КУРСОВ ПРОЕКТ

Обучение с поощрения

в средата на „Акробот“

**Факултет по Математика и Информатика**  
**Студент:** Борислав Стоянов Марков  
**Факултетен номер:** **0MI3400048**  
**Учебен план:** **Изкуствен Интелект (редовно, магистър)**  
Курс: **Курс 1**; Група: **Група 1**  
**Активен период**: 2021/2022 летен, магистри  
**Дисциплина**: Обучение по метода „поощрение/наказание“

Дата: 16.06.2022г.

# 1. Съдържание

[1. Съдържание 2](#_Toc106298978)

[2. Увод 2](#_Toc106298979)

[3. Средата „Акробот“[3] в Gym 2](#_Toc106298980)

[4. Алгоритъм Актьор-критика 4](#_Toc106298981)

[5. Реализация на проекта 6](#_Toc106298982)

[5.1 Анализ на резултатите 7](#_Toc106298983)

[6. Идеи за бъдещо развитие и подобрения 9](#_Toc106298984)

[7. Източници и използвана литература 9](#_Toc106298985)

[Приложения 9](#_Toc106298986)

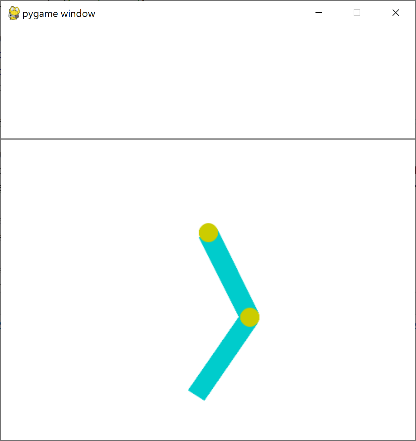
[1. Сорс код (Source code) 9](#_Toc106298987)

# 2. Увод

Идеята за проекта е да се обучи една от игрите предлагани от средата Джим (Gym) [3]. Играта се казва „Акробот“ (Acrobot). Имаме мотор и висяща част от две рамена. С импулси подавани на ротора трябва да се премине над определната черта от средата. Реализацията е на Питон(Python) с библиотека ПайТорч (Pytorch).

# 3. Средата „Акробот“[3] в Gym

Средата Акробот е част от средите с класически контрол за обучителни цели с Метод на поощрение и наказание (Reinforcement Learning). На следващата фигура е показана картинка от примерно обучение на тази среда.



Фигура 3.1

Основна информация за средата са дадени в следващата таблица.

|  |  |
| --- | --- |
| Пространство на действията | Discrete(3) |
| Вектор на наблюдението | (6) |
| Максимални стойности | [ 1. 1. 1. 1. 12.57 28.27] |
| Минимални стойности | [ -1. -1. -1. -1. -12.57 -28.27] |
| Импортиране в Питон | gym.make("Acrobot-v1") |

Таблица 3.1. Общи параметри за средата

Системата се състои от верига с две прави рамена свързани в краищата. Дейстивето е дискретно и се изразява в това как движим взаимно рамената в точката на свързване на рамената. Съответно действието е дискретно и може да бъде 0,1 или 2. Съответно 0 дава въртящ момент от -1 [Nm], 1 дава 0 [Nm] и 2 дава 1 [Nm] въртящ момент в точката на свързване на рамената. Поощрението е -1 на всяка стъпка и на последната стъпка е 0, като целта е за под 100 стъпки да бъде пресечена горната черна линия от края на второто рамо. Задачата се счита за решена ако успеем да приключим епизода с награда над -100.

В таблица 3.2 даваме значенията на вектора на състоянието за всеки един компонент.

|  |  |  |  |
| --- | --- | --- | --- |
| Номер | Вектор на състоянието | Минимимум | Максимум |
| 0 | cos(theta1) | -1 | 1 |
| 1 | sin(theta1) | -1 | 1 |
| 2 | cos(theta2) | -1 | 1 |
| 3 | sin(theta2) | -1 | 1 |
| 4 | Ъглова скорост theta1 | ~ -12.567 (-4 \* pi) | ~ 12.567 (4 \* pi) |
| 5 | Ъглова скорост theta2 | ~ -28.274 (-9 \* pi) | ~ 28.274 (9 \* pi) |

Таблица 3.2. Вектор на състоянието за средата

# 4. Алгоритъм Актьор-критика

За алгоритъм считам, че е подходящо да се използва Актьор-Критика, описан в [1], което е епизодичен алгоритъм без запаметяване на повече от две стъпки и попада в графата на TD(0) алгоритмите. Схематично можем да го опишем с псевдокод на следващата таблица.

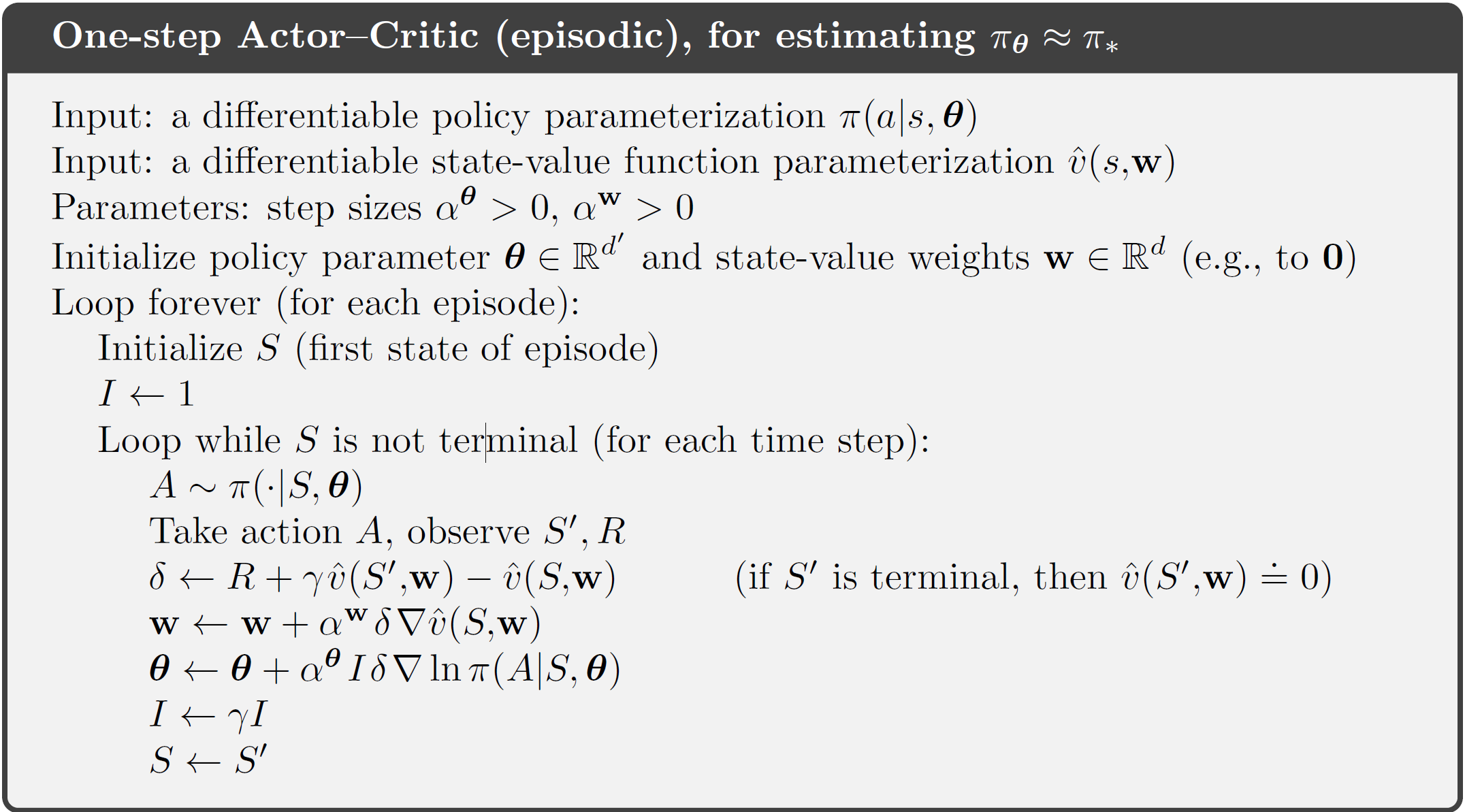


Таблица 4.1. Алгоритъм

Съответно тук следва да отбележим, че актьорът и критиката са реализирани като две отделни невронни мрежи и по този начин те са диференцируеми. В следващата таблица даваме кода за първата невронна мрежа.

