`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.

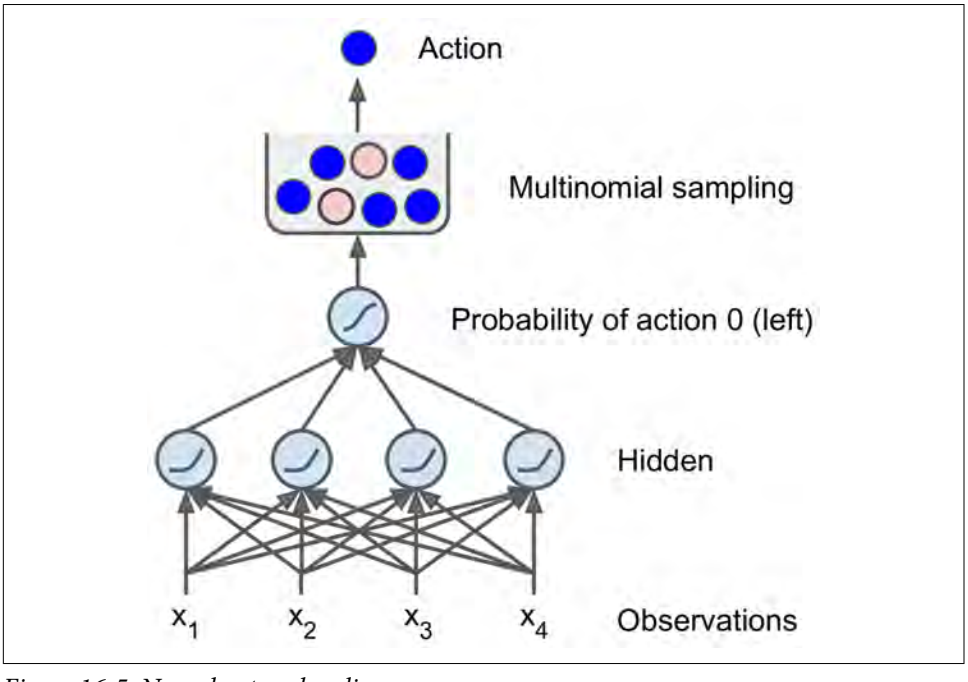

Figure 16-5. Neural network policy

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
n_outputs = 1 # only outputs the probability of accelerating 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.