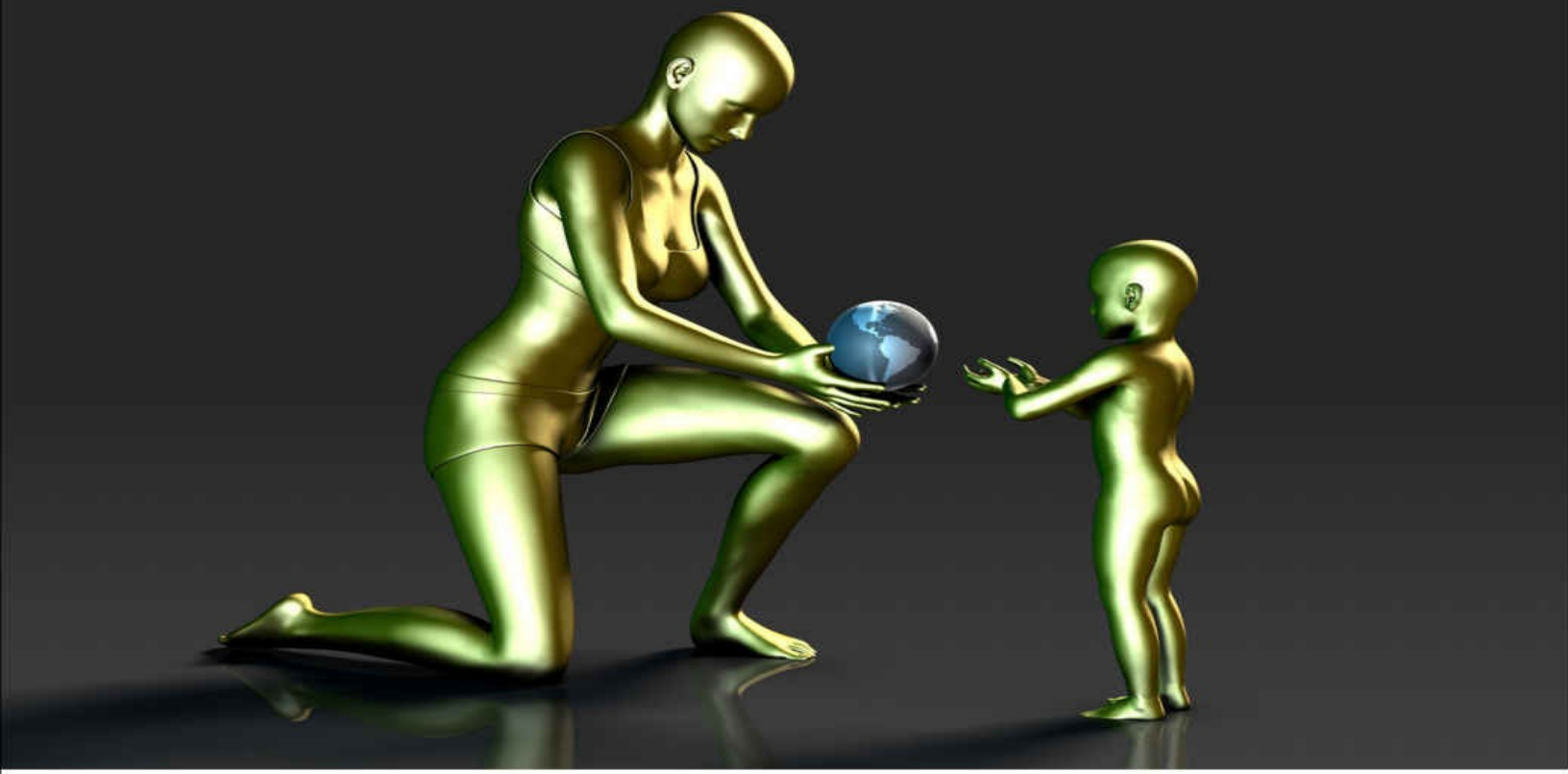
REINFORCEMENT LEARNING WITH PYTHON

MASTER REINFORCEMENT LEARNING IN PYTHON WITHOUT BEING AN EXPERT!



BOB STORY

Reinforcement Learning with Python

Master Reinforcement Learning in Python Without Being an Expert

By Bob Story

Table of Contents:

Additionally, the information in the following pages is intended only for informational purposes and should thus be thought of as universal. As befitting its nature, it is presented without assurance regarding its prolonged validity or interim quality. Trademarks that are mentioned are done without written consent and can in no way be considered an endorsement from the trademark holder.

Introduction

If you own a dog then you are probably familiar with the whole reinforcement learning concept. The two scenarios of training a machine to learn and training your dog to complete a simple 'sit' command have more in common than you think. Throw a reward their way, and you will see how motivation leads to improvement. After all, isn't playing for a prize much more fun than no prize? When the learning is reinforced, the outcome is more desirable.

Welcome to the world of reinforced learning. This is a world where self-driving cars can be seen on real roads, where programs can beat world champions, where robots are not only a part of futuristic movies. Sound too scientifically involved for your expertise? Don't worry; reinforcement learning is much more straightforward than you think.

You do not need a college degree or to be a world-class developer in order to build a reinforcement learning application. Your decent Python programming skills and a basic knowledge of Machine Learning is all it takes for this book to turn you into an RL expert.

By describing the concept of reinforcement learning in a simple, non-technical way, teaching you its elements, applications, and algorithms in the most comprehensive way possible while giving you a great jumping-off point with some amazing Python implementations, this book is a definite must-have for everyone who wants to master this popular branch of AI without drowning in the technical nonsense.
If this sounds like a good deal to you, read this book and become a Reinforcement Learning expert in a matter of days.

