How computers can learn better

June 3 2013, by Larry Hardesty

Reinforcement learning is a technique, common in computer science, in which a computer system learns how best to solve some problem through trial-and-error. Classic applications of reinforcement learning involve problems as diverse as robot navigation, network administration and automated surveillance.

At the Association for Uncertainty in Artificial Intelligence's annual conference this summer, researchers from MIT's Laboratory for Information and Decision Systems (LIDS) and Computer Science and Artificial Intelligence Laboratory will present a new reinforcement-learning algorithm that, for a wide range of problems, allows computer systems to find solutions much more efficiently than previous algorithms did.

The paper also represents the first application of a new programming framework that the researchers developed, which makes it much easier to set up and run reinforcement-learning experiments. Alborz Geramifard, a LIDS postdoc and first author of the new paper, hopes that the software, dubbed RLPy (for reinforcement learning and Python, the programming language it uses), will allow researchers to more efficiently test new algorithms and compare algorithms' performance on different tasks. It could also be a useful tool for teaching computer-science students about the principles of reinforcement learning.

Geramifard developed RLPy with Robert Klein, a master's student in MIT's Department of Aeronautics and Astronautics. RLPy and its source code were both released online in April.

Every reinforcement-learning experiment involves what's called an agent, which in artificial-intelligence research is often a computer system being trained to perform some task. The agent might be a robot learning to navigate its environment, or a software agent learning how to automatically manage a computer network. The agent has reliable information about the current state of some system: The robot might know where it is in a room, while the network administrator might know which computers in the network are operational and which have shut down. But there's some information the agent is missing—what obstacles the room contains, for instance, or how computational tasks are divided up among the computers.

Finally, the experiment involves a "reward function," a quantitative measure of the progress the agent is making on its task. That measure could be positive or negative: The network administrator, for instance, could be rewarded for every failed computer it gets up and running but penalized for every computer that goes down.

The goal of the experiment is for the agent to learn a set of policies that will maximize its reward, given any state of the system. Part of that process is to evaluate each new policy over as many states as possible. But exhaustively canvassing all of the system's states could be prohibitively time-consuming.

Consider, for instance, the network-administration problem. Suppose that the administrator has observed that in several cases, rebooting just a few computers restored the whole network. Is that a generally applicable solution?

One way to answer that question would be to evaluate every possible failure state of the network. But even for a network of only 20 machines, each of which has only two possible states—working or not—that would mean canvassing a million possibilities.

Faced with such a combinatorial explosion, a standard approach in reinforcement learning is to try to identify a set of system "features" that approximate a much larger number of states. For instance, it might turn out that when computers 12 and 17 are down, it rarely matters how many other computers have failed: A particular reboot policy will almost always work. The failure of 12 and 17 thus stands in for the failure of 12, 17 and 1; of 12, 17, 1 and 2; of 12, 17 and 2, and so on.

Geramifard—along with Jonathan How, the Richard Cockburn Maclaurin Professor of Aeronautics and Astronautics, Thomas Walsh, a postdoc in How's lab, and Nicholas Roy, an associate professor of aeronautics and astronautics—developed a new technique for identifying pertinent features in reinforcement-learning tasks. The algorithm first builds a data structure known as a tree—kind of like a family-tree diagram—that represents different combinations of features. In the case of the network problem, the top layer of the tree would be individual machines, the next layer would be combinations of two machines, the third layer would be combinations of three machines, and so on.

The algorithm then begins investigating the tree, determining which combinations of features dictate a policy's success or failure. The relatively simple key to its efficiency is that when it notices that certain combinations consistently yield the same outcome, it stops exploring them. For instance, if it notices that same policy seems to work whenever machines 12 and 17 have failed, it stops considering combinations that include 12 and 17 and begins looking for others.

Geramifard believes that this approach captures something about how human beings learn to perform new tasks. "If you teach a small child what a horse is, at first it might think that everything with four legs is a horse," he says. "But when you show it a cow, it learns to look for a different feature—say, horns." In the same way, Geramifard explains, the new algorithm identifies an initial feature on which to base judgments and then looks for complementary features that can refine the initial judgment.

RLPy allowed the researchers to quickly test their new algorithm against a number of others. "Think of it as like a Lego set," Geramifard says. "You can snap one module out and snap another one in its place."

In particular, RLPy comes with a number of standard modules that represent different machine-learning algorithms; different problems (such as the network-administration problem, some standard control-theory problems that involve balancing pendulums, and some standard surveillance problems); different techniques for modeling the computer system's environment; and different types of agents.

It also allows anyone familiar with the Python programming language to build new modules. They just have to be able to hook up with existing modules in prescribed ways.

Geramifard and his colleagues found that in computer simulations, their new algorithm evaluated policies more efficiently than its predecessors, arriving at more reliable predictions in one-fifth the time.

RLPy can be used to set up experiments that involve computer simulations, such as those that the MIT researchers evaluated, but it can also be used to set up experiments that collect data from real-world interactions. In one ongoing project, for instance, Geramifard and his colleagues plan to use RLPy to run an experiment involving an autonomous vehicle learning to navigate its environment. In the project's initial stages, however, he's using simulations to begin building a battery of reasonably good policies. "While it's learning, you don't want to run it into a wall and wreck your equipment," he says.

More information: Paper: "Batch-iFDD for Representation Expansion in Large MDPs" (PDF)

This story is republished courtesy of MIT News (web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology