Nemesyst

Release 2.0.6.r6.68bebbc

GeorgeRaven

Mar 24, 2020

TABLE OF CONTENTS

CHAPTER

ONE

MIT LICENSE

CHAPTER

TWO

WHY USE NEMESYST

Nemesyst is a highly configurable hybrid parallelization deep learning framework, for distributed deep learning, that uses other backend framework(s) of your choice (Pytorch, TensorFlow, etc.) for training.

Fig. 1: This image is a use case example of Nemesyst applied to a distributed refrigeration fleet over multiple sites, and both online and offline learning capabilities occuring simultaneously.

Nemesyst uses MongoDB as its core message passing interface (MPI). This means MongoDB is used to store, distribute, retrieve, and transform the data; store, distribute, and retrieve the trained models. In future we also hope to use it to transfer more specific processing instructions to individual learners. This way we use the already advanced functionality of MongoDB to handle complex and non-trivial problems such as tracing models back to the specific data trained with, the results and arguments present at the point of training, and being able to reload pre-trained models for further use, and, or training. This also means the same data can be transformed differently for different learners from the same source dynamically at the point of need.

CHAPTER THREE

INSTALLATION

Note: Certain distributions link `python` to `python2` and others link it to `python3`. For disambiguation python, pip, and virtualenv shall mean their python v3 versions here, i.e. `python3`, `pip3`, `virtualenv3`.

Warning: You will need to have git, and python installed for any of the below methods to work. You will also need MongoDB if you intend to create a local database, (more than likely), but Nemesyst will still connect to already running databases without it if you happen to have one already.

This section will outline various methods for installation of Nemesyst, and its dependencies. Not all methods are equal there are slight variations between them, which are outlined in the respective sections below, along with instructions for each method:

3.1 Files-only/ development

This method of files-only installation provides the user with all the additional utility files, and examples needed during development. This includes the files necessary for the *Full MNIST Example*, and is advised when first starting to use Nemesyst so that you can better understand what is going on. In production however you do not need all these additional files so other slimmer/ more streamlined methods of installation are better.

Pros:

- All the example files for quickly getting to grips with Nemesyst.
- Easy to understand as the files are not filed away somewhere obscure.
- Easy to install example dependencies as you can `pip install -r requirements.txt` or whatever other requirements list we include.
- Unit tests available.

Cons:

- You are responsible for ensuring the requirements are met for Nemesyst, such as python, git, and MongoDB.
- It is less repeatable/ deployable as most steps are manual as opposed to the other available methods of installation.

3.1.1 Getting the files

To retrieve the Nemesyst files you will need git installed. To download the Nemesyst directory in your current working directory you can run:

```
git clone https://github.com/DreamingRaven/nemesyst
```

3.1.2 Installing dependancies

To make use of Nemesyst directly now that you have the files you need to have installed:

System dependencies:

1. python (required): Nemesyst is written in python, you wont get far without it.
2. git (required): To install, and manage Nemesyst files.
3. MongoDB (recommended): If you want to be able to create, and destroy a local MongoDB database.
4. Docker (optional): If you want to manage local containerized MongoDB databases.

Python dependencies:

./nemesyst/requirements.txt

```
ConfigArgParse>=0.14.0
pymongo>=3.8.0
future>=0.17.1
```

You can install these quickly using:

Bash shell installing dependancies from file

```
pip install -r ./nemesyst/requirements.txt
```

or:

Bash shell installing Nemesyst and dependancies using setup.py

```
python setup.py install
```

Optionally if you would like to build the Nemesyst documentation, and/ or use the full testing suite you will require `./nemesyst/docs/requirements.txt`:

```
sphinx>=2.1.2
sphinx-argparse>=0.2.5
sphinx-rtd-theme>=0.4.3
ConfigArgParse>=0.14.0
pymongo>=3.8.0
future>=0.17.1
```

3.2 Automated

This section discusses the more automated and repeatable installation methods for Nemesyst, but they do not contain all the files needed to learn, and begin developing Nemesyst integrated applications, rather this includes just the bare-bones Nemesyst ready for your deployment.

3.2.1 Generic

pip

For now you can use pip via:

```
pip install git+https://github.com/DreamingRaven/nemesyst.git#branch=master
```

Docker

see *Dockerisation* for docker instructions.

3.2.2 Archlinux

Install nemesyst-git^AUR.

3.3 Manual

3.3.1 Generic

```
git clone https://github.com/DreamingRaven/nemesyst
cd nemesyst
python setup.py install
```

3.3.2 Archlinux

```
git clone https://github.com/DreamingRaven/nemesyst
cd nemesyst/.arch/
makepkg -si
```

3.4 Virtual env

To create the python-virtualenv:

```
vituralenv venv
```

To then use the newly created virtual environment:

```
source venv/bin/activate
```

OR if you are using a terminal like fish:

```
source venv/bin/activate.fish
```

To install Nemesyst and all its dependencies into a virtual environment while it is being used (activated):

```
pip install git+https://github.com/DreamingRaven/nemesyst.git#branch=master
```

To exit the virtual environment:

```
deactivate
```

CHAPTER

FOUR

OVERVIEW

Note: Throughout this overview and in certain other sections the examples provided are for *Files-only/ development* installations, however this is only to make it easier to use the inbuilt examples/ sample files rather than having to force the user to define his/ her own cleaning, learning, infering scripts, for the sake of simplicity.

If you are not using the *Files-only/ development* installation you will have to point nemesyst to cleaners, learners, predictors etc that you want to use. Although even if you are using *Files-only/ development*, eventually once you have better understood and tested Nemesyst then you should likeley move to creating your own ones that you require, and using a normal installation of Nemesyst such as one of the *Automated* examples.

4.1 Nemesyst literal un-abstract stages

Fig. 1: This image is a use case example of Nemesyst applied to a distributed refrigeration fleet over multiple sites, and both online and offline learning capabilities occuring simultaneously.

Nemesyst has been made to be generic enough to handle many possible configurations, but we cannot possibly handle all possible scenarios. Sometimes it may be necessary to manually configure certain aspects of the process, especially regarding MongoDB as it is quite a well developed, mature, database, with more features than we could, and should automate.

4.2 Nemesyst Abstraction of stages

Fig. 2: Nemesyst has abstracted, grouped, and formalised what we believe are the core stages of applying deep learning at all scales.

Deep learning can be said to include 3 stages, data-wrangling, test-training, and inferring. Nemesyst adds an extra layer we call serving, which is the stage at which databases are involved as the message passing interface (MPI), and generator, between the layers, machines, and algorithms, along with being the data, and model storage mechanism.

4.3 Nemesyst Parallelisation

As of: 2.0.1.r6.f9f92c3

Fig. 3: Nemesyst parallelises each script, up the the maximum number of processes in the process pool.

