eve-mli

Release 0.1.0

densechen

Mar 19, 2022

CONTENTS:

eve-mli is an open-source deep learning framework used to devise various architectures in a more flexible and interesting way.

CHAPTER

ONE

OPENSPIKING

GitHub last commit Documentation Status PyPI PyPI - Python Version

OpenSpiking is an open-source deep learning framework used to devise and modify a network architecture in a flexible and interesting way.

We provide several jupyter notebooks under `./examples` to demonstrate how to build various network structures.

The most features of OpenSpiking are that: it provides a well defined framework which make your network structure can be upgraded along with the learning of weights.

Any contributions to OpenSpiking is welcome!

1.1 Installation

Install from PyPI:

```
pip install eve-mli
# pip install git+https://github.com/densechen/eve-mli.git
```

Developers can download and install the latest version from:

GitHub:

```
git clone https://github.com/densechen/eve-mli.git
cd eve-mli
python setup.py install
```

Gitee:

```
git clone https://gitee.com/densechen/eve-mli.git
cd eve-mli
python setup.py install
```

Vailidate installation:

```
python -c "import eve; print(eve.__version__)"
```

1.2 Quick Start

The core module of eve-mli is `eve.core.Eve`, this module is a wrapper of `torch.nn.Module`.

In `Eve`, the parameter ended with `_eve` will be treated as an eve parameters, and we call the rest as torch parameters. In the same way, we also define eve buffers and torch buffers.

As for eve parameters, you can fetch and attach an `.obs` properties via `eve.core.State` class, and assign an upgrade function to modify the eve parameter. As for eve buffers, it is useful to cache the hidden states, all the eve buffers will be cleared once we call `Eve.reset()`.

In default, the model defined by `Eve` is the same with `nn.Module`. You can train it directly for obtaining a baseline model. Then, `Eve.spike()` will turn it into a spiking neural network module, and `Eve.quantize()` will trun it into a quantization neural network model.

1.3 About the project

The documentation can be found here. (Auto-building of documentation fails sometimes, you can build it manually via `cd docs; make html`).

The project remains in development. We encourage more volunteers to come together!

eve-mli-v0.1.0 is released!

1.4 Next to do

Add CUDA support for speeding up, refer to here.

1.5 About the authors

Dengsheng Chen Master @ National University of Defense Technology densechen@foxmail.com

1.6 References

PyTorch

stable-baselines3

spikingjelly

Model-Compression-Deploy

Awesome-Deep-Neural-Network-Compression

CHAPTER

TWO

MIT LICENSE

CHAPTER

THREE

GUIDANCE

3.1 Spiking Neural Network with Eve

In this script, we will build a general spiking neural network with eve-mli, then we will further investigating the influence of different Node, i.e. IFNode and LIFNode combined with different surrotage functions.

```
# import necessary packages.
# at the beginning, ensure that the eve-mli package is in your python path.
# or you just install it via `pip install eve-mli`.

import os
import time
from datetime import datetime

import random
import numpy as np
import torch as th
import torch.nn as nn
import torch.nn.functional as F

import eve
import eve.app
import eve.app.model
import eve.app.trainer
import eve.core
import eve.app.space as space
import eve.core.layer

from matplotlib import pyplot as plt
%matplotlib inline

os.environ["CUDA_VISIBLE_DEVICES"] = '1'
```

```
# build a basic network for trainer, use Poisson Encoder as default
class mnist(eve.core.Eve):
    def __init__(self,
                 timesteps: int = 5,
                 node: str="IFNode",
                 voltage_threshold: float = 1.0,
                 voltage_reset: float = 0.0,
```

(continues on next page)

(continued from previous page)

```
                learnable_threshold: bool = False,
                learnable_reset: bool = False,
                time_dependent: bool = True,
                neuron_wise: bool = True,
                surrogate_fn: str = "Sigmoid",
                binary: bool = True,
                **kwargs,):
        super().__init__()

        # we support IFNode and LIFNode,
        # in non-spiking mode, the IFNode equals to a ReLU operation
        # the LIFNode equals to a LeakyReLU operation.
        node = getattr(eve.core.node, node)

        def build_node(state):
            return node(state,
                        voltage_threshold=voltage_threshold,
                        voltage_reset=voltage_reset,
                        learnable_threshold=learnable_threshold,
                        learnable_reset=learnable_reset,
                        time_dependent=time_dependent,
                        neuron_wise=neuron_wise,
                        surrogate_fn=surrogate_fn,
                        binary=binary,
                        **kwargs,)

        self.encoder = eve.core.layer.PoissonEncoder(timesteps=timesteps)

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 4, 3, stride=2, padding=1),
            nn.BatchNorm2d(4),
        )
        self.node1 = build_node(eve.core.State(self.conv1))

        self.conv2 = nn.Sequential(
            nn.Conv2d(4, 8, 3, stride=2, padding=1),
            nn.BatchNorm2d(8),
        )
        self.node2 = build_node(eve.core.State(self.conv2))

        self.conv3 = nn.Sequential(
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
        )
        self.node3 = build_node(eve.core.State(self.conv3))

        self.linear1 = nn.Linear(16 * 4 * 4, 16)
        self.node4 = build_node(eve.core.State(self.linear1))

        self.linear2 = nn.Linear(16, 10)

    def forward(self, x):
```

(continues on next page)

(continued from previous page)

```
        encoder = self.encoder(x)

        conv1 = self.conv1(encoder)
        node1 = self.node1(conv1)

        conv2 = self.conv2(node1)
        node2 = self.node2(conv2)

        conv3 = self.conv3(node2)
        node3 = self.node3(conv3)

        node3 = th.flatten(node3, start_dim=1).unsqueeze(dim=1)

        linear1 = self.linear1(node3)
        node4 = self.node4(linear1)

        linear2 = self.linear2(node4)

        return linear2.squeeze(dim=1)
```

```
# define a MnistClassifier
# Classifier uses the corss entropy as default.
# in most case, we just rewrite the `prepare_data`.
class MnistClassifier(eve.app.model.Classifier):
    def prepare_data(self, data_root: str):
        from torch.utils.data import DataLoader, random_split
        from torchvision import transforms
        from torchvision.datasets import MNIST

        train_dataset = MNIST(root=data_root,
                              train=True,
                              download=True,
                              transform=transforms.ToTensor())
        test_dataset = MNIST(root=data_root,
                             train=False,
                             download=True,
                             transform=transforms.ToTensor())
        self.train_dataset, self.valid_dataset = random_split(
            train_dataset, [55000, 5000])
        self.test_dataset = test_dataset

        self.train_dataloader = DataLoader(self.train_dataset,
                                           batch_size=128,
                                           shuffle=True,
                                           num_workers=4)
        self.test_dataloader = DataLoader(self.test_dataset,
                                          batch_size=128,
                                          shuffle=False,
                                          num_workers=4)
        self.valid_dataloader = DataLoader(self.valid_dataset,
                                           batch_size=128,
                                           shuffle=False,
```

(continues on next page)

(continued from previous page)

```
                                            num_workers=4)

    def forward(self, batch_idx, batch, *args, **kwargs):
        # in spiking mode, you should reset the membrane voltage every time.
        self.reset()

        x, y = batch

        y_hat = [self.model(x) for _ in range(self.model.encoder.timesteps)]
        y_hat = th.stack(y_hat, dim=0).mean(dim=0)

        return {
            "loss": F.cross_entropy(y_hat, y),
            "acc": self._top_one_accuracy(y_hat, y),
        }
```

```
# store accuracy result
y = {}
def plot():
    global y
    keys, values = list(y.keys()), list(y.values())
    for k, v in y.items():
        plt.plot(v,
                 color='green' if random.random() > 0.5 else "red",
                 marker='o' if random.random() > 0.5 else "*",
                 linestyle='-' if random.random() > 0.5 else ":",
                 label=k)
    plt.title('accuracy over epoches (train)')
    plt.xlabel('epochs')
    plt.ylabel('accuracy')
    plt.legend(loc="upper left")
    plt.show()
```

```
def train(net, exp_name: str = "snn", data_root: str = "/home/densechen/dataset"):
    global y
    # replace the data_root for your path.
    classifier = MnistClassifier(net)
    classifier.prepare_data(data_root=data_root)

    # use default configuration
    classifier.setup_train()

    # assign model to trainer
    eve.app.trainer.BaseTrainer.assign_model(classifier)

    trainer = eve.app.trainer.BaseTrainer()

    # train 10 epoches and report the final accuracy
    y[exp_name] = []
    tic = datetime.now()
    for _ in range(10):
```

(continues on next page)

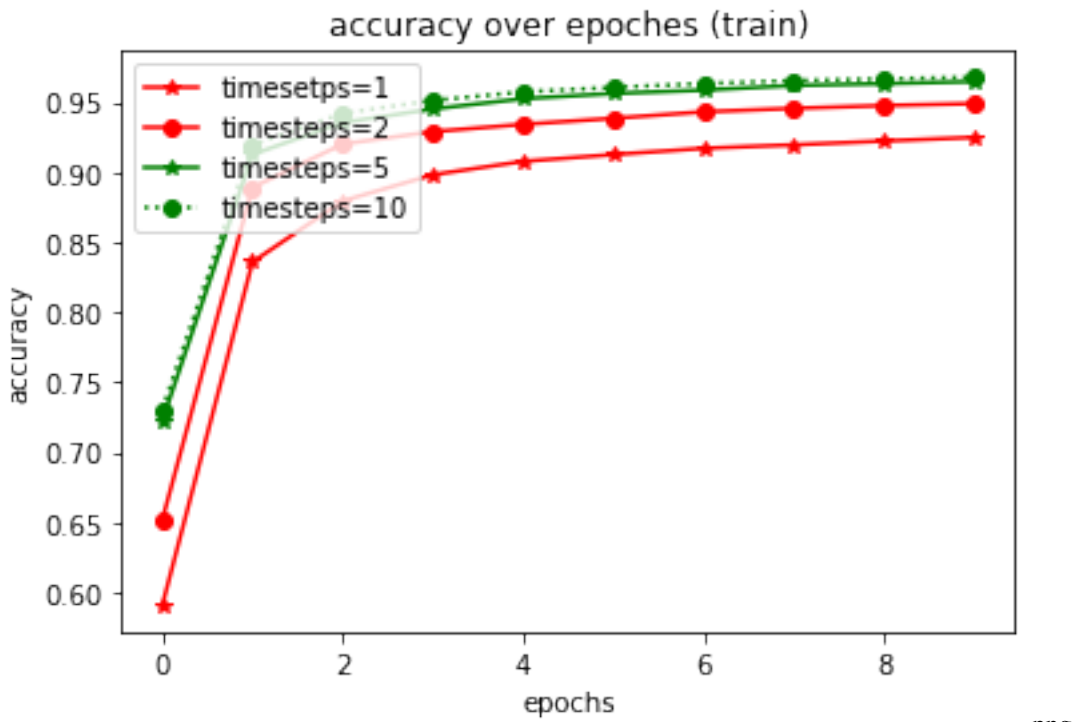
(continued from previous page)

```
        info = trainer.fit()
        y[exp_name].append(info["acc"])
    info = trainer.test()
    toc = datetime.now()
    y[exp_name] = np.array(y[exp_name])
    print(f"Test Accuracy: {info['acc']*100:.2f}%, Elapsed time: {toc-tic}")
```

3.1.1 IFNode with different timesteps

```
# reset y
y = {}

# define spiking neural network with different timesteps
ifnode_spiking_neural_network_1_timesteps = mnist(timesteps=1).spike()
ifnode_spiking_neural_network_2_timesteps = mnist(timesteps=2).spike()
ifnode_spiking_neural_network_5_timesteps = mnist(timesteps=5).spike()
ifnode_spiking_neural_network_10_timesteps = mnist(timesteps=10).spike()

# train and compare the influence of different timesteps.
print("===> timesteps: 1")
train(ifnode_spiking_neural_network_1_timesteps, "timesetps=1")

print("===> timesteps: 2")
train(ifnode_spiking_neural_network_2_timesteps, "timesteps=2")

print("===> timesteps: 5")
train(ifnode_spiking_neural_network_5_timesteps, "timesteps=5")

print("===> timesteps: 10")
train(ifnode_spiking_neural_network_10_timesteps, "timesteps=10")

plot()
```

```
===> timesteps: 1
Test Accuracy: 92.84%, Elapsed time: 0:01:00.376058
===> timesteps: 2
Test Accuracy: 95.23%, Elapsed time: 0:01:27.988771
===> timesteps: 5
Test Accuracy: 96.73%, Elapsed time: 0:02:39.402739
===> timesteps: 10
Test Accuracy: 96.38%, Elapsed time: 0:04:20.826122
```

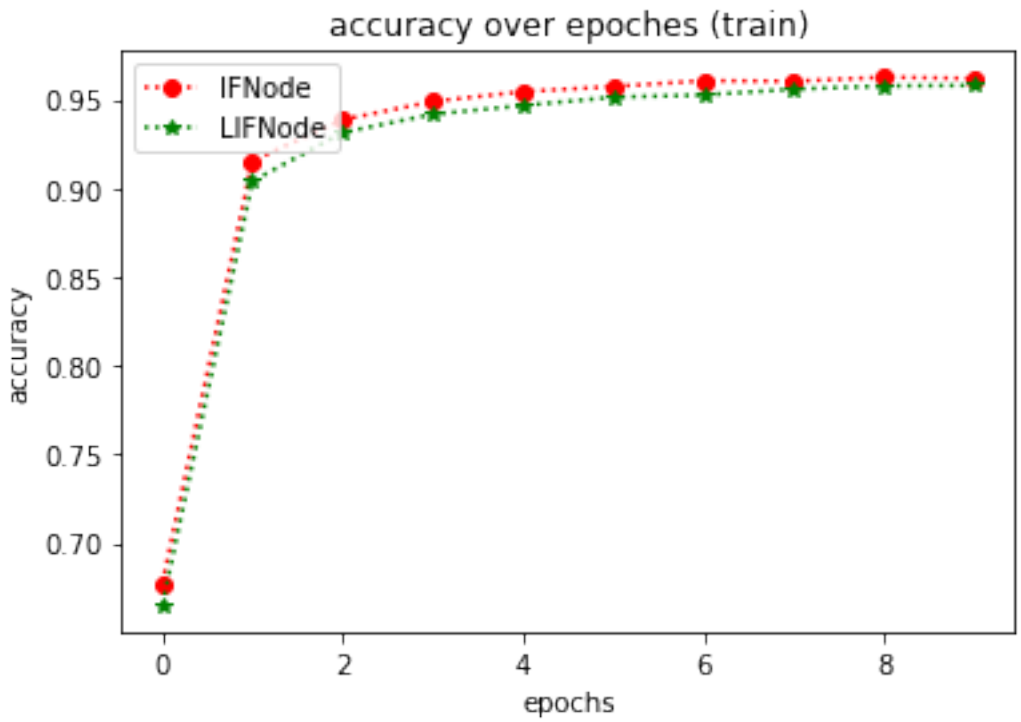


png

3.1.2 IFNode vs LIF Node

```
# reset y
y = {}

# define IFNode network
ifnode_spiking_neural_network = mnist(node="IFNode").spike()

# define LIFNode network
lifnode_spiking_neural_network = mnist(node="LIFNode", tau=1.0).spike()

# train and compare
print("===> IFNode")
train(ifnode_spiking_neural_network, "IFNode")

print("===> LIFNode, tau=1.0")
train(lifnode_spiking_neural_network, "LIFNode")

plot()
```

```
===> IFNode
Test Accuracy: 96.10%, Elapsed time: 0:02:40.809373
===> LIFNode, tau=1.0
Test Accuracy: 96.30%, Elapsed time: 0:02:54.205864
```



png

3.1.3 IFNode with different configurations

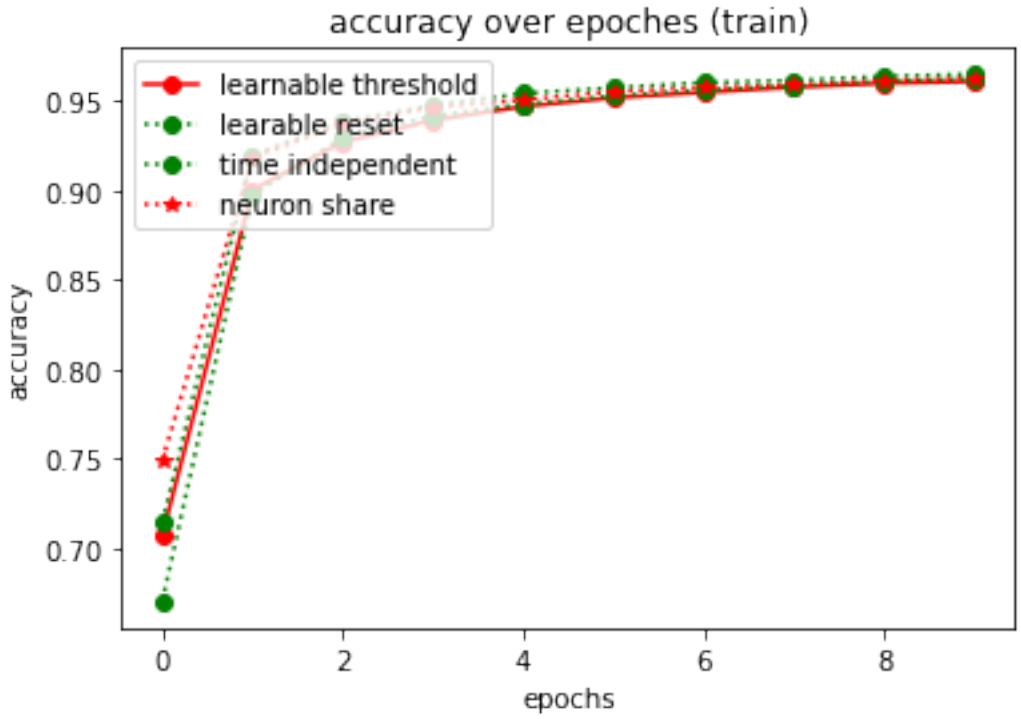
```
# reset y
y = {}

# define IFNode network with different configuration
ifnode_spiking_neural_network_learnable_threshold = mnist(learnable_threshold = True).
↪spike()
print("===> IFNode learnable threshold")
train(ifnode_spiking_neural_network_learnable_threshold, "learnable threshold")

ifnode_spiking_neural_network_learnable_reset = mnist(learnable_reset=True).spike()
print("===> IFNode learnable reset")
train(ifnode_spiking_neural_network_learnable_reset, "learable reset")

ifnode_spiking_neural_network_time_independent = mnist(time_dependent=False).spike()
print("===> IFNode time independent")
train(ifnode_spiking_neural_network_time_independent, "time independent")

ifnode_spiking_neural_network_neuron_share = mnist(neuron_wise=False).spike()
print("===> IFNode neuron share")
train(ifnode_spiking_neural_network_neuron_share, "neuron share")

plot()
```

```
===> IFNode learnable threshold
Test Accuracy: 95.34%, Elapsed time: 0:02:47.300100
```

