Game Playing with DQfD and DQN

ADL Final Project

Team: Praise the Sun

Department of Computer Science, National Taiwan University

Abstract

Deep reinforcement learning are learning models that combines traditional reinforcement learning algorithms with modern state-of-the-art deep learning models. Currently, many applications, such as robotics and game playing, have reached astonishing performances with these kinds of models. However, these models require huge amount of time/self-generated data to achieve favorable results. What worse is that the performances are sometimes even unstable: the results may differ even given the same environments and model settings. Some recent advanced algorithms may make the training process more efficient, for instance, the *Deep Q-Learning from Demonstrations*(DQfD) proposed by (T. Hester et al., 2017) [1], which introduced the so-called "demonstration data" that can train deep RL models in a supervised manner. In this project, we used some Atari games as the training environment, and we did some experiments on some traditional value-based deep reinforcement learning algorithms and on DQfD, and made some comparisons and analysis on our experiments.

Background

A reinforcement learning model contains an environment and an agent, in which the behavior of the agent can be modeled as a Markov decision process (MDP). MDP can be formally represented by a 5-tuple $(S, A, R(\cdot,\cdot), T(\cdot,\cdot,\cdot), \gamma)$, where S denotes the set of states, A represents the set of all possible actions, $R(s,a)$ is a reward function (given current state s and action a, R returns the reward), $T\ s, a, s' = P(s'|s,a)$ is a transition function, which follows some distribution, and γ is the discount factor. The agent can be regarded as being applying some policy function $\pi(s)$ in the environment to take actions.

The most common reinforcement learning models can be classified as *policy-baced* and *value-based*. The former tries to find a policy function π such that it can be as close with the optimal policy as possible (i.e. $\pi \to \pi^*$), while the latter is to learn a value function $Q^\pi(s,a)$, whose objective is to estimate the expected value given the current state s and action a, and we hope that the value function we are training can be as close to the optimal one as possible (i.e. $Q^\pi(s,a) \to Q^*(s,a)$). Under such circumstance, the optimal policy for the agent will be $\pi^*(s) = \arg\max_{a \in A} Q^*(s,a)$.

Deep Q Learning (DQN) is one of the most common value-based deep learning algorithms. Its optimal value function can be represented as a Bellman equation:

$$Q^*(s,a) = \mathbb{E}\left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')\right]$$

In a DQN model, we'll use a neural network to represent Q^π , and we expect that Q^π will eventually converge to Q^*. In practice, we often use mean squared error (MSE) as the loss function for training, and moreover, for stablility, we also stablize the weights of the target network, and the loss function is as follows:

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{R(s,a) + \gamma \max_{a'} Q(s',a',w^-)}_{\text{target, update slowly}} - \underbrace{Q(s,a,w)}_{\text{online, update quickly}}\right)^2\right]$$

There are some other common DQN models, including Double Q-Learning (DDQN) (H. van Hasselt et al., 2015) [5], Dueling Network (Z. Wang et al., 2015) [6], etc. The former points out that upward bias may occurs when using target network to select the action that has the largest expected value, and thus the (new) loss function is as follows:

$$\mathcal{L}(w) = \mathbb{E}\left[\left(R(s,a) + \gamma Q(s', \underbrace{\arg\max_{a'} Q(s',a',w)}_{\text{online network chooses optimal } a'}, w^-) - Q(s,a,w)\right)^2\right]$$

, while the latter modifies the network as:

$$Q(s,a,w) = \underbrace{V(s,w)}_{\textit{value}\text{, action-independent}} + \underbrace{A(s,a,w)}_{\textit{advantage}\text{, action-dependent}}$$

The basic idea is that some states are just better (regardless of the actions taken), while others not. Hence the expected value can be cosidered to be the sum of the expected value of the state (action-independent) and the added/subtracted value given a certain action.

DQfD : Learning from Demonstrated Data

Additional Settings DQfD is a kind of deep reinforcement learning algorithm that contains elements of supervised learning. Unlike the original DQN, DQfD will perform pre-training on some pre-collected demonstration data prior to the real reinforcement learning process. One can imagine that the agent is similar to an athlete, and the athlete will first train his/her skills from a couch (i.e. expert's demonstration), and then accumulate his/her experiences in matches (i.e. environment), but not taking part in matches in the very beginning. Besides, the loss function of DQfD are also different. First, we must make sure that the model does learn the actions of the demonstrator, so we add the supervised loss, which is shown as follows:

$$\mathcal{L}_E(w) = \max_{a \in A}[Q(s,a,w) + l(a_E, a)] - Q(s, a_E), \text{ where } l(a_E, a) = \begin{cases} 0 & \text{, if } a = a_E \\ k & \text{, otherwise and } k \text{ is a positive number} \end{cases}$$

This will force the expected values of all the other actions to become at least a margin (k) lower than the value of a_E, such that the model will have more tendency to learn the actions from demonstrated data. Other than that, we also impose an L2-regularization loss $(\mathcal{L}_{L2}(w))$ on network weights and bias so as to prevent overfitting on demonstrated data. Finally, according to the original paper, in order to satisfy the Bellman equation, an n-step loss $(\mathcal{L}_n(w))$ is also computed. The total loss is:

$$\mathcal{L}(w) = \underbrace{\mathcal{L}_{DQ}(w)}_{\text{original loss}} + \lambda_n \underbrace{\mathcal{L}_n(w)}_{\text{n-step loss}} + \lambda_E \underbrace{\mathcal{L}_E(w)}_{\text{supervised loss}} + \lambda_{L2} \underbrace{\mathcal{L}_{L2}(w)}_{\text{L2-regularization loss}}$$

Additional Settings After pre-training, as the origin DQN, the model will also save the previous records into a replay buffer. Generally, the size of the memory is finite, and older data will be popped out if the memory is full. Although both demonstrated data and exploration data are stored into a replay buffer, the demonstrated data will never be popped out, while older exploration data will. Futhermore, the original paper also suggests to use the prioritized replay buffer (T. Schaul et al., 2015) [4] to ensure that the demonstrated data have the higher priority to be sampled.

In brief, DQfD, compared to DQN, has the following differences:

- Demonstation
- Pre-training
- Different loss function

Experiment Settings

In our project, we'll use OpenAI's `gym` as our training environment, and we selected three Atari games (Seaquest, Enduro, SpaceInvader) for our experiments. Our model will take 4 most recent preprocessed grayscale images (with size $4 \times 84 \times 84$) as input. Our experiments will compare the following settings:

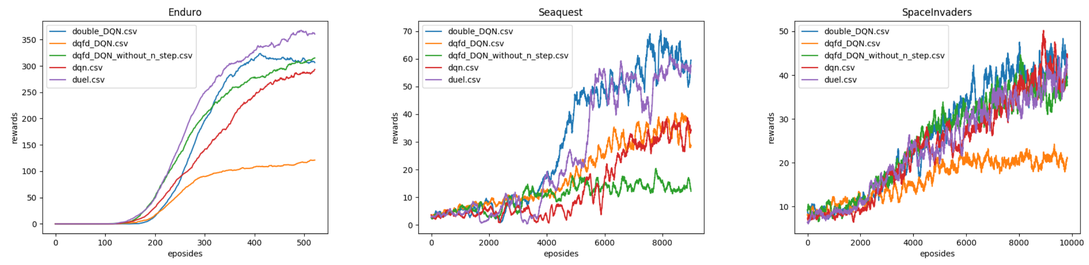
- vanilla DQN
- Double DQN
- Dueling DQN
- DQfD, using n-step loss
- DQfD, without n-step loss

Before training. we first collected some demonstation data from human. Our main approach is to make the game environments in `gym` human-controllable. While playing, it will automatically record the state, action, reward, next state of every step. Since the game speed is quite fast in `gym`, we made the frame rate slower such that human can catch up with the gaming speed. The human players are three of our team members, each of with played a game, and for each game, over 50000 steps of data are collected.

We implemented these networks on `pytorch 0.3.0`. the optimizer is RMSProp, the learning rate is 0.0001, γ is 0.99. The target network will duplicate its weight from online network every 1000 steps, while the online network will update every 4 steps. The size of replay buffer is 10000, and the batch size for sampling replay buffer data is 32.

As for DQfD, the pre-training steps is 350000, the probability of sampling demonstrated data is 0.3. We took 10 for calculating n-step loss, and if n-step loss is applied, λ_n is 1. The weight of supervised loss λ_E is 1, and the margin size (k) is 0.8.

Results & Discussion



We can see that DQfD didn't outperform other settings. Following is some of the reasons.

- We didn't implement Prioritized Experience Replay, and fix 30% of batch size of data sampled from demonstration. This might be the reason why it didn't work.
- According to the figure above, we can found n-step loss didn't help. There might be some implementation issues there.

References

[1] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[2] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *CoRR*, abs/1708.07902, 2017.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[6] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.