Отчёт-аннотация по выбранным моделям обучения агента LunarLander-v2

## Собственная модель обучения

Данная модель представляет собой модель обучения, основанная на 2-х способах: полном переборе таблицы состояний приземляемого модуля и на RDF, построенном на основе этой таблицы. Выбираются лучшие действия, в соответствие с наградой за траекторию. Для табличного способа выбор из строк таблицы на основании евклидового расстояния между состояниями, а RDF выбирает класс внутри себя, основываясь на минимизации скорости спускаемого модуля, его угла поворота и координаты X.

Для корректной работы модели необходимо подключить и/или докачать следующий набор библиотек:

* *import gym*
* *from EnvModel import EnvironmentModel*
* *from SolverClass import Solver*
* *from ControlClass import ControlModule*
* *import pylab*

В собственной модели RL содержатся следующие классы:

1. **class SimpleOwnModel**

В класс входят следующие методы:

* конструктор, сигнатура функции: *def \_\_init\_\_(self):*
* метод запуска обучения на *count\_of\_episodes* эпизодов, сигнатура функции: *def Run(self, count\_of\_episodes):*

1. **class EnvironmentModel**, который вынесен в отдельный файл EnvModel.py

В класс входят следующие методы:

* конструктор, сигнатура функции: *def \_\_init\_\_(self):*
* метод переобучения RDF, сигнатура: *def \_\_learn\_model(self):*
* метод обучения модели на обновлённом наборе траекторий, где *trajectories* - новые траектории, сигнатура функции: *def Retrain(self, trajectories):*
* метод расчёта Евклидова расстояния между двумя траекториями, где *state\_1* - 1-ое состояние, *state\_2* - 2-ое состояние, *dist* - расстояние между ними, сигнатура функции: *def \_\_euqlidDist(self, state\_1, state\_2):*
* метод предсказания наград на основе RDF по текущему состоянию и действию, где *current*\_*states* - состояние, возвращает *prediction* – предсказанные награды по состоянию и действию, сигнатура функции: *def predictByRDF(self, current\_state):*
* метод предсказания действия табличным способом по текущему состоянию, где *current*\_*states* - состояние, возвращает *prediction* - предсказанное действие по состоянию, сигнатура функции: *def predictByTable(self, current\_state):*

1. **class Solver**, который вынесен в отдельный файл SolverClass.py

В класс входят следующие методы:

* конструктор, сигнатура функции: *def \_\_init\_\_(self, envModel : EnvironmentModel):*
* метод подбора оптимального действия по текущему состоянию, сигнатура: *def Solve(self, state):*
* метод отправки траекторий *trajectories* на переобучение, сигнатура: *def UpdateEnv(self, trajectories):*

1. **class ControlModule**, который вынесен в отдельный файл ControlClass.py

В класс входят следующие методы:

* конструктор, сигнатура: *def \_\_init\_\_(self, solver : Solver):*
* метод получения управления по состоянию, сигнатура: *def GetControl(self, state):*
* метод добавления траектории со стоимостью *reward* для дальнейшего переобучения, сигнатура: *def AddTrajectory(self, reward):*

## Q-Learning модель обучения

Данная модель обучения представляет собой метод, применяемый в искусственном интеллекте при агентном подходе. Относится к экспериментам вида обучения с подкреплением. На основе получаемого от среды вознаграждения агент формирует функцию полезности Q, что впоследствии дает ему возможность уже не случайно выбирать стратегию поведения, а учитывать опыт предыдущего взаимодействия со средой.

