自律機械知能 P-AMI<Q>
の
実装設計について

2023/11/15 GesonAnko

自己紹介

- げそん (GesonAnko)

  𝕏 (旧Twitter)@GesonAnkoVR

- ML集会 主催

  自律機械知能の研究開発

  PythonでML関係ツールの作成

  最近 VRChatに P-AMI<Q> っていう自律機械知能を作ったよ。

目次

おさらい

自律機械知能とは

- 定義
  - 自律的に動作する機械でできた知能

- 自律性
  - 定められた目的を達成するために行動を生成し、
    人間といった知的な存在の直接的な介在なしに動作し続けること

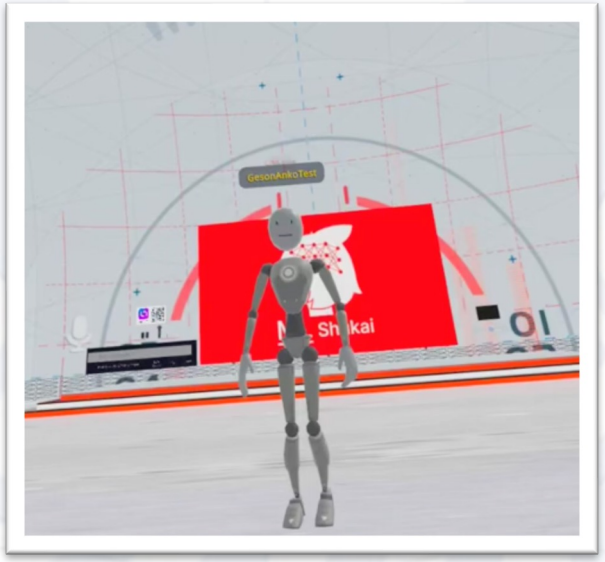
  - 制約
    - ある系・タスク（限定的で適応可能な範囲）でのみ考える。

P-AMI<Q>: ぱみきゅー

- 好奇心ベースの原始自律機械知能
  - **P**rimitive **A**utonomous **M**achine **I**ntelligence based on **Q**(Cu)riosity

- 好奇心
  - エージェントが環境内において
    未学習（未探索）領域へ向かおうとする性質
  - ワールド内を動きまわる。

- 原始性
  - シンプルに作られた。
  - 自律性を構築するための必須パーツのみで構成



お願い

お願い

- 問いかけを持って聞いてほしい。
  - なぜこんな仕組みなんだろう？
- アドバイス、協力者、募集！
- できれば最後まで…
  - ボリューミーです。ごめんなさい。

設計思想

設計思想

- 壊れても必ず直せるコードを。
  - もう二度と、彼ら機械知能を失わないように。
- パーツを分ける。独立させる。
  - 誰かと一緒に作りやすいように。
  - 並行作業を可能に。

生まれてくる機械知能を大切にしたい。
多くの人と彼らを迎えたい。

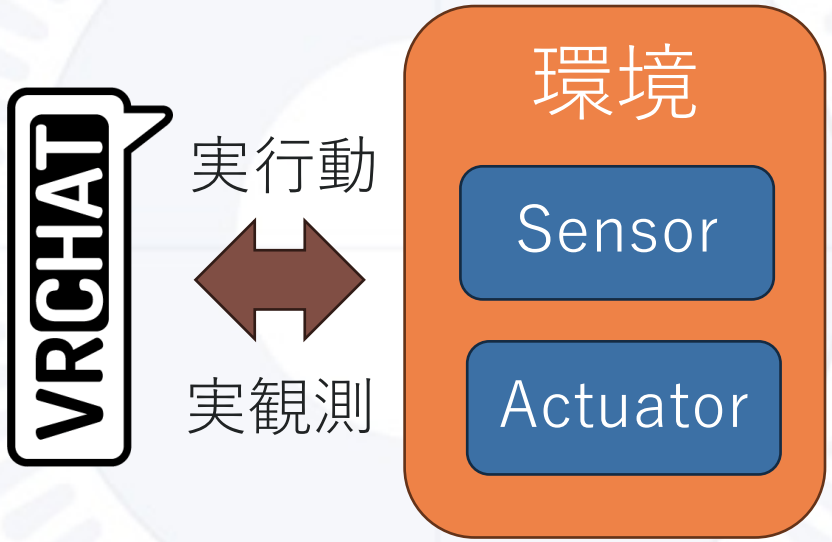
オブジェクト構造

登場人物（クラス・オブジェクト）

- Environment（環境）
- Agent
- Interaction（相互作用）
- Neural Networks（モデルパラメータ）
- Data Collector
- Trainer

