PYCON KOREA 2018

RL Adventure

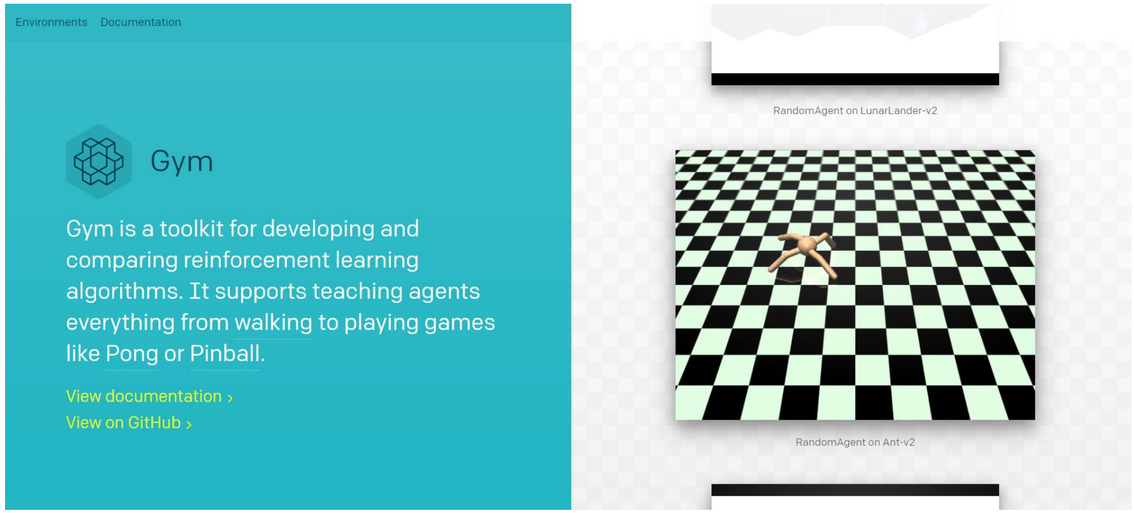
RAINBOW

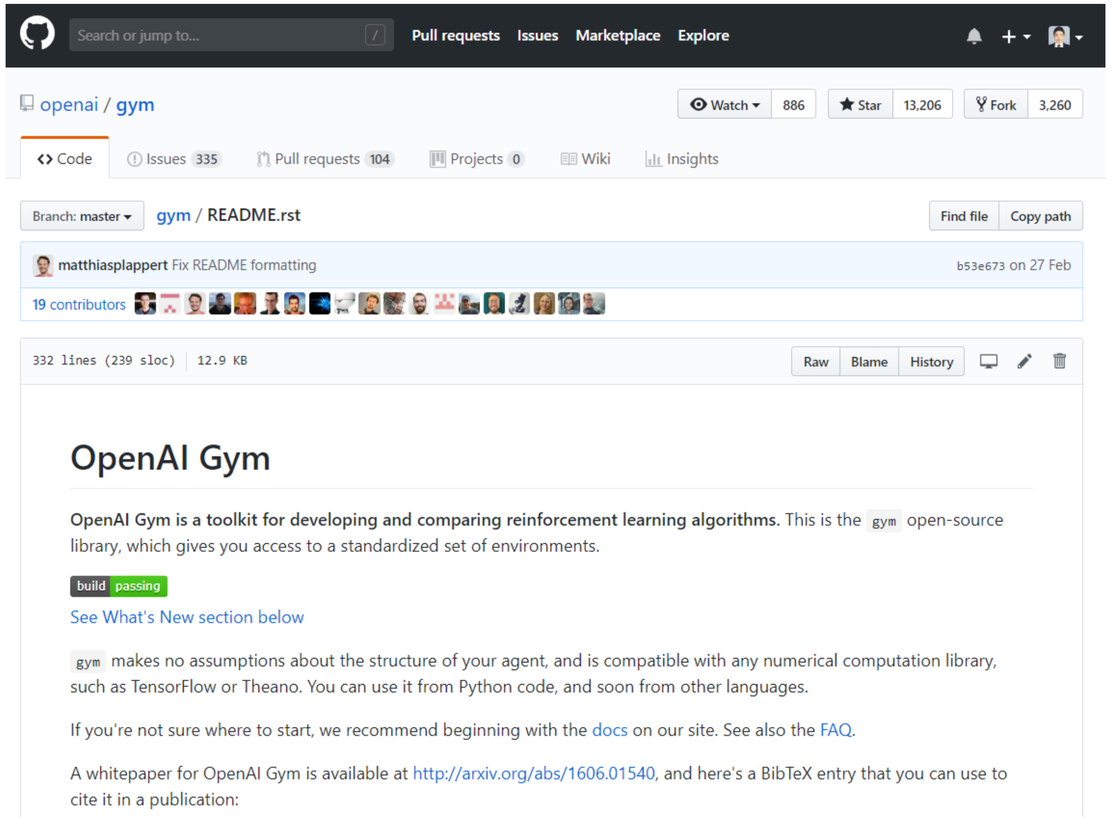
김예찬

INDEX

1. EXPERIMENT ENVIRONMENT

OPENAI GYM





HTTPS://GYM.OPENAI.COM

HTTPS://GITHUB.COM/OPENAI/GYM

2. BEFORE RAINBOW : DOUBLE DQN

DDQN(Double Deep Q-Learning)

Key Point

- DQN의 확장판 : Q-Learning의 over-estimation을 개선한 알고리즘
- 큐함수의 **action selection(행동선택)**과 **estimation**을 나눠서 접근함

Q-function 변경사항

- 기존의 큐함수

$$q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma \max_{a'} q(s',a') - q(s,a))$$

- Double Q-Learning의 큐함수는 2개를 설정

$$q_1(s,a) \leftarrow q_1(s,a) + \alpha(r + \gamma \max_{a'} q_2(s',a') - q_1(s,a))$$
$$q_2(s,a) \leftarrow q_2(s,a) + \alpha(r + \gamma \max_{a'} q_1(s',a') - q_2(s,a))$$

- 2015년도 DQN모형에 target network가 존재하기 때문에 이를 활용함
  - current network로 행동을 선택, target nerwork로 큐함수를 estimate함

$$q_\theta(s,a) \leftarrow q_\theta(s,a) + \alpha(r + \gamma q_{\theta^-}(s', \text{argmax}_{a'} q_\theta(s',a')) - q_\theta(s,a))$$

