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 RL and the course together. The project 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. 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) we our total "state" is a combination of the 8-component state provided by the environment and the **one-hot-encoded** action vector.

B. Choice of hyperparameters

1) **Exploration probability ϵ:** We employed a TD(0)-like RL algorithm with an ϵ-greedy exploration strategy. The parameter ϵ is initialized at 1.0 and is decayed by 1% ($\epsilon_{\text{decay}} = 0.99$) at each step until it reaches a minimum of 0.1. The motivation to keep exploring is to avoid being stuck in local maxima. One local optimum we observed in the training of our agent was when the agent avoids landing. To avoid crashing with a reward of -100, it keeps firing the main engine until the end of episode. The particular value of $\epsilon_{\text{decay}} = 0.99$ keeps the probability of exploration $> 10\%$ until 100 episodes. The DQN paper uses similar (though not identical) values.
2) **Discount factor γ:** We found that lunar lander episodes last on average for a 100 steps. So the RL horizon is practically finite and we do not expect γ to impact the solution very much. We therefore started our experiments with $\gamma = 0.99$ and found that this value led to convergence. We repeated the experiments for values $0.70 \leq \gamma \leq 0.99$ and found little effect on the convergence behavior.
3) **Replay memory and training batch size:** After our failed experiments with batch-mode updates using states in a single episode, we settled on batching and experience replay as recommended 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. Once again we found no appreciable difference in convergence or stability for batch sizes of $\{32, 64, 128\}$.
4) **Learning rate α:** The learning rate crucially affects the convergence of Q learning. We chose a **constant** learning rate of $\alpha = 10^{-4}$ by logarithmically varying it from 10^{-5} to 10^{-2}. Our chosen value offered a fast monotonic convergence.
5) **Neural net training epochs per episode:** 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 a particular batch. To speed up training significantly and to avoid overfitting, we fixed number of training epochs per episode to 2.

C. Motivation for 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, SVM and decision tree learners (regressors). Table ?? summarizes the maximum average reward per 100 episodes that these learners were able to achieve.

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. Regression trees are provably **universal function approximators**. In theory they have equal expressive power as neural nets. 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).

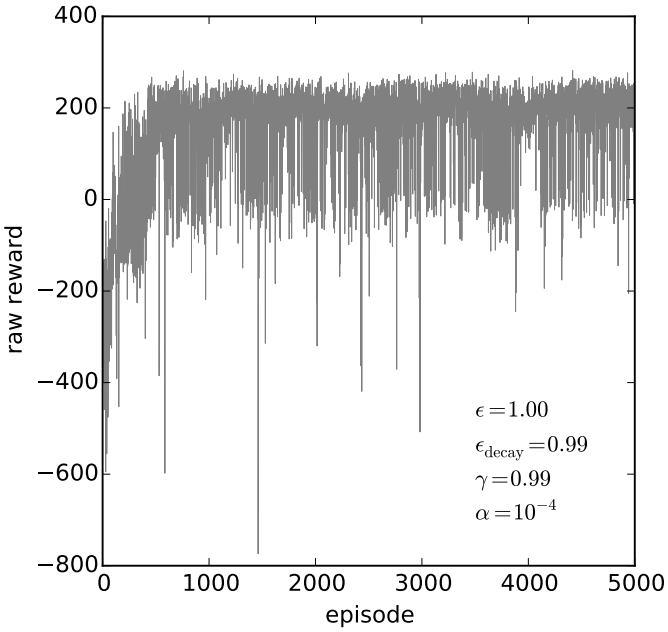

Fig. 1. fig1

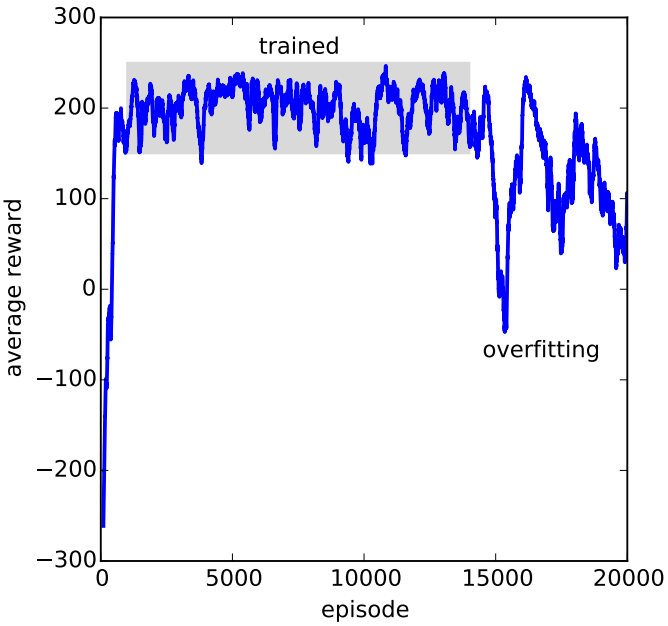

Fig. 2. fig2

D. Neural net structure

We use a simple feed-forward neural network with two hidden layers of 50 nodes each and rectified activation. Our output layer is a single neuron without any activation. As mentioned previously, the network is trained with a constant learning rate of $\alpha = 10^{-4}$. As a result of fusing the raw state with (encoded) action vector, our implementation needs to do 4 passes for the "prediction" part. The chosen "action" is the one that maximizes the output value.

E. 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.

III. Experiments

Conclusion

References

[1] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," *arXiv:1312.5602 [cs.LG]* 2013.
[2] Richard S. Sutton, *Machine Learning*, **8** p. 9–44, 1988.
[3] Richard S. Sutton and Andrew G. Barto, "Reinforcement Learning," *MIT Press*, 1998.

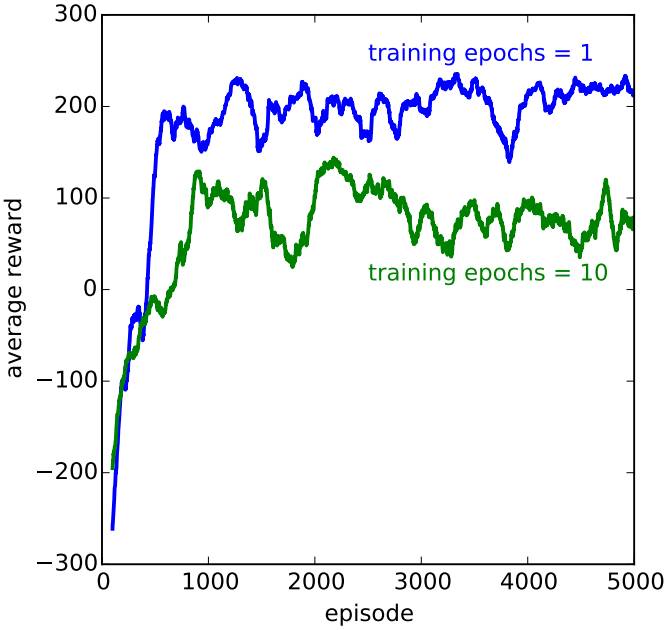

Fig. 3. fig3