16. Reinforcement Learning - Hands-On Machine Learning with Scikit-Learn and TensorFlow

Chapter 16. Reinforcement Learning

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years, in particular in games (e.g., *TD-Gammon*, a *Backgammon* playing program) and in machine control, but seldom making the headline news. But a revolution took place in 2013 when researchers from an English startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch, eventually outperforming humans in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games. This was the first of a series of amazing feats, culminating in March 2016 with the victory of their system AlphaGo against Lee Sedol, the world champion of the game of *Go*. No program had ever come close to beating a master of this game, let alone the world champion. Today the whole field of RL is boiling with new ideas, with a wide range of applications. DeepMind was bought by Google for over 500 million dollars in 2014.

So how did they do it? With hindsight it seems rather simple: they applied the power of Deep Learning to the field of Reinforcement Learning, and it worked beyond their wildest dreams. In this chapter we will first explain what Reinforcement Learning is and what it is good at, and then we will present two of the most important techniques in deep Reinforcement Learning: *policy gradients* and *deep Q-networks* (DQN), including a discussion of *Markov decision processes* (MDP). We will use these techniques to train a model to balance a pole on a moving cart, and another to play Atari games. The same techniques can be used for a wide variety of tasks, from walking robots to self-driving cars.

Learning to Optimize Rewards

In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*. Its objective is to learn to act in a way that will maximize its expected long-term rewards. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see Figure 16-1):

a. The agent can be the program controlling a walking robot. In this case, the environment is the 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.

b. The agent can be the program controlling Ms. Pac-Man. In this case, 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.

c. Similarly, the agent can be the program playing a board game such as the game of *Go*.

d. The agent does not have to control a physically (or virtually) moving thing. For example, it can be a 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.

e. The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

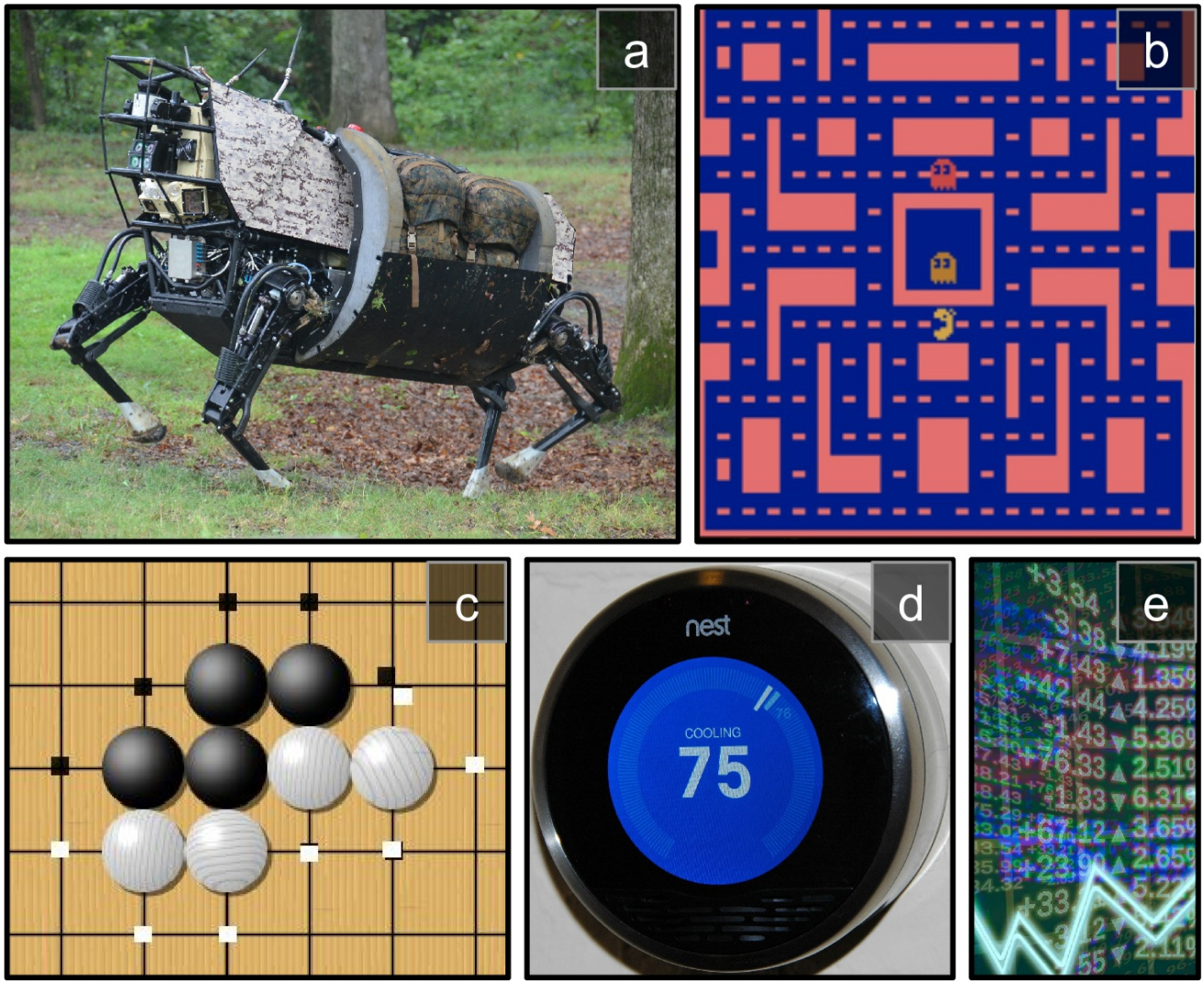

Figure 16-1. Reinforcement Learning examples: (a) walking robot, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader

Note that 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! There are many other examples of tasks where Reinforcement Learning is well suited, such as self-driving cars, placing ads on a web page, or controlling where an image classification system should focus its attention.

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 (see Figure 16-2).

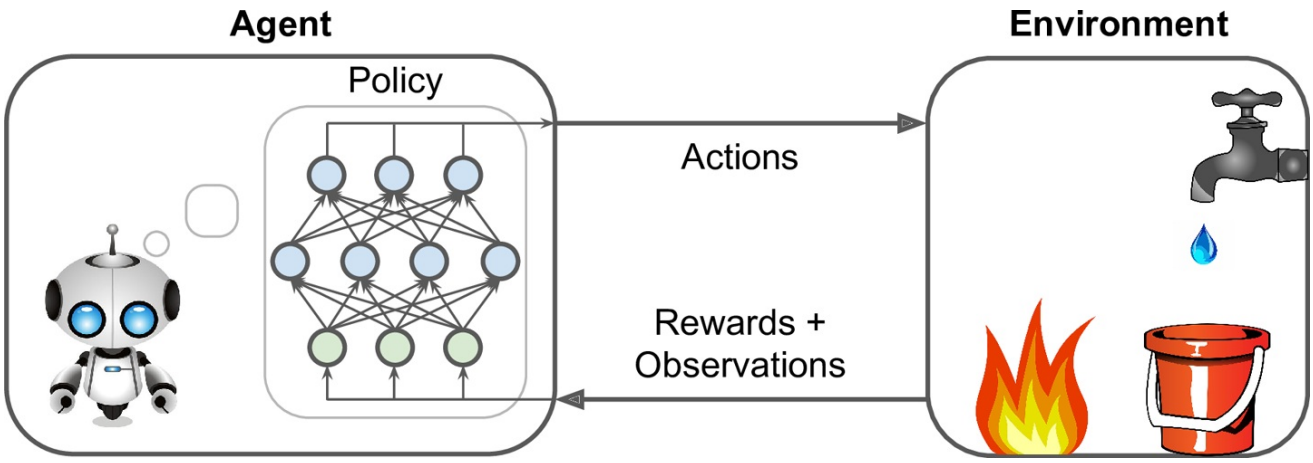

Figure 16-2. Reinforcement Learning using a neural network policy

The policy can be any algorithm you can think of, and it does not even have to be deterministic. For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability *p* every second, or randomly rotate left or right with probability 1 – *p*. The rotation angle would be a random angle between –r and +r. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is: how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability *p* and the angle range *r*. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 16-3). This is an example of *policy search*, in this case using a brute force approach. However, when the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could 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.

