nanopq Documentation

Yusuke Matsui

Jul 24, 2019

Contents

CHAPTER 1

Installation

You can install the package via pip. This library works with Python 3.5+ on linux.

```
$ pip install nanopq
```

CHAPTER 2

Contents

2.1 Tutorial

2.1.1 Basic of PQ

This tutorial shows the basic usage of Nano Product Quantization Library (nanopq). Product quantization (PQ) is one of the most widely used algorithms for memory-efficient approximated nearest neighbor search, especially in the field of computer vision. This package contains a vanilla implementation of PQ and its improved version, Optimized Product Quantization (OPQ).

Let us first prepare 10,000 12-dim vectors for database, 2,000 vectors for training, and a query vector. They must be np.ndarray with np.float32.

```
import nanopq
import numpy as np

X = np.random.random((10000, 12)).astype(np.float32)
Xt = np.random.random((2000, 12)).astype(np.float32)
query = np.random.random((12, )).astype(np.float32)
```

The basic idea of PQ is to split an input *D*-dim vector into *M* *D/M*-dim sub-vectors. Each sub-vector is then quantized into an identifier of the nearest codeword.

First of all, a PQ class (`nanopq.PQ`) is instantiated with the number of sub-vector (*M*) and the number of codeword for each sub-space (*Ks*).

```
pq = nanopq.PQ(M=4, Ks=256, verbose=True)
```

Note that *M* is a parameter to control the trade off of accuracy and memory-cost. If you set larger *M*, you can achieve better quantization (i.e., less reconstruction error) with more memory usage. *Ks* specifies the number of codewords for quantization. This is tyically 256 so that each sub-space is represented by 8 bits = 1 byte = np.uint8. The memory cost for each pq-code is *M * log_2 Ks* bits.

Next, you need to train this quantizer by running k-means clustering for each sub-space of the training vectors.

```
pq.fit(vecs=Xt, iter=20, seed=123)
```

If you do not have training data, you can simply use the database vectors (or a subset of them) for training: `pq.fit(vecs=X[:1000])`. After that, you can see codewords by *pq.codewords*.

Note that, alternatively, you can instantiate and train an instance in one line if you want:

```
pq = nanopq.PQ(M=4, Ks=256).fit(vecs=Xt, iter=20, seed=123)
```

Given this quantizer, database vectors can be encoded to PQ-codes.

```
X_code = pq.encode(vecs=X)
```

The resulting PQ-code (a list of identifiers) can be regarded as a memory-efficient representation of the original vector, where the shape of *X_code* is (N, M).

For the querying phase, the asymmetric distance between the query and the database PQ-codes can be computed efficiently.

```
dt = pq.dtable(query=query)  # dt.dtable.shape = (4, 256)
dists = dt.adist(codes=X_code)  # (10000,)
```

For each query, a distance table (*dt*) is first computed online. *dt* is an instance of `nanopq.DistanceTable` class, which is a wrapper of the actual table (np.array), *dtable*. The elements of *dt.dtable* are computed by comparing each sub-vector of the query to the codewords for each sub-subspace. More specifically, *dt.dtable[m][ks]* contains the squared Euclidean distance between (1) the *m*-th sub-vector of the query and (2) the *ks*-th codeword for the *m*-th sub-space (*pq.codewords[m][ks]*).

Given *dtable*, the asymmetric distance to each PQ-code can be efficiently computed (*adist*). This can be achieved by simply fetching pre-computed distance value (the element of *dtable*) using PQ-codes.