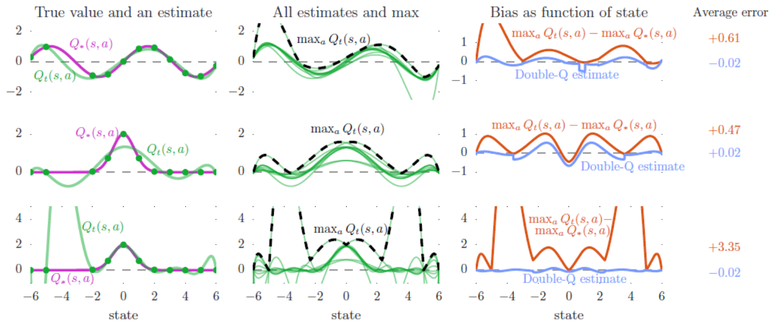


Figure 2: Illustration of overestimations during learning. In each state (x-axis), there are 10 actions. The **left column** shows the true values $V_*(s)$ (purple line). All true action values are defined by $Q_*(s,a) = V_*(s)$. The green line shows estimated values $Q(s,a)$ for one action as a function of state, fitted to the true value at several sampled states (green dots). The **middle column** plots show all the estimated values (green), and the maximum of these values (dashed black). The maximum is higher than the true value (purple, left plot) almost everywhere. The **right column** plots shows the difference in orange. The blue line in the right plots is the estimate used by Double Q-learning with a second set of samples for each state. The blue line is much closer to zero, indicating less bias. The three **rows** correspond to different true functions (left, purple) or capacities of the fitted function (left, green). (Details in the text)

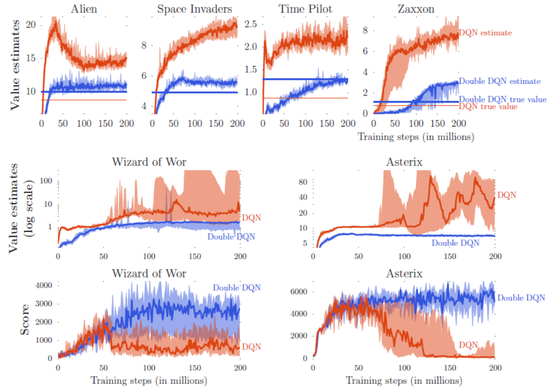


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias. The **middle** row shows the value estimates (in log scale) for two games in which DQN's overoptimism is quite extreme. The **bottom** row shows the detrimental effect of this on the score achieved by the agent as it is evaluated during training: the scores drop when the overestimations begin. Learning with Double DQN is much more stable.

HTTPS://ARXIV.ORG/ABS/1509.06461

2. BEFORE RAINBOW : DUELING DQN

Dueling DQN

Key Point

- 마찬가지로 DQN의 extention : CNN, LSTM, auto-encoder와 같은 model free RL의 **Baseline 아키텍처**를 만들어보자는 아이디어
- **value function**과 **advantage**를 나눠서 접근함
  - 행동선택에 있어서 큐함수만을 보는 것을 advantage로 대체
- advantage : $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s,a)$
  - value function V : 특정 상태의 가치 자체에 집중
  - state-action function Q : 특정 상태에서 행동을 선택된 가치 자체에 집중
  - advantage : **특정 상태에서 평균적으로 얻을 수 있는 가치 대비 각 행동이 얼마나 좋은가**
- DDPG, PER와 함께 사용이 가능함

Q-function

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a'} A(s,a';,\theta,\alpha)\right)$$

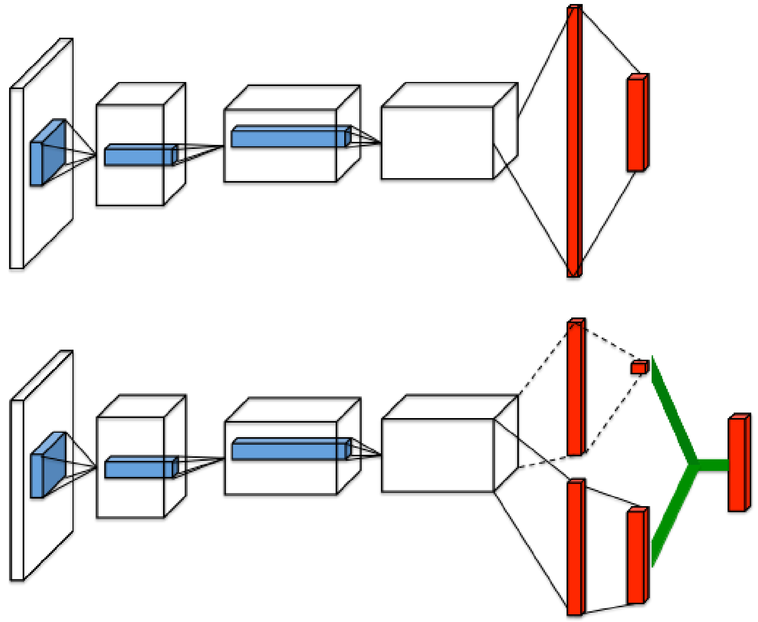


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

HTTPS://ARXIV.ORG/ABS/1511.06581

2. BEFORE RAINBOW : DUELING DQN

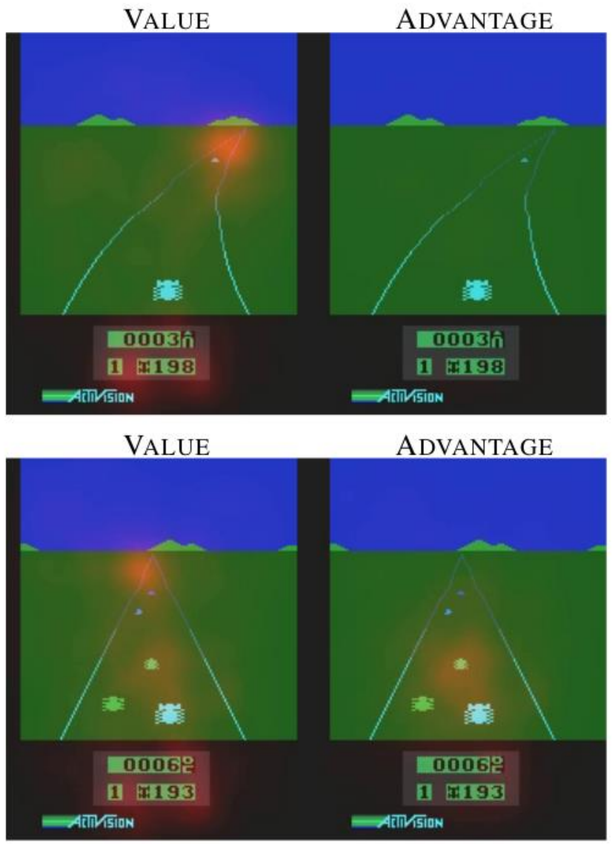


Figure 2. See, attend and drive: Value and advantage saliency maps (red-tinted overlay) on the Atari game Enduro, for a trained dueling architecture. The value stream learns to pay attention to the road. The advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions.

