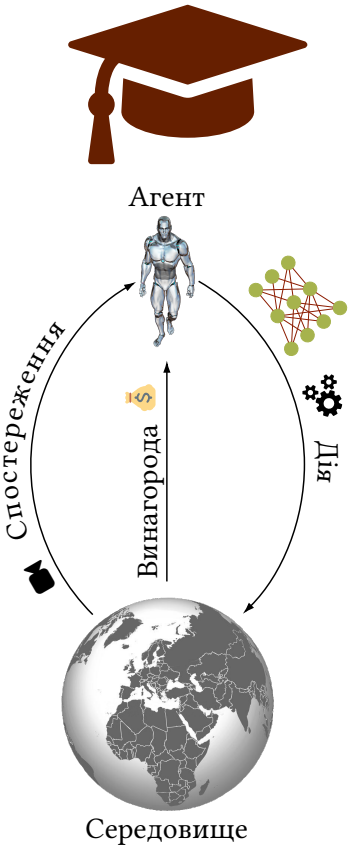
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Навчання з підкріпленням

Методичні вказівки для виконання практичних робіт | Осінній семестр



Практична 1: Перевернутий маятник (Cartpole), понг (Pong) та MountainCar

"The difference between stupidity and genius is that genius has its limits."
"Різниця між геніальністю і дурістю в тому, що у першої є свої межі."

– Альберт Ейнштейн

Вступ

Мета цієї роботи – познайомитися на практиці як можна створити середовище та навчити агента дотримуватись в ньому певної стратегії. Вам буде надано файл-розв'язок завдання з `MIT 6.S191 Introduction to Deep Learning` 2021 р. У цьому файлі-розв'язку подано приклади моделювання двох середовищ різної складності з використанням бібліотеки Gym.

- Файл-розв'язок: відкрити в COLAB

Приклад: перевернутий маятник

Завдання агента – оптимізувати (максимізувати) загальну винагороду, отриману ним при взаємодії з навколишнім середовищем. На кожному кроці в момент часу t агент:

- Отримує спостереження O_t та винагороду R_t
- Виконує дію A_t

Середовище:

- Отримує дію A_t
- Продукує спостереження O_{t+1} та винагороду R_{t+1}

Винагорода – це скалярний сигнал, який отримує агент у якості зворотного зв'язку від середовища. Вона показує, наскільки добре працює агент у момент часу t відповідно до поставленої мети. Формально агент має максимізувати кумулятивну винагороду:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots \tag{1}$$

- G_t називається загальною винагородою (return) – сума всіх винагород, які агент розраховує отримати при дотриманні стратегії від певного стану до кінця епізоду[1].

Навчання з підкріпленням базується на гіпотезі винагороди:

> "Будь-яка мета може бути формалізована як результат максимізації сукупної винагороди."

[1] Епізод – кожна спроба агента вивчити середовище.

Функції цінності:

- Цінність станів

  Очікувана сукупна винагорода, яку агент розраховує отримати від стану s називається **цінністю (value)**:

$$v(s) = \mathbb{E}\left[G_t \mid S_t = s\right] = \\ = \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots \mid S_t = s\right] \tag{2}$$

- Цінність дій – Q-функція[2]

  Q-функція дозволяє оцінити цінність (якість) дій:

$$q(s,a) = \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right] = \\ = \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots \mid S_t = s, A_t = a\right] \tag{3}$$

Навчання з підкріпленням фундаментально відрізняється від контрольованого (з учителем) та неконтрольованого (без учителя) / напів-контрольованого навчання, оскільки ми тренуємо модель управляти діями нашого агента з метою знаходження ним оптимальної послідовності дій, які приведуть його до вирішення поставленого завдання з максимальним кінцевим значенням загальної винагороди. Іншими словами метою навчання агента є визначення найкращого наступного стану у який має перейти агент для отримання найбільшої кінцевої винагороди.

У цій практичній роботі ми сфокусуємось на створенні алгоритму навчання з підкріпленням для вивчення трьох різних середовищ з різним рівнем складності:

1. Перевернутий маятник (Cartpole): балансування стовпа, що стоїть на візку у вертикальному положенні, пересуваючи візок вліво та вправо.
2. Понг (Pong): виграйте ваших опонентів (інший штучний інтелект або людей) у грі Pong. Великопросторове середовище спостереження – навчання напряму з піксельних даних.
3. MountainCar: агент (автомобіль) знаходиться між двома пагорбами, його завдання заїхати на пагорб праворуч до прапорця, проте двигун агента недостатньо потужний, щоб піднятися на гору просто рухаючись вперед. Тому єдиний спосіб досягти успіху – це рухатися вперед-назад, щоб набрати потрібних обертів.

