprocessing text

Converting text to tensors

Building models for NLP tasks

PyTorch utilities for NLP

Before

TorchText: datasets, tokenizers, iterators

PyTorch utilities for NLP

Before

TorchText: datasets, tokenizers, iterators

Now

Tokenization libraries and pretrained models

DeepLearning.AI

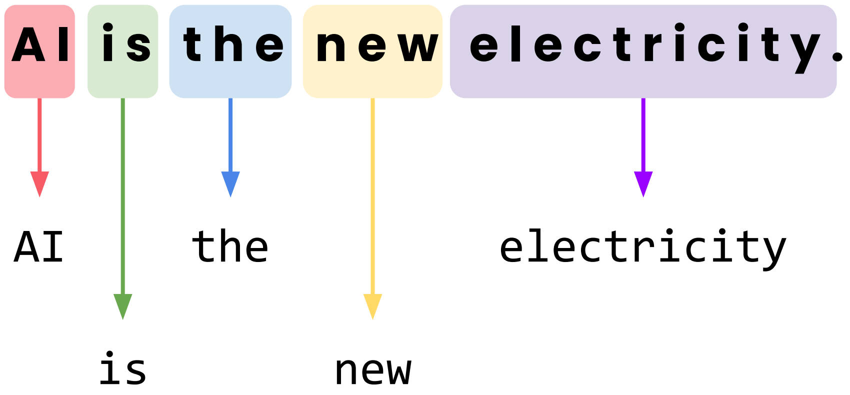
Tokenization

Working with text using PyTorch

Tokenization: Dividing text into tokens

Tokenization: Dividing text into tokens

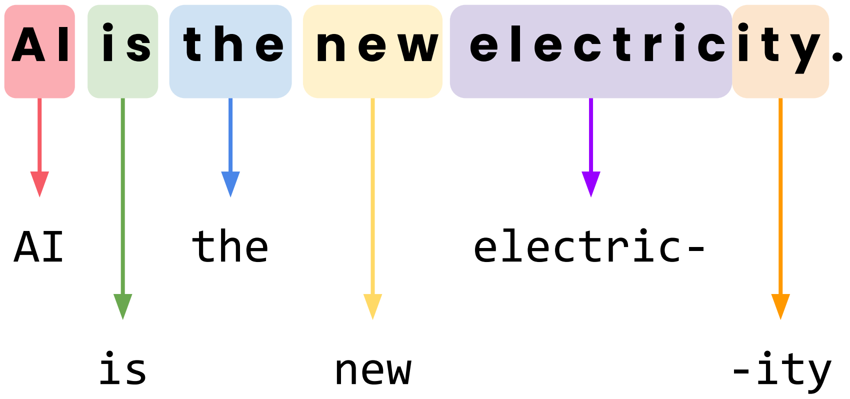
Words



Tokenization: Dividing text into tokens

Words

Subwords

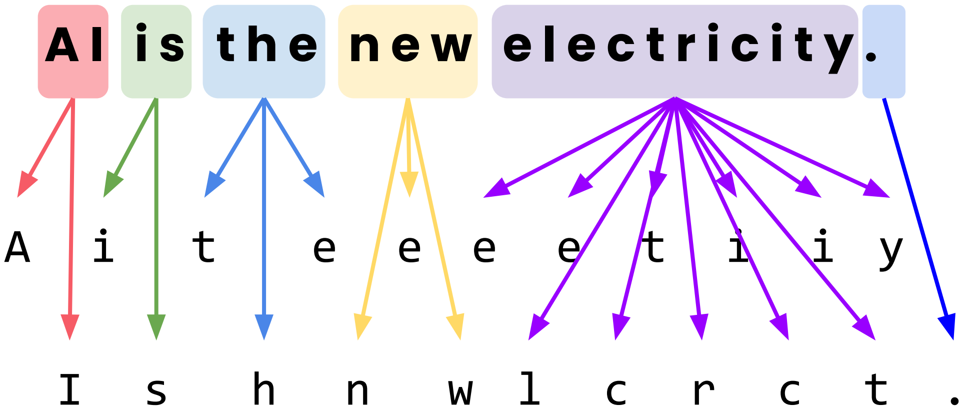


Tokenization: Dividing text into tokens

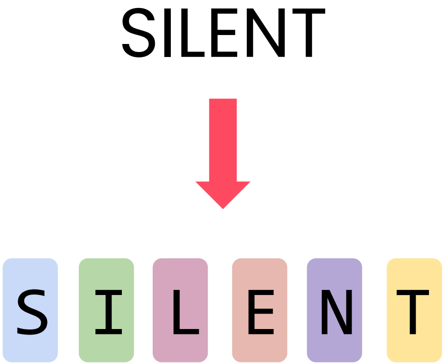
Words

Subwords

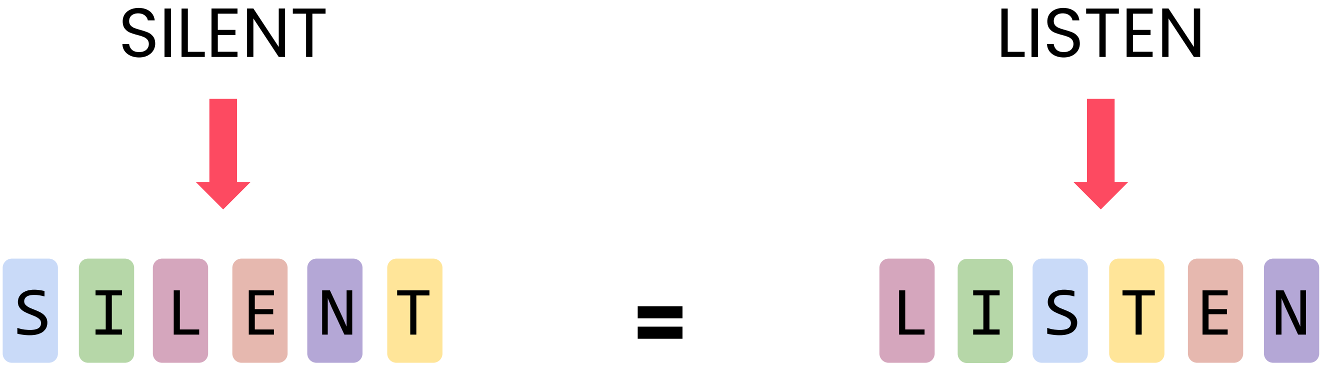
Characters



Character tokenization



Character tokenization



Numeric encodings: ASCII

Decimal	Character
64	@
65	A
66	B
67	C
68	D
...	...

Laurence Moroney

Word tokenization

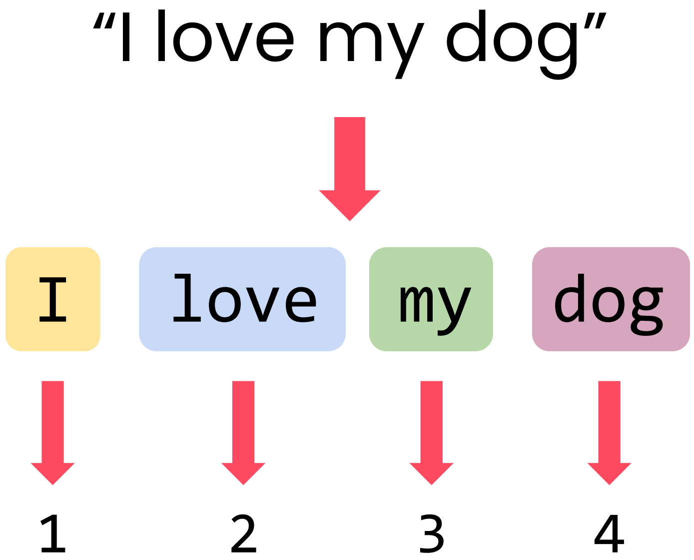
Word tokenization

"I love my dog"

I | love | my | dog

Laurence Moroney

Word tokenization



Special tokens

<UNK>	Unknown word
<PAD>	Pad sequences to match length
<START>	Beginning of sentence
<END>	End of sentence

Special tokens

<UNK>	Unknown word
<PAD>	Pad sequences to match length
<START>	Beginning of sentence
<END>	End of sentence
<CLS>	Classification inputs
<SEP>	Separate segments

Challenges with word tokenization

Vocabulary size

Challenges with word tokenization

Vocabulary size

Normalization

"Dog" vs. "dog"
"Hi!" vs. "Hi"
"don't" vs. "do not"

Building a tokenizer in PyTorch

```
sentences = [
    'I love my dog',
    'I love my cat'
]

# Tokenization function
def tokenize(text):
 # Lowercase the text and split by whitespace
 return text.lower().split()
```

Building a tokenizer in PyTorch

```
sentences = [
    'I love my dog',
    'I love my cat'
]

# Tokenization function
def tokenize(text):
 # Lowercase the text and split by whitespace
 return text.lower().split()
```

Building a tokenizer in PyTorch

```
sentences = [
    'I love my dog',
    'I love my cat'
]

# Tokenization function
def tokenize(text):
 # Lowercase the text and split by whitespace
 return text.lower().split()
```

```python
# Build the vocabulary
def build_vocab(sentences):
    vocab = {}
    # Iterate through each sentence.
    for sentence in sentences:
        # Tokenize the current sentence
        tokens = tokenize(sentence)
        # Iterate through each token in the sentence
        for token in tokens:
            # If the token is not already in the vocabulary
            if token not in vocab:
                # Add token to the vocabulary and assign it a unique integer ID
                # IDs start from 1; 0 can be reserved for padding.
                vocab[token] = len(vocab) + 1
    return vocab
```

```
# Build the vocabulary
def build_vocab(sentences):
    vocab = {}
    # Iterate through each sentence.
    for sentence in sentences:
        # Tokenize the current sentence
        tokens = tokenize(sentence)
        # Iterate through each token in the sentence
        for token in tokens:
            # If the token is not already in the vocabulary
            if token not in vocab:
                # Add token to the vocabulary and assign it a unique integer ID
                # IDs start from 1; 0 can be reserved for padding.
                vocab[token] = len(vocab) + 1
    return vocab
```

```python
# Build the vocabulary
def build_vocab(sentences):
    vocab = {}
    # Iterate through each sentence.
    for sentence in sentences:
        # Tokenize the current sentence
        tokens = tokenize(sentence)
        # Iterate through each token in the sentence
        for token in tokens:
            # If the token is not already in the vocabulary
            if token not in vocab:
                # Add token to the vocabulary and assign it a unique integer ID
                # IDs start from 1; 0 can be reserved for padding.
                vocab[token] = len(vocab) + 1
    return vocab
```

Building a tokenizer in PyTorch

```
# Create the vocabulary index
vocab = build_vocab(sentences)

print("Vocabulary Index:", vocab)
```

Output

```
Vocabulary Index: {'i': 1, 'love': 2, 'my': 3, 'dog': 4, 'cat': 5}
```

Variable-length problem

Different sentences have different lengths

vs.

5/28/2019

I have only written a couple of reviews in my life but after the amount of stress caused by my experience here I needed to say something. I came here for a cut and color. Originally a brunette, I had shared a detailed description of my hair history weeks prior to the apt, and images of the soft ash balayage I wanted as well as my current hair state. My hairdresser washed and cut my hair first, then dried it to apply color. From there I was explained that my hair was damaged and not able to create the look we had initially discussed. I understood, and happily agreed to something more subtle then we initially discussed (soft ashy natural brown balayage highlights was what we originally talked about). After the wash I was brought back to the station only to see a full transformation of my "damaged hair" from brunette to orange lemon blonde (that was not even toned to an ash, it was left orange). My hair was so bleached and broken, with patchy sections, and a far stretch from anything we discussed or agreed on. Im at a loss of understand how my hair was original damaged and unable to become what we agreed on but ok enough to be damaged far beyond anything we discussed. And a complete loss of what information I could have possibly given that would lend the dresser to think I wanted or requested to become a blonde. I paid and left (and even tipped). Only to scramble the next day trying to find someone to fix it before I had to meet my future mother in law for lunch. My hair is still falling out, and it's been a borderline tears and stress mixture since. I will not be coming back.

Variable-length problem

Padding

Adding <PAD> tokens to match the longest token in a batch

Variable-length problem

Padding

Adding <PAD> tokens to match the longest token in a batch

Truncation

Cutting longer sequences down to a fixed length

Padding

`sequence length = 4`

“I love my dog”

“I love”

Padding

sequence length = 4

"I love my dog"	→	[1, 2, 3, 4]
"I love"	→	[1, 2, <PAD>, <PAD>]

Truncation

`sequence length = 2`

“I love my dog”

“I love”

`[1, 2]`

Padding has a cost

“I love my dog, he is so playful and smart”

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

“I love”

[1, 2,<PAD>,<PAD>,<PAD>,<PAD>,<PAD>,<PAD>,<PAD>,<PAD>]

Minimize the cost of padding

Bucketing

Group sequences of
similar length
together

Minimize the cost of padding

Bucketing

Group sequences of similar length together

Smart truncation

Preserve the important parts of a long text

Attention masks

Binary tensors that flag actual words to tell them apart from padding

Attention masks

Binary tensors that flag actual words to tell them apart from padding

Sentence: "I love"

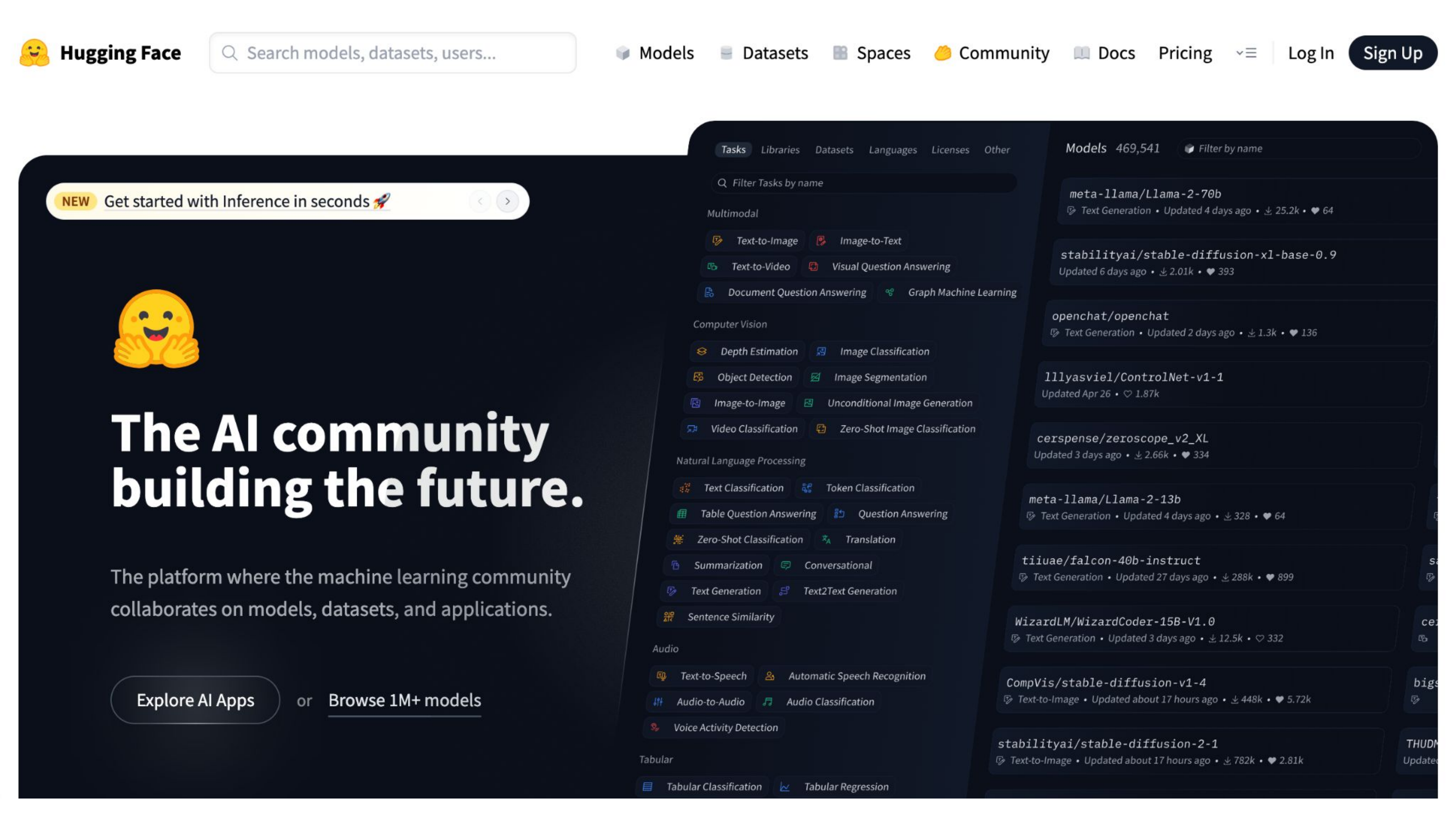
Tokens: [1, 2, <PAD>, <PAD>]

Mask: [1, 1, 0, 0]

Using a Pretrained Tokenizer

Working with text using PyTorch

DeepLearning.AI


Hugging Face
Search models, datasets, users...
Models
Datasets
Spaces
Community
Docs
Pricing
Log In
Sign Up
NEW Get started with Inference in seconds
The AI community building the future.
The platform where the machine learning community collaborates on models, datasets, and applications.
Explore AI Apps
or
Browse 1M+ models
Tasks
Libraries
Datasets
Languages
Licenses
Other
Filter Tasks by name
Multimodal
Text-to-Image
Image-to-Text
Text-to-Video
Visual Question Answering
Document Question Answering
Graph Machine Learning
Computer Vision
Depth Estimation
Image Classification
Object Detection
Image Segmentation
Image-to-Image
Unconditional Image Generation
Video Classification
Zero-Shot Image Classification
Natural Language Processing
Text Classification
Token Classification
Table Question Answering
Question Answering
Zero-Shot Classification
Translation
Summarization
Conversational
Text Generation
Text2Text Generation
Sentence Similarity
Audio
Text-to-Speech
Automatic Speech Recognition
Audio-to-Audio
Audio Classification
Voice Activity Detection
Tabular
Tabular Classification
Tabular Regression
Models 469,541
Filter by name
meta-llama/Llama-2-70b
Text Generation • Updated 4 days ago • 25.2k • 64
stabilityai/stable-diffusion-xl-base-0.9
Updated 6 days ago • 2.01k • 393
openchat/openchat
Text Generation • Updated 2 days ago • 1.3k • 136
lllyasviel/ControlNet-v1-1
Updated Apr 26 • 1.87k
cerspense/zeroscope_v2_XL
Updated 3 days ago • 2.66k • 334
meta-llama/Llama-2-13b
Text Generation • Updated 4 days ago • 328 • 64
tiiuae/falcon-40b-instruct
Text Generation • Updated 27 days ago • 288k • 899
WizardLM/WizardCoder-15B-V1.0
Text Generation • Updated 3 days ago • 12.5k • 332
CompVis/stable-diffusion-v1-4
Text-to-Image • Updated about 17 hours ago • 448k • 5.72k
stabilityai/stable-diffusion-2-1
Text-to-Image • Updated about 17 hours ago • 782k • 2.81k

Product Solutions Resources Open Source Enterprise Pricing

Sign in Sign up

Hugging Face

The AI community building the future.

