euclidesdb Documentation

Release 0.2.0

Christian S. Perone

Sep 15, 2019

Contents:

1 Changelog **3**
- 1.1 Release v.0.2.0 . . . 3
- 1.2 Release v.0.1.1 . . . 4
- 1.3 Release v.0.1.0 . . . 4

2 Getting Started **5**
- 2.1 Introduction . . . 5
- 2.2 Concepts . . . 6

3 EuclidesDB Installation **9**
- 3.1 Using Docker on any system . . . 9
- 3.2 Installing on Linux . . . 9
- 3.3 Installing on MacOS . . . 10

4 Configuring EuclidesDB **11**
- 4.1 Main Configuration . . . 11
- 4.2 Search Engine Configuration . . . 12
  - 4.2.1 `annoy` Configuration . . . 12
  - 4.2.2 `exact_disk` Configuration . . . 13
  - 4.2.3 `faiss` Configuration . . . 13
- 4.3 Model Configuration . . . 14
- 4.4 How to add a new model . . . 14

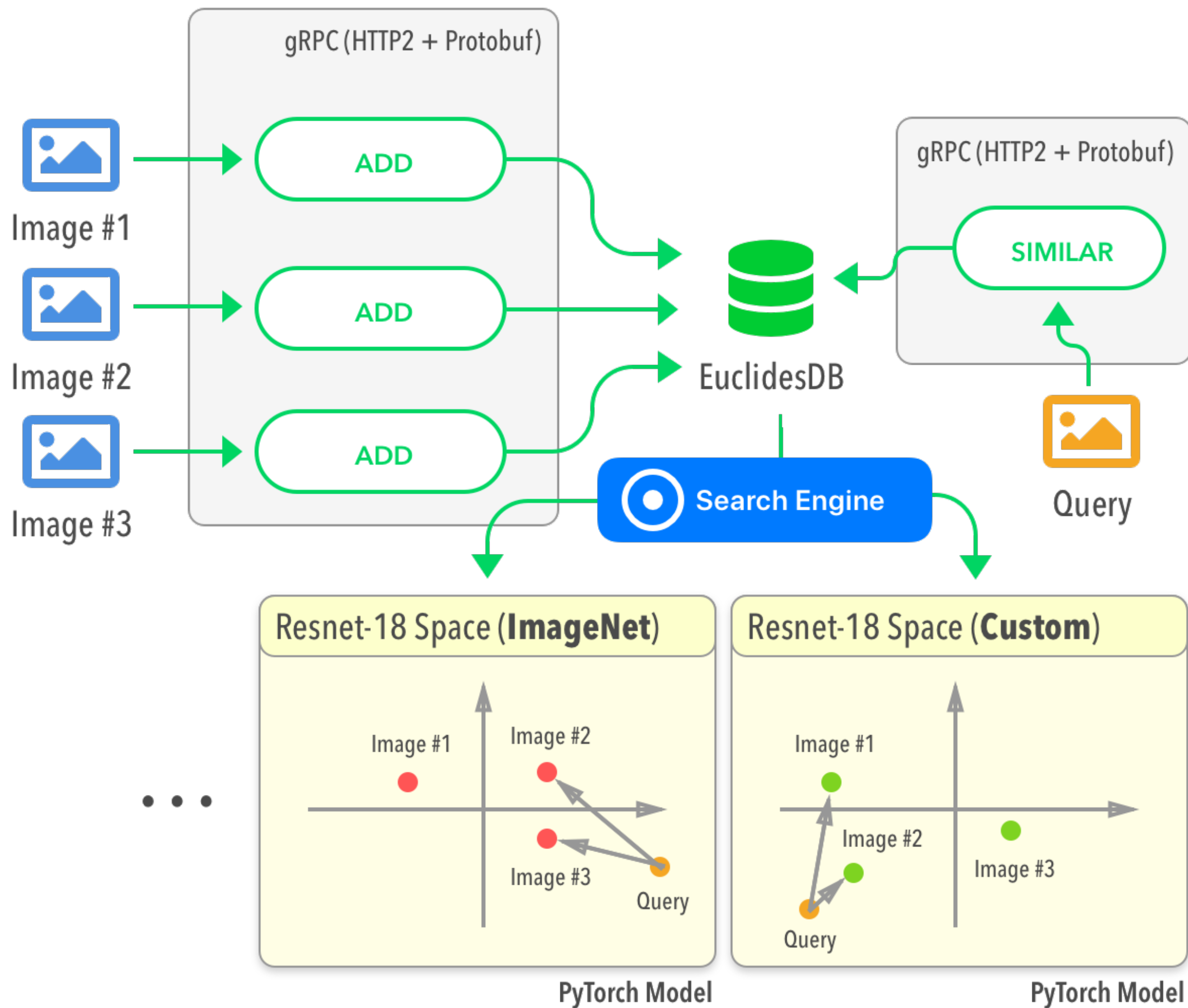
5 Client APIs **17**
- 5.1 Python Client API . . . 17