Для того, щоб змоделювати вище згадані середовища, ми будемо використовувати API Gym, розроблений компанією OpenAI. Gym пропонує кілька попередньо визначених середовищ для тренування та тестування агентів навчання з підкріпленням, включаючи класичні завдання для фізичного контролю, відеоігри Atari, а також симуляції роботів. Для інсталяції базової бібліотеки Gym використовуйте:

```
pip install gym
```

Додаткові деталі щодо інсталяції можна знайти тут: https://github.com/openai/gyminstallation. Google Colab має попередньо інстальовану Gym бібліотеку, версію якої можна перевірити наступним чином:

```
import gym
gym.__version__
```

У перевернутого маятника стовп прикріплений за допомогою шарніру до візка, який може рухатись вліво-вправо без тертя. На початку епізоду стовп знаходиться у вертикальному положенні і метою є запобігання

[2] Q – перша літеру англійського слова Quality, означає якість.

його падіння. Система контролюється шляхом прикладання сили зі значенням +1 чи -1 до візка. Винагорода +1 надається за кожен момент часу, коли стовп залишається вертикальним. Епізод закінчується, коли стовп відхиляється більше ніж на 15 градусів від вертикалі, або ж якщо візок від'їзжає більше ніж на 2.4 одиниці від центра. Візуальна репрезентація візка зображена на рисунку 1.

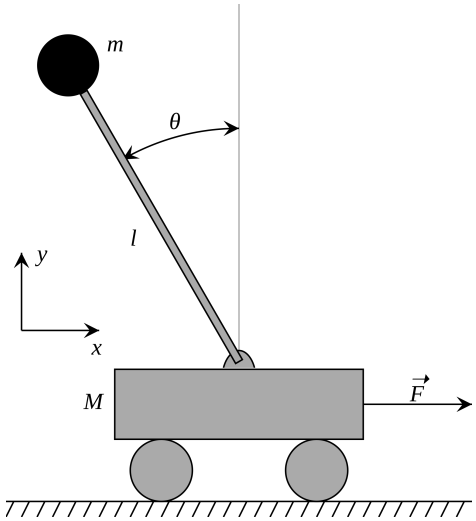


Рис. 1: Перевернутий маятник

Для ініціалізації середовища потрібно викликати функцію `make` з бібліотеки `gym` та передати в якості параметра назву потрібного середовища. Однією з проблем з якою ми можемо зіштовхнутися створюючи алгоритми навчання з підкріпленням, це те, що багато аспектів навчання за своєю сутністю є випадковими: ініціалізація стану гри, зміни в середовищі, дії агента. Тому гарною ідеєю є встановлення початкового `seed` для середовища, аби забезпечити деякий рівень контролю. Так само як можна використовувати `numpy.random.seed`, ми викликаємо таку саму функцію у `gym` з нашим середовищем, для того аби наше середовище мало однакові випадкові величини при різних запусках:

```
### Instantiate the Cartpole environment ###

env = gym.make("CartPole-v1")
env.seed(1)
```

Визначивши середовище та цілі гри, ми можемо подумати про те, які :

1. спостереження можуть допомогти визначити стан середовища
2. дії агент може виконувати у цьому середовищі

Спочатку, подумаємо про спостереження. У цьому середовищі нашими спостереженнями будуть:

1. позиція візка
2. швидкість візка
3. кут стовпа
4. швидкість повороту стовпа

Для того, щоб перевірити розмір простору, виведемо його на екран:

```
n_observations = env.observation_space
print("Environment has observation space =", n_observations)
```

```
Environment has observation space = Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
```

Далі, ми розглядаємо простір дій. На кожному кроці, агент може пересуватися або вліво або вправо. Ми знову ж таки можемо перевірити розмір простору, вивівши значення на екран:

```
n_actions = env.action_space.n
print("Number of possible actions that the agent can choose from =", n_actions)
```

```
Number of possible actions that the agent can choose from = 2
```

