Deep Reinforcement Learning Report

Jiankai Sun
Shanghai Jiao Tong University
800, Rd Dongchuan, Shanghai, China
jiankai@sjtu.edu.cn

Abstract

In this paper, I make a summary of the Reinforcement Learning lessons given by David Silver. There is also some analysis of the codes Atari and TensorFlow implementation of Deep Reinforcement Learning papers.

1. Introduction

Reinforcement Learning includes Markov Decision Process(MDP), Planning by Dynamic Processes, Model-Free Prediction, Policy Gradient Method and so on. Persistent advantage learning, bootstrapped, dueling, double, deep recurrent, Q-network are implemented using Torch 7 in Atari project.

1.1. Markov Decision Process

Major components of an RL Agent are Policy, Value Function and Model. A state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t] \quad (1)$$

A future is independent of the past given the present. We use state transition matrix P to define transition probabilities from all states s to all successor states s'

$$P_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s] \quad (2)$$

A Markov Reward Process is a tuple $< S, A, P, R, \gamma >$

- S is a finite set of states
- A is a finite set of actions
- P is a state transition probability matrix,

$$P^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$

- R is a reward function,

$$R^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

- γ is a discount factor, $\gamma \in [0, 1]$

Return G_t

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

State value function $v_\pi(s)$

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

Action value function $q_\pi(s, a)$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

Policy π

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

There are also introductions about optimal function, Bellman expectation equation and Bellman optimal equation. Partially Observable Markov Decision Process is an MDP with hidden states.

1.2. Planning by Dynamic Programming

As for policy evaluation, there are iterative policy evaluation and improvement. As for value evaluation, A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$ (the principle of optimality), if and only if

- For any state s' reachable from s
- π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$

Dynamic programming algorithms include asynchronous (in-place, prioritised sweeping, real-time) and synchronous. Extensions include full-width and sample backups. Approximate dynamic programming is used to approximate the value function.
Contraction Mapping Theorem: For any metric space V that is complete (i.e. closed) under an operator $T(v)$, where T is a γ-contraction,

- T converges to a unique fixed point
- At a linear convergence rate of γ

1.3. Model-Free Prediction

Temporal-Difference Learning can learn before knowing the final outcome. It can learn online after every step It can learn from incomplete sequences. It works in continuing (non-terminating) environments .However, TD has low variance, some bias and MC has high variance, zero bias. TD exploits Markov property while MC does not.
The sum of online updates is identical for forward-view and backward-view TD(λ)

$$\sum_{t=1}^{T} \alpha \delta_t E_t(s) = \sum_{t=1}^{T} \alpha (G_t^{\lambda} - V(S_t)) 1(S_t = s) \quad (3)$$

1.4. Model-Free Control

On-policy Monte-Carlo Control ϵ-Greedy Policy improvement.
For any ϵ-greedy policy, the ϵ-greedy policy π' with respect to $q\pi$ is an improvement, $v_{\pi'}(s) \geq v_{\pi}(s)$

$$\begin{aligned} q_{\pi}(s, \pi'(s)) &= \sum_{a \in A} \pi'(a|s) q_{\pi}(s, a) \\ &\geq \sum_{a \in A} \pi(a|s) q_{\pi}(s, a) \\ &= v_{\pi}(s) \end{aligned} \quad (4)$$

Greedy in the Limit with Infinite Exploration (GLIE) Monte-Carlo control converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$
Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC), which including lower variance, online, incomplete sequences. State Action Reward State Action (SARSA) converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$, under the following conditions:

- GLIE sequence of policies $\pi_t(a|s)$
- Robbins-Monro sequence of step-sizes α_t

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Off-policy Learning include importance sampling for Off-policy Monte-Carlo and for Off-policy TD. Q-Learning doesn't require importance sampling. Q-learning control converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$

1.5. Value Function Approximation

To solve large problems, we can adopt value function approximation. In the part of incremental methods, the lecturer introduces gradient descent, linear function approximation, incremental prediction algorithm. As for Batch Reinforcement Learning, gradient descent is simple. But it is not sample efficient. Batch methods seek to find the best fitting value function. Least squares algorithms find parameter vector w minimizing sum-squared error between $\hat{v}(s_t, w)$ and target values v_t^{π}. Deep Q-Networks uses experience replay and fixed Q-targets. Linear Least Squares Prediction Algorithms. Least Squares Policy Iteration Algorithm repeatedly re-evaluates experience D with different policies.