6 Low-level gRPC API **19**
- 6.1 `AddImage` – add a new image item into the database . . . 19
- 6.2 `RemoveImage` – removes an image item from the database . . . 20
- 6.3 `FindSimilarImageById` – find similar items to an item existing in the database . . . 20
- 6.4 `FindSimilarImage` – find similar items to a new item . . . 21
- 6.5 `Shutdown` – request a shutdown command (shutdown/refresh indexes) . . . 21

7 Contributing to EuclidesDB **23**
- 7.1 Reporting bugs . . . 23
- 7.2 Contributing code and documentation . . . 23
- 7.3 Setting a development environment . . . 23

8 License 25

9 Indices and tables 27



EuclidesDB is a multi-model machine learning feature database that is tight coupled with PyTorch and provides a backend for including and querying data on the model feature space. Some features of EuclidesDB are listed below:

- Written in C++ for performance;
- Uses protobuf for data serialization;
- Uses gRPC for communication;
- LevelDB integration for database serialization;
- Many indexing methods implemented (Annoy, Faiss, etc);
- Tight PyTorch integration through libtorch;
- Easy integration for new custom fine-tuned models;
- Easy client language binding generation;
- Free and open-source with permissive license;

Note: EuclidesDB is still in its **initial release** and many new features are going to come in the next versions. The client API **might change** in the upcoming releases before we stabilize on a robust API design. Contributions are also

welcome ! If you want to contribute, please refer to the *Contributing to EuclidesDB* section.

CHAPTER 1

Changelog

Changelog for the EuclidesDB releases.

1.1 Release v.0.2.0

This is a bug-fix and feature addition release with many good news ! The main new features are: integration with Faiss (see *Search Engine Configuration* for more information), new models, database compression, new exact linear search and internal codebase refactoring.

Thanks for all the users that opened issues and contributors who helped with this release.

Changes in this release:

- **[Enhancement]**: using libtorch 1.0.1 now, latest stable release (#19);
- **[Enhancement]**: examples doesn't require `torchvision` anymore (#8);
- **[Bug]**: wrong model name in client call can cause the server to quit (#1);
- **[Enhancement]**: major refactoring of indexing types, they're now called **Search Engines** and have their own units and configuration;
- **[Bug]**: search engines were called with Variables instead of Tensors;
- **[Enhancement]**: added the new search engine called `exact_dist` that will do a on-disk search (as opposed to in-memory search) using linear exact search (see *Search Engine Configuration* for more information);
- **[Enhancement]**: each search engine has now their own requirement for refresh the index upon adding new items or not;
- **[Enhancement]**: added the new search engine called `faiss` that integrated Faiss/OpenMP/Blas together with EuclidesDB, any Faiss index type is now supported on EuclidesDB (see *Search Engine Configuration* for more information);
- **[Enhancement]**: to avoid memory allocations and improve performance, the reply vectors are now pre-allocated with top-k size;

- **[Enhancement]**: enabled database compression support (snappy);
- **[Enhancement]**: added Resnet101 model support;
- **[Enhancement]**: added internal database versioning machanism to support future underlying changes;
- **[Bug]**: fixed an issue with Python API (missing `close()` channel call);
- **[Enhancement]**: `FindSimilar` RPC call is now called `FindSimilarImage`;
- **[Enhancement]**: added a new RPC call called `FindSimilarImageById` to search similar items based on items already indexed;
- **[Enhancement]**: added documentation for each Search Engine and their configurations (see *Search Engine Configuration* for more information);
- **[Enhancement]**: added documentation for each low-level gRPC call for advanced users (see *Low-level gRPC API* for more information);

1.2 Release v.0.1.1

- Bug-fix release;
- Fixed the issue with models prediction softmax (#2).

