# Starting Kit Guidance

This starting kit will guide you through the workflow of developing a solution for an offline RL problem. It contains two components:

1. `baseline` : an example solution of all relative procedures to build an effective offline RL model: preprocess offline data, build a virtual environment, and train a policy based on the virtual environment.

2. `sample_submission` : a ready-to-submit bundle based on the baseline model. You can directly submit the `sample_submission.zip` downloaded along with starting kit, or replace the parameters file with your own locally trained baseline model.

By running `run_baseline.sh` in a bash-compatible shell, all these procedures will run sequentially and participants will obtain a usable baseline model along with an up-to-date sample submission bundle.

You may use the baseline model and sample submission code as the entrypoint to your own model implementation. The baseline is provided with `revive` SDK and `stablebaselines3` library, but there is no restriction on how your solution is implemented.

## Requirements

1. `Linux x86_64` . Revive SDK requires this, participants on Windows machine could use WSL2 and seamlessly develop on it with `VSCode` + `Remote WSL extension` . GPU is also supported on WSL2 in Windows 11 or a newer Windows 10 build.

2. `Python 3.6, Python 3.7 or Python 3.8` . Revive SDK does not support Python 3.9 yet, so you may create a py38 environment with conda for baseline model. **Note**: the evaluation program on the competition platform runs on Python 3.9, so feel free to use the latest Python features in your own model.

3. Since a non-ascii file path (e.g. path containing Chinese characters) will trigger `UnicodeEncodeError` from some library as reported by participants, it is recommended to store this starting kit in a path only containing ASCII characters.

## Data organization

All data files related to baseline model will be saved and organized in `baseline/data` folder, including:

1. Public data
   - `offline_592_1000.csv` : Offline dataset downloaded from public data in development phase, containing data of 1000 customers in 60 days.
2. Preprocessing data

- `offline_592_3_dim_state.csv` : Processed offline dataset with user state inserted as a 3-dim vector ( `total_num` , `average_num` , `average_fee` ).
- `user_states_by_day.npy` : State of all users in `60-x` days, with data in first `x` days reduced as initial state. `x` is set to 30 in this baseline.
- `evaluation_start_states.npy` : The last day's states in the offline dataset, where online evaluation will start from this day.

3. Train a virtual environment
   - `license.lic` : License file required by Revive SDK. Downloaded along with `revive-sdk-0.5.0.zip` .
   - `venv.yaml` : Metadata of Revive describing the decision flow of Offline RL problem. It is the only file pre-included in baseline data folder.
   - `venv.npz` : Actual dataset corresponding to the layout described in `venv.yaml` . It is generated in data preprocessing phase.
   - `venv.pkl` : Trained parameters of virtual environment. It is generated after running Revive for training virtual environment.
4. Train a policy model
   - `model_checkpoints/rl_model_*_steps.zip` : Checkpoints of trained paramters of policy validation model.
   - `rl_model.zip` : The final policy model chosen to be submitted.

**Note**: `evaluation_start_states.npy` and `rl_model.zip` will be copied to `sample_submission/data` folder as the parameters for baseline model in online policy evaluation.

Below steps in this notebook is majorly a breakdown of `run_baseline.sh` . Let's start with defining the baseline root envrionment variable:

In [ ]:
```
import os

baseline_root = f"{os.getcwd()}/baseline"

%env BASELINE_ROOT=$baseline_root
```

Append the baseline root to Python module search path:

In [ ]:
```
import sys

sys.path.append(baseline_root)
```

And install some basic requirements for this baseline:

In [ ]:
```
!python3 -m pip install -r $BASELINE_ROOT/requirements.txt
```

# Step 1: Derive user states from offline data

To begin with, switch to the data directory and download the public data from development phase:

In [ ]:
```
%pushd $baseline_root/data
!wget -q https://codalab.lisn.upsaclay.fr/my/datasets/download/eea9f5b7-3933-47cf-
```

```
!unzip -o public_data_dev.zip
%popd # Leave data directory
```

The public offline dataset contains only the company promotion action, the user response action, and metadata like index, step, date. We need to derive a user's state (the depiction of a user) from this offline dataset, and use it to build our virtual environment model.

As a baseline, here we define the user state as a simple 3-dim features:

| Feature | Description |
|---|---|
| total_num | The total number of orders in the user's history |
| average_num | The average number of per-day orders in the user's history (days of no order is not counted) |
| average_fee | The average fee of per-day orders in the user's history (days of no fee is not counted) |

Beware that **defining proper user states is key to an effective virtual environment**. States defined in above table is easy and straightforward, but it is necessary for participants to define more robust and reasonable user states.

Note that this is a recurrent state definition, meaning a user's state in a certain day is an accmulated result of all days in this user's history. Therefore, user state at day 31 is the reduced state of all previous 30 days.