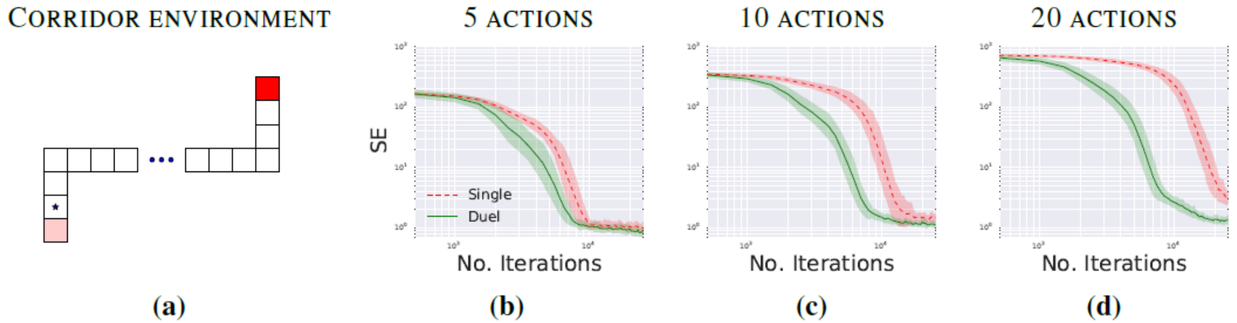


Figure 3. **(a)** The corridor environment. The star marks the starting state. The redness of a state signifies the reward the agent receives upon arrival. The game terminates upon reaching either reward state. The agent's actions are going up, down, left, right and no action. Plots **(b), (c)** and **(d)** shows squared error for policy evaluation with 5, 10, and 20 actions on a log-log scale. The dueling network (Duel) consistently outperforms a conventional single-stream network (Single), with the performance gap increasing with the number of actions.

HTTPS://ARXIV.ORG/ABS/1511.06581

2. BEFORE RAINBOW : MULTI-STEP LEARNING

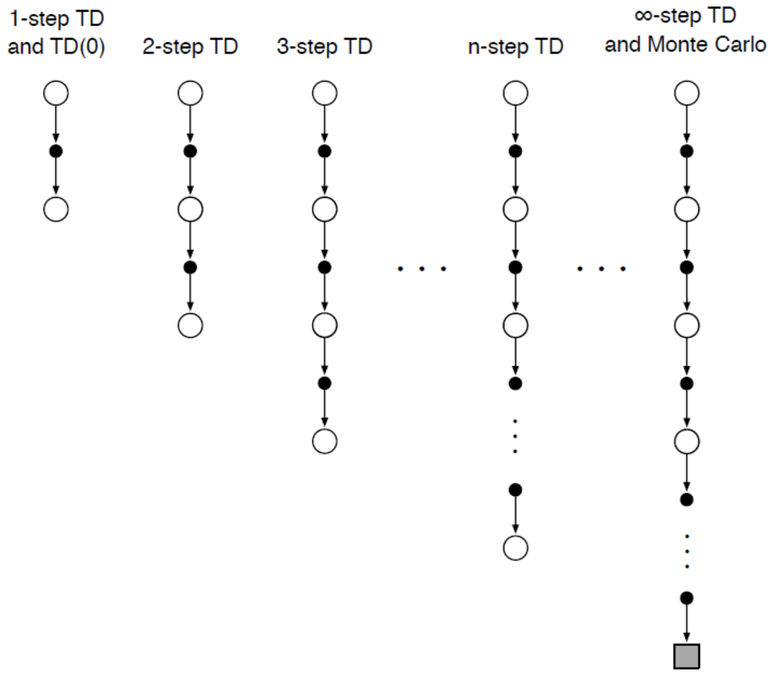


Figure 7.1: The backup diagrams of n-step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

N-Step Bootstrapping

n-step TD Prediction

- State-value learning algorithm for using n-step return:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\left[G_{t:t+n} - V_{t+n-1}(S_t)\right], \quad 0 \le t < T$$

$$V_{t+n}(s) = V_{t+n-1}(s), \forall s \neq S_t, \ T : \text{last time step of episode}$$

- **MC(Monte Carlo) updates' target** : Complete return

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

- **1-step TD updates' target** : first reward + discounted estimated value of next state(One-step return)

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}), \quad V_t : \text{estimate at } t \text{ of } V_{pi}$$

- **n-step updates' target** : For all t, n

$$G_{t:t+n} \doteq \begin{cases} R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), & 0 \le t < T-n \\ G_t, & t \ge T-n \end{cases}$$

In Rainbow Notation

- n-step return from a given state S_t

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

- multi-step(n-step) td loss in DQN

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2$$

2. BEFORE RAINBOW : PER

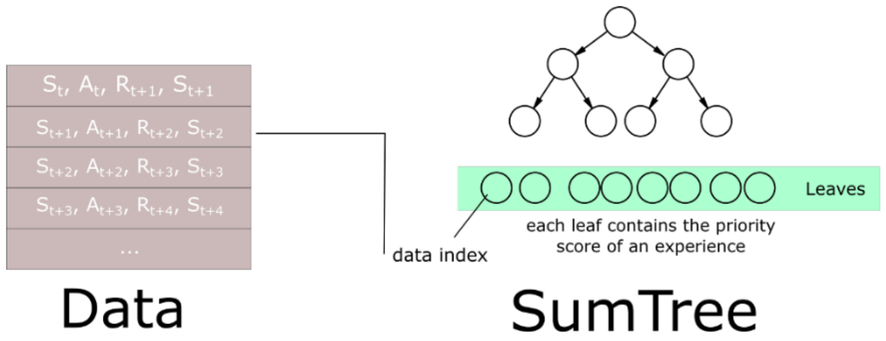
PER(Prioritized Experience Replay)

Key Point

- 학습에 도움이 되는 샘플들을 사용하자 : **Stochastic Sampling**
- Stochastic Sampling 방법은 **2가지** : Proportional, Rank-based
- DQN의 **TD error**값(δ)을 활용

$$\delta = r + \gamma \max_{a'} q(s', a') - q(s, a)$$

- δ을 replay memory에 추가하는 형태를 지님
- Importance Sampling(IS)
- priority queues, SumTree ...