|  |
| --- |
| class PolicyNetwork(nn.Module):   *# Takes in observations and outputs actions* def \_\_init\_\_(self, observation\_space, action\_space):  super(PolicyNetwork, self).\_\_init\_\_()  self.input\_layer = nn.Linear(observation\_space, 128)  self.output\_layer = nn.Linear(128, action\_space)   def forward(self, x):  x = self.input\_layer(x)  x = F.relu(x)  actions = self.output\_layer(x)  *# get softmax for a probability distribution* return F.softmax(actions, dim=1)   def save(self, path):  print(f"Saving Policy network in '{path}'")  torch.save(self.state\_dict(), path)   def select\_action(self, state):  *# make torch tensor of shape [BATCH x observation\_size]* state = torch.from\_numpy(state).view(1, -1).to(DEVICE)   *# use network to predict action probabilities* action\_probs = self(state)   *# sample an action using the probability distribution* m = Categorical(action\_probs)  *# action will be a single value tensor: [0] or [1] or [2]* action = m.sample()  *# return action as number and log probability* return action.item(), m.log\_prob(action) |

Таблица 4.2

Последният слой на мрежата за актьора е с активационна функция softmax(). Действието при обучение се взема на базата на вероятности, а не като argmax() от върнатите вероятности за класове. Вземането на действие на базата на вероятност се реализира с обекта **torch.distributions.Categorical** и неговия метод **sample()** който ни връща стойност според вероятността върната от softmax().

Невронната мрежа за критиката ще дадем в следващата таблица. Това е двуслойна мрежа със 128 неврона в скрития слой и един изходен, който ни е самата стойност на V(S). За изходящата стойност не се прилага активация, защото това би попречило да се апроксимира правилно функцията. Между първият и вторият слой се прилага RELU активация.