Understanding Reinforcement Learning

How do we learn? Sure, you can get a book, learn the theory, and consider yourself knowledgeable. That is one way to learn. But is it the best way? Think about it for a second. The major sources of our knowledge are not the things that we have learned from a book but through interaction with the environment. Whether we want to learn how to walk or learn to have a conversation, we do it with awareness of how the environment responds to our actions, and seek to influence the next actions with our behavior.

To simplify this, imagine a child that struggles to learn how to walk. Do children learn walking from someone? No. They learn while interacting with the environment.

1. First, the child notices the way you walk. They notice the way you move one leg at a time and transfer yourself from one place to another. Then, they try to replicate your movement.
2. But walking is not that simple. The child now realizes that in order to take steps with his/her legs, the child needs to be standing. So he/she begins to stagger and slip, in an attempt to get up.
3. Once the child lifts himself/herself off the ground, he/she faces the challenge to maintain a standing still position, which is a lot harder than getting up.
4. Now, the real challenge begins. The child learns that there is more to walking than meets the eye. It takes balance, and most importantly, practice.

If you let the child master walking on their own, they probably will. However, is there something you can do to speed up the process? There sure is. It is

called reinforcement learning.

Reinforcement learning means learning what steps to take and how to map different situations to actions, all in an attempt to receive a reward. There is no one to teach the one that learns what to do or how to behave, but instead, the one who learns discovers what the right actions are by being rewarded after performing the desired behavior.
In other words, if you offer your child a reward (for instance, chocolate) every time he or she takes a step, the child will be more motivated to repeat the right actions that yield the reward.

While I am not so sure that giving your child that much chocolate is considered to be good parenting, I believe that this example has served its purpose and you are more familiar with the concept of reinforcement learning now.

In reinforcement learning, the *observations* are taken by the *agent,* who then takes *actions* within the *environment*. In return for those actions, the agent receives *a reward.* If we formalize the previous example, we will see that the agent is the child, who in an attempt to manipulate the environment (the ground on which the child walks), takes actions (steps). These actions force the child to go from one state to another (with the steps he/she takes). For each submodule of the process (in this case a couple of steps), the child gets a reward (chocolate). The child will not receive the reward unless he/she completes the action first.

Through this reinforcement form of learning the agent is motivated to take the right actions in order to maximize the reward. Reinforcement learning solves closed loop problems because the actions of the learning system are what

influence the later inputs. The agent here is not told what actions to take, but has to find out which of them will bring the reward by trying these actions out.

Perhaps a simpler example is that of a dog's behavior. Imagine that you have a naughty dog. How do you train him to behave? With the help of a reward and 'punishment,' your dog will be forced and motivated to find out what the right behavior is. Every time the dog (the agent) makes a mess in your room (the environment), you reduce its doggy treats. Every time your dog behaves well, double its treats (the reward). Eventually, the dog will learn that by behaving well it gets tasty snacks, which will be the reason for it to start choosing those actions that maximize its reward.

The Elements

Between the environment and the agent, there are four main elements that are part of every reinforcement learning system.

A Policy – The policy represents the way the agent behaves at a given time. To be more precise, a policy is a mapping from the environmental states to the actions that have to be taken when the agent is in those states. The policies are the backbone of the reinforcement learning as they alone can determine the agent's behavior. Policies are often stochastic and can be as simple as a function or represent a complex process.

A Reward Signal – This represents the goal of the problem. With each step, the agent is given a reward by the environment. The main objective of the agent is to ensure maximization of the reward in the long run. The reward signal is what defines the good and bad events. To better understand this, think of the reward system as pleasure and disappointment. When we are facing a problem, the outcome of the solution is either pleasant or disappointing.

The reward signals are the defining features of the problem. The agent cannot change the function that generates the rewarding signal, even though the signal itself can be altered with different actions, In other words, the problem is unchangeable.

The basis for changing the policies is the reward signal. If the action that a certain policy selects is followed by an unsatisfactory reward, the agent then might choose to change the policy in order to select a different action that might generate a higher reward.

A Value Function – If the reward signal determines what is good at the moment, the value function indicates what is good in the long run. To be more

precise, the value of one state equals all of the rewards that the agent can expect to receive in the future, beginning with that one state. Values, however, come secondary, as there cannot be any values where there are no rewards. However, that does not mean that if a state always indicates a low reward, it has to have a low value by default. The state can be followed by other states that have high rewards which will then yield in high value as well. Obviously, this can also go the other way around.

A Model of the Environment – This is what usually mimics the environment's behavior. These models are used for planning since they can predict the next state and reward. They help us decide on a course of action by taking into considerations all of the possible outcomes before we actually experience them. The model can be for low-level and trial and error type of learning, or high-level and deliberative planning.

Reinforcement Learning Vs Other Machine Learning Types

Reinforcement learning is a type of learning that belongs to a larger class of machine learning, and it differs from the rest of the machine learning methodologies.

Although some may think of reinforcement learning as supervised up to some point, the truth is, it is very different from the supervised type of learning. *Supervised machine learning,* which is actively studied in most current machine learning research, is a type of learning methodology that comes with a set of examples, labeled, and provided by a supervisor, or in other words, a teacher. Supervised machine learning means that there is a teacher who possesses knowledge of the environment and shares the knowledge with the agent, in order for the task to be completed.

However, since there can be many different combinations of subtasks that the agent has to perform in order to complete the task, it is not uncommon for many problems to occur. Take chess for example. In a single game of chess, there are thousands of ways in which the players can move their pieces, so you can understand how playing upon a knowledge base will not be that productive. Instead, the right thing to do is to gain the knowledge through experience.