Prioritization ¶

1. Probability of sampling transition i :

$$P(i) = \frac{p_i^\alpha}{\sum_k P_k^\alpha}$$

- α : replay buffer에서 샘플링할 때 randomness를 다시 부여하기 위한 하이퍼파라미터
  - $\alpha = 0$: pure uniform randomness
  - $\alpha = 1$: select the experience with the highest priorities
- $p_i > 0$

2. Rank-based Prioritization
   Priority of transition i :

$$p\left(\frac{1}{rank(i)}\right)$$

- rank(i) : rank of transision i when replay memory is sorted according to δ_i
- P becomes a power-law distribution with exponent α

3. Proportional Prioritization
   Priority of transition i :

$$p(|\delta| + \epsilon)^\alpha \quad , 0 < \epsilon << 1$$

- ϵ을 추가하여 낮은 priority를 갖는 sample들이 뽑힐 확률이 0이 되지 않도록 설정

4. Importance Sampling(IS) : bias를 막아주기 위해 사용

$$\left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b$$

- N : Replay Buffer Size
- $P(i)$:Sampling probability
- b : IS의 w를 control하기 위해 설정. 0에서부터 1까지 decay

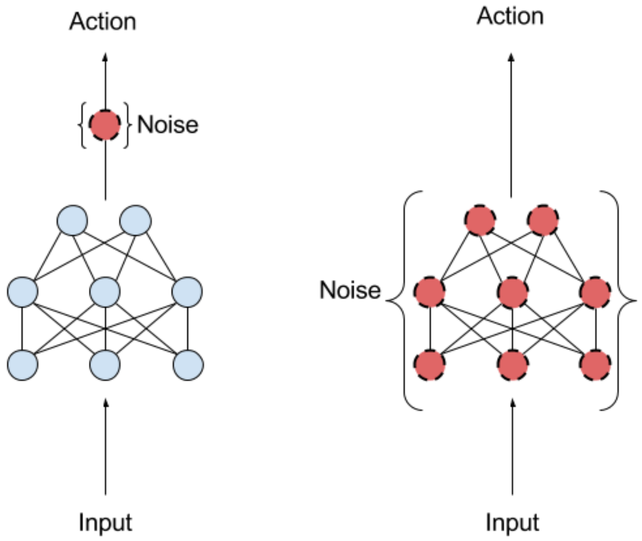
HTTPS://ARXIV.ORG/ABS/1511.05952

2. BEFORE RAINBOW : NOISY NETWORK

Noisy DQN

Key Point

- epsilon-greedy, entropy regularization과 같은 방법을 활용하여 exploration/exploitation의 trade-off를 고려하는 방식이 아니다
- action space에 noise를 설정하는 것이 아니라 Network 자체에 noise를 주입
- Dueling Network의 Fully Conntected Layer를 대체함
- noisy distribution으로 부터 sampling하여 **perturbation**, 파라미터에 대한 평균과 분산을 학습
- 방법은 2가지
  - **A. Independent Gaussian Noise**
  - **B. Factorised Gaussian Noise**



Initialisation : Independent vs Factorised

1. Independent Gaussian Noise

- $\mu_{i,j} : u\left[-\sqrt{\frac{3}{p}}, +\sqrt{\frac{3}{p}} \right]$
- p : number of corresponding linear layer
- $\sigma_{i,j}$: 0.017 for all parameters
- ϵ : unit gaussian distribution

2. Factorised Gaussian Noise

- $\mu_{i,j} : u\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}} \right]$
- p : number of corresponding linear layer
- $\sigma_{i,j} : \frac{\sigma_0}{\sqrt{p}}$, σ_0 is 0.5
- $\epsilon : \epsilon^w_{i,j} = f(\epsilon_i) f(\epsilon_j),\ \epsilon^b_j = f(\epsilon_j)$
- f : real-valued function, $f(x) = sign(x)\sqrt{|x|}$

HTTPS://ARXIV.ORG/ABS/1706.10295

2. BEFORE RAINBOW : NOISY NETWORK

$$y = wx + b$$
$$y = f_\theta(x)$$

$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b$$

$$\theta = \mu + \sigma \odot \epsilon$$

where $\zeta = (\mu, \sigma)$, learnable parameters
ϵ, vector of zero-mean noise with statics

In this Appendix we provide a graphical representation of noisy layer.

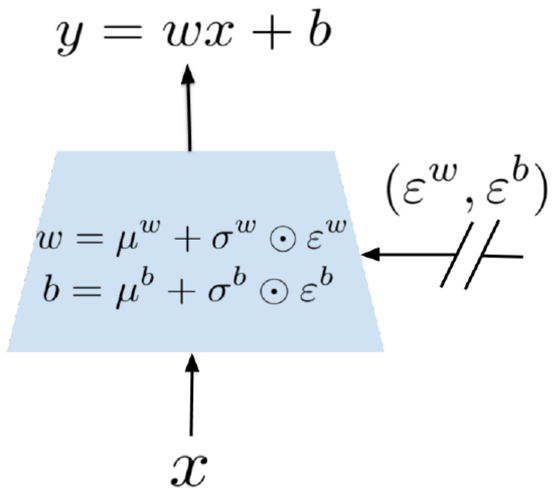


Figure 4: Graphical representation of a noisy linear layer. The parameters μ^w, μ^b, σ^w and σ^b are the learnables of the network whereas ε^w and ε^b are noise variables which can be chosen in factorised or non-factorised fashion. The noisy layer functions similarly to the standard fully connected linear layer. The main difference is that in the noisy layer both the weights vector and the bias is perturbed by some parametric zero-mean noise, that is, the noisy weights and the noisy bias can be expressed as $w = \mu^w + \sigma^w \odot \varepsilon^w$ and $b = \mu^b + \sigma^b \odot \varepsilon^b$, respectively. The output of the noisy layer is then simply obtained as $y = wx + b$.

HTTPS://ARXIV.ORG/ABS/1706.10295

2. BEFORE RAINBOW : CATEGORICAL DQN(C51)

Categorical DQN

Key Point

- Q-values의 평균은 stable하지 않음, 환경의 uncertainty를 극복해보자
- 반환값을 scalar value 대신 distribution of values로 보자. 즉, random variable으로 접근
- Distributional Bellman operator for distribution $Z(x, a)$
  - $T^{\pi}Z(x,a) = R(x,a) + \gamma P^{\pi}Z(x,a)$
- Distribution간의 거리를 계산하는 방법은 KL-divergence
  - $D_{KL}(P\|Q) = \int_{-\infty}^{\infty} p(x)\log\frac{p(x)}{p(y)}dx$
- KL-divergence가 발산할 수 있기 때문에 Wasserstein metric을 사용하며, Wasserstein distance를 minimize함
- C51은 각 행동마다 51개의 값을 softmax로 리턴함

HTTPS://ARXIV.ORG/PDF/1707.06887.PDF

2. BEFORE RAINBOW : CATEGORICAL DQN(C51)

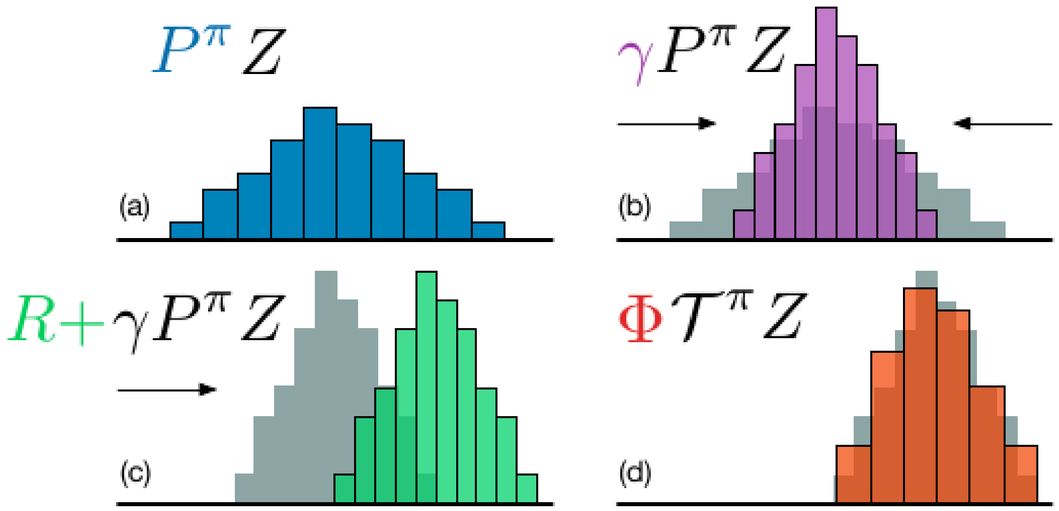


Figure 1. A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 4).