Verified Sponsor

52.4k followers NYC + Paris https://huggingface.co/ @huggingface

Overview Repositories 339 Projects 6 Packages People 81 Sponsoring 1

Pinned

transformers Public

🤗 Transformers: the model-definition framework for state-of-the-art machine learning models in text, vision, audio, and multimodal models, for both inference and training.

Python 148k 29.9k

diffusers Public

🤗 Diffusers: State-of-the-art diffusion models for image, video, and audio generation in PyTorch and FLAX.

Python 30.2k 6.2k

datasets Public

🤗 The largest hub of ready-to-use datasets for AI models with fast, easy-to-use and efficient data manipulation tools

Python 20.5k 2.9k

peft Public

🤗 PEFT: State-of-the-art Parameter-Efficient Fine-Tuning.

Python 19.3k 2k

accelerate Public

optimum Public

People

View all

Sponsoring

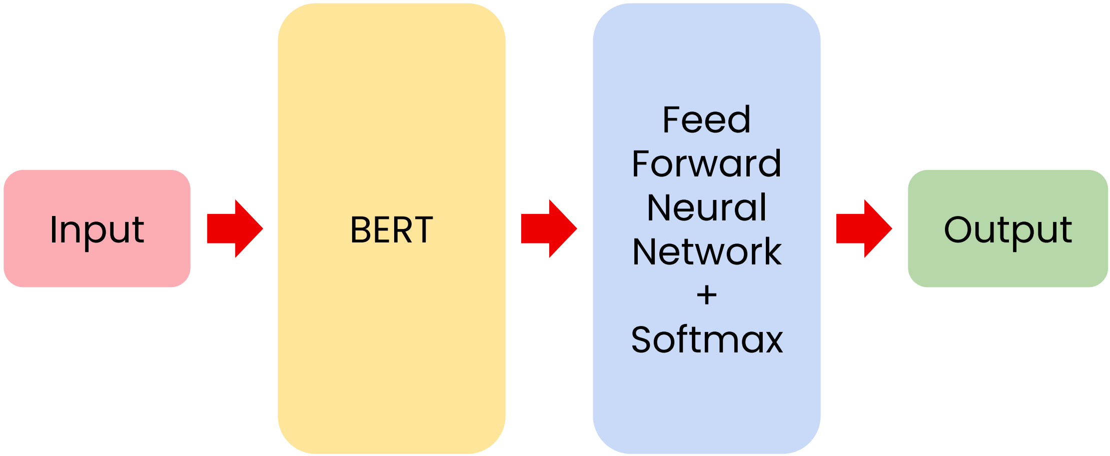
Top languages

BERT: An influential NLP model

BERT: Bidirectional Encoder Representations from Transformers

Trained on:

- Wikipedia
- BooksCorps



BERT: An influential NLP model

"A visually stunning rumination on love"

↓

`A` `visually` `stunning` `rum` `##ination` `on` `love`

BERT: An influential NLP model

"A visually stunning rumination on love"

<CLS>	A	visually	stunning	rum	##ination	on	love	<SEP>

BERT: An influential NLP model

“A visually stunning rumination on love”

<CLS>	A	visually	stunning	rum	##ination	on	love	<SEP>
101	1037	17453	14726	19379	12758	2006	2293	102

```python
from transformers import BertTokenizerFast

sentences = [
    'I love my dog',
    'I love my cat'
]

local_tokenizer_path = "./bert_tokenizer_local"

# If loading directly from Hugging Face server, pass ‘bert-base-uncased’ as an argument
tokenizer = BertTokenizerFast.from_pretrained(local_tokenizer_path)

encoded_inputs = tokenizer(sentences, padding=True,
                           truncation=True, return_tensors='pt')

tokens = [tokenizer.convert_ids_to_tokens(ids)
         for ids in encoded_inputs["input_ids"]]

word_index = tokenizer.get_vocab() # For BertTokenizerFast, get_vocab() returns the vocab
```


Laurence Moroney

```python
from transformers import BertTokenizerFast

sentences = [
    'I love my dog',
    'I love my cat'
]

local_tokenizer_path = "./bert_tokenizer_local"

# If loading directly from Hugging Face server, pass ‘bert-base-uncased’ as an argument
tokenizer = BertTokenizerFast.from_pretrained(local_tokenizer_path)

encoded_inputs = tokenizer(sentences, padding=True,
                           truncation=True, return_tensors='pt')

tokens = [tokenizer.convert_ids_to_tokens(ids)
          for ids in encoded_inputs["input_ids"]]

word_index = tokenizer.get_vocab() # For BertTokenizerFast, get_vocab() returns the vocab
```

```
from transformers import BertTokenizerFast

sentences = [
    'I love my dog',
    'I love my cat'
]

local_tokenizer_path = "./bert_tokenizer_local"

# If loading directly from Hugging Face server, pass ‘bert-base-uncased’ as an argument
tokenizer = BertTokenizerFast.from_pretrained(local_tokenizer_path)

encoded_inputs = tokenizer(sentences, padding=True,
                           truncation=True, return_tensors='pt')

tokens = [tokenizer.convert_ids_to_tokens(ids)
          for ids in encoded_inputs["input_ids"]]

word_index = tokenizer.get_vocab() # For BertTokenizerFast, get_vocab() returns the vocab
```

```python
from transformers import BertTokenizerFast

sentences = [
    'I love my dog',
    'I love my cat'
]

local_tokenizer_path = "./bert_tokenizer_local"

# If loading directly from Hugging Face server, pass ‘bert-base-uncased’ as an argument
tokenizer = BertTokenizerFast.from_pretrained(local_tokenizer_path)

encoded_inputs = tokenizer(sentences, padding=True,
                          truncation=True, return_tensors='pt')

tokens = [tokenizer.convert_ids_to_tokens(ids)
          for ids in encoded_inputs["input_ids"]]

word_index = tokenizer.get_vocab() # For BertTokenizerFast, get_vocab() returns the vocab
```

```python
from transformers import BertTokenizerFast

sentences = [
   'I love my dog',
   'I love my cat'
]

local_tokenizer_path = "./bert_tokenizer_local"

# If loading directly from Hugging Face server, pass ‘bert-base-uncased’ as an argument
tokenizer = BertTokenizerFast.from_pretrained(local_tokenizer_path)

encoded_inputs = tokenizer(sentences, padding=True,
                          truncation=True, return_tensors='pt')

tokens = [tokenizer.convert_ids_to_tokens(ids)
         for ids in encoded_inputs["input_ids"]]

word_index = tokenizer.get_vocab() # For BertTokenizerFast, get_vocab() returns the vocab
```

```
from transformers import BertTokenizerFast

sentences = [
   'I love my dog',
   'I love my cat'
]

local_tokenizer_path = "./bert_tokenizer_local"

# If loading directly from Hugging Face server, pass 'bert-base-uncased' as an argument
tokenizer = BertTokenizerFast.from_pretrained(local_tokenizer_path)

encoded_inputs = tokenizer(sentences, padding=True,
                          truncation=True, return_tensors='pt')

tokens = [tokenizer.convert_ids_to_tokens(ids)
         for ids in encoded_inputs["input_ids"]]

word_index = tokenizer.get_vocab() # For BertTokenizerFast, get_vocab() returns the vocab
```

Using a pretrained tokenizer

```
# Print the human-readable `tokens` for each sentence
print("Tokens:", tokens)

print("\nToken IDs:", encoded_inputs['input_ids'])

# Print unique tokens from your sentences mapped to their unique IDs
helper_utils.print_unique_token_id_mappings(tokens, encoded_inputs['input_ids'])
```

Using a pretrained tokenizer

Output

```
Tokens: [['[CLS]', 'i', 'love', 'my', 'dog', '[SEP]'], ['[CLS]', 'i', 'love', 'my', 'cat',
'[SEP]']]

Token IDs: tensor([[ 101, 1045, 2293, 2026, 3899,  102],
        [ 101, 1045, 2293, 2026, 4937,  102]])

--- Unique Token to ID Mappings (for these sentences) ---
[CLS] -->   101
[SEP] -->   102
i     -->   1045
my    -->   2026
love  -->   2293
dog   -->   3899
cat   -->   4937
```

DeepLearning.AI

Tensorization

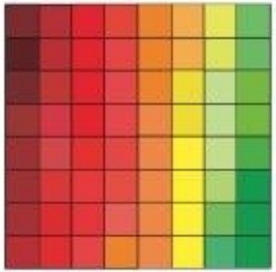
Working with text using PyTorch

Tensors represent batches of text

1D

Tensors represent batches of text

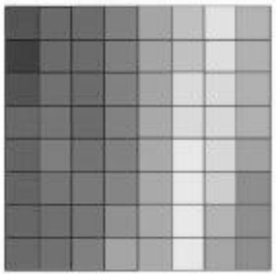
1D



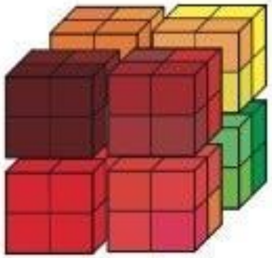
2D

Tensors represent batches of text

1D



2D



3D

Tensorization

"I love my dog" → [1, 2, 3, 4]

```
import torch

tokens = torch.tensor([1, 2, 3, 4])
```

Tensorization

"I love my dog"	→	[1, 2, 3, 4]
"I love my cat"	→	[1, 2, 3, 5]
"I like having pets"	→	[1, 6, 7, 8]

Tensorization

"I love my dog" → [1, 2, 3, 4]

"I love my cat" → [1, 2, 3, 5]

"I like having pets" → [1, 6, 7, 8]

```
[
  [1, 2, 3, 4],
  [1, 2, 3, 5],
  [1, 6, 7, 8]
]
```

One-hot encoding

Words are represented as a vector with all zeros except in one position

One-hot encoding

Vocabulary:

	'i'	'love'	'my'	'dog'	'cat'
ID:	1	2	3	4	5

"**dog**"   [0 0 0 1 0]

Words are represented as a vector with all zeros except in one position

Bag of words: Count

Vocabulary:

	'i'	'love'	'my'	'dog'	'cat'
ID:	1	2	3	4	5

"I **love love** my dog"	[1 2 1 1 0]
"I love my cat"	[1 1 1 0 1]

DeepLearning.AI

Laurence Moroney

Bag of words: Count

Vocabulary:

	'i'	'love'	'my'	'dog'	'cat'
ID:	1	2	3	4	5

"I love love my dog"	[1	2	1	1	0]
"I love my cat"	[1	1	1	0	1]
"my cat I love"	[1	1	1	0	1]

Term frequency-inverse document frequency

TF	→	How often words appear in one document
IDF	→	Down-weighs words that appear in many documents

$$TF(t,d) = \frac{number\ of\ times\ t\ appears\ in\ d}{total\ number\ of\ terms\ in\ d}$$

$$IDF(t) = log\frac{N}{1+df}$$

$$TF-IDF(t,d) = TF(t,d) * IDF(t)$$

```
t: term
d: document
N: total number of documents
df: number of documents containing t
```

DeepLearning.AI

Laurence Moroney

Term frequency-inverse document frequency

Vocabulary:

	'i'	'love'	'my'	'dog'	'cat'
ID:	1	2	3	4	5

"I love love my dog"	[0	0	0	0.139	0]
"I love my cat"	[0	0	0	0	0.173]

DeepLearning.AI

Introduction to Embeddings

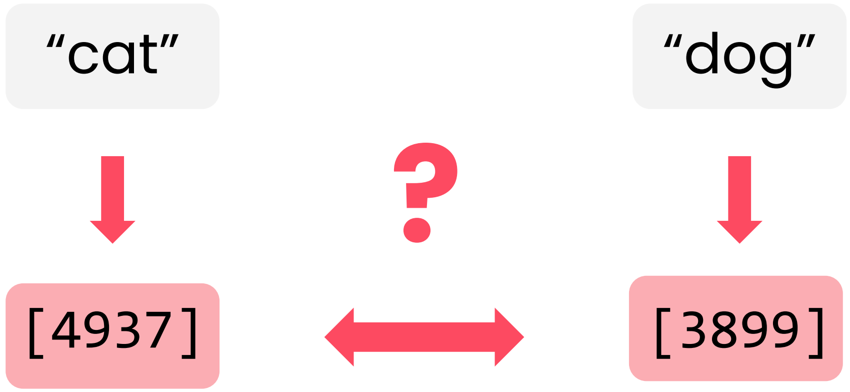
Working with text using PyTorch

Token ID’s don’t carry any meaning

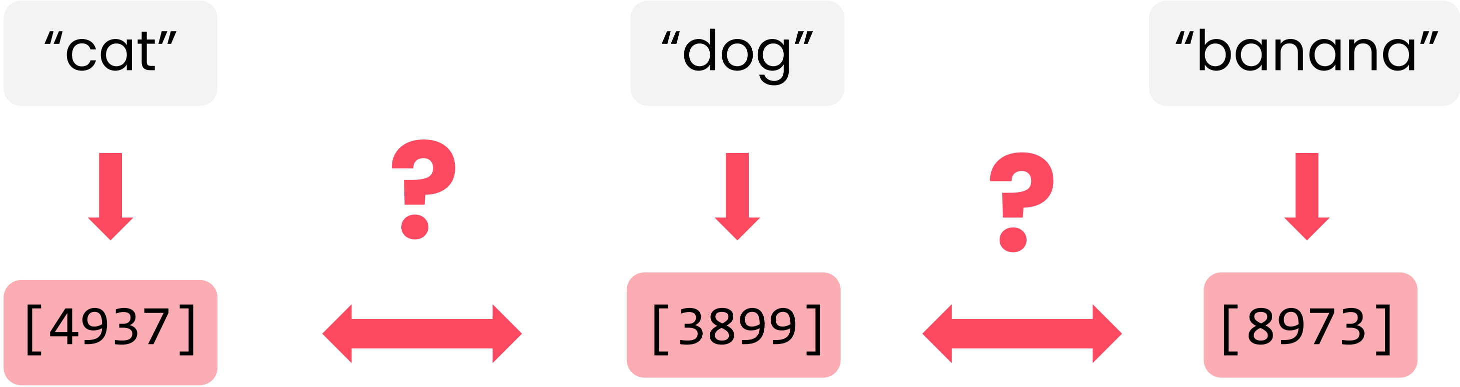
“cat” → [4937]

“dog” → [3899]