1.3 Release v.0.1.0

- Initial release.

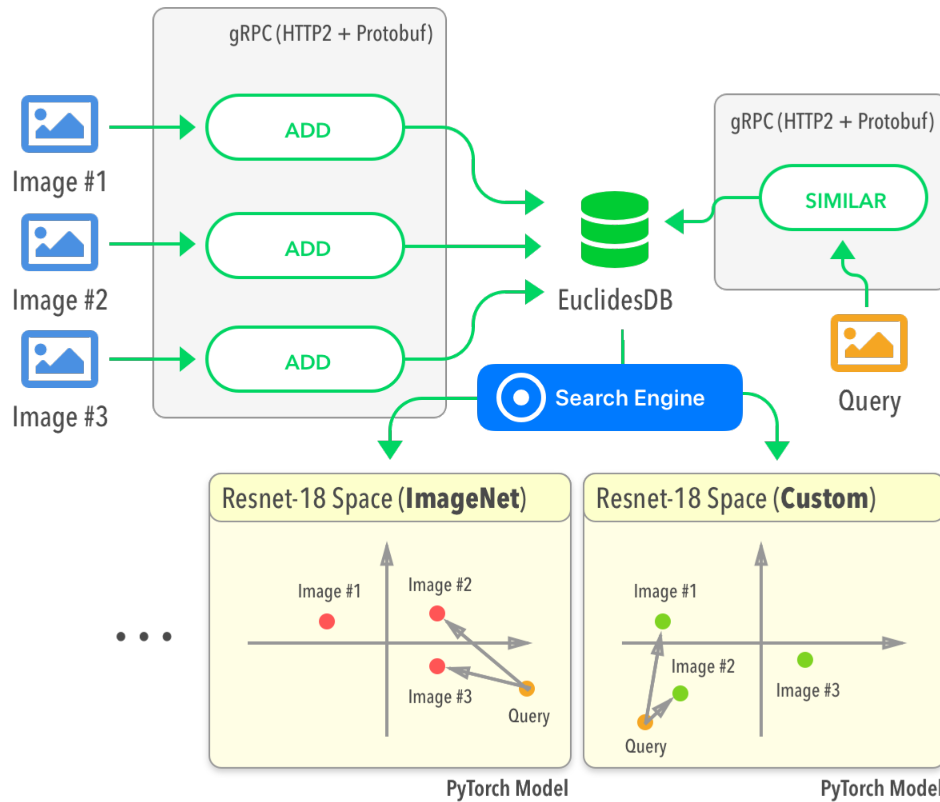
CHAPTER 2

Getting Started

In this getting started section you'll learn more about the concepts behind EuclidesDB and hot to start using it.

2.1 Introduction

To understand EuclidesDB you need to understand the concepts of its underlying architecture below:



Nowadays, many people are still serving machine learning/deep learning models for requests containing binary data using serialization formats and communication protocols such as `JSON+Base64` and `HTTP/1.1`, which isn't appropriate for many reasons (a burden for the wire protocol). Serving machine learning models also poses some unique challenges, and although there are many search engines available for feature search, they're not tight coupled with deep learning frameworks. What happens in practice, is that a lot of different companies end up creating their own systems for model serving, similarity search on the feature space, etc.

A simple use case that might make the EuclidesDB role clear is the case where you want to do similarity search for, let's say, fashion industry and you have for instance multiple models trained for each item category (such as shoes, t-shirts, etc), and you want to use different model spaces to index and query different items.

EuclidesDB tries to solve some issues in this context by providing a very simple standalone server that can store, build indexes and serve requests using efficient serialization (protobuf) and protocols (gRPC+HTTP2) with an easy API that can be consumed in many different languages thanks to gRPC. It offers APIs for including new data into its database and quering it later, it also provised a very tight integration with PyTorch, where the libtorch is used as the backend to run traced models, providing a very easy pipeline to integrate new models (written and trained in Python) into the EuclidesDB C++ backend.