Note that the above two lines can be chained in a single line.

```
dists = pq.dtable(query=query).adist(codes=X_code)  # (10000,)
```

The nearest feature is the one with the minimum distance.

```
min_n = np.argmin(dists)
```

Note that the search result is similar to that by the exact squared Euclidean distance.

```
# The first 30 results by PQ
print(dists[:30])

# The first 30 results by the exact scan
dists_exact = np.linalg.norm(X - query, axis=1) ** 2
print(dists_exact[:30])
```

2.1.2 Decode (reconstruction)

Given PQ-codes, the original *D*-dim vectors can be approximately reconstructed by fetching codewords

```
X_reconstructed = pq.decode(codes=X_code)  # (10000, 12)
# The following two results should be similar
print(X[:3])
print(X_reconstructed[:3])
```

2.1.3 I/O by pickling

A PQ instance can be pickled. Note that PQ-codes can be pickled as well because they are just a numpy array.

```
import pickle

with open('pq.pkl', 'wb') as f:
    pickle.dump(pq, f)

with open('pq.pkl', 'rb') as f:
    pq_dumped = pickle.load(f)  # pq_dumped is identical to pq
```

2.1.4 Optimized PQ (OPQ)

Optimized Product Quantizaion (OPQ; *nanopq.OPQ*), which is an improved version of PQ, is also available with the same interface as follows.

```
opq = nanopq.OPQ(M=4).fit(vecs=Xt, pq_iter=20, rotation_iter=10, seed=123)
X_code = opq.encode(vecs=X)
dists = opq.dtable(query=query).adist(codes=X_code)
```

The resultant codes approximate the original vectors finer, that usually leads to the better search accuracy. The training of OPQ will take much longer time compared to that of PQ.

2.1.5 Relation to PQ in faiss

Note that PQ is implemented in Faiss, whereas Faiss is one of the most powerful ANN libraries developed by the original authors of PQ:

- faiss.ProductQuantizer: The core component of PQ.
- faiss.IndexPQ: The search interface. IndexPQ = ProductQuantizer + PQ-codes.

Since Faiss is highly optimized, you should use PQ in Faiss if the runtime is your most important criteria. The difference between PQ in *nanopq* and that in Faiss is highlighted as follows:

- Our *nanopq* can be installed simply by pip without any third party dependencies such as Intel MKL
- The core part of *nanopq* is a vanilla implementation of PQ written in a single python file. It would be easier to extend that for further applications.
- A standalone OPQ is implemented.
- The result of *nanopq.DistanceTable.adist()* is **not** sorted. This would be useful when you would like to know not only the nearest but also the other results.
- The accuracy (reconstruction error) of *nanopq.PQ* and that of *faiss.IndexPQ* are almost same.

You can convert an instance of *nanopq.PQ* to/from that of *faiss.IndexPQ* by *nanopq.nanopq_to_faiss()* or *nanopq.faiss_to_nanopq()*.

```
# nanopq -> faiss
pq_nanopq = nanopq.PQ(M).fit(vecs=Xt)
pq_faiss = nanopq.nanopq_to_faiss(pq_nanopq)  # faiss.IndexPQ

# faiss -> nanopq
import faiss
```

(continues on next page)

(continued from previous page)

```
pq_faiss2 = faiss.IndexPQ(D, M, nbits)
pq_faiss2.train(x=Xt)
pq_faiss2.add(x=Xb)
# pq_nanopq2 is an instance of nanopq.PQ.
# Cb is encoded vectors
pq_nanopq2, Cb = nanopq.faiss_to_nanopq(pq_faiss2)
```

2.2 API Reference

2.2.1 Product Quantization (PQ)

class nanopq.**PQ**(*M*, *Ks=256*, *verbose=True*)

Pure python implementation of Product Quantization (PQ) [Jegou11].

For the indexing phase of database vectors, a *D*-dim input vector is divided into *M* *D*/*M*-dim sub-vectors. Each sub-vector is quantized into a small integer via *Ks* codewords. For the querying phase, given a new *D*-dim query vector, the distance beween the query and the database PQ-codes are efficiently approximated via Asymmetric Distance.

