Open in Colab

T81-558: Applications of Deep Neural Networks

Module 12: Reinforcement Learning

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 12 Video Material

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    from google.colab import drive
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

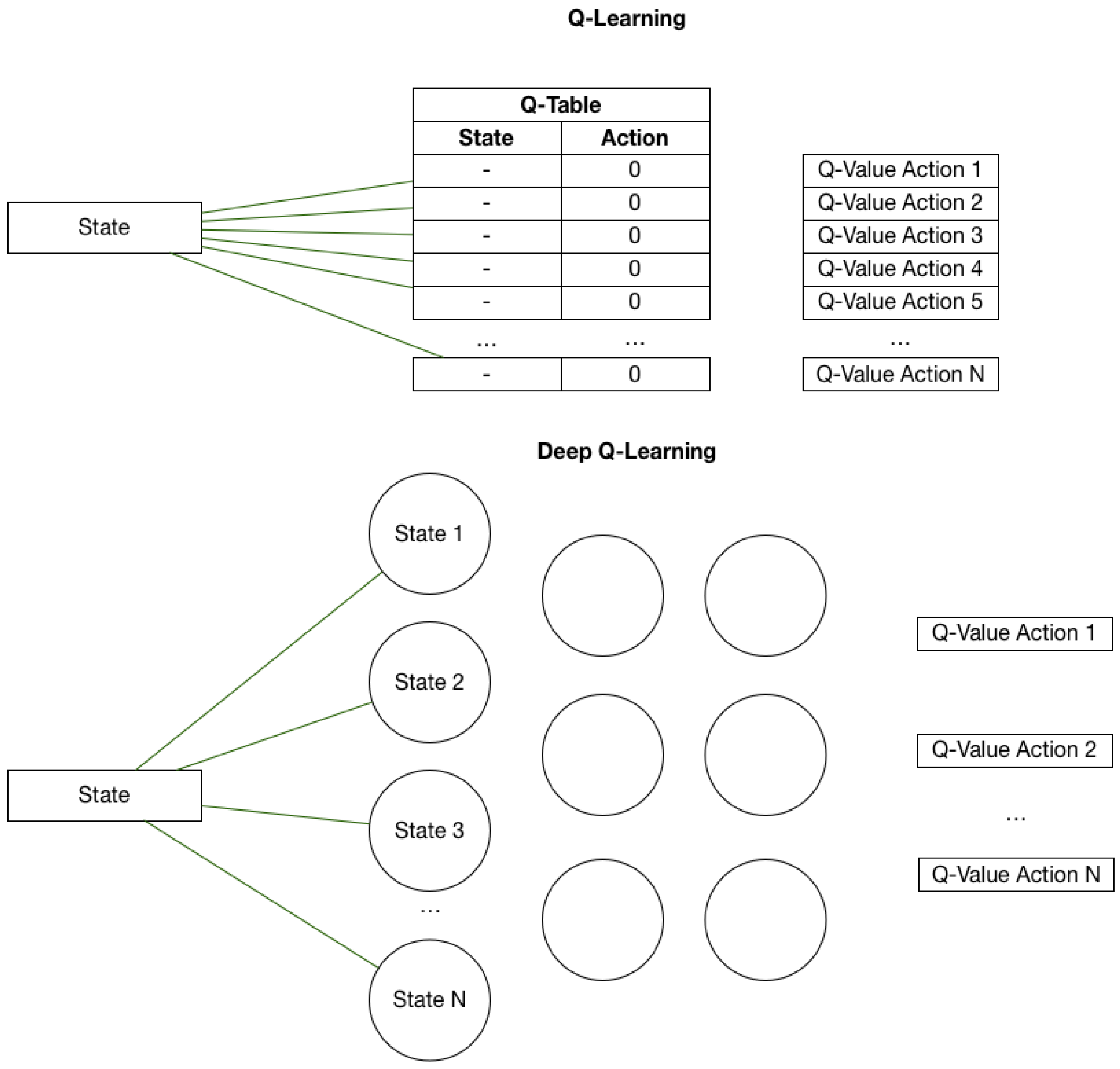
```
Note: using Google CoLab
```

Part 12.3: Keras Q-Learning in the OpenAI Gym

As we covered in the previous part, Q-Learning is a robust machine learning algorithm. Unfortunately, Q-Learning requires that the Q-table contain an entry for every possible state that the environment can take. Traditional Q-learning might be a good learning algorithm if the environment only includes a handful of discrete state elements. However, the Q-table can become prohibitively large if the state space is large.

Creating policies for large state spaces is a task that Deep Q-Learning Networks (DQN) can usually handle. Neural networks can generalize these states and learn commonalities. Unlike a table, a neural network does not require the program to represent every combination of state and action. A DQN maps the state to its input neurons and the action Q-values to the output neurons. The DQN effectively becomes a function that accepts the state and suggests action by returning the expected reward for each possible action. Figure 12.DQL demonstrates the DQN structure and mapping between state and action.

