Reinforcement Learning with Deep Architectures

Daniel Selsam
Stanford University
dselsam@stanford.edu

Abstract

There is both theoretical and empirical evidence that deep architectures may be more appropriate than shallow architectures for learning functions which exhibit hierarchical structure, and which can represent high level abstractions. An important development in machine learning research in the past few years has been a collection of algorithms that can train various deep architectures effectively. These methods have already led to many successes in the areas of supervised and unsupervised learning. They may prove to be just as useful in reinforcement learning as well, since solving a reinforcement learning problem depends on effectively approximating one or more state-value functions, which in general are just as likely to exhibit hierarchical structure as functions encountered in other settings. In this paper, we consider some of the issues that arise when trying to integrate ideas from deep learning into the reinforcement learning framework, present a class of algorithms which we refer to as Iterative Feature Extracting Learning Agents (I-FELAs), and compare their performance on the inverted pendulum problem to more standard approaches.

1 Introduction

In order to solve a reinforcement learning problem using an approximation architecture, the architecture must satisfy the following two conditions. First, it must be robust enough to effectively approximate the state-value functions of the policies generated; or to be more precise, to effectively capture the important local contours of the state-value functions, thus assigning higher values to better actions.[1] In particular, it must be robust enough to represent the important structure of the optimal state-value function.

However, being able to represent the important structure of the state-value functions is not in itself sufficient, or else the architecture consisting of all conceivable functions would always be our best choice. This brings us to our second condition: for each such policy, we must be able to find a setting of the parameters that yields a sufficient approximation of the corresponding state-value function.[2] For non-convex architectures, this can be a very hard problem even in the simpler case of learning with full supervision. In the supervised case we often have to sample many local optima and hope to find one that yields an acceptable approximation.

Yet it is even harder in the context of reinforcement learning, because we cannot simply sample trajectories using the optimal policy. In general, we can only gradually approach the optimal policy, and we do so by some variation of the following: continually improve our current policy by choosing a policy that is greedy with respect to our current approximation of the state-value function corresponding to our current policy. The new policy selected in this manner is only an improvement over

[1] By local we mean local with respect to the underlying MDP rather than local with respect to the representation of the state used as input for the architecture.

[2] This second condition encompasses the need to avoid over-fitting, which is not quite as big a challenge in RL as it is in supervised learning because it is often easy to simulate trajectories and thus to generate an abundance of samples.

the previous policy if the approximate state-value function accurately captured the local contours of the state-value function of the previous policy. Thus in order to reach an acceptable approximation to the optimal state-value function, we must successively reach reasonable approximations of the policies that we consider in sequence. Moreover, for an optimistic approach that does not re-initialize the architecture after each successive policy update, there must be a path in parameter space so that for each successive approximation, the setting of the parameters at the local optimum of the given approximation must be reachable from the setting of the parameters at the local optimum of the previous approximation.

The potential benefits of using deep architectures in reinforcement learning algorithms in the context of the first condition should be clear: as discussed above, there is both theoretical and empirical evidence that deep architectures may be more appropriate than shallow architectures for learning many kinds of functions. Since it is necessary in reinforcement learning to approximate state-value functions in order to find good policies, adding deep architectures to the reinforcement learning toolkit may increase the range of state-value functions that we can effectively represent and thus increase the range of problems that we may be able to solve. The main challenge of using deep architectures in the context of the second condition should also be clear: they are much harder to train, even in the context of supervised learning, and the difficulty is only exacerbated in the context of reinforcement learning.

2 Challenges

In general, the approximation architecture and the learning algorithm can be chosen independently, so that it is straightforward to use any type of approximation architecture with any of the standard reinforcement learning algorithms. However, trying to use deep approximation architectures presents a unique challenge, because the individual layers cannot be trained effectively all at once. Our main tool for training deep architectures has been first performing some form of greedy layer-wise unsupervised pre-training, with the hope that this pre-training sets the parameters in such a way that an acceptable local optimum can be reached by standard local descent methods [1].