Тепер, коли ми задали середовище та зрозуміли розмірність просторів спостереження та дій агент, можемо визначати нашого агента. У глибинному навчанні з підкріпленням, агент задається глибинною нейронною мережею. Ця мережа буде брати на вхід спостереження середовища та видавати ймовірності виконання кожної з можливих дій. Оскільки Cartpole є простором з малим простором спостережень, для агента підійде проста повнозв'язна (feed-forward) нейронна мережа. Ми задамо її за допомогою Sequential API:

```
### Define the Cartpole agent ###

# Defines a feed-forward neural network
def create_cartpole_model():
    model = tf.keras.models.Sequential([
        # First Dense layer
        tf.keras.layers.Dense(units=32, activation='relu'),

        # Think about the space the agent needs to act in!
        tf.keras.layers.Dense(units=n_actions, activation=None)

    ])
    return model

cartpole_model = create_cartpole_model()
```

Тепер, коли ми задали основну архітектуру мережі, ми визначимо функцію дій, що виконуватиме прямий прохід по мережі на основі множини спостережень, а також збирає вихід з мережі. На основі цього буде прийматися рішення про наступний крок агента.

```
### Define the agent's action function ###

# Function that takes observations as input, executes a forward pass through model,
#   and outputs a sampled action.
# Arguments:
#   model: the network that defines our agent
#   observation: observation(s) which is/are fed as input to the model
#   single: flag as to whether we are handling a single observation or batch of
#     observations, provided as an np.array
# Returns:
#   action: choice of agent action
def choose_action(model, observation, single=True):
    # add batch dimension to the observation if only a single example was provided
    observation = np.expand_dims(observation, axis=0) if single else observation

    ''' Feed the observations through the model to predict the log probabilities of each possible action.'
    logits = model.predict(observation)

    '''Choose an action from the categorical distribution defined by the log
       probabilities of each possible action.'''
    action = tf.random.categorical(logits, num_samples=1)
```

```
    action = action.numpy().flatten()

    return action[0] if single else action
```

Тепер, коли ми визначили середовище та архітектуру мережі агента, а також функцію дій, ми готові перейти до наступного кроку:

1. **Ініціалізація середовища та агента:** тут ми опишемо різні спостереження та дії, що їх може виконати агент у середовищі.
2. **Визначення пам'яті агента**: це дасть змогу агенту запам'ятовувати свої попередні дії та винагороди.
3. **Задання алгоритму навчання**: за допомогою цього буде виконано підкріплення хороших дій та покарання поганих дій агента.

У навчанні з підкріпленням, тренування виконується завдяки взаємодії агента з середовищем та виконання дій агентом; епізодом називається послідовність дій, що закінчується у якомусь кінцевому стані, як-от падіння стовпа чи виїзд візка за межі. Агенту потрібно буде пам'ятати усі свої дії та спостереження, так що коли закінчується епізод, він зможе "підкріпити" хороші дії та "покарати" погані. Першим кроком є задання простого буферу пам'яті, що зберігає спостереження агента, його дії, а також отримані винагороди за конкретні епізоди.

```
### Agent Memory ###

class Memory:
    def __init__(self):
        self.clear()

  # Resets/restarts the memory buffer
    def clear(self):
        self.observations = []
        self.actions = []
        self.rewards = []

  # Add observations, actions, rewards to memory
    def add_to_memory(self, new_observation, new_action, new_reward):
        self.observations.append(new_observation)
        '''Update the list of actions with new action'''
        self.actions.append(new_action)
        # ['''TODO''']
        '''Update the list of rewards with new reward'''
        self.rewards.append(new_reward)


    def __len__(self):
        return len(self.actions)

# Instantiate a single Memory buffer
memory = Memory()
```

Ми майже готові почати навчати нашого агента! Наступним кроком є підрахунок винагород, які агент отримає за свої дії у середовищі. Оскільки ми (та агент) не знає напевно, коли закінчиться гра чи завдання (коли звалиться стовп), має сенс отримувати миттєву винагороду, а не замислюватись над тим, яка винагорода може бути отримана пізніше. Ця ідея схожа з отриманням відсотків від депозиту.

Для підрахунку очікуваної сумарної винагороди отриманої агентом з моменту часу t, ми сумуємо винагороду з фактором знецінювання у межах епізоду навчання з проектуванням у майбутнє:

$$G\_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

- Коефіцієнт знецінювання $\gamma \in [0, 1]$ показує на цінність майбутніх винагород
- Чим менший коефіцієнт знецінювання, тим менше агент замислюється над вигодою від майбутніх своїх дій.