Note: For the moment, only binaries with CPU support are available, GPU support will be implemented soon.

2.2 Concepts

There are some important concepts in EuclidesDB:

- **Module/Model**: we use the concept Module/Model interchangeably because we use PyTorch modules to represent every computation;

- **Model Space**: a model space is the space of features that a model generated and that will be consistent within the same model, given that multiple models are supported, you can add a new image in the database only for some particular models or query only some particular model space;
- **Search Engine**: this is how EuclidesDB index and search for items in the database. EuclidesDB supports a wide range of different indexes that are described in the *Search Engine Configuration* section;

When you add a new image or other kind of data (*we're expanding the support for other kind of items*) into the database, you also specify which model should be used to index this data. Then this data is forwarded into these specified models and their features are saved into a local key-value database to be used later on the construction of a querying index.

The same happens when you query for similar items on a model space, you make a request with a new image and specify on which model spaces you want to find similar items, and the similar items for each model space will be returned together with their relevance.

CHAPTER 3

EuclidesDB Installation

EuclidesDB can be installed both on Linux and MacOS systems. We also provide a Docker image with a single ResNet-18 model already embedded.

See below how to install EuclidesDB in different systems.

3.1 Using Docker on any system

The easiest way to execute EuclidesDB on any system is to use Docker. There is an image already pre-made with ResNet-18 model already embedded, to execute the server, you just need to execute the following line below:

```
docker run -p 50000:50000 \
    -v ~/database:/database \
    -it euclidesdb/euclidesdb
```

This command will host EuclidesDB on the local port 50000 (for RPC calls) and it will store the database data into the host (local) folder `~/database`.

Note: If the database doesn't exists, it will be created by EuclidesDB on the first run.

3.2 Installing on Linux

To install EuclidesDB on Linux systems, you just have to download the last release and then de-compress it and follow the instructions to configure and setup the models:

```
~$ tar zxvf euclidesdb-<version>-Linux.tar.gz
~$ cd euclidesdb
~/euclidesdb$ ./euclidesdb -c euclidesdb.conf
```

EuclidesDB has static linking and ships with all of its external dependencies, so it should work fine on many modern linux distributions without requiring external packages. See how to configure EuclidesDB on the *Configuring EuclidesDB* section.

3.3 Installing on MacOS

To install EuclidesDB in MacOS, the best approach is to install dependencies using homebrew as shown below:

```
brew install grpc
brew install leveldb
brew install libomp
```

After this, please go to the Release Download page in Github and download the latest stable MacOS build, extract the file and you're ready to go. See how to configure EuclidesDB on the *Configuring EuclidesDB* section.

CHAPTER 4

Configuring EuclidesDB

This section provides information on how to configure EuclidesDB, how to add new models and how to execute the server. EuclidesDB has two main kinds of configuration: the configuration for the server and configuration for each model you add on EuclidesDB.

4.1 Main Configuration

The main configuration is responsible for the settings related to the server itself, an example of this configuration can be seen below:

```
[server]
address = 127.0.0.1:50000
log_file_path = /home/user/euclidesdb/logfile.log
search_engine = annoy

[annoy]
tree_factor = 2

[models]
dir_path = /home/user/euclidesdb/models

[database]
db_path = /home/user/euclidesdb/database
```

As you can see, there are three sections in this configuration: server, models and database. The description of each one of these fields are shown below:

- `server.address`: the address server will use to listen, if you with to listen on all interfaces, please use the IP `0.0.0.0` and the port you want to use;
- `server.log_file_path`: this is the path for logging file. Logging is also output to the stdout, but it will also be written in this file;

- `server.search_engine`: this is the search engine that will be used, it can be one of: `annoy`, `faiss` or `exact_disk`. Configuration for each search engine is described later;
- `models.dir_path`: this is the directory path for the models, please refer to the section *Model Configuration* for more information, this path points to a folder where each model is present;
- `database.db_path`: this is the directory path for the database storage. EuclidesDB uses a key-value database based on LevelDB to store all features from each item added into the database;

Note: Remember to always use **absolute paths** in EuclidesDB configuration files.

4.2 Search Engine Configuration

EuclidesDB comes with many different search engines. To choose the search engine, please set the `search_engine` configuration parameter in the `server` section of the configuration file. This parameter will specify which search engine EuclidesDB will use for index/search.