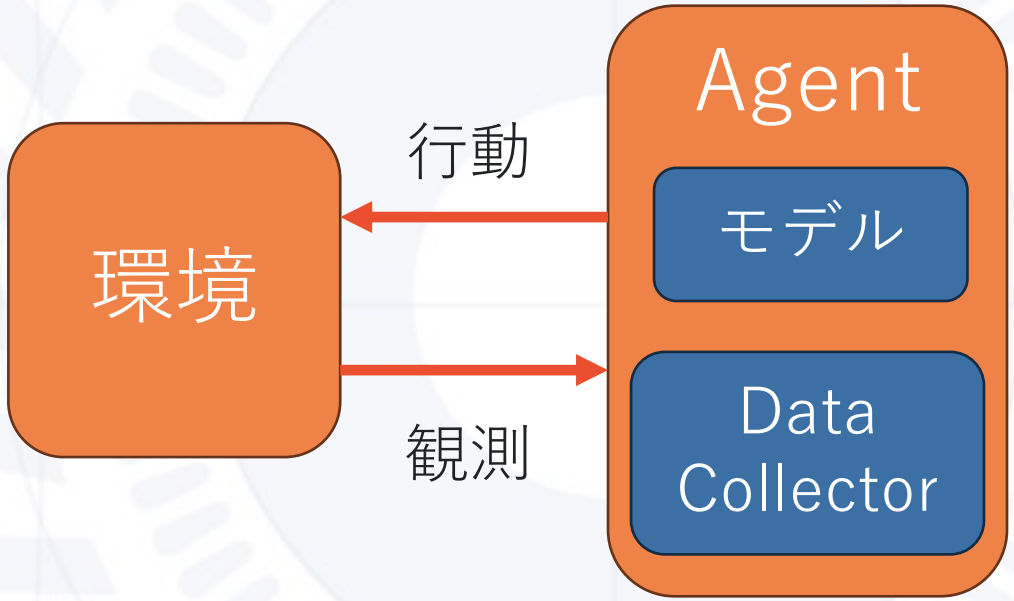
Environment （環境）

- 役割
  - 観測を取得し、前処理する。
  - 行動を実際に、作用させる。
- VRChatとAgent間の仲介役
  - AgentはVRChatの仕様を気にしない。