(continues on next page)

(continued from previous page)

```
===> IFNode learnable reset
Test Accuracy: 88.60%, Elapsed time: 0:02:44.121851
===> IFNode time independent
Test Accuracy: 96.38%, Elapsed time: 0:02:36.499136
===> IFNode neuron share
Test Accuracy: 95.99%, Elapsed time: 0:02:41.603358
```



png

3.1.4 IFNode with different surrogate functions

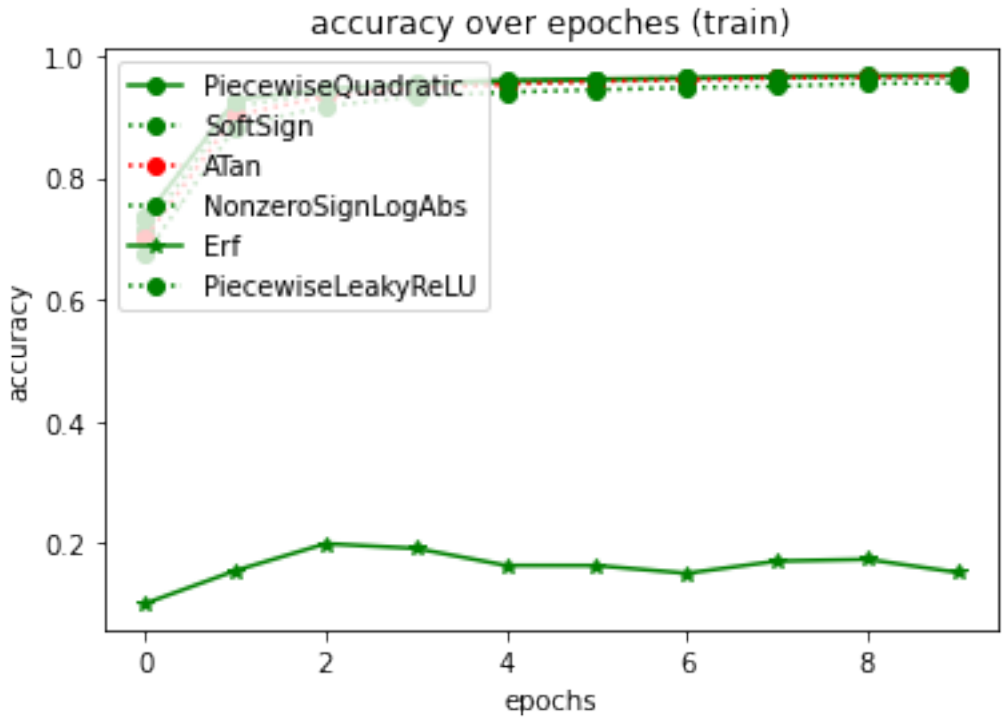
```
# reset y
y = {}

ifnode_spiking_neural_network_PiecewiseQuadratic = mnist(surrogate_fn="PiecewiseQuadratic
↪").spike()
print("===> IFNode PiecewiseQuadratic")
train(ifnode_spiking_neural_network_PiecewiseQuadratic, "PiecewiseQuadratic")

ifnode_spiking_neural_network_SoftSign = mnist(surrogate_fn="SoftSign").spike()
print("===> IFNode SoftSign")
train(ifnode_spiking_neural_network_SoftSign, "SoftSign")

ifnode_spiking_neural_network_ATan = mnist(surrogate_fn="ATan").spike()
print("===> IFNode ATan")
train(ifnode_spiking_neural_network_ATan, "ATan")

ifnode_spiking_neural_network_NonzeroSignLogAbs = mnist(surrogate_fn="NonzeroSignLogAbs
↪").spike()
```

(continues on next page)

(continued from previous page)

```
print("===> IFNode NonzeroSignLogAbs")
train(ifnode_spiking_neural_network_NonzeroSignLogAbs, "NonzeroSignLogAbs")

ifnode_spiking_neural_network_Erf = mnist(surrogate_fn="Erf").spike()
print("===> IFNode Erf")
train(ifnode_spiking_neural_network_Erf, "Erf")

ifnode_spiking_neural_network_PiecewiseLeakyReLU = mnist(surrogate_fn="PiecewiseLeakyReLU
↪").spike()
print("===> IFNode PiecewiseLeakyReLU")
train(ifnode_spiking_neural_network_PiecewiseLeakyReLU, "PiecewiseLeakyReLU")

plot()
```

```
===> IFNode PiecewiseQuadratic
Test Accuracy: 96.59%, Elapsed time: 0:02:50.522003
===> IFNode SoftSign
Test Accuracy: 96.74%, Elapsed time: 0:02:46.623716
===> IFNode ATan
Test Accuracy: 94.56%, Elapsed time: 0:02:44.426878
===> IFNode NonzeroSignLogAbs
Test Accuracy: 95.71%, Elapsed time: 0:02:39.817249
===> IFNode Erf
Test Accuracy: 26.70%, Elapsed time: 0:02:43.860484
===> IFNode PiecewiseLeakyReLU
Test Accuracy: 96.82%, Elapsed time: 0:02:47.646427
```

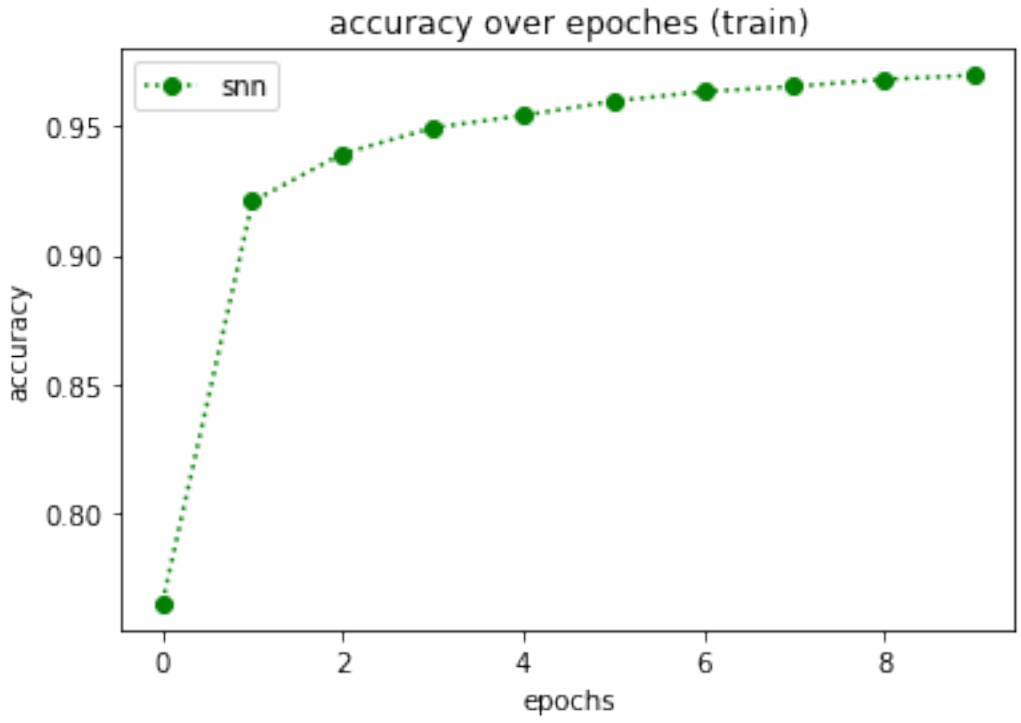


png

3.1.5 Hybrid

```
# reset y
y = {}

spiking_neural_network = mnist(
                  timesteps = 10,
                  node ="IFNode",
                  voltage_threshold = 1.0,
                  voltage_reset = 0.0,
                  learnable_threshold = False,
                  learnable_reset = True,
                  time_dependent = True,
                  neuron_wise = True,
                  surrogate_fn = "SoftSign",
                  binary = True,).spike()
print("===> Hybrid Spiking Neural Network")
train(spiking_neural_network)

plot()
```

```
===> Hybrid Spiking Neural Network
Test Accuracy: 96.68%, Elapsed time: 0:04:11.330691
```



png

3.2 Quantization Neural Network with Eve

In this script, we will build a quantization neural network with eve-mli, different kinds of quantization function will be compared.

```
# import necessary packages.
# at the beginning, ensure that the eve-mli package is in your python path.
# or you just install it via `pip install eve-mli`.

import os
import time
from datetime import datetime

import random
import numpy as np
import torch as th
import torch.nn as nn
import torch.nn.functional as F

import eve
import eve.app
import eve.app.model
import eve.app.trainer
import eve.core
import eve.app.space as space
import eve.core.layer
import eve.core.quan

from matplotlib import pyplot as plt
%matplotlib inline

os.environ["CUDA_VISIBLE_DEVICES"] = '1'
```

```
# build a basic network for trainer, use Poisson Encoder as default
class mnist(eve.core.Eve):
    def __init__(self,
            quan_on_a: bool = True,
            quan_on_w: bool = True,
            bits: int = 8,
            quantize_fn: str = "Round",
            range_tracker: str = "average_tracker",
            average_tracker_momentum: float = 0.1,
            upgrade_bits: bool = False,
            neuron_wise: bool = False,
            asymmetric: bool = False,
            signed_quantization: bool = False,
            learnable_alpha: bool = None,
            upgrade_fn: callable = None,
            **kwargs,):
        super().__init__()

        def build_quantizer(state):
```

(continues on next page)

(continued from previous page)

```
            return eve.core.quan.Quantizer(state,
                        bits = bits,
                        quantize_fn = quantize_fn,
                        range_tracker = range_tracker,
                        average_tracker_momentum = average_tracker_momentum,
                        upgrade_bits = upgrade_bits,
                        neuron_wise = neuron_wise,
                        asymmetric = asymmetric,
                        signed_quantization = signed_quantization,
                        learnable_alpha = learnable_alpha,
                        upgrade_fn = upgrade_fn,
                        **kwargs,)
        if quan_on_w:
            self.conv1 = eve.core.layer.QuanBNFuseConv2d(1, 4, 3, stride=2, padding=1)
            self.conv1.assign_quantizer(
                build_quantizer(eve.core.State(self.conv1, apply_on="param")))
        else:
            self.conv1 = nn.Sequential(nn.Conv2d(1, 4, 3, stride=2, padding=1), nn.
↪BatchNorm2d(4))
        self.relu1 = nn.ReLU()
        if quan_on_a:
            self.act_quan1 = build_quantizer(eve.core.State(self.conv1, apply_on="data"))
        else:
            self.act_quan1 = nn.Sequential()

        if quan_on_w:
            self.conv2 = eve.core.layer.QuanBNFuseConv2d(4, 8, 3, stride=2, padding=1)
            self.conv2.assign_quantizer(
                build_quantizer(eve.core.State(self.conv2, apply_on="param")))
        else:
            self.conv2 = nn.Sequential(nn.Conv2d(4, 8, 3, stride=2, padding=1), nn.
↪BatchNorm2d(8))
        self.relu2 = nn.ReLU()
        if quan_on_a:
            self.act_quan2 = build_quantizer(eve.core.State(self.conv2, apply_on="data"))
        else:
            self.act_quan2 = nn.Sequential()

        if quan_on_w:
            self.conv3 = eve.core.layer.QuanBNFuseConv2d(8, 16, 3, stride=2, padding=1)
            self.conv3.assign_quantizer(
                build_quantizer(eve.core.State(self.conv3, apply_on="param")))
        else:
            self.conv3 = nn.Sequential(nn.Conv2d(8, 16, 3, stride=2, padding=1), nn.
↪BatchNorm2d(16))
        self.relu3 = nn.ReLU()
        if quan_on_a:
            self.act_quan3 = build_quantizer(eve.core.State(self.conv3, apply_on="data"))
        else:
            self.act_quan3 = nn.Sequential()

        if quan_on_w:
```

(continues on next page)

(continued from previous page)

```
            self.linear1 = eve.core.layer.QuanLinear(16 * 4 * 4, 16)
            self.linear1.assign_quantizer(
                build_quantizer(eve.core.State(self.linear1, apply_on="param")))
        else:
            self.linear1 = nn.Linear(16 * 4 * 4, 16)
        self.relu4 = nn.ReLU()
        if quan_on_a:
            self.act_quan4 = build_quantizer(eve.core.State(self.linear1, apply_on="data
↪"))
        else:
            self.act_quan4 = nn.Sequential()

        self.linear2 = nn.Linear(16, 10)

    def forward(self, x):
        conv1 = self.conv1(x)
        relu1 = self.relu1(conv1)
        act_quan1 = self.act_quan1(relu1)

        conv2 = self.conv2(act_quan1)
        relu2 = self.relu2(conv2)
        act_quan2 = self.act_quan2(relu2)

        conv3 = self.conv3(act_quan2)
        relu3 = self.relu3(conv3)
        act_quan3 = self.act_quan3(relu3)

        act_quan3 = th.flatten(act_quan3, start_dim=1).unsqueeze(dim=1)

        linear1 = self.linear1(act_quan3)
        relu4 = self.relu4(linear1)
        act_quan4 = self.act_quan4(relu4)

        linear2 = self.linear2(act_quan4)

        return linear2.squeeze(dim=1)
```

```
# define a MnistClassifier
# Classifier uses the corss entropy as default.
# in most case, we just rewrite the `prepare_data`.
class MnistClassifier(eve.app.model.Classifier):
    def prepare_data(self, data_root: str):
        from torch.utils.data import DataLoader, random_split
        from torchvision import transforms
        from torchvision.datasets import MNIST

        train_dataset = MNIST(root=data_root,
                              train=True,
                              download=True,
                              transform=transforms.ToTensor())
        test_dataset = MNIST(root=data_root,
                             train=False,
```

(continues on next page)

(continued from previous page)

```
                              download=True,
                              transform=transforms.ToTensor())
        self.train_dataset, self.valid_dataset = random_split(
            train_dataset, [55000, 5000])
        self.test_dataset = test_dataset

        self.train_dataloader = DataLoader(self.train_dataset,
                                           batch_size=128,
                                           shuffle=True,
                                           num_workers=4)
        self.test_dataloader = DataLoader(self.test_dataset,
                                          batch_size=128,
                                          shuffle=False,
                                          num_workers=4)
        self.valid_dataloader = DataLoader(self.valid_dataset,
                                           batch_size=128,
                                           shuffle=False,
                                           num_workers=4)
```

```
# store accuracy result
y = {}
def plot():
    global y
    keys, values = list(y.keys()), list(y.values())
    for k, v in y.items():
        plt.plot(v,
                 color='green' if random.random() > 0.5 else "red",
                 marker='o' if random.random() > 0.5 else "*",
                 linestyle='-' if random.random() > 0.5 else ":",
                 label=k)
    plt.title('accuracy over epoches (train)')
    plt.xlabel('epochs')
    plt.ylabel('accuracy')
    plt.legend(loc="upper left")
    plt.show()
```

```
def train(net, exp_name: str = "quan", data_root: str = "/home/densechen/dataset"):
    global y
    # replace the data_root for your path.
    classifier = MnistClassifier(net)
    classifier.prepare_data(data_root=data_root)

    # use default configuration
    # use a smaller lr for that alpha is unstable during training
    classifier.setup_train(lr=1e-4)

    # assign model to trainer
    eve.app.trainer.BaseTrainer.assign_model(classifier)

    trainer = eve.app.trainer.BaseTrainer()
```

(continues on next page)

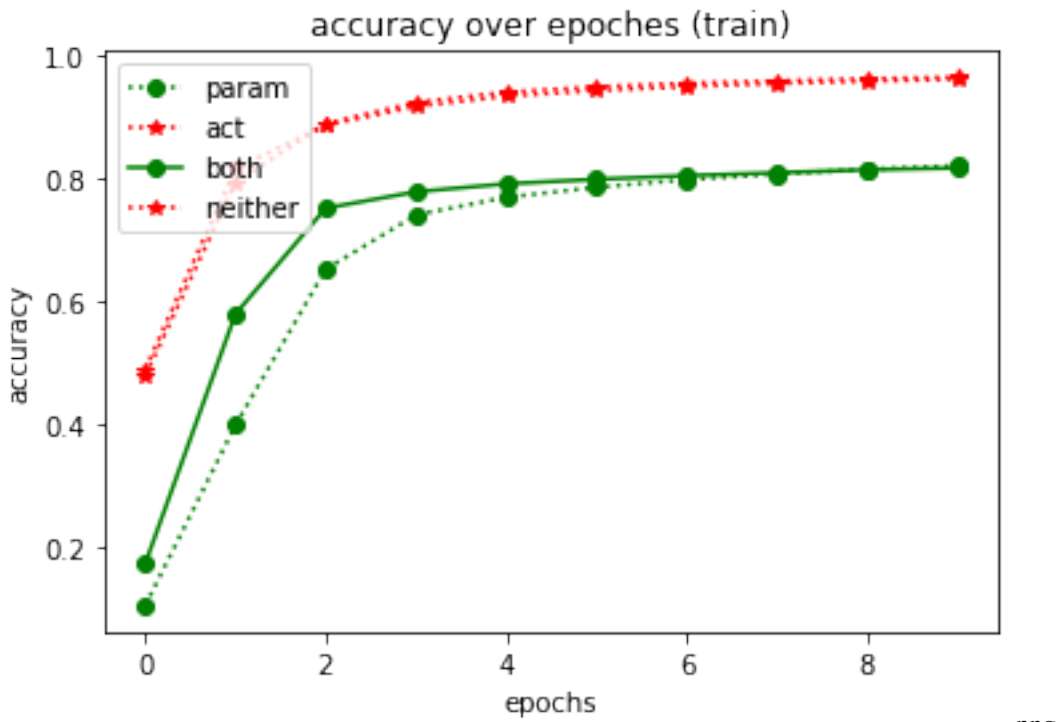
(continued from previous page)

```
    # train 10 epoches and report the final accuracy
    y[exp_name] = []
    tic = datetime.now()
    for _ in range(10):
        info = trainer.fit()
        y[exp_name].append(info["acc"])
    info = trainer.test()
    toc = datetime.now()
    y[exp_name] = np.array(y[exp_name])
    print(f"Test Accuracy: {info['acc']*100:.2f}%, Elapsed time: {toc-tic}")
```

