|  |  |  |  |
| --- | --- | --- | --- |
|  | Politechnika Bydgoska im. J. J. Śniadeckich  Wydział Telekomunikacji,  Informatyki i Elektrotechniki | |  |
| **Przedmiot** | Analiza Regresji i Szeregów Czasowych | | |
| **Prowadzący** | dr inż. Damian Ledziński | | |
| **Temat** | RL Stable Baselines i Własny model | | |
| **Student** | Cezary Tytko | | |
| **Nr lab.** | 5 i 6 | **Data wykonania** | 24.06.2024 |
| **Ocena** |  | **Data oddania spr.** |  |

Zadaniem jest rozwiązanie problemu RL dla gry z Atari z wykorzystaniem bibliotek Gymnasium i Stable Baselines.

**Lab 5. Model Predefiniowany**

Jako grę w którą model będzie uczył się grać wybrałem FrozenLake w wersji planszy 8x8: „FrozenLake8x8-v1”.

Najpierw wybrałem Algorytm PPO i model predefiniowany MlpPolicy i trenowałem go w specjalnie przygotowanej pętli, która co iterację treningu, generowała kilka testowych klatek działania algorytmu, tym samym obrazując zmianę w podejściu modelu do problemu w czasie

1. # Wyświetl wszystkie dostępne środowiska

2. print(gym.envs.registry.keys())

3.

4. # Utwórz środowisko

5. env = gym.make('FrozenLake8x8-v1', render\_mode='rgb\_array')

6. env = DummyVecEnv([lambda: env])

7.

8. # Utwórz model

9. model = PPO('MlpPolicy', env, verbose=1)

10.

11. timesteps\_per\_iteration = 8000

12. iterations = 100

13. # Trenuj model

14. frames = []

15. for \_ in range(iterations):

16. obs = env.reset()

17. model.learn(total\_timesteps=timesteps\_per\_iteration, reset\_num\_timesteps=False)

18. obs = env.reset()

19. for \_ in range(10):

20. action, \_states = model.predict(obs)

21. obs, rewards, dones, info = env.step(action)

22. frame = env.render("rgb\_array")

23. frames.append(frame)

24.

25. # Zapisz model

26. model.save("ppo\_FrozenLake")

27.

28. output\_file\_avi = 'testowanie\_modelu.avi' # Przykładowe rozszerzenie AVI

29. fps = 10 # Szybkość klatek na sekundę

30.

31. # Zapisz animację do pliku AVI

32. imageio.mimsave(output\_file\_avi, frames, fps=fps)

33.

 

Wytrenowałem również inny algorytm uczenia RL: A2C dla tego samego problemu i z tym samym domyślnym modelem MlpPolicy:

1. print(gym.envs.registry.keys())

2.

4. env = gym.make('FrozenLake8x8-v1', render\_mode='rgb\_array')

5. env = DummyVecEnv([lambda: env])

6.

8. model = A2C('MlpPolicy', env, verbose=1)

9.

10. timesteps\_per\_iteration = 8000

11. iterations = 100

13. frames = []

14. for \_ in range(iterations):

15. obs = env.reset()

16. model.learn(total\_timesteps=timesteps\_per\_iteration, reset\_num\_timesteps=False)

17. obs = env.reset()

18. for \_ in range(10):

19. action, \_states = model.predict(obs)

20. obs, rewards, dones, info = env.step(action)

21. frame = env.render("rgb\_array")

22. frames.append(frame)

23.

25. model.save("A2C\_FrozenLake")

26.

27. output\_file\_avi = 'testowanie\_modelu\_A2C.avi'

28. fps = 10

29.

31. imageio.mimsave(output\_file\_avi, frames, fps=fps)

32.

 

**Lab 6. Własny Model do RL**

Własne rozwiązania w przypadku RL można implementować na różnych poziomach: Analizy wejścia (np. obrazu) i ekstrakcji nowych cech które służą do uczenia Modelu odpowiadającego za podejmowanie decyzji, którego wyjściem są akcje jakie możemy podjąć, oraz model do oceny wartości stanu gry (jak dobry jest stan na planszy).