Зверніть увагу на форму цієї суми – нам потрібно буде подумати про те, як її реалізувати. Конкретніше, нам потрібно буде ініціалізувати масив нулями, з довжиною рівною кількості кроків, та заповнювати його реальними значеннями винагороди зі знижкою по ходу того, як ми ітеруємо по винагородам епізоду, що зберігаються у пам'яті агента. Остаточно нам важливо, які дії кращі у порівнянні з іншими діями у цьому епізоді – тому ми нормалізуємо підраховані винагороди за допомогою середнього та стандартного відхилення винагород по всім епізодам.

```
### Reward function ###

# Helper function that normalizes an np.array x
def normalize(x):
    x -= np.mean(x)
    x /= np.std(x)
    return x.astype(np.float32)

# Compute normalized, discounted, cumulative rewards (i.e., return)
# Arguments:
#   rewards: reward at timesteps in episode
#   gamma: discounting factor
# Returns:
#   normalized discounted reward
def discount_rewards(rewards, gamma=0.95):
    discounted_rewards = np.zeros_like(rewards)
    R = 0
    for t in reversed(range(0, len(rewards))):
        # update the total discounted reward
        R = R * gamma + rewards[t]
        discounted_rewards[t] = R

    return normalize(discounted_rewards)
```

Тепер ми можемо почати визначати алгоритм навчання, що буде використовуватися для підкріплення хороших дій агента та покарання за погані дії. У цій роботі ми сфокусуємося на методах стратегій градієнту, метою яких є максимізація вірогідності дій, що призводять до великої винагороди. Аналогічно, це означає що потрібно мінімізувати негативну вірогідність цих же дій. Ми досягаємо цього множенням вірогідностей на їх відповідну винагороду – таким чином підсилюючи силу дій з великою винагородою.

Оскільки функція логарифму монотонно зростає, це означає що мінімізація негативної вірогідності еквівалентна мінімізації негативного логарифму вірогідності. Згадаймо, що ми можемо легко підрахувати негативний логарифм вірогідності дискретних дій, обчисливши його softmax cross entropy. Як і у навчанні з учителем, можна використати стохастичний градієнтний спуск для отримання бажаної мінімізації.

```
### Loss function ###
```

```
2
3  # Arguments:
4  #   logits: network's predictions for actions to take
5  #   actions: the actions the agent took in an episode
6  #   rewards: the rewards the agent received in an episode
7  # Returns:
8  #   loss
9  def compute_loss(logits, actions, rewards):
10     '''Complete the function call to compute the negative log probabilities'''
11     neg_logprob = tf.nn.sparse_softmax_cross_entropy_with_logits(
12         logits=logits, labels=actions) # TODO
13
14     '''Scale the negative log probability by the rewards'''
15     loss = tf.reduce_mean( neg_logprob * rewards )
16     return loss
```

Тепер використаємо функцію втрат для визначення кроку навчання агента:

```
1  ### Training step (forward and backpropagation) ###
2
3  def train_step(model, loss_function, optimizer, observations, actions, discounted_rewards, custom_fwd_fn=No
4      with tf.GradientTape() as tape:
5          # Forward propagate through the agent network
6          if custom_fwd_fn is not None:
7              prediction = custom_fwd_fn(observations)
8          else:
9              prediction = model(observations)
10
11         '''call the compute_loss function to compute the loss'''
12         loss = loss_function(prediction, actions, discounted_rewards)
13
14     '''run backpropagation to minimize the loss using the tape.gradient method.
15             Unlike supervised learning, RL is *extremely* noisy, so you will benefit
16             from additionally clipping your gradients to avoid falling into
17             dangerous local minima. After computing your gradients try also clipping
18             by a global normalizer. Try different clipping values, usually clipping
19             between 0.5 and 5 provides reasonable results. '''
20     grads = tape.gradient(loss, model.trainable_variables)
21     grads, _ = tf.clip_by_global_norm(grads, 2)
22     optimizer.apply_gradients(zip(grads, model.trainable_variables))
23 ## Cartpole training! ##
24 ## Note: stoping and restarting this cell will pick up training where you
25 #        left off. To restart training you need to rerun the cell above as
26 #        well (to re-initialize the model and optimizer)
27
28 if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
29 for i_episode in range(500):
30
31     plotter.plot(smoothed_reward.get())
32     # Restart the environment
33     observation = env.reset()
34     memory.clear()
```

```
35
36      while True:
37          # using our observation, choose an action and take it in the environment
38          action = choose_action(cartpole_model, observation)
39          next_observation, reward, done, info = env.step(action)
40          # add to memory
41          memory.add_to_memory(observation, action, reward)
42
43          # is the episode over? did you crash or do so well that you're done?
44          if done:
45              # determine total reward and keep a record of this
46              total_reward = sum(memory.rewards)
47              smoothed_reward.append(total_reward)
48
49              # initiate training - remember we don't know anything about how the
50              #   agent is doing until it has crashed!
51              g = train_step(cartpole_model, compute_loss, optimizer,
52                          observations=np.vstack(memory.observations),
53                          actions=np.array(memory.actions),
54                          discounted_rewards = discount_rewards(memory.rewards))
55
56              # reset the memory
57              memory.clear()
58              break
59          # update our observatons
60          observation = next_observation
```