Distributional Categorical Algorithm (C51) Update

- The i^{th} categorical component of the projected update is calculated by distributing probability p to immediate neighbours:

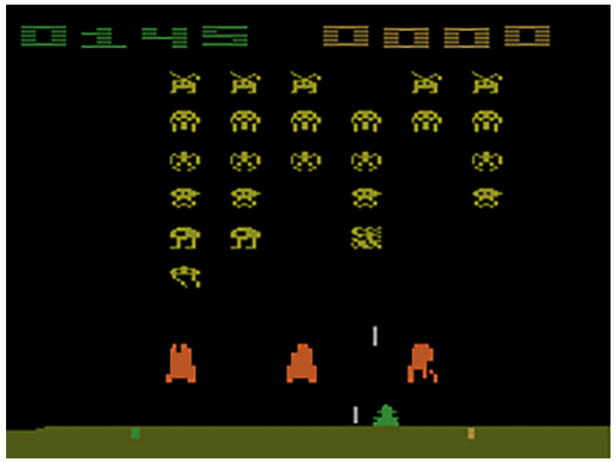
$$(\Phi \hat{T} Z_\theta(x,a))_i = \sum_{j=0}^{N-1} clip\Big[1 - \frac{|clip[\hat{T}z_j]_{V_{MIN}}^{V_{MAX}} - z_i|}{\triangle z}\Big] p_j(x', \pi(x')),$$

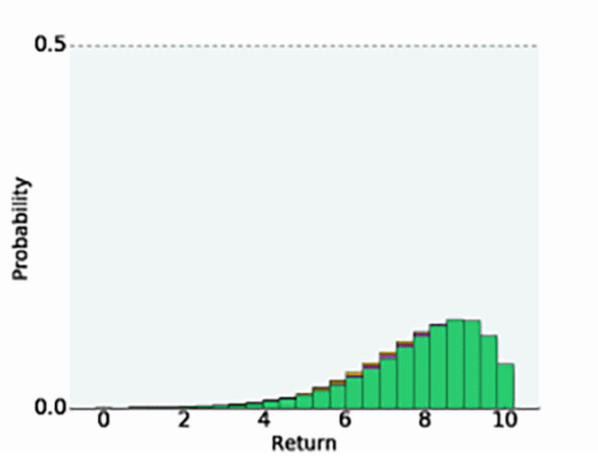
$$\text{where } z_i = V_{MIN} + i\triangle z, \quad \triangle = \frac{V_{MAX} - V_{MIN}}{N-1},$$

$$\hat{T}z_j = r + \gamma z_j, \quad p_j \text{ is probability of component j}$$

Loss function

$$L_{x,a}(\theta) = D_{KL}(\Phi \hat{T} Z_{\tilde{\theta}}(x,a) \| Z_\theta(x,a))$$





HTTPS://ARXIV.ORG/PDF/1707.06887.PDF

3. RAINBOW

RAINBOW

3. RAINBOW

RAINBOW

DDQN(Double Deep Q-Learning)

+

Dueling DQN

+

Multi-Step TD(Temporal Difference)

+

PER(Prioritized Experience Replay)

+

Noisy Network

+

Categorical DQN(C51)

3. RAINBOW

Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel DeepMind
Joseph Modayil DeepMind
Hado van Hasselt DeepMind
Tom Schaul DeepMind
Georg Ostrovski DeepMind
Will Dabney DeepMind
Dan Horgan DeepMind
Bilal Piot DeepMind
Mohammad Azar DeepMind
David Silver DeepMind

Abstract

The deep reinforcement learning community has made several independent improvements to the DQN algorithm. However, it is unclear which of these extensions are complementary and can be fruitfully combined. This paper examines six extensions to the DQN algorithm and empirically studies their combination. Our experiments show that the combination provides state-of-the-art performance on the Atari 2600 benchmark, both in terms of data efficiency and final performance. We also provide results from a detailed ablation study that shows the contribution of each component to overall performance.

Introduction

The many recent successes in scaling reinforcement learning (RL) to complex sequential decision-making problems were kick-started by the Deep Q-Networks algorithm (DQN; Mnih et al. 2013, 2015). Its combination of Q-learning with convolutional neural networks and experience replay enabled it to learn, from raw pixels, how to play many Atari games at human-level performance. Since then, many extensions have been proposed that enhance its speed or stability.

Double DQN (DDQN; van Hasselt, Guez, and Silver 2016) addresses an overestimation bias of Q-learning (van Hasselt 2010), by decoupling selection and evaluation of the bootstrap action. Prioritized experience replay (Schaul et al. 2015) improves data efficiency, by replaying more often transitions from which there is more to learn. The dueling network architecture (Wang et al. 2016) helps to generalize across actions by separately representing state values and action advantages. Learning from multi-step bootstrap targets (Sutton 1988; Sutton and Barto 1998), as used in A3C (Mnih et al. 2016), shifts the bias-variance trade-off and helps to propagate newly observed rewards faster to earlier visited states. Distributional Q-learning (Bellemare, Dabney, and Munos 2017) learns a categorical distribution of discounted returns, instead of estimating the mean. Noisy DQN (Fortunato et al. 2017) uses stochastic network layers for exploration. This list is, of course, far from exhaustive

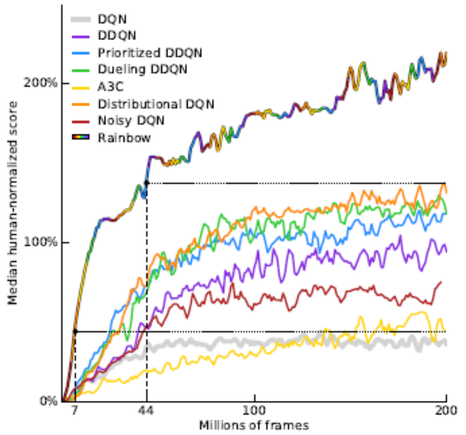


Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

radically different issues, and since they build on a shared framework, they could plausibly be combined. In some cases this has been done: Prioritized DDQN and Dueling DDQN both use double Q-learning, and Dueling DDQN was also combined with prioritized experience replay. In this paper we propose to study an agent that combines all the aforementioned ingredients. We show how these different ideas can be integrated, and that they are indeed largely complementary. In fact, their combination results in new state-