3.2.1 Quan param vs Quan act

```
# reset y
y = {}

# define quantization neural network with quantize param, quantize act and quantize on␣
↪both
quantization_neural_network_param = mnist(quan_on_w=True, quan_on_a=False).quantize()
quantization_neural_network_act = mnist(quan_on_w=False, quan_on_a=True).quantize()
quantization_neural_network_both = mnist(quan_on_w=True, quan_on_a=True).quantize()
quantization_neural_network_neither = mnist(quan_on_w=False, quan_on_a=False).quantize()
# or
# quantization_neural_network_neither = mnist().non_quanitze()

print("===> Quantization on param")
train(quantization_neural_network_param, "param")

print("===> Quantization on act")
train(quantization_neural_network_act, "act")

print("===> Quantization on both")
train(quantization_neural_network_both, "both")

print("===> Quantization on neither")
train(quantization_neural_network_neither, "neither")

plot()
```

```
===> Quantization on param
Test Accuracy: 83.58%, Elapsed time: 0:01:41.564670
===> Quantization on act
Test Accuracy: 96.52%, Elapsed time: 0:01:15.718537
===> Quantization on both
Test Accuracy: 83.30%, Elapsed time: 0:02:08.939456
===> Quantization on neither
Test Accuracy: 96.59%, Elapsed time: 0:00:46.188178
```

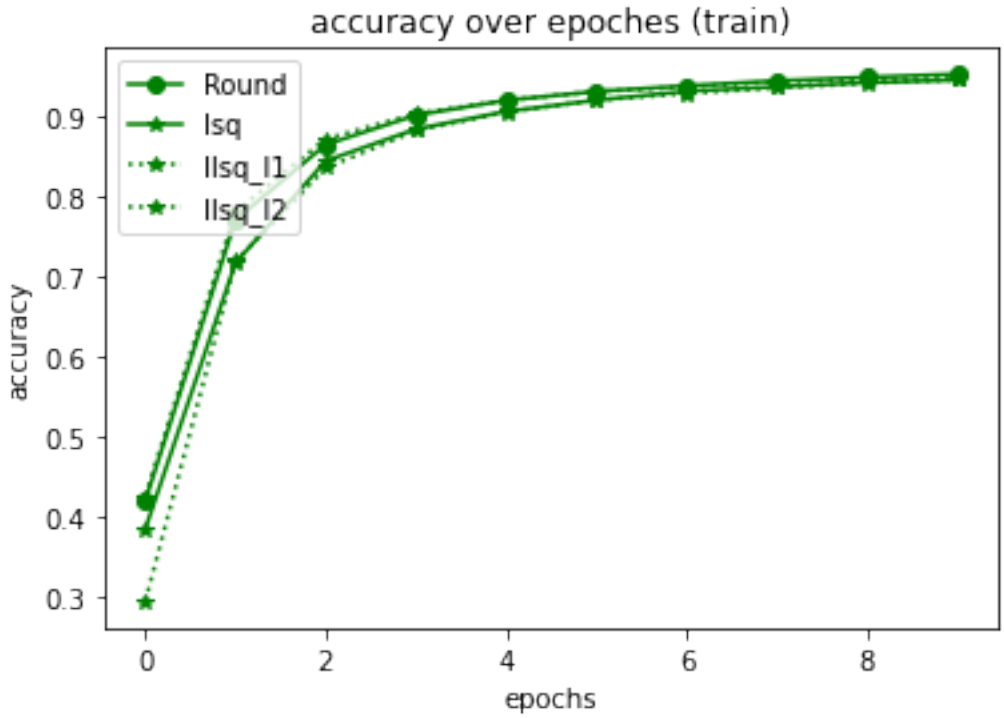


png

3.2.2 Round vs LSQ vs LLSQ on act only

```
# reset y
y = {}

# define quantization neural network with different quantization function
quantization_neural_network_round = mnist(quan_on_w=False, quantize_fn="Round").
↪quantize()
quantization_neural_network_lsq = mnist(quan_on_w=False, quantize_fn="Lsq").quantize()
quantization_neural_network_llsq_l1 = mnist(quan_on_w=False, quantize_fn="Llsq", regular=
↪"l1").quantize()
quantization_neural_network_llsq_l2 = mnist(quan_on_w=False, quantize_fn="Llsq", regular=
↪"l2").quantize()

print("===> Quantization with Round")
train(quantization_neural_network_round, "Round")

print("===> Quantization with lsq")
train(quantization_neural_network_lsq, "lsq")

print("===> Quantization with llsq_l1")
train(quantization_neural_network_llsq_l1, "llsq_l1")

print("===> Quantization with llsq_l2")
train(quantization_neural_network_llsq_l2, "llsq_l2")

plot()
```

```
===> Quantization with Round
Test Accuracy: 95.72%, Elapsed time: 0:01:14.043124
===> Quantization with lsq
Test Accuracy: 94.90%, Elapsed time: 0:01:24.896148
===> Quantization with llsq_l1
Test Accuracy: 95.08%, Elapsed time: 0:02:05.949373
===> Quantization with llsq_l2
Test Accuracy: 95.35%, Elapsed time: 0:02:11.568797
```

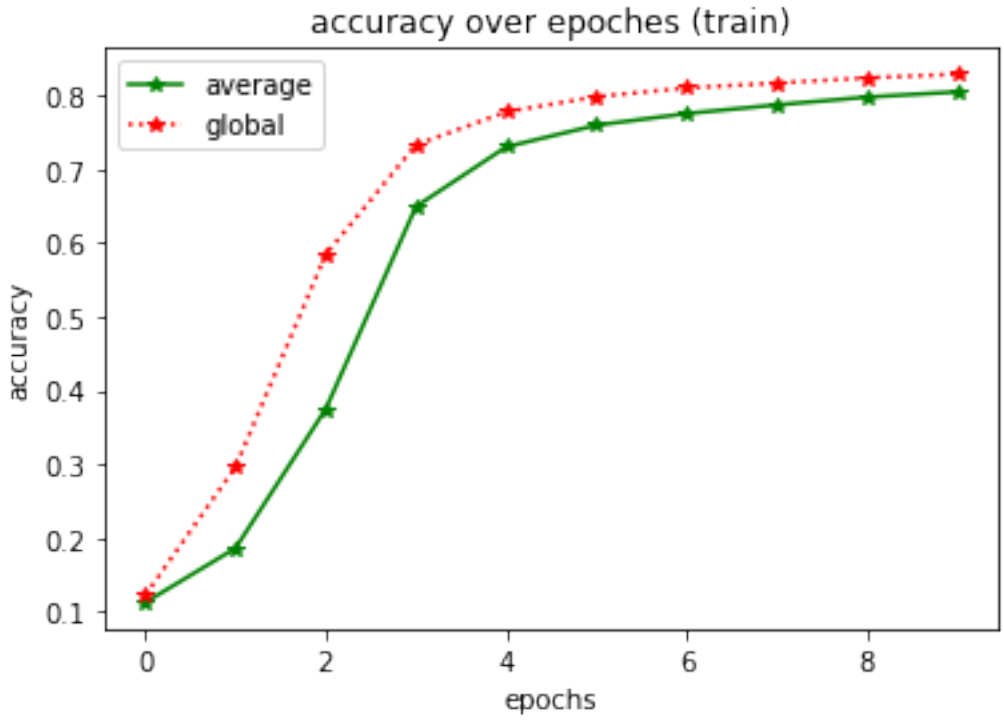


png

3.2.3 Average tracker vs Global tracker

```
# reset y
y = {}

# define quantization neural network with different quantization function
quantization_neural_network_average_tracker = mnist(range_tracker="average_tracker").
↪quantize()
quantization_neural_network_global_tracker = mnist(range_tracker="global_tracker").
↪quantize()

print("===> Quantization with average range tracker")
train(quantization_neural_network_average_tracker, "average")

print("===> Quantization with global range tracker")
train(quantization_neural_network_global_tracker, "global")

plot()
```

```
===> Quantization with average range tracker
Test Accuracy: 82.03%, Elapsed time: 0:02:10.593734
===> Quantization with global range tracker
Test Accuracy: 84.06%, Elapsed time: 0:02:12.500429
```



png

3.2.4 Round with with different bits¶

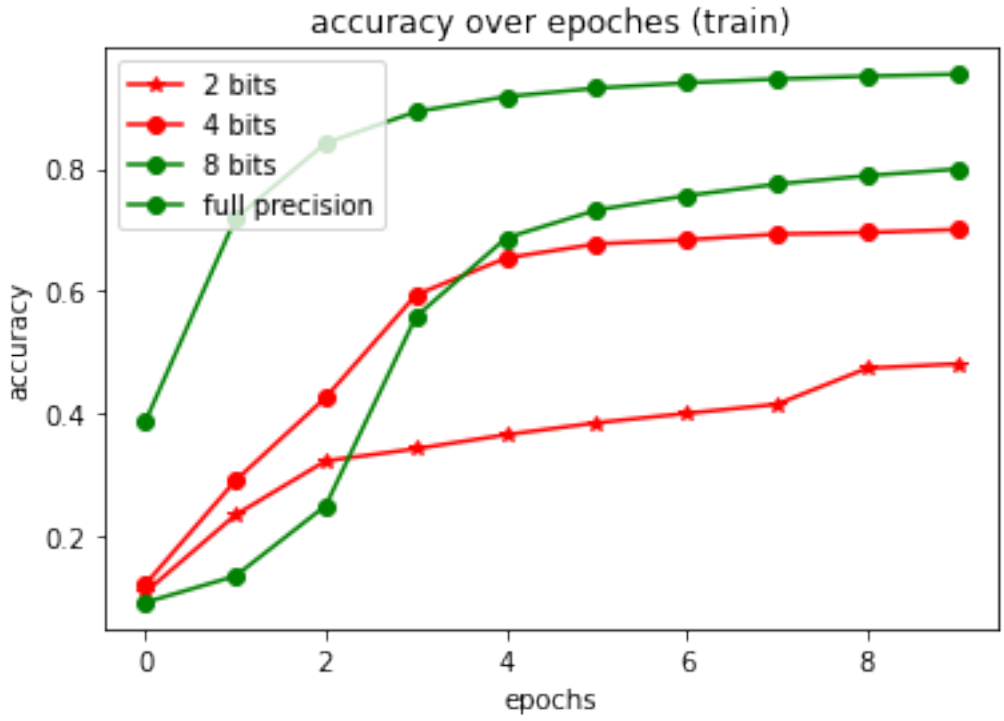
```
# reset y
y = {}

# define quantization neural network with different configuration
quantization_neural_network_bits_2 = mnist(bits=2).quantize()
quantization_neural_network_bits_4 = mnist(bits=4).quantize()
quantization_neural_network_bits_8 = mnist(bits=8).quantize()
quantization_neural_network_full_precision = mnist().non_quantize()

print("===> Quantization with 2 bits")
train(quantization_neural_network_bits_2, "2 bits")

print("===> Quantization with 4 bits")
train(quantization_neural_network_bits_4, "4 bits")

print("===> Quantization with 8 bits")
train(quantization_neural_network_bits_8, "8 bits")

print("===> Quantization with full precision")
train(quantization_neural_network_full_precision, "full precision")
```

(continues on next page)

(continued from previous page)

```
plot()
```

```
===> Quantization with 2 bits
Test Accuracy: 45.51%, Elapsed time: 0:02:10.259819
===> Quantization with 4 bits
Test Accuracy: 71.16%, Elapsed time: 0:02:08.941098
===> Quantization with 8 bits
Test Accuracy: 81.84%, Elapsed time: 0:02:07.281995
===> Quantization with full precision
Test Accuracy: 96.11%, Elapsed time: 0:01:40.056623
```



png

3.2.5 quan with other arguments

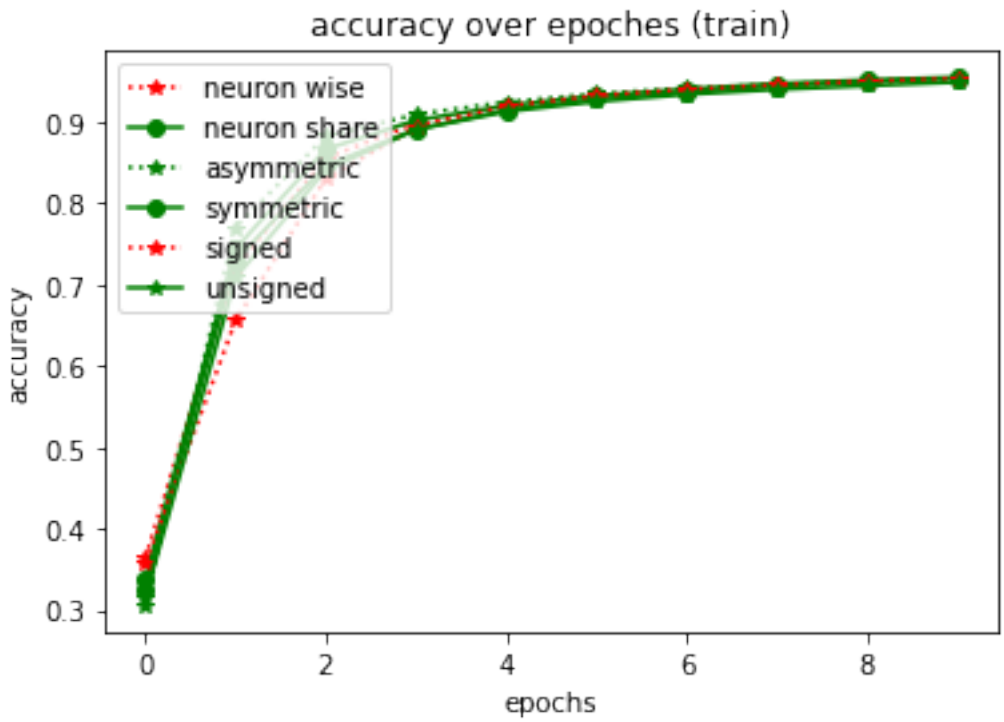
```
# reset y
y = {}

# define quantization neural network
quantization_neural_network_neuron_wise = mnist(neuron_wise=True)
quantization_neural_network_neuron_share = mnist(neuron_wise=False)

quantization_neural_network_asymmetric = mnist(asymmetric=True)
quantization_neural_network_symmetric = mnist(asymmetric=False)

quantization_neural_network_signed_quantization = mnist(signed_quantization=True)
quantization_neural_network_unsigned_quantization = mnist(signed_quantization=False)
```

(continues on next page)

(continued from previous page)

```
print("===> Quantization with neuron wise")
train(quantization_neural_network_neuron_wise, "neuron wise")

print("===> Quantization with neuron share")
train(quantization_neural_network_neuron_share, "neuron share")

print("===> Quantization with asymmetric")
train(quantization_neural_network_asymmetric, "asymmetric")

print("===> Quantization with symmetric")
train(quantization_neural_network_symmetric, "symmetric")

print("===> Quantization with signed quantization")
train(quantization_neural_network_signed_quantization, "signed")

print("===> Quantization with unsigned quantization")
train(quantization_neural_network_unsigned_quantization, "unsigned")

plot()
```

```
===> Quantization with neuron wise
Test Accuracy: 95.60%, Elapsed time: 0:01:39.704981
===> Quantization with neuron share
Test Accuracy: 95.79%, Elapsed time: 0:01:48.575900
===> Quantization with asymmetric
Test Accuracy: 95.84%, Elapsed time: 0:01:32.690768
===> Quantization with symmetric
Test Accuracy: 96.14%, Elapsed time: 0:01:42.991659
===> Quantization with signed quantization
Test Accuracy: 95.88%, Elapsed time: 0:01:51.287698
===> Quantization with unsigned quantization
Test Accuracy: 95.60%, Elapsed time: 0:01:36.948094
```



png

3.3 Network Pruning

In eve-mli, you can perform a pruning operation on network in a lightly way under the help of eve parameter.

```
# import necessary packages.
# at the beginning, ensure that the eve-mli package is in your python path.
# or you just install it via `pip install eve-mli`.

import os
import time
from datetime import datetime

import random
import numpy as np
import torch as th
import torch.nn as nn
import torch.nn.functional as F

import eve
import eve.app
import eve.app.model
import eve.app.trainer
import eve.core
import eve.core.layer
```

(continues on next page)