That is actually the biggest difference between supervised and reinforcement learning. In both methodologies there is a mapping between the input and the output, however, in reinforcement learning, there is a motivating reward that can act as a feedback for the agent.

In *Unsupervised learning*, on the other hand, there is no mapping between the

input and the output. Here, the goal is not to find the mapping, but the underlying patterns, or the structure that is hidden within unlabeled data. For instance, if the task is to recommend a movie to a user, an unsupervised algorithm will take a look at a list of previously watched movies and recommend some from the list. Reinforcement learning works differently. A reinforcement learning algorithm will not look at the movies that the user has previously watched, but by suggesting few movies and getting constant feedback from the user, it will gain a knowledge of which movies and movie genres the user likes the most.

Unsupervised machine learning means learning without a supervisor or a teacher, but it is different from reinforcement learning in the way that it strives to find the structure hidden within the data, whereas the reinforcement learning methodology is all about maximizing the reward.

RL Applications

Reinforcement learning is not only the most interesting field of Machine Learning, but it is also the oldest. Since it first appeared in the 1950s, the method called reinforcement learning has produced many different applications over the past decades. However, its revolution started a couple of years ago, in 2013 to be precise, when the researchers from DeepMind (an English startup) presented to the world some pretty interesting ideas. DeepMind presented a system that was able to play any Atari game without prior knowledge or experience. With the help of raw pixels as inputs only, this system became able to beat humans. We learned how revolutionizing this system is when it outperformed the world champion of the GO game. Since a program had never been close to beating a champion, this is a clear indicator of the new era that lies ahead of us.

Today, the field of Reinforcement Learning is bursting with many new ideas and has a huge range of applications.

TD-Gammon – One of the greatest applications of RL is that to the backgammon game by Gerry Tesauro. Backgammon is a very popular board game that has about 10^{20} different states. That being said, table-based RL is nearly impossible. So how did Tesauro succeed? By using the temporal difference algorithm (which we will talk about later) and a backpropagation three-layer neural network that was used as a function approximator, Tesauro managed to create his TD-Gammon program that can effortlessly play backgammon against some of the best professional players in the world.

Samuel's Checkers Players – Arthur Samuel began studying machine learning through game playing in the 50s because he considered game problems to be a

lot less complicated than those taken from real life. He chose to study checkers because of the game's simplicity and managed to create a great program for playing checkers. Even though Samuel's program was not at master's level, it was a huge step forward in the world of artificial intelligence and an extremely important achievement in machine learning.

Optimizing Memory Controllers – Ipek, Mutlu, Cartuana, and Martinez, designed an RL memory controller in 2008 and demonstrated that the new controller could speed up the process of program execution. Although this controller has not reached its potential due to the high fabrication cost, its developers have surely shown us that memory controllers that learn with reinforcement learning can significantly increase the performance. This approach is promising for the development of power-aware DRAM (Dynamic Random Access Memory) interfaces.

Manufacturing – The Japanese company Fanuc has created a very intelligent robot. This robot, with the help of the deep reinforcement learning method, practically learns a new job overnight. You may be wondering how this works. Well, let's say that the robot's task is to grasp and pick up objects. The robot tries grasping objects while taking video footage of the whole process. Each time that the robot picks up an object or fails to do so, it remembers exactly how that object looks. This knowledge is then used to refine a deep RL model or a neural network that will control the actions. By memorizing the objects the robot trains itself to do the job quickly and precisely.

There are many other RL applications, like the one in the finance sector for evaluating different trading strategies, in advertising for ranking, in medicine, e-commerce, etc.

The Simulated Environment

Having an environment to train the agent in is an absolute must in reinforcement learning. Think of the environment as the place where the agent lives. Before thinking about implementing algorithms to reinforce the learning, you have to create a working environment for your agent first.

Whomever your agent is, and whatever you want to teach the agent, you have to do it within a given environment. For instance, if you have a robot that walks, then the environment would be a real life surface, perhaps your home. However, know that this also has its limits. You cannot simply 'undo' the process if your walking robot crashes itself into a wall. Nor can you train thousands of robots in real life on your own. In the real world, training is a slow process that requires time, patience, and money. For that purpose, you will need to have a simulated environment created.

The best toolkit for developing RL algorithms that also provides a number of simulated environments (whether you want to play a game or teach a robot to walk) is OpenAIgym. This toolkit enables you to both train and compare your agents, plus you can develop new algorithms.

To install this toolkit, simply use pip:

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

Let's open the Python shell to create the environment. For this example, we want to train our model so it can balance a moving cart's pole.

```
>>> 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()
```

In this case, the make () function creates a CartPole environment which is a 2D simulation for the cart to be accelerated right and left so it can balance a pole when placed on top of the cart.

For this environment, the observations are 1D NumPy arrays that have 4 floats:

- The horizontal position (0.0 = center)
- The velocity
- The pole's angle (0.0 = vertical)
- The angular velocity

With the reset () method we can initialize the environment by returning the very first observations. The render () method displays the environment.

Now, let's ask the environment what possible actions can be taken:

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

Discrete (2) show that the possible actions are 0 and 1, representing the left (0) and right (1) accelerations. Our pole is leaning right, so we should accelerate it that way.

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

The action is executed by the step () function. There are four values returned:

Obs – is the new observation. The cart is headed toward the right (obs [1] < 0). The pole is tilted on the right (obs [2] > 0), with a negative velocity (obs [3] < 0). This means that the pole will most likely get tilted toward the left with the next step.

Reward – The reward in this environment is to receive 1.0 regardless of what you do. The whole point is to not stop running.