Local parallelization of your scripts occur using pythons process pools from multiprocessing. This diagram shows how the rounds of processing are abstracted and the order of them. Rounds do not continue between stages, I.E if there is a spare process but not enough scripts from that stage (e.g cleaning) it will not fill this with a script process from the next stage (e.g learning). This is to prevent the scenario where a learning script may depend on the output of a previous cleaning script.

4.4 Wrangling / cleaning

See *All Options by Category* for a full list of options.

Fig. 4: Wrangling is the stage where the data is cleaned into single atomic examples to be imported to the database.

Files-only/ development **example:**

```
nemesyst
```

4.5 Serving

See *All Options by Category* for a full list of options.

Fig. 5: Serving is the stage where the data and eventually trained models will be stored and passed to other processess potentially on other machines.

Nemesyst uses MongoDB databases through PyMongo as a data store, and distribution mechanism. The database(s) are some of the most important aspects of the chain of processes, as nothing can operate without a properly functioning database. As such we have attempted to simplify operations on both the user scripts side and our side by abstracting the slightly raw PyMongo interface into a much friendlier class of operations called *Mongo*.

A *Mongo* object is automatically passed into every one of your desired scripts entry points, so that you can also easily operate on the database if you so choose although aside from our data generator we handle the majority of use cases before it reaches your scripts.

Automated **example:**

```
# creating basic non-config, non-replica, localhost, mongodb instance
nemesyst --db-init --db-start --db-login --db-stop \
         --db-user-name USERNAME --db-password \
         --db-path DBPATH --db-log-path DBPATH/LOGDIR
```

Note: Please see *Serving with MongoDB* for more in depth serving with Nemesyst

4.6 Learning

See *All Options by Category* for a full list of options.

Fig. 6: Learning is the stage where the data is used to train new models or to update an existing model already in the database.

***Files-only/ development* example:**

```
nemesyst
```

> **Warning:** Special attention should be paid to the size of the resultant neural networks. Beyond a certain size it will be necessary to store them as GridFS objects. The basic GridFS functionality is included in nemesyst's *Mongo* however this is still experimental and should not be depended upon at this time.

4.7 Inferring / predicting

As of: 2.0.2.r7.1cf3eab

See *All Options by Category* for a full list of options.

Fig. 7: Inferring is the stage where the model(s) are used to predict on newly provided data.

***Files-only/ development* example:**

```
nemesyst
```

CHAPTER

FIVE

FULL MNIST EXAMPLE

MNIST is a popular well known dataset for evaluating machine learning models. It has been effectively solved at this point, but it is still a good starting point for getting to know how Nemesyst works, and to be able to show people how to use Nemesyst in practice. It is also relatively clean so there is little pre-processing that is required other than turning it into a directly usable form.

The dataset will be downloaded for you by the cleaning module.

5.1 Requirements

Please ensure you have both MongoDB and the following python dependencies installed as a bare minimum:

examples/requirements/mnist.txt

```
ConfigArgParse>=0.14.0
pymongo>=3.8.0
future>=0.17.1
scikit-learn>=0.21.3
keras>=2.3.1
tensorflow-gpu>=2.0.0
```

If you are using pip you can quickly install these using:

Files-only/ development **pip requirements installation example:**

```
pip install -r examples/requirements/mnist.txt
```

Note: Please also ensure you have the Nemesyst files at hand (*Files-only/ development*) as they have all the extra files you will need later on, which are only present in *Files-only/ development*

5.2 Configuring

For this example we have created a configuration file for you so there is nothing additional that needs to be done. It is advised that you read it through. It is a *.ini* style file. However each of these options can be passed in to Nemesyst as cli or environment options as well but we believed it would be a much nicer introduction to have them in a configuration file.

examples/configs/nemesyst/mnist.conf

```
# please see full documentation at:
#
# this config file assumes you are in the directory nemesyst from:
# https://github.com/DreamingRaven/nemesyst
# we use relative paths here so they may not work if you arent there.

# mongodb options for your experimental database
--db-user-name=groot           # change this to you desired username
--db-password=True             # this will create a password prompt
; --db-init=True                 # initialises the database with user
; --db-start=True                # starts the database
--db-port=65530                # sets the db port
--db-name=data                 # sets the database name
--db-path=./data_db/           # sets the path to create a db
--db-log-path=./data_db/       # sets the parent directory of log files
--db-log-name=mongo_log        # sets the file name to use for log
--db-authentication=SCRAM-SHA-1 # sets db to be connected to using user/
↪pass

# cleaning specific options
; --data-clean=True                                                #␣
↪nothing will be cleaned unless you tell nemesyst to even if you give␣
↪it the other information
--data-cleaner=examples/cleaners/mnist_cleaner.py                 # the path␣
↪to the cleaner in this case MNIST example cleaner
--data-collection=mnist                                           # sets the␣
↪collection to import to

# learning specific options
; --dl-learn=True                                                  #␣
↪nothing will be learned unless you tell nemesyst explicitly to do so␣
↪even if other information is given
--dl-learner=examples/learners/mnist_learner.py                   # the path␣
↪to the learner in this case MNIST example learner
--dl-batch-size=32                                                # set the␣
↪batch sizes to use
--dl-epochs=12                                                    # set the␣
↪number of epochs we want (times to train on the same data)
--dl-output-model-collection=models

# infering specific options
; --i-predict=True                                                 #␣
↪nothing will be predicted unless you tell nemesyst explicitly to do so␣
↪even if other information is given
--i-predictor=examples/predictors/mnist_predictor.py              # the path␣
↪to the predictor in this case MNIST example predictor
```

If you would like to the skip rest of this example for whatever reason such as you are more interested in checking Nemesyst is working simply remove the symbol ";" from the start of any lines it appears in to uncomment that line, and then run everything using:

Files-only/ development **automated example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf
```

5.3 Serving

For this example Nemesyst will create a database for us whenever we call the config file since we pass in options to initialize and start the database (see *Configuring*). We can do this using:

***Files-only/ development* serving example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf --db-init --
↪db-start
```

This example will start the database, to close the database you can:

***Files-only/ development* stopping database example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf --db-stop
```

Note: Nemesyst may ask you a password. As long as you are using the same password between runs it wont cause you issue as you are simultaneously using and creating (when using –db-init) the password for the default user in our config file, you can change this behavior but we wanted to include it so we don't end up creating universal passwords that lazy users might oversee.

For more complex scenarios pleas refer to *Serving with MongoDB*

5.4 Checking up on the database

It may be necessary after each of the following steps to check on the database to ensure it has done exactly what you expect it to be doing. To login to the database easily you can use:

***Files-only/ development* logging into running database example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf --db-login
```

This should put you in the Mongo shell which is a javascript based interface of MongoDB for direct user intervention. Where you can do all sorts of operations and checks. This is of course optional but recommended. If you would rather a more graphical interface you can use any of the plethora of tools to visualize the database but we recommend MongoDB Compass, in particular for its aggregation helper.

