Лабораторная работа №1.

Основы работы с библиотеками Gym, Tensorflow и PyTorch

Библиотека Gym

1. Установка

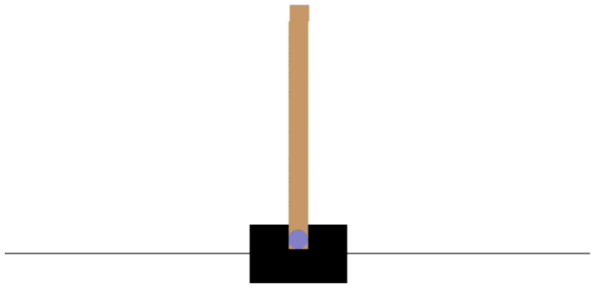
```
pip install gym
```

2. Инициализация и визуализация среды.

Запускаем экземпляр среды ***CartPole-v0*** для 1000 меток времени (шагов), отображаем среду для каждого шага.

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

После запуска появляется окно с визуализацией классической проблемы с перевернутым маятником.

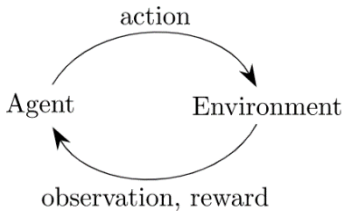


3. Наблюдения (Observations)

Функция step среды возвращает состояние среды. На самом деле step возвращает четыре значения:

- **observation** (object): объект, относящийся к среде. Например, пиксельные данные с камеры, углы и скорости суставов робота или состояние доски в настольной игре.
- **reward** (float): сумма вознаграждения, полученного за предыдущее действие.
- **done** (Boolean): флаг того, пришло ли время снова сбросить настройки среды. Большинство (но не все) задач разделены на четко определенные эпизоды, а выполненное значение True указывает на завершение эпизода. (Например, возможно, шест наклонился слишком далеко, или вы потеряли свою последнюю жизнь.)
- **info** (dict): диагностическая информация, полезная для отладки. Иногда она может быть полезна для обучения (например, она может содержать необработанные вероятности последнего изменения состояния среды).

Ниже пример реализации классического «цикла агент-среда». На каждом временном шаге агент выбирает действие, а среда возвращает наблюдение и вознаграждение.



Процесс стартует путем вызова функции reset(), которая возвращает первичное наблюдение. Таким образом, правильнее переписать предыдущий код для учета флага done:

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

В Выводе будет указано, на каком шаге закончился каждый эпизод:

…
[-0.07825026 0.17098394 0.17772266 0.04518264]
[-0.07483058 -0.02618224 0.17862631 0.38824934]
[-0.07535423 0.16601425 0.1863913 0.15677905]
[-0.07203394 -0.03121928 0.18952687 0.50198734]
[-0.07265832 -0.22843622 0.19956662 0.84790343]
Episode finished after 16 timesteps

3. Пространства (Spaces)

Каждая среда взаимодействует с двумя пространствами: **action_space** и **observation_space**. Эти пространства являются атрибутами среды, и они описывают формат валидных действий и наблюдений:

```
import gym
env = gym.make('CartPole-v0')
print(env.action_space)
#> Discrete(2)
print(env.observation_space)
#> Box(4,)
```

Вывод:

Discrete(2)
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)

Дискретное пространство допускает фиксированный диапазон неотрицательных чисел, поэтому в этом случае допустимыми действиями являются либо 0, либо 1. Пространство Box представляет собой n-мерный блок, поэтому действительные наблюдения будут массивом из 4 чисел. Мы также можем проверить границы Box:

```
print(env.observation_space.high)
#> array([ 2.4       ,          inf,  0.20943951,          inf])
print(env.observation_space.low)
#> array([-2.4       ,         -inf, -0.20943951,         -inf])
```

Вывод:

[4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]

Этот самоанализ может быть полезен для написания универсального кода, который работает во многих различных средах. Box и Discrete являются наиболее распространенными пространствами. Вы можете взять пример из пространства или проверить, что что-то ему принадлежит:

```
from gym import spaces
space = spaces.Discrete(8) # Set with 8 elements {0, 1, 2, ..., 7}
x = space.sample()
assert space.contains(x)
assert space.n == 8
```

Для получения списка зарегистрированных сред можно выполнить:

```
from gym import envs
print(envs.registry.all())
```

Вывод:

dict_values([EnvSpec(CartPole-v0), EnvSpec(CartPole-v1), EnvSpec(MountainCar-v0), EnvSpec(MountainCarContinuous-v0), EnvSpec(Pendulum-v1), EnvSpec(Acrobot-v1), EnvSpec(LunarLander-v2), EnvSpec(LunarLanderContinuous-v2), EnvSpec(BipedalWalker-v3), EnvSpec(BipedalWalkerHardcore-v3), EnvSpec(CarRacing-v0), EnvSpec(Blackjack-v1), …)

Задание №1

Создать среду Taxi-v3, отобразить кадры среды, значение временного шага, код состояния, код действия, значение вознаграждения для 10 случайных действий.