Done – Once the episode finished, the value done will become true. When the pole tilts more than it should, this will happen.

Info - Don't use this for training. This provides extra information.

The Policy

The algorithm that the agent uses in order to determine the actions that should be taken is called *policy*. The policy can be pretty much any algorithm. For instance, think of a robot that is a vacuum cleaner. The reward of the robot can be, let's say, the dust it can pick up in half an hour. The policy would be to clean in a forward direction with the probability *p* every second, or to clean from left to right randomly, with the probability of *1-p*. The angle of rotation is random, between –r and + r. Because there is randomness involved, this policy is stochastic.

Policy Search

However, the question remains – how to train the robot? How much dust can the robot pick up in half an hour? The only policies that you can tweak here are the angle range *r* and the probability *p*. The first thing that comes to mind here, obviously, is to try as many values for the parameters as you possibly can, and then choose the combination that results in best performance. However, when the space of the policy is pretty large, it is near impossible to find the right combination of parameters.

So, what can you do? One approach is to create 100 policies randomly which will be the first generation; check them out, and then eliminate the worst 80, making the best 20 policies produce 4 copy patterns – offsprings - (plus some variation) each. The pattern copies will then produce the second generation, and so on until you find the perfect policy. These algorithms are called *genetic algorithm* because the policies create offsprings.

Another way to find the right policy is by using optimization techniques. The best approach for this is called *policy gradient* (which we will talk about more later in this chapter). You can evaluate the reward's gradients and then tweak the parameters to follow the gradient towards maximized rewards. For instance, in the previous example with the vacuum cleaner, you can increase the probability and see if this will increase the amount of dust that the robot will gather within half an hour. If the dust amount is increased you can then choose to increase the probability even more, and if the robot has not gathered more dust, in that case you should reduce the probability *p*.

Hardcoding a Simple Policy

Now that we have learned how to create a simulated environment let's use that

knowledge, as well as the example from before, and create a code for a simple policy. Using the pole example from earlier, we will now create a policy that will accelerate toward the left when the pole is tilted to the right and the other way around.

```
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 maximum, as we don't want it to run forever
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

After 500 tries, this code shows us that the policy is unable to keep the pole upright for more than just sixty-eight steps, which is obviously not great. We need to try a different approach.

Neural Network Policies

Now, let's try creating a neural network policy and see if we will have any luck with keeping our pole upright. The neural network will calculate every action's probability, after which we will select one randomly. We will do it randomly so we can find a balance between *exploring* and *exploiting* those actions that work well. For instance, let's say that you have just entered a travel agency to pick a destination for your trip. The travel agent shows you a number of tempting options. They all look amazing ,so you pick one randomly. The trip turns out to be fantastic, and you have a great time which means that you will probably increase the probability of visiting that place again. However, you should not increase it all the way up to 100 % since that way you will never get to visit other places, some of which may provide you with an even better holiday.

Another thing that is important to keep in mind here is that the past observations and actions are completely irrelevant since every observation contains the full state of the environment.

For this code, you will need the open-source library, TensorFlow.

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

# 1. Specify the architecture of the neural network
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 an action randomly by looking at 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()
```

Great, we have now created a neural network policy. Now let's learn how to train it. In case you think that training the neural network can be easily done by reducing the entropy cross between the desired and calculated probability, know that that could be the case only if we knew which of the actions were the best. Since our only indicator is the reward we receive and it is usually delayed, it is hard to pinpoint exactly what actions are being rewarded. This is a similar concept to giving your child a chocolate 15 minutes after taking a few steps. Will he/she know that is being rewarded for something from 15 minutes earlier?

This is called credit assessment problem, and the best way to solve it is to calculate the action with the sum of the afterward rewards and by applying a discount rate *r* at every step of the way. For instance, if the agent chooses to go

right 5 times in a row and for that receives a reward +15 after the first step, 0 after the second, and -35 after the third, the 1st action can be calculated with this formula: $15 + r \times 0 + r^2 \times (-35)$. Suppose that our discount rate is 0.8, then the first action will be – 7.4. After running many episodes and after spending some time normalizing the scores, we can assume that those actions that have negative scores are bad actions, while the positive ones are good.

Policy Gradients

As we said earlier, the policy gradients algorithms follow the gradients toward maximizing rewards and with that optimize the policy's parameters. The most popular class of these algorithms is called reinforce algorithm. Here is its variant:

- Let the neural network policy run a couple of episodes. Calculate the gradients that can make the actions more likely, however, do not apply them yet.

- After running a couple of episodes, calculate the score of every action.

- If the score is positive, the action is good. If not, it is considered to be bad.

- Calculate the mean of the gradient vectors that have resulted. Perform a Gradient Descent step.

Now let's complete what we structured earlier, and add the target probability, the training operation, as well as the cost function. We will act as though every chosen action is the best one possible, and for that reason, the target probability will be 1.0 if the action is 0 (left), and 0.0 if the action is 1 (right).

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

Let's define the cost function and calculate 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)
```

To put all of the gradients in a list:

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

Now we need to take a list of gradient pairs which can be achieved by calling the optimizer's apply_gradients function.

```
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)
```

This is the 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()
```

Now we will add just a few extra functions to calculate the total rewards, as well as to normalize the results received throughout the different 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]
```

And now, all you have to do is simply train your policy:

```
n_iterations = 250      # number of training iterations
n_max_steps = 1000      # maximum steps per one episode
n_games_per_update = 10 # train the policy after every 10 episodes
```

```
save_iterations = 10    # save the model after 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 of the rewards from the current episode
            current_gradients = [] # all of the 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)
```

Now that we have run the policy for ten episodes, we are ready for an update of the policy with the help of the previously described algorithm.