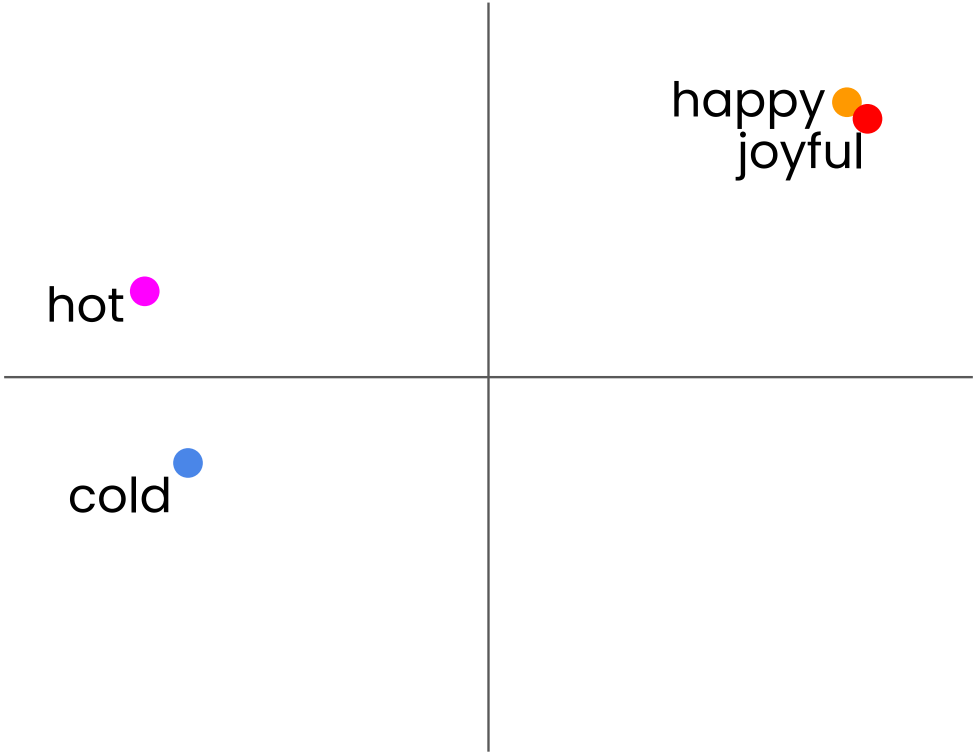
Token ID's don't carry any meaning



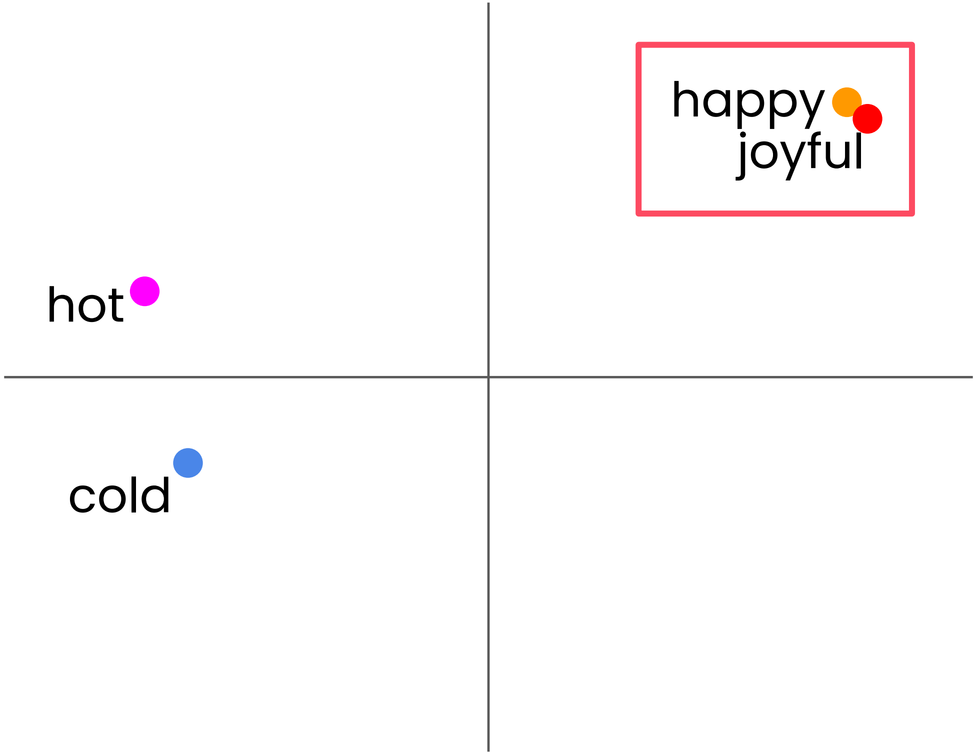
Token ID’s don’t carry any meaning



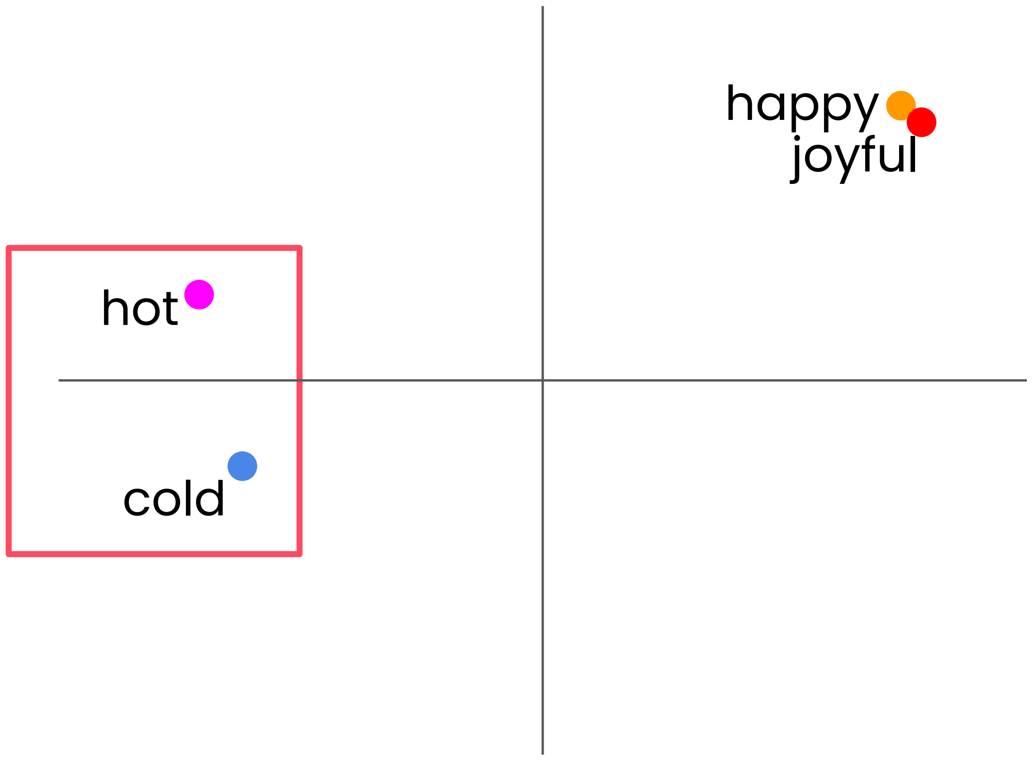
Embeddings show how words are related



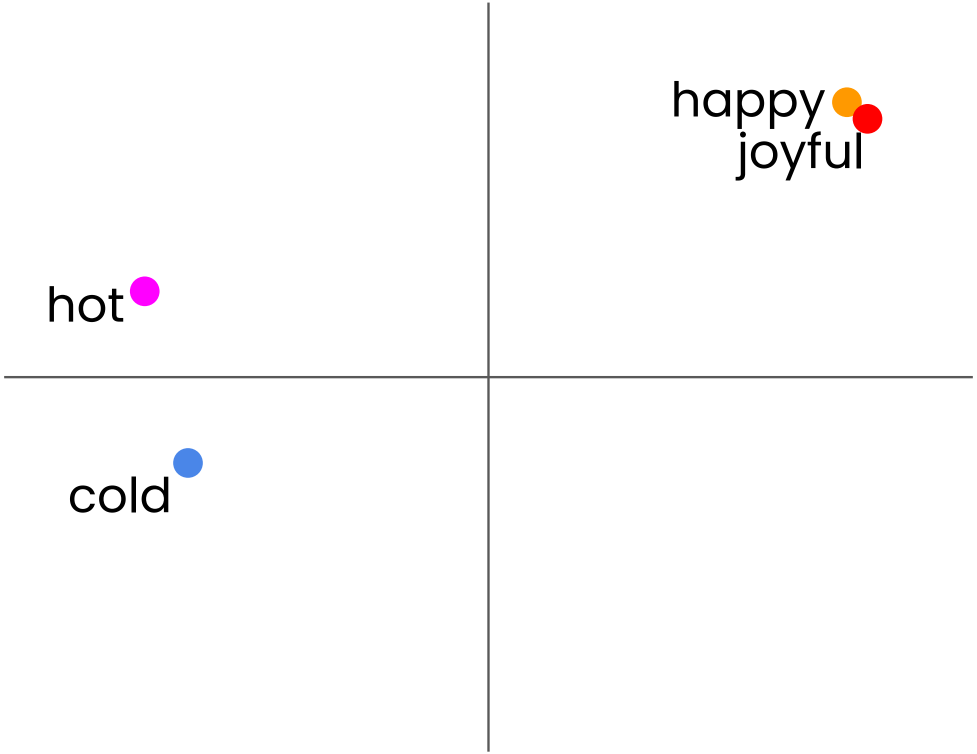
Embeddings show how words are related



Embeddings show how words are related



Embeddings show how words are related



Embeddings vs. one-hot vectors

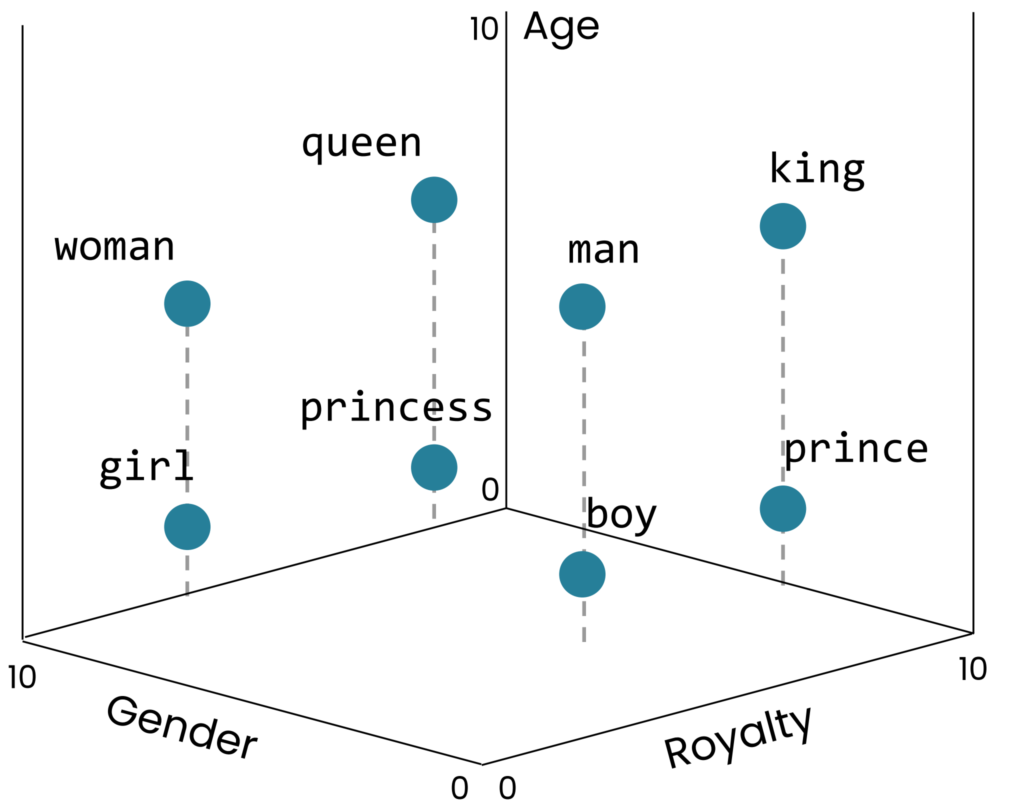
	boy	prince	king	girl	princess	dog	queen	cat
"king"	0	0	1	0	0	0	0	0
"queen"	0	0	0	0	0	0	1	0
"prince"	0	1	0	0	0	0	0	0
"princess"	0	0	0	0	1	0	0	0

Embeddings vs. one-hot vectors

	boy	prince	king	girl	princess	dog	queen	cat
“king” →	[0,	0,	1,	0,	0,	0,	0,	0]
“queen” →	[0,	0,	0,	0,	0,	0,	1,	0]
“prince” →	[0,	1,	0,	0,	0,	0,	0,	0]
“princess” →	[0,	0,	0,	0,	1,	0,	0,	0]

But how are they related?

