Lecture 12 RL

November 25, 2020

Chapter 18 – Reinforcement Learning

This notebook contains all the sample code in chapter 18.

Run in Google Colab

1 Reinforcement Learning

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest dating from discrete optimization algorithms.

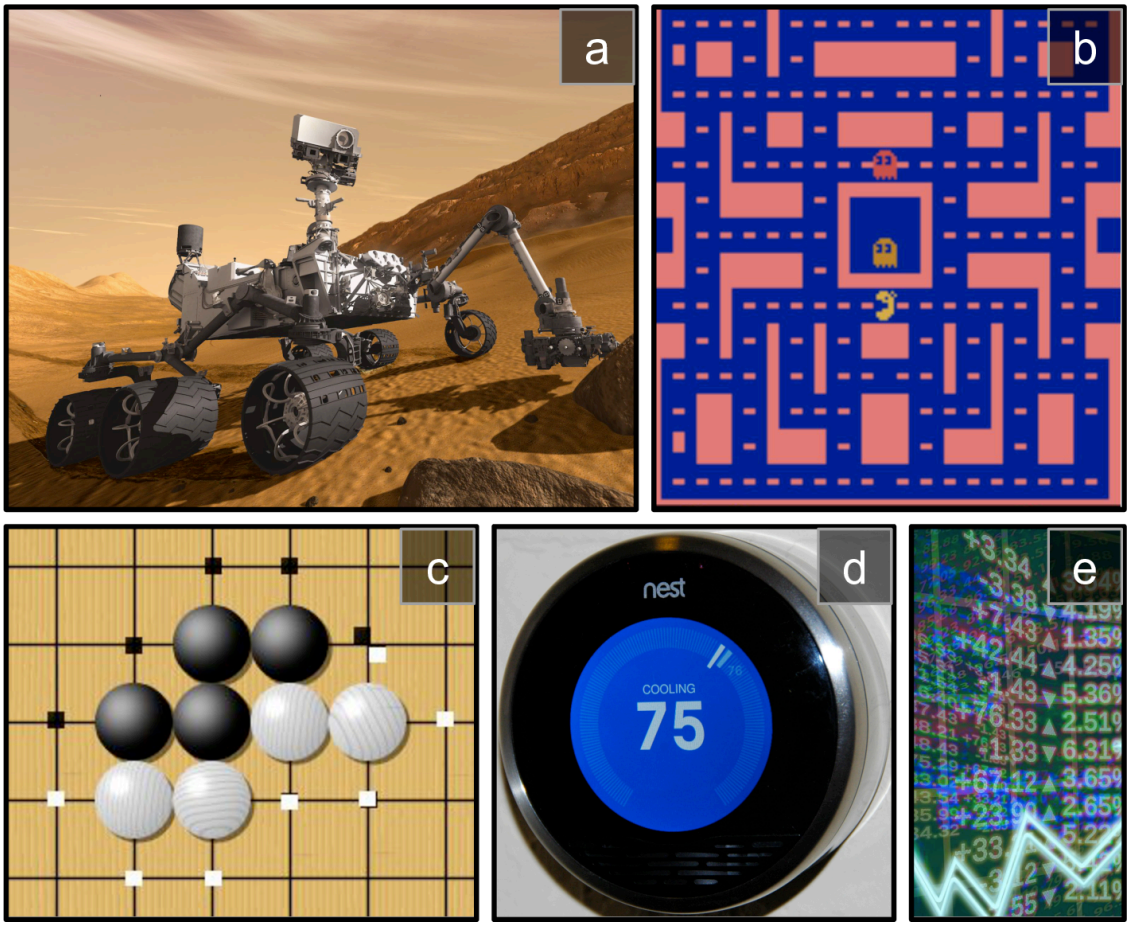
Recently it was mainly applied to games: chess, Go, etc. The idea is to learn without data, by trial and error.

We will cover the most efficient RL methods: policy gradients and deep Q-networks (DQN),including a discussion of Markov decision processes (MDP).

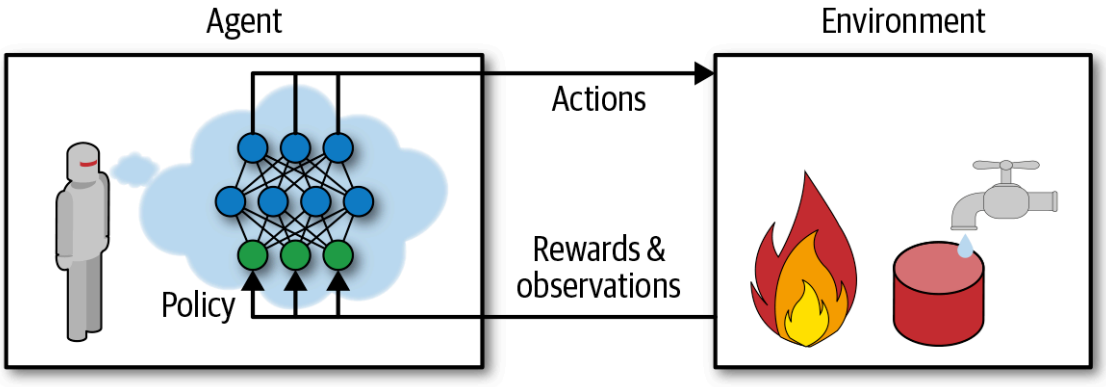
2 Learning to optimize rewards

In Reinforcement Learning, a software agent makes observations and takes actions within an environment, and in return it receives punishments and rewards. Its objective is to learn to act in a way that will maximize its expected long-term pleasure and minimize pain. Examples:

- The agent can be the program controlling a walking robot. The environment mimics real world, the agent observes the environment through a set of sensors such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time, goes in the wrong direction, or falls down. Training robots is real life is very expensive.
- The agent can be the program controlling PacMan. The environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
- Similarly, the agent can be the program playing a board game such as the game of Go.
- Decisions such as smart thermostat, getting rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
- The agent can observe stock market prices and decide how much to buy or sell every second.



There may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it better find the exit as quickly as possible.



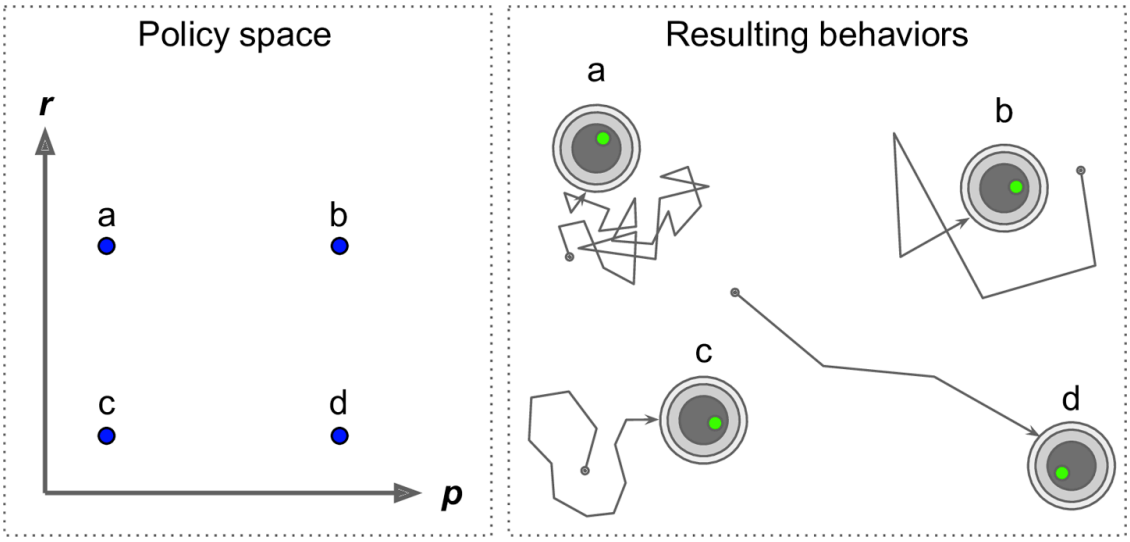
3 Policy Search

The algorithm used by the software agent to determine its actions is called its policy. For example, the policy could be a neural network taking observations as inputs and outputting the action to take.

Algorithms may involve some randomness called a stochastic policy. The policy may produce a range of outcome with probabilities associated with each of them. Then decisions will have different trajectories, but they will all try to achieve the final goal.

Imagine a vacuum-cleaner robot. There are just two policy parameters you can tweak: the probability of turn p and the angle between $-r$ and r:

- One learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best. This is an example of brute force approach. In large policy space it will not work.
- Another way to explore the policy space using genetic algorithms. We randomly create a first generation of 100 policies and try them out, then "kill" the 80 worst policies and make the 20 survivors produce 4 offspring each. An offspring is just a copy of its parent plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way, until you find a good policy.
- Another approach is to evaluate the gradients of the rewards with regards to the policy parameters, then tweaking these parameters by following the gradient toward higher rewards (gradient ascent). This approach is called policy gradients (PG). For the robot we could increase p and evaluate whether this increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase p some more, or else reduce p.



We do we need to create an environment for the agent to live in – OpenAI gym.

4 OpenAI Gym

- You need to have a working environment to train an agent (robot, gamer, pole) safely, cheap, and fast.
- OpenAI Gym simulate environments (Atari games, board games, 2D and 3D physical simulations. To install run:

[1]:
```
!pip install gym
```

```
Requirement already satisfied: gym in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (0.17.3)
Requirement already satisfied: scipy in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from gym) (1.4.1)
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from gym) (1.5.0)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from gym) (1.5.0)
Requirement already satisfied: numpy>=1.10.4 in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from gym) (1.18.5)
Requirement already satisfied: future in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from
pyglet<=1.5.0,>=1.4.0->gym) (0.18.2)
```

[22]:
```
# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)
import gym
# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
    !apt update && apt install -y libpq-dev libsdl2-dev swig xorg-dev xvfb
    !pip install -q -U tf-agents-nightly pyvirtualdisplay gym[atari]
    IS_COLAB = True
except Exception:
    IS_COLAB = False

# TensorFlow 2.0 is required
import tensorflow as tf
from tensorflow import keras
assert tf.__version__ >= "2.0"

if not tf.config.list_physical_devices('GPU'):
    print("No GPU was detected. CNNs can be very slow without a GPU.")
    if IS_COLAB:
        print("Go to Runtime > Change runtime and select a GPU hardware␣
 ↪accelerator.")

# Common imports
import numpy as np
import os
```

```
# to make this notebook's output stable across runs
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# To get smooth animations
import matplotlib.animation as animation
mpl.rc('animation', html='jshtml')

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "rl"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