5.5 Cleaning

In this step we will launch the example MNIST cleaner which downloads the data using scikit-learn to get a much cleaner version of the data set for us. Then inserting the data into individual dictionaries row wise, so that each dictionary is a single complete example/ observation, with associated target feature. To put it back into the database we need only yield each dictionary and Nemesyst will handle iteration for us. This document dictionary can also be used to house useful metadata about the dataset so that you can further filter using more advanced Nemesyst and MongoDB functionality that go beyond the scope of this simple introduction.

To begin cleaning you need only tell Nemesyst to clean the data using:

***Files-only/ development* cleaning example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf --data-clean
```

The example MNIST cleaner is shown below for convenience.

examples/cleaners/mnist_cleaner.py

```
# @Author: George Onoufriou <archer>
# @Date:   2019-08-15
# @Email:  george raven community at pm dot me
# @Filename: debug_cleaner.py
# @Last modified by:   archer
# @Last modified time: 2019-08-16
# @License: Please see LICENSE in project root

import io
import datetime
from sklearn.datasets import fetch_openml


def main(**kwargs):
    print("downloading mnist dataset...")
    x, y = fetch_openml('mnist_784', version=1, return_X_y=True)
    utc_import_start_time = datetime.datetime.utcnow()
    print("importing mnist dataset to mongodb...")
    for i in range(len(x)):  # could use enumerate but only interested␣
↪in index
        document = {
            "x": x[i].tolist(),     # converting to list to be bson␣
↪compatible
            "y": int(y[i]),         # Ensuring is num
            "img_num": i,           # saving the image number
            "utc_import_time":  utc_import_start_time,
            "dataset": "mnist",
            "img_count": len(x)
        }
        yield document
```

5.6 Learning

To learn from the now cleaned database-residing data, you can:

***Files-only/ development* learning example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf --dl-learn
```

This example trains a CNN, and yields a tuple `(metadata_dictionary, pickle.dumps(model))` which is then stored in MongoDB using gridfs as most models exceed the base MongoDB 16MB document size limit. This example is derived from one of the pre-existing Keras MNIST examples, but transformed into a relatively efficient Nemesyst variant. The major differences are that we use *fit_generator* which takes a generator (in our case a database cursor and pre-processor) for the training set, and another generator for the validation set. For this example we have simply validated against the test set as we aren't attempting to blind ourselves for the purposes of scientific rigor and over-fitting prevention. Care should be taken in reading the pipelines as they can be quite complex operations to solve very tough problems, but here we simply set them to separate the dataset into train, and validation.

examples/learners/mnist_learner.py

```
# @Author: George Onoufriou <archer>
# @Date:   2019-08-16
```

(continues on next page)

(continued from previous page)

```
# @Email:  george raven community at pm dot me
# @Filename: mnist_learner.py
# @Last modified by:   archer
# @Last modified time: 2020-01-31T16:13:08+00:00
# @License: Please see LICENSE in project root

import numpy as np
import pickle

import keras
from keras import backend as K
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D


def main(**kwargs):
    """Entry point called by Nemesyst, always yields dictionary or None.

    :param **kwargs: Generic input method to handle infinite dict-args.
    :rtype: yield dict
    """
    # # there are issues using RTX cards with tensorflow:
    # # https://github.com/tensorflow/tensorflow/issues/24496
    # # if this is the case please uncomment the following two lines:
    # import os
    # os.environ['CUDA_VISIBLE_DEVICES'] = '-1'  # use cpu

    # just making these a little nicer to read but in a real application
    # we would not want these hardcoded thankfully the database can␣
↪provide!
    args = kwargs["args"]
    db = kwargs["db"]
    img_rows, img_cols = 28, 28
    num_classes = 10
    # creating two database generators to iterate quickly through the␣
↪data
    # these are not random they will split data using 60000 as the␣
↪boundary
    train_generator = inf_mnist_generator(db=db, args=args,
                                          example_dim=(img_rows, img_
↪cols),
                                          num_classes=num_classes,
                                          pipeline=[{"$match":
                                                     {"img_num":
                                                      {"$lt": 60000}}}
                                                    ])
    test_generator = inf_mnist_generator(db=db, args=args,
                                         example_dim=(img_rows, img_
↪cols),
                                         num_classes=num_classes,
                                         pipeline=[{"$match":
                                                    {"img_num":
                                                     {"$gte": 60000}}}
                                                   ])
    # ensuring our input shape is in whatever style keras backend wants
    if K.image_data_format() == 'channels_first':
```

(continues on next page)

(continued from previous page)

```
        input_shape = (1, img_rows, img_cols)
    else:
        input_shape = (img_rows, img_cols, 1)

    model = generate_model(input_shape=input_shape,
                           num_classes=num_classes)
    model.summary()
    hist = model.fit_generator(generator=train_generator,
                               steps_per_epoch=219,   # ceil(70000/32)
                               validation_data=test_generator,
                               validation_steps=219,
                               epochs=args["dl_epochs"][args["process"]],
                               initial_epoch=0)

    excluded_keys = ["pylog", "db_password"]
    # yield metadata, model for gridfs
    best_model = ({
        # metdata dictionary (used to find model later)
        "model": "mnist_example",
        # "validation_loss": float(hist.history["val_loss"][-1]),
        # "validation_accuracy": float(hist.history["val_acc"][-1]),
        "loss": float(hist.history["loss"][-1]),
        "accuracy": float(hist.history["accuracy"][-1]),
        "args": {k: args[k] for k in set(list(args.keys())) - \
                 set(excluded_keys)},
    }, pickle.dumps(model))

    yield best_model


def generate_model(input_shape, num_classes):
    """Generate the keras CNN"""
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation="relu",
                     input_shape=input_shape))
    model.add(Conv2D(64, (3, 3), activation="relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(128, activation="relu"))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation="softmax"))

    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adadelta(),
                  metrics=['accuracy'])
    return model


def inf_mnist_generator(db, args, example_dim, num_classes,␣
↪pipeline=None):
    """Infinite generator of data for keras fit_generator.

    :param db: Mongo() object to use to fetch data.
    :param args: The user provided args and defaults for adaptation.
    :param example_dim: The tuple dimensions of a single example (row,␣
↪col).
```

(continues on next page)

