JUNE 18-22, 2023
CVPR
VANCOUVER, CANADA

Neural Search in Action

Representing, transiting & searching **multimodal data**

Han Xiao, Founder of Jina AI
@hxiao @JinaAI_

Berlin · Beijing · Shenzhen

About me & Jina AI

Han Xiao, Founder & CEO of Jina AI. Based in Berlin, Germany.

- ML PhD in 2014 TU Munich; Zalando Research; Tencent AI Lab; Creator of Fashion-MNIST.

Jina AI

- Founded in 2020, focus on multimodal AI search & create
- Opensource contributor: Jina, **DocArray (Linux Foundation)**, CLIP-as-service, ...
- 60 people, HQ in Berlin. Offices in Beijing, Shenzhen.

Jina AI Tech Spectrum

Prompt tuning

the process of crafting and refining the input prompts in order to guide its output towards specific, desired responses.

the deployment of fine-tuned models in a production environment, usually requiring substantial resources such as GPU hosting. MLOps, emphasizing the serving of mid-size to large models in a scalable, efficient, and reliable manner.

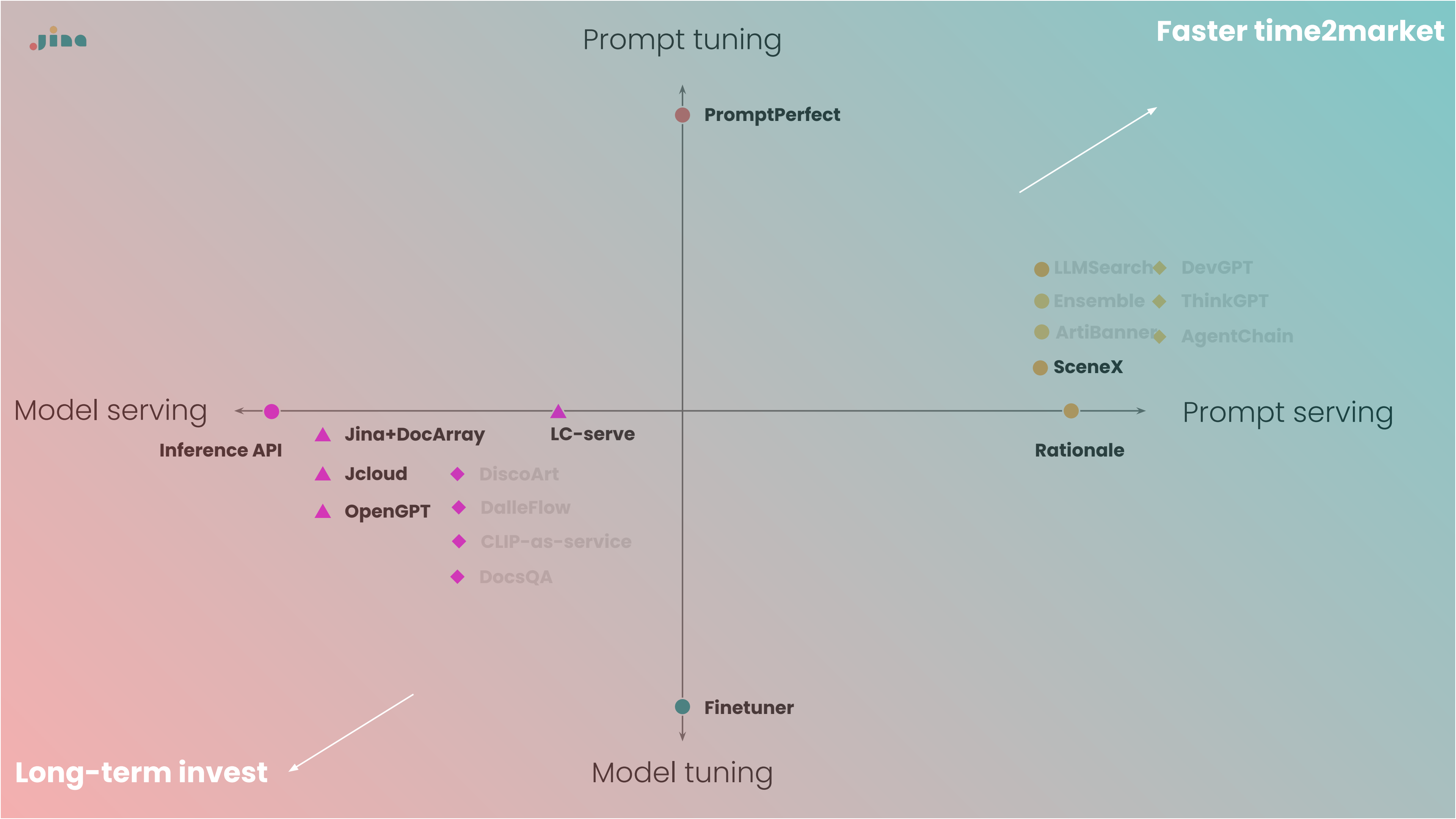
Model serving

Prompt serving

wrapping and serving prompts through an API, without hosting heavy models. The API calls a public large language model service and handles the orchestration of inputs and outputs in a chain of operations.

Also known as fine-tuning, involves adjusting the parameters of a pre-trained model on a new, often task-specific dataset to improve its performance and adapt it to a specific application.

Model tuning

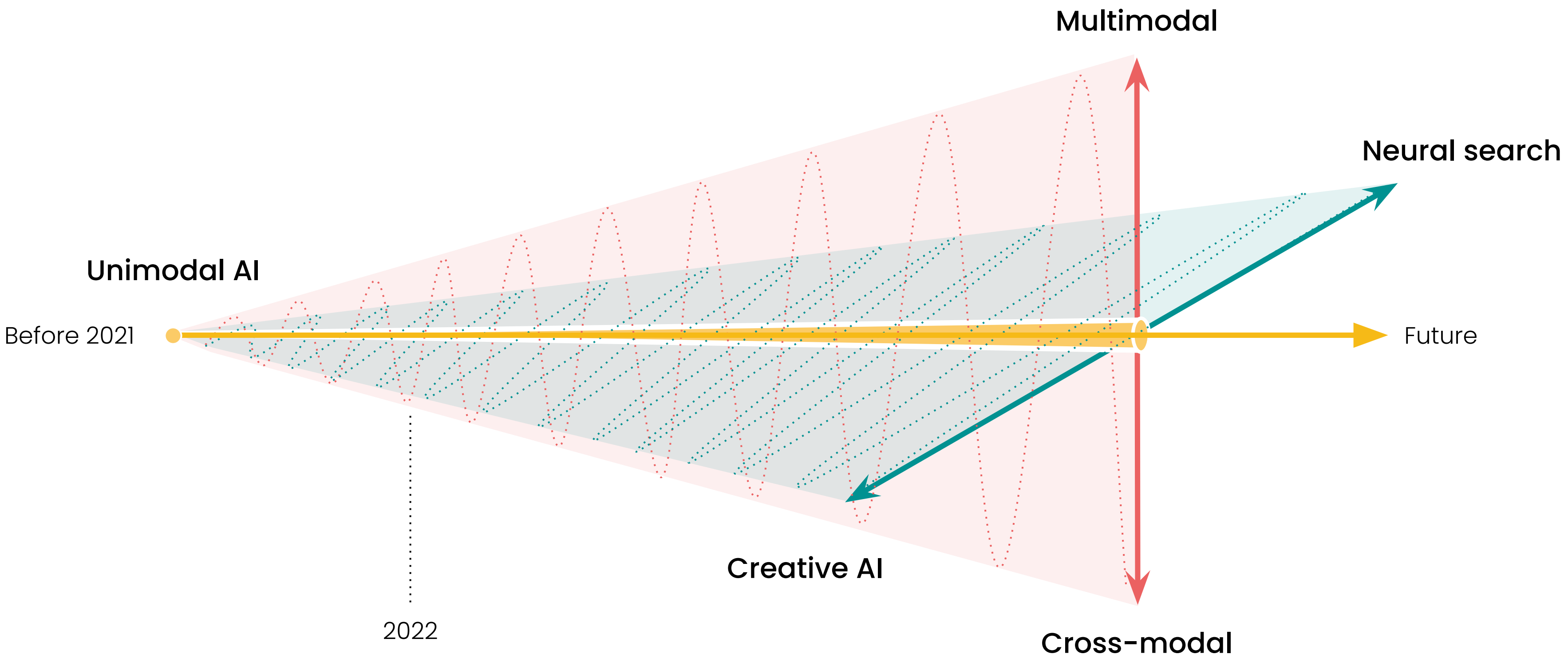

Faster time2market
Prompt tuning
PromptPerfect
LLMSearch
DevGPT
Ensemble
ThinkGPT
ArtiBanner
AgentChain
SceneX
Model serving
Inference API
Jina+DocArray
LC-serve
Jcloud
OpenGPT
DiscoArt
DalleFlow
CLIP-as-service
DocsQA
Prompt serving
Rationale
Finetuner
Model tuning
Long-term invest

Agenda

- **Preliminary: multimodal AI**
- Opensource package: DocArray
    - Motivation
    - Representing data
    - Transiting data
    - Storing data
    - Retrieving data
- Multimodal at scale in production

This tutorial may require technical knowledge. Familiarity with Python 3.7+ concepts like data classes could be helpful.

