Nebulae Brochure

A novel and simple framework based on concurrent mainstream frameworks and other image processing libraries. It is convenient to deploy almost every module independently.

H1

Installation

```
pip install nebulae
```

H2 The latest version supports PyTorch1.6 and TensorFlow2.3

Modules Overview

Fuel: easily manage and read dataset you need anytime

Toolkit: includes many utilities for better support of nebulae

---

H2

Fuel

FuelGenerator(file_dir, file_list, dtype, is_seq)

Build a FuelGenerator to spatial efficently store data.

H2

- config: [dict] A dictionary containing all parameters. The other arguments and this are mutually exclusive.
- file_dir: [str] Where your raw data is.
- file_list: [str] A csv file in which all the raw datum file name and labels are listed.
- dtype: [list of str] A list of data types of all columns but the first one in *file_list* . Valid data types are 'uint8', 'uint16', 'uint32', 'int8', 'int16', 'int32', 'int64', 'float16', 'float32', 'float64', 'str'. Plus, if you add a 'v' as initial character e.g. 'vuint8', the data of each row in this column is allowed to be saved in variable length.
- is_seq: [bool] If it is data sequence e.g. video frames. Defaults to false.

An example of file_list.csv is as follow. 'image' and 'label' are the key names of data and labels respectively. Note that the image name is a path relative to *file_dir*.

image	label
img_1.jpg	2
img_2.jpg	0
...	...
img_100.jpg	5

if *is_seq* is True, the csv file is supposed to look like the example below (when char-separator is ',' and quoting-char is '"'):

image	label
"vid_1_frame_1.jpg,vid_1_frame_2.jpg,...,vid_1_frame_24.jpg"	2
"vid_2_frame_1.jpg,vid_2_frame_2.jpg,...,vid_2_frame_15.jpg"	0
...	...
"vid_100_frame_1.jpg,vid_100_frame_2.jpg,...,vid_100_frame_39.jpg"	5

FuelGenerator.generate(dst_path, height, width, channel=3, encode='JPEG', shards=1, keep_exif=True)

- dst_path: [str] A hdf5/npz file where you want to save the data.
- height: [int] range between (0, +∞). The height of image data.
- width: [int] range between (0, +∞). The height of image data.
- channel: [int] The height of image data. Defaults to 3.
- encode: [str] The mean by which image data is encoded. Valid encoders are 'jpeg' and 'png'. 'PNG' is the way without information loss. Defaults to 'JPEG'.
- shards: [int] How many files you need to split the data into. Defaults to 1.
- keep_exif: [bool] Whether to keep EXIF information of photos. Defaults to true.

```
import nebulae as neb
# create a data generator
fg = neb.fuel.Generator(file_dir='/home/file_dir',
                        file_list='file_list.csv',
                        dtype=['vuint8', 'int8'])
# generate compressed data file
fg.generate(dst_path='/home/data/fuel.hdf5',
            channel=3,
            height=224,
            width=224)
```

FuelGenerator.modify(config=None)

You can edit properties again for generating other file.

```
fg.modify(height=200, width=200)
```

Passing a dictionary of changed parameters is equivalent.

```
config = {'height': 200, 'width': 200}
fg.modify(config=config)
```

Tank(data_path, data_specf, batch_size, shuffle, in_same_size, fetch_fn, prep_fn, collate_fn)

Build a Fuel Tank that allows you to deposit datasets.

- data_path: [str] The full path of your data file. It must be a hdf5/npz file.
- data_specf: [dict] A dictionary containing key-dtype pairs.
- batch_size: [int] The size of a mini-batch.
- shuffle: [bool] Whether to shuffle data samples every epoch. Defaults to True.
- in_same_size: [bool] Whether to ensure the last batch has samples as many as other batches. Defaults to True.
- fetch_fn: [func] The function which takes a single datum from dataset.
- prep_fn: [func] The function which preprocesses fetched datum. Defaults to None.
- collate_fn: [func] The function which concatenates data as a mini-batch. Defaults to None.

E.g.

```
from nebulae.fuel import depot
# define data-reading functions
def fetcher(data, idx):
  ret = {}
  ret['image'] = data['image'][idx]
  ret['label'] = data['label'][idx].astype('int64')
  return ret

def prep(data):
  # convert to channel-first format
  data['image'] = np.transpose(data['image'], (2, 0,
1)).astype('float32')
  return data

# create a data depot
tk = depot.Tank("/home/dataset.hdf5",
                {'image': 'vunit8', 'label': 'int64'},
                batch_size=128, shuffle=True,
                fetch_fn=fetcher, prep_fn=prep)
```

Tank.next()

Return a batch of data, labels and other information.

Tank.MPE

Attribute: how many iterations there are within an epoch for each dataset.

len(Tank)

Attribute: the number of datum in this dataset.