```
No GPU was detected. CNNs can be very slow without a GPU.
```

5 Introduction to OpenAI gym

In this notebook we will be using OpenAI gym, a great toolkit for developing and comparing Reinforcement Learning algorithms. It provides many environments for your learning *agents* to interact with. Let's start by importing `gym`:

Let's list all the available environments:

[23]:
```
gym.envs.registry.all()
```

[23]:
```
dict_values([EnvSpec(Copy-v0), EnvSpec(RepeatCopy-v0),
EnvSpec(ReversedAddition-v0), EnvSpec(ReversedAddition3-v0),
EnvSpec(DuplicatedInput-v0), EnvSpec(Reverse-v0), EnvSpec(CartPole-v0),
EnvSpec(CartPole-v1), EnvSpec(MountainCar-v0),
EnvSpec(MountainCarContinuous-v0), EnvSpec(Pendulum-v0), EnvSpec(Acrobot-v1),
EnvSpec(LunarLander-v2), EnvSpec(LunarLanderContinuous-v2),
```

EnvSpec(BipedalWalker-v3), EnvSpec(BipedalWalkerHardcore-v3),
EnvSpec(CarRacing-v0), EnvSpec(Blackjack-v0), EnvSpec(KellyCoinflip-v0),
EnvSpec(KellyCoinflipGeneralized-v0), EnvSpec(FrozenLake-v0),
EnvSpec(FrozenLake8x8-v0), EnvSpec(CliffWalking-v0), EnvSpec(NChain-v0),
EnvSpec(Roulette-v0), EnvSpec(Taxi-v3), EnvSpec(GuessingGame-v0),
EnvSpec(HotterColder-v0), EnvSpec(Reacher-v2), EnvSpec(Pusher-v2),
EnvSpec(Thrower-v2), EnvSpec(Striker-v2), EnvSpec(InvertedPendulum-v2),
EnvSpec(InvertedDoublePendulum-v2), EnvSpec(HalfCheetah-v2),
EnvSpec(HalfCheetah-v3), EnvSpec(Hopper-v2), EnvSpec(Hopper-v3),
EnvSpec(Swimmer-v2), EnvSpec(Swimmer-v3), EnvSpec(Walker2d-v2),
EnvSpec(Walker2d-v3), EnvSpec(Ant-v2), EnvSpec(Ant-v3), EnvSpec(Humanoid-v2),
EnvSpec(Humanoid-v3), EnvSpec(HumanoidStandup-v2), EnvSpec(FetchSlide-v1),
EnvSpec(FetchPickAndPlace-v1), EnvSpec(FetchReach-v1), EnvSpec(FetchPush-v1),
EnvSpec(HandReach-v0), EnvSpec(HandManipulateBlockRotateZ-v0),
EnvSpec(HandManipulateBlockRotateZTouchSensors-v0),
EnvSpec(HandManipulateBlockRotateZTouchSensors-v1),
EnvSpec(HandManipulateBlockRotateParallel-v0),
EnvSpec(HandManipulateBlockRotateParallelTouchSensors-v0),
EnvSpec(HandManipulateBlockRotateParallelTouchSensors-v1),
EnvSpec(HandManipulateBlockRotateXYZ-v0),
EnvSpec(HandManipulateBlockRotateXYZTouchSensors-v0),
EnvSpec(HandManipulateBlockRotateXYZTouchSensors-v1),
EnvSpec(HandManipulateBlockFull-v0), EnvSpec(HandManipulateBlock-v0),
EnvSpec(HandManipulateBlockTouchSensors-v0),
EnvSpec(HandManipulateBlockTouchSensors-v1),
EnvSpec(HandManipulateEggRotate-v0),
EnvSpec(HandManipulateEggRotateTouchSensors-v0),
EnvSpec(HandManipulateEggRotateTouchSensors-v1),
EnvSpec(HandManipulateEggFull-v0), EnvSpec(HandManipulateEgg-v0),
EnvSpec(HandManipulateEggTouchSensors-v0),
EnvSpec(HandManipulateEggTouchSensors-v1), EnvSpec(HandManipulatePenRotate-v0),
EnvSpec(HandManipulatePenRotateTouchSensors-v0),
EnvSpec(HandManipulatePenRotateTouchSensors-v1),
EnvSpec(HandManipulatePenFull-v0), EnvSpec(HandManipulatePen-v0),
EnvSpec(HandManipulatePenTouchSensors-v0),
EnvSpec(HandManipulatePenTouchSensors-v1), EnvSpec(FetchSlideDense-v1),
EnvSpec(FetchPickAndPlaceDense-v1), EnvSpec(FetchReachDense-v1),
EnvSpec(FetchPushDense-v1), EnvSpec(HandReachDense-v0),
EnvSpec(HandManipulateBlockRotateZDense-v0),
EnvSpec(HandManipulateBlockRotateZTouchSensorsDense-v0),
EnvSpec(HandManipulateBlockRotateZTouchSensorsDense-v1),
EnvSpec(HandManipulateBlockRotateParallelDense-v0),
EnvSpec(HandManipulateBlockRotateParallelTouchSensorsDense-v0),
EnvSpec(HandManipulateBlockRotateParallelTouchSensorsDense-v1),
EnvSpec(HandManipulateBlockRotateXYZDense-v0),
EnvSpec(HandManipulateBlockRotateXYZTouchSensorsDense-v0),
EnvSpec(HandManipulateBlockRotateXYZTouchSensorsDense-v1),

EnvSpec(HandManipulateBlockFullDense-v0), EnvSpec(HandManipulateBlockDense-v0),
EnvSpec(HandManipulateBlockTouchSensorsDense-v0),
EnvSpec(HandManipulateBlockTouchSensorsDense-v1),
EnvSpec(HandManipulateEggRotateDense-v0),
EnvSpec(HandManipulateEggRotateTouchSensorsDense-v0),
EnvSpec(HandManipulateEggRotateTouchSensorsDense-v1),
EnvSpec(HandManipulateEggFullDense-v0), EnvSpec(HandManipulateEggDense-v0),
EnvSpec(HandManipulateEggTouchSensorsDense-v0),
EnvSpec(HandManipulateEggTouchSensorsDense-v1),
EnvSpec(HandManipulatePenRotateDense-v0),
EnvSpec(HandManipulatePenRotateTouchSensorsDense-v0),
EnvSpec(HandManipulatePenRotateTouchSensorsDense-v1),
EnvSpec(HandManipulatePenFullDense-v0), EnvSpec(HandManipulatePenDense-v0),
EnvSpec(HandManipulatePenTouchSensorsDense-v0),
EnvSpec(HandManipulatePenTouchSensorsDense-v1), EnvSpec(Adventure-v0),
EnvSpec(Adventure-v4), EnvSpec(AdventureDeterministic-v0),
EnvSpec(AdventureDeterministic-v4), EnvSpec(AdventureNoFrameskip-v0),
EnvSpec(AdventureNoFrameskip-v4), EnvSpec(Adventure-ram-v0), EnvSpec(Adventure-
ram-v4), EnvSpec(Adventure-ramDeterministic-v0), EnvSpec(Adventure-
ramDeterministic-v4), EnvSpec(Adventure-ramNoFrameskip-v0), EnvSpec(Adventure-
ramNoFrameskip-v4), EnvSpec(AirRaid-v0), EnvSpec(AirRaid-v4),
EnvSpec(AirRaidDeterministic-v0), EnvSpec(AirRaidDeterministic-v4),
EnvSpec(AirRaidNoFrameskip-v0), EnvSpec(AirRaidNoFrameskip-v4), EnvSpec(AirRaid-
ram-v0), EnvSpec(AirRaid-ram-v4), EnvSpec(AirRaid-ramDeterministic-v0),
EnvSpec(AirRaid-ramDeterministic-v4), EnvSpec(AirRaid-ramNoFrameskip-v0),
EnvSpec(AirRaid-ramNoFrameskip-v4), EnvSpec(Alien-v0), EnvSpec(Alien-v4),
EnvSpec(AlienDeterministic-v0), EnvSpec(AlienDeterministic-v4),
EnvSpec(AlienNoFrameskip-v0), EnvSpec(AlienNoFrameskip-v4), EnvSpec(Alien-
ram-v0), EnvSpec(Alien-ram-v4), EnvSpec(Alien-ramDeterministic-v0),
EnvSpec(Alien-ramDeterministic-v4), EnvSpec(Alien-ramNoFrameskip-v0),
EnvSpec(Alien-ramNoFrameskip-v4), EnvSpec(Amidar-v0), EnvSpec(Amidar-v4),
EnvSpec(AmidarDeterministic-v0), EnvSpec(AmidarDeterministic-v4),
EnvSpec(AmidarNoFrameskip-v0), EnvSpec(AmidarNoFrameskip-v4), EnvSpec(Amidar-
ram-v0), EnvSpec(Amidar-ram-v4), EnvSpec(Amidar-ramDeterministic-v0),
EnvSpec(Amidar-ramDeterministic-v4), EnvSpec(Amidar-ramNoFrameskip-v0),
EnvSpec(Amidar-ramNoFrameskip-v4), EnvSpec(Assault-v0), EnvSpec(Assault-v4),
EnvSpec(AssaultDeterministic-v0), EnvSpec(AssaultDeterministic-v4),
EnvSpec(AssaultNoFrameskip-v0), EnvSpec(AssaultNoFrameskip-v4), EnvSpec(Assault-
ram-v0), EnvSpec(Assault-ram-v4), EnvSpec(Assault-ramDeterministic-v0),
EnvSpec(Assault-ramDeterministic-v4), EnvSpec(Assault-ramNoFrameskip-v0),
EnvSpec(Assault-ramNoFrameskip-v4), EnvSpec(Asterix-v0), EnvSpec(Asterix-v4),
EnvSpec(AsterixDeterministic-v0), EnvSpec(AsterixDeterministic-v4),
EnvSpec(AsterixNoFrameskip-v0), EnvSpec(AsterixNoFrameskip-v4), EnvSpec(Asterix-
ram-v0), EnvSpec(Asterix-ram-v4), EnvSpec(Asterix-ramDeterministic-v0),
EnvSpec(Asterix-ramDeterministic-v4), EnvSpec(Asterix-ramNoFrameskip-v0),
EnvSpec(Asterix-ramNoFrameskip-v4), EnvSpec(Asteroids-v0),
EnvSpec(Asteroids-v4), EnvSpec(AsteroidsDeterministic-v0),

EnvSpec(AsteroidsDeterministic-v4), EnvSpec(AsteroidsNoFrameskip-v0), EnvSpec(AsteroidsNoFrameskip-v4), EnvSpec(Asteroids-ram-v0), EnvSpec(Asteroids-ram-v4), EnvSpec(Asteroids-ramDeterministic-v0), EnvSpec(Asteroids-ramDeterministic-v4), EnvSpec(Asteroids-ramNoFrameskip-v0), EnvSpec(Asteroids-ramNoFrameskip-v4), EnvSpec(Atlantis-v0), EnvSpec(Atlantis-v4), EnvSpec(AtlantisDeterministic-v0), EnvSpec(AtlantisDeterministic-v4), EnvSpec(AtlantisNoFrameskip-v0), EnvSpec(AtlantisNoFrameskip-v4), EnvSpec(Atlantis-ram-v0), EnvSpec(Atlantis-ram-v4), EnvSpec(Atlantis-ramDeterministic-v0), EnvSpec(Atlantis-ramDeterministic-v4), EnvSpec(Atlantis-ramNoFrameskip-v0), EnvSpec(Atlantis-ramNoFrameskip-v4), EnvSpec(BankHeist-v0), EnvSpec(BankHeist-v4), EnvSpec(BankHeistDeterministic-v0), EnvSpec(BankHeistDeterministic-v4), EnvSpec(BankHeistNoFrameskip-v0), EnvSpec(BankHeistNoFrameskip-v4), EnvSpec(BankHeist-ram-v0), EnvSpec(BankHeist-ram-v4), EnvSpec(BankHeist-ramDeterministic-v0), EnvSpec(BankHeist-ramDeterministic-v4), EnvSpec(BankHeist-ramNoFrameskip-v0), EnvSpec(BankHeist-ramNoFrameskip-v4), EnvSpec(BattleZone-v0), EnvSpec(BattleZone-v4), EnvSpec(BattleZoneDeterministic-v0), EnvSpec(BattleZoneDeterministic-v4), EnvSpec(BattleZoneNoFrameskip-v0), EnvSpec(BattleZoneNoFrameskip-v4), EnvSpec(BattleZone-ram-v0), EnvSpec(BattleZone-ram-v4), EnvSpec(BattleZone-ramDeterministic-v0), EnvSpec(BattleZone-ramDeterministic-v4), EnvSpec(BattleZone-ramNoFrameskip-v0), EnvSpec(BattleZone-ramNoFrameskip-v4), EnvSpec(BeamRider-v0), EnvSpec(BeamRider-v4), EnvSpec(BeamRiderDeterministic-v0), EnvSpec(BeamRiderDeterministic-v4), EnvSpec(BeamRiderNoFrameskip-v0), EnvSpec(BeamRiderNoFrameskip-v4), EnvSpec(BeamRider-ram-v0), EnvSpec(BeamRider-ram-v4), EnvSpec(BeamRider-ramDeterministic-v0), EnvSpec(BeamRider-ramDeterministic-v4), EnvSpec(BeamRider-ramNoFrameskip-v0), EnvSpec(BeamRider-ramNoFrameskip-v4), EnvSpec(Berzerk-v0), EnvSpec(Berzerk-v4), EnvSpec(BerzerkDeterministic-v0), EnvSpec(BerzerkDeterministic-v4), EnvSpec(BerzerkNoFrameskip-v0), EnvSpec(BerzerkNoFrameskip-v4), EnvSpec(Berzerk-ram-v0), EnvSpec(Berzerk-ram-v4), EnvSpec(Berzerk-ramDeterministic-v0), EnvSpec(Berzerk-ramDeterministic-v4), EnvSpec(Berzerk-ramNoFrameskip-v0), EnvSpec(Berzerk-ramNoFrameskip-v4), EnvSpec(Bowling-v0), EnvSpec(Bowling-v4), EnvSpec(BowlingDeterministic-v0), EnvSpec(BowlingDeterministic-v4), EnvSpec(BowlingNoFrameskip-v0), EnvSpec(BowlingNoFrameskip-v4), EnvSpec(Bowling-ram-v0), EnvSpec(Bowling-ram-v4), EnvSpec(Bowling-ramDeterministic-v0), EnvSpec(Bowling-ramDeterministic-v4), EnvSpec(Bowling-ramNoFrameskip-v0), EnvSpec(Bowling-ramNoFrameskip-v4), EnvSpec(Boxing-v0), EnvSpec(Boxing-v4), EnvSpec(BoxingDeterministic-v0), EnvSpec(BoxingDeterministic-v4), EnvSpec(BoxingNoFrameskip-v0), EnvSpec(BoxingNoFrameskip-v4), EnvSpec(Boxing-ram-v0), EnvSpec(Boxing-ram-v4), EnvSpec(Boxing-ramDeterministic-v0), EnvSpec(Boxing-ramDeterministic-v4), EnvSpec(Boxing-ramNoFrameskip-v0), EnvSpec(Boxing-ramNoFrameskip-v4), EnvSpec(Breakout-v0), EnvSpec(Breakout-v4), EnvSpec(BreakoutDeterministic-v0), EnvSpec(BreakoutDeterministic-v4), EnvSpec(BreakoutNoFrameskip-v0), EnvSpec(BreakoutNoFrameskip-v4), EnvSpec(Breakout-ram-v0), EnvSpec(Breakout-ram-v4), EnvSpec(Breakout-ramDeterministic-v0), EnvSpec(Breakout-ramDeterministic-v4), EnvSpec(Breakout-

ramNoFrameskip-v0), EnvSpec(Breakout-ramNoFrameskip-v4), EnvSpec(Carnival-v0),
EnvSpec(Carnival-v4), EnvSpec(CarnivalDeterministic-v0),
EnvSpec(CarnivalDeterministic-v4), EnvSpec(CarnivalNoFrameskip-v0),
EnvSpec(CarnivalNoFrameskip-v4), EnvSpec(Carnival-ram-v0), EnvSpec(Carnival-
ram-v4), EnvSpec(Carnival-ramDeterministic-v0), EnvSpec(Carnival-
ramDeterministic-v4), EnvSpec(Carnival-ramNoFrameskip-v0), EnvSpec(Carnival-
ramNoFrameskip-v4), EnvSpec(Centipede-v0), EnvSpec(Centipede-v4),
EnvSpec(CentipedeDeterministic-v0), EnvSpec(CentipedeDeterministic-v4),
EnvSpec(CentipedeNoFrameskip-v0), EnvSpec(CentipedeNoFrameskip-v4),
EnvSpec(Centipede-ram-v0), EnvSpec(Centipede-ram-v4), EnvSpec(Centipede-
ramDeterministic-v0), EnvSpec(Centipede-ramDeterministic-v4), EnvSpec(Centipede-
ramNoFrameskip-v0), EnvSpec(Centipede-ramNoFrameskip-v4),
EnvSpec(ChopperCommand-v0), EnvSpec(ChopperCommand-v4),
EnvSpec(ChopperCommandDeterministic-v0),
EnvSpec(ChopperCommandDeterministic-v4), EnvSpec(ChopperCommandNoFrameskip-v0),
EnvSpec(ChopperCommandNoFrameskip-v4), EnvSpec(ChopperCommand-ram-v0),
EnvSpec(ChopperCommand-ram-v4), EnvSpec(ChopperCommand-ramDeterministic-v0),
EnvSpec(ChopperCommand-ramDeterministic-v4), EnvSpec(ChopperCommand-
ramNoFrameskip-v0), EnvSpec(ChopperCommand-ramNoFrameskip-v4),
EnvSpec(CrazyClimber-v0), EnvSpec(CrazyClimber-v4),
EnvSpec(CrazyClimberDeterministic-v0), EnvSpec(CrazyClimberDeterministic-v4),
EnvSpec(CrazyClimberNoFrameskip-v0), EnvSpec(CrazyClimberNoFrameskip-v4),
EnvSpec(CrazyClimber-ram-v0), EnvSpec(CrazyClimber-ram-v4),
EnvSpec(CrazyClimber-ramDeterministic-v0), EnvSpec(CrazyClimber-
ramDeterministic-v4), EnvSpec(CrazyClimber-ramNoFrameskip-v0),
EnvSpec(CrazyClimber-ramNoFrameskip-v4), EnvSpec(Defender-v0),
EnvSpec(Defender-v4), EnvSpec(DefenderDeterministic-v0),
EnvSpec(DefenderDeterministic-v4), EnvSpec(DefenderNoFrameskip-v0),
EnvSpec(DefenderNoFrameskip-v4), EnvSpec(Defender-ram-v0), EnvSpec(Defender-
ram-v4), EnvSpec(Defender-ramDeterministic-v0), EnvSpec(Defender-
ramDeterministic-v4), EnvSpec(Defender-ramNoFrameskip-v0), EnvSpec(Defender-
ramNoFrameskip-v4), EnvSpec(DemonAttack-v0), EnvSpec(DemonAttack-v4),
EnvSpec(DemonAttackDeterministic-v0), EnvSpec(DemonAttackDeterministic-v4),
EnvSpec(DemonAttackNoFrameskip-v0), EnvSpec(DemonAttackNoFrameskip-v4),
EnvSpec(DemonAttack-ram-v0), EnvSpec(DemonAttack-ram-v4), EnvSpec(DemonAttack-
ramDeterministic-v0), EnvSpec(DemonAttack-ramDeterministic-v4),
EnvSpec(DemonAttack-ramNoFrameskip-v0), EnvSpec(DemonAttack-ramNoFrameskip-v4),
EnvSpec(DoubleDunk-v0), EnvSpec(DoubleDunk-v4),
EnvSpec(DoubleDunkDeterministic-v0), EnvSpec(DoubleDunkDeterministic-v4),
EnvSpec(DoubleDunkNoFrameskip-v0), EnvSpec(DoubleDunkNoFrameskip-v4),
EnvSpec(DoubleDunk-ram-v0), EnvSpec(DoubleDunk-ram-v4), EnvSpec(DoubleDunk-
ramDeterministic-v0), EnvSpec(DoubleDunk-ramDeterministic-v4),
EnvSpec(DoubleDunk-ramNoFrameskip-v0), EnvSpec(DoubleDunk-ramNoFrameskip-v4),
EnvSpec(ElevatorAction-v0), EnvSpec(ElevatorAction-v4),
EnvSpec(ElevatorActionDeterministic-v0),
EnvSpec(ElevatorActionDeterministic-v4), EnvSpec(ElevatorActionNoFrameskip-v0),
EnvSpec(ElevatorActionNoFrameskip-v4), EnvSpec(ElevatorAction-ram-v0),

EnvSpec(ElevatorAction-ram-v4), EnvSpec(ElevatorAction-ramDeterministic-v0), EnvSpec(ElevatorAction-ramDeterministic-v4), EnvSpec(ElevatorAction-ramNoFrameskip-v0), EnvSpec(ElevatorAction-ramNoFrameskip-v4), EnvSpec(Enduro-v0), EnvSpec(Enduro-v4), EnvSpec(EnduroDeterministic-v0), EnvSpec(EnduroDeterministic-v4), EnvSpec(EnduroNoFrameskip-v0), EnvSpec(EnduroNoFrameskip-v4), EnvSpec(Enduro-ram-v0), EnvSpec(Enduro-ram-v4), EnvSpec(Enduro-ramDeterministic-v0), EnvSpec(Enduro-ramDeterministic-v4), EnvSpec(Enduro-ramNoFrameskip-v0), EnvSpec(Enduro-ramNoFrameskip-v4), EnvSpec(FishingDerby-v0), EnvSpec(FishingDerby-v4), EnvSpec(FishingDerbyDeterministic-v0), EnvSpec(FishingDerbyDeterministic-v4), EnvSpec(FishingDerbyNoFrameskip-v0), EnvSpec(FishingDerbyNoFrameskip-v4), EnvSpec(FishingDerby-ram-v0), EnvSpec(FishingDerby-ram-v4), EnvSpec(FishingDerby-ramDeterministic-v0), EnvSpec(FishingDerby-ramDeterministic-v4), EnvSpec(FishingDerby-ramNoFrameskip-v0), EnvSpec(FishingDerby-ramNoFrameskip-v4), EnvSpec(Freeway-v0), EnvSpec(Freeway-v4), EnvSpec(FreewayDeterministic-v0), EnvSpec(FreewayDeterministic-v4), EnvSpec(FreewayNoFrameskip-v0), EnvSpec(FreewayNoFrameskip-v4), EnvSpec(Freeway-ram-v0), EnvSpec(Freeway-ram-v4), EnvSpec(Freeway-ramDeterministic-v0), EnvSpec(Freeway-ramDeterministic-v4), EnvSpec(Freeway-ramNoFrameskip-v0), EnvSpec(Freeway-ramNoFrameskip-v4), EnvSpec(Frostbite-v0), EnvSpec(Frostbite-v4), EnvSpec(FrostbiteDeterministic-v0), EnvSpec(FrostbiteDeterministic-v4), EnvSpec(FrostbiteNoFrameskip-v0), EnvSpec(FrostbiteNoFrameskip-v4), EnvSpec(Frostbite-ram-v0), EnvSpec(Frostbite-ram-v4), EnvSpec(Frostbite-ramDeterministic-v0), EnvSpec(Frostbite-ramDeterministic-v4), EnvSpec(Frostbite-ramNoFrameskip-v0), EnvSpec(Frostbite-ramNoFrameskip-v4), EnvSpec(Gopher-v0), EnvSpec(Gopher-v4), EnvSpec(GopherDeterministic-v0), EnvSpec(GopherDeterministic-v4), EnvSpec(GopherNoFrameskip-v0), EnvSpec(GopherNoFrameskip-v4), EnvSpec(Gopher-ram-v0), EnvSpec(Gopher-ram-v4), EnvSpec(Gopher-ramDeterministic-v0), EnvSpec(Gopher-ramDeterministic-v4), EnvSpec(Gopher-ramNoFrameskip-v0), EnvSpec(Gopher-ramNoFrameskip-v4), EnvSpec(Gravitar-v0), EnvSpec(Gravitar-v4), EnvSpec(GravitarDeterministic-v0), EnvSpec(GravitarDeterministic-v4), EnvSpec(GravitarNoFrameskip-v0), EnvSpec(GravitarNoFrameskip-v4), EnvSpec(Gravitar-ram-v0), EnvSpec(Gravitar-ram-v4), EnvSpec(Gravitar-ramDeterministic-v0), EnvSpec(Gravitar-ramDeterministic-v4), EnvSpec(Gravitar-ramNoFrameskip-v0), EnvSpec(Gravitar-ramNoFrameskip-v4), EnvSpec(Hero-v0), EnvSpec(Hero-v4), EnvSpec(HeroDeterministic-v0), EnvSpec(HeroDeterministic-v4), EnvSpec(HeroNoFrameskip-v0), EnvSpec(HeroNoFrameskip-v4), EnvSpec(Hero-ram-v0), EnvSpec(Hero-ram-v4), EnvSpec(Hero-ramDeterministic-v0), EnvSpec(Hero-ramDeterministic-v4), EnvSpec(Hero-ramNoFrameskip-v0), EnvSpec(Hero-ramNoFrameskip-v4), EnvSpec(IceHockey-v0), EnvSpec(IceHockey-v4), EnvSpec(IceHockeyDeterministic-v0), EnvSpec(IceHockeyDeterministic-v4), EnvSpec(IceHockeyNoFrameskip-v0), EnvSpec(IceHockeyNoFrameskip-v4), EnvSpec(IceHockey-ram-v0), EnvSpec(IceHockey-ram-v4), EnvSpec(IceHockey-ramDeterministic-v0), EnvSpec(IceHockey-ramDeterministic-v4), EnvSpec(IceHockey-ramNoFrameskip-v0), EnvSpec(IceHockey-ramNoFrameskip-v4), EnvSpec(Jamesbond-v0),

EnvSpec(Jamesbond-v4), EnvSpec(JamesbondDeterministic-v0),
EnvSpec(JamesbondDeterministic-v4), EnvSpec(JamesbondNoFrameskip-v0),
EnvSpec(JamesbondNoFrameskip-v4), EnvSpec(Jamesbond-ram-v0), EnvSpec(Jamesbond-
ram-v4), EnvSpec(Jamesbond-ramDeterministic-v0), EnvSpec(Jamesbond-
ramDeterministic-v4), EnvSpec(Jamesbond-ramNoFrameskip-v0), EnvSpec(Jamesbond-
ramNoFrameskip-v4), EnvSpec(JourneyEscape-v0), EnvSpec(JourneyEscape-v4),
EnvSpec(JourneyEscapeDeterministic-v0), EnvSpec(JourneyEscapeDeterministic-v4),
EnvSpec(JourneyEscapeNoFrameskip-v0), EnvSpec(JourneyEscapeNoFrameskip-v4),
EnvSpec(JourneyEscape-ram-v0), EnvSpec(JourneyEscape-ram-v4),
EnvSpec(JourneyEscape-ramDeterministic-v0), EnvSpec(JourneyEscape-
ramDeterministic-v4), EnvSpec(JourneyEscape-ramNoFrameskip-v0),
EnvSpec(JourneyEscape-ramNoFrameskip-v4), EnvSpec(Kangaroo-v0),
EnvSpec(Kangaroo-v4), EnvSpec(KangarooDeterministic-v0),
EnvSpec(KangarooDeterministic-v4), EnvSpec(KangarooNoFrameskip-v0),
EnvSpec(KangarooNoFrameskip-v4), EnvSpec(Kangaroo-ram-v0), EnvSpec(Kangaroo-
ram-v4), EnvSpec(Kangaroo-ramDeterministic-v0), EnvSpec(Kangaroo-
ramDeterministic-v4), EnvSpec(Kangaroo-ramNoFrameskip-v0), EnvSpec(Kangaroo-
ramNoFrameskip-v4), EnvSpec(Krull-v0), EnvSpec(Krull-v4),
EnvSpec(KrullDeterministic-v0), EnvSpec(KrullDeterministic-v4),
EnvSpec(KrullNoFrameskip-v0), EnvSpec(KrullNoFrameskip-v4), EnvSpec(Krull-
ram-v0), EnvSpec(Krull-ram-v4), EnvSpec(Krull-ramDeterministic-v0),
EnvSpec(Krull-ramDeterministic-v4), EnvSpec(Krull-ramNoFrameskip-v0),
EnvSpec(Krull-ramNoFrameskip-v4), EnvSpec(KungFuMaster-v0),
EnvSpec(KungFuMaster-v4), EnvSpec(KungFuMasterDeterministic-v0),
EnvSpec(KungFuMasterDeterministic-v4), EnvSpec(KungFuMasterNoFrameskip-v0),
EnvSpec(KungFuMasterNoFrameskip-v4), EnvSpec(KungFuMaster-ram-v0),
EnvSpec(KungFuMaster-ram-v4), EnvSpec(KungFuMaster-ramDeterministic-v0),
EnvSpec(KungFuMaster-ramDeterministic-v4), EnvSpec(KungFuMaster-
ramNoFrameskip-v0), EnvSpec(KungFuMaster-ramNoFrameskip-v4),
EnvSpec(MontezumaRevenge-v0), EnvSpec(MontezumaRevenge-v4),
EnvSpec(MontezumaRevengeDeterministic-v0),
EnvSpec(MontezumaRevengeDeterministic-v4),
EnvSpec(MontezumaRevengeNoFrameskip-v0),
EnvSpec(MontezumaRevengeNoFrameskip-v4), EnvSpec(MontezumaRevenge-ram-v0),
EnvSpec(MontezumaRevenge-ram-v4), EnvSpec(MontezumaRevenge-ramDeterministic-v0),
EnvSpec(MontezumaRevenge-ramDeterministic-v4), EnvSpec(MontezumaRevenge-
ramNoFrameskip-v0), EnvSpec(MontezumaRevenge-ramNoFrameskip-v4),
EnvSpec(MsPacman-v0), EnvSpec(MsPacman-v4), EnvSpec(MsPacmanDeterministic-v0),
EnvSpec(MsPacmanDeterministic-v4), EnvSpec(MsPacmanNoFrameskip-v0),
EnvSpec(MsPacmanNoFrameskip-v4), EnvSpec(MsPacman-ram-v0), EnvSpec(MsPacman-
ram-v4), EnvSpec(MsPacman-ramDeterministic-v0), EnvSpec(MsPacman-
ramDeterministic-v4), EnvSpec(MsPacman-ramNoFrameskip-v0), EnvSpec(MsPacman-
ramNoFrameskip-v4), EnvSpec(NameThisGame-v0), EnvSpec(NameThisGame-v4),
EnvSpec(NameThisGameDeterministic-v0), EnvSpec(NameThisGameDeterministic-v4),
EnvSpec(NameThisGameNoFrameskip-v0), EnvSpec(NameThisGameNoFrameskip-v4),
EnvSpec(NameThisGame-ram-v0), EnvSpec(NameThisGame-ram-v4),
EnvSpec(NameThisGame-ramDeterministic-v0), EnvSpec(NameThisGame-

ramDeterministic-v4), EnvSpec(NameThisGame-ramNoFrameskip-v0),
EnvSpec(NameThisGame-ramNoFrameskip-v4), EnvSpec(Phoenix-v0),
EnvSpec(Phoenix-v4), EnvSpec(PhoenixDeterministic-v0),
EnvSpec(PhoenixDeterministic-v4), EnvSpec(PhoenixNoFrameskip-v0),
EnvSpec(PhoenixNoFrameskip-v4), EnvSpec(Phoenix-ram-v0), EnvSpec(Phoenix-
ram-v4), EnvSpec(Phoenix-ramDeterministic-v0), EnvSpec(Phoenix-
ramDeterministic-v4), EnvSpec(Phoenix-ramNoFrameskip-v0), EnvSpec(Phoenix-
ramNoFrameskip-v4), EnvSpec(Pitfall-v0), EnvSpec(Pitfall-v4),
EnvSpec(PitfallDeterministic-v0), EnvSpec(PitfallDeterministic-v4),
EnvSpec(PitfallNoFrameskip-v0), EnvSpec(PitfallNoFrameskip-v4), EnvSpec(Pitfall-
ram-v0), EnvSpec(Pitfall-ram-v4), EnvSpec(Pitfall-ramDeterministic-v0),
EnvSpec(Pitfall-ramDeterministic-v4), EnvSpec(Pitfall-ramNoFrameskip-v0),
EnvSpec(Pitfall-ramNoFrameskip-v4), EnvSpec(Pong-v0), EnvSpec(Pong-v4),
EnvSpec(PongDeterministic-v0), EnvSpec(PongDeterministic-v4),
EnvSpec(PongNoFrameskip-v0), EnvSpec(PongNoFrameskip-v4), EnvSpec(Pong-ram-v0),
EnvSpec(Pong-ram-v4), EnvSpec(Pong-ramDeterministic-v0), EnvSpec(Pong-
ramDeterministic-v4), EnvSpec(Pong-ramNoFrameskip-v0), EnvSpec(Pong-
ramNoFrameskip-v4), EnvSpec(Pooyan-v0), EnvSpec(Pooyan-v4),
EnvSpec(PooyanDeterministic-v0), EnvSpec(PooyanDeterministic-v4),
EnvSpec(PooyanNoFrameskip-v0), EnvSpec(PooyanNoFrameskip-v4), EnvSpec(Pooyan-
ram-v0), EnvSpec(Pooyan-ram-v4), EnvSpec(Pooyan-ramDeterministic-v0),
EnvSpec(Pooyan-ramDeterministic-v4), EnvSpec(Pooyan-ramNoFrameskip-v0),
EnvSpec(Pooyan-ramNoFrameskip-v4), EnvSpec(PrivateEye-v0),
EnvSpec(PrivateEye-v4), EnvSpec(PrivateEyeDeterministic-v0),
EnvSpec(PrivateEyeDeterministic-v4), EnvSpec(PrivateEyeNoFrameskip-v0),
EnvSpec(PrivateEyeNoFrameskip-v4), EnvSpec(PrivateEye-ram-v0),
EnvSpec(PrivateEye-ram-v4), EnvSpec(PrivateEye-ramDeterministic-v0),
EnvSpec(PrivateEye-ramDeterministic-v4), EnvSpec(PrivateEye-ramNoFrameskip-v0),
EnvSpec(PrivateEye-ramNoFrameskip-v4), EnvSpec(Qbert-v0), EnvSpec(Qbert-v4),
EnvSpec(QbertDeterministic-v0), EnvSpec(QbertDeterministic-v4),
EnvSpec(QbertNoFrameskip-v0), EnvSpec(QbertNoFrameskip-v4), EnvSpec(Qbert-
ram-v0), EnvSpec(Qbert-ram-v4), EnvSpec(Qbert-ramDeterministic-v0),
EnvSpec(Qbert-ramDeterministic-v4), EnvSpec(Qbert-ramNoFrameskip-v0),
EnvSpec(Qbert-ramNoFrameskip-v4), EnvSpec(Riverraid-v0), EnvSpec(Riverraid-v4),
EnvSpec(RiverraidDeterministic-v0), EnvSpec(RiverraidDeterministic-v4),
EnvSpec(RiverraidNoFrameskip-v0), EnvSpec(RiverraidNoFrameskip-v4),
EnvSpec(Riverraid-ram-v0), EnvSpec(Riverraid-ram-v4), EnvSpec(Riverraid-
ramDeterministic-v0), EnvSpec(Riverraid-ramDeterministic-v4), EnvSpec(Riverraid-
ramNoFrameskip-v0), EnvSpec(Riverraid-ramNoFrameskip-v4),
EnvSpec(RoadRunner-v0), EnvSpec(RoadRunner-v4),
EnvSpec(RoadRunnerDeterministic-v0), EnvSpec(RoadRunnerDeterministic-v4),
EnvSpec(RoadRunnerNoFrameskip-v0), EnvSpec(RoadRunnerNoFrameskip-v4),
EnvSpec(RoadRunner-ram-v0), EnvSpec(RoadRunner-ram-v4), EnvSpec(RoadRunner-
ramDeterministic-v0), EnvSpec(RoadRunner-ramDeterministic-v4),
EnvSpec(RoadRunner-ramNoFrameskip-v0), EnvSpec(RoadRunner-ramNoFrameskip-v4),
EnvSpec(Robotank-v0), EnvSpec(Robotank-v4), EnvSpec(RobotankDeterministic-v0),
EnvSpec(RobotankDeterministic-v4), EnvSpec(RobotankNoFrameskip-v0),

EnvSpec(RobotankNoFrameskip-v4), EnvSpec(Robotank-ram-v0), EnvSpec(Robotank-ram-v4), EnvSpec(Robotank-ramDeterministic-v0), EnvSpec(Robotank-ramDeterministic-v4), EnvSpec(Robotank-ramNoFrameskip-v0), EnvSpec(Robotank-ramNoFrameskip-v4), EnvSpec(Seaquest-v0), EnvSpec(Seaquest-v4), EnvSpec(SeaquestDeterministic-v0), EnvSpec(SeaquestDeterministic-v4), EnvSpec(SeaquestNoFrameskip-v0), EnvSpec(SeaquestNoFrameskip-v4), EnvSpec(Seaquest-ram-v0), EnvSpec(Seaquest-ram-v4), EnvSpec(Seaquest-ramDeterministic-v0), EnvSpec(Seaquest-ramDeterministic-v4), EnvSpec(Seaquest-ramNoFrameskip-v0), EnvSpec(Seaquest-ramNoFrameskip-v4), EnvSpec(Skiing-v0), EnvSpec(Skiing-v4), EnvSpec(SkiingDeterministic-v0), EnvSpec(SkiingDeterministic-v4), EnvSpec(SkiingNoFrameskip-v0), EnvSpec(SkiingNoFrameskip-v4), EnvSpec(Skiing-ram-v0), EnvSpec(Skiing-ram-v4), EnvSpec(Skiing-ramDeterministic-v0), EnvSpec(Skiing-ramDeterministic-v4), EnvSpec(Skiing-ramNoFrameskip-v0), EnvSpec(Skiing-ramNoFrameskip-v4), EnvSpec(Solaris-v0), EnvSpec(Solaris-v4), EnvSpec(SolarisDeterministic-v0), EnvSpec(SolarisDeterministic-v4), EnvSpec(SolarisNoFrameskip-v0), EnvSpec(SolarisNoFrameskip-v4), EnvSpec(Solaris-ram-v0), EnvSpec(Solaris-ram-v4), EnvSpec(Solaris-ramDeterministic-v0), EnvSpec(Solaris-ramDeterministic-v4), EnvSpec(Solaris-ramNoFrameskip-v0), EnvSpec(Solaris-ramNoFrameskip-v4), EnvSpec(SpaceInvaders-v0), EnvSpec(SpaceInvaders-v4), EnvSpec(SpaceInvadersDeterministic-v0), EnvSpec(SpaceInvadersDeterministic-v4), EnvSpec(SpaceInvadersNoFrameskip-v0), EnvSpec(SpaceInvadersNoFrameskip-v4), EnvSpec(SpaceInvaders-ram-v0), EnvSpec(SpaceInvaders-ram-v4), EnvSpec(SpaceInvaders-ramDeterministic-v0), EnvSpec(SpaceInvaders-ramDeterministic-v4), EnvSpec(SpaceInvaders-ramNoFrameskip-v0), EnvSpec(SpaceInvaders-ramNoFrameskip-v4), EnvSpec(StarGunner-v0), EnvSpec(StarGunner-v4), EnvSpec(StarGunnerDeterministic-v0), EnvSpec(StarGunnerDeterministic-v4), EnvSpec(StarGunnerNoFrameskip-v0), EnvSpec(StarGunnerNoFrameskip-v4), EnvSpec(StarGunner-ram-v0), EnvSpec(StarGunner-ram-v4), EnvSpec(StarGunner-ramDeterministic-v0), EnvSpec(StarGunner-ramDeterministic-v4), EnvSpec(StarGunner-ramNoFrameskip-v0), EnvSpec(StarGunner-ramNoFrameskip-v4), EnvSpec(Tennis-v0), EnvSpec(Tennis-v4), EnvSpec(TennisDeterministic-v0), EnvSpec(TennisDeterministic-v4), EnvSpec(TennisNoFrameskip-v0), EnvSpec(TennisNoFrameskip-v4), EnvSpec(Tennis-ram-v0), EnvSpec(Tennis-ram-v4), EnvSpec(Tennis-ramDeterministic-v0), EnvSpec(Tennis-ramDeterministic-v4), EnvSpec(Tennis-ramNoFrameskip-v0), EnvSpec(Tennis-ramNoFrameskip-v4), EnvSpec(TimePilot-v0), EnvSpec(TimePilot-v4), EnvSpec(TimePilotDeterministic-v0), EnvSpec(TimePilotDeterministic-v4), EnvSpec(TimePilotNoFrameskip-v0), EnvSpec(TimePilotNoFrameskip-v4), EnvSpec(TimePilot-ram-v0), EnvSpec(TimePilot-ram-v4), EnvSpec(TimePilot-ramDeterministic-v0), EnvSpec(TimePilot-ramDeterministic-v4), EnvSpec(TimePilot-ramNoFrameskip-v0), EnvSpec(TimePilot-ramNoFrameskip-v4), EnvSpec(Tutankham-v0), EnvSpec(Tutankham-v4), EnvSpec(TutankhamDeterministic-v0), EnvSpec(TutankhamDeterministic-v4), EnvSpec(TutankhamNoFrameskip-v0), EnvSpec(TutankhamNoFrameskip-v4), EnvSpec(Tutankham-ram-v0), EnvSpec(Tutankham-ram-v4), EnvSpec(Tutankham-ramDeterministic-v0), EnvSpec(Tutankham-ramDeterministic-v4), EnvSpec(Tutankham-ramNoFrameskip-v0), EnvSpec(Tutankham-