The `search_engine` can assume one the following parameters:

- `annoy`: uses the Annoy indexing/search method;
- `exact_disk`: uses EuclidesDB on-disk (as opposite to in-memory) linear exact search;
- `faiss`: uses the Faiss indexing/search methods;

Each one of these search engines has their pros and cons. For example, `faiss` can provide you a wide spectrum of index methods that offers various trade-offs with respect to search time, search quality, memory, training time, etc. In summary, each search engine will have their own configuration parameters.

4.2.1 `annoy` Configuration

The Annoy search engine configuration accepts only one parameter, called *tree_factor*. This parameter can be specified in the EuclidesDB configuration as seen below (with other configs omited for brevity):

```
[server]
(...)
search_engine = annoy

[annoy]
tree_factor = 2

(...)
```

Description of Annoy parameters:

- `tree_factor`: this number is multiplied by the model space feature size (512 for ResNet8 for example). The default value is 2, which means that if you have a model space with 512 features, the index will use 1024 trees. More trees gives higher precision when querying.

Note: For more information regarding how Annoy works, please see Annoy documentation or the excellent presentation from Erik Bernhardsson.

4.2.2 exact_disk Configuration

The search engine exact_disk is a very simple, but exact search engine. It will iterate over all items in the database (on the disk, hence the name exact_disk) and it will calculate the distance among the query and all items.

A configuration example is shown below (with other configs omited for brevity):

```
[server]
(...)
search_engine = exact_disk

[exact_disk]
pnorm = 2
normalize = false

(...)
```

A descripton of each parameter is shown below:

- pnorm: this is the p-norm used to calculate the distance, the default value is 2 (euclidean distance);
- normalize: when true, it will normalize feature vectors before doing the comparison. If you use a pnorm = 2 and normalize = true, you'll recover cosine similarity.

4.2.3 faiss Configuration

The faiss search engine is perhaps the one that offers the largest amount of indexing types. A configuration example is shown below (with other configs omited for brevity):

```
[server]
(...)
search_engine = faiss

[faiss]
metric = l2
index_type = Flat

(...)
```

The faiss search engine has two parameters: metric and the index_type, however, the index_type is also a way to provide other parameters to build the index according to some patterns.

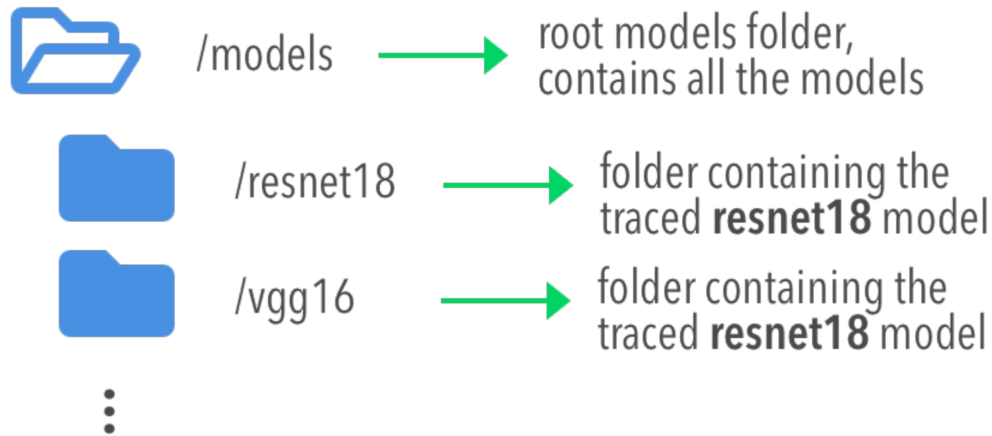
Here is a description of each parameter:

- metric: if equals to l2 (default), it will use the euclidean distance. If this parameter is equal to inner_product it will use the inner-product for the distance;
- index_type: this defines the index index factory string from Faiss. For instance, a Flat value will build an index that uses brute-force L2 distance for search. If this parameter contains the value PCA80,Flat the search engine will produce an index by applying a PCA to reduce it to 80 dimensions and then a exhaustive search.