Comburant()

Comburant is a container to pack up all preprocessing methods.

Data Source	Augmentation	Usage
Image	flip	flip matrix vertically or horizontally
	crop	crop matrix randomly with a given area and aspect ratio
	rotate	rotate matrix randomly within a given range
	brighten	adjust brightness given an increment/decrement factor
	contrast	adjust contrast given an expansion/shrinkage factor
	saturate	adjust saturation given an expansion/shrinkage factor
Sequence	sampling	positive int, denoted as theta: sample an image every theta frames

---

Aerolog

DashBoard(log_path='./aerolog', window=1, divisor=10, span=30, format=None)

- log_path: [str] The place where logs will be stored in.
- window: [int] Window size for moving average.
- divisor: [int] How many segments the Y axis is to be divided into.
- span: [int] The length of X axis.
- format: [dict of tuple] Specify the format in which results will be displayed every step. Four available format types are 'raw', 'percent', 'tailor' and 'inviz'.

H2

DashBoard is a terminal-favored tool for monitoring your training procedure. It prints a dynamically refreshed progress bar with some important metrics. Besides, it shows a real-time changed curve to visualize how well the training and test phases go.

DashBoard.gauge(entry, mile, epoch, mpe, stage, interval=1, duration=None)

- entry: [dict] K-V pairs to be displayed.
- mile: [int] The current step.
- epoch: [int] The current epoch.
- mpe: [int] The number of steps within an epoch, i. e. Miles Per Epoch
- stage: [str] The name of current stage.
- interval: [int] Display results every a fixed number of steps.
- duration: [float] The elapsed time since last step or epoch

Call this function after a step or epoch to monitor current states.

DashBoard.log(gauge, tachograph, record)

- gauge: [bool] Whether to draw metrics as a picture.
- tachograph: [bool] Whether to write down metrics as a csv file.
- record: [str] Append metrics logged this time after the recording file.

Write the intermediate results into files.

Cockpit

Engine(device, ngpus=1, least_mem=2048, available_gpus=())

H2

- device: [str] To run the model on which type of devices. You can either choose 'CPU' or 'GPU'.
- ngpus: [int] The number of GPUs to be taken. This argument doesn't work when device is CPU.
- least_mem: [int] Only the GPUs with at least this amount of memory left are regarded as available.
- available_gpus: [tuple of int] Set the available GPUs explicitly. This argument doesn't work when device is CPU. In default case, it is set to empty tuple and Nebulae will detect available GPUs automatically.

An Engine would take care of the devices to be used especially GPU environment. Usually you need to gear it into your Craft.

TimeMachine(ckpt_path, save_path, max_anchors=-1)

- ckpt_path: [str] The place where checkpoints will be read from.
- save_path: [str] The place where checkpoints will be stored.
- max_anchors: [int] The max number of checkpoints to be kept.

Manage checkpoint saving and reading by creating a TimeMachine.

TimeMachine.to(craft, file='', ckpt_scope=None, frozen=False)

- craft: [Craft] NN model.
- file: [str] The file name of checkpoint.
- ckpt_scope: [str] Only the parameters inside the scope can be loaded.
- frozen: [bool] Frozen model means omitting unmatched part is not allowed.

TimeMachine.drop(craft, file='', save_scope=None)

- craft: [Craft] NN model.
- file: [str] The file name of checkpoint.
- save_scope: [str] Only the parameters inside the scope can be saved.

Toolkit

GPUtil()

Create a tool for monitoring GPU status. In fact, it is leveraged implicitly when instantiate an Engine.

H2

GPUtil.available(ngpus, least_mem)

- ngpus: [int] The number of GPUs to be selected
- least_mem: [int] The least free memory (MiB) a valid GPU should have.

Returns a list of available GPU information including serial number, name and memory. If the available GPUs are not sufficient, it contains the most devices system can offer.

GPUtil.monitor(sec)

- sec: [int] The monitor logs every a few seconds. Default to 5.

Start monitoring GPUs. Note that setting sec as a small number might cause abnormal statistics. Turn it bigger if there are too many GPUs on your machine.

GPUtil.status()

Stop monitoring and returns GPUs status summary.

Astrobase

Craft(scope)

- scope: [str] Name of this craft. It will be the base name of saved model file by default.

Craft is a base class of neural network which is compatible with the backend framework. It is convenient especially when you want to fork open-sourced codes into nebulae or when you find it difficult to implement a desired function.

