spring is just a copy of its parent[7] 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.

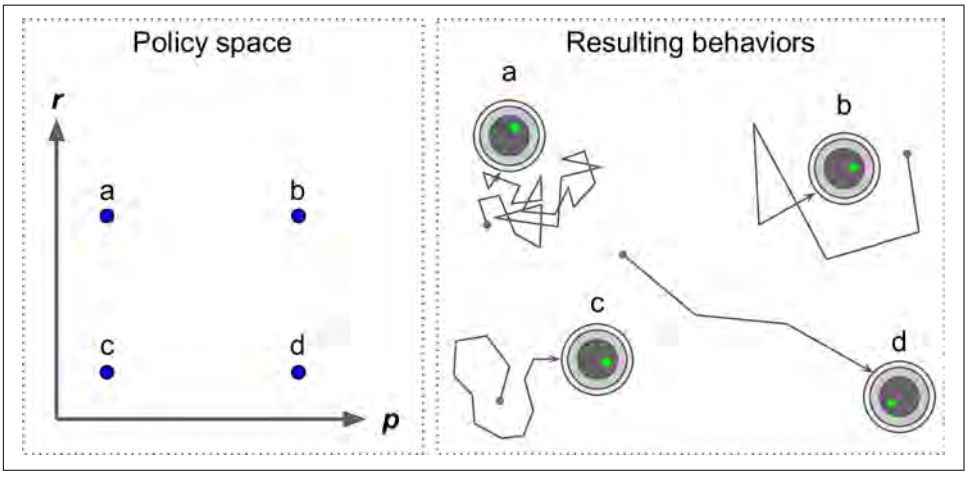


*Figure 16-3. Four points in policy space and the agent's corresponding behavior*

Yet another approach is to use optimization techniques, by evaluating 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), which we will discuss in more detail later in this chapter. For example, going back to the vacuum cleaner robot, you could slightly 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 will implement a popular PG algorithm using TensorFlow, but before we do we need to create an environment for the agent to live in, so it's time to introduce OpenAI gym.

# Introduction to OpenAI Gym

One of the challenges of Reinforcement Learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world and you can directly train your robot in that environment, but this has its limits: if the robot falls off a cliff, you can't just click "undo." You can't speed up time either; adding more computing

7 If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring's genome (in this case a set of policy parameters) is randomly composed of parts of its parents' genomes.

power won't make the robot move any faster. And it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least to bootstrap training.

*OpenAI gym*[8] is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

Let's install OpenAI gym. For a minimal OpenAI gym installation, simply use pip:

```
$ pip3 install --upgrade gym
```

Next open up a Python shell or a Jupyter notebook and create your first environment:

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2016-10-14 16:03:23,199] Making new env: MsPacman-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846, -0.03288115,  0.02337094,  0.00720711])
>>> env.render()
```

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 (see Figure 16-4). After the environment is created, we must initialize it using the `reset()` method. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a 1D NumPy array containing four floats: these floats represent the cart's horizontal position (`0.0` = center), its velocity, the angle of the pole (`0.0` = vertical), and its angular velocity. Finally, the `render()` method displays the environment as shown in Figure 16-4.

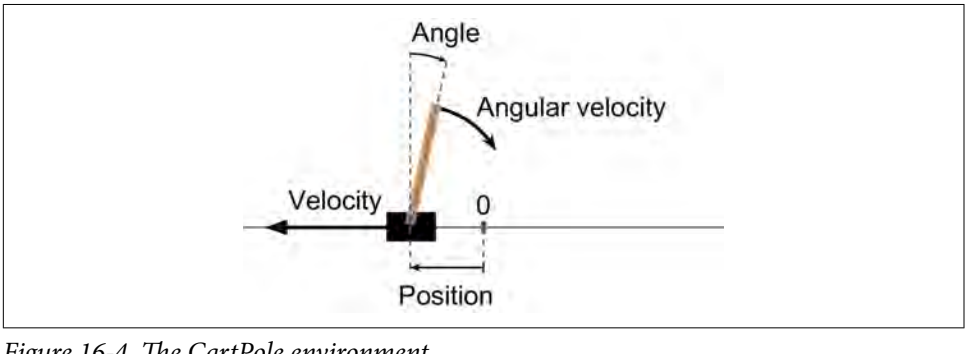


*Figure 16-4. The CartPole environment*

8 OpenAI is a nonprofit artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate it).

If you want `render()` to return the rendered image as a NumPy array, you can set the `mode` parameter to `rgb_array` (note that other environments may support different modes):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape  # height, width, channels (3=RGB)
(400, 600, 3)
```

Unfortunately, the CartPole (and a few other environments) renders the image to the screen even if you set the mode to `"rgb_array"`. The only way to avoid this is to use a fake X server such as Xvfb or Xdummy. For example, you can install Xvfb and start Python using the following command: `xvfb-run -s "-screen 0 1400x900x24" python`. Or use the xvfbwrapper package.

Let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1). Other environments may have more discrete actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right, let's accelerate the cart toward the right:

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.03865608,  0.16189797,  0.02351508, -0.27801135])
>>> reward
1.0
>>> done
False
>>> info
{}
```

The `step()` method executes the given action and returns four values:

`obs`
: This is the new observation. 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.

`reward`
: In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep running as long as possible.

`done`
: This value will be `True` when the *episode* is over. This will happen when the pole tilts too much. After that, the environment must be reset before it can be used again.

`info`
: This dictionary may provide extra debug information in other environments. This data should not be used for training (it would be cheating).

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000): # 1000 steps max, we don't want to run forever
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

This code is hopefully self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(42.125999999999998, 9.1237121830974033, 24.0, 68.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps. Not great. If you look at the simulation in the Jupyter notebooks, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. Let's see if a neural network can come up with a better policy.

# Neural Network Policies

Let's create a neural network policy. Just like the policy we hardcoded earlier, this neural network will take an observation as input, and it will output the action to be executed. More precisely, it will estimate a probability for each action, and then we will select an action randomly according to the estimated probabilities (see Figure 16-5). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability $p$ of action 0 (left), and of course the probability of action 1 (right) will be $1 - p$.