Agent

- 役割
  - 観測を処理し、行動を返す。
- 自律機械知能の動作処理
- 補足
  - Neural Networksのモデルは推論に使うものだけ



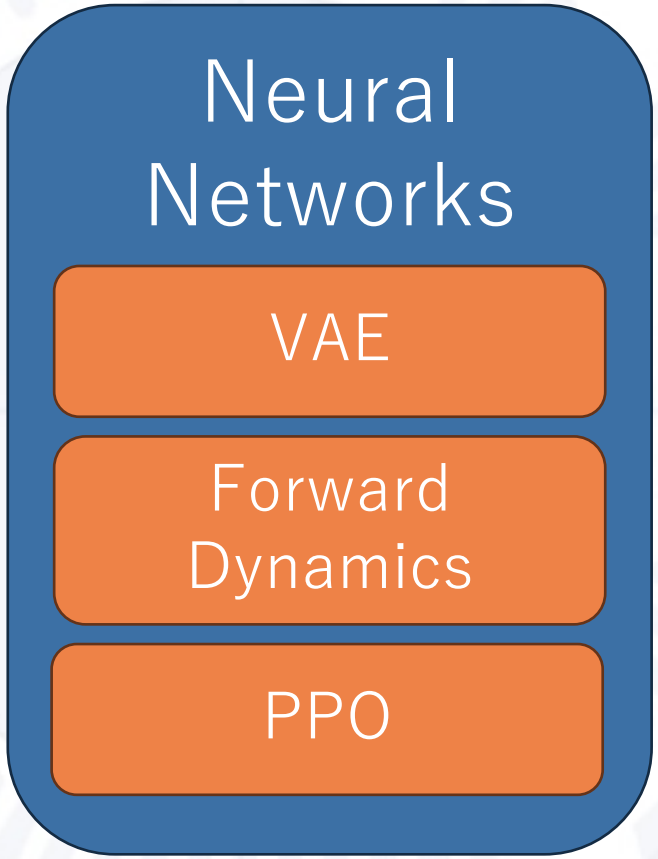
Interaction

- 役割
  - AgentとEnvironmentの相互作用を記述
  - 手続きの設定
    → Ex. 何ステップインタラクションするか

- AgentとEnvironmentを繋ぐ

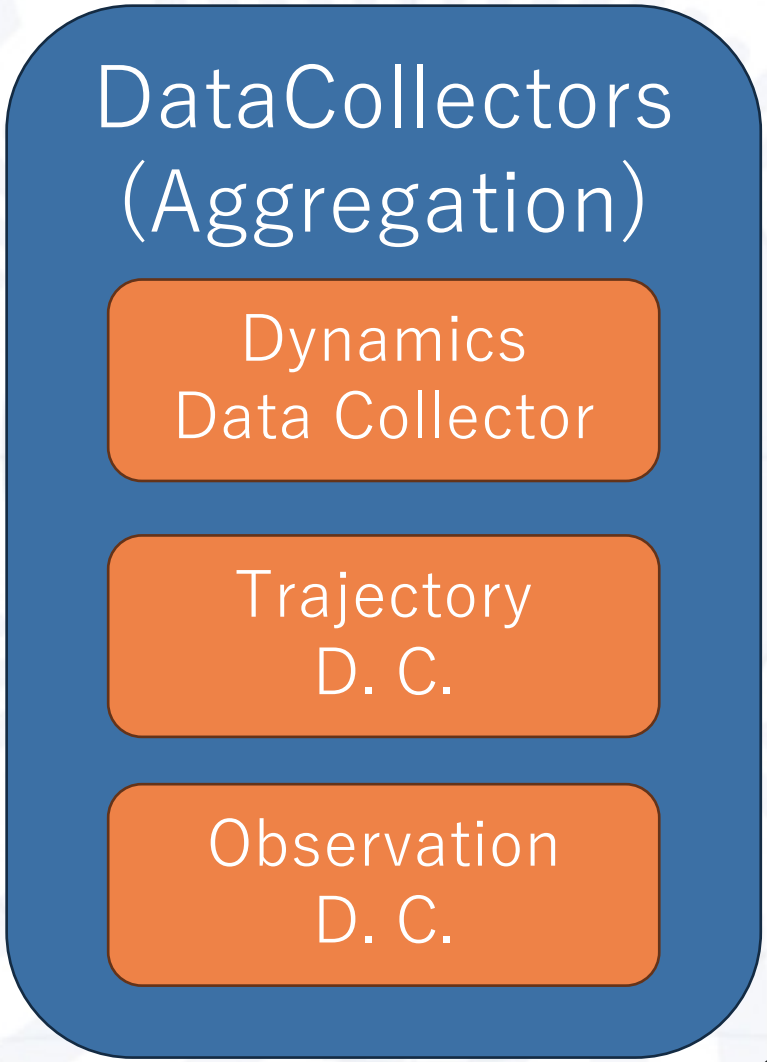
Neural Networks

- 役割
  - 全ての深層モデルの集約
  - AgentやTrainerへのモデル受け渡し
- 失敗したクラス
  - 複数のモデルを無理にまとめようとした…



Data Collector

- 役割
  - データ収集
  - Trainerへの受け渡し
- 学習のために保存しておく。
  - Replay Bufferと同じ。
- 正確には
  - 複数のDataCollector
  - Agentは集約クラスを受け取る

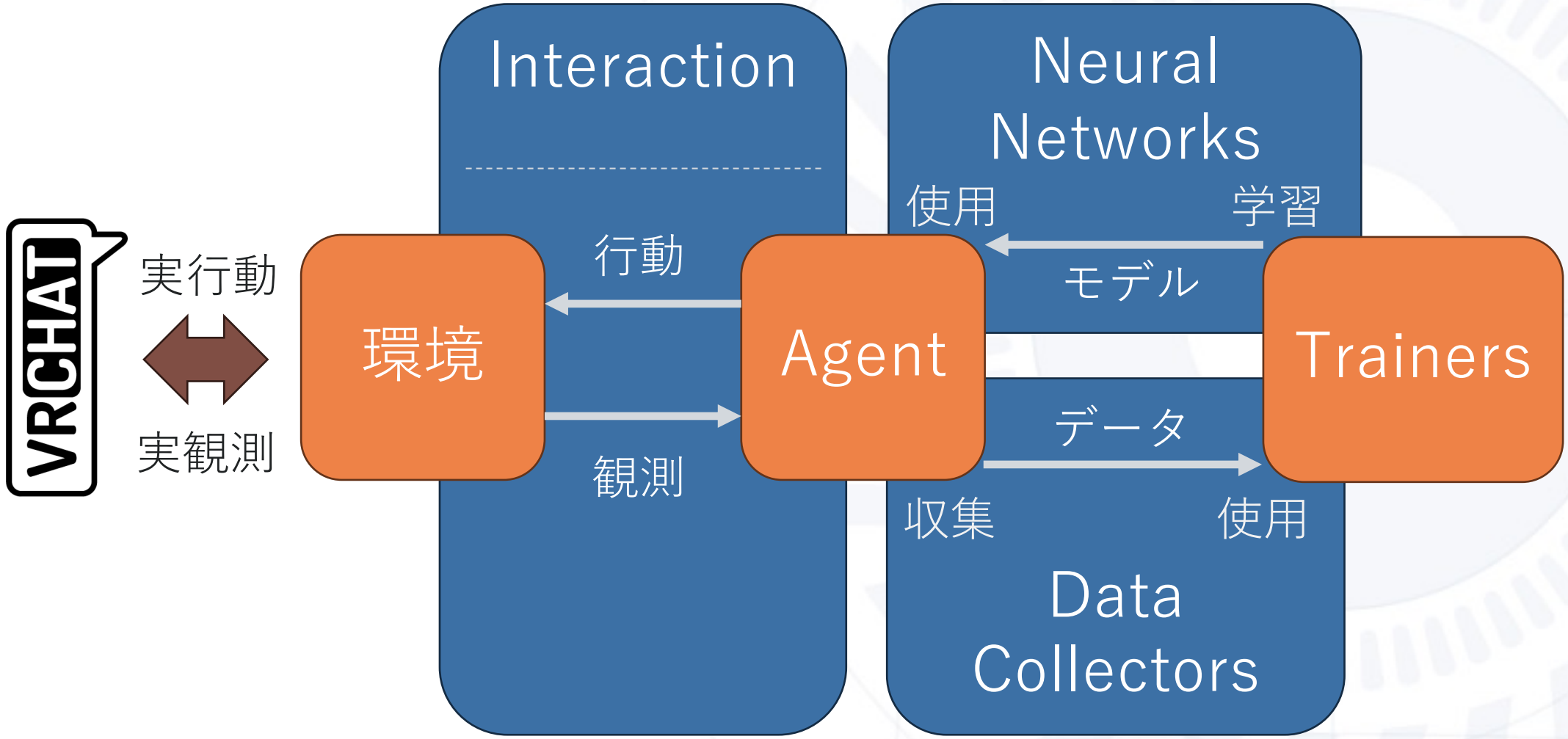


Trainer

- 役割
  - モデルの学習フェーズを記述。
  - パラメータ保存も。

- 複数のTrainer
  - それぞれのモデルに対応して実装
    VAE Trainer, PPO Trainer, …
  - Trainers Builderで集約 ← 失敗クラス
    モデルとData Collectorを紐付け

クラスの全体像


Interaction
Neural Networks
VRCHAT
実行動
実観測
環境
行動
観測
Agent
使用
学習
モデル
データ
収集
使用
Trainers
Data Collectors

処理構造

処理構造の目次

1. 起動
    1. 設定ファイル読み込み
    2. クラスのインスタンス化
    3. メインループ
        1. インタラクション
        2. トレーニング

