Methods for solving continuous state and discrete action space reinforcement learning problems

September 19, 2019

Outline

Problem at Hand

"Methods for solving continuous state and discrete action space reinforcement learning problems":

- Reinforcement Learning context – trial-and-error search, delayed rewards
- Particular class of RL problems – state is continuous, action space is discrete
- Several methods – we investigate what works: Random Agent, SARSA, Deep Q-Network
- Tested on OpenAI Lunar Lander environment
- "Solved" when the trained agent obtains satisfactory performance (more than 200 points averaged over 100 runs)
- Live demo!

Motivation (General)

Different angle on existing problems

Eg. Find shortest path in the maze, play chess, trade stocks.

New types of problems

Eg. Play computer games, navigate a drone.

Make use of a simulation environment

Eg. Given an already developed simulator (General Mission Analysis Tool by NASA), create an autopilot for satellites.

Motivation (Applications in NLP)

Article summarization

See "A Deep Reinforced Model for Abstractive Summarization" by Xiong et al (`https://arxiv.org/abs/1705.04304`).

Question answering

Given a question q reformulate q to elicit the best possible answers. The agent seeks to find the best answer by asking many questions and aggregating the returned evidence (`https://arxiv.org/abs/1705.07830`).

Text generation

Think of a dialogue as a sequential decision making with each reply being a "step" (`https://arxiv.org/abs/1606.01541`).

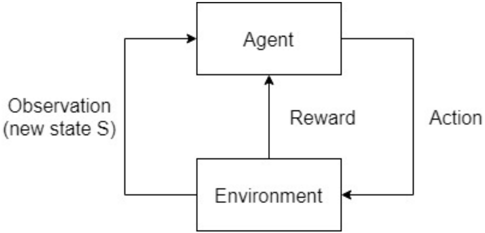
Reinforcement Learning Problem

- Learning how to map situations to actions
- Trial-and-error search
- Delayed feedback
- Trade-off between exploration and exploitation
- Sequential decision making
- Agent's actions affect the subsequent data it receives

RL World

Components:

1. State S_t
2. Action $A_t(s)$
3. Reward $R_t(s, a)$
4. Policy $\pi(s)$
5. Reward function $\mathcal{R}(s)$
6. Value function $V(s)$
7. Transition Function $Pr(s'|s, a)$
8. Model



Definitions (1)

- Value function – long-term “value” of a State s:

$$v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s\right]$$

- Policy – distribution over actions given states:

$$\pi(a|s) = \mathbb{P}\left[A_t = a | S_t = s\right]$$

- Action-value function – expected return starting from state s, taking action a, and then following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi\left(S_{t+1}, A_{t+1}\right) | S_t = s, A_t = a\right]$$

After applying the Bellman equation

Definitions (2)

- Optimal Value function – maximum value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s)$$

- Optimal Policy – policy that "beats" any other policy:

$$\pi_* \geq \pi, \forall \pi$$

where

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s$$

Solutions to Our Problem Class

Characteristics:

- Model is not known (no $\mathcal{R}$, no $\mathcal{P}$)
- State is continuous
- Action space is discrete

Solutions:

- Model-based: learn a model $\rightarrow$ solve MDP
- Value-function based: learn $Q \rightarrow$ argmax
- Policy search: directly find π

Model-free Value-function based methods

- Monte-Carlo RL
  - from complete episode
  - episode must terminate
  - unbiased
  - high variance
- Temporal-difference RL
  - bootstrapping
  - episode can be infinite
  - biased
  - lower variance
  - faster convergence

Action Selection

- ϵ-greedy (randomly selected action with probability ϵ, greedy otherwise)
- ϵ-greedy with decay
- Softmax

$$\pi(a) = \frac{\exp\left(\beta Q_t(a)\right)}{\sum_{a' \in \mathcal{A}} \exp\left(\beta Q_t\left(a'\right)\right)}$$

where $\beta > 0$ controls the greediness of action selection ($\beta \to \infty$ results in a greedy choice)

SARSA Algorithm

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Source: Sutton

Function Approximation

Estimate value function with function approximation:

$$\hat{v}(s, w) \approx v_\pi(s)$$

Update w with SGD $\rightarrow$ differentiable approximation functions:

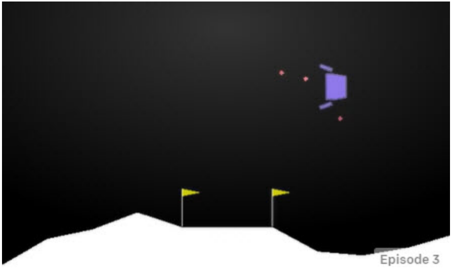
$$J(\mathbf{w}) = \mathbb{E}_\pi \left[\left(v_\pi(S) - \hat{v}(S, \mathbf{w}) \right)^2 \right]$$

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha \mathbb{E}_\pi \left[\left(v_\pi(S) - \hat{v}(S, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right]$$

$$\Delta \mathbf{w} = \alpha \left(v_\pi(S) - \hat{v}(S, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

Lunar Lander Environment

OpenAI gym provides us with an interactive environment to control a robot. We have 8-D continuous state space and action space of 4 discrete actions – do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine. We are also controlling a robot during the flight.