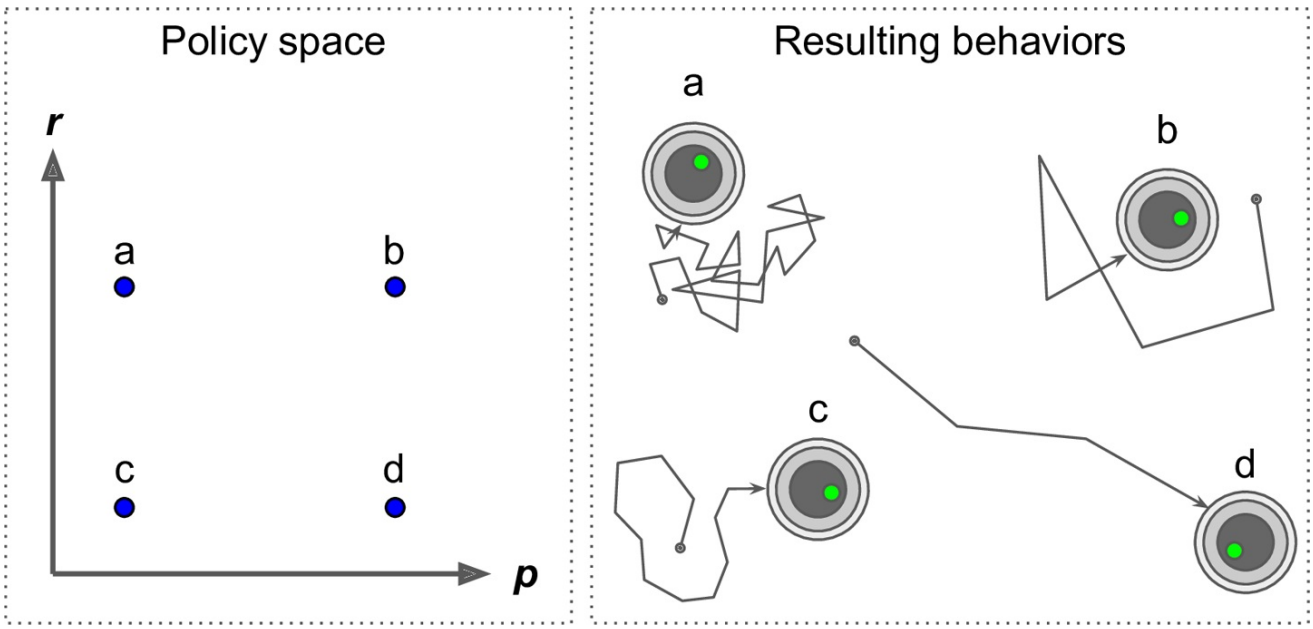

Figure 16-3. Four points in policy space and the agent's corresponding behavior

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 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 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.

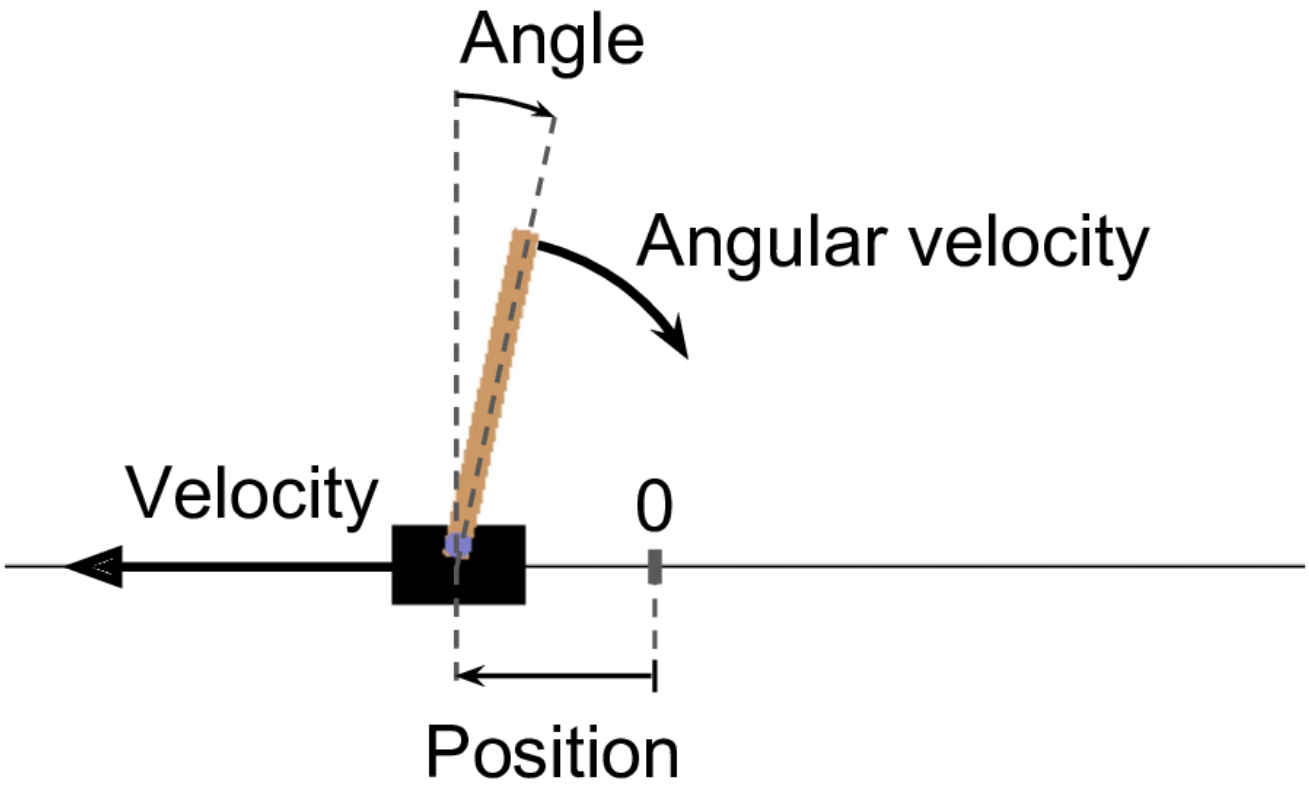

Figure 16-4. The CartPole environment

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)
```

Tip

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:

```
                # accelerate
>>> action = 1  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*. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, and action 1 with 30% probability.