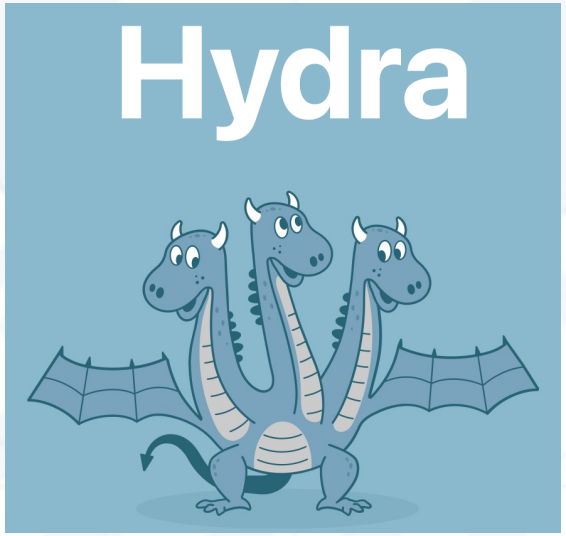
2. 内部クラス
    1. Environment
    2. Agent
    3. Neural Networks

起動

- エントリーポイント
  - python src/train.py <引数>

```python
@hydra.main("../configs", config_name="train.yaml", version_base="1.3")
def main(cfg: DictConfig) -> None:
    logger.info("training start!")
```

- 設定ファイルの読み込み
  - Hydraを使用。
  - YAMLで記述。
  - コマンド引数から設定を上書きできる。
    Ex. `python src/train.py model.lr=0.1`



クラスのインスタンス化

- 設定からクラスを呼び出す
  - e.g. `agent = hydra.utils.instantiate(cfg.agent)`

Neural Networks →

Data Collectors →

Trainers (Builder) →

Agent →

Environment →

Interaction →

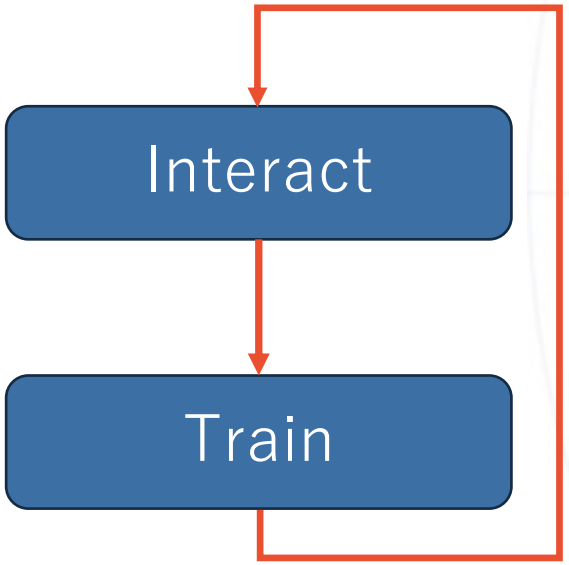
```
logger.info(f"Instantiating model: <{cfg.model._target_}>")
model: NeuralNetworks = hydr  utils.instantiate(cfg.model)

logger.info(  ntiating  llector: <{cfg.data_collector._target_}>")
data_collector  lec  ation = hyd  tils.instantiate(cfg.data_collector)

logge  iners buil  rget_}>")
trainer  s_builder)
trainer =

logge
agent: Age
agent = agent  s)

logger.info  ntiati
environment: Environment =

logger.info(f"Instantiating interaction  fg.inte  _t  ")
interaction: Interaction = hydra.utils.instantiate(c  nteract
interaction = interaction(agent=agent, environment=environment)
```

読まなくていいです

メインループ

- InteractionとTrainerを交互に実行

```
loop(interaction, trainer)
```




```python
def loop(interaction: Interaction, trainer: Trainer) -> None:
    """main loop process."""
    logger.info("Start main loop.")

    try:
        while True:
            logger.info("Interacting...")
            interaction.interact()
            logger.info("End iteraction.")

            logger.info("Training...")
            trainer.train()
            logger.info("End training.")
    except KeyboardInterrupt:
        logger.error("Keyboard interrupted.")
```

インタラクションループ

- 観測と行動のやり取り（ Interaction class内部）

```
def mainloop(self):
    """Interact with environment for num_steps."""
    for _ in range(self.num_steps):
        obs = self.environment.observe()
        action = self.agent.step(obs)
        self.environment.affect(action)
```