```
ramNoFrameskip-v4), EnvSpec(UpNDown-v0), EnvSpec(UpNDown-v4),
EnvSpec(UpNDownDeterministic-v0), EnvSpec(UpNDownDeterministic-v4),
EnvSpec(UpNDownNoFrameskip-v0), EnvSpec(UpNDownNoFrameskip-v4), EnvSpec(UpNDown-
ram-v0), EnvSpec(UpNDown-ram-v4), EnvSpec(UpNDown-ramDeterministic-v0),
EnvSpec(UpNDown-ramDeterministic-v4), EnvSpec(UpNDown-ramNoFrameskip-v0),
EnvSpec(UpNDown-ramNoFrameskip-v4), EnvSpec(Venture-v0), EnvSpec(Venture-v4),
EnvSpec(VentureDeterministic-v0), EnvSpec(VentureDeterministic-v4),
EnvSpec(VentureNoFrameskip-v0), EnvSpec(VentureNoFrameskip-v4), EnvSpec(Venture-
ram-v0), EnvSpec(Venture-ram-v4), EnvSpec(Venture-ramDeterministic-v0),
EnvSpec(Venture-ramDeterministic-v4), EnvSpec(Venture-ramNoFrameskip-v0),
EnvSpec(Venture-ramNoFrameskip-v4), EnvSpec(VideoPinball-v0),
EnvSpec(VideoPinball-v4), EnvSpec(VideoPinballDeterministic-v0),
EnvSpec(VideoPinballDeterministic-v4), EnvSpec(VideoPinballNoFrameskip-v0),
EnvSpec(VideoPinballNoFrameskip-v4), EnvSpec(VideoPinball-ram-v0),
EnvSpec(VideoPinball-ram-v4), EnvSpec(VideoPinball-ramDeterministic-v0),
EnvSpec(VideoPinball-ramDeterministic-v4), EnvSpec(VideoPinball-
ramNoFrameskip-v0), EnvSpec(VideoPinball-ramNoFrameskip-v4),
EnvSpec(WizardOfWor-v0), EnvSpec(WizardOfWor-v4),
EnvSpec(WizardOfWorDeterministic-v0), EnvSpec(WizardOfWorDeterministic-v4),
EnvSpec(WizardOfWorNoFrameskip-v0), EnvSpec(WizardOfWorNoFrameskip-v4),
EnvSpec(WizardOfWor-ram-v0), EnvSpec(WizardOfWor-ram-v4), EnvSpec(WizardOfWor-
ramDeterministic-v0), EnvSpec(WizardOfWor-ramDeterministic-v4),
EnvSpec(WizardOfWor-ramNoFrameskip-v0), EnvSpec(WizardOfWor-ramNoFrameskip-v4),
EnvSpec(YarsRevenge-v0), EnvSpec(YarsRevenge-v4),
EnvSpec(YarsRevengeDeterministic-v0), EnvSpec(YarsRevengeDeterministic-v4),
EnvSpec(YarsRevengeNoFrameskip-v0), EnvSpec(YarsRevengeNoFrameskip-v4),
EnvSpec(YarsRevenge-ram-v0), EnvSpec(YarsRevenge-ram-v4), EnvSpec(YarsRevenge-
ramDeterministic-v0), EnvSpec(YarsRevenge-ramDeterministic-v4),
EnvSpec(YarsRevenge-ramNoFrameskip-v0), EnvSpec(YarsRevenge-ramNoFrameskip-v4),
EnvSpec(Zaxxon-v0), EnvSpec(Zaxxon-v4), EnvSpec(ZaxxonDeterministic-v0),
EnvSpec(ZaxxonDeterministic-v4), EnvSpec(ZaxxonNoFrameskip-v0),
EnvSpec(ZaxxonNoFrameskip-v4), EnvSpec(Zaxxon-ram-v0), EnvSpec(Zaxxon-ram-v4),
EnvSpec(Zaxxon-ramDeterministic-v0), EnvSpec(Zaxxon-ramDeterministic-v4),
EnvSpec(Zaxxon-ramNoFrameskip-v0), EnvSpec(Zaxxon-ramNoFrameskip-v4),
EnvSpec(CubeCrash-v0), EnvSpec(CubeCrashSparse-v0),
EnvSpec(CubeCrashScreenBecomesBlack-v0), EnvSpec(MemorizeDigits-v0)])
```

6 A simple environment: the Cart-Pole

The Cart-Pole is a very simple environment composed of a cart that can move left or right, and pole placed vertically on top of it. The agent must move the cart left or right to keep the pole upright.

The make() function creates an environment, in this case a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it.

[24]:
```
env = gym.make('CartPole-v1')
```

Let's initialize the environment by calling is `reset()` method. This returns an observation:

[25]:
```
env.seed(42)
obs = env.reset()
```

Observations vary depending on the environment. In this case it is a 1D NumPy array composed of 4 floats: they represent the cart's horizontal position, its velocity, the angle of the pole (0 = vertical), and the angular velocity.

[27]:
```
obs
```

[27]:
```
array([-0.01258566, -0.00156614,  0.04207708, -0.00180545])
```

Cart is [1] units right of center, cart has velocity of [2], the pole has angle of [3] to the right, the angular velocity of the pole (how fast it falls down) is [4]

An environment can be visualized by calling its `render()` method, and you can pick the rendering mode (the rendering options depend on the environment).

Warning: some environments (including the Cart-Pole) require access to your display, which opens up a separate window, even if you specify `mode="rgb_array"`. In general you can safely ignore that window. However, if Jupyter is running on a headless server (ie. without a screen) it will raise an exception. One way to avoid this is to install a fake X server like Xvfb. On Debian or Ubuntu:

```
$ apt update
$ apt install -y xvfb
```

You can then start Jupyter using the `xvfb-run` command:

```
$ xvfb-run -s "-screen 0 1400x900x24" jupyter notebook
```

Alternatively, you can install the pyvirtualdisplay Python library which wraps Xvfb:

```
python3 -m pip install -U pyvirtualdisplay
```

And run the following code:

[7]:
```
# if you have prolems with display:
#!pip install -U pyvirtualdisplay
#import pyvirtualdisplay
```

[28]:
```
# run for separate window
env.render()
```

[28]:
```
True
```

In this example we will set `mode="rgb_array"` to get an image of the environment as a NumPy array:

[29]:
```
img = env.render(mode="rgb_array")
img.shape
```

[29]:
```
(800, 1200, 3)
```

[30]:
```
def plot_environment(env, figsize=(5,4)):
    plt.figure(figsize=figsize)
    img = env.render(mode="rgb_array")
    plt.imshow(img)
    plt.axis("off")
    return img
```

[31]:
```
plot_environment(env)
plt.show()
```

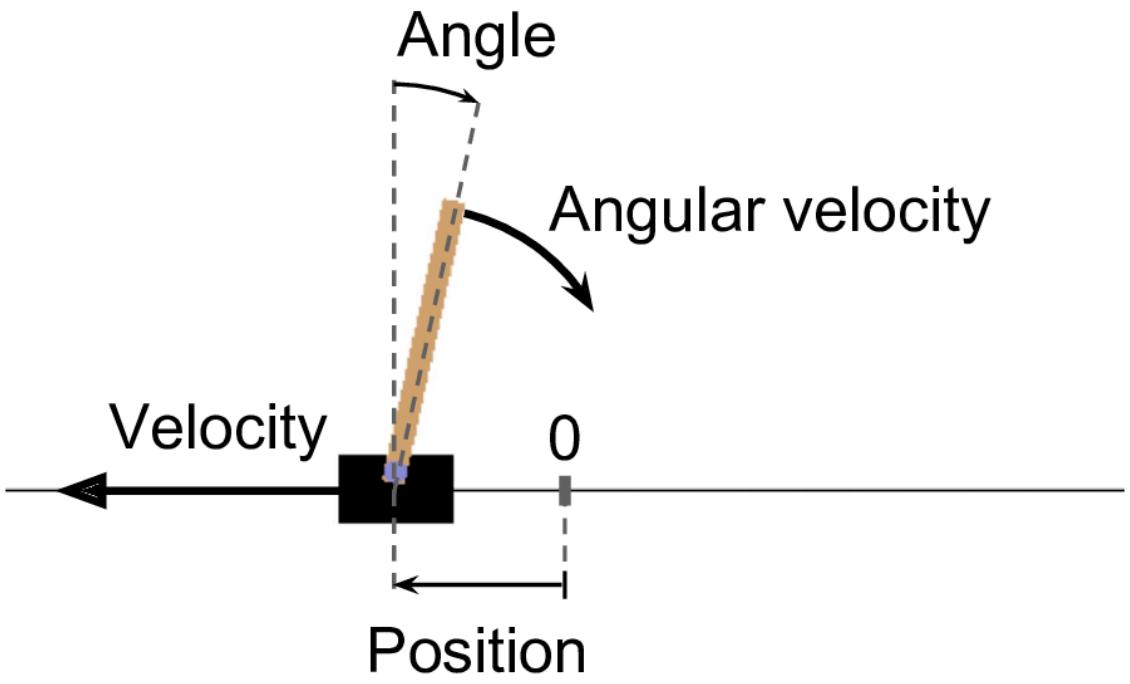
Let's see how to interact with an environment. Your agent will need to select an action from an "action space" (the set of possible actions). Let's see what this environment's action space looks like:

[20]:
```
env.action_space
```

[20]:
```
Discrete(2)
```

Yep, just two possible actions: accelerate towards the left or towards the right.

Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:



[32]:
```
action = 1  # accelerate right
obs, reward, done, info = env.step(action)
obs
```

[32]:
```
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
```

Notice that the cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted toward the left after the next step.

[33]:
```
plot_environment(env)
save_fig("cart_pole_plot")
```

```
Saving figure cart_pole_plot
```

Looks like it's doing what we're telling it to do!

The environment also tells the agent how much reward it got during the last step:

[21]:
```
reward
```

[21]:
```
1.0
```

When the game is over, the environment returns `done=True`:

[34]:
```
done
```

[34]:
```
False
```

Finally, `info` is an environment-specific dictionary that can provide some extra information that you may find useful for debugging or for training. For example, in some games it may indicate how many lives the agent has.

[121]:
```
info
```

[121]:
```
{}
```

The sequence of steps between the moment the environment is reset until it is done is called an "episode". At the end of an episode (i.e., when `step()` returns `done=True`), you should reset the environment before you continue to use it.

[122]:
```
if done:
    obs = env.reset()
```

Now how can we make the poll remain upright? We will need to define a *policy* for that. This is the strategy that the agent will use to select an action at each step. It can use all the past actions and observations to decide what to do.

7 A simple hard-coded policy

Let's hard code a simple strategy: if the pole is tilting to the left, then push the cart to the left, and *vice versa*. Let's see if that works:

[35]:
```
env.seed(42)
# function that the action
def basic_policy(obs):
    angle = obs[2]
    # if the tilt is leftward, move left, if the tile is rightward move right
    return 0 if angle < 0 else 1
# count steps
totals = []
# run for 500 experiments
for episode in range(500):
    episode_rewards = 0
    # record observations
    obs = env.reset()
    # Run each experiment for 200 steps
    for step in range(200):
        # calculate action given the chosen policy
        action = basic_policy(obs)
        # calculation new position/rewards after the action was taken
        obs, reward, done, info = env.step(action)
        # add rewards to the counter
        episode_rewards += reward
        if done:
            break
    # append statistics of an experiment to the array
    totals.append(episode_rewards)
```

[36]:
```
# calcualte averages
np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
```

[36]:
```
(41.718, 8.858356280936096, 24.0, 68.0)
```

Well, as expected, this strategy is a bit too basic: the best it did was to keep the poll up for only 68 steps, with average 41 steps. This environment is considered solved when the agent keeps the poll up for 200 steps.

Let's visualize one episode:

[37]:
```
env.seed(42)
# array of frames to plot.
```

```
frames = []
# reset enviroment
obs = env.reset()
for step in range(200):
    # convert image to the rgb array
    img = env.render(mode="rgb_array")
    # append frame
    frames.append(img)
    # take action accoriding to policy
    action = basic_policy(obs)
    # get the new enviroment information
    obs, reward, done, info = env.step(action)
    if done:
        break
```

Now show the animation:

[38]:
```
# animation and update
def update_scene(num, frames, patch):
    patch.set_data(frames[num])
    return patch,

def plot_animation(frames, repeat=False, interval=40):
    fig = plt.figure()
    patch = plt.imshow(frames[0])
    plt.axis('off')
    anim = animation.FuncAnimation(
        fig, update_scene, fargs=(frames, patch),
        frames=len(frames), repeat=repeat, interval=interval)
    plt.close()
    return anim
```

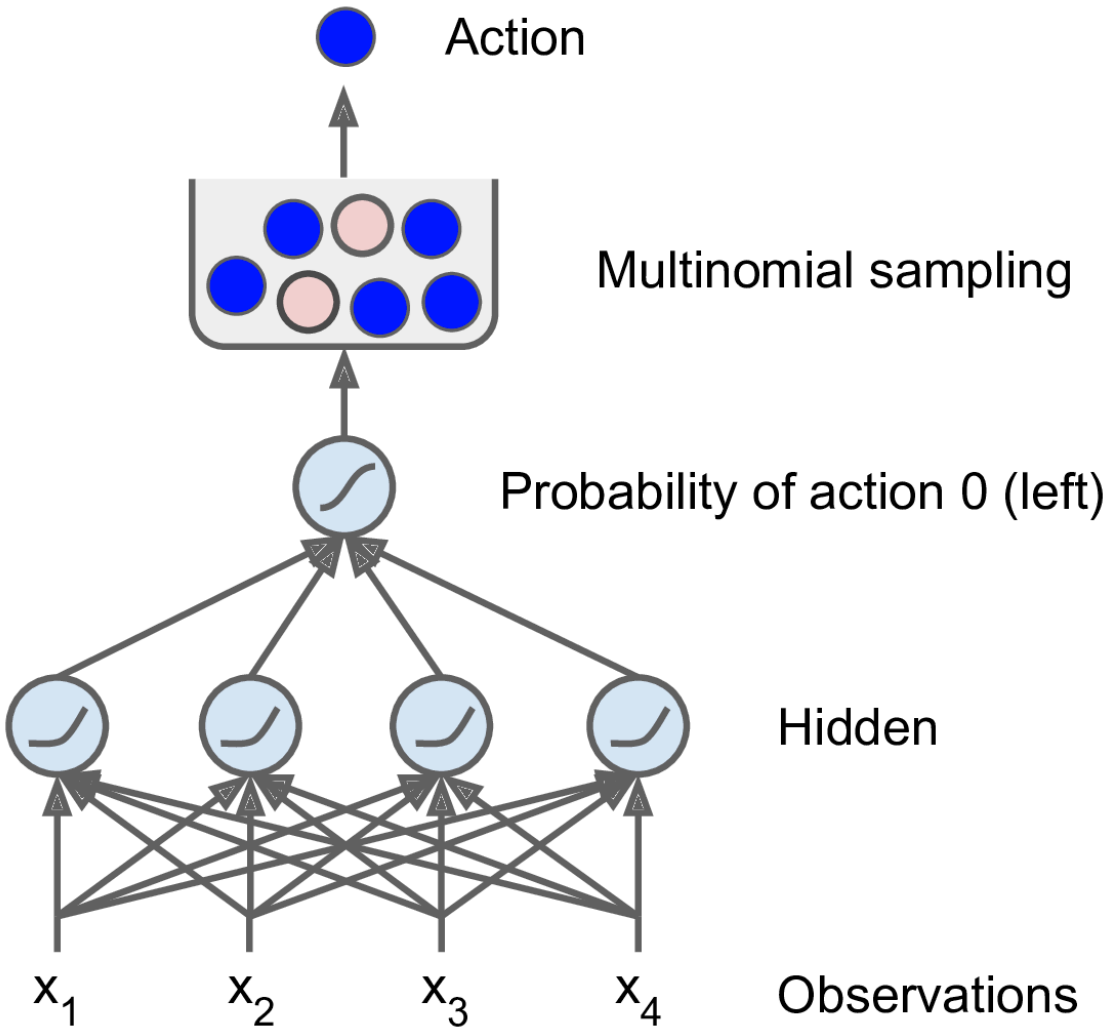
Clearly the system is unstable and after just a few wobbles, the pole ends up too tilted: game over. We will need to be smarter than that!

8 Neural Network Policies

Neural network that will take observations as inputs, and output the action to take for each observation.

To choose an action, the network will estimate a probability for each action, then we will select an action randomly according to the estimated probabilities.

In the case of the Cart-Pole environment actions are: left or right, so we only need one output neuron: it will output the probability `p` of the action 0 (left), and of course the probability of action 1 (right) will be `1 - p`. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be 1 – p. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.



[40]:
```
keras.backend.clear_session()
tf.random.set_seed(42)
np.random.seed(42)

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    # 5 neurons in the hiddel layer
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    # predict probability, so the activation is a sigmoid.
    keras.layers.Dense(1, activation="sigmoid"),
])
```

- We can ignore past actions
- For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity.

- Another example is if the observations are noisy: you may want to use the past few observations to estimate the most likely current state.

Why pick random action instead of the highest probability one? * Balance between exploring (accumulative knowledge) and exploting existing knowledge * Trying new dishes in a restuarant even if you like one dish already.

Let's write a small function that will run the model to play one episode, and return the frames so we can display an animation:

[41]:
```
# visual function, optional
def render_policy_net(model, n_max_steps=200, seed=42):
    frames = []
    env = gym.make("CartPole-v1")
    env.seed(seed)
    np.random.seed(seed)
    obs = env.reset()
    for step in range(n_max_steps):
        frames.append(env.render(mode="rgb_array"))
        left_proba = model.predict(obs.reshape(1, -1))
        action = int(np.random.rand() > left_proba)
        obs, reward, done, info = env.step(action)
        if done:
            break
    env.close()
    return frames
```

Now let's look at how well this randomly initialized policy network performs:

[42]:
```
frames = render_policy_net(model)
plot_animation(frames)
```

[42]:
```
<matplotlib.animation.FuncAnimation at 0x7fdf39b8a310>
```

Yeah… pretty bad. The neural network was not trained and will have to learn to do better.

First let's see if it is capable of learning the basic policy we used earlier: go left if the pole is tilting left, and go right if it is tilting right.

We can make the same net play in 50 different environments in parallel (this will give us a diverse training batch at each step), and train for 5000 iterations. We also reset environments when they are done. We train the model using a custom training loop so we can easily use the predictions at each training step to advance the environments.

[43]:
```
n_environments = 50
n_iterations = 5000
# get 50 random starting cart pole enviroments
envs = [gym.make("CartPole-v1") for _ in range(n_environments)]
# get distinct random seed index for each enviroment
for index, env in enumerate(envs):
```

```
    env.seed(index)
np.random.seed(42)
# get observations array for 50 enviroments
observations = [env.reset() for env in envs]
optimizer = keras.optimizers.RMSprop()
# binary accuracy: policy action vs predicted action
loss_fn = keras.losses.binary_crossentropy

for iteration in range(n_iterations):
    # if angle < 0, we want proba(left) = 1., or else proba(left) = 0.
    target_probas = np.array([([1.] if obs[2] < 0 else [0.])
                              for obs in observations])
    # GradientTape computes the gradient of a computation with respect
    #to some inputs (deriviative)
    with tf.GradientTape() as tape:
        # calculated predicted probabilities
        left_probas = model(np.array(observations))
        # Loss function
        loss = tf.reduce_mean(loss_fn(target_probas, left_probas))
    print("\rIteration: {}, Loss: {:.3f}".format(iteration, loss.numpy()),␣
↪end="")
    # take deriviative of the loss function with respect to trainable variables:
↪
    #[all weights and biases]
    grads = tape.gradient(loss, model.trainable_variables)
    # replace gradients and updated weights in the neural network
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    # get new actions. If 50 random numbers greater than model predictions,
    #then action True. The higher left_probas, the more likely False -> move␣
↪left, action =0
    # This is a random draw from the specified probabilities
    actions = (np.random.rand(n_environments, 1) > left_probas.numpy()).
↪astype(np.int32)
    for env_index, env in enumerate(envs):
        # run actions and save new enviroment
        obs, reward, done, info = env.step(actions[env_index][0])
        # save new observations. If the pole fell (Done), then reset
        observations[env_index] = obs if not done else env.reset()

for env in envs:
    env.close()
```

```
Iteration: 4999, Loss: 0.094
```

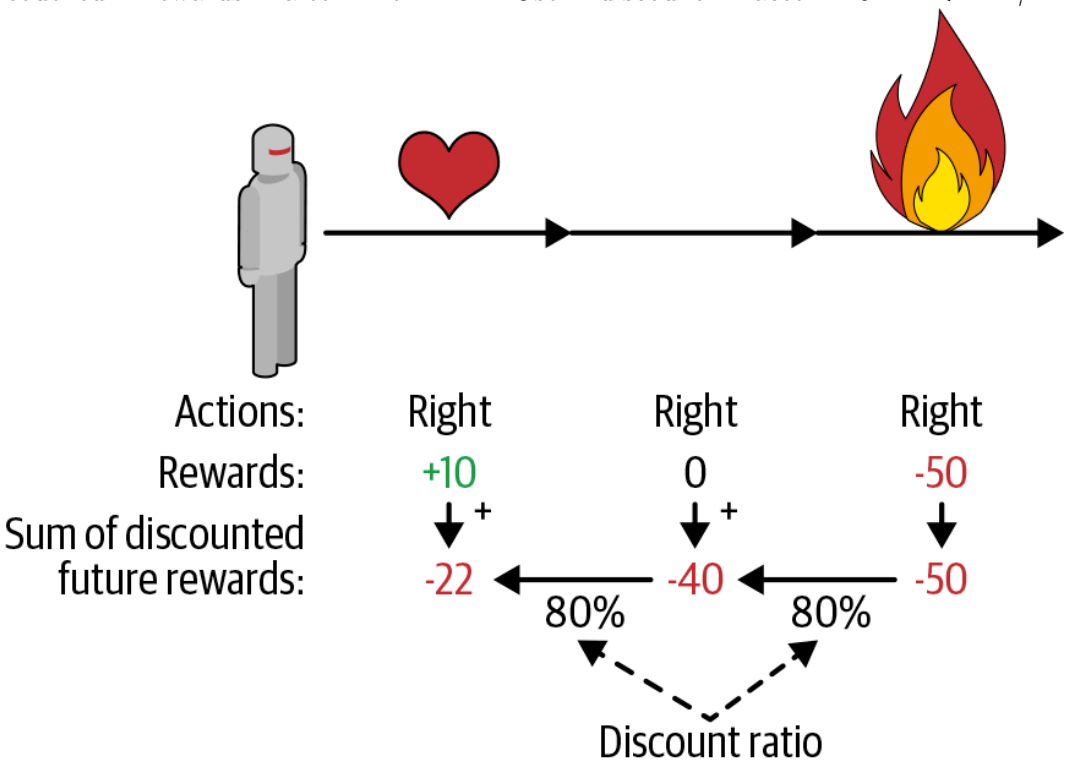
[44]:
```
frames = render_policy_net(model)
plot_animation(frames)
```

[44]: <matplotlib.animation.FuncAnimation at 0x7fdef987b760>

Looks like it learned the policy correctly. Now let's see if it can learn a better policy on its own. One that does not wobble as much.

#Evaluating Actions: The Credit Assignment Problem

- We don't know the optimal action at each step, so we cannot train the NN for it.
- In RL you observe final outcome – result of many past actions. We don't know which one were correct.
- One solution: Value of of an action equals to the sum of discoutned rewards after it. Use discount factor $0 < \gamma < 1$



- Assuming $\gamma = 0.8$ action of going right 3 times gets: $10 + \gamma * 0 + \gamma^2 * (-50) = -22$.
- Discount factor links past actions with future rewards. Large γ -> more connection with future. Typical $0.90 > \gamma > 0.9$. For $\gamma = 0.95$ the rewards 13 steps into the future count roughly for half of current rewards.
- A good action can be followed by bad actions reducing it's observed rewards, but if we run experiments enough times these effects of bad and good future choices will be cancelled out when evaluation a particular action.

9 Policy Gradients

The *Policy Gradients* algorithm assigns discounted future awards to the actions.

1. First run network several times to maximize prob of chosen action, but don't apply gradients yet.
2. Once we run several games, compute the gradient score of each action using discounted awards
3. Positive gradient is good, negative it's bad
4. Compute mean of gradient vectors resulting and perform Gradient descent

Let's start by creating a function to play a single step using the model. We will also pretend for now that whatever action it takes is the right one, so we can compute the loss and its gradients (we will just save these gradients for now, and modify them later depending on how good or bad the action turned out to be):

[47]:
```
# Here we are in step 1  and we will pretend that whatever action we take is␣
↪correct.
def play_one_step(env, obs, model, loss_fn):
    #tf function to calculate gradients
    with tf.GradientTape() as tape:
        # get predicted probabilities of left action
        left_proba = model(obs[np.newaxis])
        # if the action if higher the left_prage, move right (1-p)
        action = (tf.random.uniform([1, 1]) > left_proba)
#If the action is 0 (left), then the target probability of going left will be
#1. If the action is 1 (right), then the target probability will be 0.
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        # Loss is the difference between the action and the probability
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
    # calculate gradients
    grads = tape.gradient(loss, model.trainable_variables)
    # make and action and get new enviroment
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    # return results
    return obs, reward, done, grads
```

If `left_proba` is high, then `action` will most likely be `False` (since a random number uniformally sampled between 0 and 1 will probably not be greater than `left_proba`). And `False` means 0 when you cast it to a number, so `y_target` would be equal to 1 - 0 = 1. In other words, we set the target to 1, meaning we pretend that the probability of going left should have been 100% (so we took the right action).

Now let's create another function that will rely on the `play_one_step()` function to play multiple episodes, returning all the rewards and gradients, for each episode and each step:

[48]:
```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    # setup list of rewards and gradients
    all_rewards = []
    all_grads = []
    # loop over episodes (experiments)
    for episode in range(n_episodes):
        # setup a list of rewards and gradients for current step
        current_rewards = []
```

