Deep Reinforcement Learning for Robotics

Michael Ganger

Computer Science Honors Project Proposal

Introduction

Although it is not a new field, machine learning has seen recent popularity in many different applications. This interest is partially due to increased computational capability, which has enabled the execution of complex algorithms on large amounts of data in order that would otherwise be prohibitively computationally expensive. One particular sub-field of machine learning of interest is known as *reinforcement learning*. As implied in the name, reinforcement learning uses a positive or negative reinforcement signal, known as the *reward*, to provide feedback to the learning system so that it may improve. The techniques developed in this discipline can be applied to many scenarios, from control systems to economics [1] to artificial intelligence in games. While other types of machine learning are applicable in similar situations, the advantage of reinforcement learning is that, at its core, it is only concerned with the reward signal, which provides substantial control over the result of learning.

Reinforcement Learning

Reinforcement learning may be applied to many fields, but here we will use a guiding example from the field of robotics. Imagine we are designing a janitorial robot. This robot will be tasked with cleaning a building, but it will not be given a map to this building, nor an explicit task to accomplish. The only feedback the robot will receive from its environment, the building, is information from its sensors and a signal that effectively indicates how good of a job it is doing. The goal we give to the robot is to maximize this signal, which is called the *reward*; this ensures that it is always making decisions that will keep the building as clean as possible. We can state this as a question:

> Given the current state of the robot, what action should it take presently to maximize the reward signal, now and in the future?

We may define the *environment* as everything that the robot cannot control, which includes the layout of the building and the charge of its battery. The *state* of the robot represents information about the environment, and may include its location in the building, the current charge of its battery, and all other information relevant to cleaning the building. We can define the *agent* as the part of the robot that decides the next *action* to take, where the action may be sweeping the floor, charging the battery, or any other possible duties of a janitorial robot. Furthermore, we may define an *observation* as the information collected by the robot's sensors.

A generalization of the question above is called the reinforcement problem [2], which guides most of reinforcement learning. The problem may be stated as such:

> For an agent in a certain state at a certain time, what action should be taken that will maximize the overall reward, now and in the future?

In essence, the goal of reinforcement learning is to find a *policy* which maximizes the *reward*. In general, a *policy* (generally denoted by π), is a probability distribution that describes the probability that an agent will take a certain action, given that it is in a certain state. The *optimal policy* is the policy that, on average, leads to the

greatest total rewards. This can be very useful in many situations; by only defining rewards, we do not need to fully understand the environment. We need not assume anything about it *a priori*. In a sense, by defining the rewards, we implicitly define a policy which maximizes them. All methods of reinforcement learning attempt to find this policy, or something that approximates it.

Value Function and Action-Value Function. One technique used to find the optimal policy is to estimate the value of the agent being in each state. In other words, how much reward can an agent expect to obtain if it is in a certain state? Once this is known, ideally for all states in an environment, the best possible action the agent can take is the one that takes the agent to the state with the highest value. One technique of learning this function on its own is called value iteration [2]. The expected total reward is known as the *return*.

The value function, which gives the expected return the agent will receive given its current state, is described by

$$V(s) = E\{R_t \mid s_t = s\},$$

where E is expected value operator, R_t is the total return that an agent receives given state s_t at time t, and s is the current state that agent is in. Another solution to this problem is described by Q-learning [3], which extends from the ideas of the value function. We use a different function to determine the policy; this is called the action-value function, which is given by

$$Q(s, a) = E\{r_{t+1} + \gamma V(s_{t+1}) \mid s_t = s, a_t = a\},$$

where r_{t+1} is the reward from moving into state s_{t+1} from s_t by taking action a [2]. However, instead of determining the value of being in a certain state, it determines the value of the agent taking a given action while being in a given state. There are a number of advantages to this technique, an important one being that it lends itself nicely to *online learning*. Online learning is the situation where the agent uses its current estimate of the optimal policy while it explores the environment and tries to learn a better estimate of the optimal policy. This is in contrast to *offline learning*, which is where the agent uses a different policy than its current estimate of the optimal policy such as a random policy.

Q-learning is an iterative algorithm, which means that at each time step it progressively makes a better estimation of the action-value, and eventually converges on the true value. The update formula used to improve this estimate is given by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

[4], where $Q(s_t, a_t)$ is the *action value*, s_{t+1} is the state moved to out of s_t by taking action a_t, r_t is the reward from this transition, α is the learning rate, and γ is the discount rate which determines the significance of future rewards.

Using the action-value function instead of the value function generally simplifies the learning process. Often, is easier to learn the action-value function and use that to determine the optimal policy. Additionally, it simplifies the evaluation of a policy over using the value function. In order to determine the next action we need to take, all we need to do is find the action that maximizes the action-value function (i.e. $a = \text{argmax}_{a'} Q(s, a')$).

Q-learning is an offline algorithm; in other words, the policy that the agent is learning is not the one that it is currently using to navigate its environment. However, there are other algorithms that can be used to estimate the Q function in an online fashion, one of the simpler ones being *SARSA* (State-Action-Reward-State-Action). SARSA is a slight variation on the Q-learning algorithm in that it uses the learned policy to take actions in the environment while it is learning.