1.6. Policy Gradient

In Policy-Based Reinforcement Learning, a policy is generated directly from the value function. RL includes Value-Based RL and Policy-Based RL. In policy gradient, the score function is $\bigtriangledown_{\theta} log \pi_{\theta}(s, a)$. There are also Softmax Policy and Gaussian Policy. For any differentiable policy $\pi_{\theta}\theta(s, a)$, for any of the policy objective functions $J = J_1$; J_{avR} ; or $\frac{1}{1-\gamma} J_{avV}$ the policy gradient is

$$\bigtriangledown_{\theta} J(\theta) = \mathbb{E}_{\pi\theta}[\bigtriangledown_{\theta} log \pi_{\theta}(s, a) Q^{\pi_{\theta}}(s, a)]$$

The policy gradient has many equivalent forms: REINFORCE, Q Actor-Critic, Advantage Actor-Critic, TD Actor-Critic, TD(λ) Actor-Critic, Natural Actor-Critic, Each leads a stochastic gradient ascent algorithm.

1.7. Experiments

I learned about several Github projects related to Reinforcement Learning, which are shown as below.

Atari

https://github.com/Kaixhin/Atari

deep-rl-tensorflow

TensorFlow implementation of Deep Reinforcement Learning papers.
https://github.com/carpedm20/deep-rl-tensorflow

DeepMind-Atari-Deep-Q-Learner

The original code from the DeepMind article Human-level control through deep reinforcement learning

https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner

dqn

DQN implementation in Keras + TensorFlow + OpenAI Gym

https://github.com/tatsuyaokubo/dqn

1.7.1 Code Analysis

Take the DQN implementation in Keras + TensorFlow + OpenAI Gym as an example, there are DQN and Double DQN implementation.

DQN
Create q network

```
self.s, self.q_values,
q_network = self.build_network()
q_network_weights =
q_network.trainable_weights
```

Create target network

```
self.st, self.target_q_values,
 target_network = self.build_network()
target_network_weights =
 target_network.trainable_weights
```

Define target network update operation

```
self.update_target_network =
 [target_network_weights[i].
 assign(q_network_weights[i])
 for i in range
 (len(target_network_weights))]
```

Define loss and gradient update operation

```
 self.a, self.y, self.loss,
  self.grads_update = self.
  build_training_op(q_network_weights)
```

Initialize target network using tensorflow

```
self.sess.run(self.update_target_network)
```

Initialize target network using tensorflow

```
self.sess.run(self.update_target_network)
```

Function *build_network()* is defined using Keras, including three Convolution2D layers. The first convolution layers include 32 convolution kernels, the size each convolution kernels 8 * 8. The second convolution layers include 64 convolution kernels, the size each convolution kernels $4*4$. The second convolution layers include 64 convolution kernels, the size each convolution kernels 3 * 3. Then, there is an all connection layer, to flatten the output of the last layer from two-dimensional to one dimension. Dense is a hidden layer. s is a placeholder.
Function *build_training_op()* contains two placeholders a, y. Use $tf.one\_hot$ and $tf.reduced\_sum$ to convert action to one hot vector. $tf.reduce\_mean$ is utilized to computes the mean of elements across dimensions of a tensor. $tf.train.RMSPropOptimizer$ is an optimizer that implements the RMSProp algorithm.
Function *get_initial_state, run* and *get_action* are implemented with the help of numpy. Module *random.random()* is imported to generate random number. Anneal epsilon linearly over time. Function *run* first, clips all positive rewards at 1 and all negative rewards at -1, leaving 0 rewards unchanged, then, store transition in replay memory. Start to train the network, update target network and save network. Function *train_network()* establishes a sample random minibatch of transition from replay memory first. Then calculate the loss through tensorflow. Other functions include *setup_summary(), load_network, preprocess and get_action_at_test* . In the *main()*, using gym.make to build the environment. There are two modes: train mode and test mode.

Double DQN In class *Agent*, there are also replay memory, q network, target network, update operation (including target network and loss and gradient). The difference is in function *train_network()*

dqn.py

```
target_q_values_batch =
 self.target_q_values.eval(feed_dict=
 {self.st: np.float32(np.array
 (next_state_batch) / 255.0)})
y_batch = reward_batch + (1 -
terminal_batch) * GAMMA *
np.max(target_q_values_batch, axis=1)
```

ddqn.py

```
next_action_batch =
 np.argmax(self.q_values.eval(feed_dict
 ={self.s: next_state_batch}), axis=1)
target_q_values_batch =
 self.target_q_values.eval(feed_dict
 ={self.st: next_state_batch})
for i in xrange(len(minibatch)):
y_batch.append(reward_batch[i] +
 (1 - terminal_batch[i]) * GAMMA *
  target_q_values_batch[i]
  [next_action_batch[i]])
```