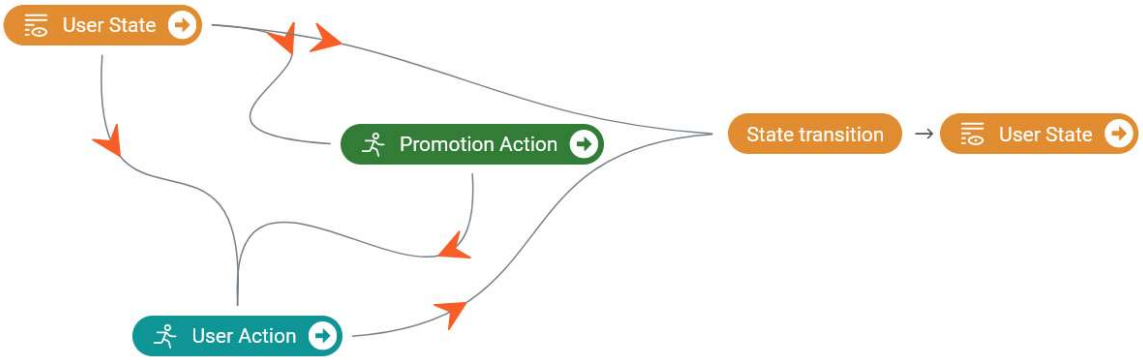
Based on this property, to keep the baseline simple, we use the state at day 31 as initial state, representing the user data collected in first 30 days. Then we can proceed to learn a virtual environment based on the transition from day 31 to day 60 (It should be noticed that, this assumption does not fit the fact that user states is influenced by different promotion actions).

Above analysis is implemented in `data_preprocess.py`, we could run it to transform public data `offline_592_1000.csv` to processed data `offline_592_3_dim_state.csv`, `user_states_by_day.npy`, `evaluation_start_states.npy` and `venv.npz`:

In [ ]:
```
%pushd $baseline_root/data
import numpy as np
from data_preprocess import data_preprocess

offline_data_with_states, user_states_by_day, evaluation_start_states, offline_dat
print(offline_data_with_states.shape)
print(user_states_by_day.shape)
print(evaluation_start_states.shape)
offline_data_with_states.to_csv('offline_592_3_dim_state.csv', index=False)
np.save('user_states_by_day.npy', user_states_by_day)
np.save('evaluation_start_states.npy', evaluation_start_states)
np.savez('venv.npz', **offline_data_with_states_npz)
%popd
```

# Step 2: Learn a virtual environment

Below shows the decision flow graph of our Offline RL problem:




Here, `User State` is defined by us, and `Promotion Action` is to be implemented in our submission policy, so for the entire flow to work we need to mock the `User Action` as a virtual envrionment. This is equivalent of learning a user policy model from offline dataset that can output user action from user states and promotion action.

In baseline we learn such a user policy (virutal environment) with Revive SDK. First download the SDK with git:

In [ ]:
```
%pushd $baseline_root/data
!git clone https://agit.ai/Polixir/revive.git
%popd
```

Here Revive SDK is extracted as `baseline/revive`, with license file to be `baseline/data/license.lic`. Now prepare the environment for Revive SDK by:

1. Install requiremented dependencies with `pip`.
2. Revive requires a license file's location set correctly within an environment variable `PYARMOR_LICENSE`.

In [ ]:
```
%pushd $baseline_root/revive
!git checkout 0.5.0
!python3 -m pip install -e .
%env PYARMOR_LICENSE=$baseline_root/data/license.lic
%popd
```

Now the environment for training venv is ready. Before running Revive, we may tune Revive's `config.json` to best suit our needs, such as number of trails in venv training, and the metric used to evaluate our virtual environment. For more about the usage, refer to Revive's documentation:

In [ ]:
```
import json

def update_config(configs, name, default_value):
    for config in configs:
        if config["name"] == name:
            config["default"] = default_value
            return

with open(f"{baseline_root}/revive/data/config.json", 'r') as f: # Write config.js
    config = json.load(f)
```

```
base_config = config["base_config"]
update_config(base_config, "train_venv_trials", 20)
update_config(base_config, "venv_metric", "wdist") # Use w-distance as metric, whi

with open(f"{baseline_root}/data/config.json", 'w') as f: # Write config.json to d
    json.dump(config, f, indent=2)
```

Then we could start learning the virtual environment:

- The `venv.yaml` contains decision flow graph definitions like the image above.
- The `venv.npz` contains the actual dataset, with each field attached to a certain node of decision flow graph described in `venv.yaml`.
- The `--policy-mode` is set to `None` to only train the virtual environment. The policy training is delegated to `stablebaselines3` library in next section.

In [ ]:
```
%pushd $baseline_root/data
# Only train minimum number of trails (3) so notebook will not be blocked for so l
!python $BASELINE_ROOT/revive/train.py -rcf config.json --data_file venv.npz --cor
%popd
```

The training log is located at `baseline/revive/logs/venv_baseline/`, with the best target model parameters file to be `venv_baseline/env.pkl`.

**NOTE**: To earlier obtain a baseline model, you may stop this training cell as long as an available `env.pkl`, or a `TorchTrainable` folder in `venv_tune` is generated. Actual venv training may be put as background process outside of this notebook so as to not block its kernel from subsequent commands.

# Evaluate the virtual environment

During virtual environment training, multiple venv parameters in different trails are generated. It is important for us to select a best trained environment for subsequent policy learning, or else the policy will be trained on a environment far different from the real one.

