Function Approximation in Reinforcement Learning

Greg Maslov <maslov@cs.unc.edu>

***Abstract*—I discuss the background and motivation for the Neural Fitted Q Iteration algorithm, evaluate my implementation on a standard benchmark, and discuss future work.**

I. Introduction

I'm always on the lookout for ways to progress towards stronger AI. Reinforcement learning would be a great way to make robots do what we want — if only it worked! Unfortunately designing and training RL systems tends to be at least as difficult as solving the problem more directly. Part of the difficulty lies in finding a good representation for the Q function, policy, and/or state transition function (depending on your flavour of RL). Multilayer Perceptrons have been used as function approximators to this end, but that model comes with its own set of intractable difficulties. Neural Fitted Q Iteration addresses some of these.

Deep learning architectures are a relatively recent development in machine learning which shows great promise [1]. Their similarity to MLP models naturally suggests that some of this success could also be applied to reinforcement learning.

The goal of this project was to investigate the application of deep belief networks (DBNs) in place of MLPs as the Q function approximator, building on Abtahi's work[2]. Unfortunately I was only able to get as far as implementing NFQ on a plain MLP, without any deep structure, pretraining, or other bells and whistles.

II. Background

A. Q Learning

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a)+\alpha\left[R(s')+\gamma\max_{a'}Q(s',a')\right] \quad (1)$$

Equation (1) is the well-known update rule for Q-learning. The function $Q(s,a)$ represents the expected utility (total discounted future reward) of taking an action a in state s. If the values of Q are accurate, then an optimal policy π^* is to take the action with maximum Q-value in each state.

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

Applying the update rule whenever a new reward is received may cause the estimated Q to eventually converge to the true Q^*, depending on the policy being followed. Finding a policy that effectively balances exploration with exploitation in a complex environment is an open problem.

Besides that difficulty, there is also the question of how to store Q-values in a compact way that still allows for efficient updates. With a small discrete state and action space, there is no problem simply keeping a table of values. Alas, most interesting robotics tasks take place in the high-dimensional, continuous state and action spaces that characterize the real world.

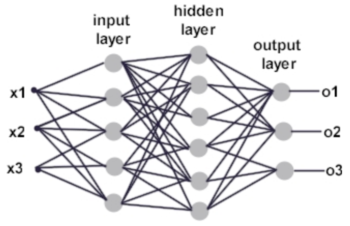

Fig. 1. Multilayer perceptron.

B. Multilayer Perceptron

The multilayer perceptron (MLP) is the quintessential feed-forward neural network architecture. Each node in the network applies weights, bias, and a nonlinearity (often a sigmoidal function) to the outputs of the previous layer. The network's entire topology and weights define a continuous vector-valued function of a vector, well-suited to representing Q.

With a differentiable nonlinearity, the parameters of the entire network can be modified using gradient descent to bring the output closer to a desired value for a given input. This is a natural way to implement the Q learning update rule. However, this naïve approach does not work well in practice, for several different reasons.

1) Combinatorial Explosion: Firstly, it must be noted that an MLP with one layer is merely a modified linear transformation of the inputs, and cannot represent complicated functions (as all but the most trivial of Q functions are). An MLP with two layers can represent any smooth function, and an MLP with three or more layers can, in theory, represent any function with a finite number of discontinuities.

This sounds great, but there is an unfortunate tradeoff hidden in the assumptions behind these theoretical results. If a Q function has many "features" – or worse, is periodic – then a two-layer MLP may require a truly excessive number of nodes to represent it. This is analogous to the ability, in digital electronics, of being able to represent any function of Boolean logic with a sufficiently large 2-layer AND-OR network. But the more nodes there are, the more weights need to be learned (going as the square), and the worse the performance, and the greater the danger of overfitting.

2) Local Minima: On the other hand, a network with more layers can compactly represent complicated functions by using earlier layers to compute useful intermediate transformations of the input. But remember that the network is being trained using gradient descent; the deeper back into the network it goes, the smaller the gradients become (due to fan-out), and the slower the training process becomes. There are methods of mitigating this effect (such as RPROP[3]), but the more fundamental problem is that the cost landscape of the network becomes vastly more complex as layers are added. That is, it tends to acquire a great deal of local minima, which become more and more difficult to escape. An MLP with one layer is already a non-convex optimization problem – adding more makes the convergence and optimality situation very bad, very fast.