```
        all_rewards = discount_and_normalize_rewards(all_rewards)
```

```
        feed_dict = {}
        for var_index, grad_placeholder in enumerate(gradient_placeholders):
            # multiply the gradients by the scores of the actions and calculate 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")
```

Finally, we are done. This code will train the policy, as well as balance the pole on the moving cart.

The PG algorithm is a direct way to optimize the policy and maximize rewards, but there are also many different algorithms that are not so direct. To understand them, we must first understand Markov Decision Processes.

Markov Decision Processes

You may be familiar with Markov chains, the stochastic processes studied by Andrey Markov. These processes have no memory, have a number of fixed states, and the process randomly goes from one state to another, where the probability is fixed and it does not depend on past or future states.

Markov decision processes, first described in the 1950s, are somewhat similar to Markov chains, except here, the agent can choose one of the possible actions, and the probability, obviously, depends on the actions chosen. Also, the state transitions of the Markov decision processes provide a reward, and the agent aims to find the most rewarding action.

Bellman, the founder of Markov decision processes, found a way to calculate the optimal state value for all states, and it is the sum of all of the discounted rewards that the agent can expect after reaching the state *s*.

$$V^*(s) = max_a \sum_s T(s, a, s')[R(s, a, s') + y.V^*(s')]$$

T(s, a, s') – The transition from one state to another after the agent chooses the action *a*

R(s, a, s') – The reward the agent receives after going from one state to another, given that he chose the action *a*.

y – The discount rate

This equation leads to an algorithm that can calculate the state value for all of the possible states. You need to initialize the state value estimates to 0, and then you update them iteratively with this algorithm:

$$\leftarrow V_{k+1}(s) \qquad \max_a \sum_s T(s, a, s')[R(s, a, s') + y.V_k(s')]$$

This can be particularly useful in evaluating policies; however, it does not tell the agent what to do. Luckily, Bellman found a similar algorithm to calculate the optimal state to action values, called *Q values*. This algorithm is also the sum of the rewards that the agent can expect to receive after performing action a and reaching state s, however, in this case that is before there is any visible outcome of the action.

Once again, you begin the process by initializing the Q Value estimates to 0, and then updating them with this algorithm:

$$\leftarrow Q_{k+1}(s,a) \qquad \sum_s T(s, a, s')[R(s, a, s') + y.\max_a Q_k(s', a')]$$

To run the Q Value algorithm in Python, use this code:

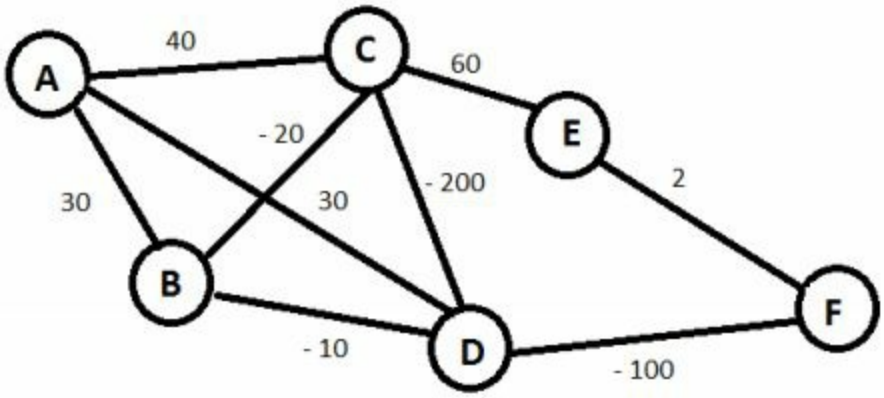
```
Q = np.full((3, 3), -np.inf)  # -info for impossible actions
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # Initial value = 0.0, for all possible actions

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)
    ])
```

Dynamic Programming

Most programs in the world of computer science are created with the same goal in mind – to optimize some value. For instance, find the shortest path between point A and point B, or find the smallest set of objects in order to satisfy certain criteria, etc. Dynamic programming is one of the best strategies for these problems. To explain an optimization problem, I will use a simple example that involves a traveling salesman.

One traveling salesman is getting ready to go on a sales tour. Starting from his hometown, in this case A, he needs to get to his destination, in this case F. There are many cities between A and F, and many different routes to take. Some of them carry costs, but on some of them, there is a way for the salesman to earn some money on his way.



The aim is to choose the most affordable route from place A to place F.

- The set of states here are the nodes (A, B, C, D, E, F).

- The action is to travel from one place to another.
- The reward function is the value that is represented by edge, which is the cost.
- The policy is the right way to complete the task, which is the right route to take.

The obvious choice here is to take the greedy approach and choose the next step to be the best possible. In this case, that is to go from A to D, and then from D to A. That way the value is -70, which means that you have no costs, but can earn some money on your way your final destination.

This type of reinforcement learning algorithm is known as *epsilon greedy,* which means taking a big problem and trying to solve it right away.

This well-known traveling salesman problem is a combinatorial optimization problem. Here is a simple Python code for solving the TSP problem using brute-force and heuristic algorithm.

```
import doctest
from itertools import permutations


def distance(point1, point2):
    """
    Returns the Euclidean distance of two points in the Cartesian Plane.

    >>> distance([3,4],[0,0])
    5.0
```

```
    >>> distance([3,6],[10,6])
    7.0
    """
    return ((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2) ** 0.5


def total_distance(points):
    """
    Returns the length of the path passing through
    all of the points in the exact given order.

