# What is a Vector Embedding?

A vector embedding is a way to represent text (or any data) as a list of numbers — a vector — in a high-dimensional space.

A word like “apple” is represented as a vector:

[0.12, -0.43, 0.88, ..., 0.04] → maybe 384 or 768 numbers

Each number is a dimension in a high-dimensional space.

So if you have 384 values, then that vector lives in 384-dimensional space (liên tưởng đến 2D, 3D space).

These vectors capture meaning based on context and semantics, not just raw characters or words.

Example in code:

from sentence\_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")

sentence = "How do I reset my password?"

embedding = model.encode(sentence)

print(embedding[:10]) # Just the first 10 values

print("Length:", len(embedding)) # Should be 384

# Models that generate vector embedding

The deep neural network **models** that generate embeddings (like Sentence Transformers, BERT, MiniLM) are the product of years of research and training by top AI labs.

The sentence-transformers library (by UKPLab) wraps models like BERT, RoBERTa, MiniLM, etc., and trains them for sentence embeddings using contrastive learning.

## How it works?

### 1. Input Text → Tokens

"How do I reset my password?"

→ ["[CLS]", "How", "do", "I", "reset", "my", "password", "?", "[SEP]"]

### 2. Token Embedding Layer

Each token is mapped to a fixed-size vector (usually 768D or 1024D).

These vectors are learned during pretraining on large corpora (like Wikipedia or books).

Example:

"reset" → [0.02, -0.13, 0.55, ..., 0.07] # 768-dimensional vector

### 3. Transformer Layers

A Transformer consists of multiple encoder layers (e.g., 12 for BERT-base).

Each token’s final vector reflects its meaning in context.

Example: "bank" in “river bank” ≠ "bank" in “money in the bank”

### 4. Output

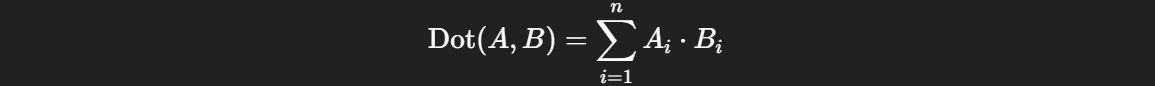
After all transformer layers, each token has a deeply contextualized embedding. These represent the meaning of that word **in that specific sentence**.

# How Does Vector Search Work in Qdrant?

<https://qdrant.tech/documentation/overview/vector-search/>

# What are the distance metrics?

## 🔵 1. Dot Product — Dot



It measures how aligned two vectors are.

Doesn’t care about magnitude unless you're using normalized vectors.

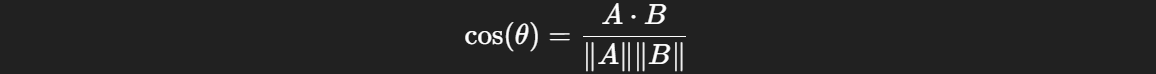
Use case:

✅ Works well when vectors are not normalized (e.g., some dense transformer embeddings).

⚠️ High dot product = high similarity.

Note: Qdrant converts it into a distance internally, so a larger dot product = smaller distance.

## 🟣 2. Cosine Similarity — Cosine



Measures the angle between two vectors.

Range: [-1, 1] → closer to 1 means more similar.

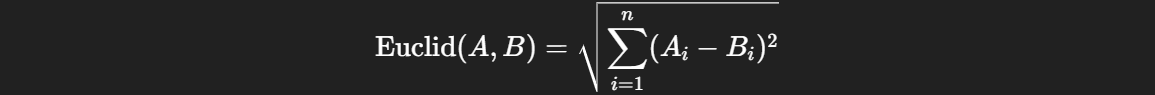
Use case:

✅ Great for text embeddings (e.g., sentence transformers).

✅ Ignores vector length, focuses on direction.

Very popular for semantic search.

## 🟢 3. Euclidean Distance — Euclid



Classic straight-line distance.

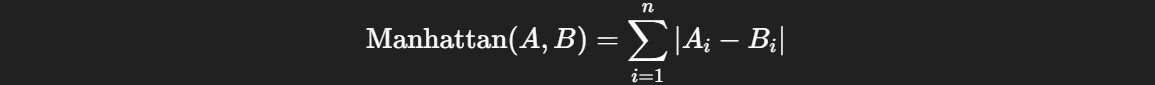
Takes both direction and magnitude into account.

Use case:

✅ Good for spatial data or when scale matters.

⚠️ Sensitive to vector length — may not work well for unnormalized embeddings.

## 🟠 4. Manhattan Distance — Manhattan



Also called L1 distance.

Measures distance like walking city blocks (no diagonal cuts).