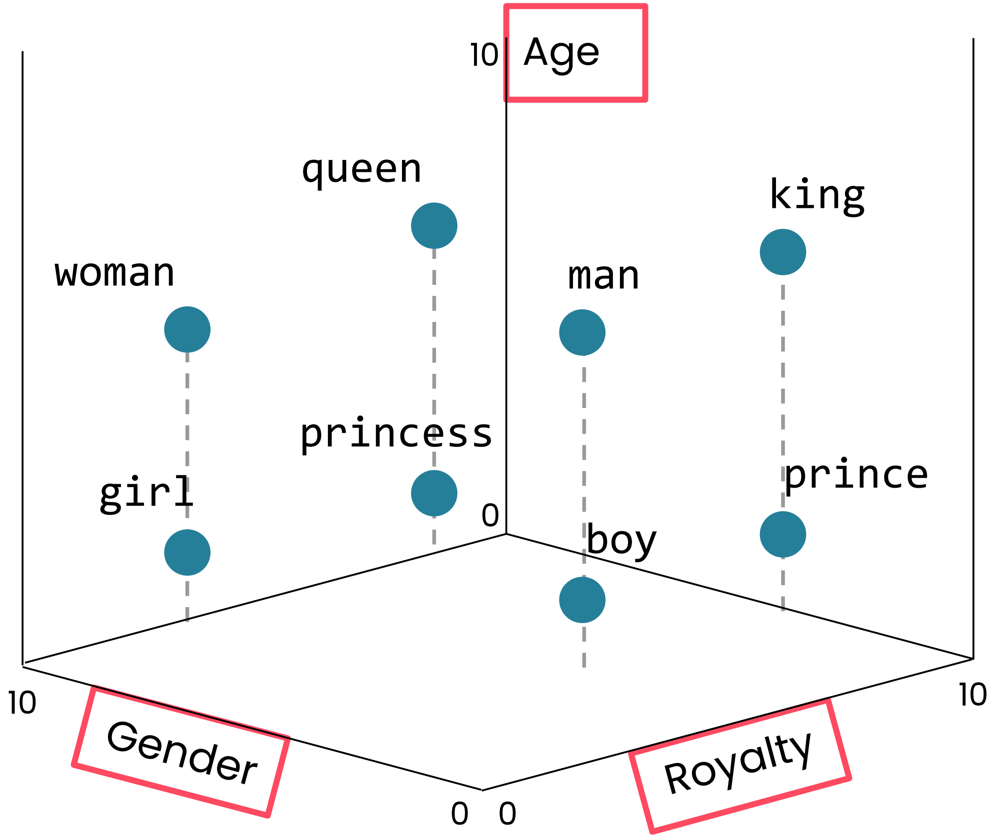
Embeddings provide a map of meaning



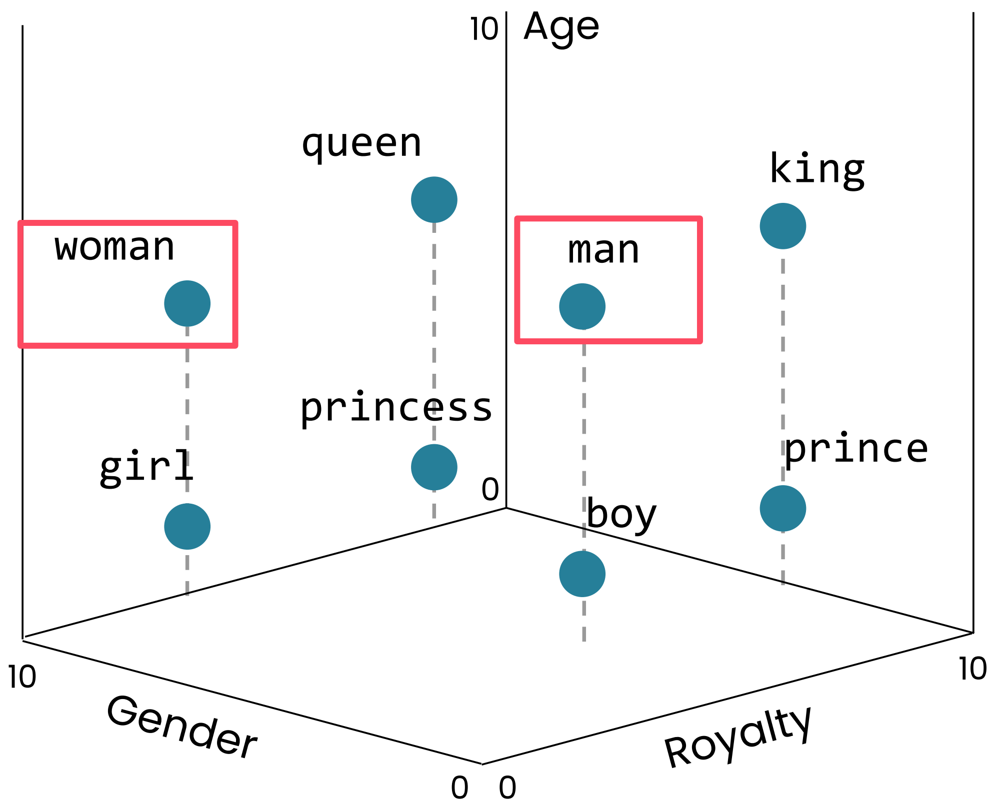
DeepLearning.AI

Laurence Moroney

Embeddings provide a map of meaning



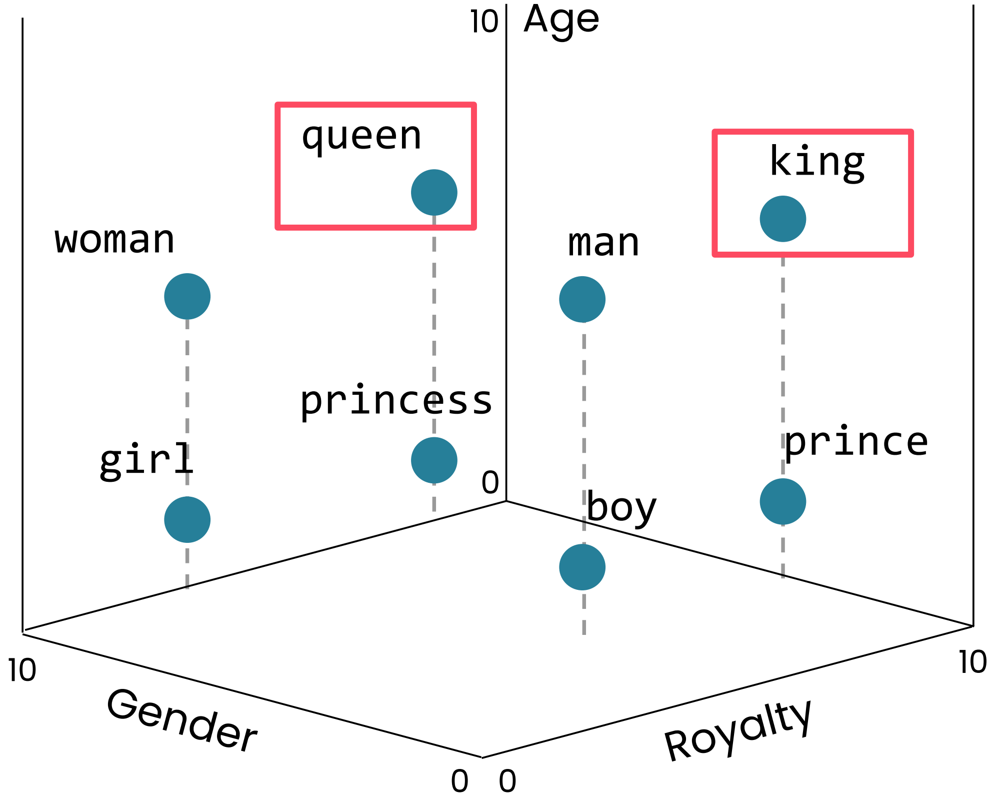
Embeddings provide a map of meaning



DeepLearning.AI

Laurence Moroney

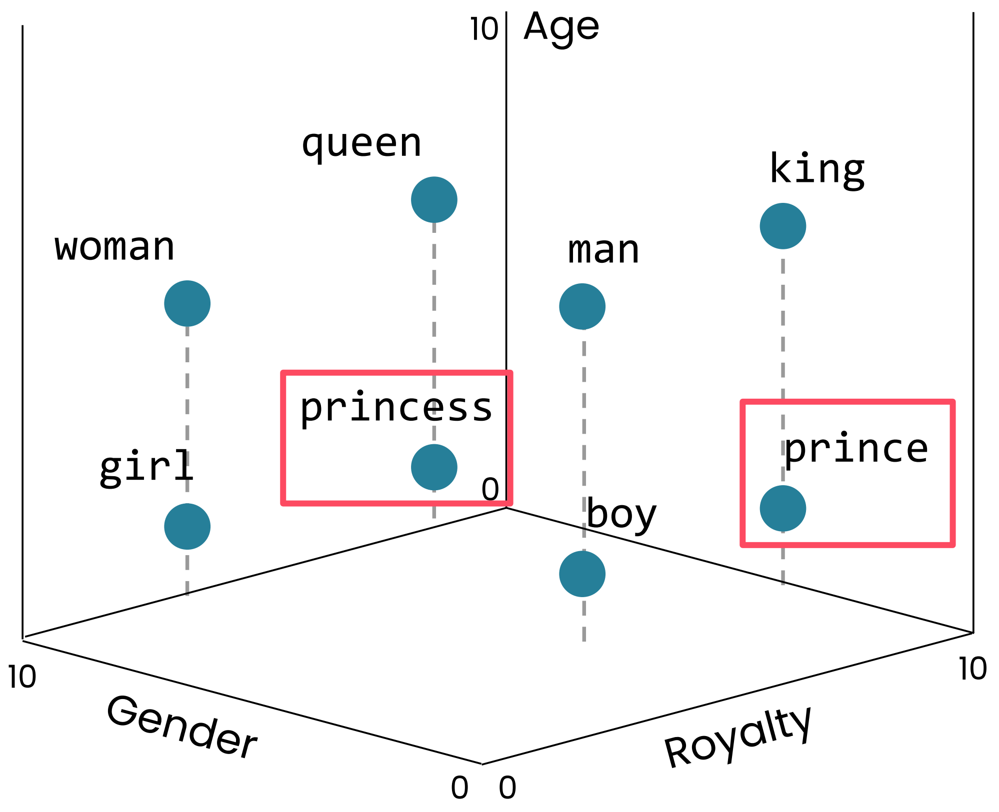
Embeddings provide a map of meaning



DeepLearning.AI

Laurence Moroney

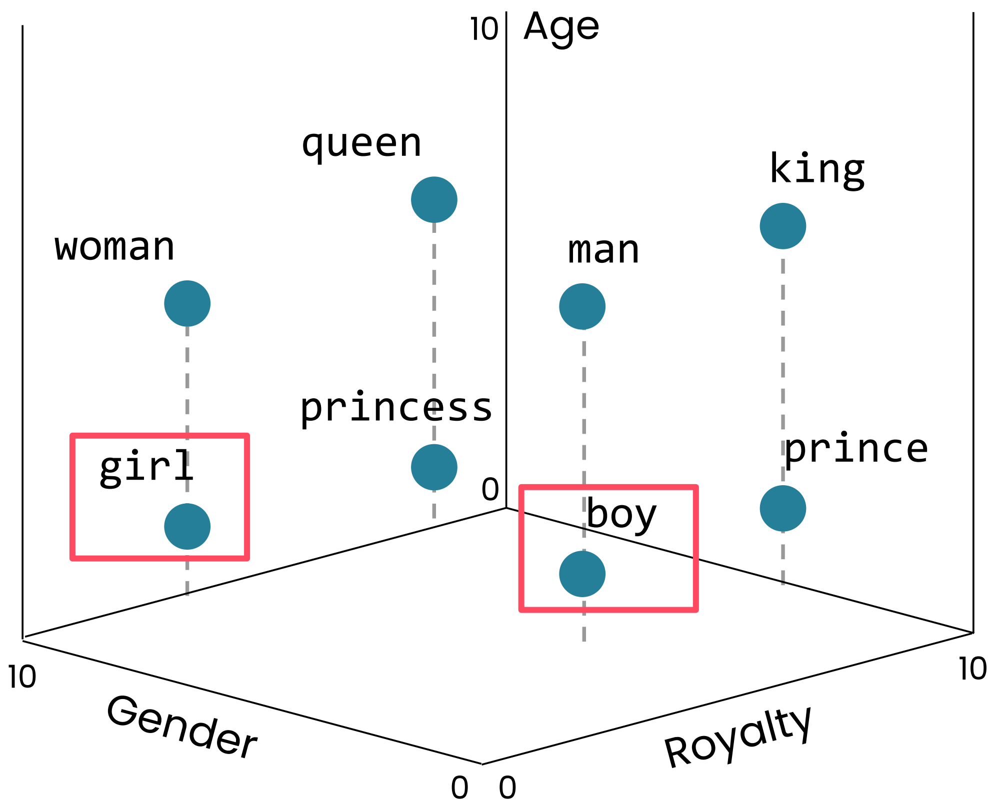
Embeddings provide a map of meaning



DeepLearning.AI

Laurence Moroney

Embeddings provide a map of meaning



DeepLearning.AI

Laurence Moroney

Operations on embedding vectors

`king` - `man`

Operations on embedding vectors

`king` - `man` + `woman`

Operations on embedding vectors

`king` - `man` + `woman` ≈ `queen`

DeepLearning.AI

Laurence Moroney

Operations on embedding vectors

```
king  = [4.2, 1.8, 7.5]
man   = [2.1, 0.9, 5.0]
queen = [4.2, 2.1, 7.4]
woman = [2.0, 1.0, 5.1]
```

Operations on embedding vectors

```
king - man = [4.2 - 2.1, 1.8 - 0.9, 7.5 - 5.0]

           = [2.1, 0.9, 2.5]
```

Operations on embedding vectors

`king` - `man` + `woman` = `[2.1 + 2.0, 0.9 + 1.0, 2.5 + 5.1]`

= `[4.1, 1.9, 7.6]`

Operations on embedding vectors

`king` - `man` + `woman` = [2.1 + 2.0, 0.9 + 1.0, 2.5 + 5.1]

= [4.1, 1.9, 7.6]

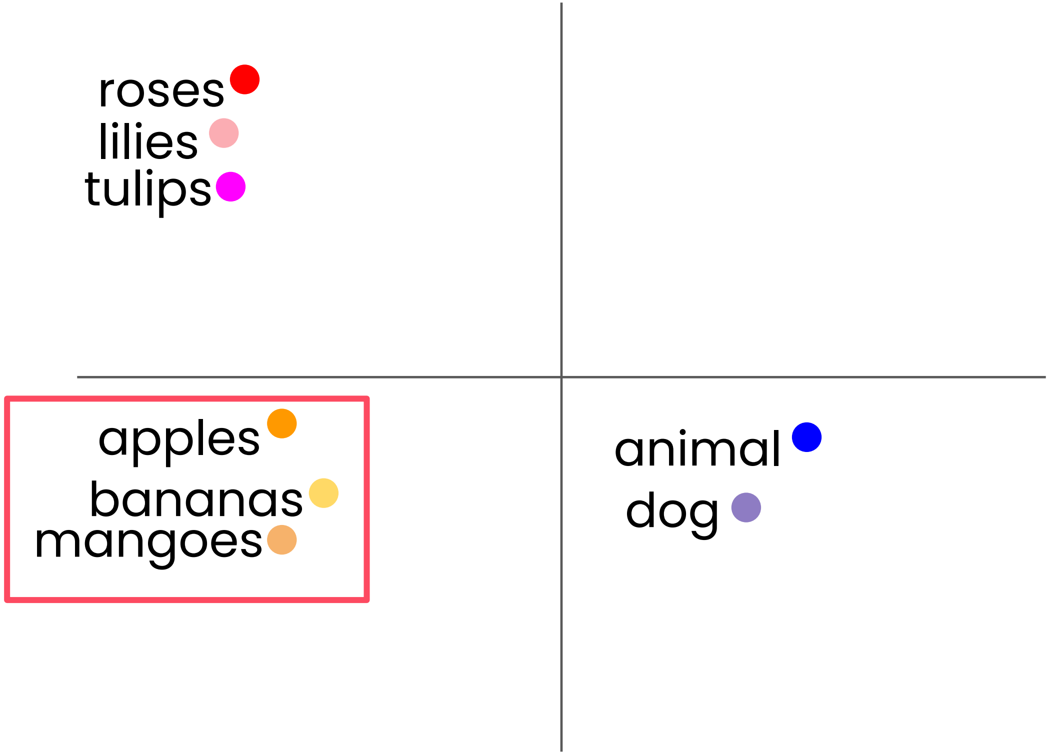
≈

`queen` = [4.2, 2.1, 7.4]

