CS8803-O03 Reinforcement learning Project 2 report

Rohan D. Kekatpure
Email: rdk@gatech.edu

I. Introduction

The aim of this project is to train a reinforcement learning (RL) agent in a discrete-action continuous-state-space environment. The chosen agent is the lunar lander in the Box-2D environment. This project brings together many aspects of the course together and is a miniature version of latest advances in the field of RL. As such, despite the pain and nearly 100+ hours spent on this project, it has forced synthesis of the class material.

II. Quick theory tour

The model-tree reinforcement learning (control) problem can be condensed into three equations:

$$Q(s,a) = f(s,a;w) \tag{1}$$

$$\Delta w_t = \alpha\left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right] \tag{2}$$

$$\begin{aligned} a_{t+1} &= \arg\max_{a'} Q(s_{t+1},a') \\ &= \arg\max_{a'} f(s_{t+1},a';w_t), \end{aligned} \tag{3}$$

where $Q(s,a)$ is the action-value function specifying the value of taking an action a in state s. For large and/or continuous state spaces, the Q function needs to be approximated. This approximation is expressed through the parameterized generalization function f. The **optimal** next action is the one that yields largest Q function value when evaluated with current parameters. Whether this action is **chosen** or not depends on the exploit-explore strategy for the problem. The formulation also has a number of hyperparameters: the learning rate α, the discount factor γ and the exploration probability ϵ.

Note that there are alternative formulations of the problem. Particularly, the RL part can include more number of discounted states (e.g. TD(1) or Sarsa). The function approximation can be non-parameterized (nearest neighbors or table-lookup). But Eqs. (1)–(3) specify the mathematical formulation of the implementation presented in this report.

III. Implementation methodology

A. Overview

Our implementation for this project is a simplified version of the methodology presented in the DQN paper[1]. While the RL part of the overall algorithm is identical to Algorithm 1 in the paper, our function approximation technique differs in from the DQN implementation in two important ways: (1) we use a simple, fully connected feed-forward neural network for function approximation (as opposed to convolutional net (CNN) in the DQN paper) and (2) our "state" is a combination of the 8-component vector provided by the environment and the **one-hot-encoded** action vector.

For Q function generalization, we use a simple feed-forward neural network (motivation in § IV-A) with two hidden layers of 50 nodes each and rectified activation. Our output layer is a single neuron without any activation. As a result of fusing the raw state with (encoded) action vector, our implementation needs to do 4 passes for the "prediction" part. The optimal action is the one that maximizes the output value.

B. Software dependencies

Our solution is implemented within standard Python ecosystem; the only external dependencies are numpy and scikit-learn. In particular, we **don't** use high-performance neural network learning frameworks such as Theano or Tensorflow, or their wrappers like Keras.

IV. Experiments

A. Choice of learner ("do we really need a neural net?")

RL success stories (TD-Gammon, Atari, Go) leave little doubt about the efficacy of neural nets as the function approximators. Yet, it is not obvious (at least to this author) why other supervised algorithms would not be able to outperform neural nets. To test the strength of other supervised algorithms on RL tasks, we experimented with linear, support-vector and decision tree learners (regressors). Table I summarizes the maximum average reward per 100 episodes that these learners were able to achieve.

Learner	Max avg. reward	sklearn params
Linear regression	< -200	-
Support vector	< -170	$C = 0.01, \epsilon = 0.2$
Regression tree	< -115	tree depth ≤ 15
Neural net	$> +55$	layers = $(50, 50), \alpha = 10^{-3}$

TABLE I
Maximum 100-episode average reward achieved by different learners on Q learning.