It is not obvious, however, how to perform greedy-layer-wise unsupervised pre-training in the reinforcement learning context, for two main reasons. First, the only way to get samples is by taking actions, and choosing actions generally depends on evaluating the approximate state-value function, which prior to pre-training the layers, cannot be expected to yield an acceptable approximation of the state-value function. Second, the actual distribution of inputs is non-stationary, and in principle can vary with every single policy change, which, depending on the decision procedure employed, may in turn vary with every single update to the state-value approximation function. And while there may be some supervised learning problems in which the assumption of stationarity is not merited, in reinforcement learning problems there is often inherent, systematic non-stationarity; indeed, the main point of improving a policy is to sample states from a different distribution.

Because of these reasons, it would be problematic to perform the unsupervised pre-training while following an arbitrary policy, since we could be learning features for a very different problem than the one we actually care about. As Bengio writes about the auto-encoder: "...because [the encoding learned] is viewed as a lossy compression of x, it cannot be a good compression (with small loss) for all x, so learning drives it to be one that is a good compression in particular for training examples, and hopefully for others as well...but not for arbitrary inputs."[3]. Learning an encoding based on an arbitrary policy can in many cases be equivalent to learning an encoding for arbitrary inputs. On the other hand, if one waits until a good policy has been learned before extracting features and building a deep architecture, one risks waiting a long time–potentially forever–and missing out on the benefits a deep architecture might provide. Thus we need to develop a more sophisticated approach to pre-training the network, in which the unsupervised training and the decision generation improve together and feed off of each other.

[3] [1], 25

3 Iterative Feature Extracting Learning Agents (Iterative FELAs)

As discussed at the end of the previous section, the main challenge is to develop a framework in which the unsupervised training and the decision generation can improve together and feed off of each other. The basic idea behind our approach is simple: continually use the best policies known to pre-train a deep network, and then use that network to generate better policies. Before we go into the details, let us first consider a special case that is conceptually simple, though not computationally ideal for most problems.

The variant of reinforcement learning that is closest to a sequence of independent supervised learning problems is non-optimistic approximate policy iteration. In approximate policy iteration, we start with some policy π_0, and use some approximation architecture to approximate the state-value function corresponding to that policy as well as possible. Note that here we are not trying to improve the policy as our approximation improves; rather, we hold the policy constant, and fully evaluate the corresponding state-value function. Once our approximation has converged, we pick a new policy π_1 that is greedy with respect to our state-value estimates for π_0, reinitialize our architecture, and repeat.

Using a deep architecture with such a method is fairly straightforward. Given a policy π_j, we are essentially dealing with a single supervised learning problem, and can approach it in the standard way: continue generating sample trajectories, while first pre-training the layers of the deep network in an unsupervised fashion, and only after doing so, training the entire network as a whole in a supervised fashion. Then once we are satisfied with our approximation, we pick a greedy policy π_{j+1}, re-initialize our architecture, and repeat.

This approach is generally not desirable, however, because many of the problems we care about have additional structure that this method disregards. In particular, a single policy update may not change the underlying distribution so dramatically that we need to re-approximate its state-value function from scratch. Rather, the changes tend to be more gradual as we improve our policies, and thus this method may be performing a substantial amount of extra work by treating each policy as completely distinct from the previous one. But as we discussed at length above, it is also unsound to ignore the fact that the distribution is changing at each policy update; the state-value function corresponding to a given policy may have little in common with the state-value function for a policy that is learned either much earlier or much later. Therefore we need to find a middle ground between, one the one hand, retraining an architecture from scratch for each policy update, and, on the other hand, trying to learn the state-value functions corresponding to every policy of interest starting from a single initial setting for the parameters which was found by pre-training layers with respect to an arbitrary policy.