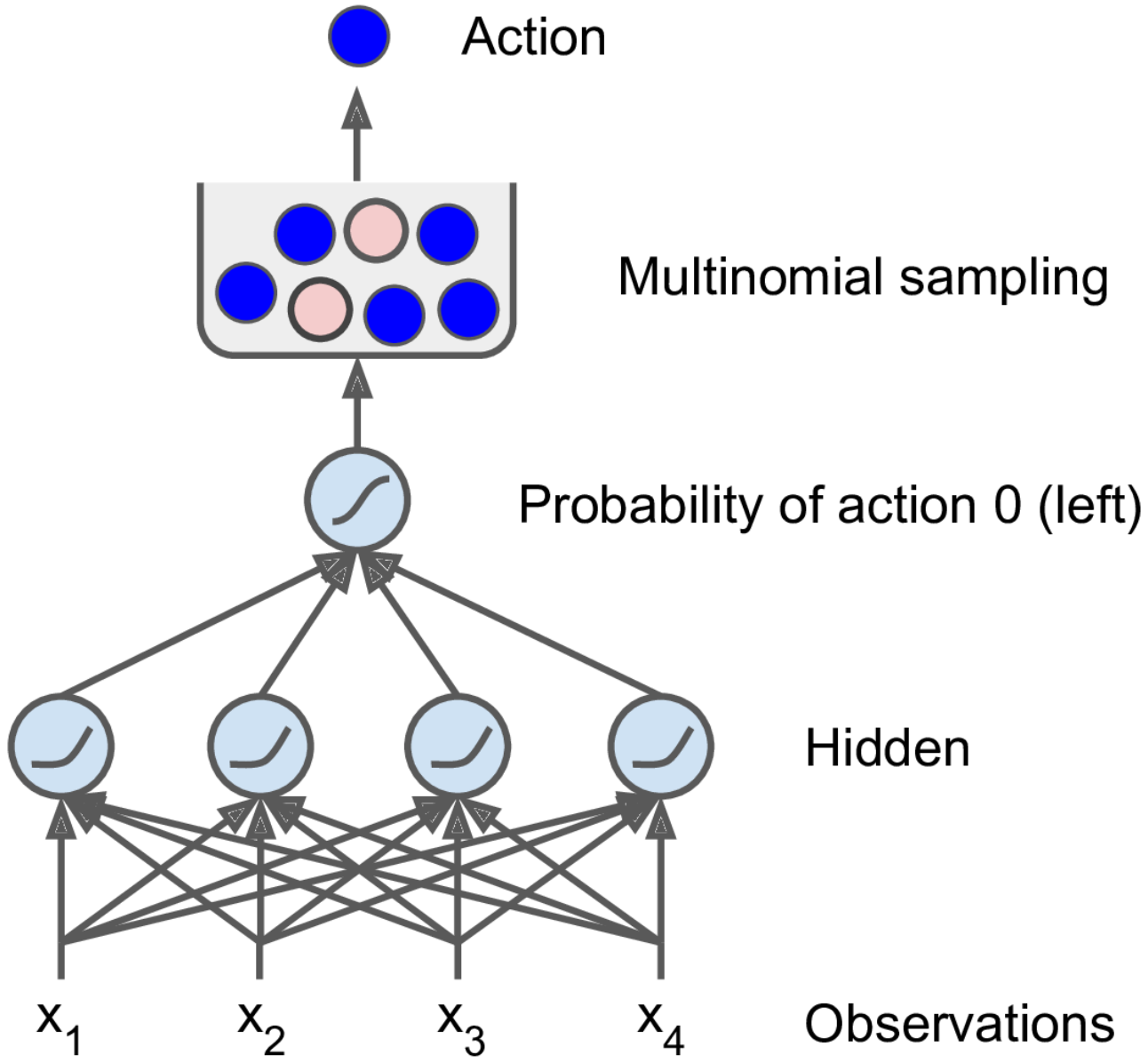

Figure 16-5. Neural network policy

You may wonder why we are picking a random action based on the probability given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing so you randomly pick one. If it turns out to be good, you can increase the probability to order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you may need to consider past actions and observations as well. 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 when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free and they contain the environment's full state.

Here is the code to build this neural network policy using TensorFlow:

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

# 1. Specify the neural network
architecture
              # ==
n_inputs = 4  env.observation_space.shape[0]
n_hidden = 4
# it's a simple task, we don't need more hidden
neurons
              # only outputs the probability of accelerating
n_outputs = 1 left
initializer = tf.contrib.layers.variance_scaling_initializer()

# 2. Build the neural
network
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,
                         weights_initializer=initializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,
                         weights_initializer=initializer)
outputs = tf.nn.sigmoid(logits)

# 3. Select a random action based on the estimated
probabilities
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

init = tf.global_variables_initializer()
```

Let's go through this code:

1. After the imports, we define the neural network architecture. The number of inputs is the size of the observation space (which in the case of the CartPole is four), we just have four hidden units and no need for more, and we have just one output probability (the probability of going left).

2. Next we build the neural network. In this example, it's a vanilla Multi-Layer Perceptron, with a single output. Note that the output layer uses the logistic (sigmoid) activation function in order to output a probability from 0.0 to 1.0. If there were more than two possible actions, there would be one output neuron per action, and you would use the softmax activation function instead.

3. Lastly, we call the `multinomial()` function to pick a random action. This function independently samples one (or more) integers, given the log probability of each integer. For example, if you call it with the array `[np.log(0.5), np.log(0.2), np.log(0.3)]` and with `num_samples=5`, then it will output five integers, each of which will have a 50% probability of being 0, 20% of being 1, and 30% of being 2. In our case we just need one integer representing the action to take. Since the `outputs` tensor only contains the probability of going left, we must first concatenate `1-outputs` to it to have a tensor containing the probability of both left and right actions. Note that if there were more than two possible actions, the neural network would have to output one probability per action so you would not need the concatenation step.

Okay, we now have a neural network policy that will take observations and output actions. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability and the target probability. It would just be regular supervised learning. However, in Reinforcement Learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount rate r* at each step. For example (see Figure 16-6), if an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount rate $r = 0.8$, the first action will have a total score of $10 + r \times 0 + r^2 \times (-50) = -22$. If the discount rate is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount rate is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount rates are 0.95 or 0.99. With a discount rate of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount rate of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount rate of 0.95 seems reasonable.

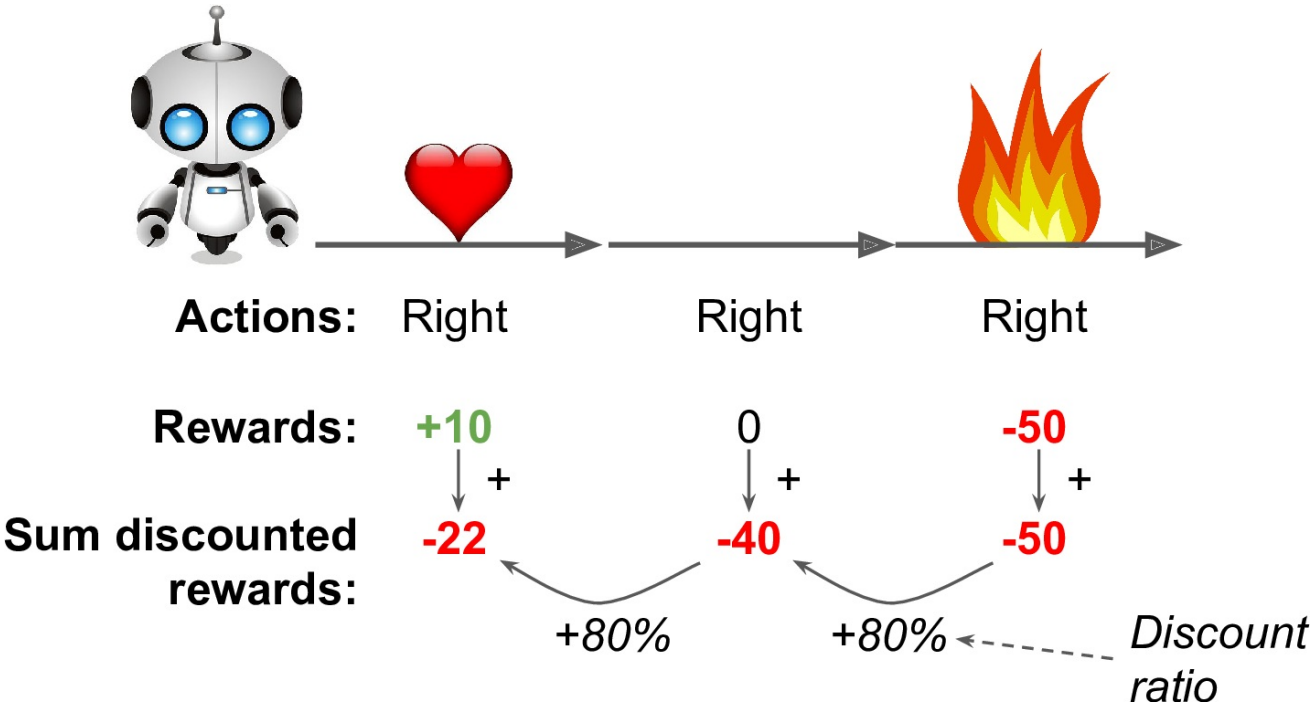

Figure 16-6. Discounted rewards

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low score (similarly, a good actor may sometimes star in a terrible movie). However, if we play the game enough times, on average good actions will get a better score than bad ones. So, to get fairly reliable action scores, we must run many episodes and normalize all the action scores (by subtracting the mean and dividing by the standard deviation). After that, we can reasonably assume that actions with a negative score were bad while actions with a positive score were good. Perfect—now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called *REINFORCE algorithms*, was introduced back in 1992 by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times and at each step compute the gradients that would make the chosen action even more likely, but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's score (using the method described in the previous paragraph).
3. If an action's score is positive, it means that the action was good and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the score is negative, it means the action was bad and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is simply to multiply each gradient vector by the corresponding action's score.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

Let's implement this algorithm using TensorFlow. We will train the neural network policy we built earlier so that it learns to balance the pole on the cart. Let's start by completing the construction phase we coded earlier to add the target probability, the cost function, and the training operation. Since we are acting as though the chosen action is the best possible action, the target probability must be 1.0 if the chosen action is action 0 (left) and 0.0 if it is action 1 (right):

```
y = 1. - tf.to_float(action)
```

Now that we have a target probability, we can define the cost function (cross entropy) and compute the gradients:

```
learning_rate = 0.01

cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
                    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

Note that we are calling the optimizer's `compute_gradients()` method instead of the `minimize()` method. This is because we want to tweak the gradients before we apply them.[10] The `compute_gradients()` method returns a list of gradient vector/variable pairs (one pair per trainable variable). Let's put all the gradients in a list, to make it more convenient to obtain their values:

```
gradients = [grad for grad, variable in grads_and_vars]
```