Distributional hypothesis

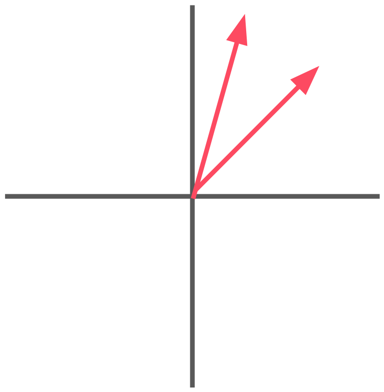
Distributional hypothesis

Distributional hypothesis

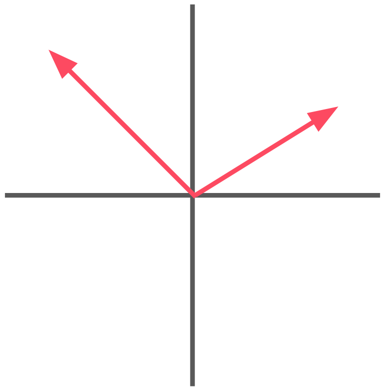


Distributional hypothesis

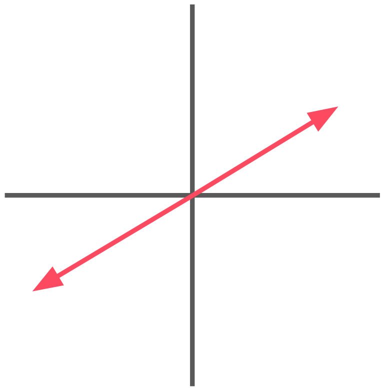
Cosine similarity



Cosine ≈ 1
Similar



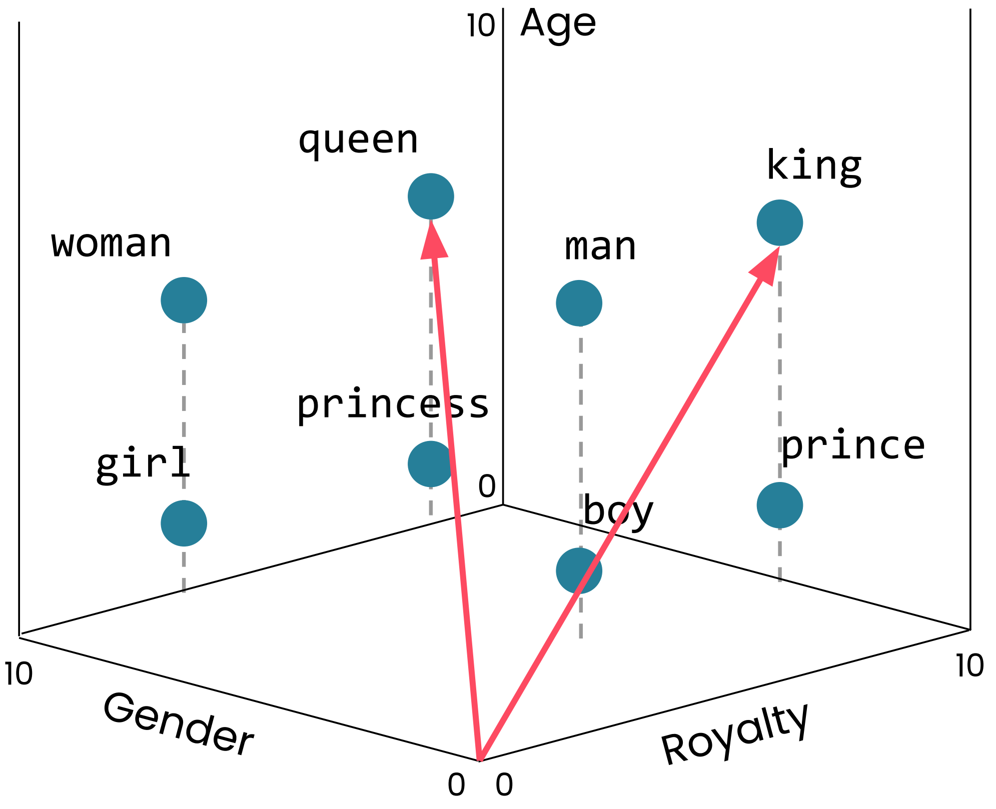
Cosine ≈ 0
Orthogonal



Cosine ≈ −1
Opposite

$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A||\ ||B||}$$

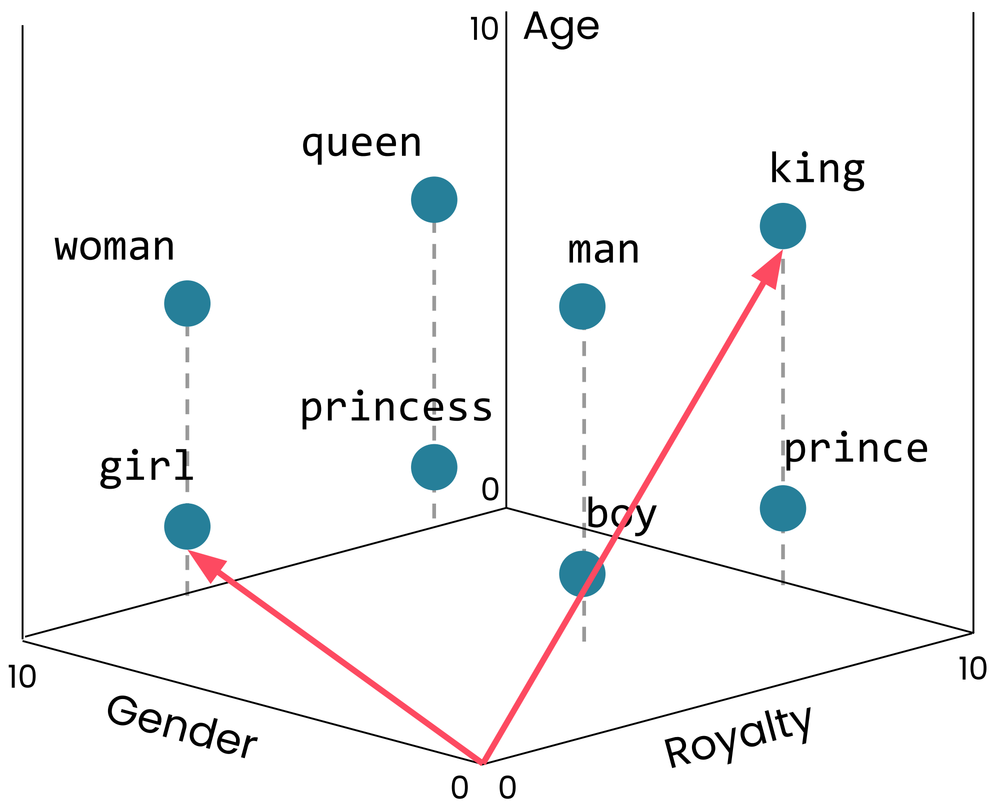
Cosine similarity



DeepLearning.AI

Laurence Moroney

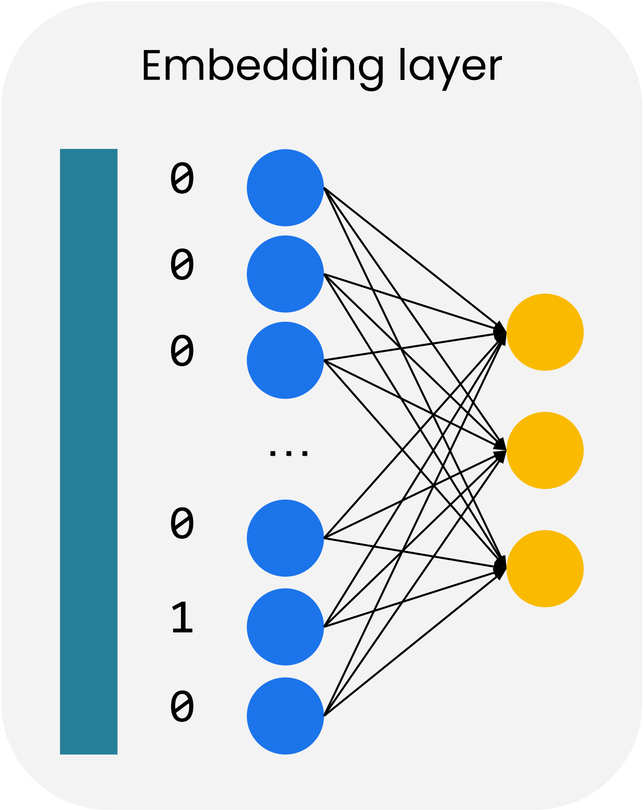
Cosine similarity



DeepLearning.AI

Laurence Moroney

Embedding layers

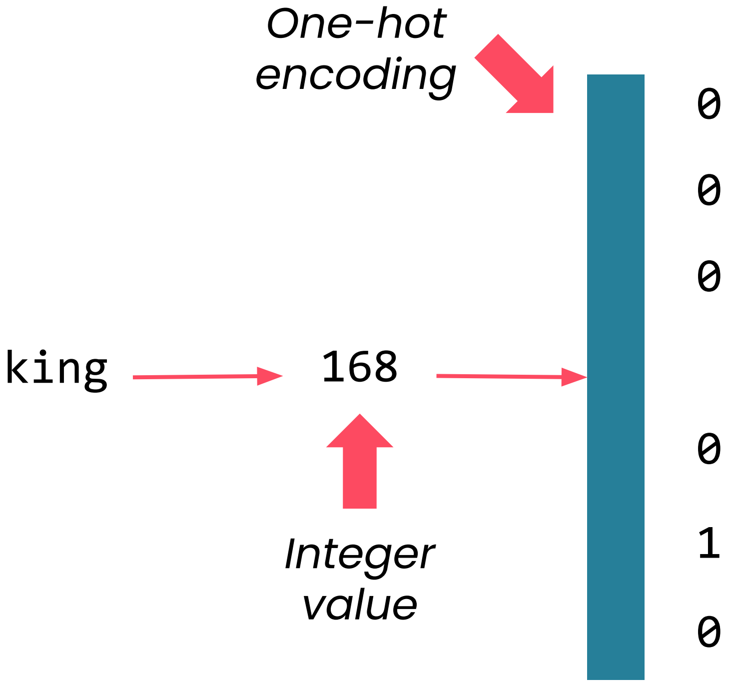


Embedding layers

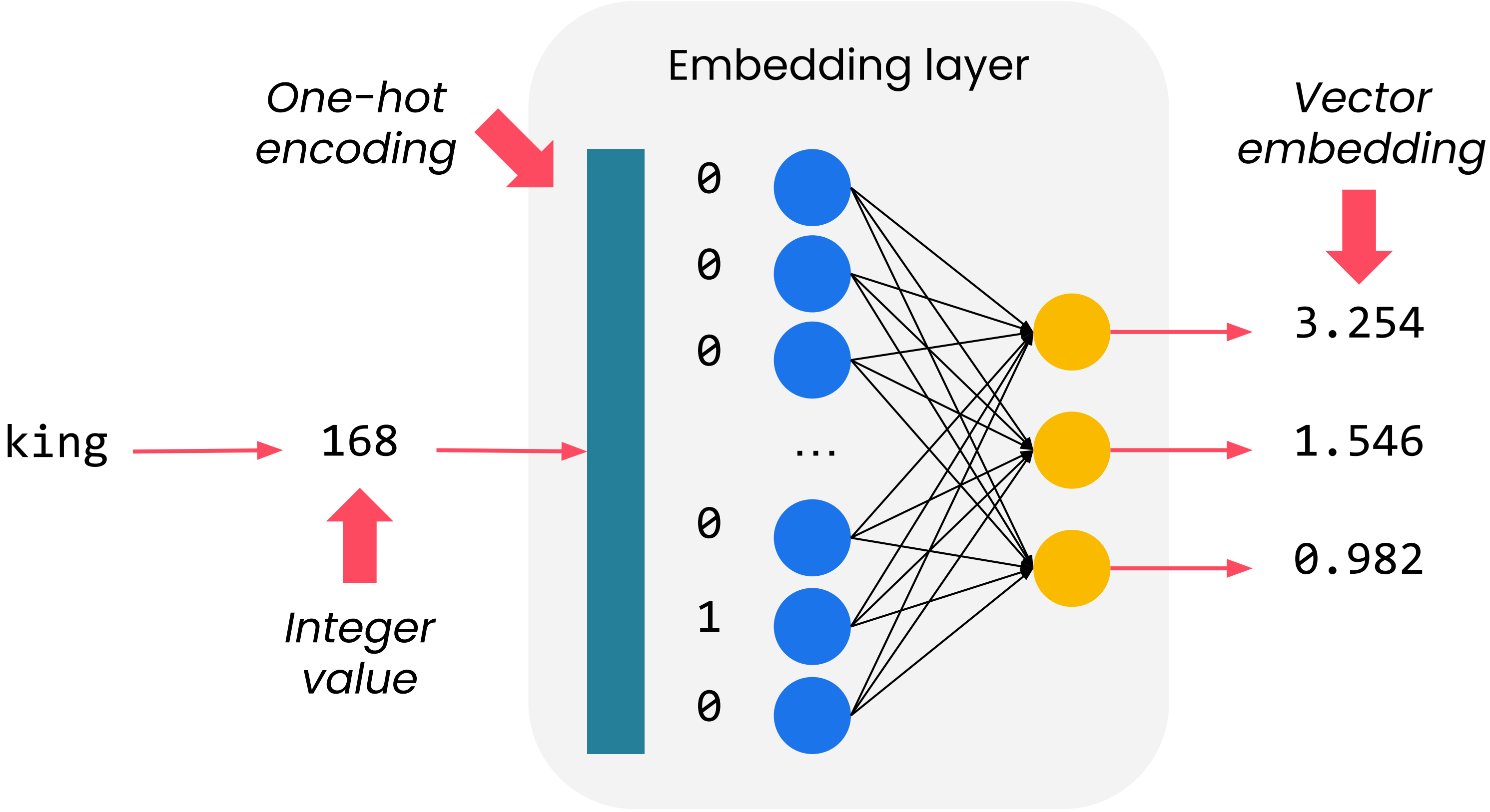
king → 168

Integer value

Embedding layers



Embedding layers



Static embeddings

They map each word in the vocabulary to one fixed vector

Static embeddings

They map each word in the vocabulary to one fixed vector

“flying **bat**”

“baseball **bat**”

`bat` = `[0.32, 2.24, 6.30, 0.76]`

Static embedding models

Word2Vec

GloVe

FastText

Contextual embeddings

They assign embedding vectors based on context

Contextual embeddings

They assign embedding vectors based on context

"flying **bat**" → bat = [0.12, 2.65, 6.27, 0.56]

"baseball **bat**" → bat = [0.09, 2.51, 6.90, 0.42]

Contextual embedding models

BERT

GPT

ELMo

Laurence Moroney

Trade-offs with contextual embeddings

Pros

- Catch subtle differences
- Good for sentiment analysis, NER and translation

Trade-offs with contextual embeddings

Pros

- Catch subtle differences
- Good for sentiment analysis, NER and translation

Cons

- Computationally heavier
- Compute new embeddings for every sentence

DeepLearning.AI

Implementing Embeddings in PyTorch

Working with text using PyTorch

GloVe: A static embedding model

Stands for Global Vectors for Word Representation

GloVe: A static embedding model

- Stands for Global Vectors for Word Representation
- Learns by looking at how often words appear together

GloVe: A static embedding model

- Stands for Global Vectors for Word Representation
- Learns by looking at how often words appear together
- GloVe 6B: 6 billion tokens from Wikipedia and Gigaword

Using GloVe in PyTorch

```
# Download the data for the GloVe 6B 100d model
helper_utils.download_glove6B()

# Specify the path to the 100d GloVe file
glove_file = './glove_data/glove.6B.100d.txt'

# Load the pre-trained word vectors from the file
glove_embeddings = helper_utils.load_glove_embeddings(glove_file)
```

Using GloVe in PyTorch

```
# Download the data for the GloVe 6B 100d model
helper_utils.download_glove6B()

# Specify the path to the 100d GloVe file
glove_file = './glove_data/glove.6B.100d.txt'

# Load the pre-trained word vectors from the file
glove_embeddings = helper_utils.load_glove_embeddings(glove_file)
```

Using GloVe in PyTorch

```
# Download the data for the GloVe 6B 100d model
helper_utils.download_glove6B()

# Specify the path to the 100d GloVe file
glove_file = './glove_data/glove.6B.100d.txt'

# Load the pre-trained word vectors from the file
glove_embeddings = helper_utils.load_glove_embeddings(glove_file)
```

```python
def find_closest_words(embedding, embeddings_dict, exclude_words=[], top_n=5):

    filtered_words = [word for word in embeddings_dict.keys() if word not in exclude_words]

    if not filtered_words:
        return None

    embedding_matrix = np.array([embeddings_dict[word] for word in filtered_words])

    target_embedding = embedding.reshape(1, -1)

    similarity_scores = cosine_similarity(target_embedding, embedding_matrix)

    closest_word_indices = np.argsort(similarity_scores[0])[::-1][:top_n]

    return [(filtered_words[i], similarity_scores[0][i]) for i in closest_word_indices]
```

```
def find_closest_words(embedding, embeddings_dict, exclude_words=[], top_n=5):
    filtered_words = [word for word in embeddings_dict.keys() if word not in exclude_words]

    if not filtered_words:
        return None

    embedding_matrix = np.array([embeddings_dict[word] for word in filtered_words])

    target_embedding = embedding.reshape(1, -1)

    similarity_scores = cosine_similarity(target_embedding, embedding_matrix)

    closest_word_indices = np.argsort(similarity_scores[0])[::-1][:top_n]

    return [(filtered_words[i], similarity_scores[0][i]) for i in closest_word_indices]
```

```python
def find_closest_words(embedding, embeddings_dict, exclude_words=[], top_n=5):

    filtered_words = [word for word in embeddings_dict.keys() if word not in exclude_words]

    if not filtered_words:
        return None

    embedding_matrix = np.array([embeddings_dict[word] for word in filtered_words])

    target_embedding = embedding.reshape(1, -1)

    similarity_scores = cosine_similarity(target_embedding, embedding_matrix)

    closest_word_indices = np.argsort(similarity_scores[0])[::-1][:top_n]

    return [(filtered_words[i], similarity_scores[0][i]) for i in closest_word_indices]
```

```python
def find_closest_words(embedding, embeddings_dict, exclude_words=[], top_n=5):

    filtered_words = [word for word in embeddings_dict.keys() if word not in exclude_words]

    if not filtered_words:
        return None

    embedding_matrix = np.array([embeddings_dict[word] for word in filtered_words])

    target_embedding = embedding.reshape(1, -1)

    similarity_scores = cosine_similarity(target_embedding, embedding_matrix)

    closest_word_indices = np.argsort(similarity_scores[0])[::-1][:top_n]

    return [(filtered_words[i], similarity_scores[0][i]) for i in closest_word_indices]
```

Using GloVe in PyTorch

```
if all(word in glove_embeddings for word in ['king', 'man', 'woman']):
   king = glove_embeddings['king']
   man = glove_embeddings['man']
   woman = glove_embeddings['woman']

result_embedding = king - man + woman

top_n = 5

closest_words_with_scores = find_closest_words(
   result_embedding,
   glove_embeddings,
   exclude_words=['king', 'man', 'woman'],
   top_n=top_n
)
```

Using GloVe in PyTorch

```
if all(word in glove_embeddings for word in ['king', 'man', 'woman']):
    king = glove_embeddings['king']
    man = glove_embeddings['man']
    woman = glove_embeddings['woman']

result_embedding = king - man + woman

top_n = 5

closest_words_with_scores = find_closest_words(
    result_embedding,
    glove_embeddings,
    exclude_words=['king', 'man', 'woman'],
    top_n=top_n
)
```

Using GloVe in PyTorch

```
if all(word in glove_embeddings for word in ['king', 'man', 'woman']):
    king = glove_embeddings['king']
    man = glove_embeddings['man']
    woman = glove_embeddings['woman']

result_embedding = king - man + woman

top_n = 5

closest_words_with_scores = find_closest_words(
    result_embedding,
    glove_embeddings,
    exclude_words=['king', 'man', 'woman'],
    top_n=top_n
)
```

Using GloVe in PyTorch

```
if all(word in glove_embeddings for word in ['king', 'man', 'woman']):
   king = glove_embeddings['king']
   man = glove_embeddings['man']
   woman = glove_embeddings['woman']

result_embedding = king - man + woman

top_n = 5

closest_words_with_scores = find_closest_words(
   result_embedding,
   glove_embeddings,
   exclude_words=['king', 'man', 'woman'],
   top_n=top_n
)
```

Using GloVe in PyTorch

Output

```
king - man + woman ≈ queen (Score: 0.7834)

--- Other Top 4 Results ---
monarch (Score: 0.6934)
throne (Score: 0.6833)
daughter (Score: 0.6809)
prince (Score: 0.6713)
```

Static embeddings can't handle polysemy

"A **bat** flew out of the cave"

Static embeddings can't handle polysemy

"A **bat** flew out of the cave" →

"He swung the baseball **bat**" →

Laurence Moroney

GloVe produces the same vector for ‘bat’

```python
sentence1 = "A bat flew out of the cave."
sentence2 = "He swung the baseball bat."

bat_from_sentence1 = glove_embeddings["bat"]
bat_from_sentence2 = glove_embeddings["bat"]

are_identical = np.array_equal(bat_from_sentence1, bat_from_sentence2)
print(f"Are the vectors for 'bat' from each sentence identical? {are_identical}")
```

GloVe produces the same vector for ‘bat’

```
sentence1 = "A bat flew out of the cave."
sentence2 = "He swung the baseball bat."

bat_from_sentence1 = glove_embeddings["bat"]
bat_from_sentence2 = glove_embeddings["bat"]

are_identical = np.array_equal(bat_from_sentence1, bat_from_sentence2)
print(f"Are the vectors for 'bat' from each sentence identical? {are_identical}")
```