Preliminary:
from unimodal to multimodal


Multimodal
Neural search
Unimodal AI
Before 2021
Future
Creative AI
2022
Cross-modal

From unimodal to multimodal

"**modality**" roughly means "**data type**".

- Unimodal AI refers to applying AI to one specific type of data.
- Most early machine learning works fall into this category.
- Even today, when you open any machine learning literature, unimodal AI is still the majority of the content.

Unimodal – NLP

LDA was the 2010's transformer

"Arts"	"Budgets"	"Children"	"Education"
NEW	MILLION	CHILDREN	SCHOOL
FILM	TAX	WOMEN	STUDENTS
SHOW	PROGRAM	PEOPLE	SCHOOLS
MUSIC	BUDGET	CHILD	EDUCATION
MOVIE	BILLION	YEARS	TEACHERS
PLAY	FEDERAL	FAMILIES	HIGH
MUSICAL	YEAR	WORK	PUBLIC
BEST	SPENDING	PARENTS	TEACHER
ACTOR	NEW	SAYS	BENNETT
FIRST	STATE	FAMILY	MANIGAT
YORK	PLAN	WELFARE	NAMPHY
OPERA	MONEY	MEN	STATE
THEATER	PROGRAMS	PERCENT	PRESIDENT
ACTRESS	GOVERNMENT	CARE	ELEMENTARY
LOVE	CONGRESS	LIFE	HAITI

The William Randolph Hearst Foundation will give $1.25 million to Lincoln Center, Metropolitan Opera Co., New York Philharmonic and Juilliard School. "Our board felt that we had a real opportunity to make a mark on the future of the performing arts with these grants an act every bit as important as our traditional areas of support in health, medical research, education and the social services," Hearst Foundation President Randolph A. Hearst said Monday in announcing the grants. Lincoln Center's share will be $200,000 for its new building, which will house young artists and provide new public facilities. The Metropolitan Opera Co. and New York Philharmonic will receive $400,000 each. The Juilliard School, where music and the performing arts are taught, will get $250,000. The Hearst Foundation, a leading supporter of the Lincoln Center Consolidated Corporate Fund, will make its usual annual $100,000 donation, too.

JMLR: Workshop and Conference Proceedings 13: 63-78
2nd Asian Conference on Machine Learning (ACML2010), Tokyo, Japan, Nov. 8–10, 2010.

Efficient Collapsed Gibbs Sampling For Latent Dirichlet Allocation

Han Xiao XIAOH@IN.TUM.DE
Thomas Stibor STIBOR@IN.TUM.DE
Department of Informatics
Technical University of Munich, GERMANY

Editor: Masashi Sugiyama and Qiang Yang

Abstract

Collapsed Gibbs sampling is a frequently applied method to approximate intractable integrals in probabilistic generative models such as latent Dirichlet allocation. This sampling method has however the crucial drawback of high computational complexity, which makes it limited applicable on large data sets. We propose a novel *dynamic sampling* strategy to significantly improve the efficiency of collapsed Gibbs sampling. The strategy is explored in terms of efficiency, convergence and perplexity. Besides, we present a straight-forward parallelization to further improve the efficiency. Finally, we underpin our proposed improvements with a comparative study on different scale data sets.

Keywords: Gibbs sampling, Optimization, Latent Dirichlet Allocation

1. Introduction

Latent Dirichlet allocation (LDA) is a generative probabilistic model that was first proposed by Blei et al. (2003) to discover topics in text documents. LDA is based on the

Unimodal tasks in NLP

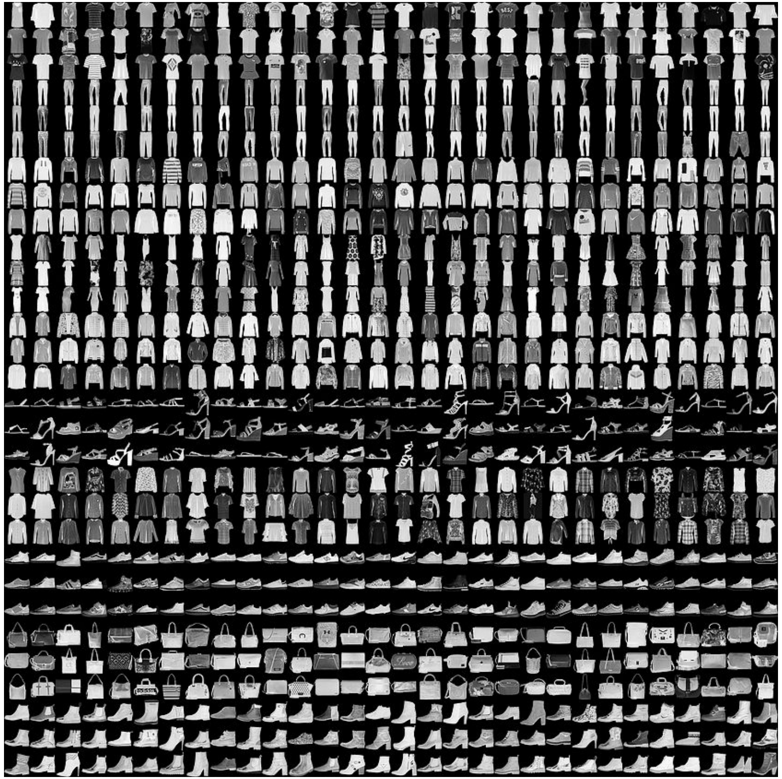
Adhoc methods for NLP problems

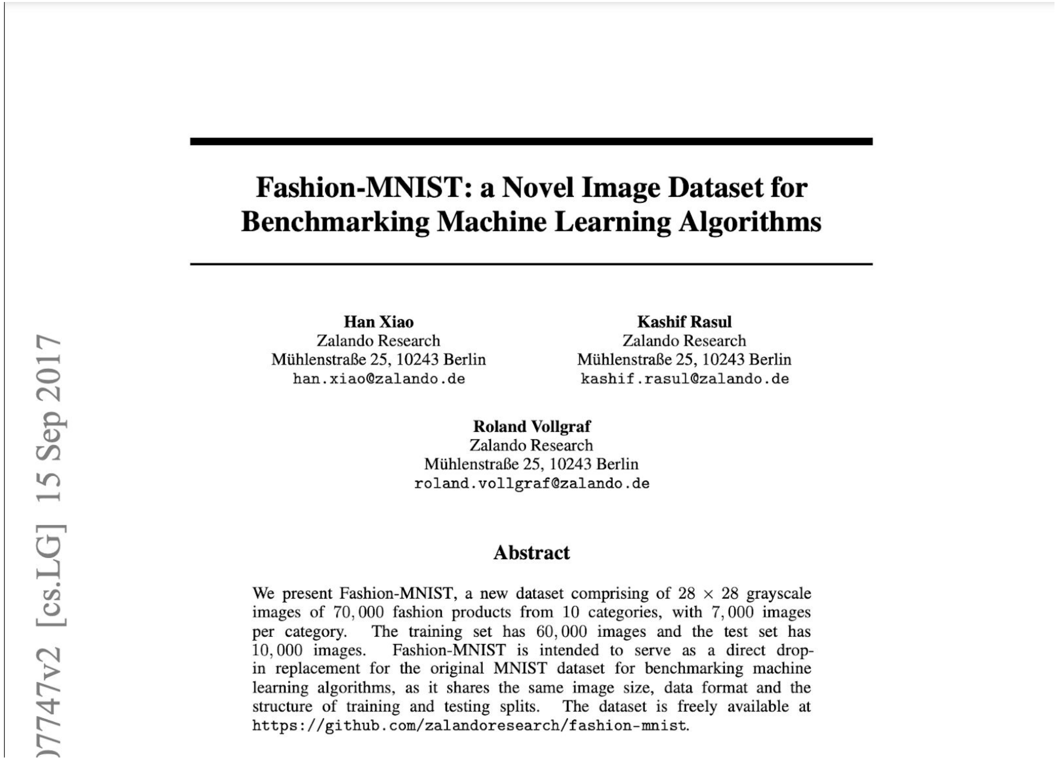
Sentiment analysis	Text classification	Topic modeling	Text summarization	Natural language generation
Named entity recognition	Word sense disambiguation	Parts-of-speech tagging	Grammatical parsing	Machine translation
Question answering	Spam filtering	Language modeling	Dialog systems	Information extraction
Semantic role labeling	Part-of-speech induction	Co-reference resolution	Pronoun resolution	Sentence segmentation

Textual Modality

Unimodal - CV

Fashion-MNIST, 2017





)7747v2 [cs.LG] 15 Sep 2017

Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms

Han Xiao
Zalando Research
Mühlenstraße 25, 10243 Berlin
han.xiao@zalando.de

Kashif Rasul
Zalando Research
Mühlenstraße 25, 10243 Berlin
kashif.rasul@zalando.de

Roland Vollgraf
Zalando Research
Mühlenstraße 25, 10243 Berlin
roland.vollgraf@zalando.de

Abstract

We present Fashion-MNIST, a new dataset comprising of 28×28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category. The training set has 60,000 images and the test set has 10,000 images. Fashion-MNIST is intended to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms, as it shares the same image size, data format and the structure of training and testing splits. The dataset is freely available at https://github.com/zalandoresearch/fashion-mnist.