## (a) Automatic evaluation with specific metrics

Revive uses a specific metric to automaticlly evaluate the environment, sort them in metric ascending order, and choose the environment with least metric as the best one. This behavior can be checked in `train_venv.json`:

In [ ]:
```
%pushd $baseline_root/revive/logs/venv_baseline
!cat train_venv.json
%popd
```

Here, the `metrics` field lists all trails with their metric value, accuracy, and its corresponding trail subfolder. The trails in `metrics` are sort in metric ascending order, with `best_id` field to be the first trail in `metrics`.

The specific metric to use is defined in `venv_metric` field in `config.json`, including:

- `mae`: Mean Absolute Error, computed between expert data and **1-step rollout from expert data**

- `mse` : Mean Square Error, computed between expert data and **1-step rollout from expert data**
- `nll` : Negative Log Likelihood, computed between expert data and **1-step rollout from expert data**
- `wdist` : Wasserstein Distance, computed between expert data and **Multi-step rollout from expert data**
- `shooting_mae` : Mean Absolute Error, computed between expert data and **Multi-step rollout from expert data**
- `shooting_mse` : Mean Square Error, computed between expert data and **Multi-step rollout from expert data**

That is, for `mae` , `mse` and `nll` , the test data is generated by unrolling only 1 step on the expert data, and compute the metric between the test data and expert data; for `wdist` , `shooting_mae` and `shooting_mse` , the test data is generated by rollout many steps defined in `venv_rollout_horizon` field in `config.json` (default to 10).

There are some more detailed difference between a 1-step metric and multi-step metric. But what's important is that for our offline rl problem, it is more suitable to use a multi-step metric to better represent the similarity of offline data and virtual environment in a rollout scope.

Our baseline selects `wdist` as the metric. You could develop your own metric if you found all above not suitable by implementing related code yourself in Revive SDK (Contact organizer for help of reading Revive code if you decided to do this and have problem in understanding Revive source code).

To view a more specific training log, we can select one trail id from `train_venv.json` :

In [ ]:
```
%pushd $baseline_root/revive/logs/venv_baseline

import json

with open("train_venv.json") as f:
    report = json.load(f)

best_id = report["best_id"]

# Change the id to the trail you want to see here
trail_id = best_id

trail_dir = report["metrics"][str(trail_id)]["traj_dir"]

print(f"* --- Trail id {trail_id} in dir {trail_dir} --- *")

%popd
```

And 1) view statistics in `progress.csv` , or 2) view its training curves through tensorboard:

In [65]:
```
%load_ext tensorboard

%tensorboard --bind_all --logdir $trail_dir
```

## (b) Manual evaluation with histogram and rollout image

Sometimes using metric is not enough to choose a reasonable virtual environment out of all trails, and you may need to figure a proper one out yourself by manually examining on all of the trails.

Within each trail dir, Revive also prints some histograms and rollout images to help your decision, which are formed in this way:

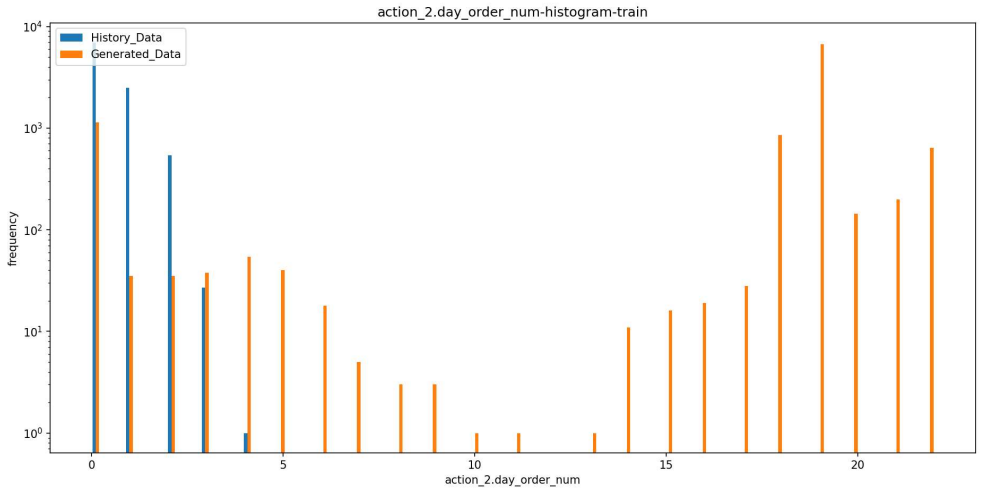
- histogram/
  - action_1.field_1-train/val.png
  - action_2.field_1-train/val.png
  - action_2.field_2-train/val.png
  - next*state.field**-train/val.png
- rollout_images/
  - action_1/
    - 0_action_1.png
    - 1_action_1.png
  - action_2/
    - *_action_2.png
  - next_state/
    - *_next_state.png

In `venv.yaml`, we define `action_2` as the user action, which is the virtual environment to learn. So we mainly focus on `action_2` related images here to determine whether a virtual environment is generating correct data.