GloVe produces the same vector for 'bat'

```
sentence1 = "A bat flew out of the cave."
sentence2 = "He swung the baseball bat."

bat_from_sentence1 = glove_embeddings["bat"]
bat_from_sentence2 = glove_embeddings["bat"]

are_identical = np.array_equal(bat_from_sentence1, bat_from_sentence2)
print(f"Are the vectors for 'bat' from each sentence identical? {are_identical}")
```

GloVe produces the same vector for ‘bat’

```python
sentence1 = "A bat flew out of the cave."
sentence2 = "He swung the baseball bat."

bat_from_sentence1 = glove_embeddings["bat"]
bat_from_sentence2 = glove_embeddings["bat"]

are_identical = np.array_equal(bat_from_sentence1, bat_from_sentence2)
print(f"Are the vectors for 'bat' from each sentence identical? {are_identical}")
```

Output

```
Are the vectors for 'bat' from each sentence identical? True
```

Using a contextual embedding model: BERT

```
helper_utils.download_bert()
bert_path = './bert_model'
tokenizer, model_bert = helper_utils.load_bert(bert_path)
```

Using a contextual embedding model: BERT

```
# --- Process and Print Vectors for Sentence 1 ---
print("--- Sentence 1 (first 5 values) ---")
inputs1 = tokenizer(sentence1, return_tensors='pt')
with torch.no_grad():
   outputs1 = model_bert(**inputs1)
last_hidden_state1 = outputs1.last_hidden_state[0] # Embeddings for all tokens

tokens1 = tokenizer.convert_ids_to_tokens(inputs1['input_ids'][0])
```

Using a contextual embedding model: BERT

```
# --- Process and Print Vectors for Sentence 1 ---
print("--- Sentence 1 (first 5 values) ---")
inputs1 = tokenizer(sentence1, return_tensors='pt')
with torch.no_grad():
    outputs1 = model_bert(**inputs1)
last_hidden_state1 = outputs1.last_hidden_state[0] # Embeddings for all tokens

tokens1 = tokenizer.convert_ids_to_tokens(inputs1['input_ids'][0])
```

Using a contextual embedding model: BERT

```
# --- Process and Print Vectors for Sentence 1 ---
print("--- Sentence 1 (first 5 values) ---")
inputs1 = tokenizer(sentence1, return_tensors='pt')
with torch.no_grad():
    outputs1 = model_bert(**inputs1)
last_hidden_state1 = outputs1.last_hidden_state[0] # Embeddings for all tokens

tokens1 = tokenizer.convert_ids_to_tokens(inputs1['input_ids'][0])
```

Using a contextual embedding model: BERT

```
# --- Process and Print Vectors for Sentence 1 ---
print("--- Sentence 1 (first 5 values) ---")
inputs1 = tokenizer(sentence1, return_tensors='pt')
with torch.no_grad():
    outputs1 = model_bert(**inputs1)
last_hidden_state1 = outputs1.last_hidden_state[0] # Embeddings for all tokens

tokens1 = tokenizer.convert_ids_to_tokens(inputs1['input_ids'][0])
```

Using a contextual embedding model: BERT

```
# --- Process and Print Vectors for Sentence 2 ---
print("--- Sentence 2 (first 5 values) ---")
inputs2 = tokenizer(sentence2, return_tensors='pt')
with torch.no_grad():
    outputs2 = model_bert(**inputs2)
last_hidden_state2 = outputs2.last_hidden_state[0] # Embeddings for all tokens

tokens2 = tokenizer.convert_ids_to_tokens(inputs2['input_ids'][0])
```

BERT produces contextual embeddings

```
bat_animal_vector = last_hidden_state1[2].numpy()
bat_sport_vector = last_hidden_state2[5].numpy()
are_identical = np.array_equal(bat_animal_vector, bat_sport_vector)
print(f"Are the contextual BERT vectors for 'bat' identical? {are_identical}")
```

BERT produces contextual embeddings

```
bat_animal_vector = last_hidden_state1[2].numpy()
bat_sport_vector = last_hidden_state2[5].numpy()
are_identical = np.array_equal(bat_animal_vector, bat_sport_vector)
print(f"Are the contextual BERT vectors for 'bat' identical? {are_identical}")
```

Output

```
Are the contextual BERT vectors for 'bat' identical? False
```

When to use static vs. contextual?

Static embeddings

Simple tasks where speed and memory efficiency are prioritized

When to use static vs. contextual?

Static embeddings

Simple tasks where speed and memory efficiency are prioritized

Contextual embeddings

Advanced tasks where subtle differences are critical

Visualization techniques for embeddings

PCA
Principal Component Analysis

Efficient, but can distort local relationships

Visualization techniques for embeddings

PCA
Principal Component Analysis

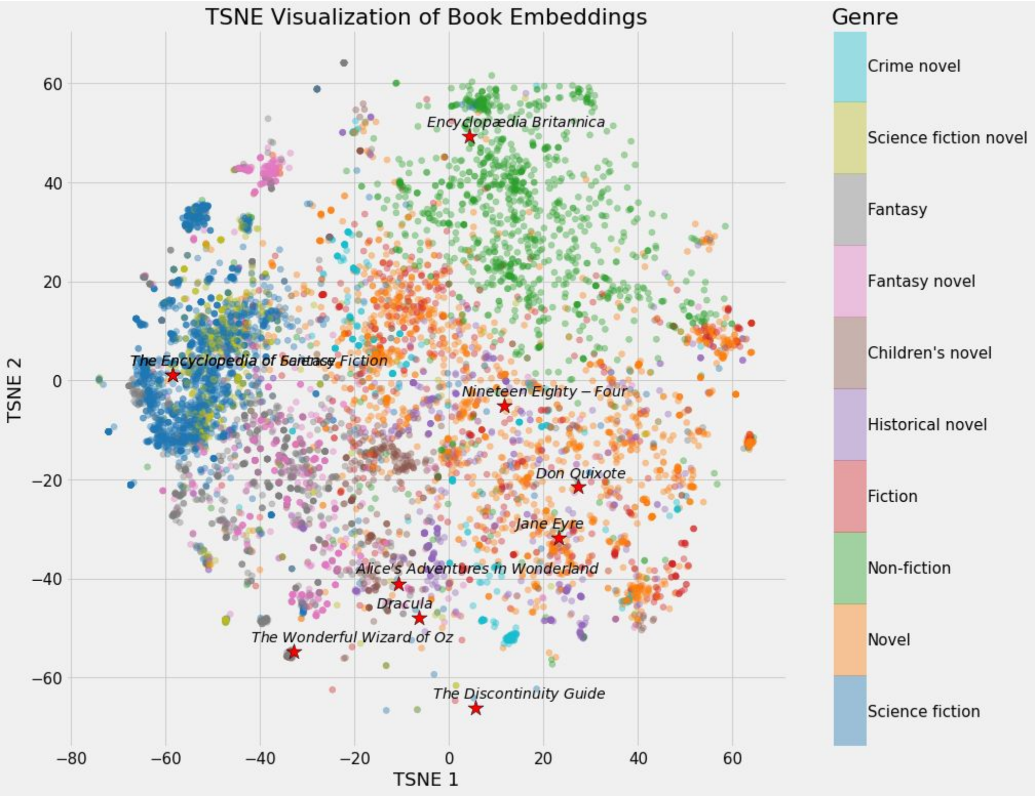
Efficient, but can distort local relationships

t-SNE
t-distributed Stochastic Neighbor Embedding

More intensive, but preserves local structure and clustering

Visualizing word embeddings

Visualizing word embeddings

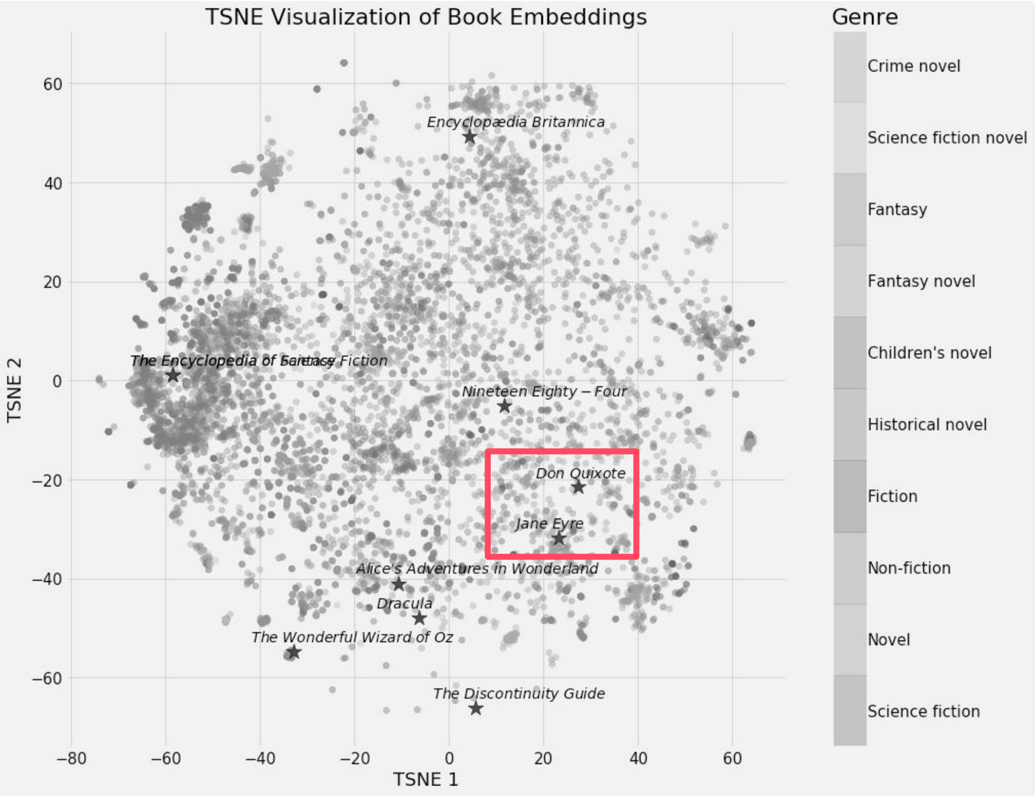


Koehrsen, 2018

DeepLearning.AI

Laurence Moroney

Visualizing word embeddings

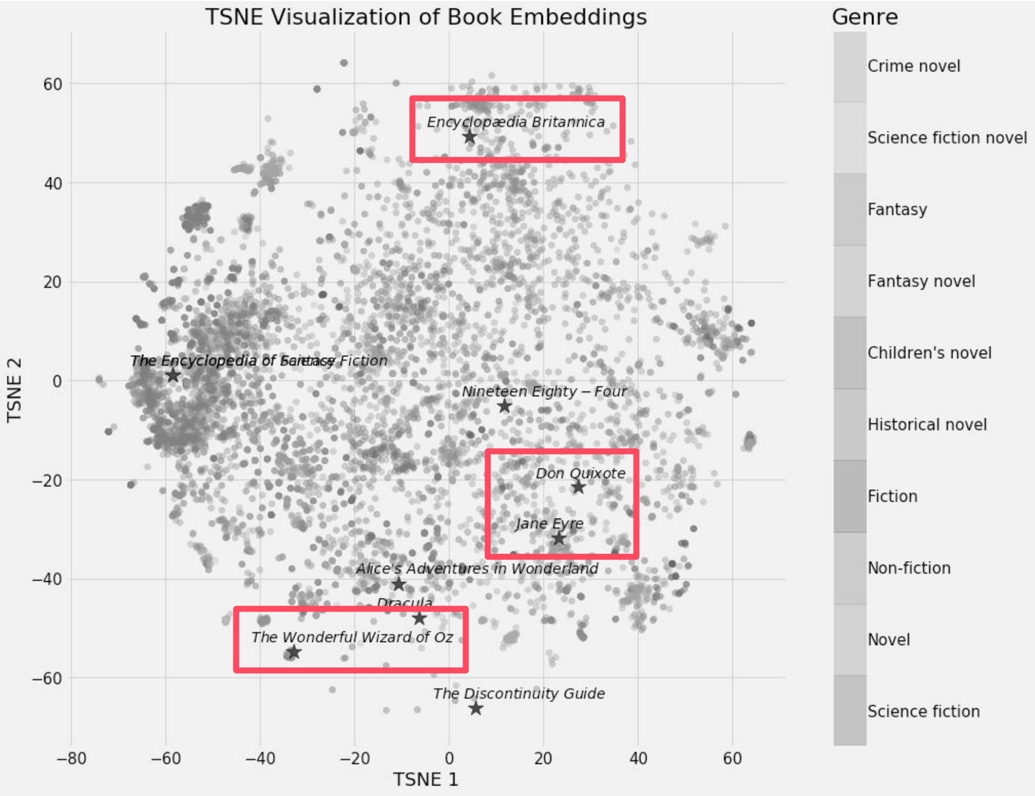


Koehrsen, 2018

DeepLearning.AI

Laurence Moroney

Visualizing word embeddings



Koehrsen, 2018

DeepLearning.AI

Laurence Moroney

Embeddings visualization in PyTorch

```
words_to_visualize = ['car', 'bike', 'plane',      # Category: Vehicles
                      'cat', 'dog', 'bird',         # Category: Pets
                      'orange', 'apple', 'grape'    # Category: Fruits
]

visualization_dict = {
   'Vehicle': ['car', 'bike', 'plane'],
   'Pet': ['cat', 'dog', 'bird'],
   'Fruit': ['orange', 'apple', 'grape']
}
```

Embeddings visualization in PyTorch

```
embedding_vectors_list = []

for word in words_to_visualize:
   embedding_vectors_list.append(glove_embeddings[word])

embedding_vectors = np.array(embedding_vectors_list)

reducer = PCA(n_components=2)

coords_2d = reducer.fit_transform(embedding_vectors)

helper_utils.plot_embeddings(coords=coords_2d,
                            labels=words_to_visualize,
                            label_dict=visualization_dict,
                            title='GloVe Pre-Trained Embeddings'
                           )
```

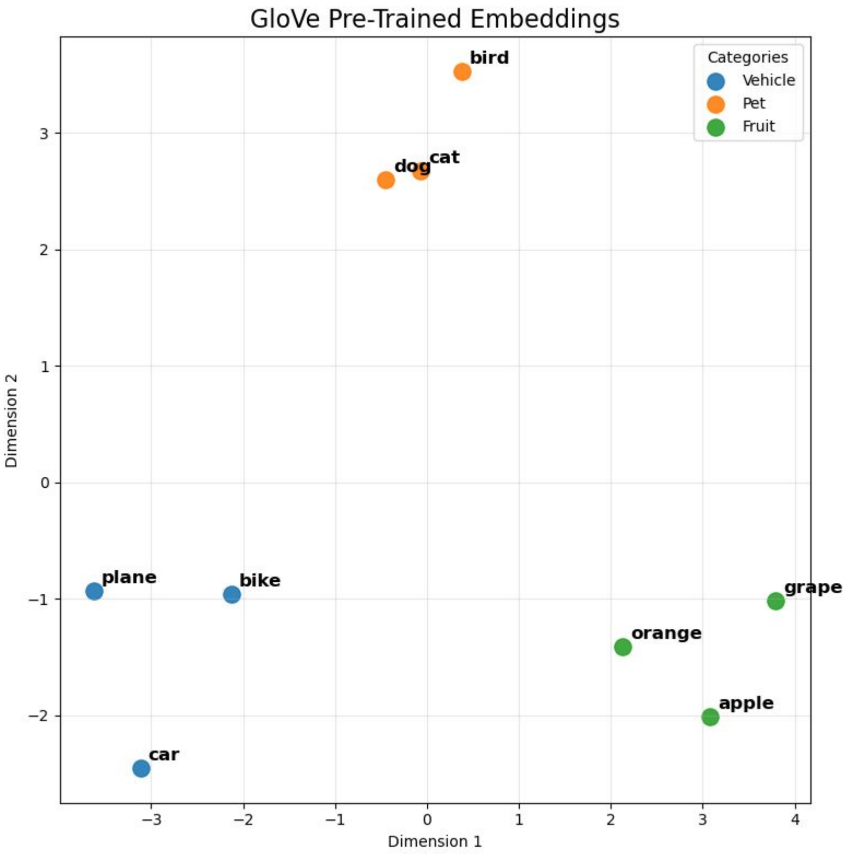
Embeddings visualization in PyTorch

```
embedding_vectors_list = []

for word in words_to_visualize:
    embedding_vectors_list.append(glove_embeddings[word])

embedding_vectors = np.array(embedding_vectors_list)

reducer = PCA(n_components=2)

coords_2d = reducer.fit_transform(embedding_vectors)

helper_utils.plot_embeddings(coords=coords_2d,
                             labels=words_to_visualize,
                             label_dict=visualization_dict,
                             title='GloVe Pre-Trained Embeddings'
                            )
```

Embeddings visualization in PyTorch

```
embedding_vectors_list = []

for word in words_to_visualize:
    embedding_vectors_list.append(glove_embeddings[word])

embedding_vectors = np.array(embedding_vectors_list)

reducer = PCA(n_components=2)

coords_2d = reducer.fit_transform(embedding_vectors)

helper_utils.plot_embeddings(coords=coords_2d,
                            labels=words_to_visualize,
                            label_dict=visualization_dict,
                            title='GloVe Pre-Trained Embeddings'
                           )
```