Okay, now comes the tricky part. During the execution phase, the algorithm will run the policy and at each step it will evaluate these gradient tensors and store their values. After a number of episodes it will tweak these gradients as explained earlier (i.e., multiply them by the action scores and normalize them) and compute the mean of the tweaked gradients. Next, it will need to feed the resulting gradients back to the optimizer so that it can perform an optimization step. This means we need one placeholder per gradient vector. Moreover, we must create the

operation that will apply the updated gradients. For this we will call the optimizer's `apply_gradients()` function, which takes a list of gradient vector/variable pairs. Instead of giving it the original gradient vectors, we will give it a list containing the updated gradients (i.e., the ones fed through the gradient placeholders):

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape
())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))

training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

Let's step back and take a look at the full construction phase:

```
n_inputs = 4
n_hidden = 4
n_outputs = 1
initializer = tf.contrib.layers.variance_scaling_initializer()

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,
                         weights_initializer=initializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,
                         weights_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

y = 1. - tf.to_float(action)
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
                    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
gradients = [grad for grad, variable in grads_and_vars]
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape
())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

On to the execution phase! We will need a couple of functions to compute the total discounted rewards, given the raw rewards, and to normalize the results across multiple episodes:

```
def discount_rewards(rewards, discount_rate):
    discounted_rewards = np.empty(len(rewards))
    cumulative_rewards = 0
    for step in reversed(range(len(rewards))):
        cumulative_rewards = rewards[step] + cumulative_rewards *
discount_rate
        discounted_rewards[step] = cumulative_rewards
    return discounted_rewards

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean)/reward_std
            for discounted_rewards in all_discounted_rewards]
```

Let's check that this works:

```
>>> discount_rewards([10, 0, -50], discount_rate=0.8)
array([-22., -40., -
50.])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_rate=0.8)
[array([-0.28435071, -0.86597718, -
1.18910299]),
 array([ 1.26665318,  1.0727777
])]
```

The call to `discount_rewards()` returns exactly what we expect (see Figure 16-6). You can verify that the function `discount_and_normalize_rewards()` does indeed return the normalized scores for each action in both episodes. Notice that the first episode was much worse than the second, so its normalized scores are all negative; all actions from the first episode would be considered bad, and conversely all actions from the second episode would be considered good.

We now have all we need to train the policy:

```
                        # number of training
n_iterations = 250      iterations
                        # max steps per
n_max_steps = 1000      episode
                        # train the policy every 10
n_games_per_update = 10 episodes
                        # save the model every 10 training
save_iterations = 10    iterations
discount_rate = 0.95

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
                            # all sequences of raw rewards for each
        all_rewards = []    episode
                            # gradients saved at each step of each
        all_gradients = []  episode
        for game in range(n_games_per_update):
                                # all raw rewards from the current
            current_rewards = []   episode
                                # all gradients from the current
            current_gradients = [] episode
            obs = env.reset()
            for step in range(n_max_steps):
                action_val, gradients_val = sess.run(
                        [action, gradients],
                                                            # one
                        feed_dict={X: obs.reshape(1, n_inputs)}) obs
                obs, reward, done, info = env.step(action_val[0][0])
                current_rewards.append(reward)
                current_gradients.append(gradients_val)
                if done:
                    break
            all_rewards.append(current_rewards)
            all_gradients.append(current_gradients)

        # At this point we have run the policy for 10 episodes, and we
        are
        # ready for a policy update using the algorithm described
        earlier.
        all_rewards = discount_and_normalize_rewards(all_rewards)
        feed_dict = {}
        for var_index, grad_placeholder in enumerate(gradient_placeholders):