(continued from previous page)

```
from matplotlib import pyplot as plt
%matplotlib inline

os.environ["CUDA_VISIBLE_DEVICES"] = '1'
```

```
# build a basic network for trainer
class mnist(eve.core.Eve):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 4, 3, stride=2, padding=1),
            nn.BatchNorm2d(4),
        )
        self.pruner1 = eve.core.Pruner(eve.core.State(self.conv1))

        self.conv2 = nn.Sequential(
            nn.Conv2d(4, 8, 3, stride=2, padding=1),
            nn.BatchNorm2d(8),
        )
        self.pruner2 = eve.core.Pruner(eve.core.State(self.conv2))

        self.conv3 = nn.Sequential(
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
        )
        self.pruner3 = eve.core.Pruner(eve.core.State(self.conv3))

        self.linear1 = nn.Linear(16 * 4 * 4, 16)
        self.pruner4 = eve.core.Pruner(eve.core.State(self.linear1))

        self.linear2 = nn.Linear(16, 10)

    def forward(self, x):
        conv1 = self.conv1(x)
        pruner1 = self.pruner1(conv1)

        conv2 = self.conv2(pruner1)
        pruner2 = self.pruner2(conv2)

        conv3 = self.conv3(pruner2)
        pruner3 = self.pruner3(conv3)

        pruner3 = th.flatten(pruner3, start_dim=1).unsqueeze(dim=1)

        linear1 = self.linear1(pruner3)
        pruner4 = self.pruner4(linear1)

        linear2 = self.linear2(pruner4)

        return linear2.squeeze(dim=1)
```

```
# define a MnistClassifier
# Classifier uses the corss entropy as default.
# in most case, we just rewrite the `prepare_data`.
class MnistClassifier(eve.app.model.Classifier):
    def prepare_data(self, data_root: str):
        from torch.utils.data import DataLoader, random_split
        from torchvision import transforms
        from torchvision.datasets import MNIST

        train_dataset = MNIST(root=data_root,
                              train=True,
                              download=True,
                              transform=transforms.ToTensor())
        test_dataset = MNIST(root=data_root,
                             train=False,
                             download=True,
                             transform=transforms.ToTensor())
        self.train_dataset, self.valid_dataset = random_split(
            train_dataset, [55000, 5000])
        self.test_dataset = test_dataset

        self.train_dataloader = DataLoader(self.train_dataset,
                                           batch_size=128,
                                           shuffle=True,
                                           num_workers=4)
        self.test_dataloader = DataLoader(self.test_dataset,
                                          batch_size=128,
                                          shuffle=False,
                                          num_workers=4)
        self.valid_dataloader = DataLoader(self.valid_dataset,
                                           batch_size=128,
                                           shuffle=False,
                                           num_workers=4)
```

```
# store accuracy result
y = {}
def plot():
    global y
    keys, values = list(y.keys()), list(y.values())
    for k, v in y.items():
        plt.plot(v,
                 color='green' if random.random() > 0.5 else "red",
                 marker='o' if random.random() > 0.5 else "*",
                 linestyle='-' if random.random() > 0.5 else ":",
                 label=k)
    plt.title('accuracy over epoches (train)')
    plt.xlabel('epochs')
    plt.ylabel('accuracy')
    plt.legend(loc="upper left")
    plt.show()
```

```
def train(trainer, exp_name: str = "snn"):
```

(continues on next page)

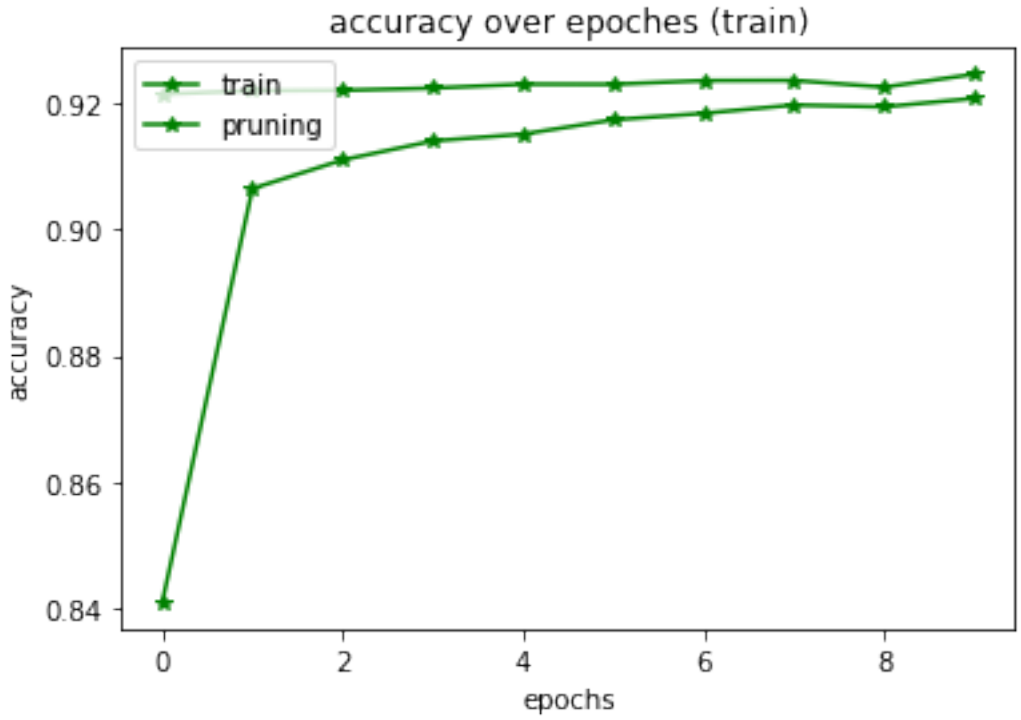
(continued from previous page)

```
    global y
    # train 10 epoches and report the final accuracy
    y[exp_name] = []
    tic = datetime.now()
    for _ in range(10):
        info = trainer.fit()
        y[exp_name].append(info["acc"])
    info = trainer.test()
    toc = datetime.now()
    y[exp_name] = np.array(y[exp_name])
    print(f"Test Accuracy: {info['acc']*100:.2f}%, Elapsed time: {toc-tic}")
```

3.3.1 l1 norm

```
# reset result
y = {}

# Don't forget to reset global statistic, otherwise may cause cuda error
eve.core.State.reset_global_statistic()

# register the global pruning function
eve.core.State.register_global_statistic("l1_norm")

# define net
pruning_neural_network_l1_norm = mnist()

# replace the data_root for your path.
classifier = MnistClassifier(pruning_neural_network_l1_norm)
classifier.prepare_data(data_root= "/home/densechen/dataset")

# use default configuration
classifier.setup_train()

# assign model to trainer
eve.app.trainer.BaseTrainer.assign_model(classifier)

trainer = eve.app.trainer.BaseTrainer()

# Train it
print("===> Train")
train(trainer, "train")

print("===> Pruning")
# use upgrader to do pruning automatically
upgrader = eve.app.upgrader.Upgrader(pruning_neural_network_l1_norm.eve_parameters())
upgrader.step()

train(trainer, "pruning")

plot()
```

```
===> Train
Test Accuracy: 92.38%, Elapsed time: 0:00:51.982259
===> Pruning
Test Accuracy: 92.35%, Elapsed time: 0:00:52.857012
```



png

3.3.2 fire rate

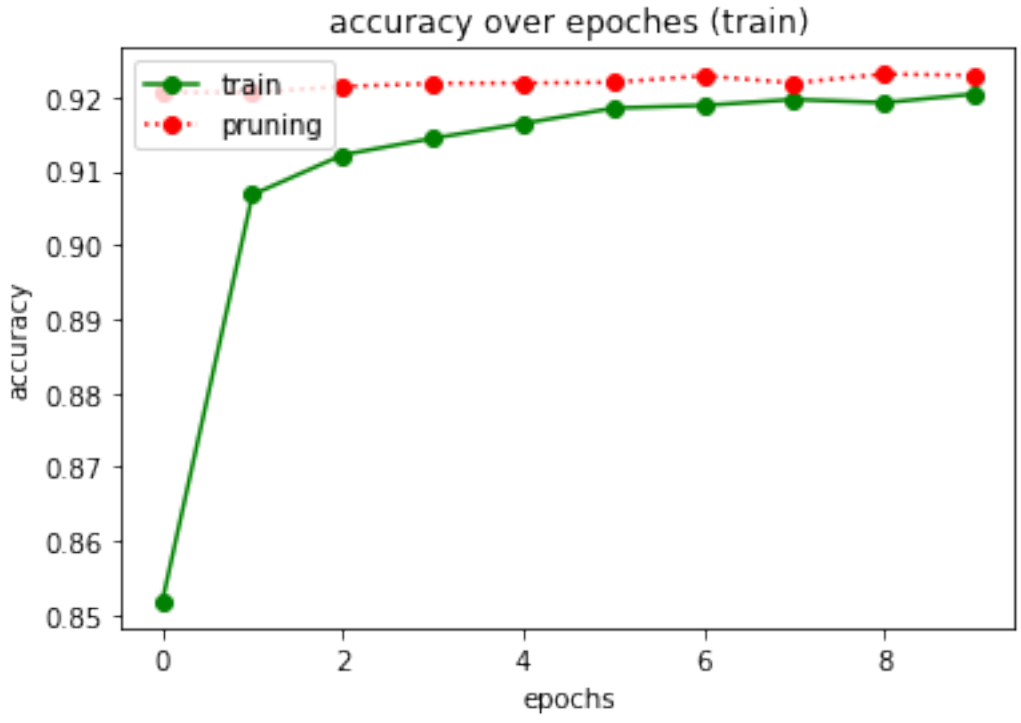
```
# reset result
y = {}

# Don't forget to reset global statistic, otherwise may cause cuda error
eve.core.State.reset_global_statistic()

# register the global pruning function
eve.core.State.register_global_statistic("fire_rate")

# define net
pruning_neural_network_fire_rate = mnist()

# replace the data_root for your path.
classifier = MnistClassifier(pruning_neural_network_fire_rate)
classifier.prepare_data(data_root="/home/densechen/dataset")

# use default configuration
classifier.setup_train()

# assign model to trainer
```

(continues on next page)

(continued from previous page)

```
eve.app.trainer.BaseTrainer.assign_model(classifier)

trainer = eve.app.trainer.BaseTrainer()

# Train it
print("===> Train")
train(trainer, "train")

print("===> Pruning")
# use upgrader to do pruning automatically
upgrader = eve.app.upgrader.Upgrader(pruning_neural_network_fire_rate.eve_parameters())
upgrader.step()

train(trainer, "pruning")

plot()
```

```
===> Train
Test Accuracy: 92.15%, Elapsed time: 0:00:53.528946
===> Pruning
Test Accuracy: 92.40%, Elapsed time: 0:00:53.454817
```



png

3.4 Network Architecture Searching with Eve

```
import os

import numpy as np
import torch as th
import torch.nn as nn
import torch.nn.functional as F

import eve
import eve.app
import eve.app.model
import eve.app.trainer
import eve.core
import eve.app.space as space

os.environ["CUDA_VISIBLE_DEVICES"] = '1'
```

```
# build a basic network for trainer
class mnist(eve.core.Eve):
    def __init__(self, neuron_wise: bool = False):
        super().__init__()

        eve.core.State.register_global_statistic("l1_norm")
        eve.core.State.register_global_statistic("kl_div")

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 4, 3, stride=2, padding=1),
            nn.BatchNorm2d(4),
        )
        # use IFNode to act as ReLU
        self.node1 = eve.core.IFNode(eve.core.State(self.conv1), binary=False)
        self.quan1 = eve.core.Quantizer(eve.core.State(self.conv1),
                                        upgrade_bits=True,
                                        neuron_wise=neuron_wise,)

        self.conv2 = nn.Sequential(
            nn.Conv2d(4, 8, 3, stride=2, padding=1),
            nn.BatchNorm2d(8),
        )
        self.node2 = eve.core.IFNode(eve.core.State(self.conv2), binary=False)
        self.quan2 = eve.core.Quantizer(eve.core.State(self.conv2),
                                        upgrade_bits=True,
                                        neuron_wise=neuron_wise,)

        self.conv3 = nn.Sequential(
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
        )
        self.node3 = eve.core.IFNode(eve.core.State(self.conv3), binary=False)
        self.quan3 = eve.core.Quantizer(eve.core.State(self.conv3),
                                        upgrade_bits=True,
```

(continues on next page)

(continued from previous page)

```
                                        neuron_wise=neuron_wise,)

        self.linear1 = nn.Linear(16 * 4 * 4, 16)
        self.node4 = eve.core.IFNode(eve.core.State(self.linear1))
        self.quan4 = eve.core.Quantizer(eve.core.State(self.linear1),
                                        upgrade_bits=True,
                                        neuron_wise=neuron_wise,)

        self.linear2 = nn.Linear(16, 10)

    def forward(self, x):
        conv1 = self.conv1(x)
        node1 = self.node1(conv1)
        quan1 = self.quan1(node1)

        conv2 = self.conv2(quan1)
        node2 = self.node2(conv2)
        quan2 = self.quan2(node2)

        conv3 = self.conv3(quan2)
        node3 = self.node3(conv3)
        quan3 = self.quan3(node3)

        quan3 = th.flatten(quan3, start_dim=1).unsqueeze(dim=1)

        linear1 = self.linear1(quan3)
        node4 = self.node4(linear1)
        quan4 = self.quan4(node4)

        linear2 = self.linear2(quan4)

        return linear2.squeeze(dim=1)
```

```
class MnistClassifier(eve.app.model.Classifier):
    def prepare_data(self, data_root: str):
        from torch.utils.data import DataLoader, random_split
        from torchvision import transforms
        from torchvision.datasets import MNIST

        train_dataset = MNIST(root=data_root,
                              train=True,
                              download=True,
                              transform=transforms.ToTensor())
        test_dataset = MNIST(root=data_root,
                             train=False,
                             download=True,
                             transform=transforms.ToTensor())
        self.train_dataset, self.valid_dataset = random_split(
            train_dataset, [55000, 5000])
        self.test_dataset = test_dataset

        self.train_dataloader = DataLoader(self.train_dataset,
```

(continues on next page)

(continued from previous page)

```
                                        batch_size=128,
                                        shuffle=True,
                                        num_workers=4)
        self.test_dataloader = DataLoader(self.test_dataset,
                                        batch_size=128,
                                        shuffle=False,
                                        num_workers=4)
        self.valid_dataloader = DataLoader(self.valid_dataset,
                                        batch_size=128,
                                        shuffle=False,
                                        num_workers=4)
```

```
class MnistTrainer(eve.app.trainer.BaseTrainer):
    def reset(self) -> np.ndarray:
        """Evaluate current trainer, reload trainer and then return the initial obs.

        Returns:
            obs: np.ndarray, the initial observation of trainer.
        """
        # do a fast valid
        self.steps += 1
        if self.steps % self.eval_steps == 0:
            self.steps = 0
            finetune_acc = self.valid()["acc"]
            # eval model
            if finetune_acc > self.finetune_acc:
                self.finetune_acc = finetune_acc
            # reset model to explore more posibility
            self.load_from_RAM()

        # save best model which achieve higher reward
        if self.accumulate_reward > self.best_reward:
            self.cache_to_RAM()
            self.best_reward = self.accumulate_reward

        # clear accumulate reward
        self.accumulate_reward = 0.0

        # reset related last value
        # WRAN: don't forget to reset self._obs_gen and self._last_eve_obs to None.
        # somtimes, the episode may be interrupted, but the gen do not reset.
        self.last_eve = None
        self.obs_generator = None
        self.upgrader.zero_obs()
        self.fit_step()
        return self.fetch_obs()

    def close(self):
        """Override close in your subclass to perform any necessary cleanup.

        Environments will automatically close() themselves when
        garbage collected or when the program exits.
```

(continues on next page)

(continued from previous page)

```
        """
        # load best model first
        self.load_from_RAM()

        finetune_acc = self.test()["acc"]

        bits = 0
        bound = 0
        for v in self.upgrader.eve_parameters():
            bits = bits + th.floor(v.mean() * 8)
            bound = bound + 8

        bits = bits.item()

        print(
            f"baseline: {self.baseline_acc}, ours: {finetune_acc}, bits: {bits} / {bound}
↪"
        )

        if self.tensorboard_log is not None:
            save_path = self.kwargs.get(
                "save_path", os.path.join(self.tensorboard_log, "model.ckpt"))
            self.save_checkpoint(path=save_path)
            print(f"save trained model to {save_path}")

    def reward(self) -> float:
        """A simple reward function.

        You have to rewrite this function based on your tasks.
        """
        self.upgrader.zero_obs()

        info = self.fit_step()

        return info["acc"] - self.last_eve.mean().item() * 0.4
```

```
# define a mnist classifier
neuron_wise = True
sample_episode = False

mnist_classifier = MnistClassifier(mnist(neuron_wise))
mnist_classifier.prepare_data(data_root="/home/densechen/dataset")
mnist_classifier.setup_train()  # use default configuration

# set mnist classifier to quantization mode
mnist_classifier.quantize()

# set neurons and states
# if neuron wise, we just set neurons as the member of max neurons of the network
# else set it to 1.
mnist_classifier.set_neurons(16 if neuron_wise else 1)
mnist_classifier.set_states(1)
```

(continues on next page)

(continued from previous page)

```
# None will use a default case
mnist_classifier.set_action_space(None)
mnist_classifier.set_observation_space(None)

# define a trainer
MnistTrainer.assign_model(mnist_classifier)

# define a experiment manager

exp_manager = eve.app.ExperimentManager(
    algo="ddpg",
    env_id="mnist_trainer",
    env=MnistTrainer,
    log_folder="examples/logs",
    n_timesteps=100000,
    save_freq=1000,
    default_hyperparameter_yaml="hyperparams",
    log_interval=100,
    sample_episode=sample_episode,
)

model = exp_manager.setup_experiment()

exp_manager.learn(model)
exp_manager.save_trained_model(model)
```

```
OrderedDict([('buffer_size', 2000),
             ('gamma', 0.98),
             ('gradient_steps', -1),
             ('learning_rate', 0.001),
             ('learning_starts', 1000),
             ('n_episodes_rollout', 1),
             ('n_timesteps', 1000000.0),
             ('noise_std', 0.1),
             ('noise_type', 'normal'),
             ('policy', 'MlpPolicy'),
             ('policy_kwargs', 'dict(net_arch=[400, 300])')])
Using 1 environments
Overwriting n_timesteps with n=100000
Applying normal noise with std 0.1
Using cuda device
Log path: examples/logs/ddpg/mnist_trainer_2
---------------------------------
| rollout/           |          |
|    ep_len_mean     | 4        |
|    ep_rew_mean     | 1.67     |
| time/              |          |
|    episodes        | 100      |
|    fps             | 38       |
|    time_elapsed    | 10       |
|    total timesteps | 400      |
```

(continues on next page)