Unimodal tasks in CV

Object classification and detection	Image segmentation	Object tracking	Action recognition	Scene understanding
3D reconstruction	Pose estimation	Depth estimation	Stereo vision	Texture recognition and classification
Material recognition	Object recognition in video	Facial recognition and identification	Human activity recognition	Image super-resolution
Neural style transfer	Image inpainting	Video frame interpolation	Multiple object tracking in 3D	SLAM

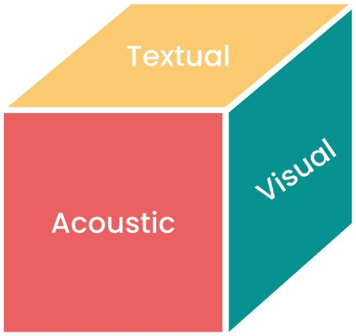
Visual Modality

Unimodal tasks in speech & audio

Automatic Speech Recognition	Text-to-Speech	Speaker Recognition	Speaker Diarization	Speech Enhancement
Music Recommendation	Music Genre Recognition	Music Artist Recognition	Music Structure Segmentation	Music Tempo Estimation
Audio Source Separation	Sound Event Detection	Sound Event Classification	Sound Event Localization	Audio Scene Recognition
Audio Captioning	Emotion Recognition	Speech Translation	Voice Activity Detection	Silence Detection

Acoustic Modality

Unimodal know-how are hardly transferable



- Tasks are specific to just one modality (e.g. textual, visual, acoustic, etc).
- Knowledge is learned from and applied to only one modality (i.e. a visual algorithm can only learn from and be applied to images).

Sentiment analysis	Text classification	Topic modeling	Text summarization	Natural language generation
Named entity recognition	Word sense disambiguation	Parts-of-speech tagging	Grammatical parsing	Machine translation
Question answering	Spam filtering	Language modeling	Dialog systems	Information extraction
Semantic role labeling	Part-of-speech induction	Co-reference resolution	Pronoun resolution	Sentence segmentation

Textual Modality

Object classification and detection	Image segmentation	Object tracking	Action recognition	Scene understanding
3D reconstruction	Pose estimation	Depth estimation	Stereo vision	Texture recognition and classification
Material recognition	Object recognition in video	Facial recognition and identification	Human activity recognition	Image super-resolution
Neural style transfer	Image inpainting	Video frame interpolation	Multiple object tracking in 3D	SLAM

Visual Modality

Automatic Speech Recognition	Text-to-Speech	Speaker Recognition	Speaker Diarization	Speech Enhancement
Music Recommendation	Music Genre Recognition	Music Artist Recognition	Music Structure Segmentation	Music Tempo Estimation
Audio Source Separation	Sound Event Detection	Sound Event Classification	Sound Event Localization	Audio Scene Recognition
Audio Captioning	Emotion Recognition	Speech Translation	Voice Activity Detection	Silence Detection

Acoustic Modality

A detour: cross-modal model

NIPS 2010, Cross-LDA

Toward Artificial Synesthesia: Linking Images and Sounds via Words

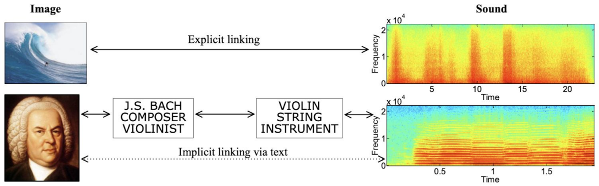
Han Xiao, Thomas Stibor
Department of Informatics
Technical University of Munich
Garching, D-85748
{xiaoh,stibor}@in.tum.de

Abstract

We tackle a new challenge of modeling a perceptual experience in which a stimulus in one modality gives rise to an experience in a different sensory modality, termed synesthesia. To meet the challenge, we propose a probabilistic framework based on graphical models that enables to link visual modalities and auditory modalities via natural language text. An online prototype system is developed for allowing human judgement to evaluate the model's performance. Experimental results indicate usefulness and applicability of the framework.

1 Introduction

A picture of a golden beach might stimulate human's hearing, probably, by imagining the sound of waves crashing against the shore. On the other hand, the sound of a baaing sheep might illustrate a green hillside in front of your eyes. In neurology, this kind of experience is termed *synesthesia*. That is, a perceptual experience in which a stimulus in one modality gives rise to an experience in a different sensory modality. Without a doubt, the creative process of humans (e.g. painting and composing) is to a large extent attributed to their synesthesia experiences. While cross-sensory links such as sound and vision are quite common to humans, machines do not possess the same



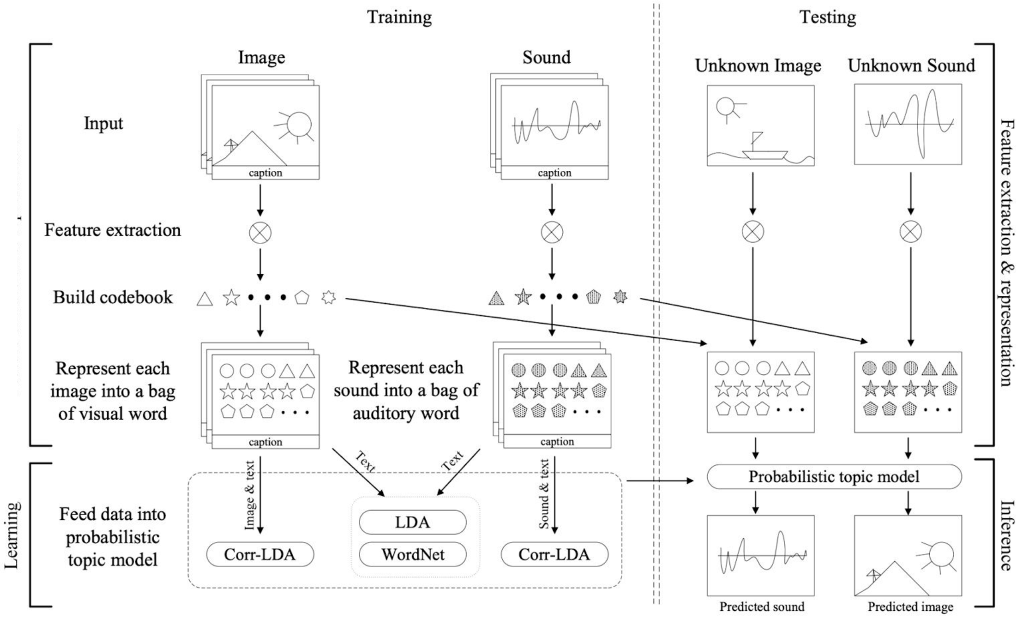
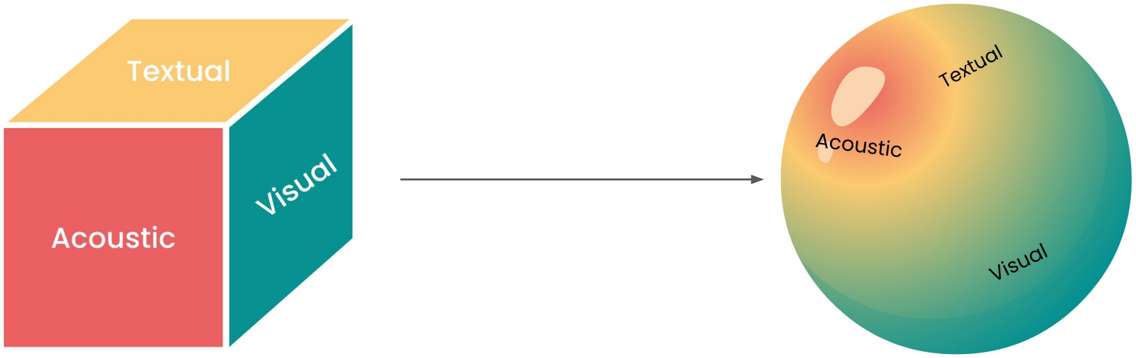


Figure 2: Probabilistic framework for performing the image composition and sound illustration task. The framework is an extension based on the work flow proposed in [8]. Images and sounds are represented in bags-of-words, so that the difference between the two modalities can be omitted. Once we have the algorithm for inferring sounds from an image, we can apply it to infer images from a sound by mirroring the algorithm.

Erase the boundary between modalities



- Tasks are shared and transferred between multiple modalities (so one algorithm can work with images and text and audio).
- Knowledge is learned from and applied to multiple modalities (so an algorithm can learn from textual data and apply that to visual data).