Figure 12.DQL: Deep Q-Learning (DQL)



As this diagram illustrates, the environment state contains several elements. For the basic DQN, the state can be a mix of continuous and categorical/discrete values. For the DQN, the discrete state elements the program typically encoded as dummy variables. The actions should be discrete when your program implements a DQN.

Other algorithms support continuous outputs, which we will discuss later in this chapter.

This chapter will use TF-Agents to implement a DQN to solve the cart-pole environment. TF-Agents makes designing, implementing, and testing new RL algorithms easier by providing well-tested modular components that can be modified and extended. It enables fast code iteration with functional test integration and benchmarking.

DQN and the Cart-Pole Problem

Barto (1983) first described the cart-pole problem. [Cite:barto1983neuronlike] A cart is connected to a rigid hinged pole. The cart is free to move only in the vertical plane of the cart/track. The agent can apply an impulsive "left" or "right" force F of a fixed magnitude to the cart at discrete time intervals. The cart-pole environment simulates the physics behind keeping the pole reasonably upright position on the cart. The environment has four state variables:

- x The position of the cart on the track.
- θ The angle of the pole with the vertical
- $\dot{x}$ The cart velocity.
- $\dot{\theta}$ The rate of change of the angle.

The action space consists of discrete actions:

- Apply force left
- Apply force right

To apply DQN to this problem, you need to create the following components for TF-Agents.

- Environment
- Agent
- Policies
- Metrics and Evaluation
- Replay Buffer
- Data Collection
- Training

These components are standard in most DQN implementations. Later, we will apply these same components to an Atari game, and after that, a problem with our design. This example is based on the cart-pole tutorial provided for TF-Agents.

First, we must install TF-Agents.

In [2]:
```
# HIDE OUTPUT
if COLAB:
  !sudo apt-get install -y xvfb ffmpeg x11-utils
  !pip install -q 'gym==0.10.11'
  !pip install -q 'imageio==2.4.0'
  !pip install -q PILLOW
  !pip install -q 'pyglet==1.3.2'
  !pip install -q pyvirtualdisplay
  !pip install -q tf-agents
  !pip install -q pygame
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
ffmpeg is already the newest version (7:3.4.8-0ubuntu0.2).
Suggested packages:
  mesa-utils
The following NEW packages will be installed:
  libxxf86dga1 x11-utils xvfb
0 upgraded, 3 newly installed, 0 to remove and 39 not upgraded.
Need to get 993 kB of archives.
After this operation, 2,982 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu bionic/main amd64 libxxf86dga1 amd64
2:1.1.4-1 [13.7 kB]
Get:2 http://archive.ubuntu.com/ubuntu bionic/main amd64 x11-utils amd64 7.7
+3build1 [196 kB]
Get:3 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 xvfb am
d64 2:1.19.6-1ubuntu4.10 [784 kB]
Fetched 993 kB in 0s (7,377 kB/s)
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based fr
ontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 7
6, <> line 3.)
debconf: falling back to frontend: Readline
debconf: unable to initialize frontend: Readline
debconf: (This frontend requires a controlling tty.)
debconf: falling back to frontend: Teletype
dpkg-preconfigure: unable to re-open stdin:
Selecting previously unselected package libxxf86dga1:amd64.
(Reading database ... 156210 files and directories currently installed.)
Preparing to unpack .../libxxf86dga1_2%3a1.1.4-1_amd64.deb ...
Unpacking libxxf86dga1:amd64 (2:1.1.4-1) ...
Selecting previously unselected package x11-utils.
Preparing to unpack .../x11-utils_7.7+3build1_amd64.deb ...
Unpacking x11-utils (7.7+3build1) ...
Selecting previously unselected package xvfb.
Preparing to unpack .../xvfb_2%3a1.19.6-1ubuntu4.10_amd64.deb ...
Unpacking xvfb (2:1.19.6-1ubuntu4.10) ...
Setting up xvfb (2:1.19.6-1ubuntu4.10) ...
Setting up libxxf86dga1:amd64 (2:1.1.4-1) ...
Setting up x11-utils (7.7+3build1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for libc-bin (2.27-3ubuntu1.3) ...
/sbin/ldconfig.real: /usr/local/lib/python3.7/dist-packages/ideep4py/lib/lib
mkldnn.so.0 is not a symbolic link

     |████████████████████████████████| 1.5 MB 16.4 MB/s
  Building wheel for gym (setup.py) ... done
     |████████████████████████████████| 3.3 MB 14.8 MB/s
  Building wheel for imageio (setup.py) ... done
ERROR: pip's dependency resolver does not currently take into account all th
e packages that are installed. This behaviour is the source of the following
dependency conflicts.
albumentations 0.1.12 requires imgaug<0.2.7,>=0.2.5, but you have imgaug 0.
2.9 which is incompatible.
     |████████████████████████████████| 1.0 MB 13.3 MB/s
     |████████████████████████████████| 1.3 MB 15.5 MB/s
```

```
    |████████████████████████████████| 626 kB 58.9 MB/s
 Installing build dependencies ... done
 Getting requirements to build wheel ... done
   Preparing wheel metadata ... done
 Building wheel for gym (PEP 517) ... done
    |████████████████████████████████| 21.8 MB 1.2 MB/s
```

We begin by importing needed Python libraries.