    >>> total_distance([[1,2],[4,6]])
    5.0
    >>> total_distance([[3,6],[7,6],[12,6]])
    9.0
    """
        return sum([distance(point, points[index + 1]) for index, point in
enumerate(points[:-1])])


def travelling_salesman(points, start=None):
    """
    Finds the shortest route to visit all the cities by bruteforce.
    Time complexity is O(N!), so never use on long lists.

    >>> travelling_salesman([[0,0],[10,0],[6,0]])
    ([0, 0], [6, 0], [10, 0])
    >>> travelling_salesman([[0,0],[6,0],[2,3],[3,7],[0.5,9],[3,5],[9,1]])
```

```
    ([0, 0], [6, 0], [9, 1], [2, 3], [3, 5], [3, 7], [0.5, 9])
    """
    if start is None:
        start = points[0]
      return min([perm for perm in permutations(points) if perm[0] == start], key=total_distance)


def optimized_travelling_salesman(points, start=None):
    """
    As solving the problem in the brute force way is too slow,
    this function implements a simple heuristic: always
    go to the nearest city.

     This algorithm is simple, but its solution it's about 25% longer than the optimal one
    and runs fast in the O(N^2) time complexity.

        >>> optimized_travelling_salesman([[i,j] for i in range(5) for j in range(5)])
    [[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [1, 4], [1, 3], [1, 2], [1, 1], [1, 0], [2, 0], [2, 1], [2, 2], [2, 3], [2, 4], [3, 4], [3, 3], [3, 2], [3, 1], [3, 0], [4, 0], [4, 1], [4, 2], [4, 3], [4, 4]]
    >>> optimized_travelling_salesman([[0,0],[10,0],[6,0]])
    [[0, 0], [6, 0], [10, 0]]
    """
    if start is None:
        start = points[0]
```

```
    must_visit = points
    path = [start]
    must_visit.remove(start)
    while must_visit:
        nearest = min(must_visit, key=lambda x: distance(path[-1], x))
        path.append(nearest)
        must_visit.remove(nearest)
    return path


def main():
    doctest.testmod()
    points = [[0, 0], [1, 5.7], [2, 3], [3, 7],
              [0.5, 9], [3, 5], [9, 1], [10, 5]]
    print("""The minimum distance to visit all the following points: {}
starting at {} is {}.

The optimized algoritmh yields a path long {}.""".format(
        tuple(points),
        points[0],
        total_distance(travelling_salesman(points)),
        total_distance(optimized_travelling_salesman(points))))


if __name__ == "__main__":
    main()
```

**Note: The distance numbers in the example pictures are different from the numbers in this code. They only serve as an example for you to gain an understanding of how you can solve the problem.*

Monte Carlo Methods

If we needed knowledge of the environment in order to choose the best possible route to get from point A to point B in the previous example, Monte Carlo methods do not require prior knowledge but only experience. These methods require actions, sample state sequences, and rewards from interactions with the environment, whether actual or simulated. Monte Carlo methods are quite striking because the agent does not have to know the environment's dynamics but can still achieve the optimal behavior.

The most famous algorithm for is the Monte Carlo Tree Search, which is a heuristic search algorithm for decision processes, mostly involved in game playing. In fact, AlphaGo's algorithm uses this Tree Search.

The Monte Carlo Tree Search is quite dominant in the field of Go games, and the best part about this technique is that it does not require previous knowledge and can be used for general game playing.

The focus of the MCTS (Monte Carlo Tree Search) is to analyze the most promising moves and then expand the search tree upon the random sampling of the search. This application is based on many playouts and in each of these playouts, the game is played by selecting random moves. The final result is then used with the game tree's nodes.

The Monte Carlo Tree Search consists of 4 steps:

1. Selection – This phase happens and lasts only while you have the statistics to treat the positions as multi-armed bandit problems.
2. Expansion – The second phase starts when you can no longer apply the

multi-armed bandit algorithms.

3. Simulation – Simulation is the remainder of the payouts. This can either be entirely random or with simple weighing heuristics.
4. Backpropagation – The final phase is the update phase, and it happens when the playouts reach the end.

Here is how you can implement the Monte Carlo Tree Search in Python.

Let's start the initialization. The board object is where the game info is going to be stored:

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        self.board = board
        self.states = []

    def update(self, state):
        self.states.append(state)
```

Since the Tree Search algorithm relies on playing multiple games from one, current state, we will add that next:

```
import datetime

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        seconds = kwargs.get('time', 30)
        self.calculation_time = datetime.timedelta(seconds=seconds)
```

```
    # ...

    def get_play(self):
        begin = datetime.datetime.utcnow()
        while datetime.datetime.utcnow() - begin < self.calculation_time:
            self.run_simulation()
```

We need to define the run_simulation next:

```
# ...
from random import choice

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.max_moves = kwargs.get('max_moves', 100)

    # ...

    def run_simulation(self):
        states_copy = self.states[:]
        state = states_copy[-1]

        for t in xrange(self.max_moves):
            legal = self.board.legal_plays(states_copy)

            play = choice(legal)
            state = self.board.next_state(state, play)
```

```
            states_copy.append(state)

            winner = self.board.winner(states_copy)
            if winner:
                break
```

Next, we start keeping statistics about the states of the game:

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.wins = {}
        self.plays = {}

    # ...

    def run_simulation(self):
        visited_states = set()
        states_copy = self.states[:]
        state = states_copy[-1]
        player = self.board.current_player(state)

        expand = True
        for t in xrange(self.max_moves):
            legal = self.board.legal_plays(states_copy)

            play = choice(legal)
            state = self.board.next_state(state, play)
```

```
            states_copy.append(state)