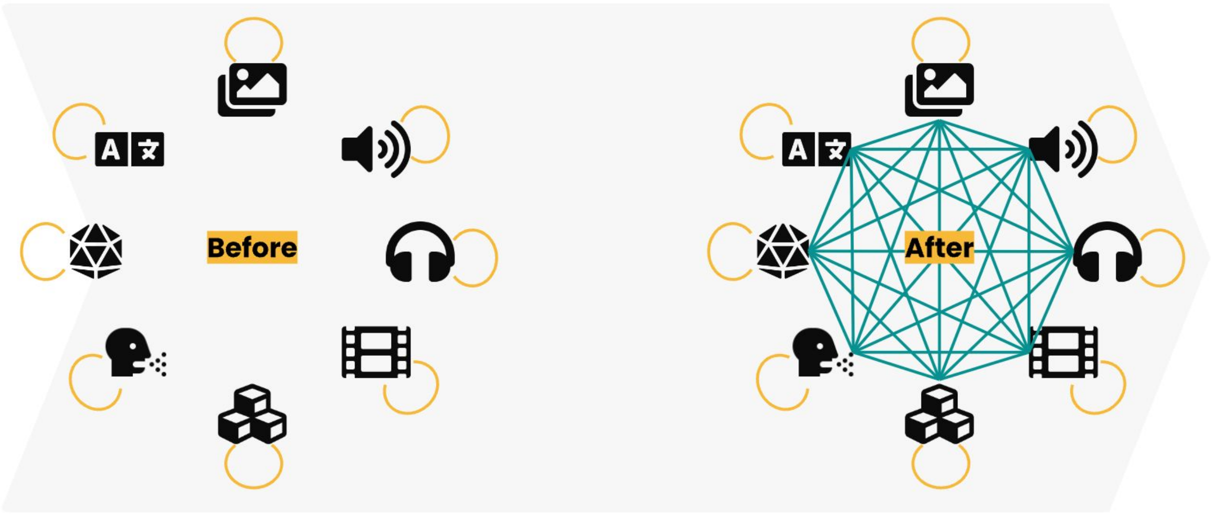
Paradigm shift from unimodal to multimodal

The rise of multimodal AI can be attributed to advances in two machine learning techniques: **Representation learning** and **transfer learning**.

- Representation learning lets models create common representations for all modalities.
- Transfer learning lets models first learn fundamental knowledge, and then fine-tune on specific domains.

CLIP, DALLE, BLIP, Bark, GPT4

We will see more and more AI applications move beyond one data modality and leverage relationships between different modalities



The paradigm shift from single-modal AI to multimodal AI

“An artificial intelligence system trained on words and sentences alone will never approximate human understanding.”

Y. Lecun in 2022 in AI And The Limits Of Language

Multimodal AI is the future, but the ML ecosystem is not yet suited for it.

Agenda

- Preliminary: multimodal AI
- **Opensource package: DocArray**
  - Motivation
  - Representing data
  - Transiting data
  - Storing data
  - Retrieving data
- Multimodal at scale in production

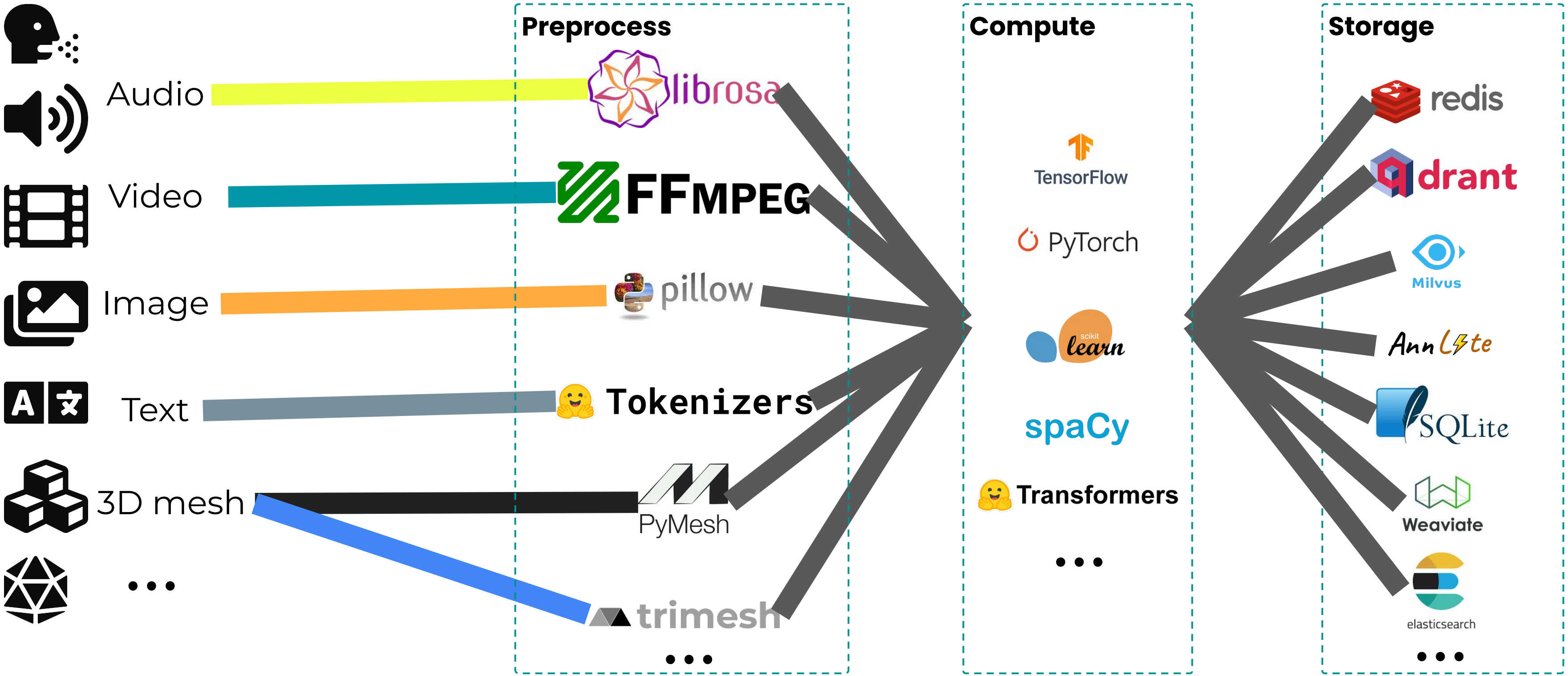
This tutorial may require technical knowledge. Familiarity with Python 3.7+ concepts like data classes could be helpful.

DocArray for representing, transiting, storing, searching multimodal data

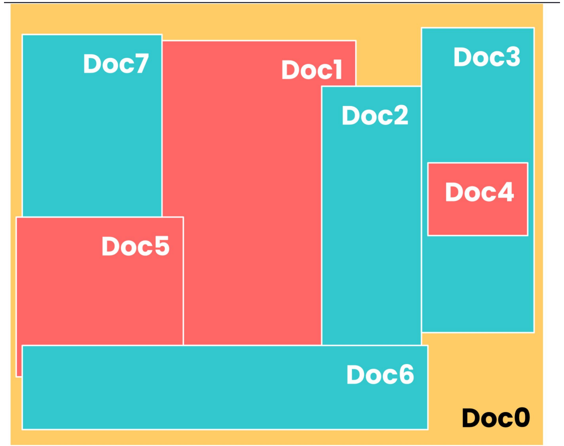
Representing multimodal data is a pain

- Lack of common interface for different modalities makes it difficult to work with multiple modalities at the same time.
- No easy way to represent unstructured and nested multimodal data.

Lack of common interface



No easy way to represent unstructured nested multimodal data



- Unstructured document
- Nested content
- Different modalities (text, image, …)

DocArray way of representing multimodal data

By the Way A Post Travel Destination

Everything to know about flying with pets, from picking your seat to keeping your animal calm

By Nathan Diller

```python
from docarray import dataclass, Docume
from docarray.typing import Image, Text, .


@dataclass
class WPArticle:
    banner: Image
    headline: Text
    meta: JSON


a = WPArticle(
    banner='dog-cat-flight.png',
    headline='Everything to know about fl
    meta={
        'author': 'Nathan Diller',
        'column': 'By the Way - A Post Tr
    },
)

doc = Document(a)
```

Frequent data transfer over network is expensive

Multimodal data is processed by multiple models and models are usually deployed in a distributed way.

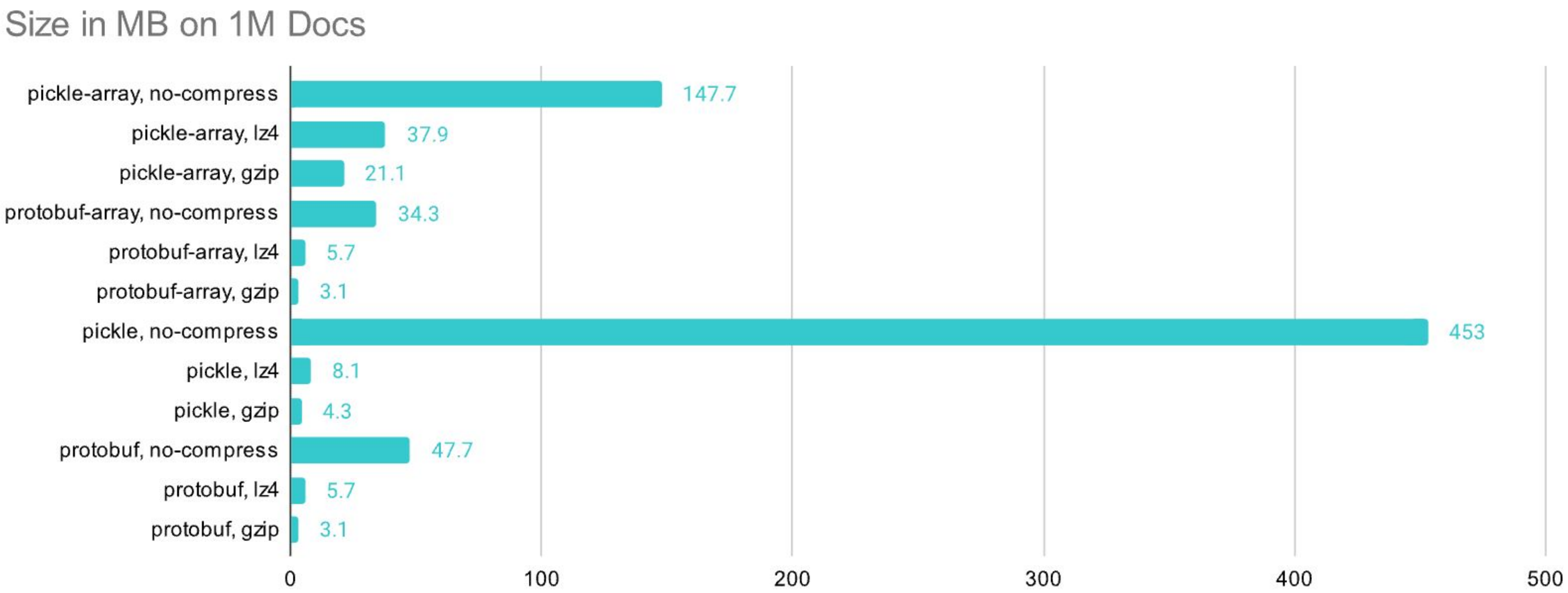
Data at rest	Data in use	Data in transit
Inactive data under very occasional changes, stored physically in database, warehouse, spreadsheet, archives, etc.	Active data under constant change, stored physically in database, warehouse, spreadsheet, etc.	Traversing a network or temporarily residing in computer memory to be read or updated

Performant serialization is important

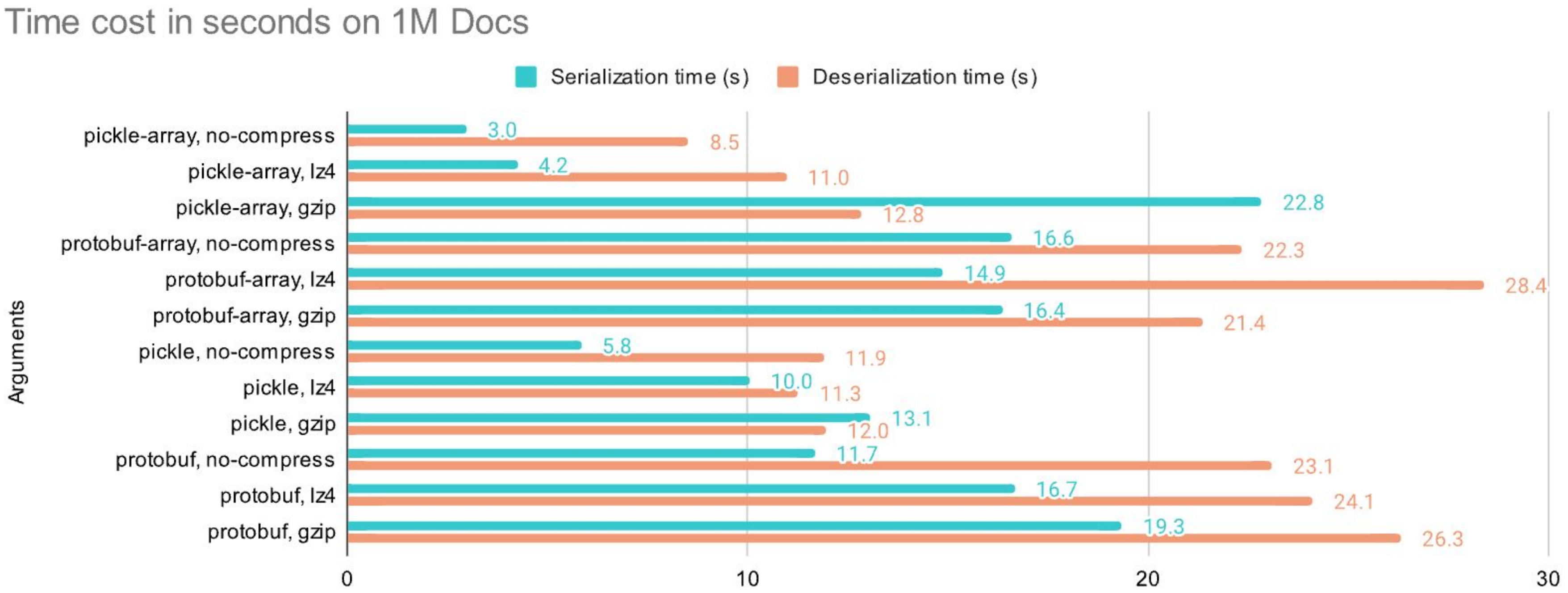
DocArray is designed to be “ready-to-wire” at anytime.

- JSON string: `.from_json()` / `.to_json()`
    - Pydantic model: `.from_pydantic_model()` / `.to_pydantic_model()`
- Bytes (compressed): `.from_bytes()` / `.to_bytes()`
    - Disk serialization: `.save_binary()` / `.load_binary()`
- Base64 (compressed): `.from_base64()` / `.to_base64()`
- Protobuf Message: `.from_protobuf()` / `.to_protobuf()`
- Python List: `.from_list()` / `.to_list()`
- Pandas Dataframe: `.from_dataframe()` / `.to_dataframe()`
- Cloud: `.push()` / `.pull()`

Binary serialization optimized for in-transit & at-rest



Binary serialization optimized for in-transit & at-rest



Storing nested data with databases is complicated

- Complex and nested schema are not directly supported in databases
- Explosion in numbers of vector databases with different APIs but no universal client

DocArray way of storing data

DocArray Storage

```python
from docarray import DocumentArray, Document

da = DocumentArray(storage='milvus',
                   config={'connection': 'example.db'})

with da:
    da.append(Document())
da.summary()
```

DocArray way of storing data

DocArray Storage

```python
from docarray import DocumentArray, Document

da = DocumentArray(storage='milvus',
                   config={'connectio

with da:
    da.append(Document())
da.summary()
```

```
'mivlus'
'qdrant'
'weaviate'
'elasticsearch'
'redis'
'opensearch'
'annlite'
'sqlite'
```

Vector Search via a consistent API

```python
from docarray import Document, DocumentArray
import numpy as np

n_dim = 3
da = DocumentArray(
    storage='annlite',
    config={'n_dim': n_dim, 'metric': 'Euclidean'},
)

with da:
    da.extend([Document(embedding=i * np.ones(n_dim)) for i in range(10)])

result = da.find(np.array([2, 2, 2]), limit=6)
result[:, 'embedding']
```

Vector Search via a consistent API

```python
from docarray import Document, DocumentArray
import numpy as np

n_dim = 3
da = DocumentArray(
    storage='annlite',
    config={'n_dim': n_dim, 'metric': 'Euclid
)

with da:
    da.extend([Document(embedding=i * np.ones

result = da.find(np.array([2, 2, 2]), limit=6
result[:, 'embedding']
```

Name	Construction	Vector search	Vector search + Filter	Filter
In memory	`DocumentArray()`	✅	✅	✅
`SQLite`	`DocumentArray(storage='sqlite')`	❌	❌	✅
`Weaviate`	`DocumentArray(storage='weaviate')`	✅	✅	✅
`Qdrant`	`DocumentArray(storage='qdrant')`	✅	✅	✅
`AnnLite`	`DocumentArray(storage='annlite')`	✅	✅	✅
`ElasticSearch`	`DocumentArray(storage='elasticsearch')`	✅	✅	✅
`Redis`	`DocumentArray(storage='redis')`	✅	✅	✅
`Milvus`	`DocumentArray(storage='milvus')`	✅	✅	✅

Quick Recap

- It’s like JSON, but for intensive computation.
- It’s like numpy.ndarray, but for unstructured data.
- It’s like pandas.DataFrame, but for nested and mixed media **data with embeddings.**
- It’s like Protobuf, but for data scientists and deep learning engineers.

Quick R

- It's like JSOI
- It's like num
- It's like pan
- It's like Prot

ings.

	DocArray	`numpy.ndarray`	JSON	`pandas.DataFrame`	Protobuf
Tensor/matrix data	✅	✅	❌	✅	✔
Text data	✅	❌	✅	✅	✅
Media data	✅	❌	❌	❌	❌
Nested data	✅	❌	✅	❌	✅
Mixed data of the above four	✅	❌	❌	❌	❌
Easy to (de)serialize	✅	❌	✅	✅	✅
Data validation (of the output)	✅	❌	❌	❌	✅
Pythonic experience	✅	✅	❌	✔	❌
IO support for filetypes	✅	❌	❌	❌	❌
Deep learning framework support	✅	✅	❌	❌	❌
multi-core/GPU support	✅	✔	❌	❌	❌
Rich functions for data types	✅	❌	❌	✅	❌

Hands-on DocArray

Install DocArray

To install `DocArray (0.33)`, you can use the following command:

```
pip install "docarray[full]"
```

https://docs.docarray.org/

For old DocArray, more compatibility and features

```
pip install "docarray[full]"==0.21
```

Representing data – Document

At the heart of `DocArray` lies the concept of `BaseDoc`.

The following Python code defines a `BannerDoc` class that can be used to represent the data of a website banner:

```
from docarray import BaseDoc
from docarray.typing import ImageUrl


class BannerDoc(BaseDoc):
    image_url: ImageUrl
    title: str
    description: str
```

Representing data – Document

You can then instantiate a `BannerDoc` object and access its attributes:

```python
banner = BannerDoc(
    image_url='https://example.com/image.png',
    title='Hello World',
    description='This is a banner',
)

assert banner.image_url == 'https://example.com/image.png'
assert banner.title == 'Hello World'
assert banner.description == 'This is a banner'
```

Representing multimodal data with nested structure

Let's say you want to represent a YouTube video in your application, perhaps to build a search system for YouTube videos.

A YouTube video is not only composed of a video, but also has a title, description, thumbnail (and more, but let's keep it simple).

All of these elements are from different modalities:

the title and description are text,

the thumbnail is an image,

and the video itself is, well, a video.

DocArray lets you represent all of this multimodal data in a single object.

Representing multimodal data with nested structure

First for the thumbnail image:

```
from docarray import BaseDoc
from docarray.typing import ImageUrl, ImageBytes


class ImageDoc(BaseDoc):
    url: ImageUrl
    bytes: ImageBytes = (
        None  # bytes are not always loaded in memory, so we make it optional
    )
```

Representing multimodal data with nested structure

Then for the video itself:

```python
from docarray import BaseDoc
from docarray.typing import VideoUrl, VideoBytes


class VideoDoc(BaseDoc):
    url: VideoUrl
    bytes: VideoBytes = (
        None  # bytes are not always loaded in memory, so we make it optional
    )
```

Representing multimodal data with nested structure

All the elements that compose a YouTube video are ready:

```python
from docarray import BaseDoc


class YouTubeVideoDoc(BaseDoc):
    title: str
    description: str
    thumbnail: ImageDoc
    video: VideoDoc
```

Representing multimodal data with nested structure

All the elements that compose a YouTube video are ready:

```
from docarray import BaseDoc


class YouTubeVideoDoc(BaseDoc):
    title: str
    description: str
    thumbnail: ImageDoc
    video: VideoDoc
```

You see here that `ImageDoc` and `VideoDoc` are also BaseDoc, and they are later used inside another BaseDoc`. This is what we call nested data representation.

BaseDoc can be nested to represent any kind of data hierarchy.

Representing multimodal data with nested structure

All the elements that compose a YouTube video are ready:

```python
from docarray import BaseDoc


class YouTubeVideoDoc(BaseDoc):
    title: str
    description: str
    thumbnail: ImageDoc
    video: VideoDoc
```

You see here that `ImageDoc` and `VideoDoc` are also BaseDoc, and they are later used inside another BaseDoc`. This is what we call nested data representation.

BaseDoc can be nested to represent any kind of data hierarchy.

This representation can be used to send or store data. You can even use it directly to train a machine learning Pytorch model on this representation.

Recap: representing multimodal data

- "Dataclass" look and feel, for defining the structure
- Strong typing, for defining modality
    - Python built-in types
    - Numpy types
    - URI types
        - **`Text`**
        - **`Image`**
        - **`Audio`**
        - **`Video`**
        - **`Mesh3D`**
        - **`PointCloud3D`**
    - Tensor types
        - **`ImageTensor`**
        - **`AudioTensor`**
        - **`VideoTensor`**
        - **`Embedding`**
    - **`Optional[]`**

```
from docarray import BaseDoc
from docarray.typing import ImageUrl, ImageBytes


class ImageDoc(BaseDoc):
    url: ImageUrl
    bytes: ImageBytes = (
        None  # bytes are not always loaded in m
    )
```

Representing an array of multimodal data

The fundamental building block of DocArray is the `BaseDoc` class which represents a *single* document, a *single* datapoint.

However, in machine learning we often need to work with an *array* of documents, and an *array* of data points.

We introduce

- `DocList` which is **a Python list** of `BaseDoc`s
- `DocVec` which is **a column-based representation** of `BaseDoc`s

Example of DocList

First you need to create a `Doc` class, our data schema. Let's say you want to represent a banner with an image, a title and a description:

```
from docarray import BaseDoc, DocList
from docarray.typing import ImageUrl


class BannerDoc(BaseDoc):
    image: ImageUrl
    title: str
    description: str
```

Example of DocList

First you need to create a `Doc` class, our data schema. Let's say you want to represent a banner with an image, a title and a description:

```
from docarray import BaseDoc, DocList
from docarray.typing import ImageUrl


cl
```

Let's instantiate several `BannerDoc` s:

```
banner1 = BannerDoc(
    image='https://example.com/image1.png',
    title='Hello World',
    description='This is a banner',
)

banner2 = BannerDoc(
    image='https://example.com/image2.png',
    title='Bye Bye World',
    description='This is (distopic) banner',
)
```

Example of DocList

`DocList` and `DocVec` are both `AnyDocArray`s. The following section will use `DocList` as an example, but the same applies to `DocVec`.

You can now collect them into a `DocList` of `BannerDoc` s:

```
docs = DocList[BannerDoc]([banner1, banner2])

docs.summary()
```

```
┌────────── DocList Summary ──────────┐
│                                     │
│   Type      DocList[BannerDoc]      │
│   Length    2                       │
│                                     │
└─────────────────────────────────────┘
┌───── Document Schema ─────┐
│                           │
│   BannerDoc               │
│   ├── image: ImageUrl     │
│   ├── title: str          │
│   └── description: str    │
│                           │
└───────────────────────────┘
```

Example of DocList

You can access documents inside it with the usual Python array API:

```python
print(docs[0])
```

```
BannerDoc(image='https://example.com/image1.png', title='Hello World', description
```

or iterate over it:

```python
for doc in docs:
    print(doc)
```

```
BannerDoc(image='https://example.com/image1.png', title='Hello World', description
BannerDoc(image='https://example.com/image2.png', title='Bye Bye World', descripti
```

Accessing member attribute at array level

At the document level:

```
print(banner1.image)
```

```
https://example.com/image1.png'
```

At the Array level:

```
print(docs.image)
```

```
['https://example.com/image1.png', 'https://example.com/image2.png']
```

Accessing member attribute at array level

At the document level:

```
print(banner1.image)
```

```
https://example.com/image1.png'
```

At the Array level:

```
print(docs.image)
```

```
['https://example.com/image1.png', 'h
```

You can even access the attributes of the nested `BaseDoc` at the Array level:

```
print(docs.banner.image)
```

```
['https://example.com/image1.png', 'https://example.com/image2.png']
```

This is just the same way that you would do it with BaseDoc:

```
print(page1.banner.image)
```

```
'https://example.com/image1.png'
```

DocList[DocType] syntax

`DocList[DocType]` creates a custom `DocList` that can only contain `DocType` Documents.

Non-typing DocList for heterogeneous data

```
from docarray import BaseDoc, DocList
from docarray.typing import ImageUrl, AudioUrl


class ImageDoc(BaseDoc):
    url: ImageUrl


class AudioDoc(BaseDoc):
    url: AudioUrl


docs = DocList(
    [
        ImageDoc(url='https://example.com/image1.png'),
        AudioDoc(url='https://example.com/audio1.mp3'),
    ]
)
```

Strong-typing DocList for homogeneous data

```
try:
    docs = DocList[ImageDoc](
        [
            ImageDoc(url='https://example.com/image1.png'),
            AudioDoc(url='https://example.com/audio1.mp3'),
        ]
    )
except ValueError as e:
    print(e)
```

```
ValueError: AudioDoc(
    id='e286b10f58533f48a0928460f0206441',
    url=AudioUrl('https://example.com/audio1.mp3', host_type='domain')
) is not a <class '__main__.ImageDoc'>
```

DocList vs DocVec

`DocList` is based on Python Lists. You can append, extend, insert, pop, and so on. In DocList, data is individually owned by each `BaseDoc` collect just different Document references.

Use `DocList` when you want to be able to rearrange or re-rank your data. One flaw of `DocList` is that none of the data is contiguous in memory, so you cannot leverage functions that require contiguous data without first copying the data in a continuous array.

`DocVec` is a columnar data structure. `DocVec` is always an array of homogeneous Documents. The idea is that every attribute of the `BaseDoc` will be stored in a contiguous array: a column.