```
        current_grads = []
        # reset enviroment
        obs = env.reset()
        # loop over steps for each episode
        for step in range(n_max_steps):
            # run function play_one step defined above
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)
            # save rewards and gradients
            current_rewards.append(reward)
            current_grads.append(grads)
            # stop loop if the pole fell
            if done:
                break
        # append rewards for the episode
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads
```

The Policy Gradients algorithm uses the model to play the episode several times (e.g., 10 times), then it goes back and looks at all the rewards, discounts them and normalizes them. So let's create couple functions for that: the first will compute discounted rewards; the second will normalize the discounted rewards across many episodes.

[49]:
```
# set up function to discount rewards
def discount_rewards(rewards, discount_rate):
    discounted = np.array(rewards)
    # step backward disounting each step. The last step is not discounted.
    for step in range(len(rewards) - 2, -1, -1):
        # discount step. Step n-1 will be discounted once, step n-2 twice, etc.
        discounted[step] += discounted[step + 1] * discount_rate
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_rate):
    # discount all rewards
    all_discounted_rewards = [discount_rewards(rewards, discount_rate)
                              for rewards in all_rewards]
    # concatenate rewards from all episodes and steps
    flat_rewards = np.concatenate(all_discounted_rewards)
    # make the rewards standard normal dist
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Say there were 3 actions, and after each action there was a reward: first 10, then 0, then -50. If we use a discount factor of 80%, then the 3rd action will get -50 (full credit for the last reward), but the 2nd action will only get -40 (80% credit for the last reward), and the 1st action will get 80% of -40 (-32) plus full credit for the first reward (+10), which leads to a discounted reward of -22:

[50]:
```
discount_rewards([10, 0, -50], discount_rate=0.8)
```

[50]:
```
array([-22, -40, -50])
```

To normalize all discounted rewards across all episodes, we compute the mean and standard deviation of all the discounted rewards, and we subtract the mean from each discounted reward, and divide by the standard deviation:

[51]:
```
discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_rate=0.8)
```

[51]:
```
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```

The first episode was much worse than the second, so its normalized advantages are all negative; all actions from the first episode would be considered bad.

Conversely all actions from the second episode would be considered good.

[52]:
```
# 150 training iterations, 10 episodes per iteration. Each episode can last 200␣
↪steps.
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_rate = 0.95
```

[53]:
```
optimizer = keras.optimizers.Adam(lr=0.01)
# binary classifier
loss_fn = keras.losses.binary_crossentropy
```

[54]:
```
# the model is almost identical to the one we used before
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[4]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

[55]:
```
env = gym.make("CartPole-v1")
env.seed(42);
# loop through the iterations
for iteration in range(n_iterations):
    # play 10 episodes per iterations
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    # total rewards per iteration
```

```
    total_rewards = sum(map(sum, all_rewards))                    # Not shown␣
↪in the book
    # pring aveage number of rewards per update
    print("\rIteration: {}, mean rewards: {:.1f}".format(          # Not shown
        iteration, total_rewards / n_episodes_per_update), end="") # Not shown
    # discount and normilized awards from the iteration
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_rate)
    all_mean_grads = []
    # loop over model parameters (weights and biases)
    for var_index in range(len(model.trainable_variables)):
        # get average gradient weighted by the average reward, so that␣
↪succeefull grad will have
        # larger weight
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             # final rewards in all steps in all episodes
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        #append average gradients for the iteration.
        all_mean_grads.append(mean_grads)
        # apply updated gradiatent to the model.
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))

env.close()
```

```
Iteration: 149, mean rewards: 175.6
```

This results is considerably better than the average rewards of 41, which we got with the hard-coded policy.

[56]:
```
frames = render_policy_net(model)
plot_animation(frames)
```

[56]:
```
<matplotlib.animation.FuncAnimation at 0x7fdf18e1b7f0>
```

The simple policy gradients algorithm we used for CartPole task would not scale well to larger and more complex tasks. We have to play this game long-time before we will see the advantage of each action. There are some more efficient solutions such as Actor-Critic algorithms.

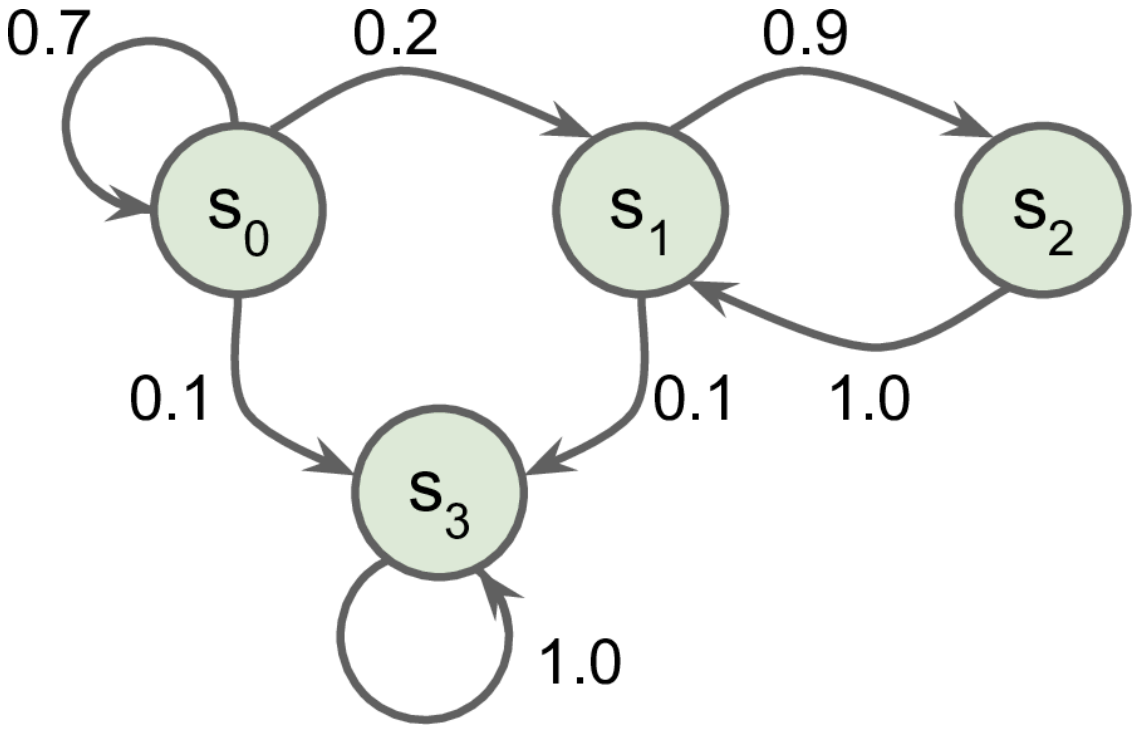
Next we will apporach policy from another Angle.

Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct: the agent learns to estimate the expected return for each state, or for each action in each state, then it uses this knowledge to decide how to act.

10 Markov Chains

- Andrey Markov studied stochastic processes with no memory, called Markov chains.

Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s is fixed, and it depends only on the pair (s, s), not on past states (no memory).



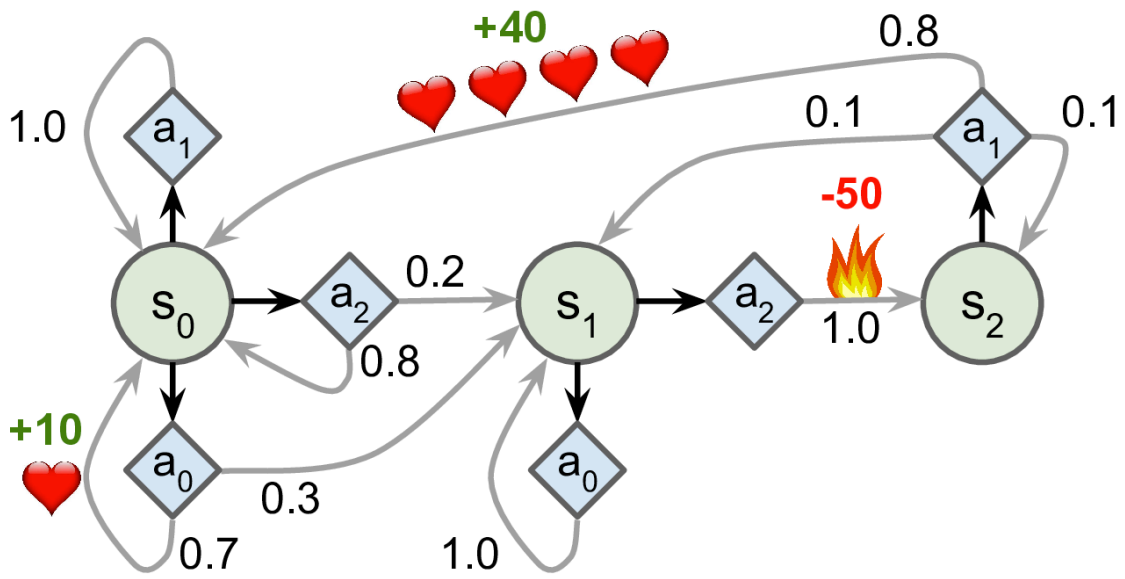
* s_3 is the terminal state

[57]:
```
np.random.seed(42)
# based on the picture above
transition_probabilities = [ # shape=[s, s']
        [0.7, 0.2, 0.0, 0.1],  # from s0 to s0, s1, s2, s3
        [0.0, 0.0, 0.9, 0.1],  # from s1 to ...
        [0.0, 1.0, 0.0, 0.0],  # from s2 to ...
        [0.0, 0.0, 0.0, 1.0]]  # from s3 to ...

n_max_steps = 50

def print_sequence():
    current_state = 0
    print("States:", end=" ")
    for step in range(n_max_steps):
        print(current_state, end=" ")
        if current_state == 3:
            break
        current_state = np.random.choice(range(4),␣
 ↪p=transition_probabilities[current_state])
    else:
        print("...", end="")
    print()
```

```
for _ in range(10):
    print_sequence()
```

```
States: 0 0 3
States: 0 1 2 1 2 1 2 1 2 1 3
States: 0 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 3
States: 0 3
States: 0 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 3
States: 0 1 3
States: 0 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
2 1 2 1 2 1 2 1 2 1 2 1 …
States: 0 0 3
States: 0 0 0 1 2 1 2 1 3
States: 0 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 3
```

Richard Bellman in 1950s described Markov Processes – Markov Chains where at each state an agent can choose and action, which would lead to a new state and new reward with some probability.



The agent starts at state s_0, the agent can choose between actions a0, a1, or a2.

In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

[ ]:

11 Markov Decision Process

Let's define some transition probabilities, rewards and possible actions. For example, in state s0, if action a0 is chosen then with proba 0.7 we will go to state s0 with reward +10, with probability 0.3

we will go to state s1 with no reward, and with never go to state s2 (so the transition probabilities are `[0.7, 0.3, 0.0]`, and the rewards are `[+10, 0, 0]`):

[58]:
```
transition_probabilities = [ # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
        [None, [0.8, 0.1, 0.1], None]]
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
        [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

12 Q-Value Iteration

Bellman estimate optimal state value by using sum of future discounted rewards assuming a player will act optimally:

$$V*(s) = \max_a \sum_s T(s,a,s')[R(s,a,s') + \gamma * V(s')] \text{ for all s} \tag{1}$$

In this equation: • $T(s,a,s\,)$ is the transition probability from state s to state s , given that the agent chose action a. For example, T(s2, a1, s0) = 0.8. • $R(s,a,s\,)$ is the reward that the agent gets when it goes from state s to state s , given that the agent chose action a. For example, $(s2, a1, s0) = +40$. • γ is the discount factor.

This equation leads directly to an algorithm that can estimates the optimal state value of every possible state: - first initialize all the state value estimates to zero - iteratively update them using the Value Iteration algorithm.

$$V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma * V_k(s')] \text{ for all s} \tag{2}$$

where $V_k*(s')$ is the k^{th} iteration of the algorithm.

Given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Knowing the optimal state values is useful, but it does not give us the optimal policy for the agent.

Another Bellman algorithm finds optimal state-action values, generally called $Q-Values$. (Quality Values).

The optimal $Q-Value$ of the state-action pair (s,a) is $Q^*(s,a)$ – the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, assuming it acts optimally after that action.

Once again, we initialize all the $Q-Value$ estimates to zero, then you update them using the Q-Value Iteration algorithm:

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma * Q_k*(s',a')] \text{ for all } s', a \tag{3}$$

Once you have the optimal Q-Values, then the optimal polic noted $\pi^*(s)$ is just the choice of the highest Q-Value in each state: $\pi^*(s) = argmax\, Q^*(s, a)$

[60]:
```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
# for possible actions set Q-value at zero
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

[62]:
```
gamma = 0.90  # the discount factor

history1 = [] # Not shown in the book (for the figure below)
# iterate over 50 steps
for iteration in range(50):
    # save old Q_value
    Q_prev = Q_values.copy()
    #append the history of Q-values
    history1.append(Q_prev) # Not shown
    # loop over 3 states
    for s in range(3):
        # loop over possible actions
        for a in possible_actions[s]:
            # value of choosing action a in state s
            Q_values[s, a] = np.sum([
                    # expected rewards
                    transition_probabilities[s][a][sp]
                #based on the choosing actions with highest Q-value. (based on␣
 ↪last step Q-values)
                    * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
# history of Q-values
history1 = np.array(history1) # Not shown
```

[63]:
```
Q_values
```

[63]:
```
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

when the agent is in state s_0 and it chooses action a_1, the expected sum of discounted future rewards is approximately 17.0.

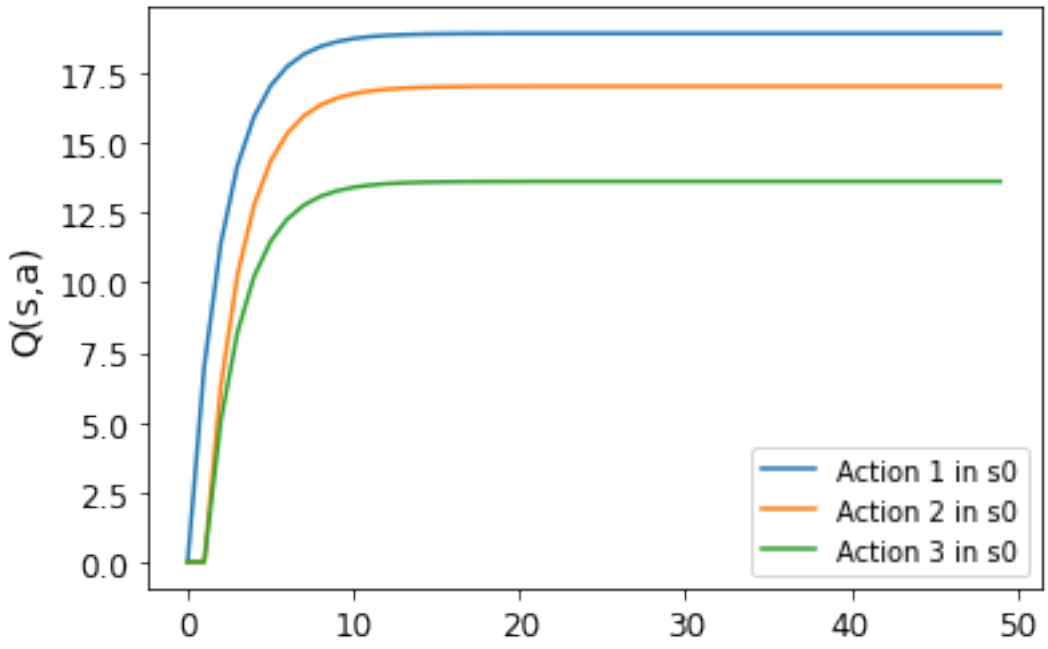
The optimal policy for this MDP, when using a discount factor of 0.90, is to choose action a0 when in state s0, and choose action a0 when in state s1, and finally choose action a1 (the only possible action) when in state s2.

[64]:
```
# plot evolution in Q-valus
np.argmax(Q_values, axis=1)
array1 = []
```

```
array2 = []
array3 = []
for i in range(0,50):
    array1.append(history1[i][0][0])
    array2.append(history1[i][0][1])
    array3.append(history1[i][0][2])
```

[65]:
```
import matplotlib.pyplot as plt
plt.plot(array1, label='Action 1 in s0')
plt.plot(array2, label='Action 2 in s0')
plt.plot(array3, label='Action 3 in s0')
plt.ylabel('Q(s,a)')
plt.legend()
# We figured out the right action quickly
```

[65]:
```
<matplotlib.legend.Legend at 0x7fdf3ffdd370>
```



Let's try again with a discount factor of 0.95:

[66]:
```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