(continued from previous page)

```
    :param pipeline: The MongoDB aggregate pipeline [{},{},{}] to use.
    :type db: Mongo
    :type args: dict
    :type example_dim: tuple
    :type num_classes: int
    :type pipeline: list(dict())
    :return: Tuple of a single data batch (x_batch,y_batch).
    :rtype: tuple
    """
    # empty pipeline if none provided
    pipeline = pipeline if pipeline is not None else [{"$match": {}}]
    # loop infiniteley over pipeline
    while True:
        c = db.getCursor(db_collection_name=str(args["data_collection"]
                                                [args["process"]]),
                         db_pipeline=pipeline)
        # itetate through the data in batches to minimise requests
        for data_batch in db.getBatches(db_batch_size=args["dl_batch_size
↪"]
                                        [args["process"]], db_data_
↪cursor=c):
            # we recommend you take a quick read of:
            # https://book.pythontips.com/en/latest/map_filter.html
            y = list(map(lambda d: d["y"], data_batch))
            y = np.array(y)  # converting list to numpy ndarray

            x = list(map(lambda d: d["x"], data_batch))
            x = np.array(x)  # converting nlists to ndarray

            # shaping the np array into whatever keras is asking for
            if K.image_data_format() == 'channels_first':
                y = y.reshape((y.shape[0], 1))
                x = x.reshape((x.shape[0], 1,
                               example_dim[0], example_dim[1]))
                # input_shape = (1, example_dim[0], example_dim[1])
            else:
                y = y.reshape((y.shape[0], 1))
                x = x.reshape((x.shape[0],
                               example_dim[0], example_dim[1], 1))
                # input_shape = (example_dim[0], example_dim[1], 1)

            # normalising to 0-1
            x = x.astype('float32')
            x /= 255

            # convert class vectors to binary class matrices
            y = keras.utils.to_categorical(y, num_classes)

            # returning completeley propper data, batch by batch thats␣
↪all.
            yield x, y
```

5.7 Inferring

Warning: Work in progress section

In this stage we retrieve the model trained previously stored in MongoDB as gridfs chunks and unpack the model again for reuse and prediction. We can predict using the gridfs stored model by passing:

***Files-only/ development* inferring example:**

```
./nemesyst --config ./examples/configs/nemesyst/mnist.conf --i-predict
```

As in the previous sections, this lets nemesyst know to run the predictor specified in the config file, which can be seen below. This predictor loads the most recent, most performant mnist model, and uses it to predict against the testing set.

examples/predictors/mnist_predictor.py

```
# @Author: George Onoufriou <archer>
# @Date:   2019-08-16
# @Email:  george raven community at pm dot me
# @Filename: debug_predictors.py
# @Last modified by:   archer
# @Last modified time: 2019-08-16
# @License: Please see LICENSE in project root


def main(**kwargs):
    """Entry point called by Nemesyst, always yields dictionary, tuple␣
↪or None.

    :param **kwargs: Generic input method to handle infinite dict-args.
    :rtype: yield dict
    """
    args = kwargs["args"]
    db = kwargs["db"]

    db.connect()

    # define a pipeline to get the latest gridfs file in any collection
    fs_pipeline = [{'$sort': {'uploadDate': -1}},   # sort most recent␣
↪first
                   {'$limit': 1},  # we only want one model
                   {'$project': {'_id': 1}}]  # we only want its _id
    args["dl_output_model_collection"]
    # we add a suffix to target the metadata collection specifically
    # at the end of the top level model collection name we specified in␣
↪our
    # config file
    model_coll_root = args["dl_output_model_collection"][args["process"]]
    model_coll_files = "{0}{1}".format(model_coll_root, ".files")
    # apply this pipeline to the collection we used to store the models
    fc = db.getCursor(db_collection_name=model_coll_files,
                      db_pipeline=fs_pipeline)
    # we could return several models but we have limited everything to␣
↪only one
```

(continues on next page)

(continued from previous page)

```
    # but to be extensible this shows how to get the models from the db
    # in batches, however since we only have one model a batch size␣
↪higher than
    # one does nothing
    for batch in db.getFiles(db_batch_size=1, db_data_cursor=fc,
                             db_collection_name=model_coll_root):
        for doc in batch:
            # now read the gridout object to get the model (pickled)
            model = doc["gridout"].read()
            print(doc, type(model))

    yield None
```

CHAPTER

SIX

SERVING WITH MONGODB

Nemesyst uses MongoDB as its primary message passing interface. This page will more elaborate on using Nemesyst with different database setups, debugging, common issues, and any nitty-gritty details that may be necessary to discuss.

Warning: While Nemesyst does support using mongodb.yaml files for complex db setup, care should be taken that Nemesyst is not overriding the values you were expecting in the config files. Things such as the DBs path are almost always overridden along with the port to use by default even if the user has not provided that argument. In future we intend to make it such that hard coded defaults when not overridden by the user, first attempt to look in the mongodb.yaml file before falling back to hard-coded values.

6.1 Creating a basic database

Disambiguation: we define a basic database as a standalone MongoDB instance with one universal administrator and one read/write user with password authentication.

While it is possible it is highly discouraged to use Nemesyst to create the users you require as this is quite complicated to manage and may lead to more problems than its worth compared to simply creating a database and adding a user manually using something like the following:

6.1.1 Manual creation of MongoDB

Files-only/ development **creation of database example:**

```
mongod --config ./examples/configs/basic_mongo_config.yaml
```

This will create a database with all the MongoDB defaults as it is an empty yaml file. If you would instead want a more complex setup please take a look at `examples/configs/authenticated_replicaset.yaml` instead, but you will need to generate certificates and keys for this so it is probably a poor place to start but will be what you will want to use in production as a bare minimum security.

6.1.2 Docker-Compose creation of MongoDB

Docker-Compose, *Files-only/ development* creation of database example:

```
docker-compose up
```

This similar to the *Manual creation of MongoDB* creation uses a simple config file to launch the database. This can be changed in `docker-compose.yaml`. At this point you will need to connect to the running MongoDB instance (see: *Connecting to a running database*) to create your main administrator user, with "userAdminAnyDatabase" role. After this you can use the following to close the Docker container with the database:

Docker-Compose, *Files-only/ development*, closing Docker-Compose database example:

```
docker-compose down
```

Note: Don't worry we set our docker-compose.yaml to save its files in `/data/db` so they are persistent between runs of docker-compose. If you need to delete the MongoDB database that is where you can find them.

6.1.3 Connecting to a running database

To be able to fine tune, create users, update etc it will be necessary to connect to MongoDB in one form or another. Nemesyst can help you log in or you can do it manually.

Note: If there is no userAdmin or userAdminAnyDatabase then unless expressly configured there will be a localhost exception which will allow you to log in and create this user. If this user exists the localhost exception will close. Please ensure you configure this user as they can grant any role or rights to anyone and would be a major security concern along with making it very difficult to admin your database.

Nemesyst

Nemesyst can be used to log you in to the mongo shell although this feature should not be depended on, and instead it is recommended to use mongo for anything more complicated than simple testing. You will need to provide any other options like ip port etc if it is not using the defaults.