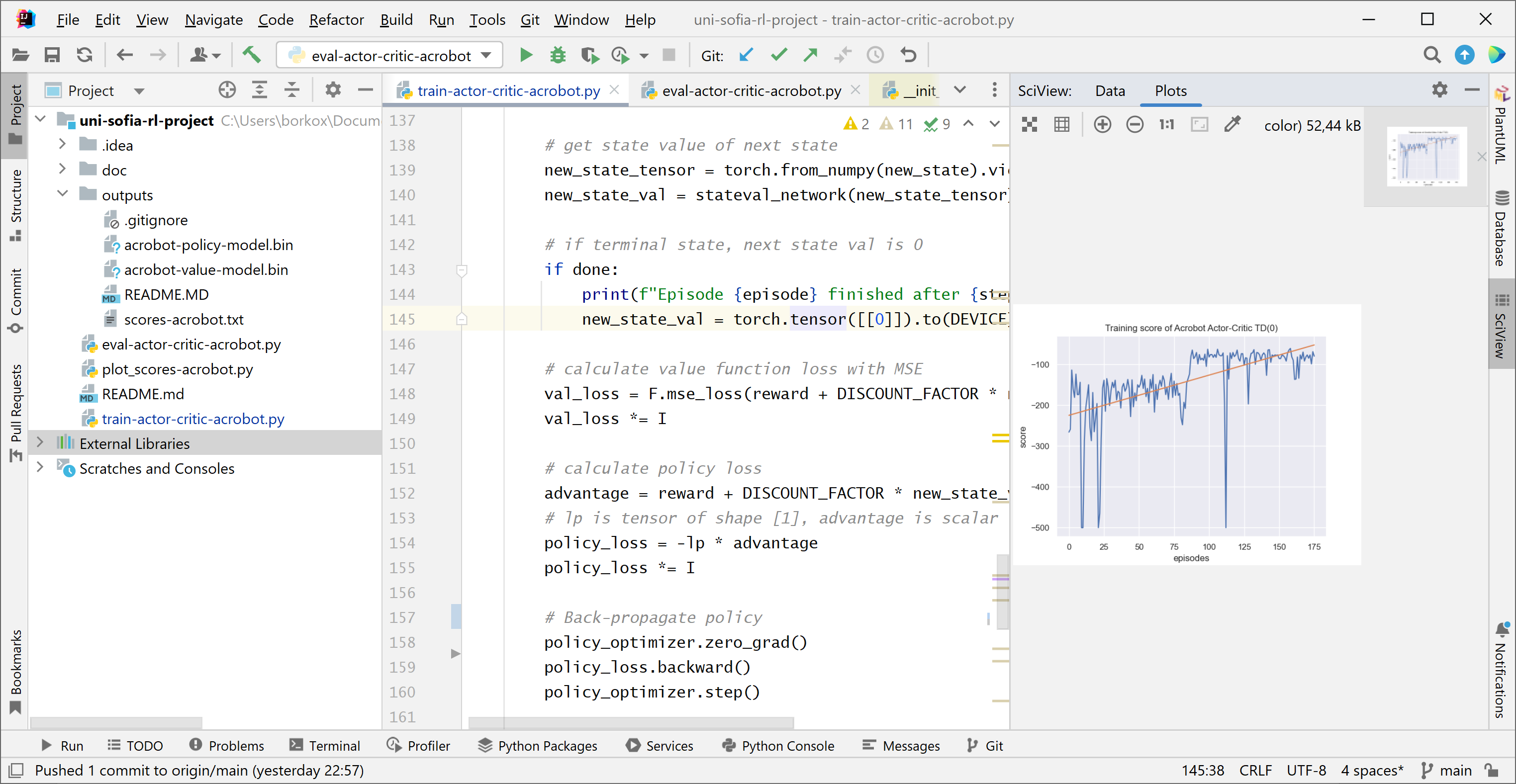
|  |
| --- |
| *# Using a neural network to learn state value* class StateValueNetwork(nn.Module):   *# Takes in state* def \_\_init\_\_(self, observation\_space):  super(StateValueNetwork, self).\_\_init\_\_()   self.input\_layer = nn.Linear(observation\_space, 128)  self.output\_layer = nn.Linear(128, 1)   def save(self, path):  print(f"Saving State-Value network in '{path}'")  torch.save(self.state\_dict(), path)   *# Expects X in shape [BATCH x observation\_space]* def forward(self, x):  x = self.input\_layer(x)  x = F.relu(x)  state\_value = self.output\_layer(x)  return state\_value |

Като оптимизатор се използва Adam с коефициент на обучение α=0.002. За коефициент на отстъпка(discount) използваме γ=0.95 . За обучение и смятане на градиента на функциите използваме инструментариума на PyTorch и неговите графове на обратно разпространение на грешката извиквани с метода backward(), даваме пример на седващата таблица.

|  |
| --- |
| *# calculate value function loss with MSE* val\_loss = F.mse\_loss(reward + DISCOUNT\_FACTOR \* new\_state\_val, state\_val) val\_loss \*= I  *# calculate policy loss* advantage = reward + DISCOUNT\_FACTOR \* new\_state\_val.item() - state\_val.item() *# lp is tensor of shape [1], advantage is scalar* policy\_loss = -lp \* advantage policy\_loss \*= I  *# Back-propagate policy* policy\_optimizer.zero\_grad() policy\_loss.**backward**() policy\_optimizer.step()  *# Back-propagate value* stateval\_optimizer.zero\_grad() val\_loss.**backward**() stateval\_optimizer.step() |

# 5. Реализация на проекта

Проектът е реализиран като github публичен проект и може да се види дори и през браузър(виж Приложения). За да се пусне локално се изисква инсталация на Python, конкретно 3.9.8 е използван тук. Линк към сорс кода е качен в гитхъб (Вж. Приложение 1) и е неразделна част от този документ. Структурата на приложението е дадена на фигура 5.1. Използваната среда за текстообработка и работа с git е IntelliJ .