# multiply the gradients by the action scores, and compute the
mean
            mean_gradients = np.mean(
                [reward * all_gradients[game_index][step][var_index]
                    for game_index, rewards in enumerate(all_rewards)
                    for step, reward in enumerate(rewards)],
                axis=0)
            feed_dict[grad_placeholder] = mean_gradients
        sess.run(training_op, feed_dict=feed_dict)
        if iteration % save_iterations == 0:
            saver.save(sess, "./my_policy_net_pg.ckpt")
```

Each training iteration starts by running the policy for 10 episodes (with maximum 1,000 steps per episode, to avoid running forever). At each step, we also compute the gradients, pretending that the chosen action was the best. After these 10 episodes have been run, we compute the action scores using the `discount_and_normalize_rewards()` function; we go through each trainable variable, across all episodes and all steps, to multiply each gradient vector by its corresponding action score; and we compute the mean of the resulting gradients. Finally, we run the training operation, feeding it these mean gradients (one per trainable variable). We also save the model every 10 training operations.

And we're done! This code will train the neural network policy, and it will successfully learn to balance the pole on the cart (you can try it out in the Jupyter notebooks). Note that there are actually two ways the agent can lose the game: either the pole can tilt too much, or the cart can go completely off the screen. With 250 training iterations, the policy learns to balance the pole quite well, but it is not yet good enough at avoiding going off the screen. A few hundred more training iterations will fix that.

Tip

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should inject as much prior knowledge as possible into the agent, as it will speed up training dramatically. For example, you could add negative rewards proportional to the distance from the center of the screen, and to the pole's angle. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

Despite its relative simplicity, this algorithm is quite powerful. You can use it to tackle much harder problems than balancing a pole on a cart. In fact, AlphaGo was based on a similar PG algorithm (plus *Monte Carlo Tree Search*, which is beyond the scope of this book).

We will now look at another popular family of algorithms. 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 sum of discounted future rewards for each state, or the expected sum of discounted future rewards for each action in each state, then uses this knowledge to decide how to act. To understand these algorithms, we must first introduce *Markov decision processes* (MDP).

Markov Decision Processes

In the early 20th century, the mathematician 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 (the system has no memory).

Figure 16-7 shows an example of a Markov chain with four states. Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back since no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever (this is a *terminal state*). Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

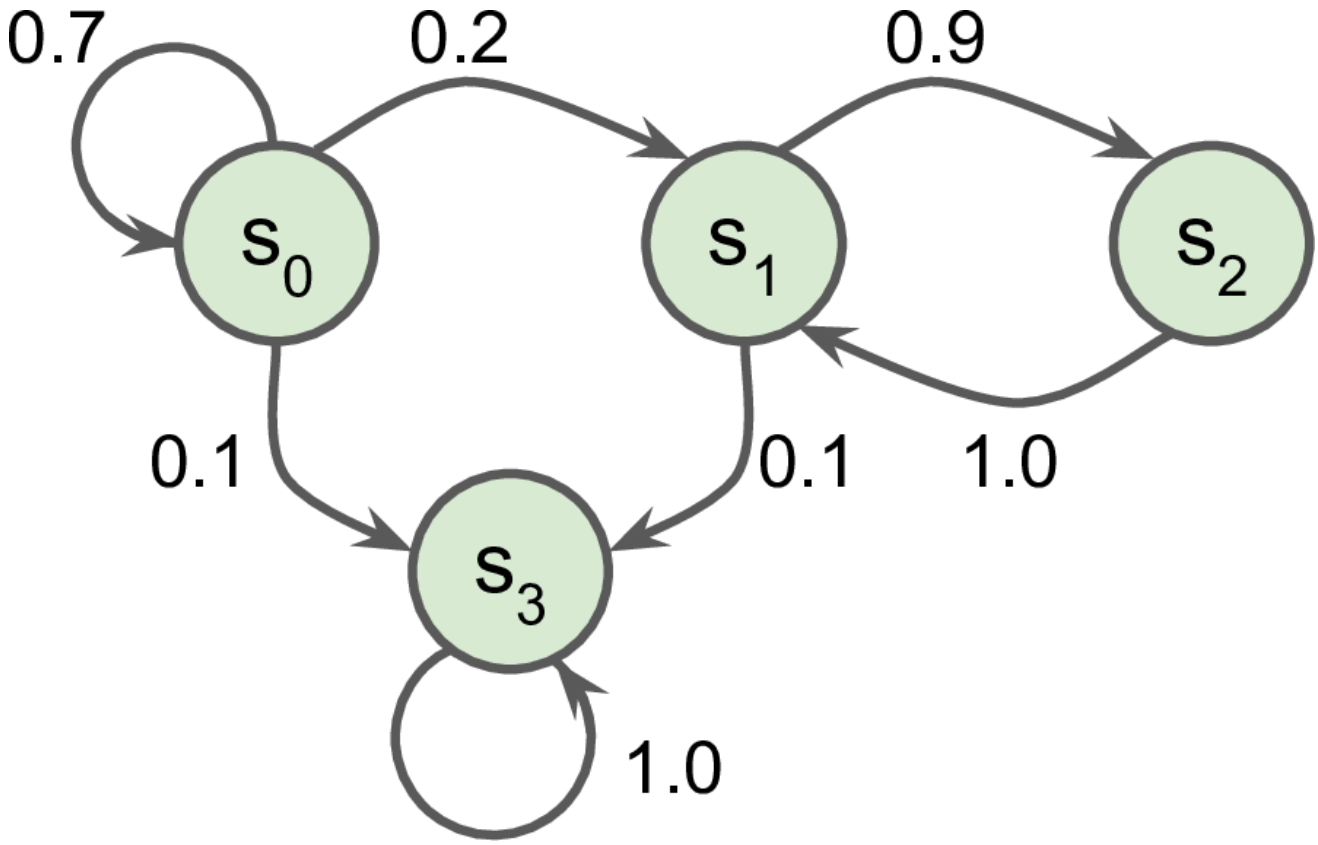

Figure 16-7. Example of a Markov chain

Markov decision processes were first described in the 1950s by Richard Bellman.[11] They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize rewards over time.

For example, the MDP represented in Figure 16-8 has three states and up to three possible discrete actions at each step. If it starts in state s_0, the agent can choose between actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10, and remaining in state s_0. It can then try again and again to gain as much reward as possible. But at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_1. It can choose to stay put by repeatedly choosing action a_1, or it can choose to move on to state s_2 and get a negative reward of –50 (ouch). In state s_3 it has no other choice than to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_3 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).

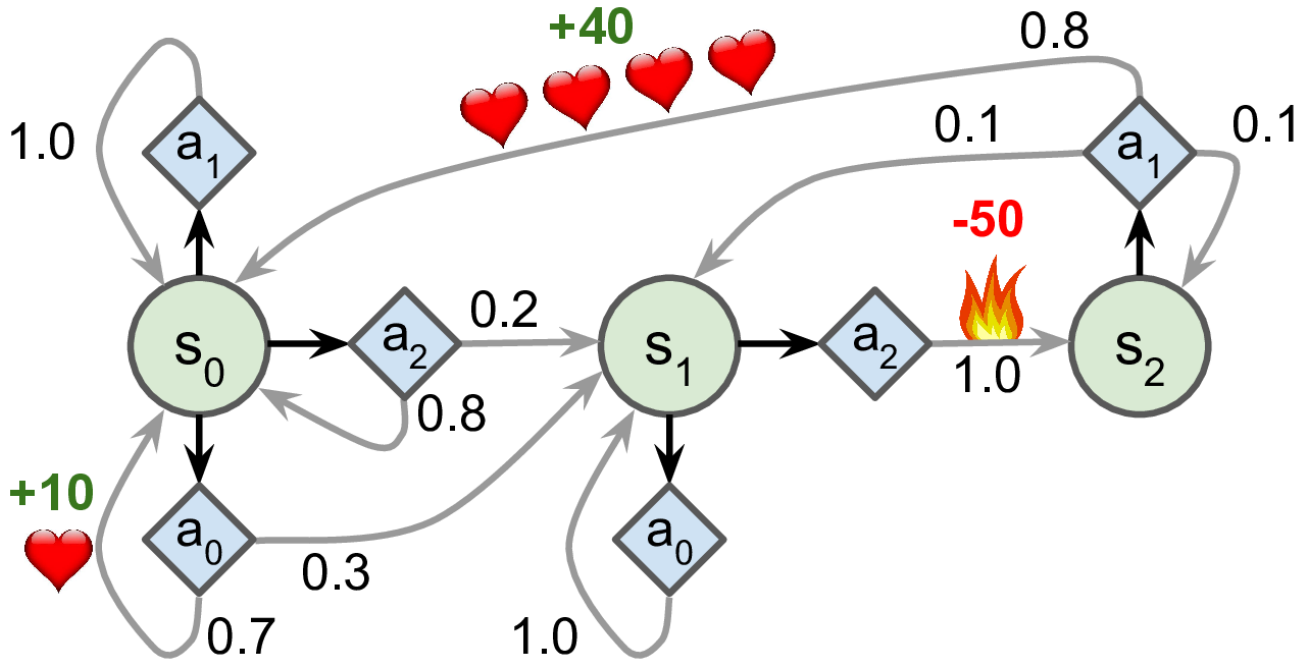

Figure 16-8. Example of a Markov decision process

Bellman found a way to estimate the *optimal state value* of any state *s*, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state *s*, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies (see Equation 16-1). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 16-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \quad \text{for all } s$$

- $T(s, a, s')$ is the transition probability from state *s* to state *s'*, given that the agent chose action *a*.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state *s* to state *s'*, given that the agent chose action *a*.
- γ is the discount rate.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero, and then you iteratively update them using the *Value Iteration* algorithm (see Equation 16-2). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 16-2. Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V_k(s')] \quad \text{for all } s$$

- $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

Note

This algorithm is an example of *Dynamic Programming*, which breaks down a complex problem (in this case estimating a potentially infinite sum of discounted future rewards) into tractable sub-problems that can be tackled iteratively (in this case finding the action that maximizes the average reward plus the discounted next state value).

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not tell the agent explicitly what to do. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values*. The optimal Q-Value of the state-action pair (*s*,*a*), noted *Q**(*s*,*a*), is the sum of discounted future rewards the agent can expect on average after it reaches the state *s* and chooses action *a*, but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see Equation 16-3).

Equation 16-3. Q-Value Iteration algorithm

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma . \max_{a'} Q_k(s', a')\right] \quad \text{for all } (s, a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state *s*, it should choose the action with the highest Q-Value for that state: .

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a)$$

Let's apply this algorithm to the MDP represented in Figure 16-8. First, we need to define the MDP:

```
            # represents impossible
nan=np.nan  actions
                # shape=[s, a,
T = np.array([  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], [nan, nan, nan], [0.0, 0.0, 1.0]],
        [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
    ])
                # shape=[s, a,
R = np.array([  s']
        [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
        [[10., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.
]],
        [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
    ])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Now let's run the Q-Value Iteration algorithm:

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

learning_rate = 0.01
discount_rate = 0.95
n_iterations = 100

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_prev[sp
]))
                for sp in range(3)
            ])
```

The resulting Q-Values look like this:

```
>>> Q
array([[ 21.89498982,  20.80024033,
16.86353093],
       [  1.11669335,         -inf,
1.17573546],
       [        -inf,  53.86946068,         -
inf]])
>>> np.argmax(Q, axis=1)
# optimal action for each
state
array([0, 2,
1])
```

This gives us the optimal policy for this MDP, when using a discount rate of 0.95: in state s_0 choose action a_0, in state s_1 choose action a_2 (go through the fire!), and in state s_2 choose action a_1 (the only possible action). Interestingly, if you reduce the discount rate to 0.9, the optimal policy changes: in state s_1 the best action becomes a_0 (stay put; don't go through the fire). It makes sense because if you value the present much more than the future, then the prospect of future rewards is not worth immediate pain.

Temporal Difference Learning and Q-Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). 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 very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general 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 (see Equation 16-4).

Equation 16-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\left(r + \gamma . V_k(s')\right)$$

- α is the learning rate (e.g., 0.01).

Tip

TD Learning has many similarities with Stochastic Gradient Descent, in particular the fact that it handles one sample at a time. Just like SGD, it can only truly converge if you gradually reduce the learning rate (otherwise it will keep bouncing around the optimum).

For each state *s*, this algorithm simply 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 (assuming it acts optimally).

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 16-5).

Equation 16-5. Q-Learning algorithm

$$Q_{k+1}(s, a) \leftarrow (1-\alpha)Q_k(s, a) + \alpha\left(r + \gamma . \max_{a'} Q_k(s', a')\right)$$

For each state-action pair (*s*, *a*), this algorithm keeps track of a running average of the rewards *r* the agent gets upon leaving the state *s* with action *a*, plus the rewards it expects to get later. Since the target policy would act optimally, we take the maximum of the Q-Value estimates for the next state.

Here is how Q-Learning can be implemented:

```
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

      # start in state
s = 0 0

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

for iteration in range(n_iterations):
                                          # choose an action
    a = rnd.choice(possible_actions[s])   (randomly)
                                          # pick next state using T[s,
    sp = rnd.choice(range(3), p=T[s, a]) a]
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = learning_rate * Q[s, a] + (1 - learning_rate) * (
            reward + discount_rate * np.max(Q[sp])
        )
           # move to next
    s = sp state
```

Given enough iterations, this algorithm will converge to the optimal Q-Values. This is called an *off-policy* algorithm because the policy being trained is not the one being executed. It is somewhat surprising that this algorithm is capable of learning the optimal policy by just watching an agent act randomly (imagine learning to play golf when your teacher is a drunken monkey). Can we do better?

Exploration Policies

Of course Q-Learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy*: at each step it acts randomly with probability ε, or greedily (choosing the action with the highest Q-Value) with probability 1-ε. The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-Value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in Equation 16-6.

Equation 16-6. Q-Learning using an exploration function

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha\left(r + \gamma . \max_{\alpha'} f(Q(s', a'), N(s', a'))\right)$$

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(q, n)$ is an *exploration function*, such as $f(q, n) = q + K/(1 + n)$, where K is a curiosity hyperparameter that measures how much the agent is attracted to to the unknown.

Approximate Q-Learning

The main problem with Q-Learning is that it does not scale well to large (or even medium) MDPs with many states and actions. Consider trying to use Q-Learning to train an agent to play Ms. Pac-Man. There are over 250 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So the number of possible states is greater than $2^{250} \approx 10^{75}$ (and that's considering the possible states only of the pellets). This is way more than atoms in the observable universe, so there's absolutely no way you can keep track of an estimate for every single Q-Value.

The solution is to find a function that approximates the Q-Values using a manageable number of parameters. This is called *Approximate Q-Learning*. For years it was recommended to use linear combinations of hand-crafted features extracted from the state (e.g., distance of the closest ghosts, their directions, and so on) to estimate Q-Values, but DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-Values is called a *deep Q-network* (DQN), and using a DQN for Approximate Q-Learning is called *Deep Q-Learning*.

In the rest of this chapter, we will use Deep Q-Learning to train an agent to play Ms. Pac-Man, much like DeepMind did in 2013. The code can easily be tweaked to learn to play the majority of Atari games quite well. It can achieve superhuman skill at most action games, but it is not so good at games with long-running storylines.

Learning to Play Ms. Pac-Man Using Deep Q-Learning

Since we will be using an Atari environment, we must first install OpenAI gym's Atari dependencies. While we're at it, we will also install dependencies for other OpenAI gym environments that you may want to play with. On macOS, assuming you have installed Homebrew, you need to run:

```
$ brew install cmake boost boost-python sdl2 swig
wget
```

On Ubuntu, type the following command (replacing `python3` with `python` if you are using Python 2):

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev libjpeg-
dev\
    xvfb libav-tools xorg-dev python3-opengl libboost-all-dev libsdl2-dev
swig
```

Then install the extra Python modules:

```
$ pip3 install --upgrade
'gym[all]'
```

If everything went well, you should be able to create a Ms. Pac-Man environment:

```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape
# [height, width,
channels]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

As you can see, there are nine discrete actions available, which correspond to the nine possible positions of the joystick (left, right, up, down, center, upper left, and so on), and the observations are simply screenshots of the Atari screen (see Figure 16-9, left), represented as 3D NumPy arrays. These images are a bit large, so we will create a small preprocessing function that will crop the image and shrink it down to 88 × 80 pixels, convert it to grayscale, and improve the contrast of Ms. Pac-Man. This will reduce the amount of computations required by the DQN, and speed up training.

```
mspacman_color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):
                             # crop and
    img = obs[1:176:2, ::2] downsize
                            # to
    img = img.mean(axis=2) greyscale
                                   # improve
    img[img==mspacman_color] = 0 contrast
    img = (img - 128) / 128 - 1
# normalize from -1. to
1.
    return img.reshape(88, 80, 1)
```

The result of preprocessing is shown in Figure 16-9 (right).

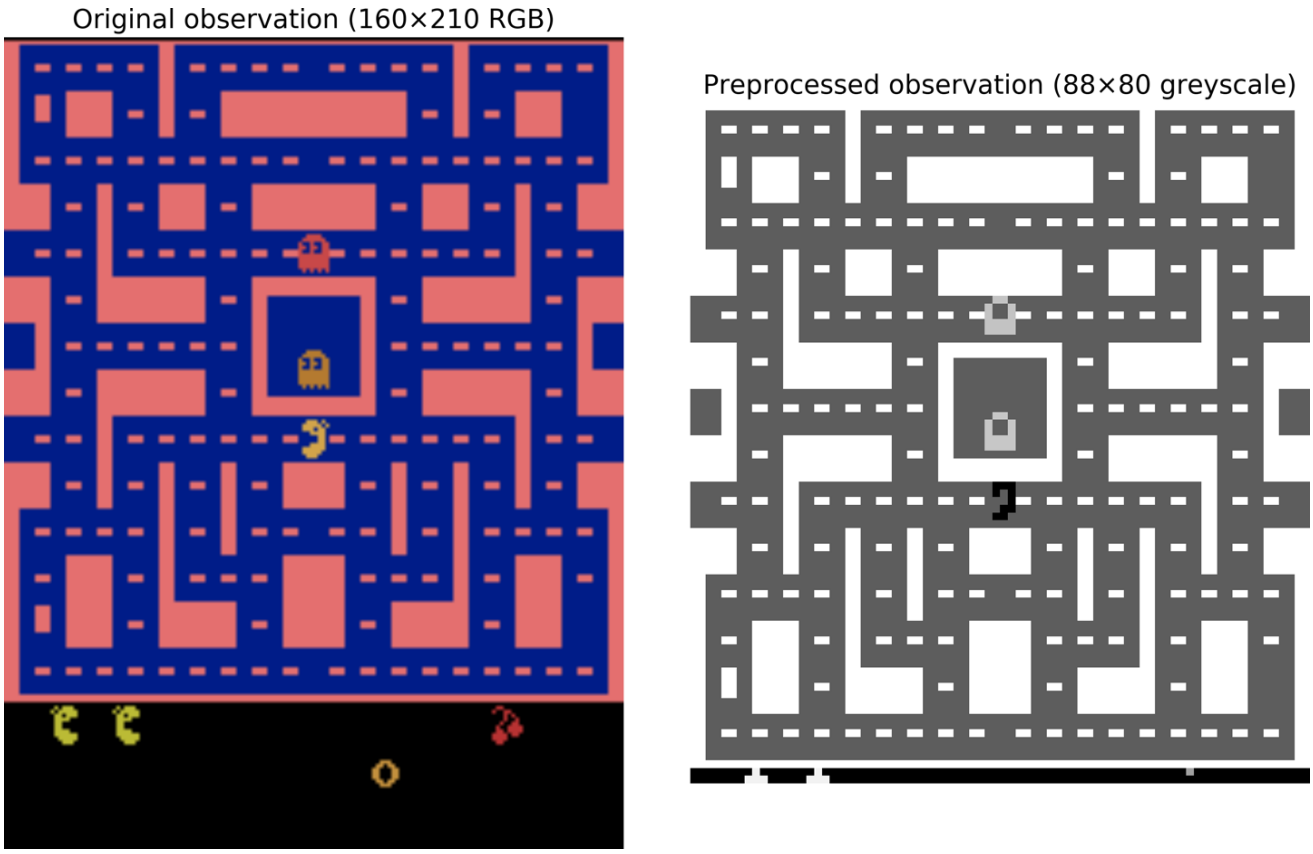

Figure 16-9. Ms. Pac-Man observation, original (left) and after preprocessing (right)

Next, let's create the DQN. It could just take a state-action pair (s,a) as input, and output an estimate of the corresponding Q-Value $Q(s,a)$, but since the actions are discrete it is more convenient to use a neural network that takes only a state s as input and outputs one Q-Value estimate per action. The DQN will be composed of three convolutional layers, followed by two fully connected layers, including the output layer (see Figure 16-10).

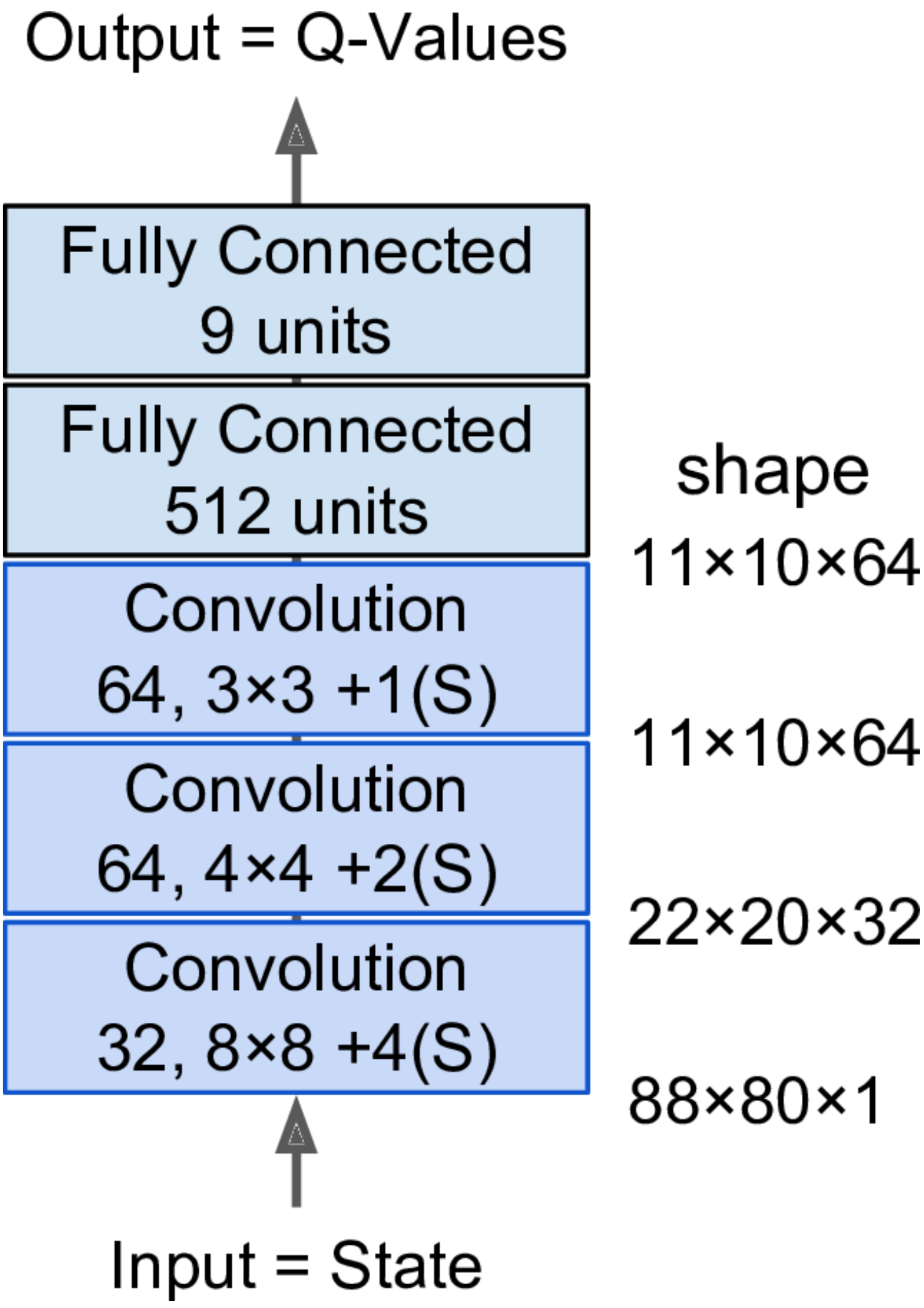

Figure 16-10. Deep Q-network to play Ms. Pac-Man

As we will see, the training algorithm we will use requires two DQNs with the same architecture (but different parameters): one will be used to drive Ms. Pac-Man during training (the *actor*), and the other will watch the actor and learn from its trials and errors (the *critic*). At regular intervals we will copy the critic to the actor. Since we need two identical DQNs, we will create a `q_network()` function to build them:

```
from tensorflow.contrib.layers import convolution2d, fully_connected

input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"]*3
conv_activation = [tf.nn.relu]*3
                            # conv3 has 64 maps of 11x10
n_hidden_in = 64 * 11 * 10  each
n_hidden = 512
hidden_activation = tf.nn.relu
                                  # 9 discrete actions are
n_outputs = env.action_space.n  available
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, scope):
    prev_layer = X_state
    conv_layers = []
    with tf.variable_scope(scope) as scope:
        for n_maps, kernel_size, stride, padding, activation in zip(
                conv_n_maps, conv_kernel_sizes, conv_strides,
                conv_paddings, conv_activation):
            prev_layer = convolution2d(
                prev_layer, num_outputs=n_maps, kernel_size=kernel_size,
                stride=stride, padding=padding, activation_fn=activation,
                weights_initializer=initializer)
            conv_layers.append(prev_layer)
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in
])
        hidden = fully_connected(
            last_conv_layer_flat, n_hidden, activation_fn=hidden_activation,
            weights_initializer=initializer)
        outputs = fully_connected(
            hidden, n_outputs, activation_fn=None,
            weights_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                       scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var
                              for var in trainable_vars}
    return outputs, trainable_vars_by_name
```

The first part of this code defines the hyperparameters of the DQN architecture. Then the `q_network()` function creates the DQN, taking the environment's state `X_state` as input, and the name of the variable scope. Note that we will just use one observation to represent the environment's state since there's almost no hidden state (except for blinking objects and the ghosts' directions).

The `trainable_vars_by_name` dictionary gathers all the trainable variables of this DQN. It will be useful in a minute when we create operations to copy the critic DQN to the actor DQN. The keys of the dictionary are the names of the variables, stripping the part of the prefix that just corresponds to the scope's name. It looks like this:

```
>>> trainable_vars_by_name
{'/Conv/biases:0': <tensorflow.python.ops.variables.Variable at
0x121cf7b50>,
 '/Conv/weights:0':
<tensorflow.python.ops.variables.Variable...>,
 '/Conv_1/biases:0':
<tensorflow.python.ops.variables.Variable...>,
 '/Conv_1/weights:0':
<tensorflow.python.ops.variables.Variable...>,
 '/Conv_2/biases:0':
<tensorflow.python.ops.variables.Variable...>,
 '/Conv_2/weights:0':
<tensorflow.python.ops.variables.Variable...>,
 '/fully_connected/biases:0':
<tensorflow.python.ops.variables.Variable...>,
 '/fully_connected/weights:0':
<tensorflow.python.ops.variables.Variable...>,
 '/fully_connected_1/biases:0':
<tensorflow.python.ops.variables.Variable...>,
 '/fully_connected_1/weights:0':
<tensorflow.python.ops.variables.Variable...>}
```

Now let's create the input placeholder, the two DQNs, and the operation to copy the critic DQN to the actor DQN:

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,
                                            input_channels])
actor_q_values, actor_vars = q_network(X_state, scope="q_networks/actor")
critic_q_values, critic_vars = q_network(X_state, scope="q_networks/critic")

copy_ops = [actor_var.assign(critic_vars[var_name])
            for var_name, actor_var in actor_vars.items()]
copy_critic_to_actor = tf.group(*copy_ops)
```

Let's step back for a second: we now have two DQNs that are both capable of taking an environment state (i.e., a preprocessed observation) as input and outputting an estimated Q-Value for each possible action in that state. Plus we have an operation called `copy_critic_to_actor` to copy all the trainable variables of the critic DQN to the actor DQN. We use TensorFlow's `tf.group()` function to group all the assignment operations into a single convenient operation.

The actor DQN can be used to play Ms. Pac-Man (initially very badly). As discussed earlier, you want it to explore the game thoroughly enough, so you generally want to combine it with an *ε-greedy policy* or another exploration strategy.

But what about the critic DQN? How will it learn to play the game? The short answer is that it will try to make its Q-Value predictions match the Q-Values estimated by the actor through its experience of the game. Specifically, we will let the actor play for a while, storing all its experiences in a *replay memory*. Each memory will be a 5-tuple (state, action, next state, reward, continue), where the "continue" item will be equal to 0.0 when the game is over, or 1.0 otherwise. Next, at regular intervals we will sample a batch of memories from the replay memory, and we will

estimate the Q-Values from these memories. Finally, we will train the critic DQN to predict these Q-Values using regular supervised learning techniques. Once every few training iterations, we will copy the critic DQN to the actor DQN. And that's it! Equation 16-7 shows the cost function used to train the critic DQN:

Equation 16-7. Deep Q-Learning cost function

$$J(\theta_{\text{critic}}) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - Q(s^{(i)}, a^{(i)}, \theta_{\text{critic}})\right)^2$$

$$\text{with } y^{(i)} = r^{(i)} + \gamma . \max_{a'} Q(s'^{(i)}, a', \theta_{\text{actor}})$$

- $s^{(i)}$, $a^{(i)}$, $r^{(i)}$ and $s'^{(i)}$ are respectively the state, action, reward, and next state of the i^{th} memory sampled from the replay memory.
- m is the size of the memory batch.
- θ_{critic} and θ_{actor} are the critic and the actor's parameters.
- $Q(s^{(i)},a^{(i)},\theta_{\text{critic}})$ is the critic DQN's prediction of the i^{th} memorized state-action's Q-Value.
- $Q(s'^{(i)}, a', \theta_{\text{actor}})$ is the actor DQN's prediction of the Q-Value it can expect from the next state $s'^{(i)}$ if it chooses action a'.
- $y^{(i)}$ is the target Q-Value for the i^{th} memory. Note that it is equal to the reward actually observed by the actor, plus the actor's *prediction* of what future rewards it should expect if it were to play optimally (as far as it knows).
- $J(\theta_{\text{critic}})$ is the cost function used to train the critic DQN. As you can see, it is just the Mean Squared Error between the target Q-Values $y^{(i)}$ as estimated by the actor DQN, and the critic DQN's predictions of these Q-Values.