Bash shell simple all defaults example:

```
nemesyst --db-login
```

Mongo

To connect to an non-sharded database with autnentication but no TLS/SSL:

Bash shell example:

```
mongo HOSTNAME:PORT -u USERNAME --authenticationDatabase DATABASENAME
```

To connect to a slightly more complicated scenario with authentication, TLS, and sharding enabled:

Bash shell example:

```
mongo HOSTNAME:PORT -u USERNAME --authenticationDatabase DATABASENAME --
↪tls --tlsCAFile PATHTOCAFILE --tlsCertificateKeyFile PATHTOCERTKEYFILE
```

6.1.4 Creating database users

You will absolutely need a user with at least "userAdminAnyDatabase" role. Connect to the running database see *Connecting to a running database*.

Mongo shell create a new role-less user:

```
db.createUser({user: "USERNAME", pwd: passwordPrompt(), roles: []})
```

Mongo shell grant role to existing user example:

```
db.grantRolesToUser(
"USERNAME",
[
  { role: "userAdminAnyDatabase", db: "admin" }
])
```

Mongo shell create user and grant userAdminAnyDatabase in one:

```
db.createUser({user: "USERNAME", pwd: passwordPrompt(), roles: [{role:
↪"userAdminAnyDatabase", db: "admin"}]})
```

Note: Since this user belongs to admin in the previous examples that means the authenticationDatabase is admin when authenticating as this user as per the instructions in "*Connecting to a running database*".

6.2 From basic database to replica sets

todo Include instructions for turning a database into several replica sets.

6.3 Troubleshooting

Please see *MongoDB/ Serving Issues*

6.4 Further reading

MongoDB config file options

CHAPTER

SEVEN

DOCKERISATION

Docker is a lightweight semi-vm that can help automate reproducibility, dependency management, deployment, and use of some code which is containerized. Considering the relative ease with which Docker is used, modified/adjusted, with only a minimal amount of code, it has quite a profound affect on work-flows, making really nightmarish scenarios much easier to handle.

For docker installation you may need to look up instructions online but after installing docker (minimum version 19.03) and nvidia container toolkit, you will need not install anything further, and can instead rely on the Dockerfile and docker to install and manage dependencies from then on. If you would like more automation/ to use docker-compose then please ensure you also have docker-compose installed.

There are two available versions of our Dockerfile:

- Archlinux based nemesyst docker `examples/containers/nemesyst/Dockerfile`; This docker is the one we seek to support since it will force us to stay up to date with the latest software and changes so we never end up in a crippling dependency requirement. It should be noted however that it is not quite complete.
- Ubuntu based nemesyst docker `examples/containers/nemesyst_ubuntu/Dockerfile`; This docker is the one we make available for the purposes of longer term support, and for those that just prefer Ubuntu (you must be crazy!). This one is the more supported by depended on projects such as tf-seal so is easier to maintain.

7.1 Docker Usage (Linux)

While docker is very portable to most platforms, we do not maintain any non-x86_64, Microsoft Windows or Mac systems, thus we cannot presume to give sound Docker usage on these other platforms. However the usage should largely remain the same, but presumably without the need for privilege escalation using sudo for Win and Mac.

Using docker usually revolves around only two steps building the image you would like to use, and then using it either interactively or by issuing explicit commands to be executed. First however we should briefly mention the two most important files related to this a .dockerignore file, and a Dockerfile .

7.1.1 Dockerfile

A Dockerfile is a short command based script that defines how to create a container. These can and usually are built on other containers. Please refer to the Dockerfile documentation for a more in depth breakdown.

Dockerfile example `examples/containers/nemesyst_ubuntu/Dockerfile`

```
FROM ubuntu:19.04

# updating and installing basic ubuntu python container
```

(continues on next page)

(continued from previous page)

```
RUN apt update && \
    apt install -y wget python3.7 python3-pip git

# getting and installing tensorflow, and tf-seal
RUN wget https://storage.googleapis.com/tf-pips/tf-c++17-support/tf_
↪nightly-1.14.0-cp37-cp37m-linux_x86_64.whl && \
    python3.7 -m pip install tf_nightly-1.14.0-cp37-cp37m-linux_x86_64.
↪whl && \
    rm tf_nightly-1.14.0-cp37-cp37m-linux_x86_64.whl && \
    python3.7 -m pip install tf-seal

# getting tf-seal repository so we have access to all of their examples␣
↪etc
RUN python3.7 -m pip install git+https://github.com/DreamingRaven/
↪nemesyst.git#branch=master && \
    git clone https://github.com/tf-encrypted/tf-seal && \
    git clone https://github.com/DreamingRaven/nemesyst
```

7.1.2 .dockerignore

A .dockerignore is similar in function to a .gitignore and supports similar syntax. Special care should be paid to .dockerignore files as they are both useful to minimise the risk of potential secrets being leaked into a container, their container size etc, but they can also cause problems with things like the `COPY` command leading to unexpected results. We personally recommend a whitelist strategy .dockerignore where you specify only what you would like to be copied in.

whitelist .dockerignore example `examples/containers/nemesyst_ubuntu/.dockerignore`

```
# ignore everything using whitelist strategy
*

# you can selectiveley allow files using the ! at the beginning of the␣
↪line
#!lib/**/*.py # this would allow all python files in subdirectories of␣
↪lib if enabled
```

7.1.3 Building

With a Dockerfile in the current directory to build a dockerfile into a docker image:

Bash shell creating a tagged docker image

```
sudo docker build -t example/nemesyst .
```

This tag "example/nemesyst" will help you reference the docker image later on, like easy removal, and general use.

7.1.4 Running

When we take a built image and run it, it is now called a container. Images are the immutable snapshots that you have built, containers are the changed containers for all the work that has happened since being an image.

To create a container from an image/ to run a docker image you can either:

Bash shell **creating/running a CPU only container from a tagged ("example/nemesyst") docker image**

```
sudo docker run -it example/nemesyst bash
```

or

Bash shell **creating/running a GPU enabled container ("example/nemesyst")**

```
sudo docker run --gpus all -it example/nemesyst bash
```

7.1.5 Cleaning up/ Removing

It may be necessary over the course of any experimentation or creation to occasionally clean up any images and containers that may still be taking up space on your system.

Bash shell **removing/ pruning everything**

```
sudo docker system prune
```

Bash shell **removing all images**

```
sudo docker rmi -f $(sudo docker images -q)
```

Bash shell **removing all containers**

```
sudo docker rm (sudo docker ps -a -q)
```

CHAPTER

EIGHT

OPTIONS

Nemesyst uses ConfigArgParse for argument handling. This means you may pass in arguments as (in order of highest priority first):

- CLI arguments
- Environment variables
- ini format .conf config files
- Hard-coded defaults

In code Nemesyst will look for config files in the following default locations, in order of priority and with expansion (highest first):

```
def default_config_files():
    """Default config file generator, for cleaner abstraction.

    :return: ordered list of config file expansions
    :rtype: list
    """
    config_files = [
        "./nemesyst.d/*.conf",
        "/etc/nemesyst/nemesyst.d/*.conf",
    ]
    return config_files
```

Using the –config argument you may specify more config files, which will be perpended to the default ones in the order supplied. Please note however config file locations are only followed once to avoid infinite loops where two configs point to each other, making Nemesyst read one then the other infinitely.