Библиотека Tensorflow

1. Установка

```
pip install tensorflow
```

2. Импорт библиотеки

```
import tensorflow as tf

from tensorflow.keras.layers import Dense, Flatten, Conv2D
from tensorflow.keras import Model
```

3. Загрузка и подготовка набора данных MNIST

http://yann.lecun.com/exdb/mnist/

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, Conv2D
from tensorflow.keras import Model

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Add a channels dimension
x_train = x_train[..., tf.newaxis].astype("float32")
x_test = x_test[..., tf.newaxis].astype("float32")
```

Вывод:

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 1s 0us/step
11501568/11490434 [==============================] - 1s 0us/step

4. Создание тренировочного и тестового наборов данных

Использование **tf.data** для пакетной обработки и перемешивания набора данных:

```
train_ds = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train)).shuffle(10000).batch(32)

test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(32)
```

5. Создание модели

Создание модели **tf.keras** , используя API подкласса модели Keras:

```
class MyModel(Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.conv1 = Conv2D(32, 3, activation='relu')
    self.flatten = Flatten()
    self.d1 = Dense(128, activation='relu')
    self.d2 = Dense(10)

  def call(self, x):
    x = self.conv1(x)
    x = self.flatten(x)
    x = self.d1(x)
    return self.d2(x)

# Create an instance of the model
model = MyModel()
```

Выбор оптимизатора и функции потерь для обучения:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

optimizer = tf.keras.optimizers.Adam()
```

Выбор показателей для измерения потерь и точности модели. Эти метрики накапливают значения за эпохи, а затем выводят общий результат.

```
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy =
tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy =
tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

6. Обучение и тестирование модели

Использование **tf.GradientTape** для обучения модели:

```
@tf.function
def train_step(images, labels):
  with tf.GradientTape() as tape:
    # training=True is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=True)
    loss = loss_object(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_accuracy(labels, predictions)
```

Тестирование модели:

```
@tf.function
def test_step(images, labels):
  # training=False is only needed if there are layers with different
  # behavior during training versus inference (e.g. Dropout).
  predictions = model(images, training=False)
  t_loss = loss_object(labels, predictions)

  test_loss(t_loss)
  test_accuracy(labels, predictions)
```

```
EPOCHS = 5

for epoch in range(EPOCHS):
  # Reset the metrics at the start of the next epoch
  train_loss.reset_states()
  train_accuracy.reset_states()
  test_loss.reset_states()
  test_accuracy.reset_states()

  for images, labels in train_ds:
    train_step(images, labels)

  for test_images, test_labels in test_ds:
    test_step(test_images, test_labels)

  print(
    f'Epoch {epoch + 1}, '
    f'Loss: {train_loss.result()}, '
    f'Accuracy: {train_accuracy.result() * 100}, '
    f'Test Loss: {test_loss.result()}, '
    f'Test Accuracy: {test_accuracy.result() * 100}'
  )
```

Вывод:

Epoch 1, Loss: 0.14055445790290833, Accuracy: 95.85333251953125, Test Loss: 0.06531758606433868, Test Accuracy: 97.93000030517578
Epoch 2, Loss: 0.04441937804222107, Accuracy: 98.68000030517578, Test Loss: 0.05592753738164902, Test Accuracy: 98.13999938964844
Epoch 3, Loss: 0.023894023150205612, Accuracy: 99.23833465576172, Test Loss: 0.057369183748960495, Test Accuracy: 98.27999877929688
Epoch 4, Loss: 0.013493617996573448, Accuracy: 99.58000183105469, Test Loss: 0.0697859525680542, Test Accuracy: 98.1199951171875
Epoch 5, Loss: 0.009707454591989517, Accuracy: 99.66999816894531, Test Loss: 0.06326959282159805, Test Accuracy: 98.2699966430664

Задание №2

Используя библиотеку tensorflow, создайте, обучите и оцените свою модель, используя датасет, отличный от MNIST

Библиотека PyTorch

Задание №3

1. Самостоятельно изучить функции библиотеки PyTorch для Python: https://pytorch.org/get-started/locally/, https://pytorch.org/docs/stable/index.html

2. Используя библиотеку pytorch, повторить создание, обучение и оценку модели из Задания №2

Требование к отчету

1. В качестве отчета принимаются три Python файла (для каждого из заданий).
2. Код должен быть в достаточной мере прокомментирован.
3. Локальные датасеты также должны входить в отчет.
4. Отчет предоставляется в виде архива zip, формат имени архива: <номер группы>_<Фамилия_Имя>_LR<номер работы>.zip