DocList vs DocVec

Let's say you want to embed a batch of Images:

```
def embed(image: NdArray['batch_size', 3, 224, 224]):
    ...
```

DocList vs DocVec

```python
from docarray import BaseDoc
from docarray.typing import NdArray


class ImageDoc(BaseDoc):
    image: NdArray[
        3, 224, 224
    ] = None  # [3, 224, 224] this just mean we know in advance the shape of the te
```

DocList vs DocVec

```python
from docarray import BaseDoc
from docarray.typing import NdArray


class I
    ima

    ] =
```

```python
from docarray import DocList
import numpy as np

docs = DocList[ImageDoc](
    [ImageDoc(image=np.random.rand(3, 224, 224)) for _ in range(10)]
)

embed (np.stack(docs.image))
...
embed (np.stack(docs.image))
```

DocList vs DocVec

```
from docarray import BaseDoc
from docarray.typing import NdArray


class ImageDoc(BaseDoc):
    image:
        3,
    ] = No
```

```
from docarray import DocVec
import numpy as np

docs = DocVec[ImageDoc](
    [ImageDoc(image=np.random.rand(3, 224, 224)) for _ in range(10)]
)

embed(docs.image)
```

Access the view of Document in DocVec

If you access a document inside a `DocVec` you will get a document view. A document view is a view of the columnar data structure which looks and behaves like a `BaseDoc` instance. It is a `BaseDoc` instance but with a different way to access the data.

```
from docarray import DocVec

docs = DocVec[ImageDoc](
    [ImageDoc(image=np.random.rand(3, 224, 224)) for _ in range(10)]
)

my_doc = docs[0]

assert my_doc.is_view()  # True
```

whereas with DocList:

```
docs = DocList[ImageDoc](
    [ImageDoc(image=np.random.rand(3, 224, 224)) for _ in range(10)]
)

my_doc = docs[0]

assert not my_doc.is_view()  # False
```

Access the view of Document in DocVec

If you access a document inside a `DocVec` you will get a document view. A document view is a view of the columnar data structure which looks and behaves like a `BaseDoc` instance. It is a `BaseDoc` instance but with a different way to access the data.

you should use `DocVec` when you need to work with contiguous data, and you should use `DocList` when you need to rearrange or extend your data.

```
docs = DocList[ImageDoc](
    [ImageDoc(image=np.random.rand(3, 224, 224)) for _ in range(10)]
)

my_doc = docs[0]

assert not my_doc.is_view()  # False
```

Storing & retrieving via Vector Database

```python
from docarray import DocList, BaseDoc
from docarray.index import HnswDocumentIndex
import numpy as np

from docarray.typing import ImageUrl, ImageTensor, NdArray


class ImageDoc(BaseDoc):
    url: ImageUrl
    tensor: ImageTensor
    embedding: NdArray[128]


# create some data
dl = DocList[ImageDoc](
    [
        ImageDoc(
            url="https://upload.wikimedia.org/wikipedia/commons/2/2f/Alpamayo.jpg",
            tensor=np.zeros((3, 224, 224)),
            embedding=np.random.random((128,)),
        )
        for _ in range(100)
    ]
)

# create a Document Index
index = HnswDocumentIndex[ImageDoc](work_dir='/tmp/test_index2')


# index your data
index.index(dl)

# find similar Documents
query = dl[0]
results, scores = index.find(query, limit=10, search_field='embedding')
```

Storing & retrieving via Vector Database

```python
from docarray import DocList, BaseDoc
from docarray.index import HnswDocumentIndex
import numpy as np

from docarray.typing import ImageUrl, ImageTensor, NdArray


class ImageDoc(BaseDoc):
    url: ImageUrl
    tensor: ImageTensor
    embedding: NdArray[128]


# create some data
dl = DocList[ImageDoc](
    [
        ImageDoc(
            url="https://upload.wikimedia.org/wikipedia/commons/2/2f/Alpamayo.jpg",
            tensor=np.zeros((3, 224, 224)),
            embedding=np.random.random((128,)),
        )
        for _ in range(100)
    ]
)

# create a Document Index
index = HnswDocumentIndex[ImageDoc](work_dir='/tmp/test_index2')


# index your data
index.index(dl)

# find similar Documents
query = dl[0]
results, scores = index.find(query, limit=10, search_field='embedding')
```

Document Index: ORM for vector DBs

Document Index provides a unified interface to a number of vector databases.

You can think of Document Index as an **ORM for vector databases**.

Currently, DocArray supports the following vector databases:

- Weaviate | Docs
- Qdrant | Docs
- Elasticsearch v7 and v8 | Docs
- HNSWlib | Docs

*Old DocArray v0.21 supports Milvus, Redis, Opensearch

Construct a HNSWDocumentIndex

To use HnswDocumentIndex, you need to install extra dependencies with the following command: `pip install "docarray[hnswlib]"`

To create a Document Index, you first need a document that defines the schema of your index:

```
from docarray import BaseDoc
from docarray.index import HnswDocumentIndex
from docarray.typing import NdArray


class MyDoc(BaseDoc):
    embedding: NdArray[128]
    text: str


db = HnswDocumentIndex[MyDoc](work_dir='./my_test_db')
```

Construct a HNSWDocumentIndex

To use HnswDocumentIndex, you need to install extra dependencies with the following command: `pip install "docarray[hnswlib]"`

To create a Document Index, you first need a document th

```
from docarray import BaseDoc
from docarray.index import HnswDocumentIndex
from docarray.typing import NdArray


class MyDoc(BaseDoc):
    embedding: NdArray[128]
    text: str


db = HnswDocumentIndex[MyDoc](work_dir='./my_tes
```

In this code snippet, `HnswDocumentIndex` takes a schema of the form of `MyDoc`. The Document Index then *creates a column for each field in* `MyDoc`.

Construct a HNSWDocumentIndex

To use HnswDocumentIndex, you need to install extra dependencies with the following command: `pip install "docarray[hnswlib]"`

To create a Document Index, you first need a document th

```
from docarray import BaseDoc
from docarray.index import HnswDocumentIndex
from docarray.typing import NdArray


class MyDoc(BaseDoc):
    embedding: NdArray[128]
    text: str


db = HnswDocumentIndex[MyDoc](work_dir='./my_tes
```

In this code snippet, `HnswDocumentIndex` takes a schema of the form of `MyDoc`. The Document Index then *creates a column for each field in* `MyDoc`.

The column types in the backend database are determined by the type hints of the document's fields. Optionally, you can customize the database types for every field.

Construct a HNSWDocumentIndex

To use HnswDocumentIndex, you need to install extra dependencies with the following command: `pip install "docarray[hnswlib]"`

To create a Document Index, you first need a document th

```
from docarray import BaseDoc
from docarray.index import HnswDocumentIndex
from docarray.typing import NdArray


class MyDoc(BaseDoc):
    embedding: NdArray[128]
    text: str


db = HnswDocumentIndex[MyDoc](work_dir='./my_tes
```

In this code snippet, `HnswDocumentIndex` takes a schema of the form of `MyDoc`. The Document Index then *creates a column for each field in* `MyDoc`.

The column types in the backend database are determined by the type hints of the document's fields. Optionally, you can customize the database types for every field.

Most vector databases need to know the dimensionality of the vectors that will be stored. Here, that is automatically inferred from the type hint of the `embedding` field: `NdArray[128]` means that the database will store vectors with 128 dimensions.

Index data

Now that you have a Document Index, you can add data to it, using the index() method:

```
import numpy as np
from docarray import DocList

# create some random data
docs = DocList[MyDoc](
    [MyDoc(embedding=np.random.rand(128), text=f'text {i}') for i in range(100)]
)

# index the data
db.index(docs)
```

Index data

Now that you have a Document Index, you can add data to it, using the index() method:

```
import numpy as np
from docarray import DocList

# create some random data
docs = DocList[MyDoc](
    [MyDoc(embedding=np.random.rand(128), text=f'text {i}'
)

# index the data
db.index(docs)
```

```
from docarray import BaseDoc
from docarray.index import HnswDocume
from docarray.typing import NdArray


class MyDoc(BaseDoc):
    embedding: NdArray[128]
    text: str


db = HnswDocumentIndex[MyDoc](work_dir='./my_test_db')
```

As you can see, `DocList[MyDoc]` and `HnswDocumentIndex[MyDoc]` are both parameterized with `MyDoc`. This means that they share the same schema, and in general, the schema of a Document Index and the data that you want to store need to have compatible schemas

Vector search

Search by Document | Search by raw vector

```
# create a query Document
query = MyDoc(embedding=np.random.rand(128), text='query')

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Vector search

Search by Document | Search by raw vector

```python
# create a query Document
query = MyDoc(embedding=np.random.rand(128), text='query')

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Search by Document | Search by raw vector

```python
# create a query vector
query = np.random.rand(128)

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Vector search

Search by Document | Search by raw vector

```python
# create a query Document
query = MyDoc(embedding=np.random.rand(128), text='query')

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Search by Document | Search by raw vector

```python
# create a query vector
query = np.random.rand(128)

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Search by Documents | Search by raw vectors

```python
# create some query Documents
queries = DocList[MyDoc](
    MyDoc(embedding=np.random.rand(128), text=f'query {i}') for i in range(3)
)

# find similar Documents
matches, scores = db.find_batched(queries, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches[0].text=}')
print(f'{scores=}')
```

Vector search

Search by Document | Search by raw vector

```python
# create a query Document
query = MyDoc(embedding=np.random.rand(128), text='query')

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Search by Document | Search by raw vector

```python
# create a query vector
query = np.random.rand(128)

# find similar Documents
matches, scores = db.find(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches.text=}')
print(f'{scores=}')
```

Search by Documents | Search by raw vectors

```python
# create some query Documents
queries = DocList[MyDoc](
    MyDoc(embedding=np.random.rand(128), text=f'query {i}') for i in range(3)
)

# find similar Documents
matches, scores = db.find_batched(queries, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches[0].text=}')
print(f'{scores=}')
```

Search by Documents | Search by raw vectors

```python
# create some query vectors
query = np.random.rand(3, 128)

# find similar Documents
matches, scores = db.find_batched(query, search_field='embedding', limit=5)

print(f'{matches=}')
print(f'{matches[0].text=}')
print(f'{scores=}')
```

Hybrid search through the query builder

Document Index supports atomic operations for vector similarity search, text search and filter search.