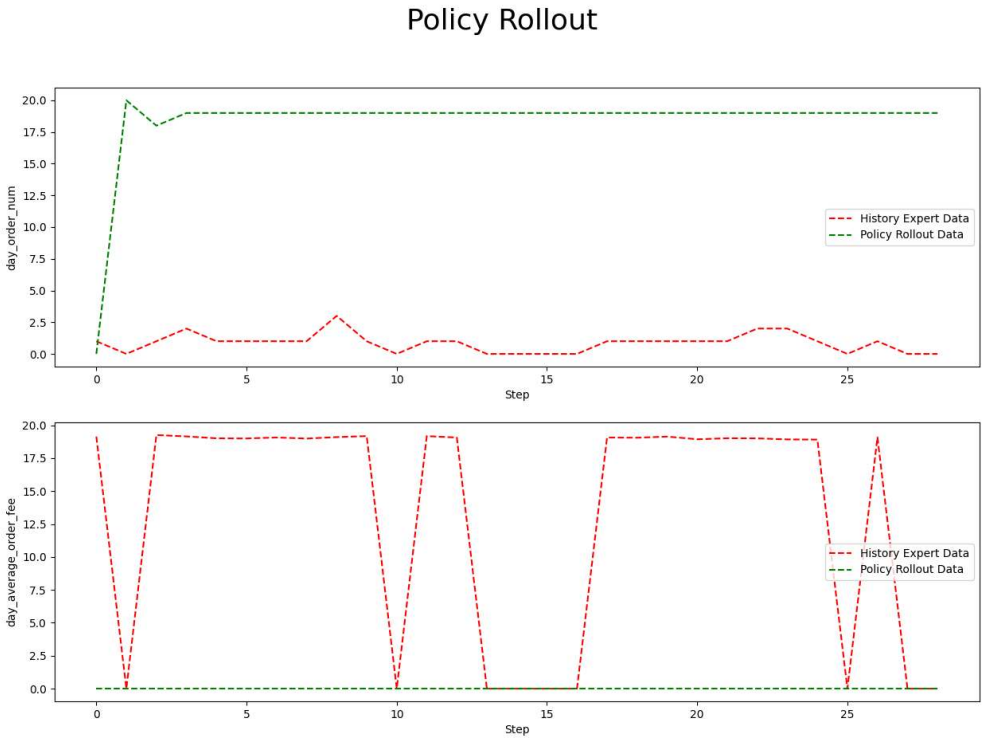
Let us display some images from the trail selected in last section:

In [63]:
```
%pushd $trail_dir

from IPython.display import Image, display

print("Frequency of each day's order num from user:")

display(Image(filename="histogram/action_2.day_order_num-train.png"))

print("User actions from one of the rollout:")

display(Image(filename="rollout_images/action_2/0_action_2.png"))

%popd
```

```
/tmp/starting_kit/baseline/revive/logs/venv_baseline_wdist/venv_tune/TorchTrainabl
e_f914b36e_9_d_lr=0.000183,d_steps=5,g_lr=7.3e-05,g_steps=1,ppo_runs=2_2022-01-02_
16-50-01
Frequency of each day's order num from user:
```




```
User actions from one of the rollout:
```



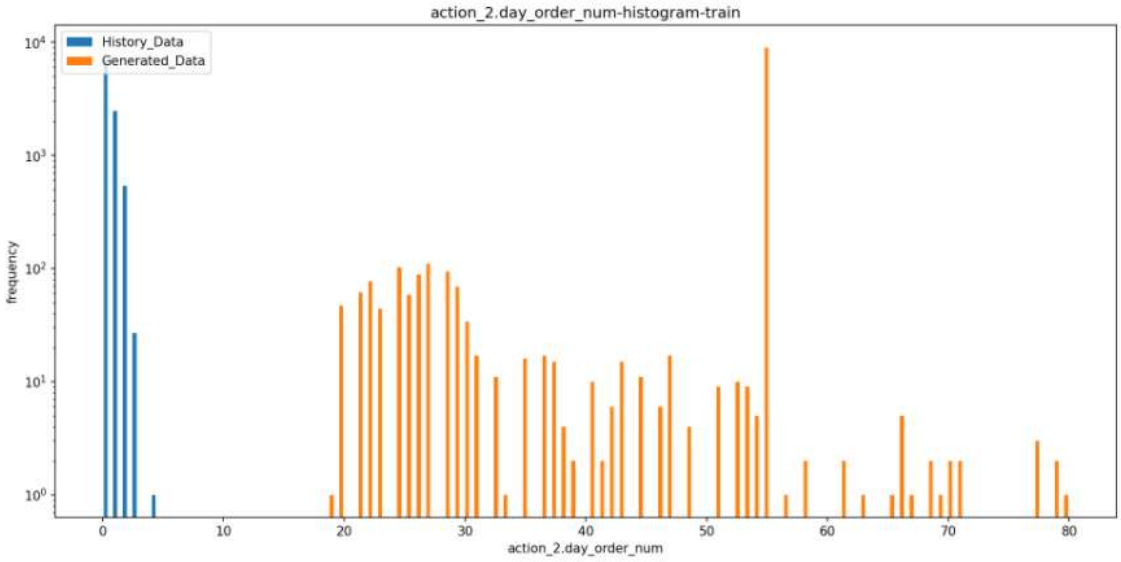

```
/tmp/starting_kit/baseline/data
popd -> ~/Projects/polixir/codalab/codalab-polixir/competition-bundle/starting_ki
t/baseline/data
```

In the first image, the histogram is the frequency of a specific value of action (e.g. The user gives 1, 3, 5 or 10 orders a day), collected from all users in all days.