In [3]:
```python
import base64
import imageio
import IPython
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image
import pyvirtualdisplay

import tensorflow as tf

from tf_agents.agents.dqn import dqn_agent
from tf_agents.drivers import dynamic_step_driver
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.eval import metric_utils
from tf_agents.metrics import tf_metrics
from tf_agents.networks import q_network
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.utils import common
```

To allow this example to run in a notebook, we use a virtual display that will output an embedded video. If running this code outside a notebook, you could omit the virtual display and animate it directly to a window.

In [4]:
```python
# Set up a virtual display for rendering OpenAI gym environments.
display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

Hyperparameters

We must define Several hyperparameters for the algorithm to train the agent. The TF-Agent example provided reasonably well-tuned hyperparameters for cart-pole. Later we will adapt these to an Atari game.

In [5]:
```python
# How long should training run?
num_iterations = 20000
# How many initial random steps, before training start, to
# collect initial data.
initial_collect_steps = 1000
# How many steps should we run each iteration to collect
# data from.
```

```
collect_steps_per_iteration = 1
# How much data should we store for training examples.
replay_buffer_max_length = 100000

batch_size = 64
learning_rate = 1e-3
# How often should the program provide an update.
log_interval = 200

# How many episodes should the program use for each evaluation.
num_eval_episodes = 10
# How often should an evaluation occur.
eval_interval = 1000
```

Environment

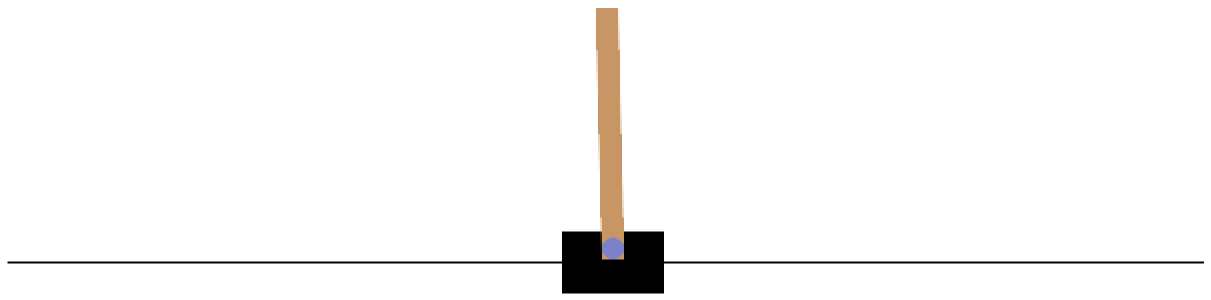
TF-Agents use OpenAI gym environments to represent the task or problem to be solved. Standard environments can be created in TF-Agents using **tf_agents.environments** suites. TF-Agents has suites for loading environments from sources such as the OpenAI Gym, Atari, and DM Control. We begin by loading the CartPole environment from the OpenAI Gym suite.

In [6]:
```
env_name = 'CartPole-v0'
env = suite_gym.load(env_name)
```

We will quickly render this environment to see the visual representation.

In [7]:
```
env.reset()
PIL.Image.fromarray(env.render())
```

Out[7]:



The `environment.step` method takes an `action` in the environment and returns a `TimeStep` tuple containing the following observation of the environment and the reward for the action.

The `time_step_spec()` method returns the specification for the `TimeStep` tuple. Its `observation` attribute shows the shape of observations, the data types, and the ranges of allowed values. The `reward` attribute shows the same details for the reward.

In [8]:
```
print('Observation Spec:')
print(env.time_step_spec().observation)
```

```
Observation Spec:
BoundedArraySpec(shape=(4,), dtype=dtype('float32'), name='observation', min
imum=[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], maximum=
[4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38])
```

In [9]:
```
print('Reward Spec:')
print(env.time_step_spec().reward)
```

```
Reward Spec:
ArraySpec(shape=(), dtype=dtype('float32'), name='reward')
```

The `action_spec()` method returns the shape, data types, and allowed values of valid actions.

In [10]:
```
print('Action Spec:')
print(env.action_spec())
```

```
Action Spec:
BoundedArraySpec(shape=(), dtype=dtype('int64'), name='action', minimum=0, m
aximum=1)
```

In the Cartpole environment:

- `observation` is an array of 4 floats:
  - the position and velocity of the cart
  - the angular position and velocity of the pole
- `reward` is a scalar float value
- `action` is a scalar integer with only two possible values:
  - `0` — "move left"
  - `1` — "move right"

In [11]:
```python
time_step = env.reset()
print('Time step:')
print(time_step)

action = np.array(1, dtype=np.int32)

next_time_step = env.step(action)
print('Next time step:')
print(next_time_step)
```