W ramach ćwiczenia przygotowałem przykład jak własnoręcznie utworzyć model do podejmowania akcji i analizy wartości:

1. class CustomNetwork(nn.Module):

10.

11. def \_\_init\_\_(

12. self,

13. feature\_dim: int,

14. last\_layer\_dim\_pi: int = 64,

15. last\_layer\_dim\_vf: int = 64,

16. ):

17. super().\_\_init\_\_()

18.

19. # IMPORTANT:

20. # Save output dimensions, used to create the distributions

21. self.latent\_dim\_pi = last\_layer\_dim\_pi

22. self.latent\_dim\_vf = last\_layer\_dim\_vf

23.

24. # Policy network

25. self.policy\_net = nn.Sequential(

26. nn.Linear(feature\_dim, last\_layer\_dim\_pi), nn.ReLU()

27. )

28. # Value network

29. self.value\_net = nn.Sequential(

30. nn.Linear(feature\_dim, last\_layer\_dim\_vf), nn.ReLU()

31. )

32.

33. def forward(self, features: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:

38. return self.forward\_actor(features), self.forward\_critic(features)

39.

40. def forward\_actor(self, features: torch.Tensor) -> torch.Tensor:

41. return self.policy\_net(features)

42.

43. def forward\_critic(self, features: torch.Tensor) -> torch.Tensor:

44. return self.value\_net(features)

45.

46.

47. class CustomActorCriticPolicy(ActorCriticPolicy):

48. def \_\_init\_\_(

49. self,

50. observation\_space: spaces.Space,

51. action\_space: spaces.Space,

52. lr\_schedule: Callable[[float], float],

53. \*args,

54. \*\*kwargs,

55. ):

56. # Disable orthogonal initialization

57. kwargs["ortho\_init"] = False

58. super().\_\_init\_\_(

59. observation\_space,

60. action\_space,

61. lr\_schedule,

62. # Pass remaining arguments to base class

63. \*args,

64. \*\*kwargs,

65. )

66.

67.

68. def \_build\_mlp\_extractor(self) -> None:

69. self.mlp\_extractor = CustomNetwork(self.features\_dim)

70.

71. # Tworzenie środowiska

72. env = gym.make('FrozenLake-v1', render\_mode='rgb\_array')

73. env = DummyVecEnv([lambda: env])

74.

75. model = PPO(CustomActorCriticPolicy, env, verbose=1)

76.

77. frames = []

78.

79. timesteps\_per\_iteration = 8000

80. iterations = 100

81.

82. # Trenuj model

83. frames = []

84. for \_ in range(iterations):

85. obs = env.reset()

86. model.learn(total\_timesteps=timesteps\_per\_iteration, reset\_num\_timesteps=False)

87. obs = env.reset()

88. for \_ in range(30):

89. action, \_states = model.predict(obs)

90. obs, rewards, dones, info = env.step(action)

91. frame = env.render("rgb\_array")

92. frames.append(frame)

93.

94. # Zapisz model

95. model.save("ppo\_FrozenLake\_custom")

96.

97. # Zapisz animację do pliku AVI

98. output\_file\_avi = 'PPO\_Custom\_testowanie\_modelu.avi'

99. fps = 10 # Szybkość klatek na sekundę

100. # Zapisz animację do pliku AVI

101. imageio.mimsave(output\_file\_avi, frames, fps=fps)

102.

103. print(f"Zapisano animację do pliku {output\_file\_avi}")

104.



Klasa „CustomNetwork” stanowi nasz model, natomiast „CustomActorCriticPolicy” opakowuje go i pozwala przekazać jako własną politykę do algorytmu uczenia RL PPO.

Wnioski:

Uczenie ze wzmocnieniem jest bardzo trudnym i skomplikowanym zagadnieniem, nie tylko pod kątem samego modelu gdzie mamy elementy takie jak aktor, środowisko i dostępne akcje, ale implementacja w środowisku gymnasium i stable baselines również nie należy do najłatwiejszych, głównie przez mnogość funkcji, dlatego zaprezentowane rozwiązania działają, ale modele słabo się uczą i nie prezentują sobą idealnego rozwiązania, jednak na podstawie przedstawionych przykładów, można starać się je rozbudować.