In the second image, the rollout image is the action sequence generated in a specific rollout.

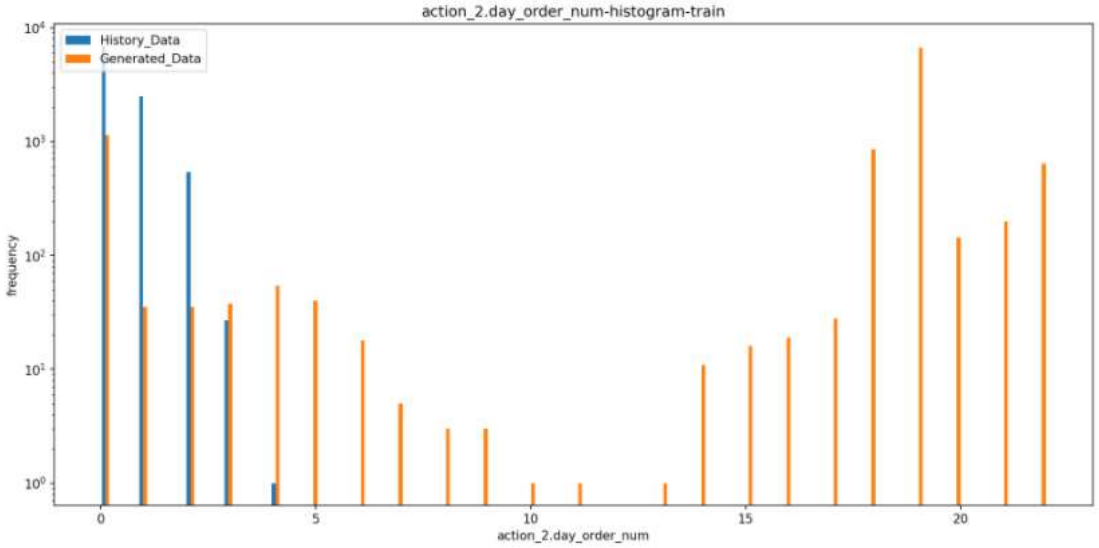
Therfore, these two kinds of images depicts the quality of virtual environment in two ways: 1) the overall user tendency and 2) a specific user's response in multiple days. They are all expected to be simliar with the one generated in expert data, so participants could use them to manually select a good virtual environment.

Here is an example histogram of poor learned virtual environment:




Also expert data have shown that users tend to not place more than 5 orders a day, the virtual environment is learned to predict that user will place more than 20 orders, even to a maximum of 80 orders a day, which will result in a too high reward in policy learning.

Here is another example histogram of relatively well learned virtual environment:




Although this virtual environment is still far from correct than the expert data, this histogram's upper bound (around 22) is nearly the same as the lower bound of the poor learned one (around 19). This policy learning on this venv will generate reasonable reward described in next section.

# Get the model parameters for virtual environment

After obtaining a desired virtual environment, it could be copied to `baseline/data` folder in one of these two ways:

## (a) Use the model with best metric

Model of best metric are saved as `venv_baseline/env.pkl` :

In [ ]:
```
!cp -f $BASELINE_ROOT/revive/logs/venv_baseline/env.pkl $BASELINE_ROOT/data/venv.p
```

### (b) Use the model with specific trail id

Model with specific trail dir are saved as `venv.pkl`. We use the trail dir selected in last section:

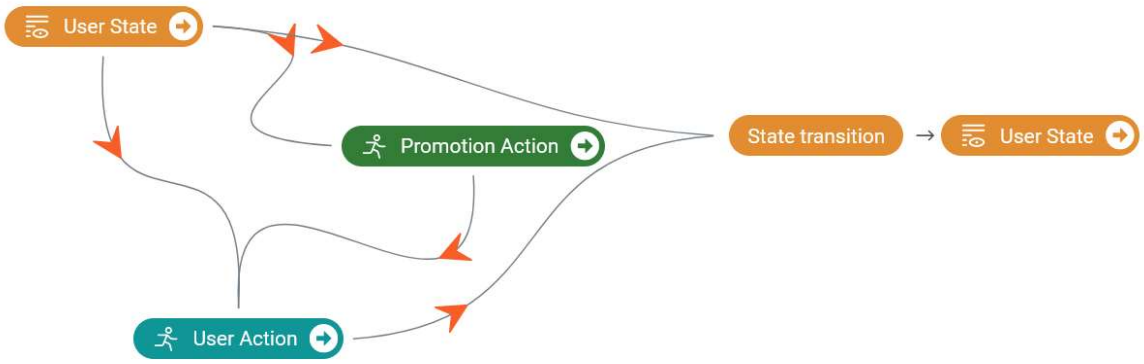
In [ ]:
```
%env TRAIL_DIR=$trail_dir

!cp -f $TRAIL_DIR/venv.pkl $BASELINE_ROOT/data/venv.pkl
```

# Step 3: Learn a fair promotion policy from virtual environment

After learning a virtual environment, we could then get started to learn a fair promotion policy based on it. An unfair promotion policy takes individual user state as input, then outputs discriminated promotion action to every individual user. On the contrary, to learn a fair policy, the input should be the states of the entire user community (such as all users in a city), and output same promotion action to all users.