```
Time step:
TimeStep(
{'discount': array(1., dtype=float32),
 'observation': array([-0.03279859,  0.03562892, -0.04014493, -0.04911802],
dtype=float32),
 'reward': array(0., dtype=float32),
 'step_type': array(0, dtype=int32)})
Next time step:
TimeStep(
{'discount': array(1., dtype=float32),
 'observation': array([-0.03208601,  0.23130283, -0.04112729, -0.35419184],
dtype=float32),
 'reward': array(1., dtype=float32),
 'step_type': array(1, dtype=int32)})
```

Usually, the program instantiates two environments: one for training and one for evaluation.

In [12]:
```python
train_py_env = suite_gym.load(env_name)
eval_py_env = suite_gym.load(env_name)
```

The Cartpole environment, like most environments, is written in pure Python and is converted to TF-Agents and TensorFlow using the **TFPyEnvironment** wrapper. The original environment's API uses Numpy arrays. The **TFPyEnvironment** turns these to **Tensors** to make them compatible with Tensorflow agents and policies.

In [13]:
```
train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
```

Agent

An Agent represents the algorithm used to solve an RL problem. TF-Agents provides standard implementations of a variety of Agents:

- DQN (used in this example)
- REINFORCE
- DDPG
- TD3
- PPO
- SAC.

You can only use the DQN agent in environments with a discrete action space. The DQN uses a QNetwork, a neural network model that learns to predict Q-Values (expected returns) for all actions given a state from the environment.

The following code uses **tf_agents.networks.q_network** to create a QNetwork, passing in the **observation_spec**, **action_spec**, and a tuple describing the number and size of the model's hidden layers.

In [14]:
```
fc_layer_params = (100,)

q_net = q_network.QNetwork(
    train_env.observation_spec(),
    train_env.action_spec(),
    fc_layer_params=fc_layer_params)
```

Now we use **tf_agents.agents.dqn.dqn_agent** to instantiate a **DqnAgent**. In addition to the **time_step_spec**, **action_spec** and the QNetwork, the agent constructor also requires an optimizer (in this case, **AdamOptimizer**), a loss function, and an integer step counter.

In [15]:
```
optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate)

train_step_counter = tf.Variable(0)

agent = dqn_agent.DqnAgent(
    train_env.time_step_spec(),
    train_env.action_spec(),
    q_network=q_net,
    optimizer=optimizer,
    td_errors_loss_fn=common.element_wise_squared_loss,
    train_step_counter=train_step_counter)

agent.initialize()
```

Policies

A policy defines the way an agent acts in an environment. Typically, reinforcement learning aims to train the underlying model until the policy produces the desired outcome.

In this example:

- The desired outcome is keeping the pole balanced upright over the cart.
- The policy returns an action (left or right) for each `time_step` observation.

Agents contain two policies:

- **agent.policy** - The algorithm uses this main policy for evaluation and deployment.
- **agent.collect_policy** - The algorithm this secondary policy for data collection.

In [16]:
```
eval_policy = agent.policy
collect_policy = agent.collect_policy
```

You can create policies independently of agents. For example, use **random_tf_policy** to create a policy that will randomly select an action for each **time_step**. We will use this random policy to create initial collection data to begin training.

In [17]:
```
random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())
```

To get an action from a policy, call the **policy.action** method. The **time_step** contains the observation from the environment. This method returns a **PolicyStep**, which is a named tuple with three components:

- **action** - The action to be taken (in this case, 0 or 1).
- **state** - Used for stateful (that is, RNN-based) policies.
- **info** - Auxiliary data, such as log probabilities of actions.

Next, we create an environment and set up the random policy.

In [18]:
```python
example_environment = tf_py_environment.TFPyEnvironment(
    suite_gym.load('CartPole-v0'))
time_step = example_environment.reset()
random_policy.action(time_step)
```

Out[18]:
```
PolicyStep(action=<tf.Tensor: shape=(1,), dtype=int64, numpy=array([0])>, s
tate=(), info=())
```

Metrics and Evaluation

The most common metric used to evaluate a policy is the average return. The return is the sum of rewards obtained while running a policy in an environment for an episode. Several episodes are run, creating an average return. The following function computes the average return, given the policy, environment, and number of episodes. We will use this same evaluation for Atari.

In [19]:
```python
def compute_avg_return(environment, policy, num_episodes=10):

    total_return = 0.0
    for _ in range(num_episodes):

        time_step = environment.reset()
        episode_return = 0.0

        while not time_step.is_last():
            action_step = policy.action(time_step)
            time_step = environment.step(action_step.action)
            episode_return += time_step.reward
        total_return += episode_return

    avg_return = total_return / num_episodes
    return avg_return.numpy()[0]