[67]:
```
gamma = 0.95  # the discount factor

for iteration in range(50):
```

```
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

[68]:
```
Q_values
```

[68]:
```
array([[21.73304188, 20.63807938, 16.70138772],
       [ 0.95462106,        -inf,  1.01361207],
       [       -inf, 53.70728682,        -inf]])
```

[69]:
```
np.argmax(Q_values, axis=1)
```

[69]:
```
array([0, 2, 1])
```

Now the policy has changed! In state s1, we now prefer to go through the fire (choose action a2). This is because the discount factor is larger so the agent values the future more, and it is therefore ready to pay an immediate penalty in order to get more future rewards.

13 Temporal Difference in Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes.

The problem is that the agent initially has no idea what the transition probabilities T and it does not know what the rewards are going to be either R.

It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The Temporal Difference Learning (TD Learning) algorithm is the Q-Values tweaked to take into account the fact that the agent has little knowledge of the MDP.

We assume that the agent initially knows only the possible states and actions, and nothing more.

The agent uses an exploration policy—for example, a purely random policy—to explore the MDP, and as it progresses, the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed.

TD Learning Algorithm

$$V_{k+1} \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma * V_k(s')) \tag{4}$$

or equivalently

$$V_{k+1} \leftarrow V_k(s) + \alpha * \lambda_k(s, r, s') \tag{5}$$

with $\lambda_k(s, r, s') = r + \gamma * V_k(s') - V_k(s)$ In the equation: * α is the learning rate (e.g. 0.01) when we learn rewards and transitions probabilities * $r + \gamma * V_k(s')$ is the TD target, what we want to learn. * $\lambda_k(s, r, s')$ is the TD error, new information we learnt in the step $k + 1$.

A more consise notation would be how we update out expectation of a from the new information obtained b through the learning rate α:

$a \underset{\alpha}{\leftarrow} b = a_{k+1} \leftarrow (1-\alpha) * a_k + \alpha * b$ then we can rewrite:

$V(s) \underset{\alpha}{\leftarrow} r + \gamma * V(s')$

For each state s, this algorithm keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later

14 Q-Learning

Q-Learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-Values. Once it has accurate Q-Value estimates (or close enough), then the optimal policy consists in choosing the action that has the highest Q-Value (i.e., the greedy policy).

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \underset{a'}{\max} Q(s',a') \tag{6}$$

For each state-action pair (s,a) we track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get.

To estimate this sum, we take the maximum of the Q-Value estimates for the next state s , since we assume that the target policy would act optimally from then on.

We will need to simulate an agent moving around in the environment, so let's define a function to perform some action and get the new state and a reward:

[70]:
```
# expoloration function.
def step(state, action):
    # get transitional probability to the next state after the action is taken
    probas = transition_probabilities[state][action]
    #Find randomly next state according to defined probabilities
    next_state = np.random.choice([0, 1, 2], p=probas)
    # get the reward of the next state
    reward = rewards[state][action][next_state]
    # return next state and reward
    return next_state, reward
```

We also need an exploration policy, which can be any policy, as long as it visits every possible state many times. We will just use a random policy, since the state space is very small:

[71]:
```
# randomly choose a possible action available in this state
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Now let's initialize the Q-Values like earlier, and run the Q-Learning algorithm:

[72]:
```
np.random.seed(42)
# start with negative inf values for Q(s,a)
```

```
Q_values = np.full((3, 3), -np.inf)
# for possible actions set Q(s,a)= 0
for state, actions in enumerate(possible_actions):
    Q_values[state][actions] = 0

alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state
history2 = [] # Save history of Q-values learning
# interation over steps to learn Q-values
for iteration in range(10000):
    history2.append(Q_values.copy()) # append each step to history
    action = exploration_policy(state) # take random action to explore space
    next_state, reward = step(state, action) # get next state action and the␣
 ↪reward
    next_value = np.max(Q_values[next_state]) # greedy policy at the next step,␣
 ↪choose highest Q-value
    alpha = alpha0 / (1 + iteration * decay) # decay learning rate
    Q_values[state, action] *= 1 - alpha # The share of the updated Q-value
    Q_values[state, action] += alpha * (reward + gamma * next_value) # add new␣
 ↪learnt information to the Q-value
    state = next_state

history2 = np.array(history2) # add to history of Q-values
```

[73]:
```
Q_values
```

[73]:
```
array([[18.77621289, 17.2238872 , 13.74543343],
       [ 0.        ,        -inf, -8.00485647],
       [       -inf, 49.40208921,        -inf]])
```

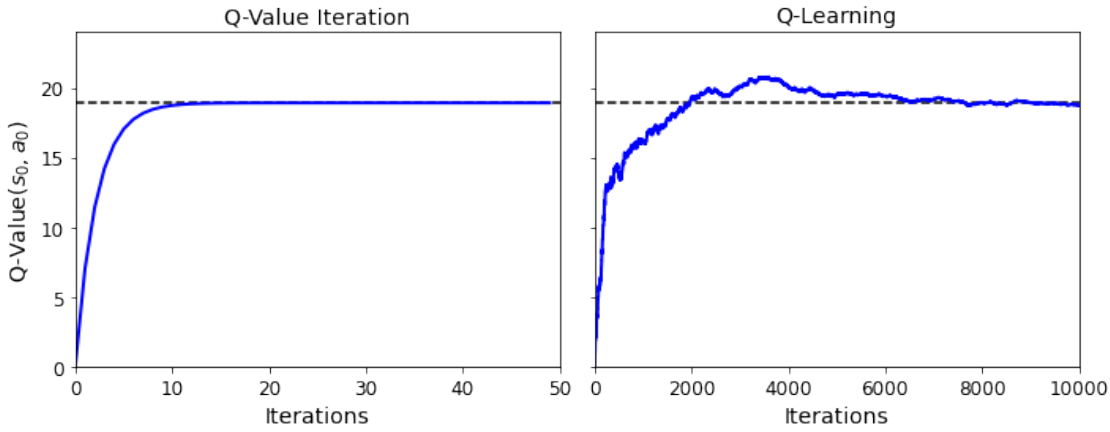
[74]:
```
np.argmax(Q_values, axis=1) # optimal action for each state
```

[74]:
```
array([0, 0, 1])
```

[75]:
```
# Plot histroy of learning Q(s0,a0)
true_Q_value = history1[-1, 0, 0]

fig, axes = plt.subplots(1, 2, figsize=(10, 4), sharey=True)
axes[0].set_ylabel("Q-Value$(s_0, a_0)$", fontsize=14)
axes[0].set_title("Q-Value Iteration", fontsize=14)
axes[1].set_title("Q-Learning", fontsize=14)
for ax, width, history in zip(axes, (50, 10000), (history1, history2)):
    ax.plot([0, width], [true_Q_value, true_Q_value], "k--")
    ax.plot(np.arange(width), history[:, 0, 0], "b-", linewidth=2)
    ax.set_xlabel("Iterations", fontsize=14)
```

```
    ax.axis([0, width, 0, 24])

save_fig("q_value_plot")
```

```
Saving figure q_value_plot
```



Q-learning converges to the optimal Q-Values, but it will take too long and too much hyperparameter tuning.

Q-Value Iteration algorithm (left) converges very quickly, in fewer than 20 iterations, Q-Learning takes 8,000 iterations. Not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

The problem with Q-learning is that is tries to learn about the enviroment and optimize the policy in the same time, which is very hard. Can we do better?

15 Exploration Policies

Q-Learning can work only if the exploration policy fully explores the MDP. A fully random policy will eventually explore all options, but a better is ϵ-greedy policy: - randomly explore with the probability ϵ - with probability 1– explore/choose actions with the highest Q-values, so we explore more valuable paths. Importatn for a large choice sets.

Usually start with large ϵ close to 1, and gradually reduce it to move from exploration to the optimal path search.

Alternative to this strategy is to explore paths that has not been tried before.

16 Approximate Q-Learning and Deep Q-Learning

Q-Learning does not scale for the enviroments with the large action space, like Pac-Man: 150 pellets to eat and 8 actions to choose. The number of states is $2^{150} \approx 1045$. The number of states with different combinations for the possible positions of ghosts makes the total number larger than the number of atoms on Earth.

The solution the function $Q_\theta(s,a)$ that approximates Q-Value of the pair (s,a) under some small number of parameters θ. DNN can estimate Q-Values in Deep Q-Network (DQN).

The Q-value should approximate the reward r under (s,a) plus the discounted value of playing optimally from then on. DQN estimates this sum a Q-Values at (s',a') for all possible actions a'. Then we pick the best a' and use it to value to calculated the discounted $Q(s',a')$:

$$Q_{target}(s,a) = r + \gamma * \max_{a'} Q_\theta(s',a') \tag{7}$$

With this target Q-Value, we can run a train out network using Gradient Descent algorithm to minimize the squared error between the estimated Q-Value $Q(s,a)$ and the target Q-Value.

Let's build the DQN. Given a state, it will estimate, for each possible action, the sum of discounted future rewards it can expect after it plays that action (but before it sees its outcome):

[77]:
```
keras.backend.clear_session()
tf.random.set_seed(42)
np.random.seed(42)

env = gym.make("CartPole-v1")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # We extimate Q-Values for going left or right at each step.
# simple NN
model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

To select an action using this DQN, we just pick the action with the largest predicted Q-value. However, to ensure that the agent explores the environment, we choose a random action with probability `epsilon`.

[86]:
```
# Epsilon-greedy policy.
def epsilon_greedy_policy(state, epsilon=0):
    # randomly choose exploration
    if np.random.rand() < epsilon:
        return np.random.randint(2) # randomly choose 0 or 1
    else:
        # randomly choose max Q-value
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

We will also need a replay memory. It will contain the agent's experiences, in the form of tuples: `(obs, action, reward, next_obs, done)`. We can use the `deque` class for that:

[87]:
```
# A fast list of elements connected to immediate neighbors
from collections import deque
```

```
# Store all experience in this memory list. We will sample from it using␣
↪training.
replay_memory = deque(maxlen=2000)
```

And let's create a function to sample experiences from the replay memory. It will return 5 NumPy arrays: `[obs, actions, rewards, next_obs, dones]`.

[88]:
```
# get a sample of experiences (situations)
def sample_experiences(batch_size):
    # random indexes from the experience buffer
    indices = np.random.randint(len(replay_memory), size=batch_size)
    #draw batch_size number of random experiences
    batch = [replay_memory[index] for index in indices]
    # get information from these experiences.
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    # return information
    return states, actions, rewards, next_states, dones
```

Now we can create a function that will use the DQN to play one step, and record its experience in the replay memory:

[89]:
```
def play_one_step(env, state, epsilon):
    # chosen action
    action = epsilon_greedy_policy(state, epsilon)
    # find the next state and reward from the chosen action
    next_state, reward, done, info = env.step(action)
    # add the action, results and experience in one array
    replay_memory.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

Lastly, let's create a function that will sample some experiences from the replay memory and perform a training step:

Note: the first 3 releases of the 2nd edition were missing the `reshape()` operation which converts `target_Q_values` to a column vector (this is required by the `loss_fn()`).

[90]:
```
batch_size = 32
discount_rate = 0.95 # gamma
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error # MSE because Q-Values are numbers

def training_step(batch_size):
    experiences = sample_experiences(batch_size)  # get experiences
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states) #predict Q-values of available␣
↪actions
```

```
    max_next_Q_values = np.max(next_Q_values, axis=1) # choose the best action
    # Target Q-value: reward + discounted Q-value of the best action
    target_Q_values = (rewards +
                       # add discounted Q-value is this is not the last step
                       (1 - dones) * discount_rate * max_next_Q_values)
    # Q-values of the two actions available
    target_Q_values = target_Q_values.reshape(-1, 1)
    # add zeros to account for the actions not takes: actions 1,1,0 becomes␣
↪maskes [[0,1], [0,1], [0,1]],
    # we can multiply this vector of maked actions by gradients or Q-values,␣
↪getting zeros for the Q-values
    # we don't want.
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        # get predicted Q-Values from the model
        all_Q_values = model(states)
        # get the Q-Values fro the chosen actions
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        # calculate loss between predicted Q-values and experienced␣
↪state-action pairs.
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

And now, let's train the model!

[91]:
```
env.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

rewards = []
best_score = 0
```

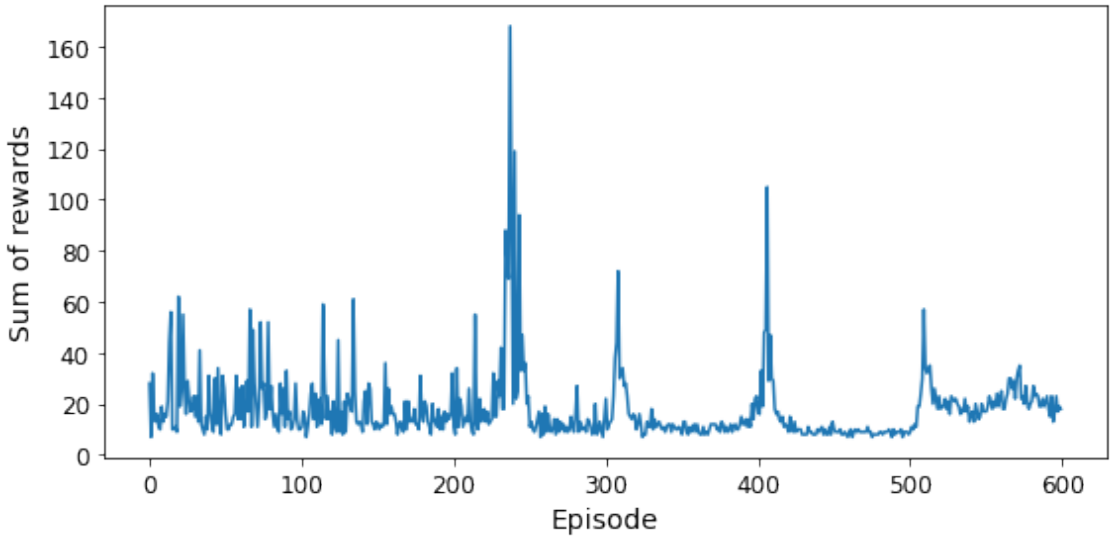
[92]:
```
# Run 600 episodes (games), with 200 steps in each episodes.
for episode in range(600):
    obs = env.reset()
    for step in range(200): # loop over the steps
        epsilon = max(1 - episode / 500, 0.01) # reduce exploration for later␣
↪episodes
        obs, reward, done, info = play_one_step(env, obs, epsilon) # get␣
↪enviroment variables
        if done:
            break
    rewards.append(step) # append rewards (number of turns survived) for the␣
↪episode
    if step > best_score: # If the rewards are the best in history, update␣
↪weights
```

```
        best_weights = model.get_weights() #  save best weights
        best_score = step # best score
    print("\rEpisode: {}, Steps: {}, eps: {:.3f}".format(episode, step + 1,␣
↪epsilon), end="") # Not shown
    if episode > 50:# there is not training for the first 50 episodes, we want␣
↪the buffer to fill-up, we are still
        # learning about the enviroment.
        training_step(batch_size)

model.set_weights(best_weights)
```

```
Episode: 599, Steps: 19, eps: 0.0108
```

[93]:
```
plt.figure(figsize=(8, 4))
plt.plot(rewards)
plt.xlabel("Episode", fontsize=14)
plt.ylabel("Sum of rewards", fontsize=14)
save_fig("dqn_rewards_plot")
plt.show()
```

```
Saving figure dqn_rewards_plot
```



There was no progress for 300 episodes, the we suddely were able to finish the game with 200 (max score) and quickly lost our progress.

We experienced catastrophic forgetting – a common problem for RL. SGD has a short memory and the expiences may change. We can try to fix it by increasing the size of the replay buffer or reducing the learning rate.

Generally, RL is hard and fragile: As the researcher Andrej Karpathy put it: “Supervised learning

wants to work. [...] RL must be forced to work."

This is the main reason RL is not widely adopted yet relative to DL.

We did not plot loss because it is not really relevant to the task. A loss can go down from overfitting, we can predict Q-Values very well, but the pole will still fall.

[94]:
```
env.seed(42)
state = env.reset()

frames = []

for step in range(200):
    action = epsilon_greedy_policy(state)
    state, reward, done, info = env.step(action)
    if done:
        break
    img = env.render(mode="rgb_array")
    frames.append(img)

plot_animation(frames)
```

[94]:
```
<matplotlib.animation.FuncAnimation at 0x7fdf0961c7f0>
```

Not bad at all!

16.1 Online Model

In the basic Deep Q-Learning algorithm, the model is used both to make predictions and to set its own targets. This may create unstable feedback loop.

DeepMind researchers proposed using two DQNs instead of one: the first is the online model, which learns at each step and is used to move the agent around, and the other is the target model used only to define the targets.

The target model is just a clone of the online model. Then, in the training_step() function, we use the target model instead of the online model when computing the Q-Values of the next states. Finally, in the training loop, we must copy the weights of the online model to the target model, at regular intervals (e.g., every 50 episodes).

Since the target model is updated much less often than the online model, the Q-Value targets are more stable.

16.2 Double DQN

DeepMind researchers tweaked their DQN algorithm, increasing its performance and stabilizing training.

They called this variant Double DQN. The update was based on the observation that the target network is prone to overestimating Q-Values. If all actions are equally good, but one just randomly looks better, DQN would greatly overcluse this action, which looks good by pure chance.

We fix this by using the online model instead of the target model when selecting the best actions for the next states, and using the target model only to estimate the Q-Values for these best actions.