3. RAINBOW

HYPERPARAMETERS

Parameter	Value
Min history to start learning	80K frames
Adam learning rate	0.0000625
Exploration ϵ	0.0
Noisy Nets σ_0	0.5
Target Network Period	32K frames
Adam ϵ	1.5×10^{-4}
Prioritization type	proportional
Prioritization exponent ω	0.5
Prioritization importance sampling β	$0.4 \rightarrow 1.0$
Multi-step returns n	3
Distributional atoms	51
Distributional min/max values	$[-10, 10]$

Table 1: Rainbow hyper-parameters

Hyper-parameter	value
Grey-scaling	True
Observation down-sampling	(84, 84)
Frames stacked	4
Action repetitions	4
Reward clipping	[-1, 1]
Terminal on loss of life	True
Max frames per episode	108K

Table 3: **Preprocessing**: the values of these hyper-parameters are the same used by DQN and its variants. They are here listed for completeness. Observations are grey-scaled and rescaled to 84×84 pixels. 4 consecutive frames are concatenated as each state's representation. Each action selected by the agent is repeated for 4 times. Rewards are clipped between $-1, +1$. In games where the player has multiple lives, transitions associated to the loss of a life are considered terminal. All episodes are capped after 108K frames.

Hyper-parameter	value
Q network: channels	32, 64, 64
Q network: filter size	$8 \times 8, 4 \times 4, 3 \times 3$
Q network: stride	4, 2, 1
Q network: hidden units	512
Q network: output units	Number of actions
Discount factor	0.99
Memory size	1M transitions
Replay period	every 4 agent steps
Minibatch size	32

Table 4: **Additional hyper-parameters**: the values of these hyper-parameters are the same used by DQN and it's variants. The network has 3 convolutional layers: with 32, 64 and 64 channels. The layers use 8×8, 4×4, 3×3 filters with strides of 4, 2, 1, respectively. The value and advantage streams of the dueling architecture have both a hidden layer with 512 units. The output layer of the network has a number of units equal to the number of actions available in the game. We use a discount factor of 0.99, which is set to 0 on terminal transitions. We perform a learning update every 4 agent steps, using mini-batches of 32 transitions.

3. RAINBOW

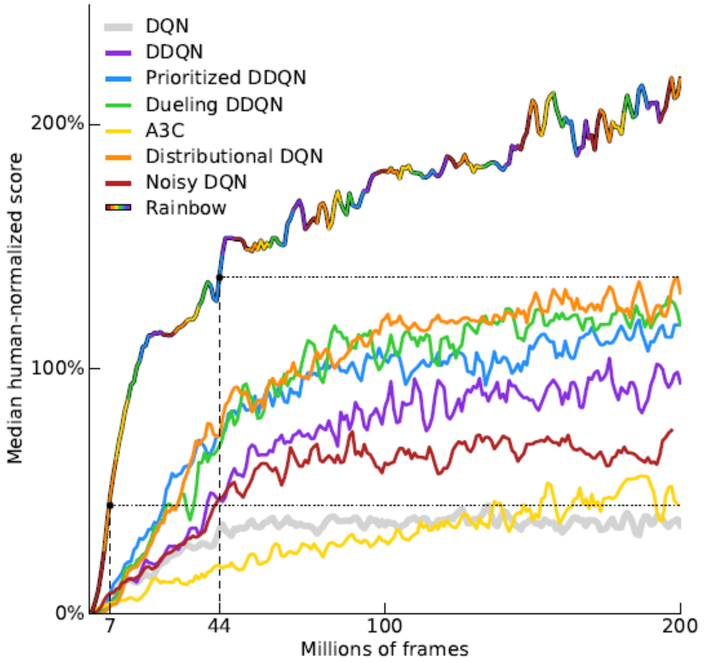


Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

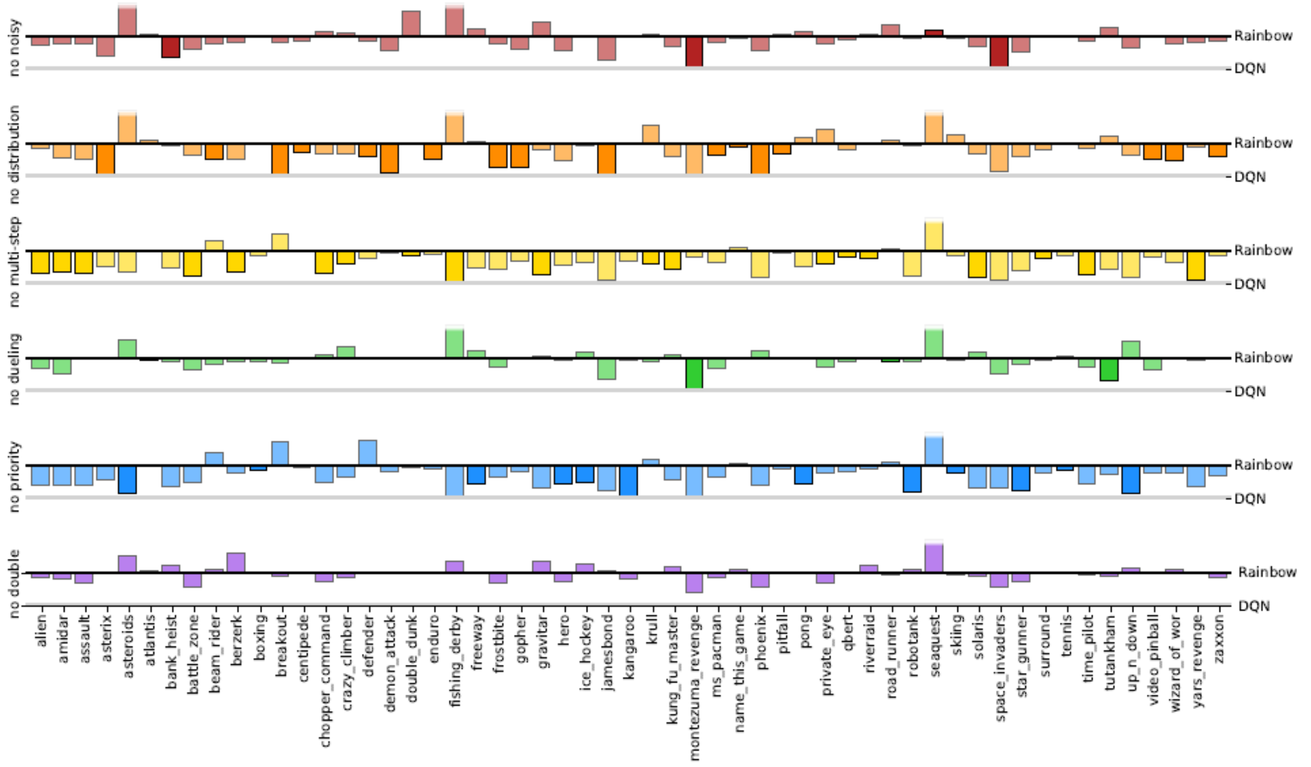


Figure 4: **Performance drops of ablation agents** on all 57 Atari games. Performance is the area under the learning curve, normalized relative to the Rainbow agent and DQN. Two games where DQN outperforms Rainbow are omitted. The ablation leading to the strongest drop is highlighted for each game. The removal of either prioritization or multi-step learning reduces performance across most games, but the contribution of each component varies substantially per game.