Our empirical evidence, though not exhaustive, hints to superiority of neural nets over other learning algorithms for Q function generalization. One reason might be the non-linearity coupled with extreme non-convexity in the Q function landscape. As a result, linear regression may have insufficient flexibility (i.e. high bias) to approximate the Q function. In theory regression trees can approximate nonlinear and non-convex functions. Yet, in practice it may be difficult to search

the hypothesis space efficiently in decision-tree representation. In addition, the hidden layers in neural nets have been interpreted to be representations of latent information in raw features (e.g. "face-like" or "nose-like" etc). This made it clear that we needed the expressive power of a neural net for Q-function approximation for this task.

B. Replay memory

After failures with batch learning using a single episode, we settled on training in every step of every episode using uniform random sampling of the total experience (termed "experience replay" in the DQN paper). We noticed that training with states from a single episode makes the weights diverge. For our agent we fixed replay memory at 50000 and the batch size at 32.

C. Neural net parameters

1) Hidden layers and batch size: One drawback of neural network is the requirement for fine tuning of a number of hyperparameters. We ran a small number of experiments to get a general idea of the number of **layers**, number of **nodes per layer**, number of **epochs** and the **batch size**. It is computationally a difficult task to perform extensive parameter sweeps, so the choice of experiments is somewhat arbitrary.

The fixed parameters for our neural net were warm start set to `True` (i.e. use a momentum term), and "adam" as the SGD solver. As seen in Table II, the number of neural net training epochs per episode or larger number of hidden nodes don't necessarily lead to better fits. Reducing the epochs and hidden nodes does lead to significant speedup at the cost of a slight drop in max reward. As a result, our final neural net implementation used **two** hidden layers with **50 nodes** each and a **32 sample** batch size.

layers	epochs	solver	iters	batch	speed	max rew
(50, 50)	50	"adam"	1000	128	slow	+124
(100, 100)	50	"adam"	1000	256	slow	+85
(50, 50)	20	"adam"	700	32	ok	+132
(50, 50)	2	"adam"	700	32	fast	+90

TABLE II
MAXIMUM 100-EPISODE AVERAGE REWARD ACHIEVED BY DIFFERENT NEURAL NET ARCHITECTURES.

2) Number of training epochs: We train our neural net at each step of every episode. Allowing the neural net training to run for the default number of epochs (200 in sklearn's `MLPRegressor`) not only increases the training period, but also **overfits** to the given batch. To speed up training and to avoid overfitting, we fixed number of training epochs per episode to 2. Figure 1(a) demonstrates that a wrong choice of the training epochs can lead to under-training and unoptimized agents.

3) Learning rate: We estimated the optimal range of the learning rate α by fitting a 2 periods of a sine wave (Fig. 1(b) inset) using the neural net fixed using the parameters defined in §IV-C. The sine wave was selected as a proxy for a non-convex, nonlinear function. The learning rate vs the fit quality is plotted in Fig. 1(b). It is clear that $\alpha > 0.01$ can lead to unstable behavior even for this simple one-dimensional learning

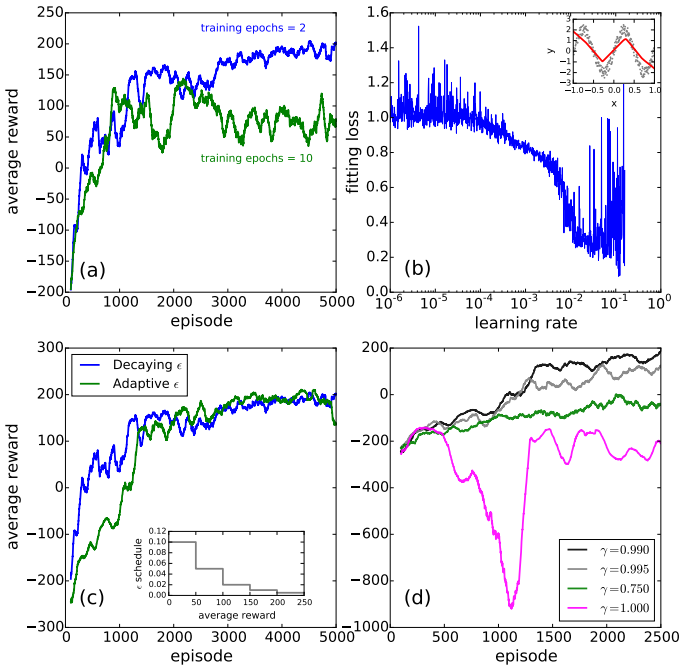

Fig. 1. Experiments on hyperparameter tuning. (a) Training effectiveness (100-episode average reward) vs episodes for two different epochs per episode. Optimal epochs per episode $\simeq 2$. (b) Learning rate (α) vs fit quality measured using a 2-period sine wave(inset) fitted using our neural net regressor. Optimal learning rate $\simeq \alpha = 10^{-4}$. (c) 100-episode average reward vs episodes for two different exploration strategies ϵ. The blue curve is fixed geometric decay and green curve is adaptive decay using the schedule shown in the inset. (d) 100-episode average reward for four discount factor γ values shown in the legend. Optimal $\gamma = 0.990$

task. To keep the learning behavior stable we conservatively set our learning rate at a **fixed** value of $\alpha = 10^{-4}$. This value offered a fast monotonic convergence.

D. Exploit-explore strategy

The explore-exploit strategy is captured by the decay schedule for the ϵ parameter. We tried two different strategies, whose results are shown in Fig. 1(c). The first experiment, shown in blue is standard time-dependent **decay** in which we exponentially decay the initial ϵ until it reaches a minimum value: $\epsilon_t \sim \max(\epsilon_{\text{min}}, \epsilon_0^t)$. The second strategy is **adaptive** to the average reward; the exploration probability reduces near and optimum and jumps back up if the agent moves away from the optimum. This reward-adaptive schedule is shown in the inset to Fig. 1(c). There is a slight difference between the initial learning speed, but the agent converge to similar terminal states. As a result we chose time-dependent ϵ decay schedule with $\epsilon_0 = 0.999$ and $\epsilon_{\text{min}} = 0.01$.

E. Discount rate

Typical episode length for the lunar lander is between 100 and 500–i.e. practically finite horizon. Our initial intuition therefore was that the discount rate would have little effect on the learning. Our experiments proved this intuition **wrong**. As seen in Fig. 1(d) the discount rate γ hugely affects the rate of learning as well as the converged value. Our best

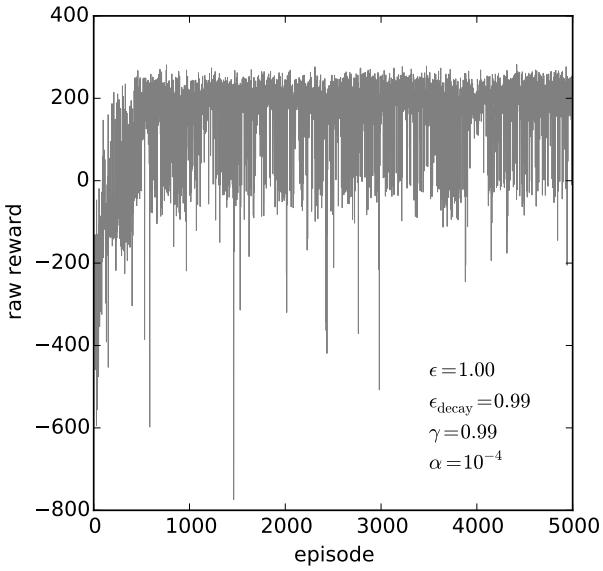

Fig. 2. Reward at each episode during agent training. The parameters used for training are shown. The agent receives a 100-episode average reward > 200 at episode 1165 and is considered **trained**.

performance was obtained at $\gamma = 0.990$. The performance deteriorated slightly for $\gamma = 0.995$ and was significantly worse for $\gamma = 0.750$. For undiscounted learning (i.e. $\gamma = 1.000$), the agent showed **no evidence of learning** for 2500 episodes.