To combine these operations into a single, hybrid search query, you can use the query builder that is accessible through build_query():

```
# prepare a query
q_doc = MyDoc(embedding=np.random.rand(128), text='query')

query = (
    db.build_query()  # get empty query object
    .find(query=q_doc, search_field='embedding')  # add vector similarity search
    .filter(filter_query={'text': {'$exists': True}})  # add filter search
    .build()  # build the query
)

# execute the combined query and return the results
results = db.execute_query(query)
print(f'{results=}')
```

Customize vector DB configuration

```
db = HnswDocumentIndex[MyDoc](work_dir='/tmp/my_db')

db.configure(
    default_column_config={
        np.ndarray: {
            'dim': -1,
            'index': True,
            'space': 'ip',
            'max_elements': 2048,
            'ef_construction': 100,
            'ef': 15,
            'M': 8,
            'allow_replace_deleted': True,
            'num_threads': 5,
        },
        None: {},
    }
)
```

Indexing and searching multimodal data

In the following example you can see a complex schema that contains nested Documents. The `YouTubeVideoDoc` contains a `VideoDoc` and an `ImageDoc`, alongside some "basic" fields:

Year in Review: 2021 in Graphic Design
Linus Boman
119K views • 1 year ago

```python
from docarray.typing import ImageUrl, VideoUrl, AnyTensor


# define a nested schema
class ImageDoc(BaseDoc):
    url: ImageUrl
    tensor: AnyTensor = Field(space='cosine', dim=64)


class VideoDoc(BaseDoc):
    url: VideoUrl
    tensor: AnyTensor = Field(space='cosine', dim=128)


class YouTubeVideoDoc(BaseDoc):
    title: str
    description: str
    thumbnail: ImageDoc
    video: VideoDoc
    tensor: AnyTensor = Field(space='cosine', dim=256)


# create a Document Index
doc_index = HnswDocumentIndex[YouTubeVideoDoc](work_dir='/tmp2')

# create some data
index_docs = [
    YouTubeVideoDoc(
        title=f'video {i+1}',
        description=f'this is video from author {10*i}',
        thumbnail=ImageDoc(url=f'http://example.ai/images/{i}', tensor=np.ones(64)),
        video=VideoDoc(url=f'http://example.ai/videos/{i}', tensor=np.ones(128)),
        tensor=np.ones(256),
    )
    for i in range(8)
]

# index the Documents
doc_index.index(index_docs)
```

Indexing and searching multimodal data

You can perform search on any nesting level by using the dunder operator to specify the field defined in the nested data.

```python
# create a query Document
query_doc = YouTubeVideoDoc(
    title=f'video query',
    description=f'this is a query video',
    thumbnail=ImageDoc(url=f'http://example.ai/images/1024', tensor=np.ones(64)),
    video=VideoDoc(url=f'http://example.ai/videos/1024', tensor=np.ones(128)),
    tensor=np.ones(256),
)

# find by the `youtubevideo` tensor; root level
docs, scores = doc_index.find(query_doc, search_field='tensor', limit=3)

# find by the `thumbnail` tensor; nested level
docs, scores = doc_index.find(query_doc, search_field='thumbnail__tensor', limit=3)

# find by the `video` tensor; neseted level
docs, scores = doc_index.find(query_doc, search_field='video__tensor', limit=3)
```

Nested DocList with subindex

Documents can be nested by containing a `DocList` of other documents, which is a slightly more complicated scenario than the previous one.

In this case, the nested DocList will be represented as a **new sub-index** (or table, collection, etc., depending on the database backend), that is linked with **the parent index** (table, collection, ...).

```python
class ImageDoc(BaseDoc):
    url: ImageUrl
    tensor_image: AnyTensor = Field(space='cosine', dim=64)


class VideoDoc(BaseDoc):
    url: VideoUrl
    images: DocList[ImageDoc]
    tensor_video: AnyTensor = Field(space='cosine', dim=128)


class MyDoc(BaseDoc):
    docs: DocList[VideoDoc]
    tensor: AnyTensor = Field(space='cosine', dim=256)


# create a Document Index
doc_index = HnswDocumentIndex[MyDoc](work_dir='/tmp3')

# create some data
index_docs = [
    MyDoc(
        docs=DocList[VideoDoc](
            [
                VideoDoc(
                    url=f'http://example.ai/videos/{i}-{j}',
                    images=DocList[ImageDoc](
                        [
                            ImageDoc(
                                url=f'http://example.ai/images/{i}-{j}-{k}',
                                tensor_image=np.ones(64),
                            )
                            for k in range(10)
                        ]
                    ),
                    tensor_video=np.ones(128),
                )
                for j in range(10)
            ]
        ),
        tensor=np.ones(256),
    )
    for i in range(10)
]

# index the Documents
doc_index.index(index_docs)
```

Search by subindex

```python
# find by the `VideoDoc` tensor
root_docs, sub_docs, scores = doc_index.find_subindex(
    np.ones(128), subindex='docs', search_field='tensor_video', limit=3
)

# find by the `ImageDoc` tensor
root_docs, sub_docs, scores = doc_index.find_subindex(
    np.ones(64), subindex='docs__images', search_field='tensor_image', limit=3
)

```

Transiting data over network

Sending via REST API/JSON -> Backend: FastAPI

```
import numpy as np
from fastapi import FastAPI
from docarray.base_doc import DocArrayResponse
from docarray import BaseDoc
from docarray.documents import ImageDoc
from docarray.typing import NdArray

class InputDoc(BaseDoc):
    img: ImageDoc
    text: str


class OutputDoc(BaseDoc):
    embedding_clip: NdArray
    embedding_bert: NdArray


app = FastAPI()


@app.post("/embed/", response_model=OutputDoc, response_class=DocArrayResponse)
async def create_item(doc: InputDoc) -> OutputDoc:
    ## call my fancy model to generate the embeddings
    doc = OutputDoc(
        embedding_clip=embed(doc.image)), embedding_bert=embed(doc.text))
    )
    return doc

```

Sending via gRPC/ws -> Backend: Jina microservice

Transiting data over network

Sending via REST API/JSON -> Backend: FastAPI

Sending via gRPC/ws -> Backend: Jina microservice

```python
import numpy as np
from fastapi import FastAPI
from docarray.base_doc import DocArrayResponse
from docarray import BaseDoc
from docarray.documents import ImageDoc
from docarray.typing import NdArray

class InputDoc(BaseDoc):
    img: ImageDoc
    text: str


class OutputDoc(BaseDoc):
    embedding_clip: NdArray
    embedding_bert: NdArray


app = FastAPI()


@app.post("/e
async def cre
    ## call m
    doc = Out
        embed
    )
    return do

```

```python
async with AsyncClient(app=app, base_url="http://test") as ac:
    response = await ac.post("/doc/", data=docs.to_json())  # sending docs as json

assert response.status_code == 200
# You can read FastAPI's response in the following way
docs = DocList[TextDoc].from_json(response.content.decode())
```

Transiting data over network

Sending via gRPC/ws -> Backend: Jina microservice

```python
class WhisperExecutor(Executor):
    def __init__(self, device: str, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.model = whisper.load_model("medium.en", device=device)

    @requests
    def transcribe(self, docs: DocList[AudioURL], **kwargs) -> DocList[Response]:
        response_docs = DocList[Response]()
        for doc in docs:
            transcribed_text = self.model.transcribe(str(doc.audio))['text']
            response_docs.append(Response(text=transcribed_text))

        return response_doc

```

Transiting data over network

Sending via gRPC/ws -> Backend: Jina microservice

```python
class WhisperExecutor(Executor):
    def __init__(self, device: str, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.model = whisper.load_model("medium.en", device=device)

    @requests
    def transcribe(self, docs: DocList[AudioURL], **kwargs) -> DocList[Response]:
        response_docs = DocList[Respo
        for doc in docs:
            transcribed_text = self.m
            response_docs.append(Resp

        return response_doc

```

```python
dep = Deployment(
    uses=WhisperExecutor, uses_with={'device': "cpu"}, port=12349,
timeout_ready=-1
)

with dep:
    docs = d.post(
        on='/transcribe',
        inputs=[AudioURL(audio='resources/audio.mp3')],
        return_type=DocList[Response],
    )

print(docs[0].text)

```

Agenda

- Preliminary: multimodal AI
- Opensource package: DocArray
  - Motivation
  - Representing data
  - Transiting data
  - Storing data
  - Retrieving data
- **Multimodal at scale in production**

This tutorial may require technical knowledge. Familiarity with Python 3.7+ concepts like data classes could be helpful.

An end to end example

https://docs.docarray.org/how_to/multimodal_training_and_serving/

Thanks for your attention

jina.ai
@JinaAI_
han.xiao@jina.ai

Berlin · Beijing · Shenzhen