# See also the metrics module for standard implementations
# of different metrics.
# https://github.com/tensorflow/agents/tree/master/tf_agents/metrics
```

Running this computation on the `random_policy` shows a baseline performance in the environment.

In [20]:
```python
compute_avg_return(eval_env, random_policy, num_eval_episodes)
```

Out[20]:
```
15.2
```

Replay Buffer

The replay buffer keeps track of data collected from the environment. This tutorial uses **TFUniformReplayBuffer**. The constructor requires the specs for the data it will be collecting. This value is available from the agent using the **collect_data_spec** method. The batch size and maximum buffer length are also required.

In [21]:
```
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=train_env.batch_size,
    max_length=replay_buffer_max_length)
```

For most agents, **collect_data_spec** is a named tuple called **Trajectory**, containing the specs for observations, actions, rewards, and other items.

In [22]:
```
agent.collect_data_spec
```

Out[22]:
```
Trajectory(
{'action': BoundedTensorSpec(shape=(), dtype=tf.int64, name='action', minim
um=array(0), maximum=array(1)),
 'discount': BoundedTensorSpec(shape=(), dtype=tf.float32, name='discount',
minimum=array(0., dtype=float32), maximum=array(1., dtype=float32)),
 'next_step_type': TensorSpec(shape=(), dtype=tf.int32, name='step_type'),
 'observation': BoundedTensorSpec(shape=(4,), dtype=tf.float32, name='obser
vation', minimum=array([-4.8000002e+00, -3.4028235e+38, -4.1887903e-01, -3.
4028235e+38],
      dtype=float32), maximum=array([4.8000002e+00, 3.4028235e+38, 4.188790
3e-01, 3.4028235e+38],
      dtype=float32)),
 'policy_info': (),
 'reward': TensorSpec(shape=(), dtype=tf.float32, name='reward'),
 'step_type': TensorSpec(shape=(), dtype=tf.int32, name='step_type')})
```

Data Collection

Now execute the random policy in the environment for a few steps, recording the data in the replay buffer.

In [23]:
```
def collect_step(environment, policy, buffer):
    time_step = environment.current_time_step()
    action_step = policy.action(time_step)
    next_time_step = environment.step(action_step.action)
    traj = trajectory.from_transition(time_step, action_step, \
                                      next_time_step)

    # Add trajectory to the replay buffer
    buffer.add_batch(traj)


def collect_data(env, policy, buffer, steps):
    for _ in range(steps):
        collect_step(env, policy, buffer)
```

```
collect_data(train_env, random_policy, replay_buffer, steps=100)

# This loop is so common in RL, that we provide standard implementations.
# For more details see the drivers module.
# https://www.tensorflow.org/agents/api_docs/python/tf_agents/drivers
```

The replay buffer is now a collection of Trajectories. The agent needs access to the replay buffer. TF-Agents provides this access by creating an iterable **tf.data.Dataset** pipeline, which will feed data to the agent.

Each row of the replay buffer only stores a single observation step. But since the DQN Agent needs both the current and following observation to compute the loss, the dataset pipeline will sample two adjacent rows for each item in the batch (**num_steps=2**).

The program also optimizes this dataset by running parallel calls and prefetching data.

In [24]:
```
# Dataset generates trajectories with shape [Bx2x...]
dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size,
    num_steps=2).prefetch(3)


dataset
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/py
thon/autograph/impl/api.py:377: ReplayBuffer.get_next (from tf_agents.replay
_buffers.replay_buffer) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Use `as_dataset(..., single_deterministic_pass=False) instead.
```

Out[24]:
```
<PrefetchDataset element_spec=(Trajectory(
{'action': TensorSpec(shape=(64, 2), dtype=tf.int64, name=None),
 'discount': TensorSpec(shape=(64, 2), dtype=tf.float32, name=None),
 'next_step_type': TensorSpec(shape=(64, 2), dtype=tf.int32, name=None),
 'observation': TensorSpec(shape=(64, 2, 4), dtype=tf.float32, name=None),
 'policy_info': (),
 'reward': TensorSpec(shape=(64, 2), dtype=tf.float32, name=None),
 'step_type': TensorSpec(shape=(64, 2), dtype=tf.int32, name=None)}), Buffe
rInfo(ids=TensorSpec(shape=(64, 2), dtype=tf.int64, name=None), probabiliti
es=TensorSpec(shape=(64,), dtype=tf.float32, name=None)))>
```

In [25]:
```
iterator = iter(dataset)

print(iterator)
```

```
<tensorflow.python.data.ops.iterator_ops.OwnedIterator object at 0x7f05c0006
c10>
```

Training the agent

Two things must happen during the training loop:

- Collect data from the environment
- Use that data to train the agent's neural network(s)

This example also periodically evaluates the policy and prints the current score.

The following will take ~5 minutes to run.

In [26]:
```
# (Optional) Optimize by wrapping some of the code in a graph
# using TF function.
agent.train = common.function(agent.train)

# Reset the train step
agent.train_step_counter.assign(0)

# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, agent.policy,
                                num_eval_episodes)
returns = [avg_return]

for _ in range(num_iterations):

    # Collect a few steps using collect_policy and
    # save to the replay buffer.
    for _ in range(collect_steps_per_iteration):
        collect_step(train_env, agent.collect_policy, replay_buffer)