3. RAINBOW

Game	DQN	A3C	DDQN	Prior. DDQN	Duel. DDQN	Distrib. DQN	Noisy DQN	Rainbow
alien	634.0	518.4	1033.4	900.5	1,486.5	1,997.5	533.3	**6,022.9**
amidar	178.4	**263.9**	169.1	218.4	172.7	237.7	148.0	202.8
assault	3489.3	5474.9	6060.8	7,748.5	3,994.8	5,101.3	5,124.3	**14,491.7**
asterix	3170.5	22140.5	16837.0	31,907.5	15,840.0	**395,599.5**	8,277.3	280,114.0
asteroids	1458.7	**4474.5**	1193.2	1,654.0	2,035.4	2,071.7	4,078.1	2,249.4
atlantis	292491.0	**911,091.0**	319688.0	593,642.0	445,360.0	289,803.0	303,666.5	814,684.0
bank_heist	312.7	970.1	886.0	816.8	**1,129.3**	835.6	955.0	826.0
battle_zone	23750.0	12950.0	24740.0	29,100.0	31,320.0	32,250.0	26,985.0	**52,040.0**
beam_rider	9743.2	22707.9	17417.2	**26,172.7**	14,591.3	15,002.4	15,241.5	21,768.5
berzerk	493.4	817.9	1011.1	1,165.6	910.6	1,000.0	670.8	**1,793.4**
bowling	56.5	35.1	69.6	65.8	65.7	76.8	**79.3**	39.4
boxing	70.3	59.8	73.5	68.6	**77.3**	62.1	66.3	54.9
breakout	354.5	**681.9**	368.9	371.6	411.6	548.7	423.3	379.5
centipede	3973.9	3755.8	3853.5	3,421.9	4,881.0	**7,476.9**	4,214.4	7,160.9
chopper_command	5017.0	7021.0	3495.0	6,604.0	3,784.0	9,600.5	8,778.5	**10,916.0**
crazy_climber	98128.0	112646.0	113782.0	131,086.0	124,566.0	154,416.5	98,576.5	**143,962.0**
defender	15917.5	56533.0	27510.0	21,093.5	33,996.0	32,246.0	18,037.5	**47,671.3**
demon_attack	12550.7	**113,308.4**	69803.4	73,185.8	56,322.8	109,856.6	25,207.8	109,670.7
double_dunk	-6.0	-0.1	-0.3	**2.7**	-0.8	-3.7	-1.0	-0.6
enduro	626.7	-82.5	1216.6	1,884.4	2,077.4	**2,133.4**	1,021.5	2,061.1
fishing_derby	-1.6	18.8	3.2	9.2	-4.1	-4.9	-3.7	**22.6**
freeway	26.9	0.1	28.8	27.9	0.2	28.8	27.1	**29.1**
frostbite	496.1	190.5	1448.1	2,930.2	2,332.4	2,813.9	418.8	**4,141.1**
gopher	8190.4	10022.8	15253.0	57,783.8	20,051.4	27,778.3	13,131.0	**72,595.7**
gravitar	298.0	303.5	200.5	218.0	297.0	422.0	250.5	**567.5**
hero	14992.9	32464.1	14892.5	20,506.4	15,207.9	28,554.2	2,454.2	**50,496.8**
ice_hockey	-1.6	-2.8	-2.5	-1.0	-1.3	**-0.1**	-2.4	-0.7
kangaroo	4496.0	94.0	**11204.0**	10,241.0	10,334.0	9,555.5	7,465.0	10,841.0
krull	6206.0	5560.0	6796.1	7,406.5	**8,051.6**	6,757.8	6,833.5	6,715.5
kung_fu_master	20882.0	28819.0	30207.0	31,244.0	24,288.0	**33,890.0**	27,921.0	28,999.8
montezuma_revenge	47.0	67.0	42.0	13.0	22.0	130.0	55.0	**154.0**
ms_pacman	1092.3	653.7	1241.3	1,824.6	2,250.6	2,064.1	1,012.1	**2,570.2**
name_this_game	6738.8	10476.1	8960.3	**11,836.1**	11,185.1	11,382.3	7,186.4	11,686.5
phoenix	7484.8	52894.1	12366.5	27,430.1	20,410.5	31,358.3	15,505.0	**103,061.6**
pitfall	-113.2	-78.5	-186.7	**-14.8**	-46.9	-342.8	-154.4	-37.6
pong	18.0	5.6	**19.1**	18.9	18.8	18.9	18.0	19.0
private_eye	207.9	206.9	-575.5	179.0	292.6	5,717.5	**5,955.4**	1,704.4
qbert	9271.5	15148.8	11020.8	11,277.0	14,175.8	15,035.9	9,176.6	**18,397.6**
road_runner	35215.0	34216.0	43156.0	56,990.0	**58,549.0**	56,086.0	35,376.5	54,261.0
robotank	58.7	32.8	59.1	55.4	**62.0**	49.8	50.9	55.2
seaquest	4216.7	2355.4	14498.0	**39,096.7**	37,361.6	3,275.4	2,353.1	19,176.0
skiing	-12142.1	-10911.1	-11490.4	**-10,852.8**	-11,928.0	-13,247.7	-13,905.9	-11,685.8
solaris	1295.4	1956.0	810.0	2,238.2	1,768.4	2,530.2	2,608.2	**2,860.7**
space_invaders	1293.8	**15,730.5**	2628.7	9,063.0	5,993.1	6,368.6	1,697.2	12,629.0
star_gunner	52970.0	**138218.0**	58365.0	51,959.0	90,804.0	67,054.5	31,864.5	123,853.0
surround	-6.0	-9.7	1.9	-0.9	4.0	4.5	-3.1	**7.0**
tennis	11.1	-6.3	-7.8	-2.0	4.4	**22.6**	-2.1	-2.2
time_pilot	4786.0	**12,679.0**	6608.0	7,448.0	6,601.0	7,684.5	5,311.0	11,190.5
tutankham	45.6	**156.3**	92.2	33.6	48.0	124.3	123.3	126.9
venture	136.0	23.0	21.0	244.0	200.0	**462.0**	10.5	45.0
video_pinball	154414.1	331628.1	367823.7	374,886.9	110,976.2	455,052.7	241,851.7	**506,817.2**
wizard_of_wor	1609.0	**17,244.0**	6201.0	7,451.0	7,054.0	11,824.5	4,796.5	14,631.5
yars_revenge	4577.5	7157.5	6270.6	5,965.1	25,976.5	8,267.7	5,487.3	**93,007.9**
zaxxon	4412.0	**24,622.0**	8593.0	9,501.0	10,164.0	15,130.0	7,650.5	19,658.0