We propose a new class of algorithms, which we call Iterative Feature Extracting Learning Agents (I-FELAs), which attempt to find such a middle ground. An I-FELA is parameterized by a deep architecture A, a standard on-policy reinforcement learning algorithm L, and a transfer method T (which we will explain below), and works as follows. Given a deep architecture A_j, we perform the learning algorithm L, updating A_j and the policy used for decisions as indicated by L. While doing this, we use the states visited to pre-train the layers of the deep architecture A_{j+1}. Once we are satisfied with the pre-training, we transfer some of the information learned in A_j to A_{j+1} using T, and then repeat starting with the deep architecture A_{j+1}.[4]

Two examples of transfer methods are full transfers and null transfers. In a full transfer, we would suspend learning on A_j, yet continue to generate samples from the policy π_j that is greedy with respect to A_j, while training A_{j+1} until it converges to an approximation of the state-value function corresponding to π_j. In a null transfer, we would not transfer any information. Two examples of learning algorithms are optimistic TD(λ), and the null learning algorithm which performs no learning. Thus the the non-optimistic approximate policy iteration discussed above can be seen as a special case of an I-FELA with a null learning algorithm and a full transfer; that is, where the learning algorithm L performs no learning, and where the transfer method T involves using A_{j+1} to fully approximate the policy determined by A_j.

[4]Two comments are in order. First, this framework can be generalized in the obvious way to allow new architectures, learning algorithms, and transfer methods for each iteration. Second, the deep architecture A_0 may need to be initialized randomly, since we do have a distribution with which to pre-train its layers. In this case, it may be more effective to simply use a shallow architecture for A_0 and then use it to pre-train the layers of the first deep architecture A_1.

Algorithm 1 Pseudocode for I-FELA

$\{A$ is a deep architecture, L is an on-policy learning algorithm, and T is a transfer method$\}$
Initialize A_0
for $i = 1$ to n **do**
  for $j = 1$ to k **do**
    repeat
      Simulate $(s, a) -> (s', r)$ using procedure given by L and A_{i-1}
      Perform L on A_{i-1} given new sample
      Pre-train layer number j of A_i using s'
      $s \leftarrow s'$
    until Layer number j of A_i is sufficiently pre-trained
  end for
  Transfer learning from A_{i-1} to A_i using T
end for
return Best policy found so far

Ideally, the agent makes enough progress during a given stage that the layers of the next architecture will be pre-trained on a sample that is more representative of the distribution over states that we care about than the previous architecture was. This would predispose the next architecture to being able to approximation the state-value function of the recent policies better than the previous architecture could, and thus in turn lead to finding better policies than the previous architecture was able to find. Thus progress in the supervised and the unsupervised elements of the algorithm can feed off each other: the better we can approximate the state-value functions of the recent policies, the better the policies we can find; the better the policies we find, the better we can pre-train the layers of the next architecture, predisposing it to approximate the state-value functions of the recent policies more effectively.

Unfortunately, we have no guarantee of making monotonic progress. Although the I-FELA has a similar structure to a generalized policy iteration algorithm, in some cases it is perhaps more instructive to think of it as an iterative, heuristic deep-network initializer, in which we use the idea of layer-wise unsupervised pre-training to iteratively sample initial parameter settings for the deep architecture that seem promising. As discussed in the introduction, we are looking for areas of parameter space that can represent the state-value functions for several similar policies, and even with methods such as I-FELA, finding areas which contain 'paths' to the state-value functions of more desirable policies is still largely a matter of luck. If desirable policies are ever reached while performing learning on the architecture A_j, we recommend focusing on performing L on A_j, and if and only if it is necessary to move to a new architecture A_{j+1}, making an effort to transfer as much information from A_j to A_{j+1} as possible.

4 Empirical Results

The I-FELA is a very general framework, and can be used in countless different ways on different problems, by varying the approximation architecture, the learning algorithm, and the transfer method. This generality makes it hard to determine how useful this framework will turn out to be. As a first step, we have experimented with using it on the inverted pendulum problem. Our preliminary results are promising, and show the I-FELA to be a potentially powerful, if inconsistent, approach to solving reinforcement learning problems.