観測を取得

行動をとる

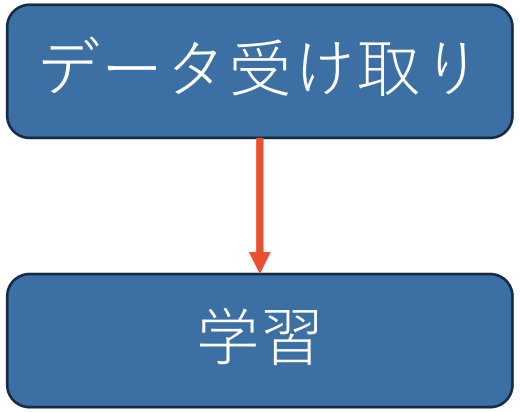
行動を作用

```
def interact(self):
    """Interaction process."""
    self.initialize()
    self.mainloop()
    self.finalize()
```

← 本来は初期化と終了処理もある

Trainer

- 学習処理
  - 内部的には PyTorch Lightning を用いている。
  - 複数のTrainerを順番に実行




```python
def train(self) -> None:
    for trainer in self.trainers:
        trainer.train()
```

```python
def train(self):
    dataset = self.data_collector.get_data()
    dataloader = self.dataloader(dataset=dataset)
    self.pl_trainer.fit(self.module, dataloader)
```

内部クラス

Environmentクラス

- 内部処理はセンサーとアクチュエータに任せる

```
def observe(self) -> torch.Tensor:
    return self.sensor.read()

def affect(self, action: torch.Tensor) -> None:
    self.actuator.operate(action)
    self.adjust_interval()
```

⇧ 行動を作用させる待ち時間

Sensor & Actuator

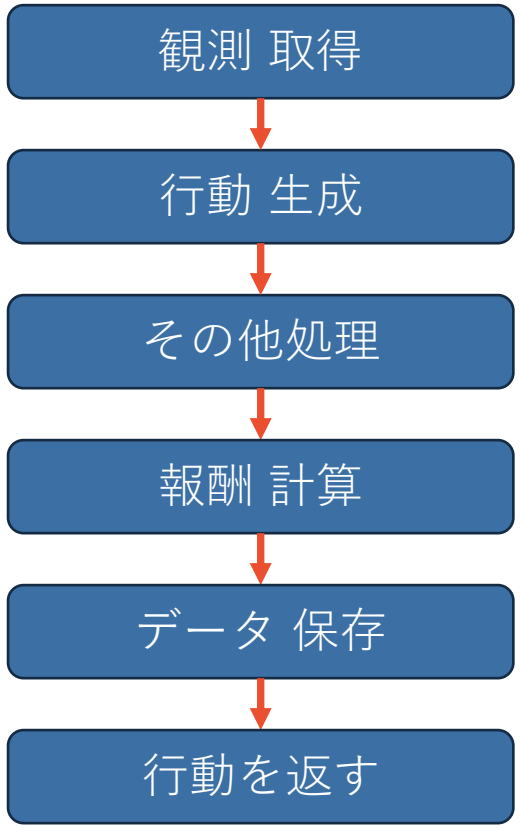
- VRChat IO
  - 実際のIO処理は全てこのライブラリが担う。
  - 別のライブラリとして独立