(continued from previous page)

```
---------------------------------
---------------------------------
| rollout/           |          |
|    ep_len_mean     | 4        |
|    ep_rew_mean     | 2.18     |
| time/              |          |
|    episodes        | 200      |
|    fps             | 37       |
|    time_elapsed    | 21       |
|    total timesteps | 800      |
---------------------------------
---------------------------------
| rollout/           |          |
|    ep_len_mean     | 4        |
|    ep_rew_mean     | 1.43     |
| time/              |          |
|    episodes        | 300      |
|    fps             | 4        |
|    time_elapsed    | 258      |
|    total timesteps | 1200     |
| train/             |          |
|    actor_loss      | -0.699   |
|    critic_loss     | 0.0808   |
|    learning_rate   | 0.001    |
|    n_updates       | 196      |
---------------------------------
baseline: 0.0, ours: 0.8412776898734177, bits: 12.0 / 32
Saving to examples/logs/ddpg/mnist_trainer_2
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

4.1 eve package

4.1.1 Subpackages

eve.app package

Subpackages

eve.app.algorithm package

Submodules

eve.app.algorithm.a2c module

eve.app.algorithm.ddpg module

eve.app.algorithm.dqn module

eve.app.algorithm.ppo module

eve.app.algorithm.sac module

eve.app.algorithm.td3 module

Module contents

Submodules

eve.app.algo module

eve.app.buffers module

class eve.app.buffers.**RolloutBufferSamples**(*observations*, *actions*, *old_values*, *old_log_prob*, *advantages*, *returns*)

Bases: tuple

Create new instance of RolloutBufferSamples(observations, actions, old_values, old_log_prob, advantages, returns)

property observations
Alias for field number 0

property actions
Alias for field number 1

property old_values
Alias for field number 2

property old_log_prob
Alias for field number 3

property advantages
Alias for field number 4

property returns
Alias for field number 5

class eve.app.buffers.**ReplayBufferSamples**(*observations*, *actions*, *next_observations*, *dones*, *rewards*)

Bases: tuple

Create new instance of ReplayBufferSamples(observations, actions, next_observations, dones, rewards)

property observations
Alias for field number 0

property actions
Alias for field number 1

property next_observations
Alias for field number 2

property dones
Alias for field number 3

property rewards
Alias for field number 4

class eve.app.buffers.**RolloutReturn**(*episode_reward*, *episode_timesteps*, *n_episodes*, *continue_training*)

Bases: tuple

Create new instance of RolloutReturn(episode_reward, episode_timesteps, n_episodes, continue_training)

property episode_reward
Alias for field number 0

property episode_timesteps
Alias for field number 1

property n_episodes
Alias for field number 2

property continue_training
Alias for field number 3

eve.app.buffers.**get_action_dim**(*action_space: eve.app.space.EveSpace*) → int
Get the dimension of the action space.

Parameters **action_space** –

Returns

eve.app.buffers.**get_obs_shape**(*observation_space: eve.app.space.EveSpace*) → Tuple[int, ...]
Get the shape of the observation (useful for the buffers).

Parameters **observation_space** –

Returns

class eve.app.buffers.**BaseBuffer**(*buffer_size: int*, *observation_space: eve.app.space.EveSpace*, *action_space: eve.app.space.EveSpace*, *device: Union[torch.device, str] = 'cpu'*, *n_envs: int = 1*, *sample_episode: bool = False*)
Bases: abc.ABC

Base class that represent a buffer (rollout or replay)

Parameters

- **buffer_size** – Max number of element in the buffer
- **observation_space** – Observation space
- **action_space** – Action space
- **device** – PyTorch device to which the values will be converted
- **n_envs** – Number of parallel environments
- **sample_episode** – If False, we will sample the observations in a ramdon states format, and will return batch_size states. If False, we will sample the observation in a random episode formot, and will return batch_size episodes. NOTE: if True, all the episodes length should keep the same, or the batch size should be 1, otherwise, we can't stack differnt length of episodes.

static swap_and_flatten(*arr: numpy.ndarray*) → numpy.ndarray
Swap and then flatten axes 0 (buffer_size) and 1 (n_envs) to convert shape from [n_steps, n_envs, ...] (when ... is the shape of the features) to [n_steps * n_envs, ...] (which maintain the order)

Parameters **arr** –

Returns

size() → int

Returns The current size of the buffer

add(**args*, ***kwargs*) → None
Add elements to the buffer.

extend(**args*, ***kwargs*) → None
Add a new batch of transitions to the buffer

reset() → None
Reset the buffer.

sample(*batch_size: int*, *env: Optional[VecNormalize] = None*)

Parameters

- **batch_size** – Number of element to sample

- **env** – associated VecEnv to normalize the observations/rewards when sampling

Returns if episode sample, return a list with episode length and contains BufferSamples, else, return BufferSamples.

to_torch(*array: numpy.ndarray*, *copy: bool = True*) → torch.Tensor
Convert a numpy array to a PyTorch tensor. Note: it copies the data by default

Parameters

- **array** –
- **copy** – Whether to copy or not the data (may be useful to avoid changing things be reference)

Returns

class eve.app.buffers.**ReplayBuffer**(*buffer_size: int*, *observation_space:* eve.app.space.EveSpace, *action_space:* eve.app.space.EveSpace, *device: Union[torch.device, str] = 'cpu'*, *n_envs: int = 1*, *sample_episode: bool = False*)

Bases: *eve.app.buffers.BaseBuffer*

Replay buffer used in off-policy algorithms like SAC/TD3.

Parameters

- **buffer_size** – Max number of element in the buffer
- **observation_space** – Observation space
- **action_space** – Action space
- **device** –
- **n_envs** – Number of parallel environments

add(*obs: numpy.ndarray*, *next_obs: numpy.ndarray*, *action: numpy.ndarray*, *reward: numpy.ndarray*, *done: numpy.ndarray*) → None

class eve.app.buffers.**RolloutBuffer**(*buffer_size: int*, *observation_space:* eve.app.space.EveSpace, *action_space:* eve.app.space.EveSpace, *device: Union[torch.device, str] = 'cpu'*, *gae_lambda: float = 1*, *gamma: float = 0.99*, *n_envs: int = 1*, *sample_episode: bool = False*)

Bases: *eve.app.buffers.BaseBuffer*

Rollout buffer used in on-policy algorithms like A2C/PPO. It corresponds to `buffer_size` transitions collected using the current policy. This experience will be discarded after the policy update. In order to use PPO objective, we also store the current value of each state and the log probability of each taken action.

The term rollout here refers to the model-free notion and should not be used with the concept of rollout used in model-based RL or planning. Hence, it is only involved in policy and value function training but not action selection.

Parameters

- **buffer_size** – Max number of element in the buffer
- **observation_space** – Observation space
- **action_space** – Action space
- **device** –
- **gae_lambda** – Factor for trade-off of bias vs variance for Generalized Advantage Estimator Equivalent to classic advantage when set to 1.

- **gamma** – Discount factor
- **n_envs** – Number of parallel environments

reset() → None

compute_returns_and_advantage(*last_values: torch.Tensor*, *dones: numpy.ndarray*) → None
Post-processing step: compute the returns (sum of discounted rewards) and GAE advantage. Adapted from Stable-Baselines PPO2.

Uses Generalized Advantage Estimation (https://arxiv.org/abs/1506.02438) to compute the advantage. To obtain vanilla advantage (A(s) = R - V(S)) where R is the discounted reward with value bootstrap, set `gae_lambda=1.0` during initialization.

Parameters

- **last_values** –
- **dones** –

add(*obs: numpy.ndarray*, *action: numpy.ndarray*, *reward: numpy.ndarray*, *done: numpy.ndarray*, *value: torch.Tensor*, *log_prob: torch.Tensor*) → None

Parameters

- **obs** – Observation
- **action** – Action
- **reward** –
- **done** – End of episode signal.
- **value** – estimated value of the current state following the current policy.
- **log_prob** – log probability of the action following the current policy.

eve.app.callbacks module

eve.app.callbacks.**sync_envs_normalization**(*env:* EveEnv, *eval_env:* EveEnv) → None
Sync eval env and train env when using VecNormalize

Parameters

- **env** –
- **eval_env** –

eve.app.callbacks.**evaluate_policy**(*model: algo.BaseAlgorithm*, *env:* EveEnv, *n_eval_episodes: int = 10*, *deterministic: bool = True*, *callback: Optional[Callable[[Dict[str, Any], Dict[str, Any]], None]] = None*, *reward_threshold: Optional[float] = None*, *return_episode_rewards: bool = False*, *warn: bool = True*) → Union[Tuple[float, float], Tuple[List[float], List[int]]]
Runs policy for `n_eval_episodes` episodes and returns average reward. This is made to work only with one env.

Note: If environment has not been wrapped with `Monitor` wrapper, reward and episode lengths are counted as it appears with `env.step` calls. If the environment contains wrappers that modify rewards or episode lengths (e.g. reward scaling, early episode reset), these will affect the evaluation results as well. You can avoid this by wrapping environment with `Monitor` wrapper before anything else.

Parameters

- **model** – The RL agent you want to evaluate.
- **env** – The environment. In the case of a `VecEnv` this must contain only one environment.
- **n_eval_episodes** – Number of episode to evaluate the agent
- **deterministic** – Whether to use deterministic or stochastic actions
- **callback** – callback function to do additional checks, called after each step. Gets locals() and globals() passed as parameters.
- **reward_threshold** – Minimum expected reward per episode, this will raise an error if the performance is not met
- **return_episode_rewards** – If True, a list of rewards and episde lengths per episode will be returned instead of the mean.
- **warn** – If True (default), warns user about lack of a Monitor wrapper in the evaluation environment.

Returns Mean reward per episode, std of reward per episode. Returns ([float], [int]) when `return_episode_rewards` is True, first list containing per-episode rewards and second containing per-episode lengths (in number of steps).

class eve.app.callbacks.**BaseCallback**(*verbose: int = 0*)

Bases: `abc.ABC`

Base class for callback.

Parameters **verbose** –

init_callback(*model: algo.BaseAlgorithm*) → None
Initialize the callback by saving references to the RL model and the training environment for convenience.

on_training_start(*locals_: Dict[str, Any]*, *globals_: Dict[str, Any]*) → None

on_rollout_start() → None

on_step() → bool
This method will be called by the model after each call to `env.step()`.

For child callback (of an `EventCallback`), this will be called when the event is triggered.

Returns If the callback returns False, training is aborted early.

on_training_end() → None

on_rollout_end() → None

update_locals(*locals_: Dict[str, Any]*) → None
Update the references to the local variables.

Parameters **locals** – the local variables during rollout collection

update_child_locals(*locals_: Dict[str, Any]*) → None
Update the references to the local variables on sub callbacks.

Parameters **locals** – the local variables during rollout collection

class eve.app.callbacks.**EventCallback**(*callback: Optional[eve.app.callbacks.BaseCallback] = None*, *verbose: int = 0*)

Bases: *eve.app.callbacks.BaseCallback*

Base class for triggering callback on event.

Parameters

- **callback** – Callback that will be called when an event is triggered.
- **verbose** –

init_callback(*model: algo.BaseAlgorithm*) → None

update_child_locals(*locals_: Dict[str, Any]*) → None
Update the references to the local variables.

Parameters **locals** – the local variables during rollout collection

class eve.app.callbacks.**CallbackList**(*callbacks: List[*eve.app.callbacks.BaseCallback*]*)
Bases: *eve.app.callbacks.BaseCallback*

Class for chaining callbacks.

Parameters **callbacks** – A list of callbacks that will be called sequentially.

update_child_locals(*locals_: Dict[str, Any]*) → None
Update the references to the local variables.

Parameters **locals** – the local variables during rollout collection

class eve.app.callbacks.**CheckpointCallback**(*save_freq: int*, *save_path: str*, *name_prefix: str = 'rl_model'*, *verbose: int = 0*)
Bases: *eve.app.callbacks.BaseCallback*

Callback for saving a model every `save_freq` steps

Parameters

- **save_freq** –
- **save_path** – Path to the folder where the model will be saved.
- **name_prefix** – Common prefix to the saved models
- **verbose** –

class eve.app.callbacks.**ConvertCallback**(*callback: Callable[[Dict[str, Any], Dict[str, Any]], bool]*, *verbose: int = 0*)
Bases: *eve.app.callbacks.BaseCallback*

Convert functional callback (old-style) to object.

Parameters

- **callback** –
- **verbose** –

class eve.app.callbacks.**EvalCallback**(*eval_env:* EveEnv, *callback_on_new_best: Optional[*eve.app.callbacks.BaseCallback*] = None*, *n_eval_episodes: int = 5*, *eval_freq: int = 10000*, *log_path: str = None*, *best_model_save_path: str = None*, *deterministic: bool = True*, *verbose: int = 1*, *warn: bool = True*)
Bases: *eve.app.callbacks.EventCallback*

Callback for evaluating an agent.

Parameters

- **eval_env** – The environment used for initialization

- **callback_on_new_best** – Callback to trigger when there is a new best model according to the mean_reward
- **n_eval_episodes** – The number of episodes to test the agent
- **eval_freq** – Evaluate the agent every eval_freq call of the callback.
- **log_path** – Path to a folder where the evaluations (evaluations.npz) will be saved. It will be updated at each evaluation.
- **best_model_save_path** – Path to a folder where the best model according to performance on the eval env will be saved.
- **deterministic** – Whether the evaluation should use a stochastic or deterministic actions.
- **verbose** –
- **warn** – Passed to evaluate_policy (warns if eval_env has not been wrapped with a Monitor wrapper)

update_child_locals(*locals_: Dict[str, Any]*) → None
Update the references to the local variables.

Parameters **locals** – the local variables during rollout collection

class eve.app.callbacks.**StopTrainingOnRewardThreshold**(*reward_threshold: float*, *verbose: int = 0*)
Bases: *eve.app.callbacks.BaseCallback*

Stop the training once a threshold in episodic reward has been reached (i.e. when the model is good enough).

It must be used with the EvalCallback.

Parameters

- **reward_threshold** – Minimum expected reward per episode to stop training.
- **verbose** –

class eve.app.callbacks.**EveryNTimesteps**(*n_steps: int*, *callback:* eve.app.callbacks.BaseCallback)
Bases: *eve.app.callbacks.EventCallback*

Trigger a callback every n_steps timesteps

Parameters

- **n_steps** – Number of timesteps between two trigger.
- **callback** – Callback that will be called when the event is triggered.

class eve.app.callbacks.**StopTrainingOnMaxEpisodes**(*max_episodes: int*, *verbose: int = 0*)
Bases: *eve.app.callbacks.BaseCallback*

Stop the training once a maximum number of episodes are played.

For multiple environments presumes that, the desired behavior is that the agent trains on each env for max_episodes and in total for max_episodes * n_envs episodes.

Parameters

- **max_episodes** – Maximum number of episodes to stop training.
- **verbose** – Select whether to print information about when training ended by reaching max_episodes

class eve.app.callbacks.**TrialEvalCallback**(*eval_env: eve.app.env.VecEnv*, *trial: optuna.trial._trial.Trial*, *n_eval_episodes: int = 5*, *eval_freq: int = 10000*, *deterministic: bool = True*, *verbose: int = 0*)

Bases: *eve.app.callbacks.EvalCallback*

Callback used for evaluating and reporting a trial.

class eve.app.callbacks.**SaveVecNormalizeCallback**(*save_freq: int*, *save_path: str*, *name_prefix: Optional[str] = None*, *verbose: int = 0*)

Bases: *eve.app.callbacks.BaseCallback*

Callback for saving a VecNormalize wrapper every `save_freq` steps.

Parameters

- **save_freq** (*int*) –
- **save_path** (*str*) – Path to the folder where `VecNormalize` will be saved, as `vecnormalize.pkl`
- **name_prefix** (*str*) – Common prefix to the saved `VecNormalize`, if None (default) only one file will be kept.

eve.app.env module

class eve.app.env.**EveEnv**

Bases: `object`

The main OpenAI class. It encapsulates an environment with arbitrary behind-the-scenes dynamics. An environment can be partially or fully observed.

The main API methods that users of this class need to know are:

step reset render close seed

And set the following attributes:

action_space: The Space object corresponding to valid actions observation_space: The Space object corresponding to valid observations reward_range: A tuple corresponding to the min and max possible rewards

Note: a default reward range set to [-inf,+inf] already exists. Set it if you want a narrower range.

The methods are accessed publicly as "step", "reset", etc…

metadata = {'render.modes': []}

reward_range = (-inf, inf)

spec = None

action_space = None

observation_space = None

step(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

`reset()`

Resets the environment to an initial state and returns an initial observation.

Note that this function should not reset the environment's random number generator(s); random variables in the environment's state should be sampled independently between multiple calls to *reset()*. In other words, each call of *reset()* should yield an environment suitable for a new episode, independent of previous episodes.

Returns the initial observation.

Return type observation (object)

`render(`*mode='human'*`)`

Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

Note:

Make sure that your class's metadata 'render.modes' key includes the list of supported modes. It's recommended to call super() in implementations to use the functionality of this method.

Parameters `mode` (*str*) – the mode to render with

Example:

class MyEnv(EveEnv): metadata = {'render.modes': ['human', 'rgb_array']}

def render(self, mode='human'):

if mode == 'rgb_array': return np.array(...) # return RGB frame suitable for video

elif mode == 'human': ... # pop up a window and render

else: super(MyEnv, self).render(mode=mode) # just raise an exception

`close()`

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

`seed(`*seed=None*`)`

Sets the seed for this env's random number generator(s).

Note: Some environments use multiple pseudorandom number generators. We want to capture all such seeds used in order to ensure that there aren't accidental correlations between multiple generators.

Returns

Returns the list of seeds used in this env's random number generators. The first value in the list should be the "main" seed, or the value which a reproducer should pass to 'seed'. Often, the main seed equals the provided 'seed', but this won't be true if seed=None, for example.

Return type list<bigint>

property unwrapped

Completely unwrap this env.

Returns The base non-wrapped EveEnv instance

Return type *EveEnv*

class eve.app.env.**GoalEnv**

Bases: *eve.app.env.EveEnv*

A goal-based environment. It functions just as any regular OpenAI environment but it imposes a required structure on the observation_space. More concretely, the observation space is required to contain at least three elements, namely *observation*, *desired_goal*, and *achieved_goal*. Here, *desired_goal* specifies the goal that the agent should attempt to achieve. *achieved_goal* is the goal that it currently achieved instead. *observation* contains the actual observations of the environment as per usual.

reset()

compute_reward(*achieved_goal*, *desired_goal*, *info*)

Compute the step reward. This externalizes the reward function and makes it dependent on a desired goal and the one that was achieved. If you wish to include additional rewards that are independent of the goal, you can include the necessary values to derive it in 'info' and compute it accordingly.

Parameters

- **achieved_goal** (*object*) – the goal that was achieved during execution
- **desired_goal** (*object*) – the desired goal that we asked the agent to attempt to achieve
- **info** (*dict*) – an info dictionary with additional information

Returns

The reward that corresponds to the provided achieved goal w.r.t. to the desired goal. Note that the following should always hold true:

ob, reward, done, info = env.step() assert reward == env.compute_reward(ob['achieved_goal'], ob['goal'], info)

Return type float

class eve.app.env.**Wrapper**(*env*)

Bases: *eve.app.env.EveEnv*

Wraps the environment to allow a modular transformation.

This class is the base class for all wrappers. The subclass could override some methods to change the behavior of the original environment without touching the original code.

Note: Don't forget to call `super().__init__(env)` if the subclass overrides `__init__()`.

property spec

classmethod class_name()

step(*action*)

reset(***kwargs*)

render(*mode='human'*, ***kwargs*)

close()

seed(*seed=None*)

compute_reward(*achieved_goal*, *desired_goal*, *info*)

property unwrapped

class eve.app.env.**ObservationWrapper**(*env*)
Bases: *eve.app.env.Wrapper*

reset(***kwargs*)

step(*action*)

observation(*observation*)

class eve.app.env.**RewardWrapper**(*env*)
Bases: *eve.app.env.Wrapper*

reset(***kwargs*)

step(*action*)

reward(*reward*)

class eve.app.env.**ActionWrapper**(*env*)
Bases: *eve.app.env.Wrapper*

reset(***kwargs*)

step(*action*)

action(*action*)

reverse_action(*action*)

class eve.app.env.**FlattenObservation**(*env*)
Bases: *eve.app.env.ObservationWrapper*

Observation wrapper that flattens the observation.

observation(*observation*)

class eve.app.env.**VecEnv**(*num_envs: int*, *observation_space:* eve.app.space.EveSpace, *action_space:* eve.app.space.EveSpace)
Bases: abc.ABC

An abstract asynchronous, vectorized environment.

Parameters

- **num_envs** – the number of environments
- **observation_space** – the observation space

- **action_space** – the action space

abstract reset() → Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]]
Reset all the environments and return an array of observations, or a tuple of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

Returns observation

abstract step_async(*actions: numpy.ndarray*) → None
Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

abstract step_wait() → Tuple[Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]], numpy.ndarray, numpy.ndarray, List[Dict]]
Wait for the step taken with step_async().

Returns observation, reward, done, information

abstract close() → None
Clean up the environment's resources.

abstract get_attr(*attr_name: str*, *indices: Union[None, int, Iterable[int]] = None*) → List[Any]
Return attribute from vectorized environment.

Parameters

- **attr_name** – The name of the attribute whose value to return
- **indices** – Indices of envs to get attribute from

Returns List of values of 'attr_name' in all environments

abstract set_attr(*attr_name: str*, *value: Any*, *indices: Union[None, int, Iterable[int]] = None*) → None
Set attribute inside vectorized environments.

Parameters

- **attr_name** – The name of attribute to assign new value
- **value** – Value to assign to *attr_name*
- **indices** – Indices of envs to assign value

Returns

abstract env_method(*method_name: str*, **method_args*, *indices: Union[None, int, Iterable[int]] = None*, ***method_kwargs*) → List[Any]
Call instance methods of vectorized environments.

Parameters

- **method_name** – The name of the environment method to invoke.
- **indices** – Indices of envs whose method to call
- **method_args** – Any positional arguments to provide in the call
- **method_kwargs** – Any keyword arguments to provide in the call

Returns List of items returned by the environment's method call

abstract env_is_wrapped(*wrapper_class: Type[*eve.app.env.Wrapper*]*, *indices: Union[None, int, Iterable[int]] = None*) → List[bool]

Check if environments are wrapped with a given wrapper.

Parameters

- **method_name** – The name of the environment method to invoke.
- **indices** – Indices of envs whose method to call
- **method_args** – Any positional arguments to provide in the call
- **method_kwargs** – Any keyword arguments to provide in the call

Returns True if the env is wrapped, False otherwise, for each env queried.

step(*actions: numpy.ndarray*) → Tuple[Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]], numpy.ndarray, numpy.ndarray, List[Dict]]

Step the environments with the given action

Parameters **actions** – the action

Returns observation, reward, done, information

abstract seed(*seed: Optional[int] = None*) → List[Union[None, int]]

Sets the random seeds for all environments, based on a given seed. Each individual environment will still get its own seed, by incrementing the given seed.

Parameters **seed** – The random seed. May be None for completely random seeding.

Returns Returns a list containing the seeds for each individual env. Note that all list elements may be None, if the env does not return anything when being seeded.

property unwrapped: *eve.app.env.VecEnv*

getattr_depth_check(*name: str*, *already_found: bool*) → Optional[str]