Оскільки у агента немає попередніх знань про середовище, він почне навчатися балансувати стовп на візку, базуючись лише на зворотньому зв'язку, отриманому від середовища! Ми побачимо як агент поступово навчається дотримуватись стратегії для оптимізації балансування стовпа так довго, як тільки це можливо. Для цього ми будемо спостерігати, як змінюються загальні винагороди агента в залежності від епізодів. У ході навчання загальні винагороди мають зростати. Допоміжні функції, які будуть використані подано нижче:

```
1  # Helper functions
2  import matplotlib.pyplot as plt
3  import tensorflow as tf
4  import time
5  import numpy as np
6
7  from IPython import display as ipythondisplay
8  from string import Formatter
9
10
11
12
13
14 def display_model(model):
15   tf.keras.utils.plot_model(model,
16               to_file='tmp.png',
17               show_shapes=True)
18   return ipythondisplay.Image('tmp.png')
19
```

```
20
21  def plot_sample(x,y,vae):
22      plt.figure(figsize=(2,1))
23      plt.subplot(1, 2, 1)
24
25      idx = np.where(y==1)[0][0]
26      plt.imshow(x[idx])
27      plt.grid(False)
28
29      plt.subplot(1, 2, 2)
30      _, _, _, recon = vae(x)
31      recon = np.clip(recon, 0, 1)
32      plt.imshow(recon[idx])
33      plt.grid(False)
34
35      plt.show()
36
37
38
39  class LossHistory:
40    def __init__(self, smoothing_factor=0.0):
41      self.alpha = smoothing_factor
42      self.loss = []
43    def append(self, value):
44      self.loss.append( self.alpha*self.loss[-1] + (1-self.alpha)*value if len(self.loss)>0 else value )
45    def get(self):
46      return self.loss
47
48  class PeriodicPlotter:
49    def __init__(self, sec, xlabel='', ylabel='', scale=None):
50
51      self.xlabel = xlabel
52      self.ylabel = ylabel
53      self.sec = sec
54      self.scale = scale
55
56      self.tic = time.time()
57
58    def plot(self, data):
59      if time.time() - self.tic > self.sec:
60        plt.cla()
61
62        if self.scale is None:
63          plt.plot(data)
64        elif self.scale == 'semilogx':
65          plt.semilogx(data)
66        elif self.scale == 'semilogy':
67          plt.semilogy(data)
68        elif self.scale == 'loglog':
69          plt.loglog(data)
70        else:
```

```
        raise ValueError("unrecognized parameter scale {}".format(self.scale))

      plt.xlabel(self.xlabel); plt.ylabel(self.ylabel)
      ipythondisplay.clear_output(wait=True)
      ipythondisplay.display(plt.gcf())

      self.tic = time.time()
```

Задання оптимізатора, швидкості навчання агента тощо:

```
## Training parameters ##
## Re-run this cell to restart training from scratch ##

# Learning rate and optimizer
learning_rate = 1e-3
optimizer = tf.keras.optimizers.Adam(learning_rate)

# instantiate cartpole agent
cartpole_model = create_cartpole_model()

# to track our progress
smoothed_reward = LossHistory(smoothing_factor=0.95)
plotter = PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Rewards')
```