Experiment Design

In order to reproduce and compare RL algorithms we devise a strict procedure for the experiments. For each model we stick to the following workflow:

- Run a training procedure for several ranges of episodes (eg. 100, 1000, 10000) and for several sets of hyper-parameters
- Analyze the trade-off between the running time and model quality, select best-performing hyper-parameters
- Save the weights for best-performing hyper-parameters
- Load the weights and run a testing procedure for 100 episodes
- Plot the score vs each testing episode

Random Agent

Random agent performance on 100, 1000 and 10000 episodes:

# Episodes	Avg Training Score	Last 100 Episodes
100	-175.399	-175.399
1000	-175.124	-197.221
10000	-182.212	-172.897

SARSA Implementation

The idea of the algorithm is to approximate an action-value function Q(s, a) with a function Q(s, a, w) and to update a weight vector w iteratively in the direction of the gradient:

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s, a) = \sum_{i=1}^{d} w_i \cdot x_i(s, a) \tag{1}$$

So that the gradient of the $\hat{q}(s, a, \mathbf{w})$ is:

$$\nabla \hat{q}(s, a, \mathbf{w}) = \nabla \mathbf{w}^T x(s, a) = x(s, a) \tag{2}$$

With the update rule for weight vector:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{q}\left(S', A', \mathbf{w}\right) - \hat{q}(S, A, \mathbf{w}) \right] x(s, a) \tag{3}$$

Empirical Results

The best results were achieved with the learning rate of 0.1, gamma of 0.9 and exponential epsilon decay:

# Episodes	Avg Training Score	Last 100 Episodes
100	-253.549	-253.549
1000	-309.499	-134.525
10000	-148.945	-126.867

SARSA – Conclusion

- Linear function approximation cannot capture the complexity of the state landscape in the Lunar Lander case.
- To use episodic semi-gradient Sarsa efficiently we need either to define a more sophisticated non-linear approximation function or to switch to another approach.
- Linear function approximation is compact. It takes a vector of $n + m$ values to represent w. For the Lunar Lander problem parameter weights use about 1 KB of disk space when saving with Python *joblib* library. Other algorithms (as we will see with state discretization) can require much more space.

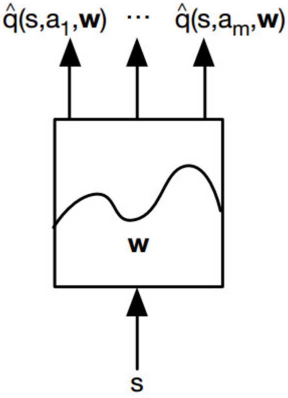
Deep Q-Network Implementation (1)

- State is represented by a feature vector $\mathbf{x}(S) = \begin{pmatrix} \mathbf{x}_1(S) \\ \vdots \\ \mathbf{x}_8(S) \end{pmatrix}$

  where the features are (x,y) coordinates, velocity components on the x and y axes, angle of the lunar lander, angular velocity, binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground

Deep Q-Network Implementation (2)

Approximator is an MLP of the architecture 8 x 500 x 200 x 100 x 4. Layers are dense with relu activation.



Deep Q-Network Implementation (3)

- Approximate the action-value function
- Minimize MSE between approximate action-value function and true action-value function
- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = \left(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

$$\Delta\mathbf{w} = \alpha\left(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

- TD(0) is

$$\Delta\mathbf{w} = \alpha\left(R_{t+1} + \gamma\hat{q}\left(S_{t+1},A_{t+1},\mathbf{w}\right) - \hat{q}\left(S_t,A_t,\mathbf{w}\right)\right)\nabla_{\mathbf{w}}\hat{q}\left(S_t,A_t,\mathbf{w}\right)$$

Deep Q-Network Implementation in Keras

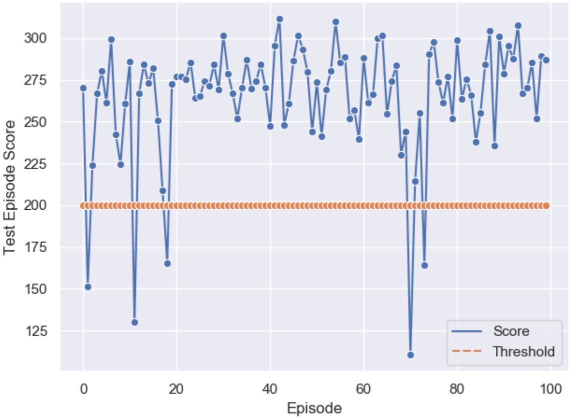
DQN uses experience replay and fixed Q-targets:

- Take action a_t according to ϵ-greedy policy
- Store transition (s_t , a_t , r_{t+1}, s_{t+1}) in replay memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimize MSE between Q-network and Q-learning targets
  $\mathcal{L}_i(w_i) =$
  $\mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; w_i^-\right) - Q\left(s, a; w_i\right)\right)^2\right]$
  Using variant of stochastic gradient descent

Source: Silver; Mnih et al

DQN Empirical Results

The results after 1 million iterations (mean is 265):



DQN – Conclusion

- DQN is able to solve the Lunar Lander problem.
- DQN takes longer time to converge (converged after 1 million iterations).
- Deep neural net approximation is not compact. In our case the network has over 100k of parameters. For the Lunar Lander problem parameter weights use about 512 KB of disk space when saving with Python *h5f* library.