There are clever methods of attacking this type of thorny optimization problem: stochastic gradient descent, simulated annealing, genetic algorithms, and various other metaheuristics. But the application of these methods tends to be something of a black art.

3) Nonlocality: Finally, a third problem is that the Q update rule demands a local change to the estimated function, but any modification to the weights of a neural network, particularly in the layers close to the input, is necessarily a global change. And the deeper the network is, the less predictable the effects of a weight change on distant Q-values will be.

Now in some sense this is desirable. The network is after all supposed to be storing the shape of Q in some compressed form, more efficiently than a simple table. These global changes are then called generalization, and can be a good thing.

Unfortunately, there is nothing forcing the network to make correct, sensible, or reasonable generalizations. Similar Q functions are nearby each other in the parameter space of the network, but the relationship between vectors in this space and inferences in the task domain is likely to be an intractable mystery.

C. Neural Fitted Q Iteration

Neural Fitted Q Iteration (NFQ) is a conceptually simple technique introduced by Riedmiller [4] in 2005 to address the tendency of MLPs to "forget" earlier training when too much time is spent in a limited region of the state space.

The idea is simply to store experiences (s, a, s', R) in a list during each RL episode while following a greedy policy, then iteratively training on all of them in a batch at the end of an episode. This has two beneficial effects: first, the change from single updates to batch updates allows the use of a more sophisticated variation on gradient descent; namely, RPROP, which is faster and far less sensitive to learning rate and batch size.

Second, this method directly addresses the problem of non-local updates and generalization described above. The effect of including earlier training pairs in the batch is to *constrain* the direction that generalization can go in, to only directions which do not disrupt the other Q values in the training set.

III. Results

I used Python2 and the Theano, RL-Glue, and RL-Library libraries to implement NFQ+RPROP on the CartPole balancing task benchmark. I used a one-hidden-layer, tanh-activation MLP with 5 hidden nodes. My agent converges to a successful policy after about 200-300 epochs. This result roughly matches that obtained by [4].

- Demo/results video:
  - http://www.youtube.com/watch?v=upoDIFzAets
- Source code:
  - https://github.com/gmaslov/rl-experiments

IV. Future Work

Unfortunately, I did not have time this semester to investigate a DBN[2], M-DBN[5], or autoencoder[6] Q-representation, nor any method of active learning.

The major difficulty that NFQ has, whether operating on an MLP or a DBN[1], is the heavily unbalanced distribution of states that arises from following a greedy policy. Furthermore, a random policy is completely ineffective for exploration in more interesting environments. That's why I believe future research in this area should focus on directed exploration and active learning.

I imagine that there must be some way to use the fact that a DBN is a generative model to create a model-based active learning scheme. Possibly inspired by RMAX or its variations[7]. I haven't been able to come up with anything, though.

References

[1] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[2] F. Abtahi and I. Fasel, "Deep Belief Nets as Function Approximators for Reinforcement Learning," in *AAAI Workshops: Lifelong Learning*, (San Francisco), AAAI, 2011.

[3] M. Riedmiller, "Rprop - Description and Implementation Details," tech. rep., 1994.

[4] M. Riedmiller, "Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method," *Machine Learning: ECML 2005*, pp. 317–328, 2005.

[5] L. Pape, F. Gomez, M. Ring, and J. Schmidhuber, "Modular deep belief networks that do not forget," *The 2011 International Joint Conference on Neural Networks*, pp. 1191–1198, July 2011.

[6] S. Lange and M. Riedmiller, "Deep auto-encoder neural networks in reinforcement learning," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, Ieee, July 2010.

[7] I. Szita and C. Szepesvári, "Model-based reinforcement learning with nearly tight exploration complexity bounds," in *27th International Conference on Machine Learning*, 2010.

[1]as in [2]; note that with no unsupervised pretraining a DBN is equivalent to an MLP.