Embeddings visualization in PyTorch



DeepLearning.AI

Laurence Moroney

Practical considerations about embeddings

Practical considerations about embeddings

Dimensionality

Practical considerations about embeddings

Dimensionality

Initialization

Practical considerations about embeddings

Dimensionality

Initialization

Out of
vocabulary
problem

Practical considerations about embeddings

Dimensionality

Initialization

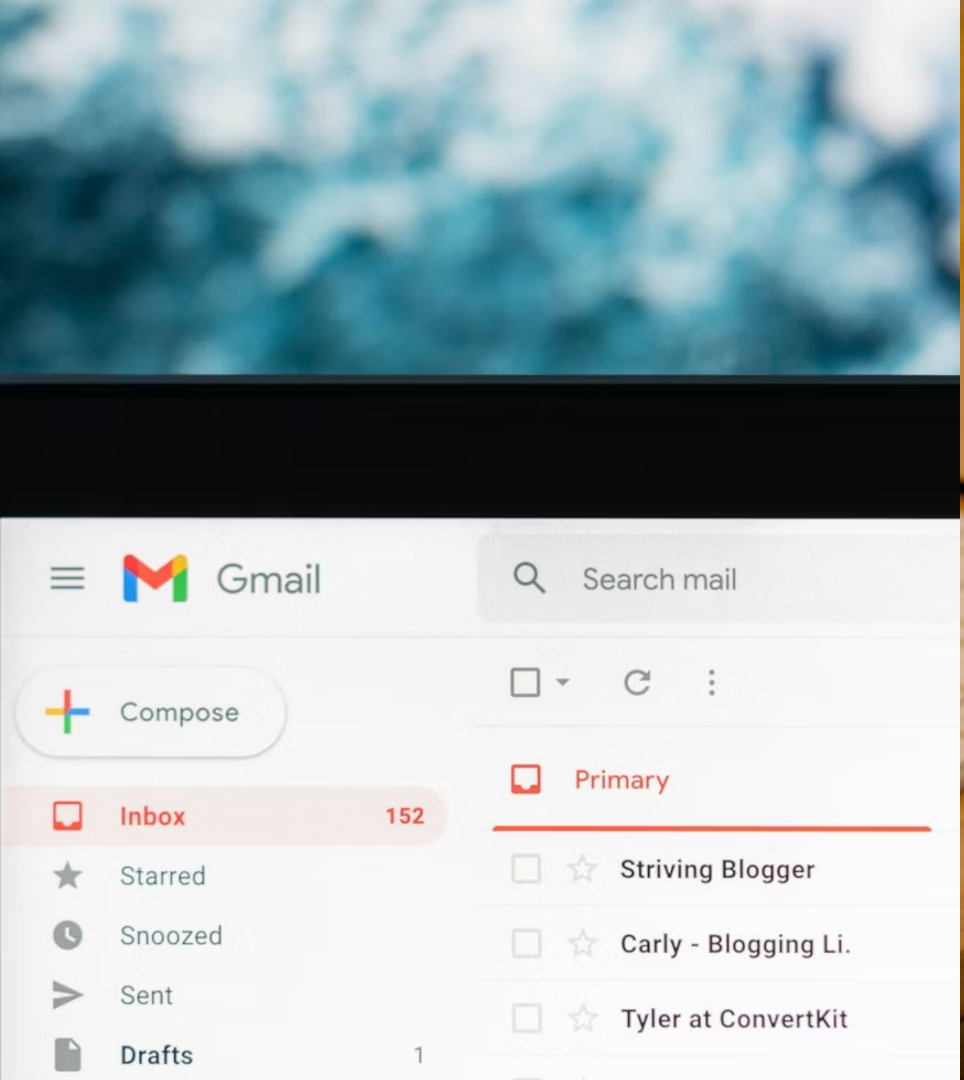
Out of vocabulary problem

Type of task

DeepLearning.AI

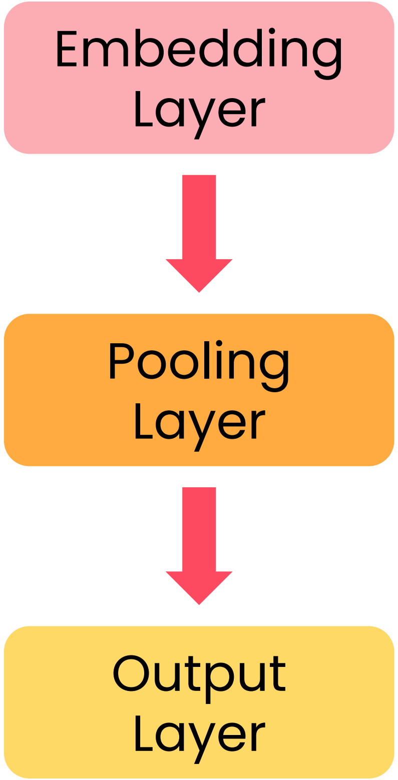
Building a Simple Text Classifier in PyTorch

Working with text using PyTorch


Gmail
Search mail
Compose
Primary
Inbox
152
Starred
Snoozed
Sent
Drafts
1
Striving Blogger
Carly - Blogging Li.
Tyler at ConvertKit


3:52
vitaleveryday
WIN

Text classification pipeline

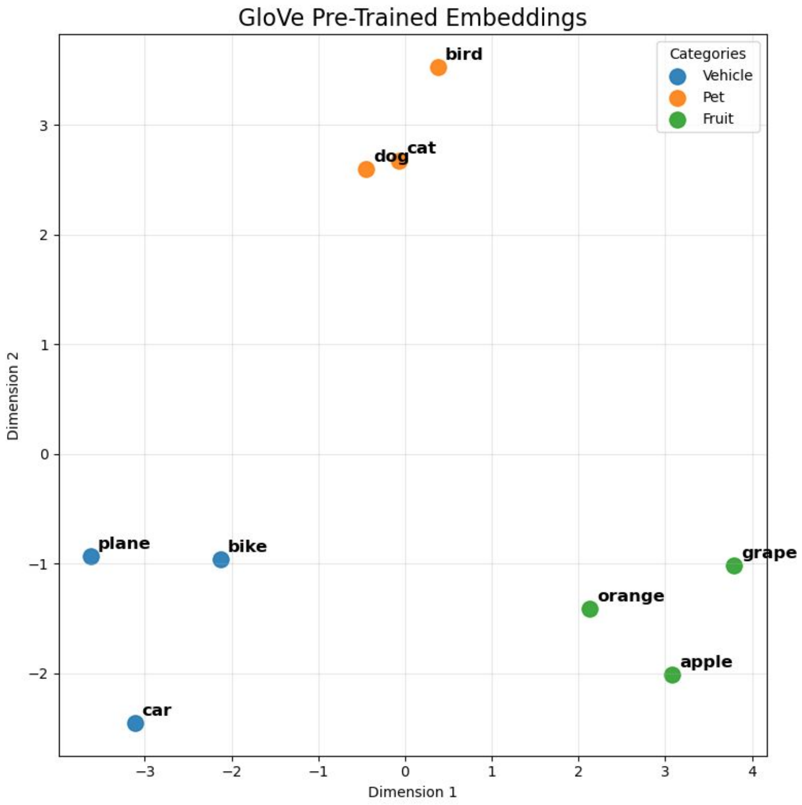


Text classification pipeline

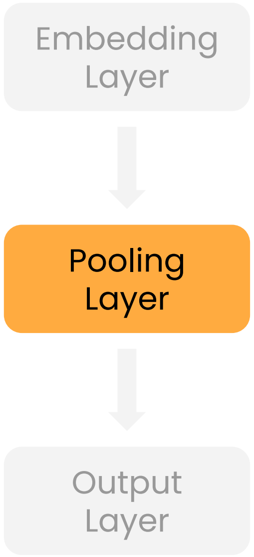
Embedding Layer

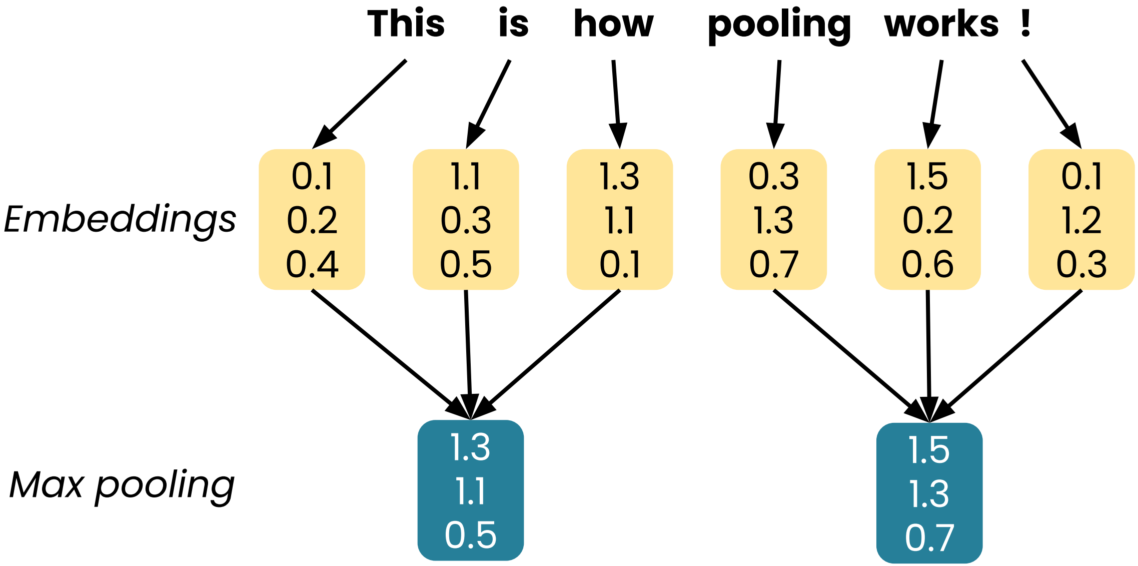
Pooling Layer

Output Layer



Text classification pipeline





Pooling techniques

Mean pooling

Takes the average across all embeddings

Pooling techniques

Mean pooling

Takes the average across all embeddings

Max pooling

Finds the dominant feature across each dimension

Pooling techniques

Mean pooling

Takes the average across all embeddings

Max pooling

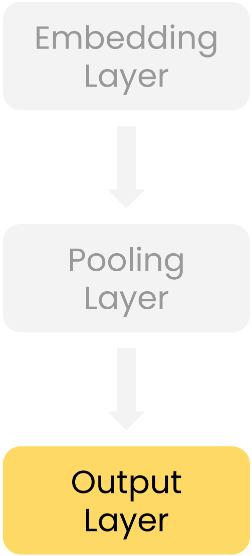
Finds the dominant feature across each dimension

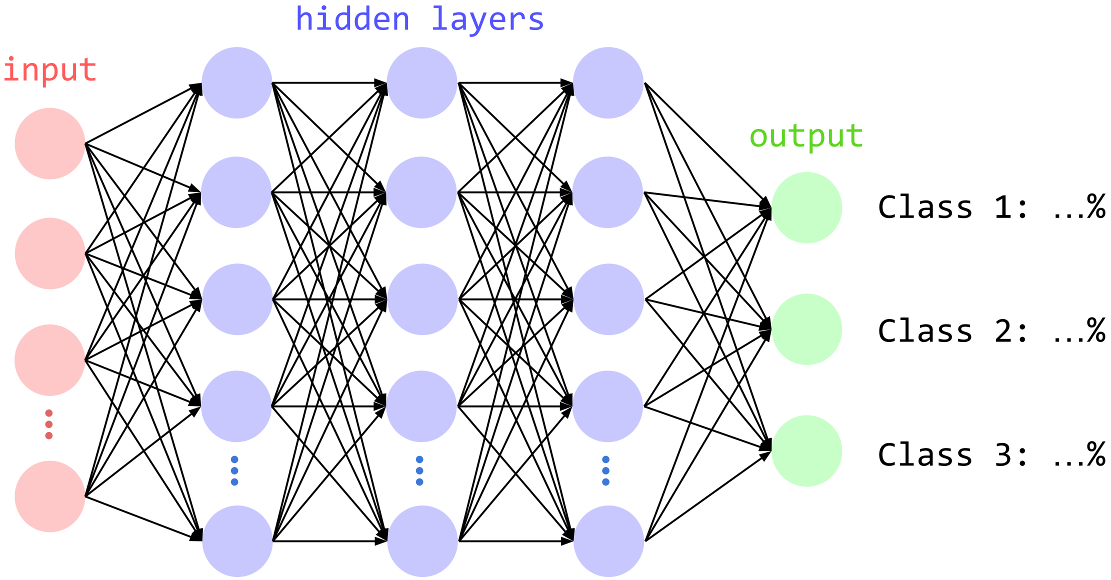
Sum pooling

Adds embeddings together

Laurence Moroney

Text classification pipeline

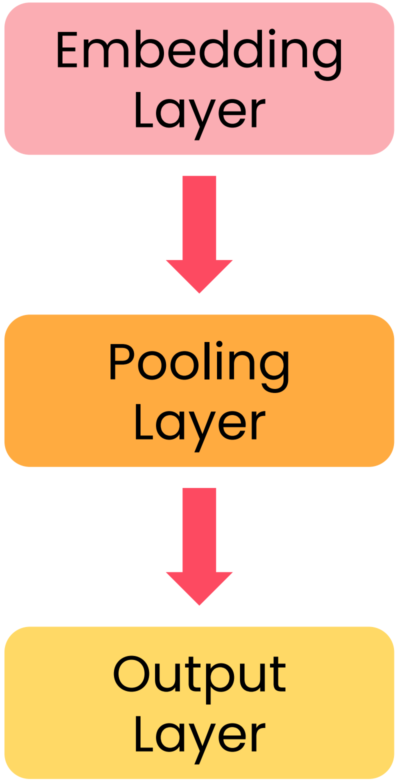




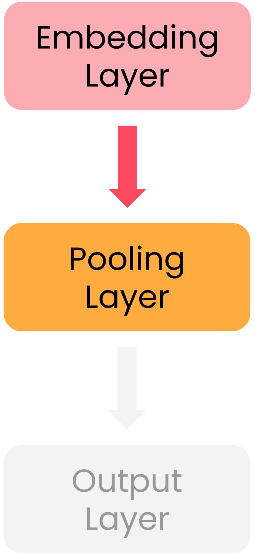
DeepLearning.AI

Laurence Moroney

Text classification pipeline



nn.EmbeddingBag



nn.EmbeddingBag

Embedding Layer

Pooling Layer

Output Layer

Why use it?

- Better performance
- Simplicity

nn.EmbeddingBag

Embedding Layer

↓

Pooling Layer

↓

Output Layer

Why use it?

- Better performance
- Simplicity

How to use it?