    # Sample a batch of data from the buffer and update
    # the agent's network.
    experience, unused_info = next(iterator)
    train_loss = agent.train(experience).loss

    step = agent.train_step_counter.numpy()

    if step % log_interval == 0:
        print('step = {0}: loss = {1}'.format(step, train_loss))

    if step % eval_interval == 0:
        avg_return = compute_avg_return(eval_env, agent.policy,
                                        num_eval_episodes)
        print('step = {0}: Average Return = {1}'.format(step, avg_return))
        returns.append(avg_return)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/py
thon/util/dispatch.py:1082: calling foldr_v2 (from tensorflow.python.ops.fun
ctional_ops) with back_prop=False is deprecated and will be removed in a fut
ure version.
Instructions for updating:
back_prop=False is deprecated. Consider using tf.stop_gradient instead.
Instead of:
results = tf.foldr(fn, elems, back_prop=False)
Use:
results = tf.nest.map_structure(tf.stop_gradient, tf.foldr(fn, elems))
step = 200: loss = 23.158374786376953
step = 400: loss = 7.158817768096924
step = 600: loss = 30.97699737548828
step = 800: loss = 9.831337928771973
step = 1000: loss = 9.664928436279297
step = 1000: Average Return = 13.199999809265137
step = 1200: loss = 10.27550220489502
step = 1400: loss = 17.813919067382812
step = 1600: loss = 8.011082649230957
step = 1800: loss = 28.170230865478516
step = 2000: loss = 28.02679443359375
step = 2000: Average Return = 40.099998474121094
step = 2200: loss = 3.2347989082336426
step = 2400: loss = 55.28818893432617
step = 2600: loss = 18.277198791503906
step = 2800: loss = 4.626098155975342
step = 3000: loss = 14.528213500976562
step = 3000: Average Return = 61.79999923706055
step = 3200: loss = 59.28561782836914
step = 3400: loss = 76.63031005859375
step = 3600: loss = 63.14342498779297
step = 3800: loss = 110.42674255371094
step = 4000: loss = 5.175446510314941
step = 4000: Average Return = 146.0
step = 4200: loss = 5.881635665893555
step = 4400: loss = 7.868609428405762
step = 4600: loss = 6.599028587341309
step = 4800: loss = 69.79821014404297
step = 5000: loss = 219.33493041992188
step = 5000: Average Return = 119.5999984741211
step = 5200: loss = 45.901084899902344
step = 5400: loss = 4.329599380493164
step = 5600: loss = 64.89082336425781
step = 5800: loss = 42.906700134277344
step = 6000: loss = 10.80639362335205
step = 6000: Average Return = 190.1999969482422
step = 6200: loss = 33.47439193725586
step = 6400: loss = 9.312165260314941
step = 6600: loss = 124.09418487548828
step = 6800: loss = 9.645675659179688
step = 7000: loss = 12.178140640258789
step = 7000: Average Return = 184.60000610351562
step = 7200: loss = 9.637611389160156
step = 7400: loss = 7.316198348999023
step = 7600: loss = 139.92269897460938
step = 7800: loss = 7.6530256271362305
step = 8000: loss = 16.512592315673828
step = 8000: Average Return = 191.3000030517578
step = 8200: loss = 104.84465789794922
step = 8400: loss = 164.78646850585938
step = 8600: loss = 124.94630432128906
step = 8800: loss = 13.862188339233398
step = 9000: loss = 154.0853271484375
step = 9000: Average Return = 200.0
step = 9200: loss = 125.97179412841797
step = 9400: loss = 153.52854919433594
step = 9600: loss = 97.84358215332031
step = 9800: loss = 55.670570373535156