SARSA With State Discretization – Implementation

- Initialize Q(s,a) table with 0 as a default value for each state-action combination.
- After receiving a state from the environment, discretize it using a binning approach.
- At each step take an ϵ-greedy action – random with probability ϵ and argmax(Q) with probability $1 - \epsilon$.
- Compute the TD-target as

$$Target = R + \gamma Q\left(S', A'\right) \tag{4}$$

- Compute the TD-error as

$$Error = Target - Q\left(S, A\right) \tag{5}$$

- Update values of Q-table:

$$Q(S, A) \leftarrow Q(S, A) + \alpha Error \tag{6}$$

Tuning Hyper-parameters

Feature dimensions

Setting the number of bins (dimensions) of each state variable determines the size of the Q-table and the amount of exploration we need to do. The number of discretized states grows exponentially with the number of feature dimensions.

Alpha

Learning rate heavily influences the outcome of the training process. Unlike neural nets, smaller alphas scored worse than alphas of $0.1 - 0.2$.

Epsilon

Several options are available: constant value, exponential decay and piece-wise linear function.

Tuning Hyper-parameters – Alpha

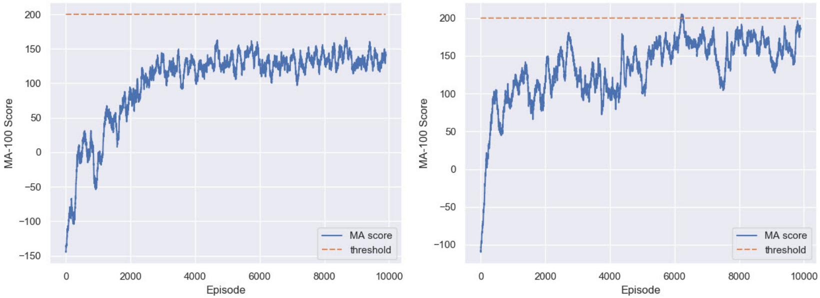


Figure: Training Scores For Alpha=0.05 (left) and Alpha=0.2 (right)

Training The Agent

Training for 20,000 episodes, gamma=0.95, alpha=0.2 and a piecewise linear epsilon schedule:

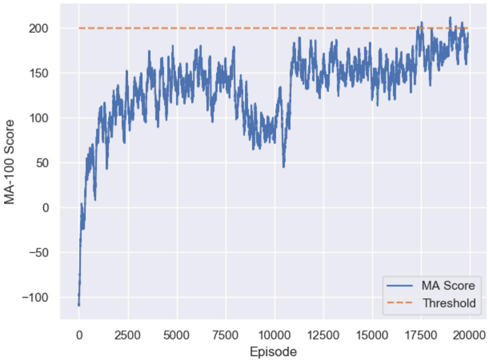


Figure: Moving Average-100 Training Scores For 20k Episodes

Testing The Agent

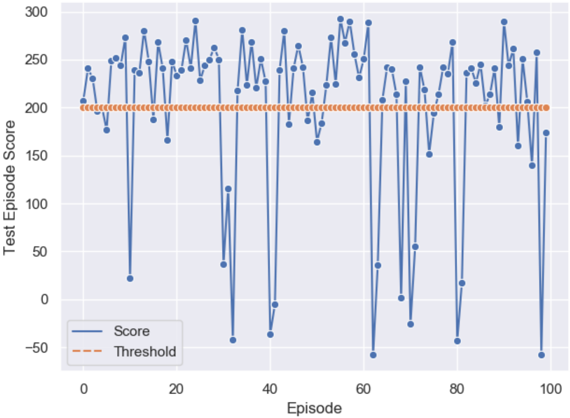


Figure: Testing Scores For 100 Episodes

Conclusion

- RL methods can be applicable to a wide variety of problems
- Out-of-the-box models work but require fine-tuning and take longer to converge
- Simple methods like state discretization are worth exploring when training speed and solution complexity are of the essence

References

Reinforcement learning: an introduction, 2nd Edition. Richard S. Sutton, Andrew G. Barto.

Reinforcement learning lectures by David Silver. UCL. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

Playing Atari with Deep Reinforcement Learning. Mnih et al. https://arxiv.org/abs/1312.5602

Thank you for your attention!