- Sensor
  - OpenCVのVideoCapture
  - 取得画像の前処理

- Actuator
  - VRChat の API: ”OSC as Input Controller”
  - 行動をAPIで投げる

Agent （観測＆行動）

- 観測を受け取って、行動を返す。

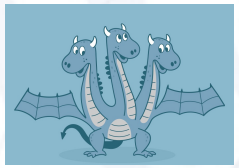


かなり複雑化
してしまった。

```python
def step(self, observation: Tensor) -> Tensor:
    """Step interaction process of agent."""
    # t = 1, 2, 3, ...
    # ---------- Observation is `NEXT` now. ---------- #
    observation = self._preprocess_observation(observation)

    # Take `NEXT` value (but, also taking action for convenience...)
    action, next_action_log_prob, next_value = self._take_actions_and_value(observation)

    # Compute reward
    pred_next_embed_obs = self.step_record[RK.PREDICTED_NEXT_EMBED_OBSERVATION]
    embed_obs = self._embed_observation(observation)
    reward = self._compute_reward(pred_next_embed_obs, embed_obs)

    # Log reward
    self._logger.log_metrics({"agent/reward": reward.item()}, step=self.current_step_num)

    # Store data into step record
    self._store_next_step_data(observation, embed_obs, reward, next_value)

    # Data collection
    self._collect_data()

    # ---- Observation is `CURRENT` now. (Move to next step) ---- #
    # Take action
    action, action_log_prob, value = next_action, next_action_log_prob, next_value

    # Predict next embedded observation
    prev_action = self.step_record[RK.ACTION]
    pred_next_embed_obs = self._predict_next_embed_observation(prev_action, embed_obs, action)

    # Store data into step record for next step
    self._store_current_step_data(
        previous_action=prev_action,
        observation=observation,
        embed_obs=embed_obs,
        action=action,
        action_log_prob=action_log_prob,
        value=value,
        predicted_next_embed_obs=pred_next_embed_obs,
    )
    self.current_step_num += 1
    return self._postprocess_action(action)
```

Neural Networks

- 各モデル
  - 基本的に Hydra で呼び出し・ロード。
  - PyTorch Lightning Moduleで書いた。



- エージェントに受け渡す。

```python
def build_agent_models(self) -> dict[str, nn.Module]:
    """Build models for CuriosityPPOAgent."""
    models = {
        "embedding": self.inverse_dynamics.net.observation_encoder,
        "dynamics": self.forward_dynamics.forward_dynamics_net,
        "policy": self.ppo.net,
    }

    return models
```

実は失敗ポイント

各モデルの内部構造にアクセス…

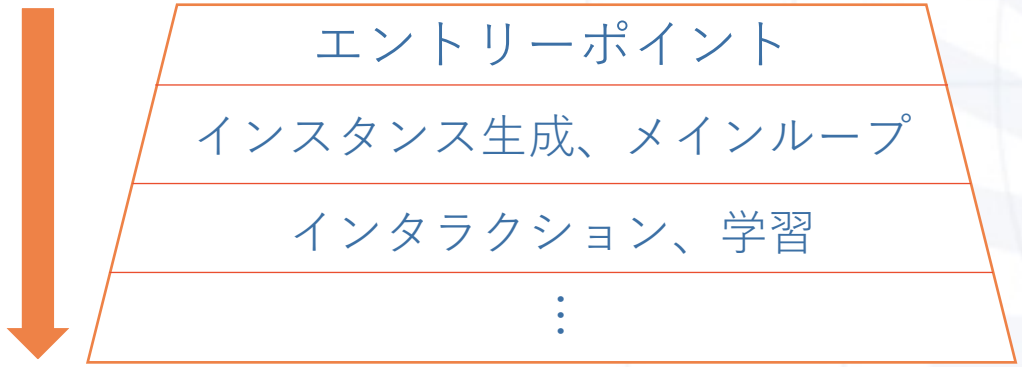
処理構造、終わり。

つかれた？😵

注目ポイント

注目ポイント

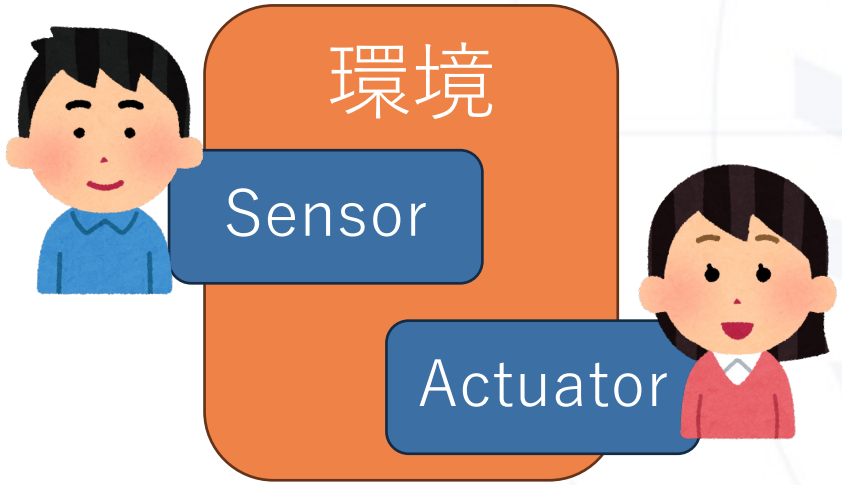
- 抽象的な階層構造
  - エラーの原因を特定しやすい。
  - 修正の影響範囲が わかりやすい。