step = 10000: loss = 15.632448196411133
step = 10000: Average Return = 200.0
step = 10200: loss = 8.194206237792969
step = 10400: loss = 13.93640422821045
step = 10600: loss = 11.790799140930176
step = 10800: loss = 444.7298278808594
step = 11000: loss = 378.853271484375
step = 11000: Average Return = 197.0
step = 11200: loss = 13.082895278930664
step = 11400: loss = 268.9317626953125
step = 11600: loss = 123.26766204833984
step = 11800: loss = 81.99503326416016
step = 12000: loss = 94.90630340576172
step = 12000: Average Return = 200.0
step = 12200: loss = 10.287437438964844
step = 12400: loss = 275.0940246582031
step = 12600: loss = 115.75547790527344
step = 12800: loss = 668.2427978515625
step = 13000: loss = 798.7186279296875
step = 13000: Average Return = 196.60000610351562
step = 13200: loss = 21.640256881713867
step = 13400: loss = 313.7167663574219
step = 13600: loss = 17.465240478515625
step = 13800: loss = 715.4552001953125
step = 14000: loss = 13.271897315979004
step = 14000: Average Return = 197.89999389648438
step = 14200: loss = 20.86071014404297
step = 14400: loss = 86.7576904296875
step = 14600: loss = 529.219970703125
step = 14800: loss = 969.0336303710938
step = 15000: loss = 298.5212707519531
step = 15000: Average Return = 198.5
step = 15200: loss = 372.925537109375
step = 15400: loss = 214.28077697753906
step = 15600: loss = 11.535277366638184
step = 15800: loss = 40.361358642578125
step = 16000: loss = 19.93735122680664
step = 16000: Average Return = 199.39999389648438
step = 16200: loss = 32.60084533691406
step = 16400: loss = 18.340595245361328
step = 16600: loss = 16.289039611816406
step = 16800: loss = 189.3881378173828
step = 17000: loss = 39.77809143066406
step = 17000: Average Return = 200.0
step = 17200: loss = 74.69547271728516
step = 17400: loss = 83.34622192382812
step = 17600: loss = 167.67913818359375
step = 17800: loss = 1286.816650390625
step = 18000: loss = 4.552798271179199
step = 18000: Average Return = 200.0
step = 18200: loss = 1149.6190185546875
step = 18400: loss = 39.40950012207031
step = 18600: loss = 785.230712890625
step = 18800: loss = 20.107412338256836
step = 19000: loss = 483.009765625
step = 19000: Average Return = 200.0
step = 19200: loss = 5.911262512207031
step = 19400: loss = 16.59900665283203
step = 19600: loss = 16.253849029541016
step = 19800: loss = 124.63180541992188
step = 20000: loss = 22.45917320251465
step = 20000: Average Return = 198.3000030517578
```

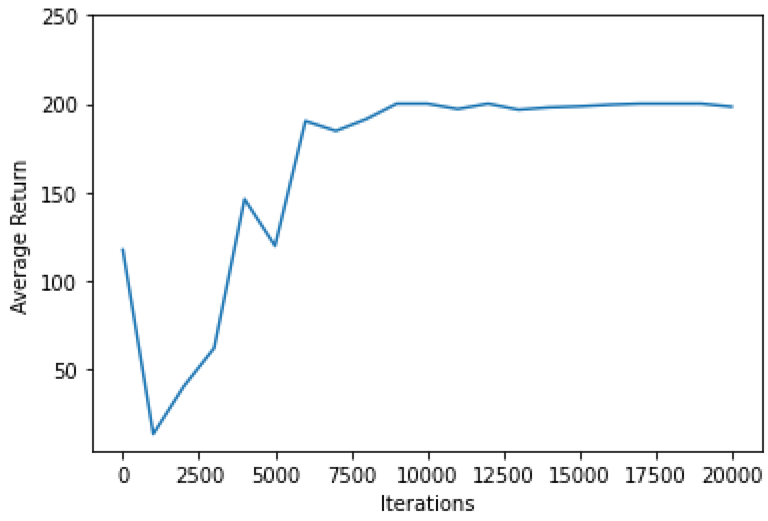
Visualization and Plots

Use **matplotlib.pyplot** to chart how the policy improved during training.

One iteration of **Cartpole-v0** consists of 200 time steps. The environment rewards `+1` for each step the pole stays up, so the maximum return for one episode is 200. The charts show the return increasing towards that maximum each time the algorithm evaluates it during training. (It may be a little unstable and not increase each time monotonically.)

In [27]:
```
iterations = range(0, num_iterations + 1, eval_interval)
plt.plot(iterations, returns)
plt.ylabel('Average Return')
plt.xlabel('Iterations')
plt.ylim(top=250)
```

Out[27]:
```
(3.859999799728394, 250.0)
```



Videos

The charts are nice. But more exciting is seeing an agent performing a task in an environment.

First, create a function to embed videos in the notebook.

