Технологии машинного обучения. Лабораторная работа №7

Студент: Уляшин В.В.

Группа: ИУ5-23М

!pip install swig

!pipinstallgym[box2d]

!pipinstallpyglet

Looking in indexes: [https://pypi.org/simple,](https://pypi.org/simple)<https://us-python.pkg.dev/colab-wheels/public/simple/>Requirement already satisfied: swig in /usr/local/lib/python3.10/dist-packages (4.1.1)

Looking in indexes: [https://pypi.org/simple,](https://pypi.org/simple)<https://us-python.pkg.dev/colab-wheels/public/simple/>

Requirement already satisfied: gym[box2d] in /usr/local/lib/python3.10/dist-packages (0.25.2)

Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gym[box2d]) (1.22.4)

Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym[box2d]) (2.2.1)

Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym[box2d]) (0.0.8)

Requirement already satisfied: box2d-py==2.3.5 in /usr/local/lib/python3.10/dist-packages (from gym[box2d]) (2.3.5)

Requirement already satisfied: pygame==2.1.0 in /usr/local/lib/python3.10/dist-packages (from gym[box2d]) (2.1.0)

Requirement already satisfied: swig==4.\* in /usr/local/lib/python3.10/dist-packages (from gym[box2d]) (4.1.1)

Looking in indexes: [https://pypi.org/simple,](https://pypi.org/simple)<https://us-python.pkg.dev/colab-wheels/public/simple/>

Requirement already satisfied: pyglet in /usr/local/lib/python3.10/dist-packages (2.0.7)

%%bash # Install additional packages for visualization sudo apt-get install -y python-opengl > /dev/null 2>&1 pip install git+https://github.com/tensorflow/docs > /dev/null 2>&1

import collections import gym import numpy as np import statistics import tensorflow as tf import tqdm

from matplotlib import pyplot as plt from tensorflow.keras import layers from typing import Any, List, Sequence, Tuple

# Create the environment env = gym.make("LunarLander-v2")

# Set seed for experiment reproducibility seed = 42 tf.random.set\_seed(seed) np.random.seed(seed)

# Small epsilon value for stabilizing division operations eps = np.finfo(np.float32).eps.item()

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should\_run\_async` will not call `transform\_ and should\_run\_async(code)

/usr/local/lib/python3.10/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initializing wrapper in old step API which retur deprecation(

/usr/local/lib/python3.10/dist-packages/gym/wrappers/step\_api\_compatibility.py:39: DeprecationWarning: WARN: Initializing environme deprecation(

Модель

class ActorCritic(tf.keras.Model): """Combined actor-critic network."""

def \_\_init\_\_( self, num\_actions: int, num\_hidden\_units: int): """Initialize.""" super().\_\_init\_\_()

self.common = layers.Dense(num\_hidden\_units, activation="relu") self.actor = layers.Dense(num\_actions) self.critic = layers.Dense(1)

def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:

x = self.common(inputs)

return self.actor(x), self.critic(x)

num\_actions = env.action\_space.n # 2 num\_hidden\_units = 128 model = ActorCritic(num\_actions, num\_hidden\_units)

# Сбор обучающих данных

# Wrap Gym's `env.step` call as an operation in a TensorFlow function. # This would allow it to be included in a callable TensorFlow graph.

def env\_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:

"""Returns state, reward and done flag given an action."""

state, reward, done, truncated = env.step(action) return (state.astype(np.float32), np.array(reward, np.int32), np.array(done, np.int32))

def tf\_env\_step(action: tf.Tensor) -> List[tf.Tensor]:

return tf.numpy\_function(env\_step, [action], [tf.float32, tf.int32, tf.int32])

def run\_episode( initial\_state: tf.Tensor, model: tf.keras.Model, max\_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]: """Runs a single episode to collect training data."""

action\_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic\_size=True) values = tf.TensorArray(dtype=tf.float32, size=0, dynamic\_size=True) rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic\_size=True)

initial\_state\_shape = initial\_state.shape state = initial\_state

for t in tf.range(max\_steps):