Other Techniques. One of the more recent developments in regard to Q-learning is called *Double Q-learning* [5]. This technique requires two independent Q functions, which may be represented by matrices. The difference between traditional Q-learning and Double Q-learning is in how the tables are updated; in all other respects they are the same at the most basic level. Using deep learning algorithms (which are discussed below), significant improvement over single Q-learning has been seen on classic Atari games [6].

A different class of techniques from value-based methods are known as *policy gradient methods* [7]. These methods circumvent the difficulty of estimating the state-value and action-value functions in the presence of incomplete state information by directly estimating the policy $\pi(s \mid a)$, which is a gives the probability of taking an action given a state. A similar type of reinforcement learning, *policy search*, has also been shown to be effective at learning robot behaviors [8].

Deep Reinforcement Learning

Markov Decision Processes. Algorithms which directly estimate the value and action-value functions assume that the information about the environment provided in the state s is a *Markov Decision Process* (MDP). A MDP is a process where the state representation s is sufficient for determining the next state; there is no hidden information. This may be easily illustrated by the path of a projectile. If s_t only contains the position of the object, one cannot predict s_{t+1}. However, if s_t also contains the objects velocity, we can use this information (along with our prior knowledge of gravity) to predict the object's next state with reasonable precision.

A serious difficulty in reinforcement learning is the incompleteness of information contained in the state, s. An environment where the full state is not fully represented is called a *Partially Observable Markov Decision Processes* (POMDP). In the example above, we have a Partially Observable MDP when we may only observe the projectile's position. In order to predict the next state, we need to at least know a previous state, and the more states we have in our memory the better we can estimate s.

Neural Networks. Often, the action-value function $Q(s, a)$ is represented as a table, where each state-action pair s, a maps to a certain value in the table. However, it is often the case that all the possible states and actions span a large space; they may even be approximately continuous. In these cases, conventional Q-learning becomes impractical. There are too many possible combinations of states and actions for their values to converge in a reasonable amount of time, and there is a high memory expense to store such a large table. In order to resolve this issue, we turn to function approximation. If we can find a function $f(s, a)$ which approximates $Q(s, a)$, we can modify our learning algorithm to learn the parameters of this function instead instead of directly learning the values.

In recent years, computer hardware advances have made practical networks with many layers. These networks are termed *deep*, and they can often have better performance than shallow networks with only a few layers. Recent interest has been centered around using deep neural networks in conjunction with Q-learning. Deep Q-networks have shown better performance playing classic Atari games than other methods of reinforcement learning [9]. They have also been found to be comparable to conventional Q-learning when run on a simplified version of the game Pacman, using an extension of a linear function approximation for Q [10]. Additionally, research has shown that deep Q-networks are effective for learning POMDPs [11].

In many situations, our observations of the environment represent a Partially Observable MDP. This problem comes up frequently in many fields. In robotic engineering, for example, the state is largely determined by the robot's sensors, and how accurate they are. The quality and quantity of sensors is often determined by cost constraints, but sometimes a sensor to measure a particular quantity may not even exist, and the missing information must be inferred. Another example can be found in natural language processing, where a large portion of the information is hidden and obscured by many layers of text. An article may only mention its subject once, but reference it multiple times, and a reinforcement learning agent must be able to recognize the first instance of the subject in order to usefully interpret the article.

There are multiple ways to solve the problem of hidden state. The simplest is to incorporate past observations into the current representation of state. However, this can often be inefficient if only a small amount of information needs to be retained between states. Another way to solve this is use neural networks that have "memory." The

class of neural networks with this property is that of *Recurrent Neural Networks* (RNN). As in the name, the main idea of a RNN is that its outputs feed back into its inputs. Because of this feature, they have a type of short term memory which may be used to supplement the incomplete state representation. Training a RNN only requires a small modification to the algorithm for training conventional neural networks [12], although training them correctly can often be difficult. Because of this, different techniques have been developed improve the training process. One such method is to initialize the network will full recurrence, and then learn what information does not need to be remembered [13]. With proper training, RNN have shown themselves effective at learning time-dependent patterns.

There are different types of sub-architectures that RNNs can take on, all which have a recurrent nature suitable for learning time dependencies. One type of RNN is called a *Clockwork RNN* [14]. This network architecture is similar to the standard RNN, except is composed of multiple *modules* that have different clock speeds. This allows the modules to learn parts of the solution that have different temporal dependencies. A further extension to RNNs is called *Long Short-Term Memory* [15]. Essentially, it defines an architecture for a *memory cell* which stores information for longer periods of time without the feedback issues experienced in conventional RNNs.

One class of functions that have shown to be effective for this purpose are *neural networks*. A neural network is a type of function approximation that is a simplified model of biological neurons, grouped into *layers*. Neural networks must be trained using example input and output data and are often used in classification algorithms, but may be applied to other problems by careful selection of *activation functions*. An activation function is a function that describes the output of a neuron. Often, these functions may have a sigmoidal shape that essentially compresses the output of the neuron to be within a certain range, so as to model the activations of biological neurons.