Note: For more information regarding the Faiss index types and index factory strings, please refer to the Faiss summary of indexes or the Faiss index factory tutorial. If you are unsure about which index to use, please take a look on the Guidelines to choose an index.

4.3 Model Configuration

The models are structured in a folder hierarchy where each sub-folder of the models directory contains a PyTorch traced module file together with a configuration file. This structure can be seen below:



The configuration file for the model must have the name **model.conf** and should follow the formatting below:

```
[model]
name = resnet18
filename = resnet18.pth
prediction_dim = 1000
feature_dim = 512
```

As you can see, this file contains settings related to the model itself. This is the description for each configuration field:

- `model.name`: this is the name of the model that will be used for the EuclidesDB calls when you want to query an index or add a new item for example. A good practice is to use the same name of the folder;
- `model.filename`: this is the serialized traced module filename, it is the output of the PyTorch tracing;
- `model.prediction_dim`: this is prediction dimension of your model. Since EuclidesDB stores the finaly prediction layer as well as model features, you should provide the dimension of the prediction classes. For example, in a model trained on ImageNet, this will be 1000, meaning that there are 1000 prediction classes;
- `model.feature_dim`: this is feature dimension of your model, depending on your model this will have a different size. For the VGG-16 module for instance, this will be 4096, meaning that there is a 4096-dimension vector for the features. As you can note, this should be a flattened vector no matter what model you use;

With these configurations, EuclidesDB is able to use any custom model.

4.4 How to add a new model

Adding a new model into EuclidesDB is straightforward, all you need is to follow the requirements below:

- **Normalization assumption**: we follow a normalization assumption similar to PyTorch torchvision models. EuclidesDB will forward images into your model `forward()` method by scaling each pixel to be between 0 and 1. Then you can normalize the data as you wish on your traced module as we'll show later;
- **Return Tensors**: EuclidesDB stores two vectors from each item (or image), the first is the predictions (class predictions) and the second is the features that you want to store and use to index images to query later. For that reason, within your `forward()` method, you should always return a tuple with **(predictions, features)** and **respecting** the ordering of the elements;

Here is a simple example from EuclidesDB, where it uses the ResNet-18 from torchvision to build a traced module that can be loaded later by EuclidesDB:

```
from torchvision.models import resnet
import torch.utils.model_zoo as model_zoo

import torchvision
import torch

import torch.nn.functional as F

class ResnetModel(resnet.ResNet):
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x_feat = x.view(x.size(0), -1)
        x = self.fc(x_feat)
        predictions = F.softmax(x, dim=0)

        return predictions, x_feat

def resnet18(pretrained=False, **kwargs):
    model = ResnetModel(resnet.BasicBlock, [2, 2, 2, 2], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(resnet.model_urls['resnet18']))
    return model

class Resnet18Module(torch.jit.ScriptModule):
    def __init__(self):
        super(Resnet18Module, self).__init__()
        self.means = torch.nn.Parameter(torch.tensor([0.485, 0.456, 0.406])
                                        .resize_(1, 3, 1, 1))
        self.stds = torch.nn.Parameter(torch.tensor([0.229, 0.224, 0.225])
                                        .resize_(1, 3, 1, 1))
        resnet_model = resnet18(pretrained=True)
        resnet_model.eval()
        self.resnet = torch.jit.trace(resnet_model,
                                      torch.rand(1, 3, 224, 224))

    @torch.jit.script_method
```

(continues on next page)

(continued from previous page)

```
    def helper(self, input):
        return self.resnet((input - self.means) / self.stds)

    @torch.jit.script_method
    def forward(self, input):
        return self.helper(input)

model = Resnet18Module()
model.eval()
traced_net = torch.jit.trace(model,
                             torch.rand(1, 3, 224, 224))
traced_net.save("resnet18.pth")
```

As you can see, this script is doing some stiching to keep the pre-trained weights from the torchvision model, however all you need is a PyTorch module that returns the predictions and features from the `forward()` method and then you just need to call the `torch.jit.trace()` to trace your model and produce the traced module file, which in our case is the `resnet18.pth`.

Note: Remember to set your model to `eval()` mode before tracing it, otherwise you might get inconsistent results due to layers that have different behavior during training and prediction time, such as Dropout and BatchNormalization.

After that, you just need to add this model into a sub-folder inside the models folder and add the configuration file for the model specifying the name of the model and other settings as show in the previous section.

Note: For more help on how to trace PyTorch modules, please refer to PyTorch TorchScript documentation.

CHAPTER 5

Client APIs

This section will show how to use the multiple client APIs that can communicate with EuclidesDB.

5.1 Python Client API

Before using the Python client API, you just have to install it using `pip`:

```
pip install euclides
```

After that, if you want o add a new item into the database, just follow the example below:

```
import euclides

with euclides.Channel("localhost", 50000) as channel:
    db = euclides.EuclidesDB(channel)
    ret_add = db.add_image(image_id, models, image)
```

All images are assumed to be PIL images, the same type handled by `torchvision`. You can see a complete example below, for more examples, see the Python package examples folder.

```
import sys
import argparse

import euclides

from PIL import Image
import numpy as np

from torchvision.transforms import functional as F


def run_main():
    parser = argparse.ArgumentParser(description='Add a new image into database.')
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument('--id', dest='image_id', type=int, required=True,
                        help='ID of the image to add into EuclidesDB.')
    parser.add_argument('--file', dest='filename', type=str, required=True,
                        help='Image file name.')
    args = parser.parse_args()

    image = Image.open(args.filename)
    image_id = int(args.image_id)
    image.thumbnail((300, 300), Image.ANTIALIAS)
    image = F.center_crop(image, 224)

    with euclides.Channel("localhost", 50000) as channel:
        db = euclides.EuclidesDB(channel)
        ret_add = db.add_image(image_id, ["resnet18"], image)

        # After finishing adding items, you need to tell
        # the database to refresh the indexes to add newly
        # indexed items.
        db.refresh_index()

    predictions = ret_add.vectors[0].predictions
    print("Preds Len: ", len(predictions))

    # Category should be 281: 'tabby, tabby cat' for cat.jpg
    # Classes from https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a
    print("Category : ", np.array(predictions).argmax())


if __name__ == "__main__":
    run_main()
```

See also:

See the Python package examples folder for more information.

CHAPTER 6

Low-level gRPC API

This section describes the low-level gRPC API that you can use from any other language or any other server/service proxy.

EuclidesDB implements the following gRPC calls:

```
service Similar {
    rpc Shutdown (ShutdownRequest) returns (ShutdownReply) {}
    rpc FindSimilarImage (FindSimilarImageRequest) returns (FindSimilarImageReply) {}
    rpc FindSimilarImageById (FindSimilarImageByIdRequest) returns (FindSimilarImageReply) {}
    rpc AddImage (AddImageRequest) returns (AddImageReply) {}
    rpc RemoveImage (RemoveImageRequest) returns (RemoveImageReply) {}
}
```

Each one of these RPC calls are described in the next sections. Errors are returned as gRPC errors with a `CANCELED` status.

See also:

See the gRPC documentation for more information. If you're not familiar with `protobuf` syntax, please take a look on these tutorials.

6.1 `AddImage` – add a new image item into the database

The prototype of the `AddImage` call is the following:

```
rpc AddImage (AddImageRequest) returns (AddImageReply) {}
```

This RPC call will accept a `AddImageRequest` request object as input and it will return a `AddImageReply` as result. The definition of these objects are described below:

```
message AddImageRequest {
    int32 image_id = 1;
    bytes image_data = 2;
    bytes image_metadata = 3;
    repeated string models = 4;
}

message AddImageReply {
    repeated ItemVectors vectors = 1;
}
```

These definitions are quite simple and the field names describe the meaning of each field. The `ItemVectors` is described below:

```
message ItemVectors {
    string model = 1;
    repeated float predictions = 2;
    repeated float features = 3;
}
```

Which is the predictions and features for each model space.

6.2 `RemoveImage` – removes an image item from the database

The prototype of the `RemoveImage` call is the following:

```
rpc RemoveImage (RemoveImageRequest) returns (RemoveImageReply) {}
```

This RPC call will accept a `RemoveImageRequest` request object as input and it will return a `RemoveImageReply` as result. The definition of these objects are described below:

```
message RemoveImageRequest {
    int32 image_id = 1;
}

message RemoveImageReply {
    int32 image_id = 1;
}
```

This call will accept a `image_id` as input and it will answer with the same field.