We attempted to solve the inverted pendulum problem as presented in problem set 4 directly without discretization. We first normalized the state vector so that each component was guaranteed to reside in the range $(-1, 1)$, and then using the intuition that the magnitudes of the components is more important than the signs, we added the squares of each of the state components. Note that we did not use any knowledge of physics, nor of the model itself, and that our pre-processing was less extensive than that which would have gone into a discretization step.

Using this expanded state vector, we experimented with a linear architecture, a 3-layer neural network, a 3-layer I-FELA with a full transfer method, and a 3-layer I-FELA with no transfer method, all four using optimistic TD(.9), and both I-FELAs using auto-encoders for pre-training the hidden

layer. The linear architecture performed almost as well as did value iteration in the discretized setting, and consistently reached the mid triple-digit range, and furthermore, tended to reach this range fairly quickly and remain there. The 3-layer neural network reached similar heights, but tended to do so more gradually and less consistently. The 3-layer I-FELA with full transfer method occasionally scored much higher, but on average seemed to perform only marginally better than the 3-layer neural network.

The 3-layer I-FELA with no transfer method is the only method that we experimented that was able to solve the problem entirely. On several different runs, we were able to balance the pole for 100,000 steps on consecutive trials. We terminated each trial after 100,000 steps in the interest in finishing our study in a finite amount of time, and believe that it may have been able to balance the pole indefinitely. Consistent with our expectations for I-FELA, we generally reached this level on the 2rd or 3rd cycle of the algorithm, i.e. while using A_1 or A_2 to generate policies.

However, there are four caveats that we must make very clear. First, although we found a solution to the problem on several different occasions, we ran the I-FELA many more times than that, and it was much more often the case that it performed in line with the two previous methods, and many times it performed even worse than the linear architecture. Second, one of the I-FELA's successes came in the first cycle of the algorithm while using A_0, which is essentially equivalent to finding a solution while using the standard 3-layer neural network. Therefore it is hard to gauge how much the unique elements of the I-FELA were actually responsible for the successes, and how much was just chance. Third, the I-FELA did not consistently make progress from one cycle to the next; rather, it seemed much more a matter of chance whether or not a given cycle would yield good policies. Fourth, there may be many other ways of solving this problem that we did not consider, and even the approaches we did consider may have performed very differently with different feature sets.

With these four caveats in mind, we think the performance of the I-FELA constitutes some evidence of the following claims. First, when training deep architectures in the reinforcement learning context, initial conditions can be critically important. As we saw above, with some initial settings we were able to solve the problem entirely, with others we were only able to perform mediocrely, and with others we never made much progress at all. Second, in some cases the I-FELA method–that is, continually initializing the parameters of an architecture based on unsupervised layer-wise pretraining of the states visited while making decisions based on the previous architecture–may be a valuable heuristic in setting the initial parameters. Although the I-FELA did not tend to make consistent progress, it still seemed to be sampling initial conditions from a better-than-random distribution, which given the importance of initial conditions, can in some cases be tremendously valuable.

5 Looking Forward: Possible Applications

I expect the most likely source of relevant problems will be the reinforcement learning counterparts of the kinds of problems that we already know can be addressed effectively with deep architectures. However, deep learning is still a young field, and we have only begun to explore its useful applications, even in the simpler supervised setting. Thus the true potential of using deep architectures in reinforcement learning is still not known. But over the next few years, as our understanding of both the theory and the useful applications of deep learning increases, I suspect it will become increasingly important to many kinds of supervised learning problems, and I doubt its relevance to reinforcement learning problems will lag too far behind. I hope the issues I have discussed and the algorithms I have proposed are helpful to other researchers as they begin to explore the potential of using deep learning in the reinforcement learning domain. Needless to say, a lot of experimentation is still required to discover which variants do and do not work on the real-world problems that we care about.

References

[1] Benjio, Y. (2009). Learning Deep Architectures for AI. *Foundations & Trends in Machine Learning*, 1-127.

[2] Bertsekas, D.P. & Tsitsiklis, J.N. (1996). *Neuro-Dynamic Programming.* Belmont, Massachusetts: Athena Scientific.

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