# Convert state into a batched tensor (batch size = 1) state = tf.expand\_dims(state, 0)

# Run the model and to get action probabilities and critic value action\_logits\_t, value = model(state)

# Sample next action from the action probability distribution action = tf.random.categorical(action\_logits\_t, 1)[0, 0] action\_probs\_t = tf.nn.softmax(action\_logits\_t)

# Store critic values values = values.write(t, tf.squeeze(value))

# Store log probability of the action chosen

action\_probs = action\_probs.write(t, action\_probs\_t[0, action])

# Apply action to the environment to get next state and reward state, reward, done = tf\_env\_step(action) state.set\_shape(initial\_state\_shape)

# Store reward rewards = rewards.write(t, reward)

if tf.cast(done, tf.bool): break

action\_probs = action\_probs.stack() values = values.stack() rewards = rewards.stack()

return action\_probs, values, rewards

def get\_expected\_return( rewards: tf.Tensor, gamma: float, standardize: bool = True) -> tf.Tensor: """Compute expected returns per timestep.""" n = tf.shape(rewards)[0]

returns = tf.TensorArray(dtype=tf.float32, size=n)

# Start from the end of `rewards` and accumulate reward sums

# into the `returns` array rewards = tf.cast(rewards[::-1], dtype=tf.float32) discounted\_sum = tf.constant(0.0) discounted\_sum\_shape = discounted\_sum.shape for i in tf.range(n): reward = rewards[i] discounted\_sum = reward + gamma \* discounted\_sum discounted\_sum.set\_shape(discounted\_sum\_shape) returns = returns.write(i, discounted\_sum) returns = returns.stack()[::-1]

if standardize:

returns = ((returns - tf.math.reduce\_mean(returns)) / (tf.math.reduce\_std(returns) + eps)) return returns Actor-Critic loss

huber\_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute\_loss( action\_probs: tf.Tensor, values: tf.Tensor, returns: tf.Tensor) -> tf.Tensor: """Computes the combined Actor-Critic loss.""" advantage = returns - values

action\_log\_probs = tf.math.log(action\_probs) actor\_loss = -tf.math.reduce\_sum(action\_log\_probs \* advantage) critic\_loss = huber\_loss(values, returns) return actor\_loss + critic\_loss

# Функция шага обучения

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.01)

@tf.function def train\_step( initial\_state: tf.Tensor, model: tf.keras.Model, optimizer: tf.keras.optimizers.Optimizer, gamma: float, max\_steps\_per\_episode: int) -> tf.Tensor: """Runs a model training step.""" with tf.GradientTape() as tape:

# Run the model for one episode to collect training data action\_probs, values, rewards = run\_episode( initial\_state, model, max\_steps\_per\_episode)

# Calculate the expected returns returns = get\_expected\_return(rewards, gamma)

# Convert training data to appropriate TF tensor shapes action\_probs, values, returns = [ tf.expand\_dims(x, 1) for x in [action\_probs, values, returns]]

# Calculate the loss values to update our network loss = compute\_loss(action\_probs, values, returns)

# Compute the gradients from the loss grads = tape.gradient(loss, model.trainable\_variables)

# Apply the gradients to the model's parameters optimizer.apply\_gradients(zip(grads, model.trainable\_variables)) episode\_reward = tf.math.reduce\_sum(rewards) return episode\_reward

# Цикл обучения

%%time

min\_episodes\_criterion = 100 max\_episodes = 10000 max\_steps\_per\_episode = 500

# `CartPole-v1` is considered solved if average reward is >= 475 over 500

# consecutive trials reward\_threshold = 140 running\_reward = 0

# The discount factor for future rewards gamma = 0.99

# Keep the last episodes reward

episodes\_reward: collections.deque = collections.deque(maxlen=min\_episodes\_criterion)

t = tqdm.trange(max\_episodes) for i in t:

initial\_state = env.reset()

initial\_state = tf.constant(initial\_state, dtype=tf.float32) episode\_reward = int(train\_step( initial\_state, model, optimizer, gamma, max\_steps\_per\_episode))

episodes\_reward.append(episode\_reward) running\_reward = statistics.mean(episodes\_reward)

t.set\_postfix( episode\_reward=episode\_reward, running\_reward=running\_reward)

# Show the average episode reward every 10 episodes if i % 10 == 0:

pass # print(f'Episode {i}: average reward: {avg\_reward}')

if running\_reward > reward\_threshold and i >= min\_episodes\_criterion: break print(f'\nSolved at episode {i}: average reward: {running\_reward:.2f}!')