6.3 `FindSimilarImageById` – find similar items to an item existing in the database

The prototype of the `FindSimilarImageById` call is the following:

```
rpc FindSimilarImageById (FindSimilarImageByIdRequest) returns␣
↪(FindSimilarImageReply) {}
```

This RPC call will accept a `FindSimilarImageByIdRequest` request object as input and it will return a `FindSimilarImageReply` as result. The definition of these objects are described below:

```
message FindSimilarImageByIdRequest {
    int32 top_k = 1;
    int32 image_id = 2;
    repeated string models = 3;
}

message FindSimilarImageReply {
    repeated SearchResults results = 1;
}
```

This RPC call will accept a `top_k` that is the number of similar items you want EuclidesDB to return, the item id and the model spaces you want to search. The definition of the `SearchResults` is described below:

```
message SearchResults {
    repeated int32 top_k_ids = 1;
    repeated float distances = 2;
    string model = 3;
}
```

Which is basically the ids of the closest items, their distances and the model where these ids were found.

6.4 `FindSimilarImage` – find similar items to a new item

The prototype of the `FindSimilarImage` call is the following:

```
rpc FindSimilarImage (FindSimilarImageRequest) returns (FindSimilarImageReply) {}
```

This RPC call will accept a `FindSimilarImageRequest` request object as input and it will return a `FindSimilarImageReply` as result. The definition of these objects are described below:

```
message FindSimilarImageRequest {
    int32 top_k = 1;
    bytes image_data = 2;
    repeated string models = 3;
}

message FindSimilarImageReply {
    repeated SearchResults results = 1;
}
```

This RPC call will accept a `top_k` that is the number of similar items you want EuclidesDB to return, the image data and the model spaces you want to search. The definition of the `SearchResults` is the same described in the `FindSimilarImageById` call.

6.5 `Shutdown` – request a shutdown command (shutdown/refresh indexes)

The prototype of the `Shutdown` call is the following:

```
rpc Shutdown (ShutdownRequest) returns (ShutdownReply) {}
```

This RPC call will accept a `ShutdownRequest` request object as input and it will return a `ShutdownReply` as result. The definition of these objects are described below:

```
message ShutdownRequest {
    int32 shutdown_type = 1;
}

message ShutdownReply {
    bool shutdown = 1;
}
```

The `shutdown_type` can be one of the following:

- `0` - a regular database shutdown, it will shutdown EuclidesDB immediately after waiting for all the calls to complete gracefully;
- `1` - a request for EuclidesDB to refresh its indexes. This must be called after adding items into the database (at the end after adding all items). The semantics of this action is that EuclidesDB will gracefully wait for all requests to finish, it will then do a momentary stop while refreshing its memory indexes (this depend on the amount of data in the database and search engine selected) and then it will start to accept requests again. Any call during the refreshing process will not be processed.

This call will return `true` if the request was accepted or `false` otherwise. Currently, there is no `false` return from this call, because the call is always accepted.

CHAPTER 7

Contributing to EuclidesDB

Bug reports and code and documentation patches are welcome. You can help this project also by using the development version of EuclidesDB and by reporting any bugs you might encounter.

7.1 Reporting bugs

To report any bug, please open a new issue on our Github repository.

7.2 Contributing code and documentation

You can also contribute by coding, testing or adding documentation, but before doing it, please consider opening an issue in GitHub to discuss it before implementing to avoid rejected pull-requests.

7.3 Setting a development environment

To set a development environment, you can just clone the repository and use `cmake` to generate makefiles:

```
git clone https://github.com/perone/euclidesdb.git
mkdir build
cd build
cmake ..
make -j2
```

For preparing a release version with optimizations enabled:

```
git clone https://github.com/perone/euclidesdb.git
mkdir build
cd build
```

(continues on next page)

(continued from previous page)

```
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j2
```

To create release package:

```
git clone https://github.com/perone/euclidesdb.git
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j2
make package
```

There is also some Docker files in the repository where we show how to build the binary package from scratch using a self-contained Docker container.

CHAPTER 8

License

```
Copyright 2018 Christian S. Perone

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search