The second instance is *Atari*, which is implemented by Torch 7. In the *Agent.lua* file, they create policy and target networks, boostrapping, recurrency. There are several parameters, including Q-learning variables, validation variables, There are two mode: training mode *training()* and evaluate mode *evaluate()*. There is a function *observe()* to observe the results of the previous transition and chooses the next action to perform. It sets ϵ based on training vs. evaluation mode. Use ensemble policy with bootstrap

heads. In training mode, they store experience tuple parts, collect validation transitions at the start and update target network every τ steps.
The learning process include the following parts: reset gradients $d\theta$, retrieve experience tuples, find the argmax, calculate Q-values, calculate TD-errors $\delta := \Delta Q(s,a) = Y - Q(s,a)$, calculate Advantage Learning update(s), calculate the loss.
Backpropagate is implemented as following:

```
self.policyNet:backward(states, QCurr)
```

Other components include optimising the network parameters θ, reporting absolute network weights and gradients, reporting stats for validation, saving network convolutional filters as images, computing a saliency map.
The third example is *deep-rl-tensorflow*. The Agent class include several components, including $train()$, $play()$, $predict()$, $q\_learning\_minibatch\_test()$ and $update\_target\_q\_network()$.
The network structure is

```
self.double_q = conf.double_q
self.pred_network = pred_network
self.target_network = target_network
self.target_network.create_copy_op
(self.pred_network)
```

In the $deep\_q.py$ file of this example, similar with the DQN implementation mentioned before, class $DeepQ()$ includes an optimizer, update rules for q and loss. There are also two modes to choose: Double Q-learning and Deep Q-learning. In the file $networks\backslash network.py$, the network is constructed using tensorflow. The dueling network contains two input network. There are three modes: simple dueling, max dueling and average dueling.

1.7.2 The Training Process

By using AWS EC2 p2 instance, I try to train Atari and DeepMind-Atari-Deep-Q-Learner on the platform. As for deep-rl-tensorflow, there is a fatal error I will solve later.

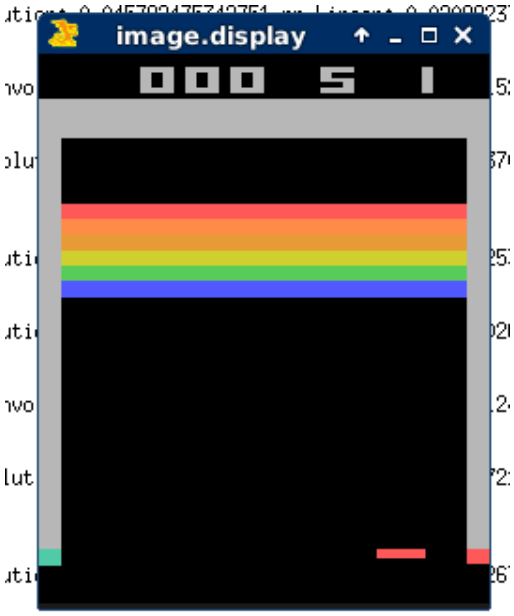

Figure 1. Snapshot of training DeepMind-Atari-Deep-Q-Learner

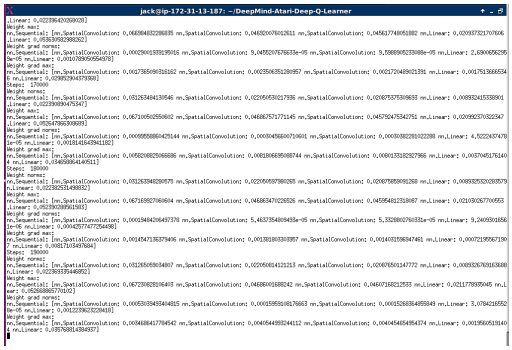

Figure 2. Snapshot of iteration

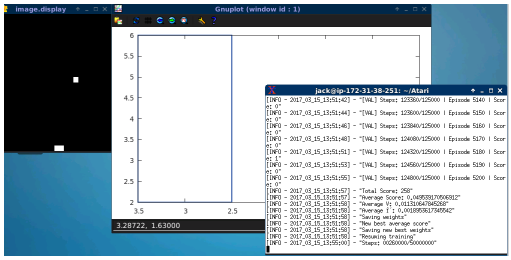

Figure 3. Snapshot of training Atari

References

[1] H. V. Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *Computer Science*, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *Computer Science*, 2013.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, d. D. G. Van, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[5] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, F. F. Li, and A. Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. 2016.