Фигура 5.1. Обща структура на проекта

Подробни инструкции на са дадени в README.md файла.

В централната папка има скрипт “train-actor-critic-acrobot.py” на програмния език Python. С него се стартира процеса на обучение. По време на обучение резултатите от точките (поощрението) се записват във файл „outputs/scores-acrobot.txt“ за последваща визуализация. Скриптът „plot\_scores-acrobot.py“ ще ни визуализира картинка с резултатите след текущото обучение. След като сме обучили невронните мрежи можем да пуснем друг скрипт „eval-actor-critic-acrobot.py“, който ще ни визуализира готовото решение и ще проиграе няколко епизода за демонстрация. За край на обучение се приема момента, когато средно аритметичната награда от последните 100 епизода е над -100.

## 5.1 Анализ на резултатите

Като анализ можем да разгледаме графики при различни коефициенти на α(коефициент на обучение) и γ(коефициент на отстъпка). При този тип експерименти точната повторяемост на експериментите е невъзможна, поради случайния характер на актьора π (изборът на действие е с вероятности), случайната инициализация на теглата на невронните мрежи, както и поради случйната подредба на параметрите на средата в Gym при всяко пускане. Следва таблица с графики за различни стойности на параметрите.

|  |  |
| --- | --- |
| α=0.001, γ=0.99 |  |
| α=0.001, γ=0.95 |  |
| α=0.002, γ=0.95 |  |

Таблица 5.1.1

Оранжевата линия представлява линейна апроксимация на всички точки през епизодите. Тя ни показва тренда на обучение. Виждаме, че при α=0.002, γ=0.95 резултатите са най добри и обучението е само за 95 епизода и съответно няма катастрофални забравяния както в първия случай. Наистина този алгоритъм е доста надежден и дава решение в много от случаите. Тук дори не се интересуваме какви са връщаните стойности на състоянието, алгоритъмът би работил с произволни дължини на вектора. Единствено би се наложила промяна при преминаване в непрекъснато пространство на действията.

# 6. Идеи за бъдещо развитие и подобрения

Бихме могли да направим повече експерименти за настройване на хиперпараметрите и така да видим къде е оптимумът. Друга оптимизация е да пробваме този алгоритъм на подобни среди и да предвидим как ще се измени от дискретното пространство към непрекъснато пространство (за действията на актьора). Така ще става за задача с произволна сложност. В момента този алгоритъм в това състояние става само за дискретни състояния на актьора.

# 7. Източници и използвана литература

[1] Reinforcement Learning: An Introduction, 2018, Richard S. Sutton and Andrew G. Barto, [PDF] <http://www.incompleteideas.net/book/the-book-2nd.html>

[2] Actor-Critic: Implementing Actor-Critic Methods, Cheng Xi Tsou, 2021, <https://medium.com/geekculture/actor-critic-implementing-actor-critic-methods-82efb998c273>

[3] Acrobot, OpenGym, <https://www.gymlibrary.ml/environments/classic_control/acrobot/>

[4] Pytorch documentation, <https://pytorch.org/docs/stable/torch.html>

[5] Probability distributions - torch.distributions, <https://pytorch.org/docs/stable/distributions.html>

# Приложения

## 1. Сорс код (Source code)

<https://github.com/borkox/uni-sofia-rl-project>

В таблици са дадени скриптовете на Питон.