            # `player` here and below refers to the player
            # who moved into that particular state.
            if expand and (player, state) not in self.plays:
                expand = False
                self.plays[(player, state)] = 0
                self.wins[(player, state)] = 0

            visited_states.add((player, state))

            player = self.board.current_player(state)
            winner = self.board.winner(states_copy)
            if winner:
                break

        for player, state in visited_states:
            if (player, state) not in self.plays:
                continue
            self.plays[(player, state)] += 1
            if player == winner:
                self.wins[(player, state)] += 1
```

By adding the wins and plays, we are now able to count every game state that is tracked. Now that we have the statistics, we can base the decisions on them:

```
from __future__ import division
# ...
```

```
class MonteCarlo(object):
    # ...

    def get_play(self):
        self.max_depth = 0
        state = self.states[-1]
        player = self.board.current_player(state)
        legal = self.board.legal_plays(self.states[:])

        # Get out early if you cannot make any real choice.
        if not legal:
            return
        if len(legal) == 1:
            return legal[0]

        games = 0
        begin = datetime.datetime.utcnow()
        while datetime.datetime.utcnow() - begin < self.calculation_time:
            self.run_simulation()
            games += 1

        moves_states = [(p, self.board.next_state(state, p)) for p in legal]

        # Display the number of the calls of `run_simulation` and the
        # elapsed time.
        print games, datetime.datetime.utcnow() - begin
```

```
        # Pick the move with the highest percentage of wins.
        percent_wins, move = max(
            (self.wins.get((player, S), 0) /
             self.plays.get((player, S), 1),
             p)
            for p, S in moves_states
        )

        # Display the stats for each possible play.
        for x in sorted(
            ((100 * self.wins.get((player, S), 0) /
              self.plays.get((player, S), 1),
              self.wins.get((player, S), 0),
              self.plays.get((player, S), 0), p)
             for p, S in moves_states),
            reverse=True
        ):
            print "{3}: {0:.2f}% ({1} / {2})".format(*x)

        print "Maximum depth searched:", self.max_depth

        return move
```

With this step, we allow three things to happen. First, we enable the get_play to return in case there are no choices left to be made, or if there is only a single one. Next, we include the output for debugging information, as well as the statistics for the moves. Finally, we include the code that will select the move that has the highest chance of winning.

There is, however, one more thing to do before we are ready to test the code in practice, and that is to implement some UCB1 (multi-armed bandit algorithms) for those positions where the plays are in the stats tables.

```
# ...
from math import log, sqrt

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.C = kwargs.get('C', 1.4)

    # ...

    def run_simulation(self):
        # We optimize here to have a local
        # variable lookup instead of an attribute access.
        plays, wins = self.plays, self.wins

        visited_states = set()
        states_copy = self.states[:]
        state = states_copy[-1]
        player = self.board.current_player(state)

        expand = True
        for t in xrange(1, self.max_moves + 1):
            legal = self.board.legal_plays(states_copy)
```

```
moves_states = [(p, self.board.next_state(state, p)) for p in legal]

if all(plays.get((player, S)) for p, S in moves_states):
    # If there are stats of the legal moves, we use them
    log_total = log(
        sum(plays[(player, S)] for p, S in moves_states))
    value, move, state = max(
        ((wins[(player, S)] / plays[(player, S)]) +
         self.C * sqrt(log_total / plays[(player, S)]), p, S)
        for p, S in moves_states
    )
else:
    # Otherwise, simply make an arbitrary decision.
    move, state = choice(moves_states)

states_copy.append(state)

# `player` here refers to the player
# who moved into that state.
if expand and (player, state) not in plays:
    expand = False
    plays[(player, state)] = 0
    wins[(player, state)] = 0
    if t > self.max_depth:
        self.max_depth = t

visited_states.add((player, state))
```

```
            player = self.board.current_player(state)
            winner = self.board.winner(states_copy)
            if winner:
                break

        for player, state in visited_states:
            if (player, state) not in plays:
                continue
            plays[(player, state)] += 1
            if player == winner:
                wins[(player, state)] += 1
```

Voila! You can now make reasonable decisions for many different board games with this magical code. Pretty cool, right?

Temporal Difference Learning

Temporal difference learning is a combination of the Monte Carlo methods and Dynamic Programming, Just like Dynamic Programming, TD learning also updates the estimates on the other learned estimates, without receiving or waiting for the outcome. Like Monte Carlo Methods, the TD learning learns directly from the environment, from experience, without a knowledge of the environment's dynamics.

What I am trying to say is that, in Temporal Difference Learning, the agent has only a partial knowledge of the Markov Decision Processes. The TD learning is a prediction-based learning method that assumes that all subsequent predictions are collated in some way.

This is the TD learning algorithm:

$$\leftarrow V_{k+1}(s) \qquad (1 - a)\, V_k(s) + a\, (r + y.\, V_k(s'))$$

Here, a is the learning rate.

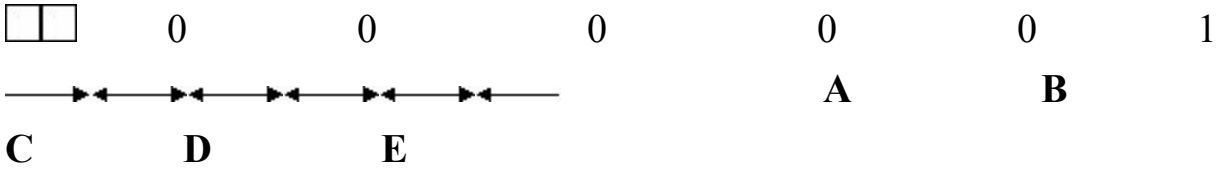
For each state s, the algorithm keeps track of the rewards that the agent leaves upon shifting to another state, as well as the rewards that the agent is supposed to receive later.

The Random Walk Policy

Random walks are famous stochastic processes that represent a path made from random steps on integers or other mathematical space. This can be

something like the prices of fluctuation stocks, the financial situation of a gambler, the molecule's traveled path in gas or liquid form, etc.

Now, to compare the prediction abilities of the TD learning, let's try out this random walk example:



In this Random Walk example, the episodes start in the center C, and proceed left or right, only by one state with each step. When the episode is on the right, a reward +1 occurs. The other rewards are 0. For instance, one episode might be C, 0, B, 0, C, 0, D, 0, E, 1. The value of the center state is 0.5. The values of all the other states, from A to E are: 1/6, 2/6, 3/6, 4/6, and 5/6.

Here is the python code for implementing the random walk example. Just note that in the code, the states are not letters from A to E, but numbers from 1 to 6.

