Unsupervised Feature Learning for Reinforcement Learning

Matthew Can
Department of Computer Science
Stanford University

Abstract

This paper explores the use of unsupervised feature learning techniques in the context of reinforcement learning problems. In reinforcement learning, a common approach is to parameterize the states by a feature representation, and then learn which actions to take based on the features of the states. A great deal of effort often goes into feature engineering when building such systems, and the success of the system can depend heavily on the features. We examine whether it is possible to learn which features to extract automatically, all during the reinforcement learning process. We introduce a method that iterates between learning a feature representation and learning a policy. On a game simulation, we show that our approach performs as well as using the raw input as the feature representation, both of which lead to nearly optimal behavior.

1 Introduction

The goal of reinforcement learning algorithms is to learn a policy, a mapping from states to actions that will guide the behavior of an autonomous agent. For many interesting problems, the state space is very large (possibly infinite), making it impossible to learn this mapping directly for each state. Instead, a common approach is to parameterize the states by a feature representation, and then learn which actions to take based on the features of the states.

Typically, the programmer specifies such features manually, and they may be highly domain-specific. Sometimes much of the engineering effort spent on such problems is time spent on feature extraction. We will examine whether it is possible to learn which features to extract automatically, all during the reinforcement learning process.

Our approach is to apply techniques in unsupervised feature learning to the reinforcement learning framework. The idea is that an autonomous agent will learn a feature representation over states while discovering an optimal policy. In typical applications of unsupervised feature learning, the feature learning is done as a preprocessing step to another learning algorithm (e.g. for a classification task). In our setting, this is not possible because we will need to first carry out reinforcement learning in order to explore the state space and gather data points for the feature learning algorithm to do its job.

The next section describes the feature learning and reinforcement learning algorithms that we will employ. Section 3 will describe our approach for integrating unsupervised feature learning into the framework of reinforcement learning. Section 4 illustrates our approach on a simulation. We then discuss the results of our experiment, concluding with a discussion of our approach.

2 Preliminaries

We now provide a brief overview of the feature learning and reinforcement learning algorithms we use. Section 3 discusses how they are used together to simultaneously perform reinforcement learning while also learning a feature representation over states.

2.1 Unsupervised Feature Learning

Many unsupervised feature learning algorithms exist. They all attempt to find structure in the input data and use this as the basis for a learned feature representation. We will consider one such algorithm, the sparse autoencoder [1].

The sparse autoencoder minimizes the following cost function.

$$J = \frac{1}{m}\sum_{i=1}^{m}\left(||h_{W,b}(x^{(i)}) - x^{(i)}||^2\right)$$

Where $h_{W,b}(x^{(i)})$ is the output of a neural network with parameters W and b, input layer $x^{(i)}$, and a hidden layer that will consist of the learned features. Informally speaking, the autoencoder tries to learn a function $h_{W,b}(x^{(i)})$ that is approximately equal to the input $x^{(i)}$. To "force" the autoencoder to discover interesting structure as the basis for the learned features, the cost function is augmented to include a penalty for activating too many of the units in the hidden layer of the network. This encourages the autoencoder to learn a sparser feature representation for each input. Furthermore, the cost function typically includes a penalty for large weights as a means of regularization.

2.2 Reinforcement Learning

In reinforcement learning, one way of dealing with large state spaces is to approximate the value function for a state as a linear combination of features extracted on that state. Instead of trying to approximate the value function, we describe a policy in terms of the state's features and use policy search to learn the parameters of the model that maximize expected rewards [2]. In particular, we model action selection using a log linear model

$$p(a|s;\theta) = \frac{e^{\theta^T\phi(s,a)}}{\sum_{a'} e^{\theta^T\phi(s,a')}}$$

With this formulation, we can use a policy gradient algorithm to learn the model parameters. Specifically, we will use stochastic gradient ascent as in [3]. In an iteration of the learning algorithm, we play through an episode of our simulation. We sample actions according to the distribution above (occasionally sampling bad actions for the sake of exploring the state space, but with decreasing probability over time, as the parameter estimates get better). When the episode is over, there is some total reward R that the agent has accumulated. The parameters of the model are updated as follows:

$$\theta := \theta + \alpha * R * \sum_t (\phi(s_t, a_t) - \sum_{a'} \phi(s_t, a')p(a'|s_t;\theta))$$

Here, the outer summation is over the time steps of the episode, and α is the learning rate. At testing time, we choose actions that maximize the above probability distribution.