Для корректной работы модели необходимо подключить и/или докачать следующий набор библиотек:

* *import pylab*
* *from collections import defaultdict*
* *import numpy as np*
* *import pandas as pd*
* *from sklearn.model\_selection import train\_test\_split*
* *from sklearn.ensemble import RandomForestClassifier*
* *import gym*

Q-learning модель RL включает в себя следующие классы:

1. **class QLearningAgent**

* конструктор, сигнатура: *def \_\_init\_\_(self, alpha, epsilon, discount, get\_legal\_actions):*
* метод обучения дерева, сигнатура: *def \_\_learn\_forest(self):*
* метод получения значения Q из таблицы по состоянию и действию, сигнатура: *def get\_qvalue(self, state, action):*
* метод установки значения Q в таблицу по состоянию и действию, сигнатура: *def set\_qvalue(self, state, action, value):*
* метод получения значения Q по состоянию, сигнатура: *def get\_value(self, state):*
* метод обновления значения Q, сигнатура: *def update(self, state, action, reward, next\_state):*
* метод получения лучшего действия из таблицы в зависимости от состояния, сигнатура: *def get\_best\_action(self, state):*
* метод получения действия по состоянию, сигнатура: *def get\_action(self, state):*

1. **class QLearningModel**

* конструктор, сигнатура: *def \_\_init\_\_(self):*
* метод запуска обучения на *count\_of\_episodes* эпизодов, сигнатура: *def Run(self, count\_of\_episodes):*

## Нейросетевая модель обучения

Данная модель обучения состоит из двух подсетей: «актера» и «критика». Модель Актер» выполняет задачу изучения того, какие действия следует предпринять в конкретном наблюдаемом состоянии окружающей среды. В случае LunarLander-v2 он принимает список из 8 значений игры в качестве входных данных, который представляет текущее состояние нашей ракеты и дает конкретное действие, какой двигатель запускать в качестве выходных данных. После завершения этих действий и получения очков награды, решения «Актера» оцениваются «Критиком». Таким образом, данные подсети дополняют друг друга, основываясь на вердиктах. Перерасчёт ошибок в этих подсетях осуществляется по методу Proximal Policy Optimization (PPO).

Для корректной работы модели необходимо подключить и/или докачать следующий набор библиотек:

* *import os*
* *os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3' –* для игнорирования предупреждений от tensorflow
* *import tensorflow as tf*
* *import numpy as np*
* *import copy*
* *import gym*
* *import pylab*
* *import tensorboardX*
* *import warnings*
* *warnings.filterwarnings('ignore') –* для игнорирования системных предупреждений
* *tf.compat.v1.disable\_eager\_execution() –* для ускорения рендера изображения

В нейросетевой модели RL содержатся следующие классы:

1. ***class Actor\_Model*** *–* класс «Актёра»

* конструктор, сигнатура: *def \_\_init\_\_(self, input\_shape, action\_space, lr, optimizer):*
* метод перерасчёта ошибки, сигнатура: *def ppo\_loss(self, y\_true, y\_pred):*
* метод выставления вероятностей действий по состоянию: *def predict(self, state):*

1. ***class Critic\_Model*** *–* класс «Критика»

* конструктор, сигнатура: *def \_\_init\_\_(self, input\_shape, action\_space, lr, optimizer):*
* метод перерасчёта ошибки, сигнатура: *def critic\_PPO2\_loss(self, values):*
* метод предсказания успешности действия по состоянию, сигнатура: *def predict(self, state):*

1. ***class NetModel*** *–* класс обучения модели

* выбор действия по предсказанным «Актёром» вероятностям, сигнатура: *def act(self, state):*
* метод перерасчёта ошибок, сигнатура: *def get\_gaes(self, rewards, dones, values, next\_values, gamma = 0.99, lamda = 0.9, normalize=True):*
* метод переобучения «Актёра» и «Критика», сигнатура: *def replay(self, states, actions, rewards, predictions, dones, next\_states):*
* загрузка весов нейросетей из файлов, сигнатура: *def load(self):*
* сохранение весов нейросетей в файлы, сигнатура: *def save(self):*
* построение графика, сигнатура: *def PlotModel(self, score, episode):*
* функция запуска обучения, сигнатура: *def Run(self, count\_of\_episodes, loading\_weights = False).* Параметр *loading\_weights* выставляется для возможной подгрузки уже имеющихся весов нейросети.