Applied Deep Learning: Final Project Report

Game Playing with DQfD and DQN

Instructors: Yun-Nung (Vivian) Chen and Hung-Yi Lee

Team: ***Praise The Sun***

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.

1 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）[3], Dueling Network （Z. Wang et al., 2015）[4], 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.

2 DQfD：Learning from Demonstrated Data

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:

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

DQN Network

type	activation	size	stride	output
input	-	-	-	$4 \times 84 \times 84$
conv	ReLU	8	4	$32 \times 20 \times 20$
conv	ReLU	4	2	$64 \times 9 \times 9$
conv	ReLU	3	1	$64 \times 7 \times 7$
flatten	-	-	-	3136
linear	ReLU	-	-	512
linear	-	-	-	(# of actions)
output	-	-	-	(# of actions)

Dueling DQN Network

type	activation	size	stride	output	remarks
input	-	-	-	$4 \times 84 \times 84$	
conv	ReLU	8	4	$32 \times 20 \times 20$	
conv	ReLU	4	2	$64 \times 9 \times 9$	
conv	ReLU	3	1	$64 \times 7 \times 7$	
flatten	-	-	-	3136	F
(value network)					
linear	ReLU	-	-	512	input: F
linear	-	-	-	1	
expand	-	-	-	(# of actions)	V
(advantage network)					
linear	ReLU	-	-	512	input: F
linear	-	-	-	(# of actions)	
zero-mean	-	-	-	(# of actions)	A
add	-	-	-	(# of actions)	V + A
output	-	-	-	(# of actions)	

Table 1: Network architecture of DQN and Dueling Network. The bottom part of the Dueling network is divided to a value network and an advantage network, both of which take the flattened output as input. The output of value network will be expanded to a vector which has the same dimension as the output of the advantage network; the output of the advantage network will normalize its distribution from (μ, σ^2) to $(0, \sigma^2)$, in case the value network will learn nothing.

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) [2] 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

3 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

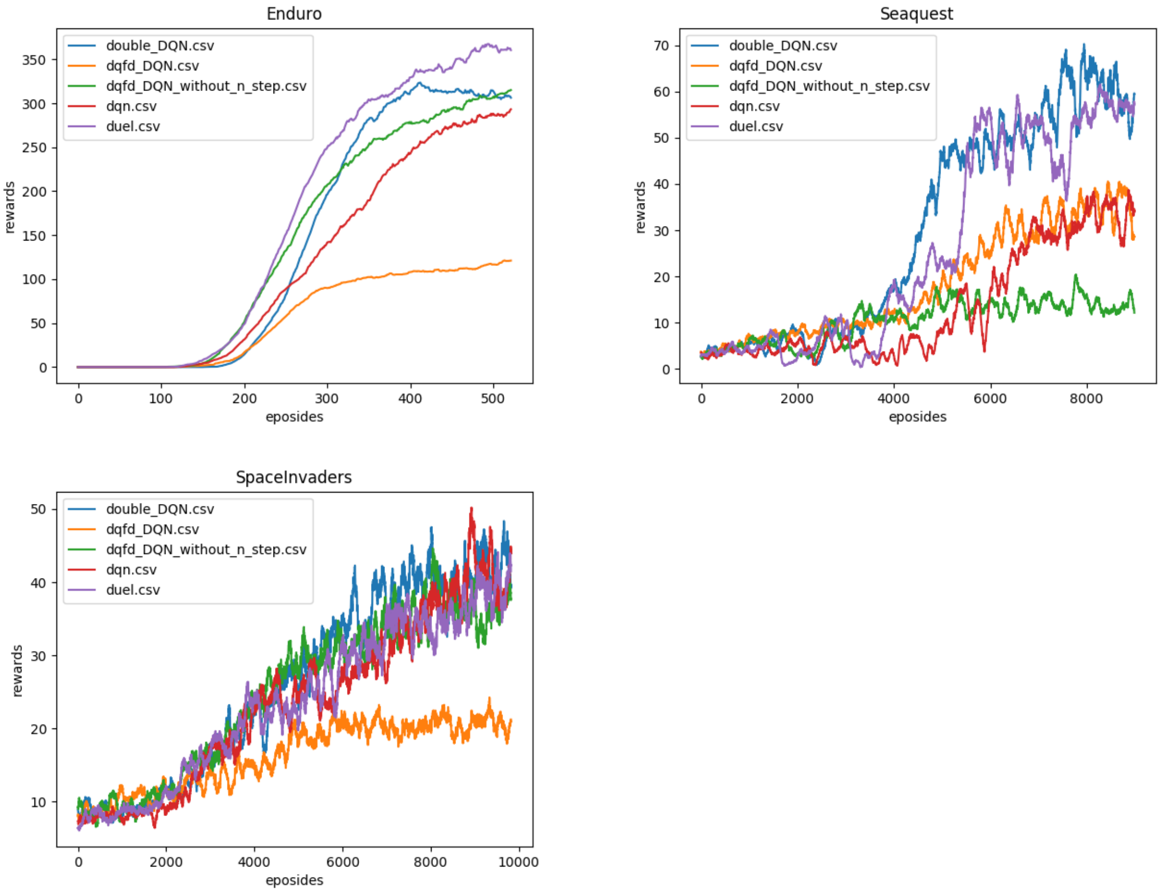

Figure 1: The learning curves of each model. Three games are used in out experiment, where Enduro is a racing game whose actions contains only “left”, “right” and “accelerate”; Seaquest and SpaceInvaders are both shooting games, in which Seaquest is more complicated. Our implementation of DQfD is on the basis of DDQN. In other words, the term $\mathcal{L}_{DQ}(w)$ in the DQfD loss function is same as the loss function of DDQN.

- 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.

Table 1 denotes the network architecture. All models uses the left one except the Dueling DQN. 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.

4 Results & Discussion

Figure 1 shows the learning curves on different games. Apparently, the performances of DQfD is under our expectation. We made some analysis on why it didn't outperform other algorithms:

- Compared to the original descriptions form the paper, we didn't implement the so-called "Priortized Replay Buffer" during sampling on replay memory. That is, the demonstrated data have not higher proirities to be sampled first, which makes the model harder to learn from the demonstrated data.
- Upon pre-training, the model is supposed to perform a non-zero score on each game, but our implementation doesn't. The main reason is that in our implementation, there exists a random exploration schedule from fully random to 5%, which might cause the pre-trained results to be refreshed.
- Our implementation with n-step loss behaves strangely. There might be some implementation issues.

In conclusion, our implementation isn't successful this time. As for the future work, we can start from the possible causes above.

References

[1] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Z. Leibo, and A. Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[2] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[3] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[4] Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.