## Environment and MDP Setup

The virtual environment we have learned in Step 2 is actually a decision graph. Take a view at the decision graph again:




To compute a specific node on this graph, values of its ingress nodes are required. Therefore, as shown in the graph, to predict user's response action, we should provide current user's state and our promotion action.

Here, we take the state from the initial states processed in Step 1, and chooses sending no coupon as our action:

In [ ]:
```
import numpy as np

# Use user states from first day (day 31) as initial states
initial_states = np.load(f"{baseline_root}/data/user_states_by_day.npy")[0]
```

```
# Send no coupon (0, 1.00) to all users
zero_actions = np.array([(0, 1.00) for _ in range(initial_states.shape[0])])
```

Then users' response action could be predicted by virtual environment in this way:

In [ ]:

```
import pickle as pk

with open(f"{baseline_root}/data/venv.pkl", "rb") as f:
    venv = pk.load(f, encoding="utf-8")

# Propogate from initial_states and zero_actions for one step, returning all nodes
node_values = venv.infer_one_step({ "state": initial_states, "action_1": zero_acti
user_action = node_values["action_2"]
print("Node values after propogated for one step:", node_values)
print("Predicted user actions:", user_action)
```

In above code, we could derive a set of user actions from a set of user states and a set of corresponding coupon actions, but this is not enough to train a policy using reinforcement learning methods. To train a policy here, we need to define our MDP with concepts like action space, observation space, and rewards.

We use `gym.Env` API here to formally setup our reinforcement-learning-ready environment.

## (a) Action space

Since a fair promotion policy requires sending a same action to all users, so the action space must be a 1-d array of `[coupon_num, coupon_discount]`.

Here, we restrict that less than 6 coupons are sent in a day. For coupon discount, since in the real world discounts are normally arranged in a fixed interval (e.g. 0.90, 0.85, 0.80...), so the possible values of coupon discount are also finite and discrete. In the competition and the baseline we restrict the coupon discount to only be one of [0.95, 0.90, 0.85, 0.80, 0.75, 0.70, 0.65, 0.60].

Therefore, the action space could be described as `gym.MultiDiscrete([6, 8])`, where

- `coupon_num = action[0]`,
- `coupon_discount = 0.95 - 0.05 * action[1]`.

For participants, feel free to change the action space used in your own policy, for example you can use any real number between [0.60, 1.00] to represent coupon discount, but note that the online evaluation environment only supports coupon discount in fixed possible values mentioned above, so remember to standardize the coupon discount to these values in your submitted policy validation program.

## (b) Observation space

For a fair promotion policy, it expect states of all users as input. However, `the states of all users` is a fairly high dimesion input, so instead of returing original user states, we perform a dimension reduction and return a low dimension observation to the policy.

As a baseline, we adopt a very straightforward method here: compute some basic statistics on all users' states, appended with a few additional information as the observation:

- `mean` : np.mean(states, axis=0)
- `std` : np.std(states, axis=0)
- `max` : np.max(states, axis=0)
- `min` : np.min(states, axis=0)
- `day_total_order_num` : Sum of orders of all users in last day
- `day_roi` : ROI of last day

Here, we compute an observation based on inital states, the day order num and ROI defaults to 0:

In [ ]:
```
from virtual_env import states_to_obs

obs = states_to_obs(initial_states, 0, 0.0)
print(obs.shape) # 14 = 3 + 3 + 3 + 3 + 1 + 1
print(obs)
```

## (c) Reward

Review the target of this competition:

**After several days of evaluation (14 in development phase, 30 in final), gain as much GMV as possible, on the premise that all days' ROI >= 6.5.**

Therefore, the reward should be designed so that:

- Return negative or reduced value if ROI does not meet the threshold.
- When ROI meets the requirement, return positive value that grows only with GMV (Since extra ROI does not count anymore).

As a baseline, we design a simple reward in this way:

- It is a delayed reward: returning non-zero only in the last day of evaluataion. Since the ROI is calculated based on all days.
- The ROI threshold is set a litter higher than the one in evaluation program: it is hard to train a policy that reliably pass the target threshold, so during training we set the threshold to be 8.0, with the expectation that it will generally produce ROI greater than 6.5.
- Positive reward is defined to be the ratio between actual GMV and the GMV when sending no coupon (set as `ZERO_GMV=81840` in baseline). Negative reward is defind to be the difference of actual ROI and threshold ROI.

# Training of the policy

After the definition of MDP, we can start to train the policy using any available reinforcement learning algorithm. In the baseline, we directly use the Proximal Policy Optimization (PPO) algorithm from `stablebaselines3` library for training.