[95]:
```
keras.backend.clear_session()
tf.random.set_seed(42)
np.random.seed(42)
# build out regular model for online training
model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=[4]),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
# copy the model and call it target
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

[97]:
```
batch_size = 32
discount_rate = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.Huber()
# The training step is the same as before
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)        # online model
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_rate * next_best_Q_values)
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)        # online model
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

[98]:
```
# save get buffer
replay_memory = deque(maxlen=2000)
```

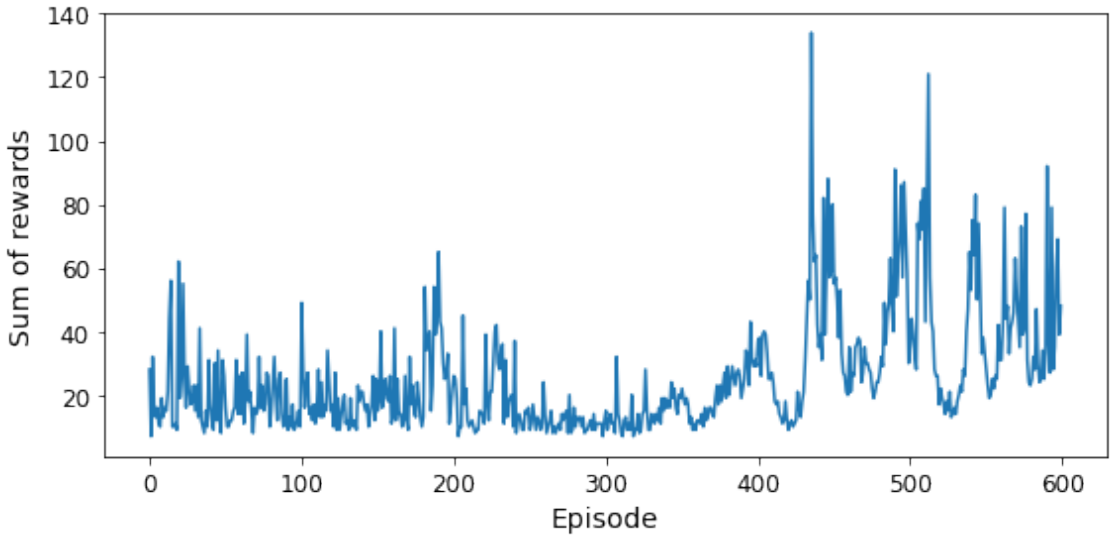
[99]:
```
env.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

rewards = []
```

```
best_score = 0

for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    rewards.append(step)
    if step > best_score:
        best_weights = model.get_weights()
        best_score = step
    print("\rEpisode: {}, Steps: {}, eps: {:.3f}".format(episode, step + 1,␣
↪epsilon), end="")
    if episode > 50:
        training_step(batch_size)
        # every 50 copy weights to the target model
    if episode % 50 == 0:
        target.set_weights(model.get_weights())
    # Alternatively, you can do soft updates at each step:
    #if episode > 50:
        #target_weights = target.get_weights()
        #online_weights = model.get_weights()
        #for index in range(len(target_weights)):
        #    target_weights[index] = 0.99 * target_weights[index] + 0.01 *␣
↪online_weights[index]
        #target.set_weights(target_weights)

model.set_weights(best_weights)
```

```
Episode: 599, Steps: 49, eps: 0.0100
```

[100]:
```
plt.figure(figsize=(8, 4))
plt.plot(rewards)
plt.xlabel("Episode", fontsize=14)
plt.ylabel("Sum of rewards", fontsize=14)
save_fig("double_dqn_rewards_plot")
plt.show()
```

```
Saving figure double_dqn_rewards_plot
```



[101]:
```
env.seed(42)
state = env.reset()

frames = []

for step in range(200):
    action = epsilon_greedy_policy(state)
    state, reward, done, info = env.step(action)
    if done:
        break
    img = env.render(mode="rgb_array")
    frames.append(img)

plot_animation(frames)
```

[101]:
```
<matplotlib.animation.FuncAnimation at 0x7fdf1b82bd30>
```

17 Dueling Double DQN

Deepmind introduced another important model: dueling DQN algorithm (DDQN) was introduced. First note that the Q-Value of a state-action pair: $Q(s,a) = V(s) + A(s,a)$, where $V(s)$ is the value of state s and $A(s,a)$ is the advantage of taking the action a in state s, compared to all other possible actions in that state.

Moreover, the value of a state is equal to the Q-Value of the best action $a*$ for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s,a*)$, which implies that A(s, a*) = 0.

In a Dueling DQN, the model estimates both the value of the state and the advantage of each

possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all pre- dicted advantages.

[102]:
```
keras.backend.clear_session()
tf.random.set_seed(42)
np.random.seed(42)

K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2) # value of the state
raw_advantages = keras.layers.Dense(n_outputs)(hidden2) # raw values of the␣
↪advantages for actions (2).
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)#␣
↪Normlized best action at 0
Q_values = state_values + advantages # total values is the ouput
model = keras.models.Model(inputs=[input_states], outputs=[Q_values])
# copt weights for target model
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

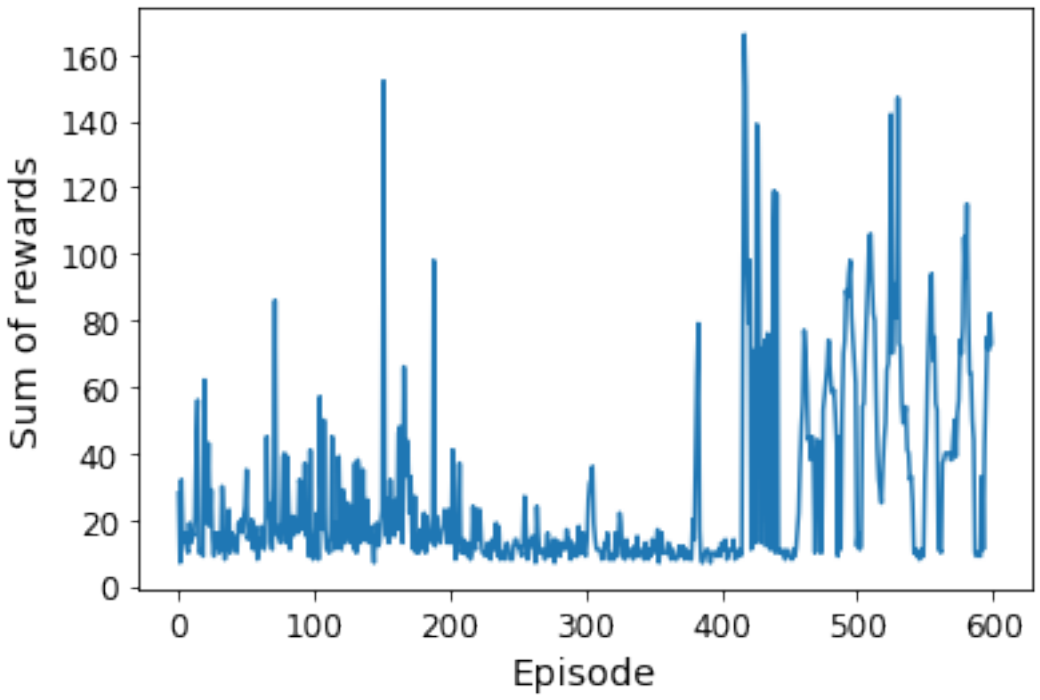
[103]:
```
batch_size = 32
discount_rate = 0.95
optimizer = keras.optimizers.Adam(lr=1e-2)
loss_fn = keras.losses.Huber() # Huber loss function: quadratice for small␣
↪values and linear for large values
#training function is the same as before
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_rate * next_best_Q_values)
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

[104]:
```
replay_memory = deque(maxlen=2000)
```

[105]:
```
env.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

rewards = []
best_score = 0

for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    rewards.append(step)
    if step > best_score:
        best_weights = model.get_weights()
        best_score = step
    print("\rEpisode: {}, Steps: {}, eps: {:.3f}".format(episode, step + 1,␣
↪epsilon), end="")
    # We mix duelling and double DQN
    if episode > 50:
        training_step(batch_size)
    if episode % 200 == 0:
        target.set_weights(model.get_weights())

model.set_weights(best_weights)
```

```
Episode: 599, Steps: 74, eps: 0.0100
```

[109]:
```
plt.plot(rewards)
plt.xlabel("Episode")
plt.ylabel("Sum of rewards")
plt.show()
```



[110]:
```
env.seed(42)
state = env.reset()

frames = []

for step in range(200):
    action = epsilon_greedy_policy(state)
    state, reward, done, info = env.step(action)
    if done:
        break
    img = env.render(mode="rgb_array")
    frames.append(img)

plot_animation(frames)
```

[110]:
```
<matplotlib.animation.FuncAnimation at 0x7f9a83070730>
```

This looks like a pretty robust agent!

[111]:
```
env.close()
```

18 Using TF-Agents to Beat Breakout

Let's use TF-Agents to create an agent that will learn to play Breakout. We will use the Deep Q-Learning algorithm, so you can easily compare the components with the previous implementation, but TF-Agents implements many other (and more sophisticated) algorithms!

18.1 TF-Agents Environments

[ ]:
```
tf.random.set_seed(42)
np.random.seed(42)
```

[ ]:
```
from tf_agents.environments import suite_gym

env = suite_gym.load("Breakout-v4")
env
```

[ ]:
```
<tf_agents.environments.wrappers.TimeLimit at 0x10b3788d0>
```

[ ]:
```
env.gym
```

[ ]:
```
<gym.envs.atari.atari_env.AtariEnv at 0x10a9d8518>
```

[ ]:
```
env.seed(42)
env.reset()
```

[ ]:
```
TimeStep(step_type=array(0, dtype=int32), reward=array(0., dtype=float32),
discount=array(1., dtype=float32), observation=array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
```

```
        [0., 0., 0.],
        [0., 0., 0.]],

       …,

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]]], dtype=float32))
```

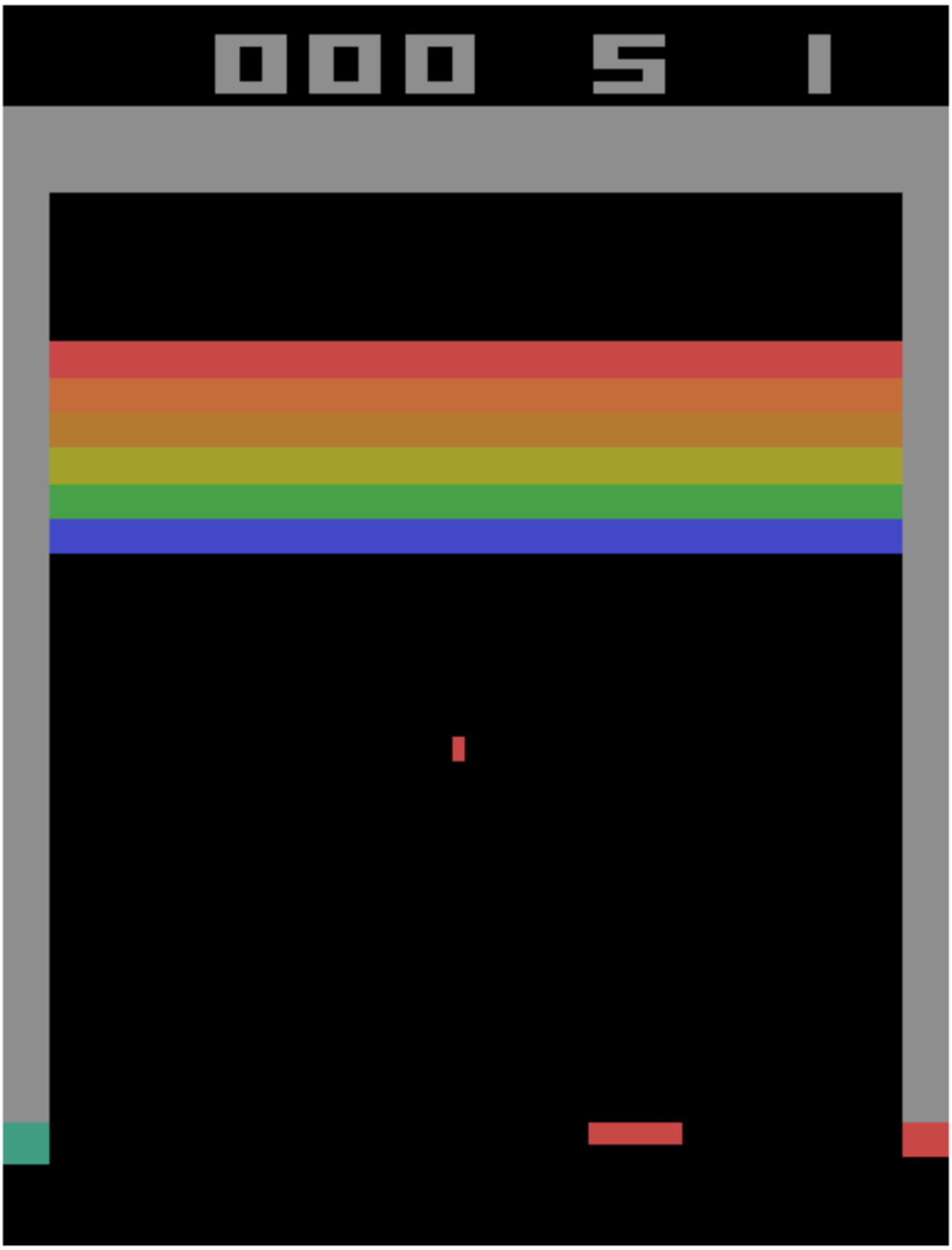
[ ]:
```
env.step(1) # Fire
```

[ ]:
```
TimeStep(step_type=array(1, dtype=int32), reward=array(0., dtype=float32),
discount=array(1., dtype=float32), observation=array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        …,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],
```

```
        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         …,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        …,

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         …,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         …,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         …,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]]], dtype=float32))
```

[ ]:
```
img = env.render(mode="rgb_array")

plt.figure(figsize=(6, 8))
plt.imshow(img)
plt.axis("off")
save_fig("breakout_plot")
plt.show()
```

```
Saving figure breakout_plot
```



[ ]:
```
env.current_time_step()
```

[ ]:
```
TimeStep(step_type=array(1, dtype=int32), reward=array(0., dtype=float32),
discount=array(1., dtype=float32), observation=array([[[0., 0., 0.],
```

```
 [0., 0., 0.],
 [0., 0., 0.],
 …,
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]],

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 …,
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]],

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 …,
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]],

…,

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 …,
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]],

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 …,
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]],

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 …,
 [0., 0., 0.],
 [0., 0., 0.],
```

```
        [0., 0., 0.]]], dtype=float32))
```

18.2 Environment Specifications

[ ]:
```
env.observation_spec()
```

[ ]:
```
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
minimum=[[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 …

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
```

```
 [0. 0. 0.]]

[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 …
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]], maximum=[[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

…

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
```

```
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]])
```

[ ]:
```
env.action_spec()
```

[ ]:
```
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None, minimum=0,
maximum=3)
```

[ ]:
```
env.time_step_spec()
```

[ ]:
```
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),
reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
discount=BoundedArraySpec(shape=(), dtype=dtype('float32'), name='discount',
minimum=0.0, maximum=1.0), observation=BoundedArraySpec(shape=(210, 160, 3),
dtype=dtype('float32'), name=None, minimum=[[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  …
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 …
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 …
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 …
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 …
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]], maximum=[[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]
```

```
 …

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]

[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 …
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]]))
```

18.3 Environment Wrappers

You can wrap a TF-Agents environments in a TF-Agents wrapper:

[ ]:
```
from tf_agents.environments.wrappers import ActionRepeat

repeating_env = ActionRepeat(env, times=4)
repeating_env
```

[ ]:
```
<tf_agents.environments.wrappers.ActionRepeat at 0x134eb87b8>
```

[ ]:
```
repeating_env.unwrapped
```

[ ]:
```
<gym.envs.atari.atari_env.AtariEnv at 0x10a9d8518>
```

Here is the list of available wrappers:

[ ]:
```
import tf_agents.environments.wrappers

for name in dir(tf_agents.environments.wrappers):
    obj = getattr(tf_agents.environments.wrappers, name)
```

```
    if hasattr(obj, "__base__") and issubclass(obj, tf_agents.environments.
↪wrappers.PyEnvironmentBaseWrapper):
        print("{:27s} {}".format(name, obj.__doc__.split("\n")[0]))
```

```
ActionClipWrapper           Wraps an environment and clips actions to spec
before applying.
ActionDiscretizeWrapper     Wraps an environment with continuous actions and
discretizes them.
ActionOffsetWrapper         Offsets actions to be zero-based.
ActionRepeat                Repeates actions over n-steps while acummulating the
received reward.
FlattenObservationsWrapper  Wraps an environment and flattens nested multi-
dimensional observations.
GoalReplayEnvWrapper        Adds a goal to the observation, used for HER
(Hindsight Experience Replay).
PyEnvironmentBaseWrapper    PyEnvironment wrapper forwards calls to the given
environment.
RunStats                    Wrapper that accumulates run statistics as the
environment iterates.
TimeLimit                   End episodes after specified number of steps.
```

The `suite_gym.load()` function can create an env and wrap it for you, both with TF-Agents environment wrappers and Gym environment wrappers (the latter are applied first).

[ ]:
```
from functools import partial
from gym.wrappers import TimeLimit

limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[partial(TimeLimit, max_episode_steps=10000)],
    env_wrappers=[partial(ActionRepeat, times=4)],
)
```

[ ]:
```
limited_repeating_env
```

[ ]:
```
<tf_agents.environments.wrappers.ActionRepeat at 0x135fa0400>
```

Create an Atari Breakout environment, and wrap it to apply the default Atari preprocessing steps:

[ ]:
```
limited_repeating_env.unwrapped
```

[ ]:
```
<gym.envs.atari.atari_env.AtariEnv at 0x135fa0588>
```

[ ]:
```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000 # <=> 108k ALE frames since 1 step = 4 frames
```

```
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

[ ]:
```
env
```

[ ]:
```
<tf_agents.environments.atari_wrappers.AtariTimeLimit at 0x135fc08d0>
```

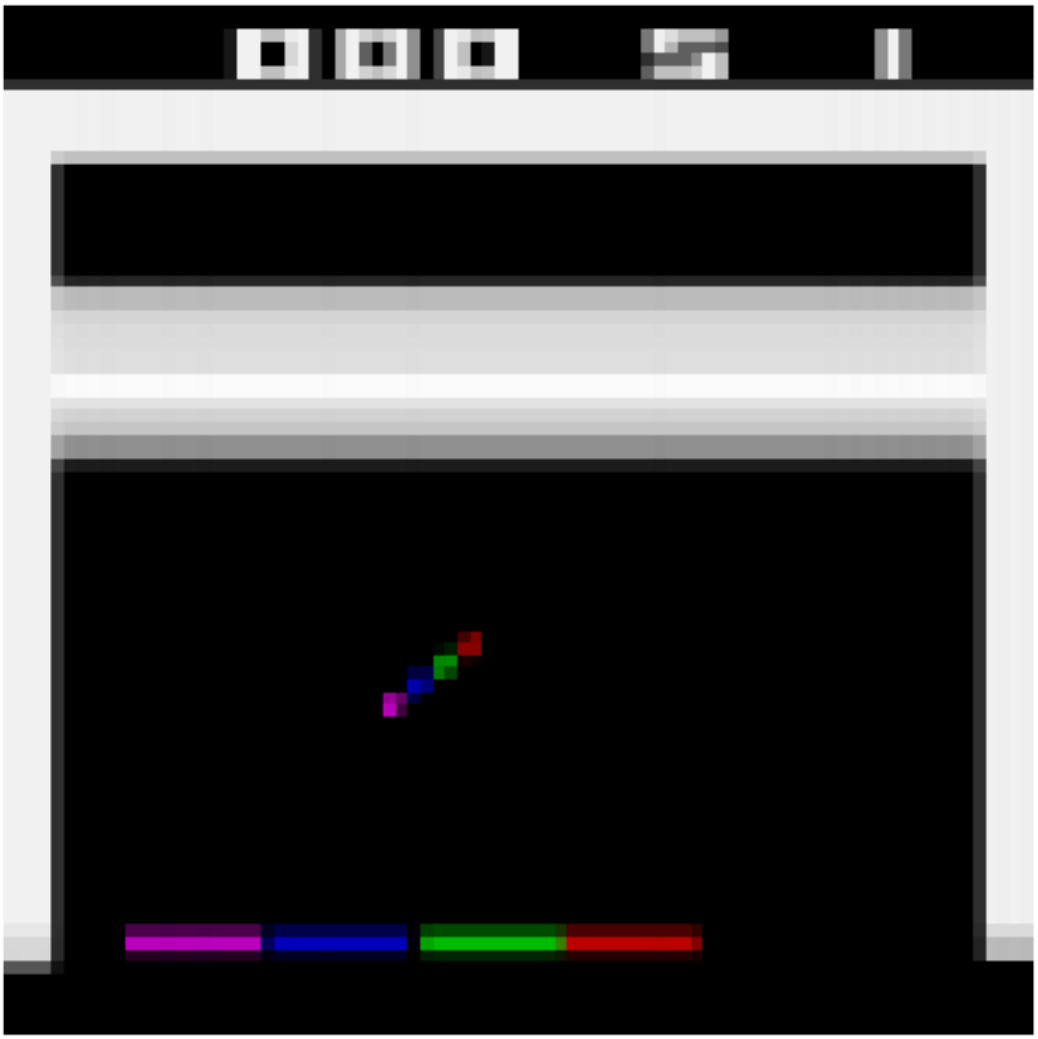
Play a few steps just to see what happens:

[ ]:
```
env.seed(42)
env.reset()
time_step = env.step(1) # FIRE
for _ in range(4):
    time_step = env.step(3) # LEFT
```

[ ]:
```
def plot_observation(obs):
    # Since there are only 3 color channels, you cannot display 4 frames
    # with one primary color per frame. So this code computes the delta between
    # the current frame and the mean of the other frames, and it adds this delta
    # to the red and blue channels to get a pink color for the current frame.
    obs = obs.astype(np.float32)
    img = obs[..., :3]
    current_frame_delta = np.maximum(obs[..., 3] - obs[..., :3].mean(axis=-1),␣
 ↪0.)
    img[..., 0] += current_frame_delta
    img[..., 2] += current_frame_delta
    img = np.clip(img / 150, 0, 1)
    plt.imshow(img)
    plt.axis("off")
```

[ ]:
```
plt.figure(figsize=(6, 6))
plot_observation(time_step.observation)
save_fig("preprocessed_breakout_plot")
plt.show()
```