8.1 All Options by Category

```
usage: nemesyst [-h] [-U] [--prevent-update] [-c CONFIG [CONFIG ...]]
                [--process-pool PROCESS_POOL] [-d DATA [DATA ...]]
                [--data-clean]
                [--data-cleaner DATA_CLEANER [DATA_CLEANER ...]]
                [--data-cleaner-entry-point DATA_CLEANER_ENTRY_POINT [DATA_CLEANER_
↪ENTRY_POINT ...]]
                [--data-collection DATA_COLLECTION [DATA_COLLECTION ...]]
                [--dl-batch-size DL_BATCH_SIZE [DL_BATCH_SIZE ...]]
                [--dl-epochs DL_EPOCHS [DL_EPOCHS ...]] [--dl-learn]
                [--dl-learner DL_LEARNER [DL_LEARNER ...]]
                [--dl-learner-entry-point DL_LEARNER_ENTRY_POINT [DL_LEARNER_ENTRY_
↪POINT ...]]
```

(continues on next page)

(continued from previous page)

```
                 [--dl-data-collection DL_DATA_COLLECTION [DL_DATA_COLLECTION ...]]
                 [--dl-data-pipeline DL_DATA_PIPELINE [DL_DATA_PIPELINE ...]]
                 [--dl-input-model-collection DL_INPUT_MODEL_COLLECTION [DL_INPUT_
↪MODEL_COLLECTION ...]]
                 [--dl-input-model-pipeline DL_INPUT_MODEL_PIPELINE [DL_INPUT_MODEL_
↪PIPELINE ...]]
                 [--dl-output-model-collection DL_OUTPUT_MODEL_COLLECTION [DL_OUTPUT_
↪MODEL_COLLECTION ...]]
                 [--dl-sequence-length DL_SEQUENCE_LENGTH [DL_SEQUENCE_LENGTH ...]]
                 [--i-predictor I_PREDICTOR [I_PREDICTOR ...]]
                 [--i-predictor-entry-point I_PREDICTOR_ENTRY_POINT [I_PREDICTOR_ENTRY_
↪POINT ...]]
                 [--i-output-prediction-collection I_OUTPUT_PREDICTION_COLLECTION [I_
↪OUTPUT_PREDICTION_COLLECTION ...]]
                 [--i-predict] [--db-replica-set-name DB_REPLICA_SET_NAME]
                 [--db-replica-read-preference DB_REPLICA_READ_PREFERENCE]
                 [--db-replica-max-staleness DB_REPLICA_MAX_STALENESS]
                 [--db-tls] [--db-tls-ca-file DB_TLS_CA_FILE]
                 [--db-tls-certificate-key-file DB_TLS_CERTIFICATE_KEY_FILE]
                 [--db-tls-certificate-key-file-password DB_TLS_CERTIFICATE_KEY_FILE_
↪PASSWORD]
                 [--db-tls-crl-file DB_TLS_CRL_FILE] [-l] [-s] [-S] [-i]
                 [--db-user-name DB_USER_NAME] [--db-password DB_PASSWORD]
                 [--db-intervention] [--db-authentication DB_AUTHENTICATION]
                 [--db-authentication-database DB_AUTHENTICATION_DATABASE]
                 [--db-user-role DB_USER_ROLE] [--db-ip DB_IP]
                 [--db-bind-ip DB_BIND_IP [DB_BIND_IP ...]] [--db-port DB_PORT]
                 [--db-name DB_NAME] [--db-collection-name DB_COLLECTION_NAME]
                 [--db-config-path DB_CONFIG_PATH] [--db-path DB_PATH]
                 [--db-log-path DB_LOG_PATH] [--db-log-name DB_LOG_NAME]
                 [--db-cursor-timeout DB_CURSOR_TIMEOUT]
                 [--db-batch-size DB_BATCH_SIZE] [--db-pipeline DB_PIPELINE]
```

8.1.1 Nemesyst options

-U, --update	Nemesyst update, and restart. Default: False
--prevent-update	Prevent nemesyst from updating. Default: False
-c, --config	List of all ini files to be used. Default: []
--process-pool	The maximum number of processes to allocate. Default: 1

8.1.2 Data pre-processing options

-d, --data	List of data file paths.
	Default: []
--data-clean	Clean specified data files.
	Default: False
--data-cleaner	Path to data cleaner(s).
	Default: []
--data-cleaner-entry-point	Specify the entry point of custom scripts to use.
	Default: [‘main’]
--data-collection	Specify data storage collection name(s).
	Default: [‘debug_data’]

8.1.3 Deep learning options

--dl-batch-size	Batch size of the data to use.
	Default: [32]
--dl-epochs	Number of epochs to train on data.
	Default: [1]
--dl-learn	Use learner scripts.
	Default: False
--dl-learner	Path to learner(s).
	Default: []
--dl-learner-entry-point	Specify the entry point of custom scripts to use.
	Default: [‘main’]
--dl-data-collection	Specify data collection name(s).
	Default: [‘debug_data’]
--dl-data-pipeline	Specify pipeline(s) for data retrieval.
	Default: [{}]
--dl-input-model-collection	Specify model storage collection to retrain from.
	Default: [‘debug_models’]
--dl-input-model-pipeline	Specify model storage collection to retrain from.
	Default: [{}]
--dl-output-model-collection	Specify model storage collection to post trained neural networks to.
	Default: [‘debug_models’]
--dl-sequence-length	List of ints for how long a sequence ofdata should be/ expected.
	Default: [32]

8.1.4 Infering options

--i-predictor	Path to predictor(s).
	Default: []
--i-predictor-entry-point	Specify the entry point of predictor custom scripts to use.
	Default: [‘main’]
--i-output-prediction-collection	Specify prediction storage collection to post trained neural network predictions to.
	Default: [‘debug_predictions’]
--i-predict	Use predictor/ inferer scripts.
	Default: False

8.1.5 MongoDb replica options

--db-replica-set-name	Set the name for the replica set to use.
--db-replica-read-preference	Set the read preference of mongo client.
	Default: “primary”
--db-replica-max-staleness	Max seconds replica can be out of sync.
	Default: -1

8.1.6 MongoDb TLS options

--db-tls	Set connection to mongodb use TLS.
	Default: False
--db-tls-ca-file	Certificat-authority certificate path.
--db-tls-certificate-key-file	Clients certificate and key pem path.
--db-tls-certificate-key-file-password	Set pass if certkey file needs password.
--db-tls-crl-file	Path to certificate revocation list file.