Навчання агента:

```
### Training step (forward and backpropagation) ###

def train_step(model, loss_function, optimizer, observations, actions, discounted_rewards, custom_fwd_fn=No
    with tf.GradientTape() as tape:
        # Forward propagate through the agent network
        if custom_fwd_fn is not None:
            prediction = custom_fwd_fn(observations)
        else:
            prediction = model(observations)

        '''call the compute_loss function to compute the loss'''
        loss = loss_function(prediction, actions, discounted_rewards)

    '''run backpropagation to minimize the loss using the tape.gradient method.
             Unlike supervised learning, RL is *extremely* noisy, so you will benefit
             from additionally clipping your gradients to avoid falling into
             dangerous local minima. After computing your gradients try also clipping
             by a global normalizer. Try different clipping values, usually clipping
             between 0.5 and 5 provides reasonable results. '''
    grads = tape.gradient(loss, model.trainable_variables)
    grads, _ = tf.clip_by_global_norm(grads, 2)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
## Cartpole training! ##
## Note: stoping and restarting this cell will pick up training where you
#        left off. To restart training you need to rerun the cell above as
#        well (to re-initialize the model and optimizer)

if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
```

```
for i_episode in range(500):

    plotter.plot(smoothed_reward.get())
    # Restart the environment
    observation = env.reset()
    memory.clear()

    while True:
        # using our observation, choose an action and take it in the environment
        action = choose_action(cartpole_model, observation)
        next_observation, reward, done, info = env.step(action)
        # add to memory
        memory.add_to_memory(observation, action, reward)

        # is the episode over? did you crash or do so well that you're done?
        if done:
            # determine total reward and keep a record of this
            total_reward = sum(memory.rewards)
            smoothed_reward.append(total_reward)

            # initiate training - remember we don't know anything about how the
            #   agent is doing until it has crashed!
            g = train_step(cartpole_model, compute_loss, optimizer,
                       observations=np.vstack(memory.observations),
                       actions=np.array(memory.actions),
                       discounted_rewards = discount_rewards(memory.rewards))

            # reset the memory
            memory.clear()
            break
        # update our observatons
        observation = next_observation
```

Деталі щодо середовищ можна переглянути за наступними посиланнями з офіційного ресурсу:

1. Pong: https://www.gymlibrary.dev/environments/atari/pong/?highlight=pongpong
2. Cart Pole: https://www.gymlibrary.dev/environments/classic_control/cart_pole/
3. Mountain Car: https://www.gymlibrary.dev/environments/classic_control/mountain_car/

Завдання для виконання

Завдання 1. Подайте короткі власні відповіді на наступні питання:

1. Як показав себе агент на практиці для кожного середовища?
2. Чи можна витратити менше часу на тренування та все ще отримати гарний результат?
3. Чи вважаєте ви, що збільшуючи час навчання агента ми могли б покращити результат ще більше?
4. Як складність між Pong та Cartpole впливає на швидкість тренування та загальний результат?
5. Що можна змінити в агенті або процесі навчання для того аби покращити результат?

Завдання 2. Створити за аналогією попереднього прикладу середовище `MountainCar-v0` та навчити агента досягати поставленої мети у цьому середовищі.

Оцінювання

Ваша оцінка за виконання практичної буде залежати від:

- 15% – подано короткі власні відповіді на питання
- 50% – створено середовище `MountainCar-v0` та навчено агента досягати поставлену мету у цьому середовищі
- 35% – підготовлено звіт: описано процес створення середовища `MountainCar-v0` та процес навчання агента у цьому середовищі

Шаблон LaTeX

Шаблон за яким потрібно підготувати звіт можна звантажити ТУТ. Якщо не бажаєте установлювати додаткове програмне забезпечення – можете скористатися для підготовки звіту цим онлайн-ресурсом: www.overleaf.com.

Структура цього шаблону:

```
...
├── code     # Директорія для файлів з кодом (лістинг коду у звіті)
│   ├── test.py
│   └── hello.c
├── images    # Основна директорія для зображень (.pdf, .jpg, .png)
│   └── NN_.png
├── logo      # Директорія для логотипу
│   └── LogoKPI.jpg
├── main.pdf       # Файл отриманий внаслідок компіляції main.tex
├── main.tex       # TeX  файл звіту
├── ref.bib         # Бібліографія (BibTeX)
└── report.cls    # Файл налаштування стилів звіту
```

Відправлення роботи на перевірку

Дедлайн: 25 жовтня 2023 року о 23:59

- Створіть архів `Прізвище Ім'я_група.zip` у який включіть:
  - Ваш звіт (.pdf файл) та решту файлів LaTeX
  - Скрипт навчання агента у середовищі `MountainCar`: `MountainCar_Прізвище Ім'я_група.ipynb`

  Після чого відправте цей архів на перевірку сюди: https://cloud.comsys.kpi.ua/s/XFJXKraXFx7jkeD