```
Saving figure preprocessed_breakout_plot
```



Convert the Python environment to a TF environment:

[ ]:
```
from tf_agents.environments.tf_py_environment import TFPyEnvironment

tf_env = TFPyEnvironment(env)
```

18.4 Creating the DQN

Create a small class to normalize the observations. Images are stored using bytes from 0 to 255 to use less RAM, but we want to pass floats from 0.0 to 1.0 to the neural network:

Create the Q-Network:

[ ]:
```
from tf_agents.networks.q_network import QNetwork

preprocessing_layer = keras.layers.Lambda(
                          lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
```

Create the DQN Agent:

[ ]:
```
from tf_agents.agents.dqn.dqn_agent import DqnAgent

# see TF-agents issue #113
#optimizer = keras.optimizers.RMSprop(lr=2.5e-4, rho=0.95, momentum=0.0,
#                                     epsilon=0.00001, centered=True)

train_step = tf.Variable(0)
update_period = 4 # run a training step every 4 collect steps
optimizer = tf.compat.v1.train.RMSPropOptimizer(learning_rate=2.5e-4, decay=0.
 ↪95, momentum=0.0,
                                     epsilon=0.00001, centered=True)
epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0, # initial
    decay_steps=250000 // update_period, # <=> 1,000,000 ALE frames
    end_learning_rate=0.01) # final
agent = DqnAgent(tf_env.time_step_spec(),
                 tf_env.action_spec(),
                 q_network=q_net,
                 optimizer=optimizer,
                 target_update_period=2000, # <=> 32,000 ALE frames
                 td_errors_loss_fn=keras.losses.Huber(reduction="none"),
                 gamma=0.99, # discount factor
                 train_step_counter=train_step,
                 epsilon_greedy=lambda: epsilon_fn(train_step))
agent.initialize()
```

Create the replay buffer:

[ ]:
```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
```

```
    batch_size=tf_env.batch_size,
    max_length=1000000)

replay_buffer_observer = replay_buffer.add_batch
```

Create a simple custom observer that counts and displays the number of times it is called (except when it is passed a trajectory that represents the boundary between two episodes, as this does not count as a step):

[ ]:
```
class ShowProgress:
    def __init__(self, total):
        self.counter = 0
        self.total = total
    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

Let's add some training metrics:

[ ]:
```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```

[ ]:
```
train_metrics[0].result()
```

[ ]:
```
<tf.Tensor: id=469, shape=(), dtype=int64, numpy=0>
```

[ ]:
```
from tf_agents.eval.metric_utils import log_metrics
import logging
logging.getLogger().setLevel(logging.INFO)
log_metrics(train_metrics)
```

```
WARNING: Logging before flag parsing goes to stderr.
I0528 08:47:44.704986 140735810999168 metric_utils.py:47]
                  NumberOfEpisodes = 0
                  EnvironmentSteps = 0
                  AverageReturn = 0.0
                  AverageEpisodeLength = 0.0
```

Create the collect driver:

[ ]:
```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + train_metrics,
    num_steps=update_period) # collect 4 steps for each training iteration
```

Collect the initial experiences, before training:

[ ]:
```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                        tf_env.action_spec())
init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80,000 ALE frames
final_time_step, final_policy_state = init_driver.run()
```

```
W0528 08:47:44.747640 140735810999168 backprop.py:820] The dtype of the watched
tensor must be floating (e.g. tf.float32), got tf.int64
W0528 08:47:44.761187 140735810999168 backprop.py:820] The dtype of the watched
tensor must be floating (e.g. tf.float32), got tf.int64
W0528 08:47:44.765793 140735810999168 backprop.py:820] The dtype of the watched
tensor must be floating (e.g. tf.float32), got tf.int64
W0528 08:47:44.770788 140735810999168 backprop.py:820] The dtype of the watched
tensor must be floating (e.g. tf.float32), got tf.int64
W0528 08:47:44.775924 140735810999168 backprop.py:820] The dtype of the watched
tensor must be floating (e.g. tf.float32), got tf.int64
```

```
20000/20000
```

Let's sample 2 sub-episodes, with 3 time steps each and display them:

[ ]:
```
tf.random.set_seed(888) # chosen to show an example of trajectory at the end of␣
↪an episode

trajectories, buffer_info = replay_buffer.get_next(
    sample_batch_size=2, num_steps=3)
```

[ ]:
```
trajectories._fields
```

[ ]:
```
('step_type',
 'observation',
 'action',
 'policy_info',
 'next_step_type',
```

```
 'reward',
 'discount')
```

[ ]:
```
trajectories.observation.shape
```

[ ]:
```
TensorShape([2, 3, 84, 84, 4])
```

[ ]:
```
from tf_agents.trajectories.trajectory import to_transition

time_steps, action_steps, next_time_steps = to_transition(trajectories)
time_steps.observation.shape
```

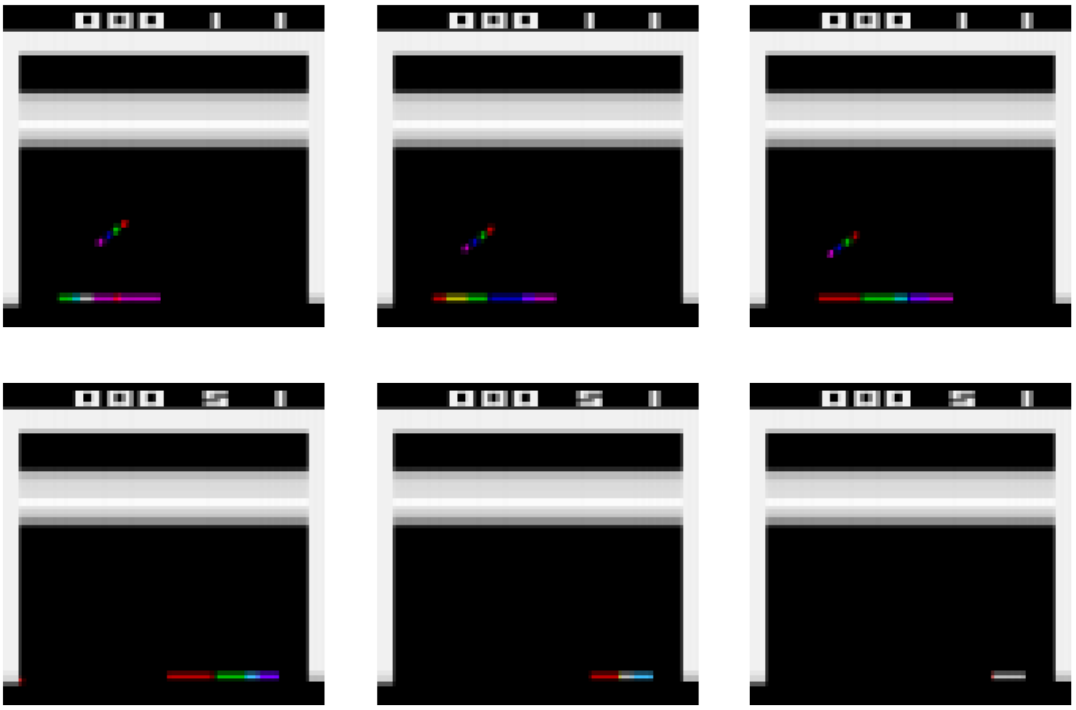
[ ]:
```
TensorShape([2, 2, 84, 84, 4])
```

[ ]:
```
trajectories.step_type.numpy()
```

[ ]:
```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

[ ]:
```
plt.figure(figsize=(10, 6.8))
for row in range(2):
    for col in range(3):
        plt.subplot(2, 3, row * 3 + col + 1)
        plot_observation(trajectories.observation[row, col].numpy())
plt.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0, wspace=0.02)
save_fig("sub_episodes_plot")
plt.show()
```

```
Saving figure sub_episodes_plot
```



Now let's create the dataset:

[ ]:
```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

Convert the main functions to TF Functions for better performance:

[ ]:
```
from tf_agents.utils.common import function

collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

And now we are ready to run the main loop!

[ ]:
```
def train_agent(n_iterations):
    time_step = None
    policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state = collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
```

```
        print("\r{} loss:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)
```

Run the next cell to train the agent for 10,000 steps. Then look at its behavior by running the following cell. You can run these two cells as many times as you wish. The agent will keep improving!

[ ]:
```
train_agent(n_iterations=10000)
```

```
W0528 08:49:00.697262 140735810999168 deprecation.py:323] From
/Users/ageron/miniconda3/envs/tf2/lib/python3.6/site-
packages/tensorflow/python/keras/optimizer_v2/learning_rate_schedule.py:409: div
(from tensorflow.python.ops.math_ops) is deprecated and will be removed in a
future version.
Instructions for updating:
Deprecated in favor of operator or tf.math.divide.
W0528 08:49:01.475340 140735810999168 deprecation.py:323] From
/Users/ageron/miniconda3/envs/tf2/lib/python3.6/site-
packages/tensorflow/python/ops/math_grad.py:1220:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
I0528 08:49:02.463025 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 0
                EnvironmentSteps = 4
                AverageReturn = 0.0
                AverageEpisodeLength = 0.0

997 loss:0.01551

I0528 08:50:16.405580 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 24
                EnvironmentSteps = 4004
                AverageReturn = 1.399999976158142
                AverageEpisodeLength = 180.5

2000 loss:0.00024

I0528 08:51:28.353239 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 47
                EnvironmentSteps = 8004
                AverageReturn = 0.8999999761581421
                AverageEpisodeLength = 165.89999389648438

2997 loss:0.00010

I0528 08:52:36.316717 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 69
```

```
                EnvironmentSteps = 12004
                AverageReturn = 0.800000011920929
                AverageEpisodeLength = 162.3000030517578

3999 loss:0.00751

I0528 08:53:47.764101 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 92
                EnvironmentSteps = 16004
                AverageReturn = 0.699999988079071
                AverageEpisodeLength = 161.89999389648438

4997 loss:0.00032

I0528 08:54:57.040647 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 111
                EnvironmentSteps = 20004
                AverageReturn = 1.0
                AverageEpisodeLength = 181.60000610351562

6000 loss:0.00006

I0528 08:56:07.210252 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 131
                EnvironmentSteps = 24004
                AverageReturn = 1.7000000476837158
                AverageEpisodeLength = 206.39999389648438

6999 loss:0.00784

I0528 08:57:18.494511 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 154
                EnvironmentSteps = 28004
                AverageReturn = 1.100000023841858
                AverageEpisodeLength = 182.6999969482422

7997 loss:0.00002

I0528 08:58:35.320452 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 175
                EnvironmentSteps = 32004
                AverageReturn = 1.7999999523162842
                AverageEpisodeLength = 196.60000610351562

8997 loss:0.00749

I0528 08:59:51.332596 140735810999168 metric_utils.py:47]
                NumberOfEpisodes = 195
                EnvironmentSteps = 36004
                AverageReturn = 1.100000023841858
                AverageEpisodeLength = 185.8000030517578

9999 loss:0.00001
```

[ ]:
```
frames = []
def save_frames(trajectory):
    global frames
    frames.append(tf_env.pyenv.envs[0].render(mode="rgb_array"))

prev_lives = tf_env.pyenv.envs[0].ale.lives()
def reset_and_fire_on_life_lost(trajectory):
    global prev_lives
    lives = tf_env.pyenv.envs[0].ale.lives()
    if prev_lives != lives:
        tf_env.reset()
        tf_env.pyenv.envs[0].step(1)
        prev_lives = lives

watch_driver = DynamicStepDriver(
    tf_env,
    agent.policy,
    observers=[save_frames, reset_and_fire_on_life_lost, ShowProgress(1000)],
    num_steps=1000)
final_time_step, final_policy_state = watch_driver.run()

plot_animation(frames)
```

If you want to save an animated GIF to show off your agent to your friends, here's one way to do it:

[ ]:
```
import PIL

image_path = os.path.join("images", "rl", "breakout.gif")
frame_images = [PIL.Image.fromarray(frame) for frame in frames[:150]]
frame_images[0].save(image_path, format='GIF',
                     append_images=frame_images[1:],
                     save_all=True,
                     duration=30,
                     loop=0)
```

[ ]:
```
%%html
<img src="images/rl/breakout.gif" />
```

```
<IPython.core.display.HTML object>
```

19 Extra material

19.1 Deque vs Rotating List

The `deque` class offers fast append, but fairly slow random access (for large replay memories):

[ ]:
```
from collections import deque
np.random.seed(42)

mem = deque(maxlen=1000000)
for i in range(1000000):
    mem.append(i)
[mem[i] for i in np.random.randint(1000000, size=5)]
```

[ ]:
```
[121958, 671155, 131932, 365838, 259178]
```

[ ]:
```
%timeit mem.append(1)
```

```
76.8 ns ± 0.31 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[ ]:
```
%timeit [mem[i] for i in np.random.randint(1000000, size=5)]
```

```
320 µs ± 23 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Alternatively, you could use a rotating list like this `ReplayMemory` class. This would make random access faster for large replay memories:

[ ]:
```
class ReplayMemory:
    def __init__(self, max_size):
        self.buffer = np.empty(max_size, dtype=np.object)
        self.max_size = max_size
        self.index = 0
        self.size = 0

    def append(self, obj):
        self.buffer[self.index] = obj
        self.size = min(self.size + 1, self.max_size)
        self.index = (self.index + 1) % self.max_size

    def sample(self, batch_size):
        indices = np.random.randint(self.size, size=batch_size)
        return self.buffer[indices]
```

[ ]:
```
mem = ReplayMemory(max_size=1000000)
for i in range(1000000):
    mem.append(i)
mem.sample(5)
```

[ ]:
```
array([757386, 904203, 190588, 595754, 865356], dtype=object)
```

[ ]:
```
%timeit mem.append(1)
```

```
761 ns ± 17.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

[ ]:
```
%timeit mem.sample(5)
```

```
2.97 µs ± 16.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

19.2 Creating a Custom TF-Agents Environment

To create a custom TF-Agent environment, you just need to write a class that inherits from the `PyEnvironment` class and implements a few methods. For example, the following minimal environment represents a simple 4x4 grid. The agent starts in one corner (0,0) and must move to the opposite corner (3,3). The episode is done if the agent reaches the goal (it gets a +10 reward) or if the agent goes out of bounds (-1 reward). The actions are up (0), down (1), left (2) and right (3).

[ ]:
```
class MyEnvironment(tf_agents.environments.py_environment.PyEnvironment):
    def __init__(self, discount=1.0):
        super().__init__()
        self._action_spec = tf_agents.specs.BoundedArraySpec(
            shape=(), dtype=np.int32, name="action", minimum=0, maximum=3)
        self._observation_spec = tf_agents.specs.BoundedArraySpec(
            shape=(4, 4), dtype=np.int32, name="observation", minimum=0,␣
↪maximum=1)
        self.discount = discount

    def action_spec(self):
        return self._action_spec

    def observation_spec(self):
        return self._observation_spec

    def _reset(self):
        self._state = np.zeros(2, dtype=np.int32)
        obs = np.zeros((4, 4), dtype=np.int32)
        obs[self._state[0], self._state[1]] = 1
        return tf_agents.trajectories.time_step.restart(obs)

    def _step(self, action):
        self._state += [(-1, 0), (+1, 0), (0, -1), (0, +1)][action]
        reward = 0
        obs = np.zeros((4, 4), dtype=np.int32)
        done = (self._state.min() < 0 or self._state.max() > 3)
        if not done:
            obs[self._state[0], self._state[1]] = 1
        if done or np.all(self._state == np.array([3, 3])):
            reward = -1 if done else +10
            return tf_agents.trajectories.time_step.termination(obs, reward)
        else:
            return tf_agents.trajectories.time_step.transition(obs, reward,
                                                               self.discount)
```

The action and observation specs will generally be instances of the `ArraySpec` or `BoundedArraySpec` classes from the `tf_agents.specs` package (check out the other specs in this package as well). Optionally, you can also define a `render()` method, a `close()` method to free resources, as well as a `time_step_spec()` method if you don't want the `reward` and `discount` to be 32-bit float scalars. Note that the base class takes care of keeping track of the current time step, which is why we must implement `_reset()` and `_step()` rather than `reset()` and `step()`.

[ ]:
```
my_env = MyEnvironment()
time_step = my_env.reset()
time_step
```

[ ]:
```
TimeStep(step_type=array(0, dtype=int32), reward=array(0., dtype=float32),
discount=array(1., dtype=float32), observation=array([[1, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int32))
```

[ ]:
```
time_step = my_env.step(1)
time_step
```

[ ]:
```
TimeStep(step_type=array(1, dtype=int32), reward=array(0., dtype=float32),
discount=array(1., dtype=float32), observation=array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int32))
```

[ ]:
```

```