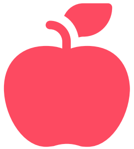
- Provide token IDs and an offset tensor
- Pools all embeddings in the chosen mode

Building a text classifier: Preprocessing

- Select and load a dataset
- Clean text
- Tokenize
- Split training vs. validation set
- Build a vocabulary
- Create DataLoader

The dataset: Food.com recipe collection

Fruit-based



“Apple a Day Milkshake”

Vegetable-based

“Zuppa Toscana”

Build a classifier based on different models

Type of embedding	EmbeddingBag	Manual pooling		
Type of pooling	Mean	Mean	Max	Sum

Building a text classifier with EmbeddingBag

```
class EmbeddingBagClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes):
        super().__init__()
        self.embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim, mode='mean')
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, text, offsets=None):
        embedded = self.embedding_bag(text, offsets)
        embedded = self.dropout(embedded)
        return self.fc(embedded)
```

Building a text classifier with EmbeddingBag

```
class EmbeddingBagClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes):
        super().__init__()
        self.embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim, mode='mean')
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, text, offsets=None):
        embedded = self.embedding_bag(text, offsets)
        embedded = self.dropout(embedded)
        return self.fc(embedded)
```

Building a text classifier with EmbeddingBag

```
class EmbeddingBagClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes):
        super().__init__()
        self.embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim, mode='mean')
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, text, offsets=None):
        embedded = self.embedding_bag(text, offsets)
        embedded = self.dropout(embedded)
        return self.fc(embedded)
```

Building a text classifier with EmbeddingBag

```
class EmbeddingBagClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes):
        super().__init__()
        self.embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim, mode='mean')
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, text, offsets=None):
        embedded = self.embedding_bag(text, offsets)
        embedded = self.dropout(embedded)
        return self.fc(embedded)
```

Building a text classifier with EmbeddingBag

```
class EmbeddingBagClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes):
        super().__init__()
        self.embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim, mode='mean')
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, text, offsets=None):
        embedded = self.embedding_bag(text, offsets)
        embedded = self.dropout(embedded)
        return self.fc(embedded)
```

Building a text classifier with manual pooling

```
class ManualPoolingClassifier(nn.Module):

   def __init__(self, vocab_size, embedding_dim, num_classes, pooling='mean'):
       super().__init__()
       self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
       self.pooling = pooling
       self.fc = nn.Linear(embedding_dim, num_classes)
       self.dropout = nn.Dropout(0.5)
```

Building a text classifier with manual pooling

```
class ManualPoolingClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes, pooling='mean'):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.pooling = pooling
        self.fc = nn.Linear(embedding_dim, num_classes)
        self.dropout = nn.Dropout(0.5)
```

Building a text classifier with manual pooling

```
class ManualPoolingClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_dim, num_classes, pooling='mean'):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.pooling = pooling
        self.fc = nn.Linear(embedding_dim, num_classes)
        self.dropout = nn.Dropout(0.5)
```

Building a text classifier with manual pooling

```
class ManualPoolingClassifier(nn.Module):

   def __init__(self, vocab_size, embedding_dim, num_classes, pooling='mean'):
       super().__init__()
       self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
       self.pooling = pooling
       self.fc = nn.Linear(embedding_dim, num_classes)
       self.dropout = nn.Dropout(0.5)
```

Building a text classifier with manual pooling

```
def forward(self, text):
    embedded = self.embedding(text)
    mask = (text != 0).float().unsqueeze(-1)
    embedded = embedded * mask

    if self.pooling == 'mean':
        pooled = embedded.sum(dim=1) / mask.sum(dim=1).clamp(min=1)
    elif self.pooling == 'max':
        embedded[mask.squeeze(-1) == 0] = float('-inf')
        pooled, _ = embedded.max(dim=1)
    elif self.pooling == 'sum':
        pooled = embedded.sum(dim=1)

    pooled = self.dropout(pooled)
    return self.fc(pooled)
```


Laurence Moroney

Building a text classifier with manual pooling

```
def forward(self, text):
    embedded = self.embedding(text)
    mask = (text != 0).float().unsqueeze(-1)
    embedded = embedded * mask

    if self.pooling == 'mean':
        pooled = embedded.sum(dim=1) / mask.sum(dim=1).clamp(min=1)
    elif self.pooling == 'max':
        embedded[mask.squeeze(-1) == 0] = float('-inf')
        pooled, _ = embedded.max(dim=1)
    elif self.pooling == 'sum':
        pooled = embedded.sum(dim=1)

    pooled = self.dropout(pooled)
    return self.fc(pooled)
```

Building a text classifier with manual pooling

```
def forward(self, text):
    embedded = self.embedding(text)
    mask = (text != 0).float().unsqueeze(-1)
    embedded = embedded * mask

    if self.pooling == 'mean':
        pooled = embedded.sum(dim=1) / mask.sum(dim=1).clamp(min=1)
    elif self.pooling == 'max':
        embedded[mask.squeeze(-1) == 0] = float('-inf')
        pooled, _ = embedded.max(dim=1)
    elif self.pooling == 'sum':
        pooled = embedded.sum(dim=1)

    pooled = self.dropout(pooled)
    return self.fc(pooled)
```

Building a text classifier with manual pooling

```
def forward(self, text):
    embedded = self.embedding(text)
    mask = (text != 0).float().unsqueeze(-1)
    embedded = embedded * mask

    if self.pooling == 'mean':
        pooled = embedded.sum(dim=1) / mask.sum(dim=1).clamp(min=1)
    elif self.pooling == 'max':
        embedded[mask.squeeze(-1) == 0] = float('-inf')
        pooled, _ = embedded.max(dim=1)
    elif self.pooling == 'sum':
        pooled = embedded.sum(dim=1)

    pooled = self.dropout(pooled)
    return self.fc(pooled)
```

Building a text classifier with manual pooling

```
def forward(self, text):
    embedded = self.embedding(text)
    mask = (text != 0).float().unsqueeze(-1)
    embedded = embedded * mask

    if self.pooling == 'mean':
        pooled = embedded.sum(dim=1) / mask.sum(dim=1).clamp(min=1)
    elif self.pooling == 'max':
        embedded[mask.squeeze(-1) == 0] = float('-inf')
        pooled, _ = embedded.max(dim=1)
    elif self.pooling == 'sum':
        pooled = embedded.sum(dim=1)

    pooled = self.dropout(pooled)
    return self.fc(pooled)
```

Building a text classifier with manual pooling

```
def forward(self, text):
    embedded = self.embedding(text)
    mask = (text != 0).float().unsqueeze(-1)
    embedded = embedded * mask

    if self.pooling == 'mean':
        pooled = embedded.sum(dim=1) / mask.sum(dim=1).clamp(min=1)
    elif self.pooling == 'max':
        embedded[mask.squeeze(-1) == 0] = float('-inf')
        pooled, _ = embedded.max(dim=1)
    elif self.pooling == 'sum':
        pooled = embedded.sum(dim=1)

    pooled = self.dropout(pooled)
    return self.fc(pooled)
```

Building a text classifier with manual pooling

```
vocab_size = len(vocab)
embedding_dim = 64
num_classes = 2

model_embag = EmbeddingBagClassifier(vocab_size, embedding_dim, num_classes)
model_manual_mean = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='mean')
model_manual_max = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='max')
model_manual_sum = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='sum')
```

Building a text classifier with manual pooling

```
vocab_size = len(vocab)
embedding_dim = 64
num_classes = 2

model_embag = EmbeddingBagClassifier(vocab_size, embedding_dim, num_classes)
model_manual_mean = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='mean')
model_manual_max = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='max')
model_manual_sum = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='sum')
```

Building a text classifier with manual pooling

```
vocab_size = len(vocab)
embedding_dim = 64
num_classes = 2

model_embag = EmbeddingBagClassifier(vocab_size, embedding_dim, num_classes)
model_manual_mean = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='mean')
model_manual_max = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='max')
model_manual_sum = ManualPoolingClassifier(vocab_size, embedding_dim,
num_classes, pooling='sum')
```

DeepLearning.AI

Fine Tuning Pretrained Text Classification Models

Working with text using PyTorch

Why use pretrained models?

Preloaded
with semantic
knowledge

Why use pretrained models?

Preloaded with semantic knowledge

Transfer learning benefits

Why use pretrained models?

Preloaded with semantic knowledge

Transfer learning benefits

Efficiency

Why use pretrained models?

Preloaded with semantic knowledge

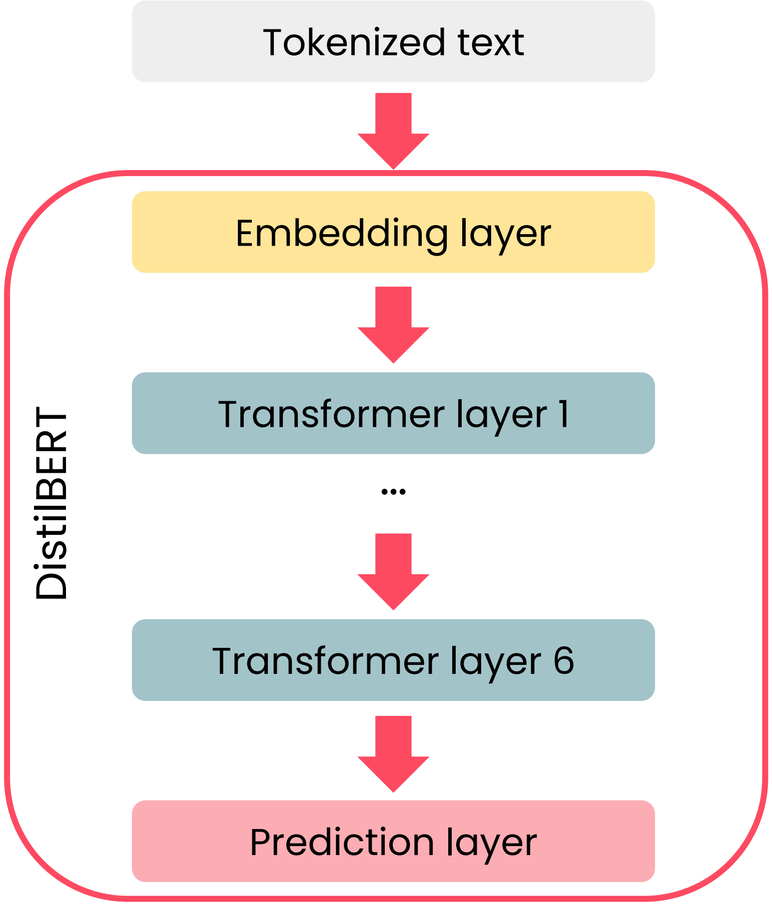
Transfer learning benefits

Efficiency

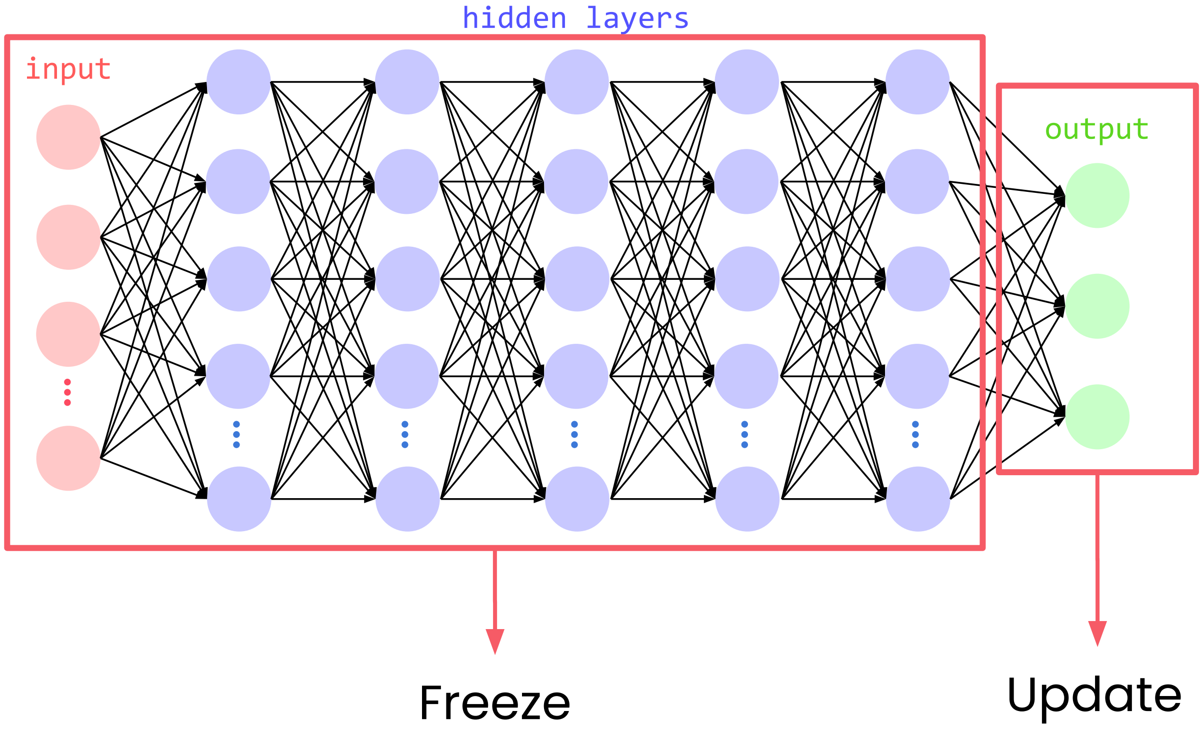
Better generalization

DistilBERT

- Transformer model
- Use cases:
  - Text classification
  - Named entity recognition
  - Question answering, and more



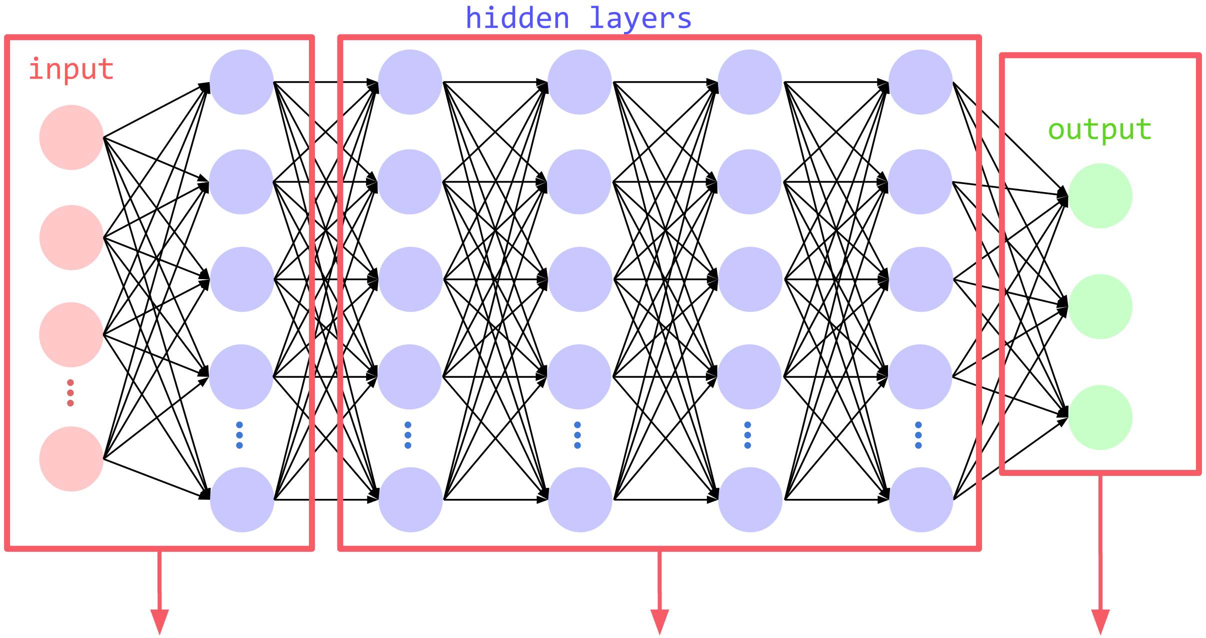
Transfer learning



DeepLearning.AI

Laurence Moroney

Fine tuning



Freeze

Unfreeze and train further

Update and retrain

Considerations when fine-tuning

Advantages

- Adapt to specific vocab
- Optimize for specific goal
- Higher performance

Considerations when fine-tuning

Advantages

- Adapt to specific vocab
- Optimize for specific goal
- Higher performance

Risks

- Catastrophic forgetting
- Overfitting
- Training instability

Loading DISTILBert

```
model_name="distilbert-base-uncased"
model_path="./distilbert-local-base"

# Ensure the model is downloaded
helper_utils.download_bert(model_name, model_path)

bert_model, bert_tokenizer = helper_utils.load_bert(model_path, num_classes=2)
```

Loading DISTILBert

```
model_name="distilbert-base-uncased"
model_path="./distilbert-local-base"

# Ensure the model is downloaded
helper_utils.download_bert(model_name, model_path)

bert_model, bert_tokenizer = helper_utils.load_bert(model_path, num_classes=2)
```

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
   param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
   layer_to_unfreeze = transformer_layers[-(i+1)]

   for param in layer_to_unfreeze.parameters():
       param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
   param.requires_grad = True

for param in bert_model.classifier.parameters():
   param.requires_grad = True
```

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
   param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
   layer_to_unfreeze = transformer_layers[-(i+1)]

   for param in layer_to_unfreeze.parameters():
       param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
   param.requires_grad = True

for param in bert_model.classifier.parameters():
   param.requires_grad = True
```

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
   param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
   layer_to_unfreeze = transformer_layers[-(i+1)]

   for param in layer_to_unfreeze.parameters():
       param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
   param.requires_grad = True

for param in bert_model.classifier.parameters():
   param.requires_grad = True
```

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
    param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
    layer_to_unfreeze = transformer_layers[-(i+1)]

    for param in layer_to_unfreeze.parameters():
        param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
    param.requires_grad = True

for param in bert_model.classifier.parameters():
    param.requires_grad = True
```


Laurence Moroney

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
   param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
   layer_to_unfreeze = transformer_layers[-(i+1)]

   for param in layer_to_unfreeze.parameters():
       param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
   param.requires_grad = True

for param in bert_model.classifier.parameters():
   param.requires_grad = True
```


Laurence Moroney

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
    param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
    layer_to_unfreeze = transformer_layers[-(i+1)]

    for param in layer_to_unfreeze.parameters():
        param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
    param.requires_grad = True

for param in bert_model.classifier.parameters():
    param.requires_grad = True
```

Partial fine-tuning with DistilBERT

```
for param in bert_model.parameters():
   param.requires_grad = False

layers_to_train = 2
transformer_layers = bert_model.distilbert.transformer.layer
for i in range(layers_to_train):
   layer_to_unfreeze = transformer_layers[-(i+1)]

   for param in layer_to_unfreeze.parameters():
       param.requires_grad = True

for param in bert_model.pre_classifier.parameters():
   param.requires_grad = True

for param in bert_model.classifier.parameters():
   param.requires_grad = True
```

Every architecture is different!

What you learned in this module

Unique challenges with text data

What you learned in this module

- Unique challenges with text data
- Tokenization and tensorization

What you learned in this module

- Unique challenges with text data
- Tokenization and tensorization
- Embeddings: turning words into vectors

What you learned in this module

- Unique challenges with text data
- Tokenization and tensorization
- Embeddings: turning words into vectors
- Building a text classification pipeline

What you learned in this module

- Unique challenges with text data
- Tokenization and tensorization
- Embeddings: turning words into vectors
- Building a text classification pipeline
- Using pretrained models and fine-tuning