Check if an attribute reference is being hidden in a recursive call to __getattr__

Parameters

- **name** – name of attribute to check for
- **already_found** – whether this attribute has already been found in a wrapper

Returns name of module whose attribute is being shadowed, if any.

class eve.app.env.**VecEnvWrapper**(*venv:* eve.app.env.VecEnv, *observation_space: Optional[*eve.app.space.EveSpace*] = None*, *action_space: Optional[*eve.app.space.EveSpace*] = None*)

Bases: *eve.app.env.VecEnv*

Vectorized environment base class

Parameters

- **venv** – the vectorized environment to wrap
- **observation_space** – the observation space (can be None to load from venv)
- **action_space** – the action space (can be None to load from venv)

step_async(*actions: numpy.ndarray*) → None

abstract reset() → Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]]

abstract step_wait() → Tuple[Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]], numpy.ndarray, numpy.ndarray, List[Dict]]

seed(*seed: Optional[int] = None*) → List[Union[None, int]]

close() → None

get_attr(*attr_name: str*, *indices: Union[None, int, Iterable[int]] = None*) → List[Any]

set_attr(*attr_name: str*, *value: Any*, *indices: Union[None, int, Iterable[int]] = None*) → None

env_method(*method_name: str*, **method_args*, *indices: Union[None, int, Iterable[int]] = None*, ***method_kwargs*) → List[Any]

env_is_wrapped(*wrapper_class: Type[*eve.app.env.Wrapper*]*, *indices: Union[None, int, Iterable[int]] = None*) → List[bool]

getattr_recursive(*name: str*) → Any
Recursively check wrappers to find attribute.

Parameters **name** – name of attribute to look for

Returns attribute

getattr_depth_check(*name: str*, *already_found: bool*) → str
See base class.

Returns name of module whose attribute is being shadowed, if any.

eve.app.env.**copy_obs_dict**(*obs: Dict[str, numpy.ndarray]*) → Dict[str, numpy.ndarray]
Deep-copy a dict of numpy arrays.

Parameters **obs** – a dict of numpy arrays.

Returns a dict of copied numpy arrays.

eve.app.env.**dict_to_obs**(*space_:* eve.app.space.EveSpace, *obs_dict: Dict[Any, numpy.ndarray]*) → Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]]
Convert an internal representation raw_obs into the appropriate type specified by space.

Parameters

- **space** – an observation space.
- **obs_dict** – a dict of numpy arrays.

Returns returns an observation of the same type as space. If space is Dict, function is identity; if space is Tuple, converts dict to Tuple; otherwise, space is unstructured and returns the value raw_obs[None].

eve.app.env.**obs_space_info**(*obs_space:* eve.app.space.EveSpace) → Tuple[List[str], Dict[Any, Tuple[int, ...]], Dict[Any, numpy.dtype]]
Get dict-structured information about a eve.app.EveSpace.

Dict spaces are represented directly by their dict of subspaces. Tuple spaces are converted into a dict with keys indexing into the tuple. Unstructured spaces are represented by {None: obs_space}.

Parameters **obs_space** – an observation space

Returns A tuple (keys, shapes, dtypes): keys: a list of dict keys. shapes: a dict mapping keys to shapes. dtypes: a dict mapping keys to dtypes.

class eve.app.env.**ObsDictWrapper**(*venv:* eve.app.env.VecEnv)
Bases: *eve.app.env.VecEnvWrapper*

Wrapper for a VecEnv which overrides the observation space for Hindsight Experience Replay to support dict observations.

Parameters **env** – The vectorized environment to wrap.

reset()

step_wait()

static convert_dict(*observation_dict: Dict[str, numpy.ndarray]*, *observation_key: str = 'observation'*, *goal_key: str = 'desired_goal'*) → numpy.ndarray

Concatenate observation and (desired) goal of observation dict.

Parameters

- **observation_dict** – Dictionary with observation.
- **observation_key** – Key of observation in dicitonary.
- **goal_key** – Key of (desired) goal in dicitonary.

Returns Concatenated observation.

class eve.app.env.**CloudpickleWrapper**(*var: Any*)

Bases: object

Uses cloudpickle to serialize contents (otherwise multiprocessing tries to use pickle)

Parameters **var** – the variable you wish to wrap for pickling with cloudpickle

class eve.app.env.**DummyVecEnv**(*env_fns: List[Callable[[], eve.app.env.EveEnv]]*)

Bases: *eve.app.env.VecEnv*

Creates a simple vectorized wrapper for multiple environments, calling each environment in sequence on the current Python process. This is useful for computationally simple environment such as `cartpole-v1`, as the overhead of multiprocess or multithread outweighs the environment computation time. This can also be used for RL methods that require a vectorized environment, but that you want a single environments to train with.

Parameters **env_fns** – a list of functions that return environments to vectorize

step_async(*actions: numpy.ndarray*) → None

step_wait() → Tuple[Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]], numpy.ndarray, numpy.ndarray, List[Dict]]

seed(*seed: Optional[int] = None*) → List[Union[None, int]]

reset() → Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]]

close() → None

get_attr(*attr_name: str*, *indices: Union[None, int, Iterable[int]] = None*) → List[Any]

Return attribute from vectorized environment (see base class).

set_attr(*attr_name: str*, *value: Any*, *indices: Union[None, int, Iterable[int]] = None*) → None

Set attribute inside vectorized environments (see base class).

env_method(*method_name: str*, **method_args*, *indices: Union[None, int, Iterable[int]] = None*, ***method_kwargs*) → List[Any]

Call instance methods of vectorized environments.

env_is_wrapped(*wrapper_class: Type[eve.app.env.Wrapper]*, *indices: Union[None, int, Iterable[int]] = None*) → List[bool]

Check if worker environments are wrapped with a given wrapper

class eve.app.env.**SubprocVecEnv**(*env_fns: List[Callable[[], eve.app.env.EveEnv]]*, *start_method: Optional[str] = None*)

Bases: *eve.app.env.VecEnv*

Creates a multiprocess vectorized wrapper for multiple environments, distributing each environment to its own process, allowing significant speed up when the environment is computationally complex.

For performance reasons, if your environment is not IO bound, the number of environments should not exceed the number of logical cores on your CPU.

Warning: Only 'forkserver' and 'spawn' start methods are thread-safe, which is important when TensorFlow sessions or other non thread-safe libraries are used in the parent (see issue #217). However, compared to 'fork' they incur a small start-up cost and have restrictions on global variables. With those methods, users must wrap the code in an `if __name__ == "__main__":` block. For more information, see the multiprocessing documentation.

Parameters

- **env_fns** – Environments to run in subprocesses
- **start_method** – method used to start the subprocesses. Must be one of the methods returned by multiprocessing.get_all_start_methods(). Defaults to 'forkserver' on available platforms, and 'spawn' otherwise.

step_async(*actions: numpy.ndarray*) → None

step_wait() → Tuple[Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]], numpy.ndarray, numpy.ndarray, List[Dict]]

seed(*seed: Optional[int] = None*) → List[Union[None, int]]

reset() → Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]]

close() → None

get_attr(*attr_name: str*, *indices: Union[None, int, Iterable[int]] = None*) → List[Any]
Return attribute from vectorized environment (see base class).

set_attr(*attr_name: str*, *value: Any*, *indices: Union[None, int, Iterable[int]] = None*) → None
Set attribute inside vectorized environments (see base class).

env_method(*method_name: str*, **method_args*, *indices: Union[None, int, Iterable[int]] = None*, ***method_kwargs*) → List[Any]
Call instance methods of vectorized environments.

env_is_wrapped(*wrapper_class: Type[*eve.app.env.Wrapper*]*, *indices: Union[None, int, Iterable[int]] = None*) → List[bool]
Check if worker environments are wrapped with a given wrapper

class eve.app.env.**RunningMeanStd**(*epsilon: float = 0.0001*, *shape: Tuple[int, ...] = ()*)
Bases: object

Calulates the running mean and std of a data stream https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm

Parameters

- **epsilon** – helps with arithmetic issues
- **shape** – the shape of the data stream's output

update(*arr: numpy.ndarray*) → None

update_from_moments(*batch_mean: numpy.ndarray*, *batch_var: numpy.ndarray*, *batch_count: int*) → None

eve.app.env.**check_for_correct_spaces**(*env:* eve.app.env.EveEnv, *observation_space:* eve.app.space.EveSpace, *action_space:* eve.app.space.EveSpace) → None

Checks that the environment has same spaces as provided ones. Used by BaseAlgorithm to check if spaces match after loading the model with given env. Checked parameters: - observation_space - action_space

Parameters

- **env** – Environment to check for valid spaces
- **observation_space** – Observation space to check against
- **action_space** – Action space to check against

class eve.app.env.**VecNormalize**(*venv:* eve.app.env.VecEnv, *training: bool = True*, *norm_obs: bool = True*, *norm_reward: bool = True*, *clip_obs: float = 10.0*, *clip_reward: float = 10.0*, *gamma: float = 0.99*, *epsilon: float = 1e-08*)

Bases: *eve.app.env.VecEnvWrapper*

A moving average, normalizing wrapper for vectorized environment. has support for saving/loading moving average,

Parameters

- **venv** – the vectorized environment to wrap
- **training** – Whether to update or not the moving average
- **norm_obs** – Whether to normalize observation or not (default: True)
- **norm_reward** – Whether to normalize rewards or not (default: True)
- **clip_obs** – Max absolute value for observation
- **clip_reward** – Max value absolute for discounted reward
- **gamma** – discount factor
- **epsilon** – To avoid division by zero

set_venv(*venv:* eve.app.env.VecEnv) → None

Sets the vector environment to wrap to venv.

Also sets attributes derived from this such as *num_env*.

Parameters **venv** –

step_wait() → Tuple[Union[numpy.ndarray, Dict[str, numpy.ndarray], Tuple[numpy.ndarray, ...]], numpy.ndarray, numpy.ndarray, List[Dict]]

Apply sequence of actions to sequence of environments actions -> (observations, rewards, news)

where 'news' is a boolean vector indicating whether each element is new.

normalize_obs(*obs: Union[numpy.ndarray, Dict[str, numpy.ndarray]]*) → Union[numpy.ndarray, Dict[str, numpy.ndarray]]

Normalize observations using this VecNormalize's observations statistics. Calling this method does not update statistics.

normalize_reward(*reward: numpy.ndarray*) → numpy.ndarray

Normalize rewards using this VecNormalize's rewards statistics. Calling this method does not update statistics.

unnormalize_obs(*obs: Union[numpy.ndarray, Dict[str, numpy.ndarray]]*) → Union[numpy.ndarray, Dict[str, numpy.ndarray]]

unnormalize_reward(*reward: numpy.ndarray*) → numpy.ndarray

get_original_obs() → Union[numpy.ndarray, Dict[str, numpy.ndarray]]
Returns an unnormalized version of the observations from the most recent step or reset.

get_original_reward() → numpy.ndarray
Returns an unnormalized version of the rewards from the most recent step.

reset() → Union[numpy.ndarray, Dict[str, numpy.ndarray]]
Reset all environments :return: first observation of the episode

static load(*load_path: str*, *venv:* eve.app.env.VecEnv) → *eve.app.env.VecNormalize*
Loads a saved VecNormalize object.

Parameters

- **load_path** – the path to load from.
- **venv** – the VecEnv to wrap.

Returns

save(*save_path: str*) → None
Save current VecNormalize object with all running statistics and settings (e.g. clip_obs)

Parameters **save_path** – The path to save to

class eve.app.env.**Monitor**(*env:* eve.app.env.EveEnv, *filename: Optional[str] = None*, *allow_early_resets: bool = True*, *reset_keywords: Tuple[str, ...] = ()*, *info_keywords: Tuple[str, ...] = ()*)
Bases: *eve.app.env.Wrapper*

A monitor wrapper for Gym environments, it is used to know the episode reward, length, time and other data.

Parameters

- **env** – The environment
- **filename** – the location to save a log file, can be None for no log
- **allow_early_resets** – allows the reset of the environment before it is done
- **reset_keywords** – extra keywords for the reset call, if extra parameters are needed at reset
- **info_keywords** – extra information to log, from the information return of env.step()

EXT = 'monitor.csv'

reset(***kwargs*) → Union[Tuple, Dict[str, Any], numpy.ndarray, int]
Calls the environment reset. Can only be called if the environment is over, or if allow_early_resets is True

Parameters **kwargs** – Extra keywords saved for the next episode. only if defined by reset_keywords

Returns the first observation of the environment

step(*action: Union[numpy.ndarray, int]*) → Tuple[Union[Tuple, Dict[str, Any], numpy.ndarray, int], float, bool, Dict]
Step the environment with the given action

Parameters **action** – the action

Returns observation, reward, done, information

close() → None
Closes the environment

get_total_steps() → int
Returns the total number of timesteps

Returns

get_episode_rewards() → List[float]
Returns the rewards of all the episodes

Returns

get_episode_lengths() → List[int]
Returns the number of timesteps of all the episodes

Returns

get_episode_times() → List[float]
Returns the runtime in seconds of all the episodes

Returns

exception eve.app.env.**LoadMonitorResultsError**
Bases: Exception

Raised when loading the monitor log fails.

eve.app.env.**get_monitor_files**(*path: str*) → List[str]
get all the monitor files in the given path

Parameters **path** – the logging folder

Returns the log files

eve.app.env.**load_results**(*path: str*) → pandas.core.frame.DataFrame
Load all Monitor logs from a given directory path matching *monitor.csv

Parameters **path** – the directory path containing the log file(s)

Returns the logged data

eve.app.env.**rolling_window**(*array: numpy.ndarray*, *window: int*) → numpy.ndarray
Apply a rolling window to a np.ndarray

Parameters

- **array** – the input Array
- **window** – length of the rolling window

Returns rolling window on the input array

eve.app.env.**window_func**(*var_1: numpy.ndarray*, *var_2: numpy.ndarray*, *window: int*, *func: Callable*) → Tuple[numpy.ndarray, numpy.ndarray]
Apply a function to the rolling window of 2 arrays

Parameters

- **var_1** – variable 1
- **var_2** – variable 2
- **window** – length of the rolling window
- **func** – function to apply on the rolling window on variable 2 (such as np.mean)

Returns the rolling output with applied function

eve.app.env.**ts2xy**(*data_frame: pandas.core.frame.DataFrame*, *x_axis: str*) → Tuple[numpy.ndarray, numpy.ndarray]
Decompose a data frame variable to x ans ys

Parameters

- **data_frame** – the input data
- **x_axis** – the axis for the x and y output (can be X_TIMESTEPS='timesteps', X_EPISODES='episodes' or X_WALLTIME='walltime_hrs')

Returns the x and y output

eve.app.env.**plot_curves**(*xy_list: List[Tuple[numpy.ndarray, numpy.ndarray]]*, *x_axis: str*, *title: str*, *figsize: Tuple[int, int] = (8, 2)*) → None

plot the curves

Parameters

- **xy_list** – the x and y coordinates to plot
- **x_axis** – the axis for the x and y output (can be X_TIMESTEPS='timesteps', X_EPISODES='episodes' or X_WALLTIME='walltime_hrs')
- **title** – the title of the plot
- **figsize** – Size of the figure (width, height)

eve.app.env.**plot_results**(*dirs: List[str]*, *num_timesteps: Optional[int]*, *x_axis: str*, *task_name: str*, *figsize: Tuple[int, int] = (8, 2)*) → None

Plot the results using csv files from `Monitor` wrapper.

Parameters

- **dirs** – the save location of the results to plot
- **num_timesteps** – only plot the points below this value
- **x_axis** – the axis for the x and y output (can be X_TIMESTEPS='timesteps', X_EPISODES='episodes' or X_WALLTIME='walltime_hrs')
- **task_name** – the title of the task to plot
- **figsize** – Size of the figure (width, height)

eve.app.env.**unwrap_vec_wrapper**(*env: Union[*eve.app.env.EveEnv, eve.app.env.VecEnv*]*, *vec_wrapper_class: Type[*eve.app.env.VecEnvWrapper*]*) → Optional[*eve.app.env.VecEnvWrapper*]

Retrieve a `VecEnvWrapper` object by recursively searching.

Parameters

- **env** –
- **vec_wrapper_class** –

Returns

eve.app.env.**unwrap_vec_normalize**(*env: Union[*eve.app.env.EveEnv, eve.app.env.VecEnv*]*) → Optional[*eve.app.env.VecNormalize*]

Parameters **env** –

Returns

eve.app.env.**is_vecenv_wrapped**(*env: Union[*eve.app.env.EveEnv, eve.app.env.VecEnv*]*, *vec_wrapper_class: Type[*eve.app.env.VecEnvWrapper*]*) → bool

Check if an environment is already wrapped by a given `VecEnvWrapper`.

Parameters

- **env** –

- **vec_wrapper_class** –

Returns

eve.app.env.**unwrap_wrapper**(*env:* eve.app.env.EveEnv, *wrapper_class: Type[*eve.app.env.Wrapper*]*) → Optional[*eve.app.env.Wrapper*]

Retrieve a `VecEnvWrapper` object by recursively searching.

Parameters

- **env** – Environment to unwrap
- **wrapper_class** – Wrapper to look for

Returns Environment unwrapped till `wrapper_class` if it has been wrapped with it

eve.app.env.**is_wrapped**(*env: Type[*eve.app.env.EveEnv*]*, *wrapper_class: Type[*eve.app.env.Wrapper*]*) → bool

Check if a given environment has been wrapped with a given wrapper.

Parameters

- **env** – Environment to check
- **wrapper_class** – Wrapper class to look for

Returns True if environment has been wrapped with `wrapper_class`.

eve.app.env.**get_wrapper_class**(*hyperparams: Dict[str, Any]*) → Optional[Callable[[*eve.app.env.EveEnv*], *eve.app.env.EveEnv*]]

Get one or more environment wrapper class specified as a hyper parameter "env_wrapper". e.g. env_wrapper: _minigrid.wrappers.FlatObsWrapper

for multiple, specify a list:

env_wrapper:

- utils.wrappers.PlotActionWrapper
- utils.wrappers.TimeFeatureWrapper

Parameters **hyperparams** –

Returns maybe a callable to wrap the environment with one or multiple Wrapper

eve.app.env.**make_vec_env**(*env_id: Union[str, Type[*eve.app.env.EveEnv*]]*, *n_envs: int = 1*, *seed: Optional[int] = None*, *start_index: int = 0*, *monitor_dir: Optional[str] = None*, *wrapper_class: Optional[Callable[[*eve.app.env.EveEnv*],* eve.app.env.EveEnv*]] = None*, *env_kwargs: Optional[Dict[str, Any]] = None*, *vec_env_cls: Optional[Type[Union[*eve.app.env.DummyVecEnv*,* eve.app.env.SubprocVecEnv*]]] = None*, *vec_env_kwargs: Optional[Dict[str, Any]] = None*, *monitor_kwargs: Optional[Dict[str, Any]] = None*) → *eve.app.env.VecEnv*

Create a wrapped, monitored `VecEnv`. By default it uses a `DummyVecEnv` which is usually faster than a `SubprocVecEnv`.

Parameters

- **env_id** – the environment ID or the environment class
- **n_envs** – the number of environments you wish to have in parallel
- **seed** – the initial seed for the random number generator
- **start_index** – start rank index

- **monitor_dir** – Path to a folder where the monitor files will be saved. If None, no file will be written, however, the env will still be wrapped in a Monitor wrapper to provide additional information about training.
- **wrapper_class** – Additional wrapper to use on the environment. This can also be a function with single argument that wraps the environment in many things.
- **env_kwargs** – Optional keyword argument to pass to the env constructor
- **vec_env_cls** – A custom `VecEnv` class constructor. Default: None.
- **vec_env_kwargs** – Keyword arguments to pass to the `VecEnv` class constructor.
- **monitor_kwargs** – Keyword arguments to pass to the `Monitor` class constructor.

Returns The wrapped environment

eve.app.env.**create_test_env**(*env_id: str*, *n_envs: int = 1*, *stats_path: Optional[str] = None*, *seed: int = 0*, *log_dir: Optional[str] = None*, *should_render: bool = True*, *hyperparams: Optional[Dict[str, Any]] = None*, *env_kwargs: Optional[Dict[str, Any]] = None*) → *eve.app.env.VecEnv*

Create environment for testing a trained agent

Parameters

- **env_id** –
- **n_envs** – number of processes
- **stats_path** – path to folder containing saved running averaged
- **seed** – Seed for random number generator
- **log_dir** – Where to log rewards
- **should_render** – For Pybullet env, display the GUI
- **hyperparams** – Additional hyperparams (ex: n_stack)
- **env_kwargs** – Optional keyword argument to pass to the env constructor

eve.app.exp_manager module

eve.app.hyperparams_opt module

eve.app.logger module

exception eve.app.logger.**FormatUnsupportedError**(*unsupported_formats: Sequence[str]*, *value_description: str*)

Bases: `NotImplementedError`

class eve.app.logger.**KVWriter**

Bases: `object`

Key Value writer

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None

Write a dictionary to file

Parameters

- **key_values** –

- **key_excluded** –
- **step** –

close() → None
Close owned resources

class eve.app.logger.**SeqWriter**
Bases: object

sequence writer

write_sequence(*sequence: List*) → None
write_sequence an array to file

Parameters **sequence** –

class eve.app.logger.**HumanOutputFormat**(*filename_or_file: Union[str, TextIO]*)
Bases: *eve.app.logger.KVWriter*, *eve.app.logger.SeqWriter*

log to a file, in a human readable format

Parameters **filename_or_file** – the file to write the log to

write(*key_values: Dict*, *key_excluded: Dict*, *step: int = 0*) → None

write_sequence(*sequence: List*) → None

close() → None
closes the file

eve.app.logger.**filter_excluded_keys**(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *_format: str*) → Dict[str, Any]
Filters the keys specified by key_exclude for the specified format

Parameters

- **key_values** – log dictionary to be filtered
- **key_excluded** – keys to be excluded per format
- **_format** – format for which this filter is run

Returns dict without the excluded keys

class eve.app.logger.**JSONOutputFormat**(*filename: str*)
Bases: *eve.app.logger.KVWriter*

log to a file, in the JSON format

Parameters **filename** – the file to write the log to

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None

close() → None
closes the file

class eve.app.logger.**CSVOutputFormat**(*filename: str*)
Bases: *eve.app.logger.KVWriter*

log to a file, in a CSV format

Parameters **filename** – the file to write the log to

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None

close() → None
closes the file

class eve.app.logger.**TensorBoardOutputFormat**(*folder: str*)
Bases: *eve.app.logger.KVWriter*

Dumps key/value pairs into TensorBoard's numeric format.

Parameters **folder** – the folder to write the log to

write(*key_values: Dict[str, Any]*, *key_excluded: Dict[str, Union[str, Tuple[str, ...]]]*, *step: int = 0*) → None

close() → None
closes the file

eve.app.logger.**make_output_format**(*_format: str*, *log_dir: str*, *log_suffix: str = ''*) → *eve.app.logger.KVWriter*
return a logger for the requested format

Parameters

- **_format** – the requested format to log to ('stdout', 'log', 'json' or 'csv' or 'tensorboard')
- **log_dir** – the logging directory
- **log_suffix** – the suffix for the log file

Returns the logger

class eve.app.logger.**Logger**(*folder: Optional[str]*, *output_formats: List[*eve.app.logger.KVWriter*]*)
Bases: object

the logger class

Parameters

- **folder** – the logging location
- **output_formats** – the list of output format

DEFAULT = <eve.app.logger.Logger object>

CURRENT = <eve.app.logger.Logger object>

record(*key: str*, *value: Any*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → None
Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many times, last value will be used.

Parameters

- **key** – save to log this key
- **value** – save to log this value
- **exclude** – outputs to be excluded

record_mean(*key: str*, *value: Any*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → None
The same as record(), but if called many times, values averaged.

Parameters

- **key** – save to log this key
- **value** – save to log this value
- **exclude** – outputs to be excluded

dump(*step: int = 0*) → None
Write all of the diagnostics from the current iteration

log(**args*, *level: int = 20*) → None
Write the sequence of args, with no separators, to the console and output files (if you've configured an output file).

level: int. (see logger.py docs) If the global logger level is higher than the level argument here, don't print to stdout.

Parameters

- **args** – log the arguments
- **level** – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

set_level(*level: int*) → None
Set logging threshold on current logger.

Parameters **level** – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

get_dir() → str
Get directory that log files are being written to. will be None if there is no output directory (i.e., if you didn't call start)

Returns the logging directory

close() → None
closes the file

eve.app.logger.**configure**(*folder: Optional[str] = None*, *format_strings: Optional[List[str]] = None*) → None
configure the current logger

Parameters

- **folder** – the save location (if None, $SB3_LOGDIR, if still None, tempdir/baselines-[date & time])
- **format_strings** – the output logging format (if None, $SB3_LOG_FORMAT, if still None, ['stdout', 'log', 'csv'])

eve.app.logger.**reset**() → None
reset the current logger

class eve.app.logger.**ScopedConfigure**(*folder: Optional[str] = None*, *format_strings: Optional[List[str]] = None*)
Bases: object

Class for using context manager while logging

usage:

```
>>> with ScopedConfigure(folder=None, format_strings=None):
>>>     {code}
```

Parameters

- **folder** – the logging folder
- **format_strings** – the list of output logging format

eve.app.logger.**record**(*key: str*, *value: Any*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → None
Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many times, last value will be used.

Parameters

- **key** – save to log this key
- **value** – save to log this value
- **exclude** – outputs to be excluded

eve.app.logger.**record_mean**(*key: str*, *value: Union[int, float]*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → None
The same as record(), but if called many times, values averaged.

Parameters

- **key** – save to log this key
- **value** – save to log this value
- **exclude** – outputs to be excluded

eve.app.logger.**record_dict**(*key_values: Dict[str, Any]*) → None
Log a dictionary of key-value pairs.

Parameters **key_values** – the list of keys and values to save to log

eve.app.logger.**dump**(*step: int = 0*) → None
Write all of the diagnostics from the current iteration

eve.app.logger.**get_log_dict**() → Dict
get the key values logs

Returns the logged values

eve.app.logger.**log**(**args*, *level: int = 20*) → None
Write the sequence of args, with no separators, to the console and output files (if you've configured an output file).

level: int. (see logger.py docs) If the global logger level is higher than the level argument here, don't print to stdout.

Parameters

- **args** – log the arguments
- **level** – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

eve.app.logger.**debug**(**args*) → None
Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the DEBUG level.

Parameters **args** – log the arguments

eve.app.logger.**info**(**args*) → None
Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the INFO level.

Parameters **args** – log the arguments

eve.app.logger.**warn**(**args*) → None
Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the WARN level.

Parameters **args** – log the arguments

eve.app.logger.**error**(**args*) → None
Write the sequence of args, with no separators, to the console and output files (if you've configured an output file). Using the ERROR level.

Parameters **args** – log the arguments

eve.app.logger.**set_level**(*level: int*) → None
Set logging threshold on current logger.

Parameters **level** – the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

eve.app.logger.**get_level**() → int
Get logging threshold on current logger. :return: the logging level (can be DEBUG=10, INFO=20, WARN=30, ERROR=40, DISABLED=50)

eve.app.logger.**get_dir**() → str
Get directory that log files are being written to. will be None if there is no output directory (i.e., if you didn't call start)

Returns the logging directory

eve.app.logger.**record_tabular**(*key: str*, *value: Any*, *exclude: Optional[Union[str, Tuple[str, ...]]] = None*) → None
Log a value of some diagnostic Call this once for each diagnostic quantity, each iteration If called many times, last value will be used.

Parameters

- **key** – save to log this key
- **value** – save to log this value
- **exclude** – outputs to be excluded

eve.app.logger.**dump_tabular**(*step: int = 0*) → None
Write all of the diagnostics from the current iteration

eve.app.logger.**read_json**(*filename: str*) → pandas.core.frame.DataFrame
read a json file using pandas

Parameters **filename** – the file path to read

Returns the data in the json

eve.app.logger.**read_csv**(*filename: str*) → pandas.core.frame.DataFrame
read a csv file using pandas

Parameters **filename** – the file path to read

Returns the data in the csv

eve.app.model module

eve.app.policies module

eve.app.space module

eve.app.space.**np_random**(*seed=None*)

eve.app.space.**hash_seed**(*seed=None*, *max_bytes=8*)

Any given evaluation is likely to have many PRNG's active at once. (Most commonly, because the environment is running in multiple processes.) There's literature indicating that having linear correlations between seeds of multiple PRNG's can correlate the outputs:

http://blogs.unity3d.com/2015/01/07/a-primer-on-repeatable-random-numbers/ http://stackoverflow.com/questions/1554958/how-different-do-random-seeds-need-to-be http://dl.acm.org/citation.cfm?id=1276928

Thus, for sanity we hash the seeds before using them. (This scheme is likely not crypto-strength, but it should be good enough to get rid of simple correlations.)

Parameters

- **seed** (*Optional[int]*) – None seeds from an operating system specific randomness source.
- **max_bytes** – Maximum number of bytes to use in the hashed seed.

eve.app.space.**create_seed**(*a=None*, *max_bytes=8*)

Create a strong random seed. Otherwise, Python 2 would seed using the system time, which might be non-robust especially in the presence of concurrency.

Parameters

- **a** (*Optional[int, str]*) – None seeds from an operating system specific randomness source.
- **max_bytes** – Maximum number of bytes to use in the seed.

class eve.app.space.**EveSpace**(*shape=None*, *dtype=None*)

Bases: object

Defines the observation and action spaces, so you can write generic code that applies to any Env. For example, you can choose a random action.

WARNING - Custom observation & action spaces can inherit from the *Space* class. However, most use-cases should be covered by the existing space classes (e.g. *Box*, *Discrete*, etc...), and container classes (*Tuple* & *Dict*). Note that parametrized probability distributions (through the *sample()* method), and batching functions (in *eve.vector.VectorEnv*), are only well-defined for instances of spaces provided in eve by default. Moreover, some implementations of Reinforcement Learning algorithms might not handle custom spaces properly. Use custom spaces with care.

property np_random

Lazily seed the rng since this is expensive and only needed if sampling from this space.

sample()

Randomly sample an element of this space. Can be uniform or non-uniform sampling based on boundedness of space.

seed(*seed=None*)

Seed the PRNG of this space.

contains(*x*)

Return boolean specifying if x is a valid member of this space

to_jsonable(*sample_n*)
Convert a batch of samples from this space to a JSONable data type.

from_jsonable(*sample_n*)
Convert a JSONable data type to a batch of samples from this space.

class eve.app.space.**EveBox**(*low*, *high*, *shape=None*, *max_neurons: typing.Optional[int] = None*, *max_states: typing.Optional[int] = None*, *dtype=<class 'numpy.float32'>*)
Bases: *eve.app.space.EveSpace*

A (possibly unbounded) box in R^n. Specifically, a Box represents the Cartesian product of n closed intervals. Each interval has the form of one of [a, b], (-oo, b], [a, oo), or (-oo, oo).

There are two common use cases:

- **Identical bound for each dimension::**