8.1.7 MongoDb options

-l, --db-login	Nemesyst log into mongodb.
	Default: False
-s, --db-start	Nemesyst launch mongodb.
	Default: False
-S, --db-stop	Nemesyst stop mongodb.
	Default: False
-i, --db-init	Nemesyst initialise mongodb files.
	Default: False

--db-user-name	Set mongodb username.
--db-password	Set mongodb password.
	Default: False
--db-intervention	Manual intervention during database setup.
	Default: False
--db-authentication	Set the mongodb authentication method.
	Default: "SCRAM-SHA-1"
--db-authentication-database	Override db_name as database to authenticate.
--db-user-role	Set the users permissions in the database.
	Default: "readWrite"
--db-ip	The ip of the database to connect to.
	Default: "localhost"
--db-bind-ip	The ip the database should be accessible from.
	Default: ['localhost']
--db-port	The port both the unauth and auth db will use.
	Default: "65535"
--db-name	The name of the authenticated database.
	Default: "nemesyst"
--db-collection-name	The name of the collection to use in database.
	Default: "test"
--db-config-path	The path to the mongodb configuration file.
--db-path	The parent directory to use for the database.
	Default: /home/docs/db
--db-log-path	The parent directory to use for the db log.
	Default: /home/docs/db/log
--db-log-name	The base name of the log file to maintain.
	Default: "mongo_log"
--db-cursor-timeout	The duration in seconds before an unused cursor will time out.
	Default: 600000
--db-batch-size	The number of documents to return from the db at once/ pre round.
	Default: 32
--db-pipeline	The file path of the pipeline to use on db.

CHAPTER NINE

LOGGER

Nemesyst logging utility/ tool. This handler helps give the user and developer more granular control of logging/ output, and leaves expansion possible for new and more complex scenarios.

9.1 API

class `logger.`**`Logger`**(*args: dict = None*)

Python logger utility.

This logger utility helps output in desired manner in slightly more configurable manner than simple print().

Parameters **`args`** (*`dictionary`*) – Dictionary of overides.

Example Logger().log("Hello, world.")

Example Logger({"log_level": 5,}).log("Hello, world.")

`log`(**text, log_level: int = None, min_level: int = None, delimiter: str = None*) → None

Log desired output to teminal.

Parameters

- **`*text`** – The desired text to log.
- **`log_level`** (*`int`*) – Current log level/ log level override.
- **`min_level`** (*`int`*) – Minimum required log level to display text.
- **`delimiter`** (*`str`*) – String to place in between positional *text.

Returns None

Example Logger({log_level:2}).log("Hello, world.", min_level=0)

Example Logger().log("Hello", "world.", delimiter=", ")

CHAPTER

TEN

MONGO

Nemesyst MongoDB abstraction/ Handler. This handler helps abstract some pymongo functionality to make it easier for us to use a MongoDB database for our deep learning purposes.

10.1 Example usage

Below follows a in code example unit test for all functionality. You can override the options using a dictionary to the constructor or as keyword arguments to the functions that use them:

```
def _mongo_unit_test():
    """Unit test of MongoDB compat."""
    import datetime
    import pickle
    # create Mongo object to use
    db = Mongo({"test2": 2, "db_port": "65535"})
    # testing magic functions
    db["test2"] = 3  # set item
    db["test2"]  # get item
    len(db)  # len
    del db["test2"]  # del item
    # output current state of Mongo
    db.debug()
    # stop any active databases already running at the db path location
    db.stop()
    # hold for 2 seconds to give the db time to start
    time.sleep(2)
    # attempt to initialise the database, as in create the database with users
    db.init()
    # hold to let the db to launch the now new unauthenticated db
    time.sleep(2)
    # start the authenticated db, you will now need a username password access
    db.start()
    # warm up time for new authentication db
    time.sleep(2)
    # create a connection to the database so we can do database operations
    db.connect()
    db.debug()
    # import data into mongodb debug collection
    db.dump(db_collection_name="test", data={
        "string": "99",
        "number": 99,
        "binary": bin(99),
        "subdict": {"hello": "world"},
```

(continues on next page)

(continued from previous page)

```
        "subarray": [{"hello": "worlds"}, {"hi": "jim"}],
        "timedate": datetime.datetime.utcnow(),
    })
    # testing gridfs insert item into database
    db.dump(db_collection_name="test", data=(
        {"utctime": datetime.datetime.utcnow()},
        b"some_test_string"
        # pickle.dumps("some_test_string")
    ))
    # log into the database so user can manually check data import
    db.login()
    # attempt to retrieve the data that exists in the collection as a cursor
    c = db.getCursor(db_collection_name="test", db_pipeline=[{"$match": {}}])
    # itetate through the data in batches to minimise requests
    for dataBatch in db.getBatches(db_batch_size=32, db_data_cursor=c):
        print("Returned number of documents:", len(dataBatch))
    # define a pipeline to get the latest gridfs file in a given collection
    fs_pipeline = [{'$sort': {'uploadDate': -1}},
                   {'$limit': 5},
                   {'$project': {'_id': 1}}]
    # get a cursor to get us the ID of files we desire
    fc = db.getCursor(db_collection_name="test.files", db_pipeline=fs_pipeline)
    # use cursor and get files to collect our data in batches
    for batch in db.getFiles(db_batch_size=2, db_data_cursor=fc):
        for doc in batch:
            # now read the gridout object
            print(doc["gridout"].read())
    # finally close out database
    db.stop()
```

This unit test also briefly shows how to use gridfs by dumping tuple items in the form (dict(), object), where the dict will become the files metadata and the object is some form of the data that can be sequentialized into the database.

Warning: Mongo uses subprocess.Popen in init, start, and stop, since these threads would otherwise lock up nemesyst, with time.sleep() to wait for the database to startup, and shutdown. Depending on the size of your database it may be necessary to extend the length of time time.sleep() as larger databases will take longer to startup and shutdown.

10.2 API

class `mongo.`**`Mongo`**(*args: dict = None*, *logger: print = None*)

Python2/3 compatible MongoDb utility wrapper.

This wrapper saves its state in an internal overridable dictionary such that you can adapt it to your requirements, if you should need to do something unique, the caveat being it becomes harder to read.

Parameters

- **`args`** (*`dictionary`*) – Dictionary of overides.
- **`logger`** (*`function address`*) – Function address to print/ log to (default: print).

Example Mongo({“db_user_name”: “someUsername”, “db_password”: “somePassword”})

Example Mongo()

connect(*db_ip: str = None*, *db_port: str = None*, *db_authentication: str = None*, *db_authentication_database=None*, *db_user_name: str = None*, *db_password: str = None*, *db_name: str = None*, *db_replica_set_name: str = None*, *db_replica_read_preference: str = None*, *db_replica_max_staleness: str = None*, *db_tls: bool = None*, *db_tls_ca_file: str = None*, *db_tls_certificate_key_file: str = None*, *db_tls_certificate_key_file_password: str = None*, *db_tls_crl_file: str = None*, *db_collection_name: str = None*) → pymongo.database.Database

Connect to a specific mongodb database.