V. Results

A. Behavior during training

After the choice of hyper parameters, we ran the training of our agent for 20000 episodes. The reward for each episode during training for first 5000 episodes is plotted in Fig. 2. Although there are many early episodes with reward > 200, they're chance occurrences. The agent is fully trained (average reward > 200) only after episode 1165. We note that the training of the agent is fairly robust as evident from the relatively flat (though noisy) training curve.

B. Behavior after training

Fig. 3(a) shows the smoothened version of Fig. 2. Here we've also shown the entire 20000 training episodes to highlight regions of training, trained and **overfitting**. The average reward is stable and between 150 and 200 for episodes between 1165 and $\sim$ 14000. After that we enter the regime of overfitting as seen from the rapid drop and downward trend in the average reward. Overfitting occurs when the weights in the network which would be zero assume large nonzero values due to continued training. This is a well known phenomenon in neural net training which leads to a recommendation of **early stopping**. Fig. 3(b) shows three agents trained with different exploration policies (other parameters unchanged) to highlight the **robustness** and **reproducibility** of our training procedure.

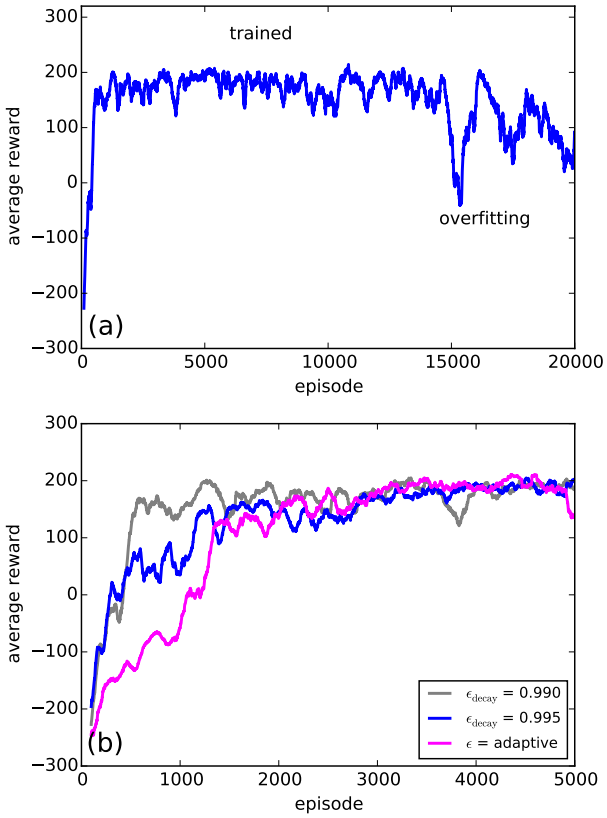

Fig. 3. (a) 100-episode average reward during entire training showing the regions of training, trained and overtraining/overfitting. (b) Demonstration of **reproducibility** of training. Three agents trained using three different exploration strategies. The adaptation strategy used is depicted in the ineset of Fig. 1(c).

Summary

Building the lunar lander control module taught us model-free learning with generalization. We evaluated linear, support vector, and neural net regressors as well as regression trees. Within our limited experiments, we found neural nets to provide the best out-of-the-box performance for lunar lander Q function generalization. For neural nets we fixed two hidden layers but experimented with number of hidden nodes, number of training epochs, learning rate and batch size. We found a range of these parameters which provides **reproducible** and **stable** training for our agent. Finally our RL experiments with the discount rate and the exploration strategy demonstrated the considerable effect of the discount rate on the optimization of the agent. With the learning in this exercise, we can build simple agents for other similar tasks including the walking robot, pong and mazes.

References

[1] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," *arXiv:1312.5602 [cs.LG]* 2013.

[2] Richard S. Sutton and Andrew G. Barto, "Reinforcement Learning," *MIT Press*, 1998.