|  |
| --- |
| train-actor-critic-acrobot.py |
| import torch import torch.nn as nn import torch.nn.functional as F import torch.optim as optim from torch.distributions import Categorical  import gym import numpy as np from collections import deque  *# Description of the gym environment # https://www.gymlibrary.ml/environments/classic\_control/acrobot/  # discount factor for future utilities* DISCOUNT\_FACTOR = 0.95 *# number of episodes to run* NUM\_EPISODES = 1000 *# max steps per episode* MAX\_STEPS = 500 *# score agent needs for environment to be solved # For Acrobot-v1, this is -100* SOLVED\_SCORE = -100 *# device to run model on* DEVICE = "cuda" if torch.cuda.is\_available() else "cpu" LEARN\_RATE = 0.002 PATH\_POLICY\_MODEL = "outputs/acrobot-policy-model.bin" PATH\_VALUE\_MODEL = "outputs/acrobot-value-model.bin" PATH\_SCORES = "outputs/scores-acrobot.txt"   *# Using a neural network to learn our policy parameters* class PolicyNetwork(nn.Module):   *# Takes in observations and outputs actions* def \_\_init\_\_(self, observation\_space, action\_space):  super(PolicyNetwork, self).\_\_init\_\_()  self.input\_layer = nn.Linear(observation\_space, 128)  self.output\_layer = nn.Linear(128, action\_space)   def forward(self, x):  x = self.input\_layer(x)  x = F.relu(x)  actions = self.output\_layer(x)  *# get softmax for a probability distribution* return F.softmax(actions, dim=1)   def save(self, path):  print(f"Saving Policy network in '{path}'")  torch.save(self.state\_dict(), path)   def select\_action(self, state):  *# make torch tensor of shape [BATCH x observation\_size]* state = torch.from\_numpy(state).view(1, -1).to(DEVICE)   *# use network to predict action probabilities* action\_probs = self(state)   *# sample an action using the probability distribution* m = Categorical(action\_probs)  *# action will be a single value tensor: [0] or [1] or [2]* action = m.sample()  *# return action as number and log probability* return action.item(), m.log\_prob(action)   *# Using a neural network to learn state value* class StateValueNetwork(nn.Module):   *# Takes in state* def \_\_init\_\_(self, observation\_space):  super(StateValueNetwork, self).\_\_init\_\_()   self.input\_layer = nn.Linear(observation\_space, 128)  self.output\_layer = nn.Linear(128, 1)   def save(self, path):  print(f"Saving State-Value network in '{path}'")  torch.save(self.state\_dict(), path)   *# Expects X in shape [BATCH x observation\_space]* def forward(self, x):  x = self.input\_layer(x)  x = F.relu(x)  state\_value = self.output\_layer(x)  return state\_value   def save\_scores(scores\_list):  print(f"Saving scores in '{PATH\_SCORES}'.")  np.savetxt(PATH\_SCORES, scores\_list, delimiter=',')   *# Make environment* env = gym.make('Acrobot-v1')  *# Init network* print(f"Observation space: {env.observation\_space.shape[0]}") print(f"Action space: {env.action\_space.n}") policy\_network = PolicyNetwork(env.observation\_space.shape[0],  env.action\_space.n).to(DEVICE) stateval\_network = StateValueNetwork(env.observation\_space.shape[0]).to(DEVICE)  *# Init optimizer* policy\_optimizer = optim.Adam(policy\_network.parameters(), lr=LEARN\_RATE) stateval\_optimizer = optim.Adam(stateval\_network.parameters(), lr=LEARN\_RATE)  *# track scores* scores = []  *# track recent scores* recent\_scores = deque(maxlen=100)  *# run episodes* for episode in range(NUM\_EPISODES):   *# init variables* state = env.reset()  done = False  score = 0  I = 1   *# run episode, update online* for step in range(MAX\_STEPS):  env.render()  *# get action and log probability* action, lp = policy\_network.select\_action(state)   *# step with action* new\_state, reward, done, \_ = env.step(action)   *# update episode score* score += reward   *# convert to torch tensor [Batch x observationsize]* state\_tensor = torch.from\_numpy(state).reshape(1, -1).to(DEVICE)  state\_val = stateval\_network(state\_tensor)   *# get state value of next state* new\_state\_tensor = torch.from\_numpy(new\_state).view(1, -1).to(DEVICE)  new\_state\_val = stateval\_network(new\_state\_tensor)   *# if terminal state, next state val is 0* if done:  print(f"Episode {episode} finished after {step} timesteps, score={score}")  new\_state\_val = torch.tensor([[0]]).to(DEVICE)   *# calculate value function loss with MSE* val\_loss = F.mse\_loss(reward + DISCOUNT\_FACTOR \* new\_state\_val, state\_val)  val\_loss \*= I   *# calculate policy loss* advantage = reward + DISCOUNT\_FACTOR \* new\_state\_val.item() - state\_val.item()  *# lp is tensor of shape [1], advantage is scalar* policy\_loss = -lp \* advantage  policy\_loss \*= I   *# Back-propagate policy* policy\_optimizer.zero\_grad()  policy\_loss.backward()  policy\_optimizer.step()   *# Back-propagate value* stateval\_optimizer.zero\_grad()  val\_loss.backward()  stateval\_optimizer.step()   if done:  break   *# move into new state, discount I* state = new\_state  I \*= DISCOUNT\_FACTOR   *# append episode score* scores.append(score)  recent\_scores.append(score)   *# early stopping if we meet solved score goal* if np.array(recent\_scores).mean() >= SOLVED\_SCORE:  print(f"Learning is complete successfully.")  policy\_network.save(PATH\_POLICY\_MODEL)  stateval\_network.save(PATH\_VALUE\_MODEL)  break  if episode % 10 == 0:  save\_scores(scores)  save\_scores(scores) |