All vectors must be np.ndarray with np.float32

Parameters

- **M** (*int*) – The number of sub-space
- **Ks** (*int*) – The number of codewords for each subspace (typically 256, so that each sub-vector is quantized into 256 bits = 1 byte = uint8)
- **verbose** (*bool*) – Verbose flag

M

The number of sub-space

Type int

Ks

The number of codewords for each subspace

Type int

verbose

Verbose flag

Type bool

code_dtype

dtype of PQ-code. Either np.uint{8, 16, 32}

Type object

codewords

shape=(M, Ks, Ds) with dtype=np.float32. codewords[m][ks] means ks-th codeword (Ds-dim) for m-th subspace

Type np.ndarray

Ds

The dim of each sub-vector, i.e., Ds=D/M

Type int

fit(*vecs*, *iter=20*, *seed=123*)
Given training vectors, run k-means for each sub-space and create codewords for each sub-space.

This function should be run once first of all.

Parameters

- **vecs** (*np.ndarray*) – Training vectors with shape=(N, D) and dtype=np.float32.
- **iter** (*int*) – The number of iteration for k-means
- **seed** (*int*) – The seed for random process

Returns self

Return type object

encode(*vecs*)
Encode input vectors into PQ-codes.

Parameters **vecs** (*np.ndarray*) – Input vectors with shape=(N, D) and dtype=np.float32.

Returns PQ codes with shape=(N, M) and dtype=self.code_dtype

Return type np.ndarray

decode(*codes*)
Given PQ-codes, reconstruct original D-dimensional vectors approximately by fetching the codewords.

Parameters **codes** (*np.ndarray*) – PQ-cdoes with shape=(N, M) and dtype=self.code_dtype. Each row is a PQ-code

Returns Reconstructed vectors with shape=(N, D) and dtype=np.float32

Return type np.ndarray

dtable(*query*)
Compute a distance table for a query vector. The distances are computed by comparing each sub-vector of the query to the codewords for each sub-subspace. *dtable[m][ks]* contains the squared Euclidean distance between the *m*-th sub-vector of the query and the *ks*-th codeword for the *m*-th sub-space (*self.codewords[m][ks]*).

Parameters **query** (*np.ndarray*) – Input vector with shape=(D,) and dtype=np.float32

Returns Distance table. which contains dtable with shape=(M, Ks) and dtype=np.float32

Return type *nanopq.DistanceTable*

2.2.2 Distance Table

class nanopq.**DistanceTable**(*dtable*)
Distance table from query to codeworkds. Given a query vector, a PQ/OPQ instance compute this DistanceTable class using `PQ.dtable()` or `OPQ.dtable()`. The Asymmetric Distance from query to each database codes can be computed by `DistanceTable.adist()`.

Parameters **dtable** (*np.ndarray*) – Distance table with shape=(M, Ks) and dtype=np.float32 computed by `PQ.dtable()` or `OPQ.dtable()`

dtable
Distance table with shape=(M, Ks) and dtype=np.float32. Note that dtable[m][ks] contains the squared Euclidean distance between (1) m-th sub-vector of query and (2) ks-th codeword for m-th subspace.

Type np.ndarray

adist(*codes*)
Given PQ-codes, compute Asymmetric Distances between the query (self.dtable) and the PQ-codes.

Parameters **codes** (*np.ndarray*) – PQ codes with shape=(N, M) and dtype=pq.code_dtype where pq is a pq instance that creates the codes

Returns Asymmetric Distances with shape=(N,) and dtype=np.float32

Return type np.ndarray

2.2.3 Optimized Product Quantization (OPQ)

class nanopq.**OPQ**(*M*, *Ks=256*, *verbose=True*)
Pure python implementation of Optimized Product Quantization (OPQ) [Ge14].

OPQ is a simple extension of PQ. The best rotation matrix *R* is prepared using training vectors. Each input vector is rotated via *R*, then quantized into PQ-codes in the same manner as the original PQ.