```
>>> Box(low=-1.0, high=2.0, shape=(3, 4), dtype=np.float32)
Box(3, 4)
```

- **Independent bound for each dimension::**

```
>>> Box(low=np.array([-1.0, -2.0]), high=np.array([2.0, 4.0]), dtype=np.
↪float32)
Box(2,)
```

is_bounded(*manner='both'*)

sample()
Generates a single random sample inside of the Box.

In creating a sample of the box, each coordinate is sampled according to the form of the interval:

- [a, b] : uniform distribution
- [a, oo) : shifted exponential distribution
- (-oo, b] : shifted negative exponential distribution
- (-oo, oo) : normal distribution

contains(*x*)

to_jsonable(*sample_n*)

from_jsonable(*sample_n*)

class eve.app.space.**EveDict**(*spaces=None*, ***spaces_kwargs*)
Bases: *eve.app.space.EveSpace*

A dictionary of simpler spaces.

Example usage: self.observation_space = spaces.Dict({“position”: spaces.Discrete(2), “velocity”: spaces.Discrete(3)})

Example usage [nested]:

```
>>> self.nested_observation_space = spaces.Dict({
>>>         'sensors':  spaces.Dict({
>>>             'position': spaces.Box(low=-100, high=100, shape=(3,)),
>>>             'velocity': spaces.Box(low=-1, high=1, shape=(3,)),
>>>             'front_cam': spaces.Tuple((
>>>                 spaces.Box(low=0, high=1, shape=(10, 10, 3)),
```

(continues on next page)

(continued from previous page)

```
>>>                 spaces.Box(low=0, high=1, shape=(10, 10, 3))
>>>             )),
>>>             'rear_cam': spaces.Box(low=0, high=1, shape=(10, 10, 3)),
>>>         }),
>>>         'ext_controller': spaces.MultiDiscrete((5, 2, 2)),
>>>         'inner_state':spaces.Dict({
>>>             'charge': spaces.Discrete(100),
>>>             'system_checks': spaces.MultiBinary(10),
>>>             'job_status': spaces.Dict({
>>>                 'task': spaces.Discrete(5),
>>>                 'progress': spaces.Box(low=0, high=100, shape=()),
>>>             })
>>>         })
>>>     })
```

seed(*seed=None*)

sample()

contains(*x*)

to_jsonable(*sample_n*)

from_jsonable(*sample_n*)

class eve.app.space.**EveDiscrete**(*n*, *max_neurons: Optional[int] = None*, *max_states: Optional[int] = None*)

Bases: *eve.app.space.EveSpace*

A discrete space in $\{0, 1,$
$dots, n-1\}$.

Example:

```
>>> EveDiscrete(2)
```

sample()

contains(*x*)

class eve.app.space.**EveMultiBinary**(*n*, *max_neurons: Optional[int] = None*, *max_states: Optional[int] = None*)

Bases: *eve.app.space.EveSpace*

An n-shape binary space.

The argument to MultiBinary defines n, which could be a number or a *list* of numbers.

Example Usage:

>> self.observation_space = spaces.MultiBinary(5)

>> self.observation_space.sample()

array([0,1,0,1,0], dtype =int8)

>> self.observation_space = spaces.MultiBinary([3,2])

>> self.observation_space.sample()

array([[0, 0], [0, 1], [1, 1]], dtype=int8)

sample()

contains(*x*)

to_jsonable(*sample_n*)

from_jsonable(*sample_n*)

class eve.app.space.**EveMultiDiscrete**(*nvec*, *max_neurons: Optional[int] = None*, *max_states: Optional[int] = None*)

Bases: *eve.app.space.EveSpace*

- The multi-discrete action space consists of a series of discrete action spaces with different number of actions in eachs
- It is useful to represent game controllers or keyboards where each key can be represented as a discrete action space
- It is parametrized by passing an array of positive integers specifying number of actions for each discrete action space

Note: Some environment wrappers assume a value of 0 always represents the NOOP action.

e.g. Nintendo Game Controller - Can be conceptualized as 3 discrete action spaces:

1) Arrow Keys: Discrete 5 - NOOP[0], UP[1], RIGHT[2], DOWN[3], LEFT[4] - params: min: 0, max: 4
2) Button A: Discrete 2 - NOOP[0], Pressed[1] - params: min: 0, max: 1
3) Button B: Discrete 2 - NOOP[0], Pressed[1] - params: min: 0, max: 1

- Can be initialized as

  MultiDiscrete([5, 2, 2])

nvec: vector of counts of each categorical variable

sample()

contains(*x*)

to_jsonable(*sample_n*)

from_jsonable(*sample_n*)

class eve.app.space.**EveTuple**(*spaces*)

Bases: *eve.app.space.EveSpace*

A tuple (i.e., product) of simpler spaces

Example usage: self.observation_space = spaces.Tuple((spaces.Discrete(2), spaces.Discrete(3)))

seed(*seed=None*)

sample()

contains(*x*)

to_jsonable(*sample_n*)

from_jsonable(*sample_n*)

eve.app.space.**flatdim**(*space*)

Return the number of dimensions a flattened equivalent of this space would have.

Accepts a space and returns an integer. Raises `NotImplementedError` if the space is not defined in `gym.spaces`.

eve.app.space.**flatten**(*space*, *x*)
Flatten a data point from a space.

This is useful when e.g. points from spaces must be passed to a neural network, which only understands flat arrays of floats.

Accepts a space and a point from that space. Always returns a 1D array. Raises `NotImplementedError` if the space is not defined in `gym.spaces`.

eve.app.space.**unflatten**(*space*, *x*)
Unflatten a data point from a space.

This reverses the transformation applied by `flatten()`. You must ensure that the `space` argument is the same as for the `flatten()` call.

Accepts a space and a flattened point. Returns a point with a structure that matches the space. Raises `NotImplementedError` if the space is not defined in `gym.spaces`.

eve.app.space.**flatten_space**(*space*)
Flatten a space into a single `Box`.

This is equivalent to `flatten()`, but operates on the space itself. The result always is a *Box* with flat boundaries. The box has exactly `flatdim(space)` dimensions. Flattening a sample of the original space has the same effect as taking a sample of the flattenend space.

Raises `NotImplementedError` if the space is not defined in `gym.spaces`.

Example:

```
>>> box = Box(0.0, 1.0, shape=(3, 4, 5))
>>> box
Box(3, 4, 5)
>>> flatten_space(box)
Box(60,)
>>> flatten(box, box.sample()) in flatten_space(box)
True
```

Example that flattens a discrete space:

```
>>> discrete = Discrete(5)
>>> flatten_space(discrete)
Box(5,)
>>> flatten(box, box.sample()) in flatten_space(box)
True
```

Example that recursively flattens a dict:

```
>>> space = Dict({"position": Discrete(2),
...               "velocity": Box(0, 1, shape=(2, 2))})
>>> flatten_space(space)
Box(6,)
>>> flatten(space, space.sample()) in flatten_space(space)
True
```

eve.app.trainer module

eve.app.upgrader module

eve.app.utils module

eve.app.utils.**set_random_seed**(*seed: int*, *using_cuda: bool = False*) → None
Seed the different random generators :param seed: :param using_cuda:

eve.app.utils.**explained_variance**(*y_pred: numpy.ndarray*, *y_true: numpy.ndarray*) → numpy.ndarray
Computes fraction of variance that ypred explains about y. Returns 1 - Var[y-ypred] / Var[y]

interpretation: ev=0 => might as well have predicted zero ev=1 => perfect prediction ev<0 => worse than just predicting zero

Parameters

- **y_pred** – the prediction
- **y_true** – the expected value

Returns explained variance of ypred and y

eve.app.utils.**update_learning_rate**(*optimizer: torch.optim.optimizer.Optimizer*, *learning_rate: float*) → None
Update the learning rate for a given optimizer. Useful when doing linear schedule.

Parameters

- **optimizer** –
- **learning_rate** –

eve.app.utils.**get_schedule_fn**(*value_schedule: Union[Callable[[float], float], float, int]*) → Callable[[float], float]
Transform (if needed) learning rate and clip range (for PPO) to callable.

Parameters **value_schedule** –

Returns

eve.app.utils.**get_linear_fn**(*start: float*, *end: float*, *end_fraction: float*) → Callable[[float], float]
Create a function that interpolates linearly between start and end between `progress_remaining` = 1 and `progress_remaining` = `end_fraction`. This is used in DQN for linearly annealing the exploration fraction (epsilon for the epsilon-greedy strategy).

Params start value to start with if `progress_remaining` = 1

Params end value to end with if `progress_remaining` = 0

Params end_fraction fraction of `progress_remaining` where end is reached e.g 0.1 then end is reached after 10% of the complete training process.

Returns

eve.app.utils.**linear_schedule**(*initial_value: Union[float, str]*) → Callable[[float], float]
Linear learning rate schedule.

Parameters **initial_value** – (float or str)

eve.app.utils.**constant_fn**(*val: float*) → Callable[[float], float]
Create a function that returns a constant It is useful for learning rate schedule (to avoid code duplication)

Parameters `val` –

Returns

`eve.app.utils.`**`get_device`**(*device: Union[torch.device, str] = 'auto'*) → torch.device
Retrieve PyTorch device. It checks that the requested device is available first. For now, it supports only cpu and cuda. By default, it tries to use the gpu.

Parameters `device` – One for 'auto', 'cuda', 'cpu'

Returns

`eve.app.utils.`**`get_latest_run_id`**(*log_path: Optional[str] = None*, *log_name: str = ''*) → int
Returns the latest run number for the given log name and log path, by finding the greatest number in the directories.

Returns latest run number

`eve.app.utils.`**`safe_mean`**(*arr: Union[numpy.ndarray, list]*) → numpy.ndarray
Compute the mean of an array if there is at least one element. For empty array, return NaN. It is used for logging only.

Parameters `arr` –

Returns

`eve.app.utils.`**`zip_strict`**(**iterables: Iterable*) → Iterable
`zip()` function but enforces that iterables are of equal length. Raises `ValueError` if iterables not of equal length. Code inspired by Stackoverflow answer for question #32954486.

Parameters `*iterables` – iterables to `zip()`

`eve.app.utils.`**`polyak_update`**(*params: Iterable[torch.nn.parameter.Parameter]*, *target_params: Iterable[torch.nn.parameter.Parameter]*, *tau: float*) → None
Perform a Polyak average update on `target_params` using `params`: target parameters are slowly updated towards the main parameters. `tau`, the soft update coefficient controls the interpolation: `tau=1` corresponds to copying the parameters to the target ones whereas nothing happens when `tau=0`. The Polyak update is done in place, with `no_grad`, and therefore does not create intermediate tensors, or a computation graph, reducing memory cost and improving performance. We scale the target params by `1-tau` (in-place), add the new weights, scaled by `tau` and store the result of the sum in the target params (in place). See https://github.com/DLR-RM/stable-baselines3/issues/93

Parameters

- **`params`** – parameters to use to update the target params
- **`target_params`** – parameters to update
- **`tau`** – the soft update coefficient ("Polyak update", between 0 and 1)

`eve.app.utils.`**`recursive_getattr`**(*obj: Any*, *attr: str*, **args*) → Any
Recursive version of getattr taken from https://stackoverflow.com/questions/31174295

Ex: > MyObject.sub_object = SubObject(name='test') > recursive_getattr(MyObject, 'sub_object.name') # return test :param obj: :param attr: Attribute to retrieve :return: The attribute

`eve.app.utils.`**`recursive_setattr`**(*obj: Any*, *attr: str*, *val: Any*) → None
Recursive version of setattr taken from https://stackoverflow.com/questions/31174295

Ex: > MyObject.sub_object = SubObject(name='test') > recursive_setattr(MyObject, 'sub_object.name', 'hello') :param obj: :param attr: Attribute to set :param val: New value of the attribute

`eve.app.utils.`**`is_json_serializable`**(*item: Any*) → bool
Test if an object is serializable into JSON

Parameters **item** – The object to be tested for JSON serialization.

Returns True if object is JSON serializable, false otherwise.

eve.app.utils.**data_to_json**(*data: Dict[str, Any]*) → str
Turn data (class parameters) into a JSON string for storing

Parameters **data** – Dictionary of class parameters to be stored. Items that are not JSON serializable will be pickled with Cloudpickle and stored as bytearray in the JSON file

Returns JSON string of the data serialized.

eve.app.utils.**json_to_data**(*json_string: str*, *custom_objects: Optional[Dict[str, Any]] = None*) → Dict[str, Any]
Turn JSON serialization of class-parameters back into dictionary.

Parameters

- **json_string** – JSON serialization of the class-parameters that should be loaded.
- **custom_objects** – Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in *keras.models.load_model*. Useful when you have an object in file that can not be deserialized.

Returns Loaded class parameters.

eve.app.utils.**open_path**(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *mode: str*, *verbose: int = 0*, *suffix: Optional[str] = None*)
Opens a path for reading or writing with a preferred suffix and raises debug information. If the provided path is a derivative of io.BufferedIOBase it ensures that the file matches the provided mode, i.e. If the mode is read ("r", "read") it checks that the path is readable. If the mode is write ("w", "write") it checks that the file is writable.

If the provided path is a string or a pathlib.Path, it ensures that it exists. If the mode is "read" it checks that it exists, if it doesn't exist it attempts to read path.suffix if a suffix is provided. If the mode is "write" and the path does not exist, it creates all the parent folders. If the path points to a folder, it changes the path to path_2. If the path already exists and verbose == 2, it raises a warning.

Parameters

- **path** – the path to open. if save_path is a str or pathlib.Path and mode is "w", single dispatch ensures that the path actually exists. If path is a io.BufferedIOBase the path exists.
- **mode** – how to open the file. "w"|"write" for writing, "r"|"read" for reading.
- **verbose** – Verbosity level, 0 means only warnings, 2 means debug information.
- **suffix** – The preferred suffix. If mode is "w" then the opened file has the suffix. If mode is "r" then we attempt to open the path. If an error is raised and the suffix is not None, we attempt to open the path with the suffix.

Returns

eve.app.utils.**open_path_str**(*path: str*, *mode: str*, *verbose: int = 0*, *suffix: Optional[str] = None*) → io.BufferedIOBase
Open a path given by a string. If writing to the path, the function ensures that the path exists.

Parameters

- **path** – the path to open. If mode is "w" then it ensures that the path exists by creating the necessary folders and renaming path if it points to a folder.
- **mode** – how to open the file. "w" for writing, "r" for reading.
- **verbose** – Verbosity level, 0 means only warnings, 2 means debug information.

- **suffix** – The preferred suffix. If mode is "w" then the opened file has the suffix. If mode is "r" then we attempt to open the path. If an error is raised and the suffix is not None, we attempt to open the path with the suffix.

Returns

eve.app.utils.**open_path_pathlib**(*path: pathlib.Path*, *mode: str*, *verbose: int = 0*, *suffix: Optional[str] = None*) → io.BufferedIOBase

Open a path given by a string. If writing to the path, the function ensures that the path exists.

Parameters

- **path** – the path to check. If mode is "w" then it ensures that the path exists by creating the necessary folders and renaming path if it points to a folder.
- **mode** – how to open the file. "w" for writing, "r" for reading.
- **verbose** – Verbosity level, 0 means only warnings, 2 means debug information.
- **suffix** – The preferred suffix. If mode is "w" then the opened file has the suffix. If mode is "r" then we attempt to open the path. If an error is raised and the suffix is not None, we attempt to open the path with the suffix.

Returns

eve.app.utils.**save_to_zip_file**(*save_path: Union[str, pathlib.Path, io.BufferedIOBase]*, *data: Optional[Dict[str, Any]] = None*, *params: Optional[Dict[str, Any]] = None*, *pytorch_variables: Optional[Dict[str, Any]] = None*, *verbose: int = 0*) → None

Save model data to a zip archive.

Parameters

- **save_path** – Where to store the model. if save_path is a str or pathlib.Path ensures that the path actually exists.
- **data** – Class parameters being stored (non-PyTorch variables)
- **params** – Model parameters being stored expected to contain an entry for every state_dict with its name and the state_dict.
- **pytorch_variables** – Other PyTorch variables expected to contain name and value of the variable.
- **verbose** – Verbosity level, 0 means only warnings, 2 means debug information

eve.app.utils.**save_to_pkl**(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *obj: Any*, *verbose: int = 0*) → None

Save an object to path creating the necessary folders along the way. If the path exists and is a directory, it will raise a warning and rename the path. If a suffix is provided in the path, it will use that suffix, otherwise, it will use '.pkl'.

Parameters

- **path** – the path to open. if save_path is a str or pathlib.Path and mode is "w", single dispatch ensures that the path actually exists. If path is a io.BufferedIOBase the path exists.
- **obj** – The object to save.
- **verbose** – Verbosity level, 0 means only warnings, 2 means debug information.

eve.app.utils.**load_from_pkl**(*path: Union[str, pathlib.Path, io.BufferedIOBase]*, *verbose: int = 0*) → Any

Load an object from the path. If a suffix is provided in the path, it will use that suffix. If the path does not exist, it will attempt to load using the .pkl suffix.

Parameters

- **path** – the path to open. if save_path is a str or pathlib.Path and mode is "w", single dispatch ensures that the path actually exists. If path is a io.BufferedIOBase the path exists.
- **verbose** – Verbosity level, 0 means only warnings, 2 means debug information.

eve.app.utils.**load_from_zip_file**(*load_path: Union[str, pathlib.Path, io.BufferedIOBase]*, *load_data: bool = True*, *device: Union[torch.device, str] = 'auto'*, *verbose: int = 0*) → Tuple[Optional[Dict[str, Any]], Optional[Dict[str, torch.Tensor]], Optional[Dict[str, torch.Tensor]]]

Load model data from a .zip archive

Parameters

- **load_path** – Where to load the model from
- **load_data** – Whether we should load and return data (class parameters). Mainly used by 'load_parameters' to only load model parameters (weights)
- **device** – Device on which the code should run.

Returns Class parameters, model state_dicts (aka "params", dict of state_dict) and dict of pytorch variables

eve.app.utils.**get_trained_models**(*log_folder: str*) → Dict[str, Tuple[str, str]]

Parameters **log_folder** – (str) Root log folder

Returns (Dict[str, Tuple[str, str]]) Dict representing the trained agent

eve.app.utils.**get_saved_hyperparams**(*stats_path: str*, *norm_reward: bool = False*, *test_mode: bool = False*) → Tuple[Dict[str, Any], str]

Parameters

- **stats_path** –
- **norm_reward** –
- **test_mode** –

Module contents

eve.core package

Submodules

eve.core.eve module

eve.core.layer module

eve.core.node module

eve.core.quan module

eve.core.quantize_fn module

4.1.2 Module contents

PYTHON MODULE INDEX

e

INDEX

F

G

H

I

S

T

U

V

W

Z