This sets the internal db client which is neccessary to connect to and use the associated database. Without it operations such as dump into the database will fail. This is replica set capable.

Parameters

- **db_ip** (*string*) – Database hostname or ip to connect to.
- **db_port** (*string*) – Database port to connect to.
- **db_authentication** (*string*) – The authentication method to use on db.
- **db_user_name** (*string*) – Username to use for authentication to db_name.
- **db_password** (*string*) – Password for db_user_name in database db_name.
- **db_name** (*string*) – The name of the database to connect to.
- **db_replica_set_name** (*string*) – Name of the replica set to connect to.
- **db_replica_read_preference** (*string*) – What rep type to prefer reads from.
- **db_replica_max_staleness** (*string*) – Max seconds behind is replica allowed.
- **db_tls** (*bool*) – use TLS for db connection.
- **db_tls_certificate_key_file** (*string*) – Certificate and key file for tls.
- **db_tls_certificate_key_file_password** (*string*) – Cert and key file pass.
- **db_tls_crl_file** (*string*) – Certificate revocation list file path.
- **db_collection_name** (*string*) – GridFS collection to use.

Returns database client object

Return type pymongo.database.Database

debug() → None

Log function to help track the internal state of the class.

Simply logs working state of args dict.

dump(*db_collection_name: str*, *data: dict*, *db: pymongo.database.Database = None*) → None

Import data dictionary into database.

Parameters

- **db_collection_name** (*string*) – Collection name to import into.
- **data** (*dictionary*) – Data to import into database.
- **db** (*pymongo.database.Database*) – Database to import data into.

Example dump(db_collection_name=”test”, data={“subdict”:{“hello”: “world”}})

getBatches(*db_batch_size: int = None*, *db_data_cursor: pymongo.command_cursor.CommandCursor = None*) → list

Get database cursor data in batches.

Parameters

- **db_batch_size** (*integer*) – The number of items to return in a single round.
- **db_data_cursor** (*command_cursor.CommandCursor*) – The cursor to use to retrieve data from db.

Returns yields a list of items requested.

Return type list of dicts

Todo desperateley needs a rewrite and correction of bug. Last value always fails. I want this in a magic function too to make it easy.

getCursor(*db: pymongo.database.Database = None*, *db_pipeline: list = None*, *db_collection_name: str = None*) → pymongo.command_cursor.CommandCursor

Use aggregate pipeline to get a data-cursor from the database.

This cursor is what mongodb provides to allow you to request the data from the database in a manner you control, instead of just getting a big dump from the database.

Parameters

- **db_pipeline** (*list of dicts*) – Mongodb aggregate pipeline data to transform and retrieve the data as you request.
- **db_collection_name** (*str*) – The collection name which we will pull data from using the aggregate pipeline.
- **db** (*pymongo.database.Database*) – Database object to operate pipeline on.

Returns Command cursor to fetch the data with.

Return type pymongo.command_cursor.CommandCursor

getFiles(*db_batch_size: int = None*, *db_data_cursor: pymongo.command_cursor.CommandCursor = None*, *db_collection_name: str = None*, *db: pymongo.database.Database = None*) → list

Get gridfs files from mongodb by id using cursor to .files.

Parameters

- **db_batch_size** (*integer*) – The number of items to return in a single round.
- **db_data_cursor** (*command_cursor.CommandCursor*) – The cursor to use to retrieve data from db.
- **db_collection_name** (*str*) – The top level collecton name not including .chunks or .files where gridfs is to operate.
- **db** (*pymongo.database.Database*) – Database object to operate pipeline on.

Returns yields a list of tuples containing (item requested, metadata).

init(*db_path: str = None*, *db_log_path: str = None*, *db_log_name: str = None*, *db_config_path: str = None*) → None

Initialise the database.

Includes ensuring db path and db log path exist and generating, creating the DB files, and adding an authentication user. All of this should be done on a localhost port so that the unprotected database is never exposed.

Parameters

- **db_path** (*string*) – Desired directory of MongoDB database files.
- **db_log_path** (*string*) – Desired directory of MongoDB log files.

- **db_log_name** (*string*) – Desired name of log file.
- **db_config_path** (*string*) – Config file to pass to MongoDB.

login (*db_port: str = None, db_user_name: str = None, db_password: str = None, db_name: str = None, db_ip: str = None*) → None

Log in to database, interrupt, and availiable via cli.

Parameters

- **db_port** (*string*) – Database port to connect to.
- **db_user_name** (*string*) – Database user to authenticate as.
- **db_password** (*string*) – User password to authenticate with.
- **db_name** (*string*) – Database to authenticate to, the authentication db.
- **db_ip** (*string*) – Database ip to connect to.

start (*db_ip: str = None, db_port: str = None, db_path: str = None, db_log_path: str = None, db_log_name: str = None, db_cursor_timeout: int = None, db_config_path: str = None, db_replica_set_name: str = None*) → subprocess.Popen

Launch an on machine database with authentication.

Parameters

- **db_ip** (*list*) – List of IPs to accept connectiongs from.
- **db_port** (*string*) – Port desired for database.
- **db_path** (*string*) – Path to parent dir of database.
- **db_log_path** (*string*) – Path to parent dir of log files.
- **db_log_name** (*string*) – Desired base name for log files.
- **db_cursor_timeout** (*integer*) – Set timeout time for unused cursors.
- **db_path** – Config file path to pass to MongoDB.

Return type subprocess.Popen

Returns Subprocess of running MongoDB.

stop (*db_path=None*) → subprocess.Popen

Stop a running local database.

Parameters **db_path** (*string*) – The path to the database to shut down.

Returns Subprocess of database closer.

Return type subprocess.Popen

CHAPTER

ELEVEN

TROUBLESHOOTING

11.1 Tensorflow Issues

tensorflow.python.framework.errors_impl.UnknownError If you are using an RTX graphics card this is more than likeley due to your tesnorflow not supporting them. Simply either use the CPU, another graphics card, or re-compile tensorflow on your system so that it has RTX support.

11.2 MongoDB/ Serving Issues

Error: not master and slaveOk=false This error means you have attempted to read from a replica set that is not the master. If you would like to read from SECONDARY-ies/ slaves (anything thats not the PRIMARY) you can:

Mongo shell:

```
rs.slaveOk()
```

pymongo.errors.OperationFailure: Authentication failed This error means likely means that your authentication credentials are incorrect, you will want to check the values you are passing to pymongo via Nemesyst to ensure they are what you are expecting. In particular pay special attention to Mongo().connect() as it is the life blood of all connections but since the driver is a lazy driver it wont fail until you attempt to use the connection.

INDEX