Start tensorboard for policy training:

In [64]:
```
%load_ext tensorboard
```

```
!mkdir -p $baseline_root/data/logs

%tensorboard --bind_all --logdir $baseline_root/data/logs
```

Then start policy training, where the progress will be logged to tensorboard above:

In [ ]:
```
%pushd $baseline_root/data

from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import CheckpointCallback
from virtual_env import get_env_instance

env = get_env_instance('user_states_by_day.npy', 'venv.pkl')
model = PPO("MlpPolicy", env, n_steps=840, batch_size=420, verbose=1, tensorboard_
checkpoint_callback = CheckpointCallback(save_freq=8e4, save_path='model_checkpoir
model.learn(total_timesteps=int(8e6), callback=[checkpoint_callback])

%popd
```

Since the policy training is progressed on a fixed set of hyper parameters, normally the model trained with more steps gains better performance. We copy such a model to `baseline/data`:

In [ ]:
```
%pushd $baseline_root/data/model_checkpoints
!cp -f $(ls -Art . | tail -n 1) $BASELINE_ROOT/data/rl_model.zip
%popd
```

# Evaluation of the policy

The most convincing evaluation of our policy is to upload it to the competition website and fetch the oneline evaluation result on real environment. What we can do locally is to evaluate the policy on the virtual environment, in which the accuracy of policy is highly dependent on the accuracy of virtual environment. Therefore, it should be noted again that **designing an effective virtual environment is very important**.

Here we perform a rollout validation using the virtual environment and the policy:

In [69]:
```
%pushd $baseline_root/data
import importlib
from stable_baselines3 import PPO
import virtual_env
from virtual_env import get_env_instance, get_next_state_by_user_action

importlib.reload(virtual_env)

env = get_env_instance("user_states_by_day.npy", "venv.pkl")
policy = PPO.load("rl_model.zip")
validation_length = 14

total_gmv = 0.0
total_cost = 0.0
obs = env.reset()
for day_index in range(validation_length):
    coupon_action, _ = policy.predict(obs, deterministic=True) # Some randomness w
    obs, reward, done, info = env.step(coupon_action)
    if reward != 0:
```

```
        info["Reward"] = reward
    print(f"Day {day_index+1}: {info}")
%popd
```

```
/tmp/starting_kit/baseline/data
Day 1: {'CouponNum': 2, 'CouponDiscount': 0.8, 'UserAvgOrders': 0.39, 'UserAvgFe
e': 0.0}
Day 2: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.561, 'UserAvgFe
e': 42.67273490142822}
Day 3: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.826, 'UserAvgFe
e': 44.91680157089233}
Day 4: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.828, 'UserAvgFe
e': 46.454910022735596}
Day 5: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.832, 'UserAvgFe
e': 48.10989497375488}
Day 6: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.839, 'UserAvgFe
e': 49.678868606567384}
Day 7: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.848, 'UserAvgFe
e': 51.16703778839111}
Day 8: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.863, 'UserAvgFe
e': 52.57806259155274}
Day 9: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.891, 'UserAvgFe
e': 53.923981864929196}
Day 10: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.915, 'UserAvgF
ee': 55.16643067932129}
Day 11: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.94, 'UserAvgFe
e': 56.31369173812866}
Day 12: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 0.972, 'UserAvgF
ee': 57.37781147003174}
Day 13: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 1.028, 'UserAvgF
ee': 58.075493255615235}
Day 14: {'CouponNum': 5, 'CouponDiscount': 0.95, 'UserAvgOrders': 1.067, 'UserAvgF
ee': 58.658014682769775, 'TotalGMV': 569127.1408798217, 'TotalROI': 22.18447046842
11, 'Reward': 6.954137460758717}
/tmp/starting_kit/baseline/data
popd -> ~/Projects/polixir/codalab/codalab-polixir/competition-bundle/starting_ki
t/baseline/data
```

There are some heuristic hints to manually determine wheather the virtual environment and the policy works properly. Here are some examples:

**(a) Check output of each infer step.** Some output may indicate that virtual environment is not learned correctly. Take sending 5 coupons of discount 0.95 from initial state as example:

In [67]:

```
import pickle as pk

with open(f"{baseline_root}/data/venv.pkl", "rb") as f:
    venv = pk.load(f, encoding="utf-8")

initial_states = np.load(f"{baseline_root}/data/user_states_by_day.npy")[10]
coupon_actions = np.array([(5, 0.95) for _ in range(initial_states.shape[0])])

node_values = venv.infer_one_step({ "state": initial_states, "action_1": coupon_ac
user_actions = node_values['action_2']
day_order_num, day_avg_fee = user_actions[:, [0]], user_actions[:, [1]]
print(day_order_num.reshape((-1,))[:60])
print(day_avg_fee.reshape((-1,))[:60])
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 2. 0. 0. 0. 0. 0.]
[51.28172302 34.74122238 49.61096954 61.51179123 43.41001892 38.08740616
 42.86062622 33.88748169 39.5667572  61.51179123 34.12815857 33.59608078
 41.19500351 34.7923317  45.3996048  56.42188263 41.19500351 40.64524841
 35.17037582 43.31049347 50.07093048 33.3241539  47.29476166 32.66636276
 34.383461   32.0194397  58.741539   35.13378143 41.19500351 39.75680542
 48.76314926 52.18894196 33.7448616  34.7509079  41.19500351 41.0095253
 45.81291199 54.23230743 60.0966835  33.57566452 42.70819092 61.51179123
 41.19500351 34.92431641 52.76737213 33.76195908 59.16353226 43.58332062
 41.19500351 61.51179123 41.19500351 46.69792557 49.9764328  31.63819504
 61.51179123 46.29754257 43.67324829 41.19500351 41.19500351 45.82607269]
```

If you find that when user did not place any order (day_order_num == 0), there is still fees generated (day_avg_fee != 0), then your venv may not learn correctly.