H2

```
from nebulae.astrobase import dock
import torch
# designate pytorch as core backend
nebulae.Law.CORE = 'pytorch'
# create a network using nebulae
class NetNeb(dock.Craft):
    def __init__(self, nclass, scope='NEB'):
        super(Net, self).__init__(scope)
        self.flat = dock.Reshape()
        self.fc = dock.Dense(512, nclass)

    def run(self, x):
        x = self.flat(x, (-1, 512))
        y = self.fc(x)
        return y
# you can create a same network using pytorch functions
class NetTorch(dock.Craft):
    def __init__(self, nclass, scope='TORCH'):
        super(Net, self).__init__(scope)
        self.fc = torch.nn.Linear(512, nclass)

    def run(self, x):
        x = x.view(-1, 512)
        y = self.fc(x)
        return y
```

Rudder()

Rudder is a context in which all gradients will be computed and backpropogated through variables accordingly.

Prober()

Prober is a context in which all gradients will be computed but not backpropogated.

Nozzle()

Nozzle is a context in which all variables are fixed and no gradient is going to be computed.

coat(datum, as_const)

- datum: [int/float/array/tensor] input datum.
- as_const: [bool] whether to make datum as an untrainable tensor. Defaults to True.

The input tensor will be put into current used device. Any legal input is going to be converted to a tensor at first.

shell(datum, as_np)

- datum: [tensor] input datum.
- as_np: [bool] whether to make datum as a regular array. Defaults to True.

The input tensor will be taken out from current used device.

autoPad(in_size, kernel, stride, dilation)

- in_size: [tuple] input size e.g. (h, w) for 2d tensor
- kernel: [tuple] kernel size
- stride: [tuple/int] convolution stride
- dilation: [tuple/int] stride in atrous convolution

Return the padding tuple leading to an output in same size with input when stride is 1. e.g. (left, right, top, bottom, front, back) for 3d tensor

Multiverse(universe, nworld=1)

- universe: [Craft] NN model.
- nworld: [int] The number of devices on which model would be distributed.

Multiverse is a parallelism manager that makes it easier to deploy distributed training. The following example code shows what's the difference between regular training and distributed training.

```
import nebulae as neb
from nebulae.fuel import depot
from nebulae.astrobase import dock
from nebulae.cockpit import Engine, TimeMachine

            ############
def launch(mv=None): ## <====================================[1]
            ############

    ############
    mv.init() ## <=========================================[2]
    ############
```

```
# ---------------------- Aerolog ----------------------- #
# defined Dashboard

# ---------------------- Cockpit ----------------------- #
                          ###################
ng = Engine(device="gpu", ngpus=mv.nworld) ## <==========[3]
                          ###################
tm = TimeMachine(save_path="./ckpt",
                 ckpt_path="./ckpt")

# ---------------------- Fuel ------------------------- #
# define dataset Tanks

# --------------------- Space Dock --------------------- #
###################
with mv.scope(): ## <====================================[4]
###################
    class Net(dock.Craft):
        def __init__(self, nclass, scope):
            super(Net, self).__init__(scope)
            # define architecutre

        def run(self, x):
            # define forward procedure

    class Train(dock.Craft):
        def __init__(self, net, scope='TRAIN'):
            super(Train, self).__init__(scope)
            self.net = net
            # initialize params

        ###############
        @mv.Executor ## <================================[5]
        ###############
        def run(self, x, z):
            # define training step
            loss = self.net(x, z)
            ########################
            loss = mv.reduce(loss) ## <==================[6]
            ########################

    class Dev(dock.Craft):
```

```
            def __init__(self, net, scope='DEVELOP'):
                super(Dev, self).__init__(scope)
                self.net = net
                # initialize params

            ###############
            @mv.Executor ## <================================[5]
            ###############
            def run(self, x, z, idx):
                # define testing step
                loss = self.net(x, z)
                ########################
                loss = mv.reduce(loss) ## <==================[6]
                ########################

    # ---------------------- Launcher ---------------------- #
    net = Net(10, 'cnn')
    net.gear(ng)
    ######################################################
    net, dataT, dataV = mv.sync(net, (dataT, dataV)) ## <====[7]
    ######################################################
    train = Train(net)
    dev = Dev(net)

    for epoch in range(10):
        for mile in range(mpe):
            # train and test

if __name__ == '__main__':
    # ------------------- Global Setting --------------------- #
    if is_distributed:
        ####################################
        mv = neb.law.Multiverse(launch, 4) ##
        mv()                                ## <==============[8]
        ####################################
    else:
        launch()
```

Multiverse.init()

Initialize the multiverse manager.

Multiverse.scope()

Create a parallel scope.

Multiverse.sync(models, data)

- models: [Craft] NN model.
- data: [tuple of Tank] All datasets in use

Synchronize the distributed models and data.

Multiverse.reduce(tensor, aggregate=False)

- tensor: [tensor] Returned tensor to be reduced.
- aggregate: [bool] Either to sum all tensors up or average on them. Defaults to False which means taking average.

Collect results returned by all distributed devices.

Multiverse.Executor

A decorator poiting out the main function which is usually charged with training and testing.