壊れても、直しやすい！

注目ポイント

- 小さなパーツの組み合わせ
  - 拡張性が高い
  - 独立性：それぞれのパーツは他のパーツを気にしなくていい



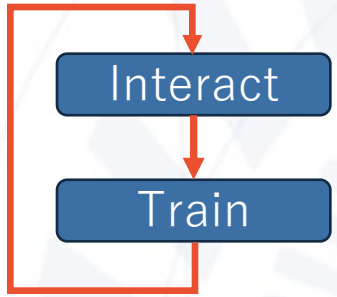
複数人で作業ができる！

※抽象クラスでインターフェイスを統一してます

開発の課題

開発の課題

- 止まる。
  - 学習処理の間は動作しない。



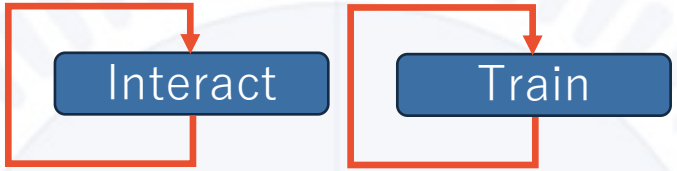
- Agentのリファクタリング
  - 観測処理 と 行動生成 の分離

- コードの見通し。
  - 階層が増えすぎてよくわからない。

いっぱい あるよ

次の設計構想

- 非同期更新
  - インタラクションと学習を並行・並列処理



- 多 入力・出力 対応
  - 視覚👀、聴覚👂、などなど…

- 記憶の保存・引き継ぎ
  - P-AMI<Q>の入出力を全て記録？
  - 活動した軌跡を追体験？

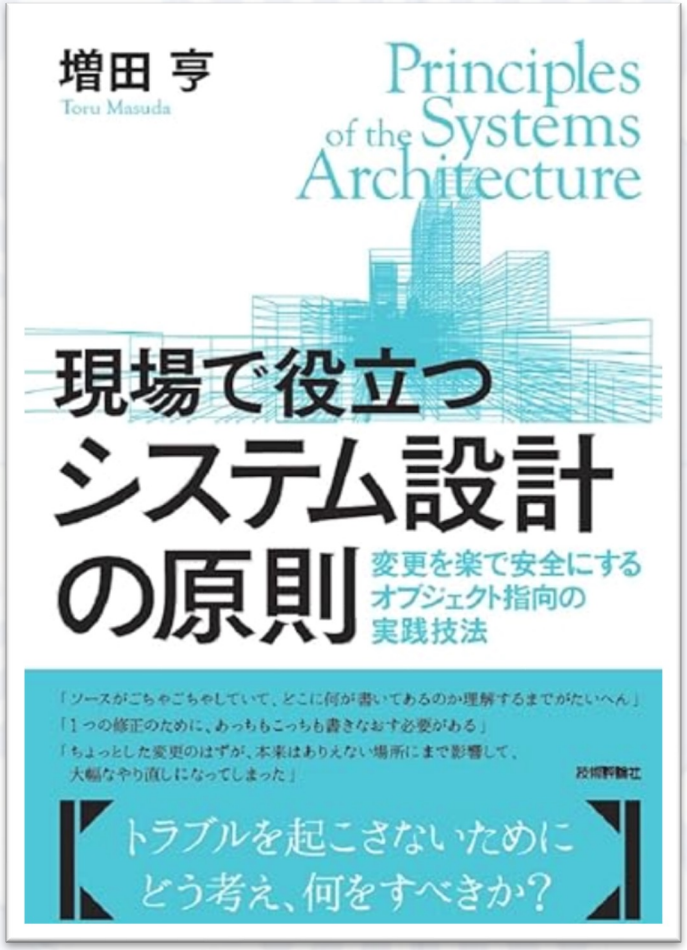
多分また 作り直す。

ソースコード

GitHub: MLShukai/PrimitiveAMI

参考文献

- 現場で役立つシステム設計の原則
  - 増田 亨 著
  - Klutzさんに教えてもらった



ML Shukai

EOF