**(b) Check reward curve.** The reward during rollout should be reasonable. A normal reward curve is usually in this pattern:

1. Start from negative reward, since at the beginning the ROI may not meet the threshold requirement. This is optional if your policy handles ROI well.

2. Reward is no greater than 10. The positive reward is defined as the ratio between actual GMV and zero action GMV (GMV when sending no coupon, set to be 81840). The GMV of taking some action will not beat zero action too far, and if you find your policy 10 times or even 50 times better than ZERO GMV (meaning reward to be > 10), then you must have selected a poor virtual environment (e.g. A venv that tells you your users will place 80 orders a day).

# Step 4: Generate a submission bundle

After obtaining a fair promotion policy, it's time to upload your model for online evaluation to get a final score. Only the policy needs to be included in your submission, since the competition platform will use the real environment to evaluate your policy.

## File structure

The uploaded file is a `.zip` bundle in such file structure (as shown in `sample_submission`):

- `data/` : data folder containing 1) the initial states defined in Step 1; 2) The policy model parameters learned in Step 3.
- `metadata` : yaml-format description file to specify the requirements of runtime environment (pytorch-1.8, pytorch-1.10, etc.)
- `policy_validation.py` : entrypoint file containing an interface class and a function to fetch participant's policy instance.
- `random_policy_validation.py` : an implementation of `PolicyValidation` that returns coupon actions randomly.
- `baseline_policy_validation.py` : an implementation of `PolicyValidation` that uses the baseline model.

# PolicyValidation file

The online evaluation program invokes participant's code through an interface named `PolicyValidation`. It is an abstract class defining required members and methods to be implemented by participants:

```
class PolicyValidation:
    """Abstract class defining the interfaces required by evaluation
program.
    """

    """initial_state is paticipant-defined first day's user state in
evaluation,
    derived from the offline data in May 18th.
    """
    initial_states: Any = None

    @abstractmethod
    def __init__(self, *args, **kwargs):
        """Initialize the members required for your model here.
        You may also provide some parameters for __init__ method,
        but you must fill the arguments yourself in the get_pv_instance
function.
        """

    @abstractmethod
    def get_next_states(self, cur_states: Any, coupon_action: np.ndarray,
user_actions: List[np.ndarray]) -> Any:
        """Generate next day's user state from current day's coupon
action and user's response action.
        """
        pass

    @abstractmethod
    def get_action_from_policy(self, user_states: Any) -> np.ndarray:
        """Generate current day's coupon action based on current day's
user states depicted by participants.
        """
        pass
```

(For detailed interface documentation, see `policy_validation.py`).

Along with `PolicyValidation` class, there is a function `get_pv_instance() -> PolicyValidation`, which will be invoked by evaluation program to fetch participant's implementation of `PolicyValidation`:

```
def get_pv_instance() -> PolicyValidation:
    from my_policy_validation import MyPolicyValidation
    return MyPolicyValidation(<your arguments...>)
```

Participants should inherit the `PolicyValidation` class, implement their own policy, and put initialization code in `get_pv_instance()`, so as to be successfully processed by online evaluation program.

# Metadata

Each participant's submission runs in an individual docker container. Since different teams may have their own requirements on the runtime environment (e.g. pytorch version), we provide a field `image` in `metadata` file, allowing for participants to choose what docker image to use for their evaluation process:

```
image: pytorch-1.8
```

We supports: `pytorch-1.8`, `pytorch-1.9`, `pytorch-1.10` for now. If your team requires other DL frameworks like tensorflow, keras, mxnet, or references some libraries that provided images does not contain, you can contact the team organizer to provide your list of dependencies. If appropriate, we will make the image satisfying the dependency for you.

# Create a submission

If you are using the basline model, you can use following command to update the parameters:

In [ ]:

```
%pushd $baseline_root/../sample_submission
!cp -f $BASELINE_ROOT/data/evaluation_start_states.npy ./data/evaluation_start_sta
!cp -f $BASELINE_ROOT/data/rl_model.zip ./data/rl_model.zip
%popd
```

When your submission is ready, create a zip file out of your submission folder:

In [ ]:

```
%pushd $baseline_root/../sample_submission
!zip -o -r --exclude='*.git*' --exclude='*__pycache__*' --exclude='*.DS_Store*' --
%popd
```

Congratulations! Now you can upload your submission to https://codalab.lisn.upsaclay.fr/competitions/823#participate-submit_results.