[е](https://colab.research.google.com/signup?utm_source=footer&utm_medium=link&utm_campaign=footer_links)

Платны

[ы](https://colab.research.google.com/signup?utm_source=footer&utm_medium=link&utm_campaign=footer_links)

продукт

[b](https://colab.research.google.com/signup?utm_source=footer&utm_medium=link&utm_campaign=footer_links)

Cola

-

[ь](https://colab.research.google.com/cancel-subscription)

Отменит

[у](https://colab.research.google.com/cancel-subscription)

подписк

50%|████

▉

| 4951/10000 [22:16<22:42, 3.71it/s, episode\_reward=197, running\_reward=140]

Solved at episode 4951: average reward: 140.12!

CPU times: user 24min 56s, sys: 29.8 s, total: 25min 26s

Wall time: 22min 16s

Визуализация

# Render an episode and save as a GIF file

from

IPython

import

display

as

ipythondisplay

from

PIL

import

Image

render\_env = gym.make(

"LunarLander-v2"

, render\_mode=

'rgb\_array'

)

def

render\_episode

(

env

:

gym

.

Env

,

model

:

tf

.

keras

.

Model

,

max\_steps

:

int

):

state = env.reset()

state = tf.constant(state, dtype=tf.float32)

screen = env.render()

images = [Image.fromarray(screen[

0

])]

for

i

in

range

(

1

, max\_steps +

1

):

state = tf.expand\_dims(state,

0

)

action\_probs, \_ = model(state)

action = np.argmax(np.squeeze(action\_probs))

state, reward, done, truncated = env.step(acti

on)

state = tf.constant(state, dtype=tf.float32)

# Render screen every 10 steps

if

i %

10

==

0

:

screen = env.render()

images.append(Image.fromarray(screen[

0

]))

if

done:

break

return

images

# Save GIF image

images = render\_episode(render\_env, model, max\_ste

ps\_per\_episode)

image\_file =

'cartpole-v1.gif'

# loop=0: loop forever, duration=1: play each fram

e for 1ms

images[

0

]

.save

(

image\_file, save\_all=

True

, append\_images=images[

1

:]

, loop

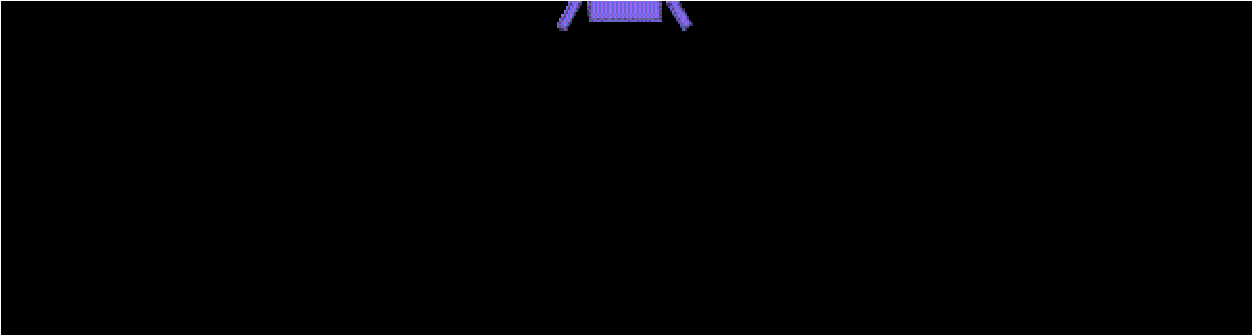
=

0

, duration=

1

)



import

tensorflow\_docs.vis.embed

as

embed

embed.embed\_file(image\_file)