Use case:

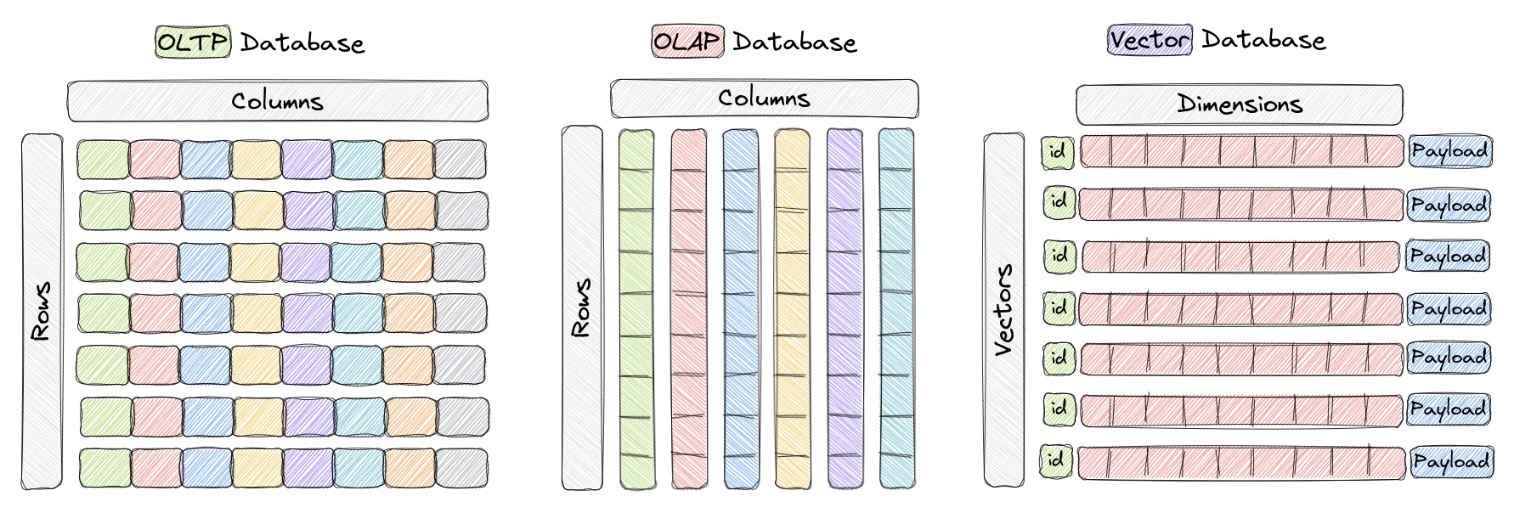
✅ Useful when differences across individual dimensions matter.

⚠️ Less common for embeddings, more used in simple tabular data or where outliers are important.

# What is a Vector Database?

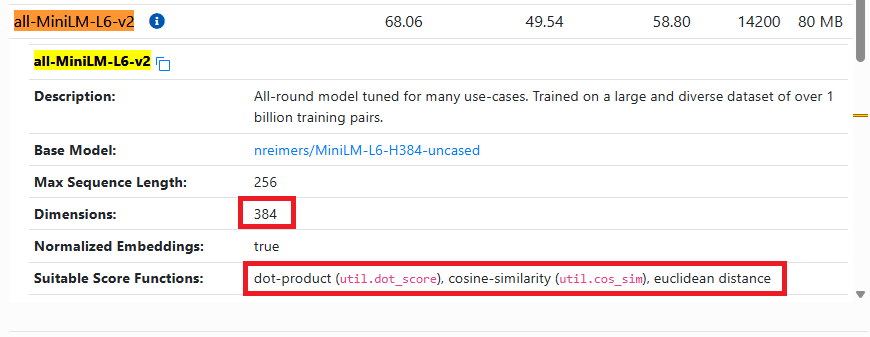
A Vector Database stores and retrieves vector embeddings — numerical representations of data (like text, images, etc.) in a high-dimensional space. These vectors are typically generated by AI models such as BERT, OpenAI, or Sentence Transformers.

Unlike traditional databases that match exact values, vector DBs enable semantic similarity search — finding data that is similar in meaning, not necessarily in exact words.



## How to know the vector's dimension space of your model?

<https://sbert.net/docs/sentence_transformer/pretrained_models.html#original-models>



**✅ Full Stack Plan: FastAPI + Qdrant Chatbot Backend**

**🏗️ Architecture Overview**

1. css
2. CopyEdit
3. [User Input]
4. ↓
5. [React Chat Window]
6. ↓
7. [FastAPI Backend]
8. ↙ ↘
9. [Qdrant Vector DB] [LLM (e.g., OpenAI)]
10. ↘ ↙
11. [Contextual Response]
12. ↓
13. [User sees chatbot reply]

**🔧 Step-by-Step Implementation**

**1. Install Dependencies**

Use a virtual environment (optional):

1. bash
2. CopyEdit
3. pip install fastapi uvicorn qdrant-client sentence-transformers openai python-dotenv

**2. Directory Structure**

1. bash
2. CopyEdit
3. chatbot-backend/
4. │
5. ├── app/
6. │ ├── main.py # FastAPI app
7. │ ├── qdrant\_utils.py # Qdrant connection & operations
8. │ ├── embedder.py # Embedding logic
9. │ └── prompt.py # LLM prompt formatting
10. │
11. ├── .env # API keys
12. └── requirements.txt

**3. Connect to Qdrant (qdrant\_utils.py)**

1. python
2. CopyEdit
3. from qdrant\_client import QdrantClient
4. from qdrant\_client.models import Distance, VectorParams, PointStruct
5. client = QdrantClient("http://localhost:6333")
6. def create\_collection(collection\_name="website\_docs"):
7. if not client.collection\_exists(collection\_name=collection\_name):
8. client.recreate\_collection(
9. collection\_name=collection\_name,
10. vectors\_config=VectorParams(size=384, distance=Distance.COSINE)
11. )

**4. Embedding Your Data (embedder.py)**

1. python
2. CopyEdit
3. from sentence\_transformers import SentenceTransformer
4. model = SentenceTransformer("all-MiniLM-L6-v2")
5. def embed\_text(texts: list[str]) -> list[list[float]]:
6. return model.encode(texts).tolist()

**5. Upserting Data to Qdrant**

1. python
2. CopyEdit
3. def add\_documents(docs: list[dict], collection="website\_docs"):
4. vectors = embed\_text([doc["content"] for doc in docs])
5. payloads = docs
6. points = [
7. PointStruct(id=i, vector=vectors[i], payload=payloads[i])
8. for i in range(len(docs))
9. ]
10. client.upsert(collection\_name=collection, points=points)

Each doc in docs is a dictionary like:

1. json
2. CopyEdit
3. {
4. "content": "We are a design studio focused on UX...",
5. "url": "/about",
6. "title": "About Us"
7. }

**6. Search Relevant Chunks**

1. python
2. CopyEdit
3. def search\_similar(query: str, collection="website\_docs", k=5):
4. query\_vector = embed\_text([query])[0]
5. hits = client.search(collection\_name=collection, query\_vector=query\_vector, limit=k)
6. return hits

**7. Prompt LLM with Context (prompt.py)**

1. python
2. CopyEdit
3. import openai
4. import os
5. openai.api\_key = os.getenv("OPENAI\_API\_KEY")
6. def generate\_answer(query: str, contexts: list[str]):
7. context\_str = "\n\n".join(contexts)
8. prompt = f"""Answer the question using the context below:
10. Context:
11. {context\_str}
12. Question:
13. {query}
14. Answer:"""
15. response = openai.ChatCompletion.create(
16. model="gpt-3.5-turbo", messages=[{"role": "user", "content": prompt}]
17. )
18. return response.choices[0].message["content"]

**8. FastAPI App (main.py)**

1. python
2. CopyEdit
3. from fastapi import FastAPI, Request
4. from app.qdrant\_utils import search\_similar
5. from app.prompt import generate\_answer
6. app = FastAPI()
7. @app.post("/chat")
8. async def chat\_endpoint(request: Request):
9. body = await request.json()
10. query = body["message"]
11. hits = search\_similar(query)
12. contexts = [hit.payload["content"] for hit in hits]
13. answer = generate\_answer(query, contexts)
14. return {"response": answer}

**9. Run Your API**

1. bash
2. CopyEdit
3. uvicorn app.main:app --reload

**🧪 Test Locally**

POST a request:

1. bash
2. CopyEdit
3. curl -X POST http://localhost:8000/chat \
4. -H "Content-Type: application/json" \
5. -d '{"message": "Tell me about your services."}'

**🔜 Future-Ready Upgrades**

| **Feature** | **Now** | **Future** |
| --- | --- | --- |
| Vector DB | Qdrant local | Qdrant Cloud |
| Embeddings | SBERT | OpenAI, custom fine-tuned |
| LLM | OpenAI GPT-3.5 | Self-host LLaMA/vLLM |
| Data | Static scrape | Dynamic scraper, live sync |
| Context strategy | Top-K | Re-ranking, MMR, context compression |

Would you like me to generate the full backend code scaffold for you as a downloadable zip or GitHub repo? Or should we continue with just the key files?