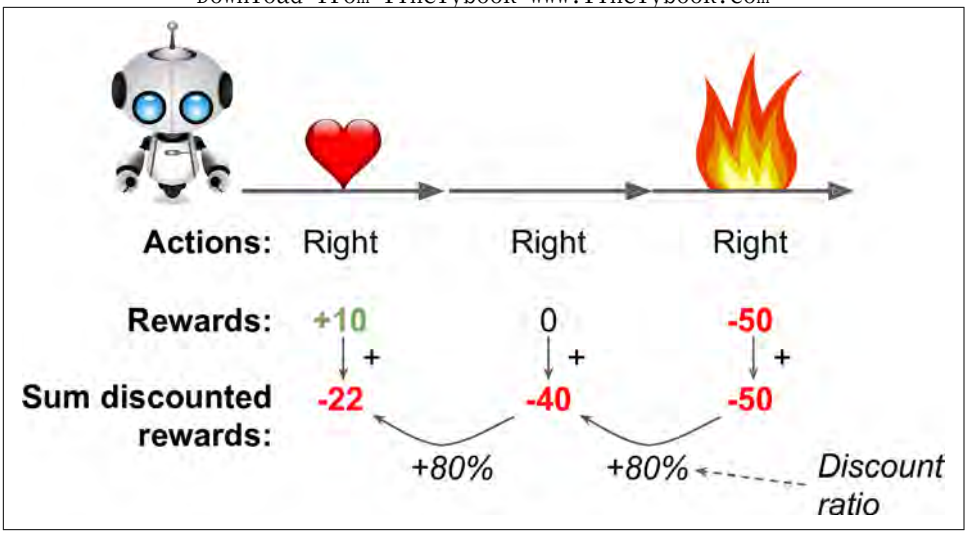

Figure 16-6. Discounted rewards

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[9] 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.

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

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

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.

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:

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

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        all_rewards = []    # all sequences of raw rewards for each episode
        all_gradients = []  # gradients saved at each step of each episode
        for game in range(n_games_per_update):
            current_rewards = []   # all raw rewards from the current episode
            current_gradients = [] # all gradients from the current episode
```

```
            obs = env.reset()
            for step in range(n_max_steps):
                action_val, gradients_val = sess.run(
                        [action, gradients],
                        feed_dict={X: obs.reshape(1, n_inputs)}) # one 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.

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.