```
In [28]: def embed_mp4(filename):
    """Embeds an mp4 file in the notebook."""
    video = open(filename, 'rb').read()
    b64 = base64.b64encode(video)
    tag = '''
  <video width="640" height="480" controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4">
  Your browser does not support the video tag.
  </video>'''.format(b64.decode())

    return IPython.display.HTML(tag)
```

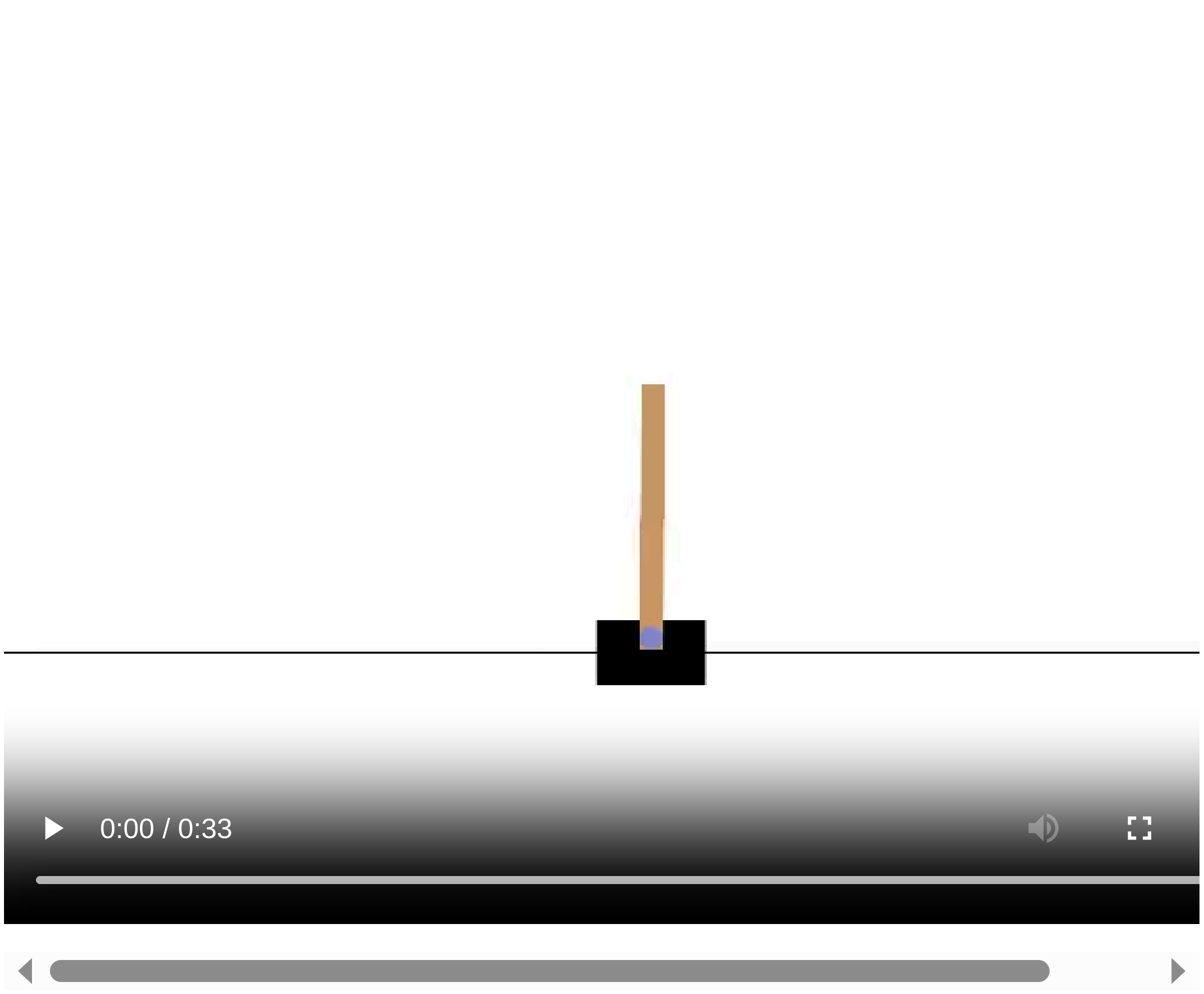
Now iterate through a few episodes of the Cartpole game with the agent. The underlying Python environment (the one "inside" the TensorFlow environment wrapper) provides a `render()` method, which outputs an image of the environment state. We can collect these frames into a video.

```
In [29]: # HIDE OUTPUT
def create_policy_eval_video(policy, filename, num_episodes=5, fps=30):
    filename = filename + ".mp4"
    with imageio.get_writer(filename, fps=fps) as video:
        for _ in range(num_episodes):
            time_step = eval_env.reset()
            video.append_data(eval_py_env.render())
            while not time_step.is_last():
                action_step = policy.action(time_step)
                time_step = eval_env.step(action_step.action)
                video.append_data(eval_py_env.render())
    return embed_mp4(filename)


create_policy_eval_video(agent.policy, "trained-agent")
```

```
WARNING:root:IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by
macro_block_size=16, resizing from (400, 600) to (400, 608) to ensure video
compatibility with most codecs and players. To prevent resizing, make your i
nput image divisible by the macro_block_size or set the macro_block_size to
None (risking incompatibility). You may also see a FFMPEG warning concerning
speedloss due to data not being aligned.
/usr/local/lib/python3.7/dist-packages/imageio/plugins/ffmpeg.py:727: Deprec
ationWarning: tostring() is deprecated. Use tobytes() instead.
  self._proc.stdin.write(im.tostring())
```

Out[29]:



For fun, compare the trained agent (above) to an agent moving randomly. (It does not do as well.)

In [30]:
```
# HIDE OUTPUT
create_policy_eval_video(random_policy, "random-agent")
```

```
WARNING:root:IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by
macro_block_size=16, resizing from (400, 600) to (400, 608) to ensure video
compatibility with most codecs and players. To prevent resizing, make your i
nput image divisible by the macro_block_size or set the macro_block_size to
None (risking incompatibility). You may also see a FFMPEG warning concerning
speedloss due to data not being aligned.
/usr/local/lib/python3.7/dist-packages/imageio/plugins/ffmpeg.py:727: Deprec
ationWarning: tostring() is deprecated. Use tobytes() instead.
  self._proc.stdin.write(im.tostring())
```

```
Out[30]:
```