|  |
| --- |
| eval-actor-critic-acrobot.py |
| import torch import torch.nn as nn import torch.nn.functional as F import gym  PATH\_POLICY\_MODEL = "outputs/acrobot-policy-model.bin" PATH\_VALUE\_MODEL = "outputs/acrobot-value-model.bin"   *# Using a neural network to learn our policy parameters* class PolicyNetwork(nn.Module):   *# Takes in observations and outputs actions* def \_\_init\_\_(self, observation\_space, action\_space):  super(PolicyNetwork, self).\_\_init\_\_()  self.input\_layer = nn.Linear(observation\_space, 128)  self.output\_layer = nn.Linear(128, action\_space)   def forward(self, x):  x = self.input\_layer(x)  x = F.relu(x)  actions = self.output\_layer(x)  *# get softmax for a probability distribution* return F.softmax(actions, dim=1)   def save(self, path):  print(f"Saving Policy network in '{path}'")  torch.save(self.state\_dict(), path)   def select\_action(self, state):  *# make torch tensor of shape [BATCH x observation\_size]* state = torch.from\_numpy(state).view(1, -1)   *# use network to predict action probabilities* action\_probs = self(state)  *# This part is different from learning policy  # There is no exploration part anymore* return torch.argmax(action\_probs)  *# Make environment* env = gym.make('Acrobot-v1')  *# Init network* print(f"Observation space: {env.observation\_space.shape[0]}") print(f"Action space: {env.action\_space.n}") policy\_network = PolicyNetwork(env.observation\_space.shape[0],  env.action\_space.n) policy\_network.load\_state\_dict(torch.load(PATH\_POLICY\_MODEL)) policy\_network.eval()  scores = [] *# run episodes* for episode in range(5):   *# init variables* state = env.reset()  done = False  score = 0   *# run episode, update online* for step in range(500):  env.render()  *# get action and log probability* action = policy\_network.select\_action(state)   *# step with action* state, reward, done, \_ = env.step(action)   *# update episode score* score += reward   *# if terminal state, next state val is 0* if done:  print(f"Episode {episode} finished after {step} timesteps, score={score}")  break  *# append episode score* scores.append(score) |

|  |
| --- |
| plot\_scores-acrobot.py |
| import matplotlib.pyplot as plt from sklearn.linear\_model import LinearRegression import seaborn as sns import numpy as np  sns.set()  scores = np.loadtxt('outputs/scores-acrobot.txt')  plt.plot(scores) plt.ylabel('score') plt.xlabel('episodes') plt.title('Training score of Acrobot Actor-Critic TD(0)')  reg = LinearRegression() reg.fit(np.arange(len(scores)).reshape(-1, 1), np.array(scores).reshape(-1, 1)) y\_pred = reg.predict(np.arange(len(scores)).reshape(-1, 1)) plt.plot(y\_pred) plt.show() |