3 Method

As stated previously, unsupervised feature learning algorithms are typically used as a preprocess to some other learning algorithm. They learn a feature representation and pass it on to the next stage in the pipeline. While it may be possible to do the same in a reinforcement learning context, we choose instead to consider a scenario where the autonomous agent does not have any information about the state space until learning and exploration begin. Of course, the agent could always begin by taking random actions in the state space solely for the purpose of gathering data (make no attempt yet at learning an

1	0	1	0	1
0	0	1	1	0
0	1	1	0	0
0	1	1	0	1
1	0	1	0	1

0	1	0	1	1
0	1	1	0	0
0	0	0	0	0
1	0	0	0	1
0	0	1	1	0

1	0	0	0	1
0	0	1	1	0
1	1	0	0	1
1	0	1	1	0
0	1	1	0	0

Figure 1: Example states from our game simulation. The leftmost state is a terminal state with reward +1 (cells in center column are all on). The state in the center is a terminal state with reward -1 (cells in the center row are all off). The rightmost state is not a terminal state and has reward 0.

optimal policy), use that data to conduct feature learning, and only then carry out the reinforcement learning.

But, we do not use this technique because our goal is to have the reinforcement learning benefit the feature learning. For example, consider the case where states from the state space may look seemingly random (as they will in the simulation we present in the next section). Unsupervised feature learning will not perform well in this setting if it samples inputs via random state space exploration. There will be no structure to exploit in the data.

However, if we look only at the states that correlate with positive and negative rewards, we will likely find that these states share similar properties. These states will not seem random and will have structure that unsupervised feature learning can pick up on. The key idea is to guide feature learning to pick up on features that correlate with rewards. In some sense, the rewards observed by the agent provide a means of supervision to the feature learning.

Our approach is an algorithm that iterates between learning features and learning an optimal policy under those features. Given a feature representation, we conduct reinforcement learning as described in Section 2.2. As we estimate parameters for the current feature representation, we keep track of states with large positive and negative rewards. Once we have accumulated enough examples, we perform feature learning on that data to learn a new feature representation, iterating over. Note that we gather data from many episodes before learning new features. In particular, it does not make sense to learn new features after every observed episode because each update to the feature representation resets the parameters of the reinforcement learning problem (the old parameters are only valid for the old feature representation, so we would constantly be resetting the parameters).

4 Experiment

We conducted experiments of our algorithm on a simulation problem that we constructed. The simulation is a game where states of the game are n-by-n grids (in our case, we used 5-by-5 grids). Each cell in a grid can be either on or off. This leads to 2^{n^2} states for an n-by-n grid. Clearly, the number of states is too large for reasonably sized grids, so it makes sense to have a feature representation of the states for the purpose of generalizing actions over the state space.

The actions in the game are to toggle the cells in the grid. Cells that are on can be toggled off and vice versa. For every state in the game, each of the n^2 cells is an available action that can be toggled. However, when the agent attempts to toggle a cell, that cell will toggle only with probability 0.8. Instead, with probability 0.2, one of the neighboring cells (top, bottom, left, right) will be toggled instead (uniformly across all available neighbors).

The game terminates when either a row or a column of the grid has all of its cells on of off. That is, when a row or column is blacked out or whited out. If a row or column has all cells on, then game provides a reward of +1. If a row or column has all cells off, then the game provides a reward of -1. All states other than these terminal states provide a reward of 0. Figure 1 provides examples of states with positive, negative, and zero rewards. We use a

Features	Average Rewards
Raw grid features	0.8400
Learned features	0.8475
Raw + Learned	0.8175

Table 1: Average reward the agent received on 100 plays of the game using different feature representations of the game states.

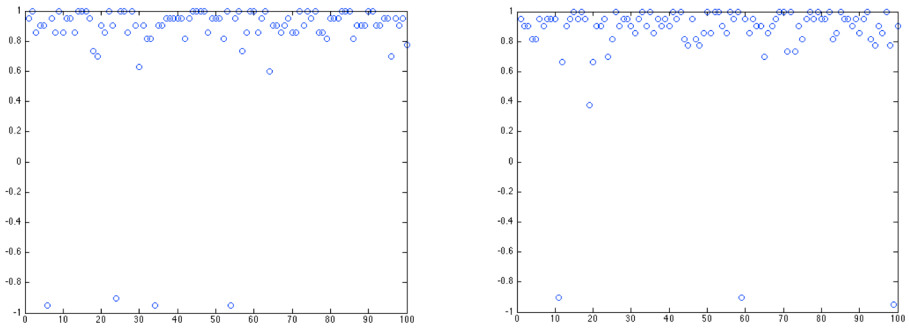

Figure 2: Rewards earned by the agent over 100 plays of the game. Left: using the grid itself as the feature vector. Right: using features learned by the sparse autoencoder. Negative rewards indicate suboptimal behavior.

discount factor of 0.95 so that terminal states that take longer to reach provide less reward.

When a new episode of the game is started, the initial state is chosen uniformly at random from the set of all possible grids (the game trivially terminates right away if a terminal state is chosen).

We can see that in this simulation we have set up, it does not make sense to simply use the sparse autoencoder in a pipeline configuration as a step before reinforcement learning. We cannot sample many states from the state space, provide them to the sparse autoencoder, and expect it to learn meaningful features. This is because we would essentially be providing it with randomness. This is why we suggested the iterative algorithm from the previous section. The hypothesis is that by training the autoencoder only on states that provide rewards, it will learn features that correlate with rewards, and that those features will lead to better action selection.

5 Results

We trained our algorithm for several iterations of repeated feature learning and reinforcement learning. For the feature learning, we trained a sparse autoencoder with 100 hidden units. In the reinforcement learning stage, we trained on 10,000 episodes game play. To provide better exploration of the state space when sampling from the policy distribution, we use a temperature parameter to smooth the distribution (set to 0.2 in our experiments). In addition, we used a learning rate of 0.1.

For comparison, we also tested the reinforcement learning algorithm by using the raw grid state as the feature vector. That is, we treated each state of the game as a binary vector and used that as the feature representation. Additionally, we concatenated the learned features to the raw input features and tested with these combined features as well.

We evaluated the learned policies by the average rewards obtained on 100 episodes of game play. In each episode, the initial state of the game was chosen uniformly at random over the non-terminal states of the game. It is important to keep this in mind, because in many cases the highest achievable reward from an initial configuration will be less than 1 if it takes

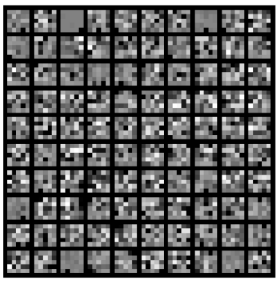

Figure 3: Visualization of features learned by the sparse autoencoder.

more than one move to reach a positive terminal state (due to the discount factor).

Table 1 contains the results of our evaluation. The average reward was slightly higher under the features learned by the sparse autoencoder than under the raw input features. However, the difference is negligible, so we can conclude that the performance is the same in both scenarios. Surprisingly, when combining the features, performance degrades slightly. We attribute this to the reinforcement learning algorithm's failure to converge. In our experience, as we increased the number of training iterations, the average reward under the combined features approached that of the other conditions.

Figure 2 plots the agent's rewards over the 100 iterations, comparing the raw input (grid) features to the learned features. Episodes that resulted in negative rewards are indicative of suboptimal behavior.

6 Discussion and Conclusion

Our hypothesis was that unsupervised feature learning could improve reinforcement learning by choosing a feature representation that facilitates reward-maximizing action selection. So, we begin our discussion by examining a visualization of the features learned by the sparse autoencoder on our game (Figure 3).

While some of the features appear to detect rows and columns that are partially blacked out or whited out, others seem to pick up on random configurations or single cells only. It is possible that it may not suffice to train the autoencoder on reward-giving states. Instead, we need a more explicit method of learning features that disambiguate between good and bad decisions.

Furthermore, our results indicated equally good performance between the raw input features and the learned features. In future work, we need to test our method on a problem where the raw features do not yield near-optimal performance in order to get a better sense of how much improvement feature learning provides.

References

[1] UFLDL Tutorial. http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial

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

[3] Branavan, S.R.K., Chen, H., Zettlemoyer, L.S. & Barzilay, R. Reinforcement Learning for Mapping Instructions to Actions. In *Procceedings of ACL 2009*.