```
#!/usr/local/bin/python

"""
The states here are as follows:
0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6

The reward for landing at the state 6 is 1.
The reward for landing at the state 0 is 0.
```

In any state, except the final ones, there are only 2 available actions: 'left' and

'right'.
Both the episodes and policy end in the final states.

Both of the actions here can be taken with 0.5 probability.
"""

```
import random

states = range(0,7)
finalStates = [0,6]
reward = [ 1 if s == 6 else 0 for s in states]

def policy(s):
  """Random walk policy: ations are 'go left' and 'go right'."""

  return random.choice(['left', 'right'])

def execute_policy(s, a):
  """Change the state according to the taken action."""

  if a == 'left':
     return s - 1
  else:
     return s + 1

def TD_0(V_star, alpha, gamma,  numOfEpisodes = 10000):
  """Use the Temporal-Difference Learning to learn V^*."""
```

```
  for episode in range(numOfEpisodes):
      # select a random beginning state
      s = random.randint(min(states)+1, max(states)-1)

      endOfEpisode = False
      while not endOfEpisode:
          if s in finalStates:
              # evaluate the value of the final state
              V_star[s] = V_star[s] + alpha*(reward[s] - V_star[s])

              #we are in the final state. End the episode
              endOfEpisode = True
              continue
          else:
              # get an action for the state from the policy
              a = policy(s)

              # execute policy => take an action
              s_prime = execute_policy(s, a)

              # evaluate the action
               V_star[s] = V_star[s] + alpha*(reward[s] + gamma*V_star[s_prime]
- V_star[s])

              s = s_prime

def V(s, d = 0):
  """Value function that is computed by dynamic programing."""
```

```
  if d > 20:
     return 0

  if s in finalStates:
     return reward[s]

  return 0.5*(V(s-1, d+1) + V(s+1, d+1))

#################################################################
##
##  Experiments
##
#################################################################

gamma = 1

print("TD(0): alpha = 0.15")

# init the value function
V_star = [0.5 for s in states]
TD_0(V_star, 0.15, gamma, 100000)
for s, V_s in enumerate(V_star):
  V_s_star = s/6.0
  print("V(%d) = %0.3f   err = %.3f" % (s, V_s, abs(V_s_star - V_s)))

print("TD(0): alpha = 0.05")
```

```
# init the value function
V_star = [0.5 for s in states]
TD_0(V_star, 0.05, gamma, 100000)
for s, V_s in enumerate(V_star):
  V_s_star = s/6.0
  print("V(%d) = %0.3f   err = %.3f" % (s, V_s, abs(V_s_star - V_s)))

print "Dynamic programing:"
for s in states:
  V_s_star = s/6.0
  V_s = V(s)
  print("V(%d) = %0.3f   err = %.3f" % (s, V_s, abs(V_s_star - V_s)))
```

Q-Learning

Q-learning is a reinforcement learning technique that is model-free and that is often used to find the best policy for selecting an action for any MDP (Markov Decision Process). To simplify, the basics of this learning technique are that there is a representation of the states s of the environment and the possible actions *a* within the environment so that you can learn the value for each action in each of the states. For the sake of simplicity, it is best to keep the state-action values to 0, and then start exploring the state-action space.

Once you try an action, you then evaluate its state. If that action has led to an unwanted outcome, the smart thing to do is to reduce the Q value, so that the values of the other actions can be greater than the one that brings you an undesirable outcome. And the other way around. If an action brings you a reward, you would want to increase its value Q, so you can choose the action again once you find yourself in the same state again.

Let's imagine a simple cat and mouse chase. Suppose you are the mouse and the cat is in front of you. If you choose to move one step forward the cat, you will end up being eaten by the cat. That will have an undesirable outcome, so next time, you will choose a different move and choose to go away to the side of the cat. This may not reduce the value of you moving forward when the cat isn't in front of you. It's a simple example but it serves the purpose well. The point of Q learning is to choose the actions which will maximize the reward and lower the value of those that bring you unsatisfactory results.

The Q learning algorithm is:

$\leftarrow Q_{k+1}(s, a)$ $\quad (1 - a)Q_k(s, a) + a(r + y.\max_a Q_k(s', a'))$

You can easily implement the Q learning algorithm in Python with this simple code:

```
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0 # start in state 0

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

for iteration in range(n_iterations):
    a = rnd.choice(possible_actions[s])  # choose an action (randomly)
    sp = rnd.choice(range(3), p=T[s, a]) # pick the next state using T[s, 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])
        )
    s = sp # move to next state
```

Conclusion

Before reading this book, you may have thought Reinforcement Learning with Python is something meant for the top-notch developers only. Now that you know different algorithms of this popular machine learning methodology, the next step is to train your algorithms to exhaustion.

Who knows, you may even be on the verge of creating the new AlphaGo!