Q-Networks. When neural networks are applied to Q-learning, the result is called a *Q-network* [9]. If we let $y_t = r_t + \gamma \max_a Q(s_{t+1}, a)$, we can rewrite the update formula for Q-learning given previously as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[y_t - Q(s_t, a_t) \right].$$

The quantity $\nabla_Q L = -[y_t - Q(s_t, a_t)]$ can be obtained by taking the gradient of the quadratic loss function L with respect to $Q(s_t, a_t)$:

$$L \equiv \frac{1}{2} \left[y_t - Q(s_t, a_t) \right]^2.$$

In this respect, the update formula adjusts $Q(s, a)$ so as to minimize the value of L. This is called *gradient descent*. We can represent a Q-network by the function $Q(s, a; \theta)$, where θ represents the parameters of the neural network being used to estimate the true action value. Using this in our loss function, we obtain the gradient with respect to the parameters θ,

$$\nabla_\theta L = - \left[y_t - Q(s_t, a_t; \theta) \right] \nabla_\theta Q(s_t, a_t; \theta).$$

The distinct feature of this gradient is the presence of $\nabla_\theta Q(s_t, a_t; \theta)$, which will change depending on the structure of the neural network used to approximate $Q(s_t, a_t)$. In the same manner as before, we may use gradient descent ot learn our network parameters,

$$\theta \leftarrow \theta + \alpha \left[r + \gamma \max_a Q(s_t, a; \theta) - Q(s_t, a_t; \theta) \right] \nabla_\theta Q(s_t, a_t; \theta).$$

If we return to our janitorial robot from before, the utility of such networks becomes apparent. Imagine that the robot is tasked with cleaning a bathroom, and that there is a sign at the beginning of a hallway that indicates which direction to turn at the end of the hallway to find the bathroom. The robot must not only recognize the sign, but also remember this value several states later. Although this may seem like a simple problem, similar situations arise frequently in engineering applications where the action taken must depend on information observed in the past.

Deep Q-Networks. Further extending the use of RNNs with memory in reinforcement learning is the addition of a deep architecture. This potentially combines the advantages of deep learning applied to reinforcement learning, which can be very effective in environments with large state-action spaces, with the advantages of memory-based learning, which can be very effective when working with POMDPs. This combination is called a *Deep Recurrent Q-Network* (DRQN), and has shown itself to be more effective than *Deep Q-Networks* (DQNs) in a flickering version of the Atari game Pong [16], where there is a random probability that a frame will be blank. The DRQN is able to estimate the velocity of the ball, which allows it to achieve relatively high scores in the game despite having incomplete observations.

Proposed Study

Applications of memory-based deep reinforcement learning are numerous in engineering fields, especially in robotics. Often, some problems that are not easily expressed in terms of supervised learning may be expressed in terms of policies and rewards, the language of reinforcement learning. However, the environment is often complicated enough that function approximation becomes necessary. Additionally, almost all real systems are partially observable, providing motivation to pursue memory-based neural networks to incorporate past knowledge.

Current research in the area of deep reinforcement learning has been mostly focused on playing classic Atari games and similar problems; however, there is not much research into physical applications of this field. We plan to investigate how deep reinforcement learning with memory will perform in simulations of a robot in different maze environments, with the intent of comparing the effectiveness of different algorithms.

References

[1] J. Moody and M. Saffell, Neural Networks, IEEE Transactions on **12**, 875 (2001).

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, 1998).

[3] C. J. C. H. Watkins, Learning from Delayed Rewards, PhD thesis, University of Cambridge England, 1989.

[4] C. J. Watkins and P. Dayan, Machine Learning **8**, 279 (1992).

[5] H. V. Hasselt, in *Advances in Neural Information Processing Systems* (2010), pp. 2613–2621.

[6] H. Van Hasselt, A. Guez, and D. Silver, ArXiv Preprint ArXiv:1509.06461 (2015).

[7] D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber, in *Artificial Neural Networks–ICANN 2007* (Springer, 2007), pp. 697–706.

[8] S. Levine, N. Wagener, and P. Abbeel, in *Robotics and Automation (ICRA), 2015 IEEE International Conference on* (IEEE, 2015), pp. 156–163.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, ArXiv Preprint ArXiv:1312.5602 (2013).

[10] A. Hochländer, Deep Learning for Reinforcement Learning in Pacman: Deep Learning Für Reinforcement Learning in Pacman, PhD thesis, 2014.

[11] M. Egorov, (2015).

[12] H. A. M. Schäfer, Institut Für Informatik Neuroinformatics Group (2008).

[13] Q. V. Le, N. Jaitly, and G. E. Hinton, ArXiv Preprint ArXiv:1504.00941 (2015).

[14] J. Koutnik, K. Greff, F. Gomez, and J. Schmidhuber, ArXiv Preprint ArXiv:1402.3511 (2014).

[15] B. Bakker, in *NIPS* (2001), pp. 1475–1482.

[16] M. Hausknecht and P. Stone, ArXiv Preprint ArXiv:1507.06527 (2015).