Parameters

- **M** (*int*) – The number of sub-spaces
- **Ks** (*int*) – The number of codewords for each subspace (typically 256, so that each sub-vector is quantized into 256 bits = 1 byte = uint8)
- **verbose** (*bool*) – Verbose flag

R
Rotation matrix with the shape=(D, D) and dtype=np.float32

Type np.ndarray

M
The number of sub-space

Type int

Ks
The number of codewords for each subspace

Type int

verbose
Verbose flag

Type bool

code_dtype
dtype of PQ-code. Either np.uint{8, 16, 32}

Type object

codewords
shape=(M, Ks, Ds) with dtype=np.float32. codewords[m][ks] means ks-th codeword (Ds-dim) for m-th subspace

Type np.ndarray

Ds
The dim of each sub-vector, i.e., Ds=D/M

Type int

fit (*vecs*, *pq_iter=20*, *rotation_iter=10*, *seed=123*)

Given training vectors, this function alternatively trains (a) codewords and (b) a rotation matrix. The procedure of training codewords is same as `PQ.fit()`. The rotation matrix is computed so as to minimize the quantization error given codewords (Orthogonal Procrustes problem)

This function is a translation from the original MATLAB implementation to that of python http://kaiminghe.com/cvpr13/index.html

If you find the error message is messy, please turn off the verbose flag, then you can see the reduction of error for each iteration clearly

Parameters

- **vecs** – (np.ndarray): Training vectors with shape=(N, D) and dtype=np.float32.
- **pq_iter** (*int*) – The number of iteration for k-means
- **rotation_iter** (*int*) – The number of iteration for leraning rotation
- **seed** (*int*) – The seed for random process

Returns self

Return type object

rotate (*vecs*)

Rotate input vector(s) by the rotation matrix.‘

Parameters **vecs** (*np.ndarray*) – Input vector(s) with dtype=np.float32. The shape can be a single vector (D,) or several vectors (N, D)

Returns Rotated vectors with the same shape and dtype to the input vecs.

Return type np.ndarray

encode (*vecs*)

Rotate input vectors by `OPQ.rotate()`, then encode them via `PQ.encode()`.

Parameters **vecs** (*np.ndarray*) – Input vectors with shape=(N, D) and dtype=np.float32.

Returns PQ codes with shape=(N, M) and dtype=self.code_dtype

Return type np.ndarray

decode (*codes*)

Given PQ-codes, reconstruct original D-dimensional vectors via `PQ.decode()`, and applying an inverse-rotation.

Parameters **codes** (*np.ndarray*) – PQ-cdoes with shape=(N, M) and dtype=self.code_dtype. Each row is a PQ-code

Returns Reconstructed vectors with shape=(N, D) and dtype=np.float32

Return type np.ndarray

dtable (*query*)

Compute a distance table for a query vector. The query is first rotated by `OPQ.rotate()`, then DistanceTable is computed by `PQ.dtable()`.

Parameters **query** (*np.ndarray*) – Input vector with shape=(D,) and dtype=np.float32

Returns Distance table. which contains dtable with shape=(M, Ks) and dtype=np.float32

Return type *nanopq.DistanceTable*

2.2.4 Convert Functions to/from Faiss

`nanopq.`**`nanopq_to_faiss`**(*pq_nanopq*)

Convert a *`nanopq.PQ`* instance to faiss.IndexPQ. To use this function, faiss module needs to be installed.

Parameters **`pq_nanopq`** (`nanopq.PQ`) – An input PQ instance.

Returns A converted PQ instance, with the same codewords to the input.

Return type faiss.IndexPQ

`nanopq.`**`faiss_to_nanopq`**(*pq_faiss*)

Convert a faiss.IndexPQ instance to *`nanopq.PQ`*. To use this function, faiss module needs to be installed.

Parameters **`pq_faiss`** (*`faiss.IndexPQ`*) – An input PQ instance.

Returns

- nanopq.PQ: A converted PQ instance, with the same codewords to the input.
- np.ndarray: Stored PQ codes in the input IndexPQ, with the shape=(N, M). This will be empty if codes are not stored

Return type tuple

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Jegou11] H. Jegou et al., “Product Quantization for Nearest Neighbor Search”, IEEE TPAMI 2011

[Ge14] T. Ge et al., “Optimized Product Quantization”, IEEE TPAMI 2014

Python Module Index

Index