Note

The replay memory is optional, but highly recommended. Without it, you would train the critic DQN using consecutive experiences that may be very correlated. This would introduce a lot of bias and slow down the training algorithm's convergence. By using a replay memory, we ensure that the memories fed to the training algorithm can be fairly uncorrelated.

Let's add the critic DQN's training operations. First, we need to be able to compute its predicted Q-Values for each state-action in the memory batch. Since the DQN outputs one Q-Value for every possible action, we need to keep only the Q-Value that corresponds to the action that was actually chosen in this memory. For this, we will convert the action to a one-hot vector (recall that this is a vector full of 0s except for a 1 at the i^{th} index), and multiply it by the Q-Values: this will zero out all Q-Values except for the one corresponding to the memorized action. Then just sum over

the first axis to obtain only the desired Q-Value prediction for each memory.

```
X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(critic_q_values * tf.one_hot(X_action, n_outputs),
                        axis=1, keep_dims=True)
```

Next let's add the training operations, assuming the target Q-Values will be fed through a placeholder. We also create a nontrainable variable called `global_step`. The optimizer's `minimize()` operation will take care of incrementing it. Plus we create the usual `init` operation and a `Saver`.

```
y = tf.placeholder(tf.float32, shape=[None, 1])
cost = tf.reduce_mean(tf.square(y - q_value))
global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(cost, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

That's it for the construction phase. Before we look at the execution phase, we will need a couple of tools. First, let's start by implementing the replay memory. We will use a `deque` list since it is very efficient at pushing items to the queue and popping them out from the end of the list when the maximum memory size is reached. We will also write a small function to randomly sample a batch of experiences from the replay memory:

```
from collections import deque

replay_memory_size = 10000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = rnd.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []]
# state, action, reward, next_state,
continue
    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3],
            cols[4].reshape(-1, 1))
```

Next, we will need the actor to explore the game. We will use the ε-greedy policy, and gradually decrease ε from 1.0 to 0.05, in 50,000 training steps:

```
eps_min = 0.05
eps_max = 1.0
eps_decay_steps = 50000

def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps
)
    if rnd.rand() < epsilon:
                                        # random
        return rnd.randint(n_outputs) action
    else:
                                    # optimal
        return np.argmax(q_values) action
```

That's it! We have all we need to start training. The execution phase does not contain anything too complex, but it is a bit long, so take a deep breath. Ready? Let's go! First, let's initialize a few variables:

```
                  # total number of training
n_steps = 100000  steps
                       # start training after 1,000 game
training_start = 1000  iterations
                       # run a training step every 3 game
training_interval = 3  iterations
                  # save the model every 50 training
save_steps = 50  steps
                  # copy the critic to the actor every 25 training
copy_steps = 25  steps
discount_rate = 0.95
skip_start = 90
# skip the start of every game (it's just waiting
time)
batch_size = 50
               # game
iteration = 0  iterations
checkpoint_path = "./my_dqn.ckpt"
            # env needs to be
done = True reset
```

Next, let's open the session and run the main training loop:

```
with tf.Session() as sess:
    if os.path.isfile(checkpoint_path):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
                  # game over, start
        if done: again
            obs = env.reset()
```

```
            obs = env.reset()
                                            # skip the start of each
            for skip in range(skip_start): game
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)

        # Actor evaluates what to
        do
        q_values = actor_q_values.eval(feed_dict={X_state: [state]})
        action = epsilon_greedy(q_values, step)

        # Actor
        plays
        obs, reward, done, info = env.step(action)
        next_state = preprocess_observation(obs)

        # Let's memorize what just
        happened
        replay_memory.append((state, action, reward, next_state, 1.0 - done
))
        state = next_state

        if iteration < training_start or iteration % training_interval != 0:
            continue

        # Critic
        learns
        X_state_val, X_action_val, rewards, X_next_state_val, continues = (
            sample_memories(batch_size))
        next_q_values = actor_q_values.eval(
            feed_dict={X_state: X_next_state_val})
        max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
        y_val = rewards + continues * discount_rate * max_next_q_values
        training_op.run(feed_dict={X_state: X_state_val,
                                   X_action: X_action_val, y: y_val})

        # Regularly copy critic to
        actor
        if step % copy_steps == 0:
            copy_critic_to_actor.run()

        # And save
        regularly
        if step % save_steps == 0:
            saver.save(sess, checkpoint_path)
```

We start by restoring the models if a checkpoint file exists, or else we just initialize the variables normally. Then the main loop starts, where `iteration` counts the total number of game steps we have gone through since the program started, and `step` counts the total number of training steps since training started (if a checkpoint is restored, the global step is restored as well). Then the code resets the game (and skips the first boring game steps, where nothing happens). Next, the actor evaluates what to do, and plays the game, and its experience is memorized in replay memory. Then, at regular intervals (after a warmup period), the critic goes through a training step. It samples a batch of memories and asks the actor to estimate the Q-Values of all actions for the next state, and it applies Equation 16-7 to compute the target Q-Value `y_val`. The only tricky part here is that we must multiply the next state's Q-Values by the `continues` vector to zero out the Q-Values corresponding to memories where the

game was over. Next we run a training operation to improve the critic's ability to predict Q-Values. Finally, at regular intervals we copy the critic to the actor, and we save the model.

Tip

Unfortunately, training is very slow: if you use your laptop for training, it will take days before Ms. Pac-Man gets any good, and if you look at the learning curve, measuring the average rewards per episode, you will notice that it is extremely noisy. At some points there may be no apparent progress for a very long time until suddenly the agent learns to survive a reasonable amount of time. As mentioned earlier, one solution is to inject as much prior knowledge as possible into the model (e.g., through preprocessing, rewards, and so on), and you can also try to bootstrap the model by first training it to imitate a basic strategy. In any case, RL still requires quite a lot of patience and tweaking, but the end result is very exciting.

Exercises

1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are possible actions? What are the rewards?
3. What is the discount rate? Can the optimal policy change if you modify the discount rate?
4. How do you measure the performance of a Reinforcement Learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay memory?
7. What is an off-policy RL algorithm?
8. Use Deep Q-Learning to tackle OpenAI gym's "BypedalWalker-v2." The Q-networks do not need to be very deep for this task.
9. Use policy gradients to train an agent to play *Pong*, the famous Atari game (`Pong-v0` in the OpenAI gym). Beware: an individual observation is insufficient to tell the direction and speed of the ball. One solution is to pass two observations at a time to the neural network policy. To reduce dimensionality and speed up training, you should definitely preprocess these images (crop, resize, and convert them to black and white), and possibly merge them into a single image (e.g., by overlaying them).
10. If you have about $100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install TensorFlow on the Pi, and go wild! For an example, check out this fun post by Lukas Biewald, or take a look at GoPiGo or BrickPi. Why not try to build a real-life cartpole by training the robot using policy gradients? Or build a robotic spider that learns to walk; give it rewards any time it gets closer to some objective (you will need sensors to measure the distance to the objective). The only limit is your imagination.

Solutions to these exercises are available in Appendix A.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, or through the *ageron/handson-ml* GitHub project.

Going forward, my best advice to you is to practice and practice: try going through all the exercises if you have not done so already, play with the Jupyter notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, meet experts. You may also want to study some topics that we did not cover in this book, including recommender systems, clustering algorithms, anomaly detection algorithms, and genetic algorithms.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

Aurélien Géron, November 26th, 2016

For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press), or David Silver's free online RL course at University College London.

"Playing Atari with Deep Reinforcement Learning," V. Mnih et al. (2013).

"Human-level control through deep reinforcement learning," V. Mnih et al. (2015).

Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and more at *https://goo.gl/yTsH6X*.

Images (a), (c), and (d) are reproduced from Wikipedia. (a) and (d) are in the public domain. (c) was created by user Stevertigo and released under Creative Commons BY-SA 2.0. (b) is a screenshot from the Ms. Pac-Man game, copyright Atari (the author believes it to be fair use in this chapter). (e) was reproduced from Pixabay, released under Creative Commons CC0.

It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the "gene pool."

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.

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).

"Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," R. Williams (1992).

[10] We already did something similar in Chapter 11 when we discussed Gradient Clipping: we first computed the gradients, then we clipped them, and finally we applied the clipped gradients.

[11] "A Markovian Decision Process," R. Bellman (1957).