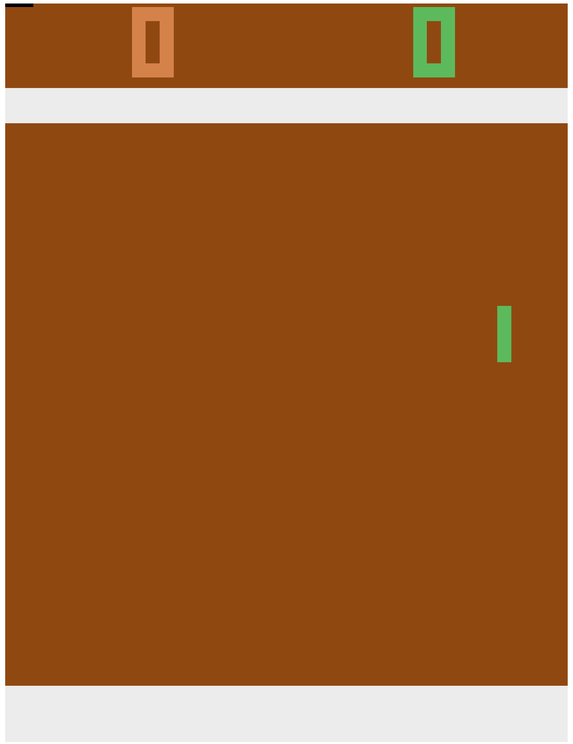
Table 5: **Human Starts** evaluation regime: Raw scores across all games, averaged over 200 testing episodes, from the agent snapshot that obtained the highest score during training. We report the published scores for DQN, A3C, DDQN, Dueling DDQN, and Prioritized DDQN. For Distributional DQN and Rainbow we report our own evaluations of the agents.

Game	DQN	DDQN	Prior. DDQN	Duel. DDQN	Distrib. DQN	Noisy DQN	Rainbow
alien	1620.0	3747.7	6,648.6	4,461.4	4,055.8	2,394.9	**9,491.7**
amidar	978.0	1793.3	2,051.8	2,354.5	1,267.9	1,608.0	**5,131.2**
assault	4280.0	5393.2	7,965.7	4,621.0	5,909.0	5,198.6	**14,198.5**
asterix	4359.0	17356.5	41,268.0	28,188.0	400,529.5	12,403.8	**428,200.3**
asteroids	1364.5	734.7	1,699.3	2,837.7	2,354.7	**4,814.1**	2,712.8
atlantis	279987.0	106056.0	427,658.0	382,572.0	273,895.0	329,010.0	**826,659.5**
bank_heist	455.0	1030.6	1,126.8	**1,611.9**	1,056.7	1,323.0	1,358.0
battle_zone	29900.0	31700.0	38,130.0	37,150.0	41,145.0	32,050.0	**62,010.0**
beam_rider	8627.5	13772.8	**22,430.7**	12,164.0	13,213.4	12,534.0	16,850.2
berzerk	585.6	1225.4	1,614.2	1,472.6	1,421.8	837.3	**2,545.6**
bowling	50.4	68.1	62.6	65.5	**74.1**	77.3	30.0
boxing	88.0	91.6	98.8	99.4	98.1	83.3	**99.6**
breakout	385.5	418.5	381.5	345.3	**612.5**	459.1	417.5
centipede	4657.7	5409.4	5,175.4	7,561.4	**9,015.5**	4,355.8	8,167.3
chopper_command	6126.0	5809.0	5,135.0	11,215.0	13,136.0	9,519.0	**16,654.0**
crazy_climber	110763.0	117282.0	**183,137.0**	143,570.0	178,355.0	118,768.0	168,788.5
defender	23633.0	35338.5	24,162.5	42,214.0	37,896.8	23,083.0	**55,105.0**
demon_attack	12149.4	58044.2	70,171.8	60,813.3	110,626.5	24,950.1	**111,185.2**
double_dunk	-6.6	-5.5	**4.8**	0.1	-3.8	-1.8	-0.3
enduro	729.0	1211.8	2,155.0	2,258.2	**2,259.3**	1,129.2	2,125.9
fishing_derby	-4.9	15.5	30.2	**46.4**	9.1	7.7	31.3
freeway	30.8	33.3	32.9	0.0	33.6	32.0	**34.0**
frostbite	797.4	1683.3	3,421.6	4,672.8	3,938.2	583.6	**9,590.5**
gopher	8777.4	14840.8	49,097.4	15,718.4	28,841.0	15,107.9	**70,354.6**
gravitar	473.0	412.0	330.5	588.0	681.0	443.5	**1,419.3**
hero	20437.8	20130.2	27,153.9	20,818.2	33,860.9	5,053.1	**55,887.4**
ice_hockey	-1.9	-2.7	0.3	0.5	**1.3**	-2.1	1.1
kangaroo	7259.0	12992.0	14,492.0	**14,854.0**	12,909.0	12,117.0	14,637.5
krull	8422.3	7920.5	10,263.1	**11,451.9**	9,885.9	9,061.9	8,741.5
kung_fu_master	26059.0	29710.0	43,470.0	34,294.0	43,009.0	34,099.0	**52,181.0**
montezuma_revenge	0.0	0.0	0.0	0.0	367.0	0.0	**384.0**
ms_pacman	3085.6	2711.4	4,751.2	**6,283.5**	3,769.2	2,501.6	5,380.4
name_this_game	8207.8	10616.0	**13,439.4**	11,971.1	12,983.6	8,332.4	13,136.0
phoenix	8485.2	12252.5	32,808.3	23,092.2	34,775.0	16,974.3	**108,528.6**
pitfall	-286.1	-29.9	0.0	0.0	-2.1	-18.2	0.0
pong	19.5	20.9	20.7	**21.0**	20.8	**21.0**	20.9
private_eye	146.7	129.7	200.0	103.0	**15,172.9**	3,966.0	4,234.0
qbert	13117.3	15088.5	18,802.8	19,220.3	16,956.0	15,276.3	**33,817.5**
road_runner	39544.0	44127.0	62,785.0	**69,524.0**	63,366.0	41,681.0	62,041.0
robotank	63.9	65.1	58.6	**65.3**	54.2	53.5	61.4
seaquest	5860.6	16452.7	44,417.4	**50,254.2**	4,754.4	2,495.4	15,898.9
skiing	-13062.3	-9021.8	-9,900.5	**-8,857.4**	-14,959.8	-16,307.3	-12,957.8
solaris	3482.8	3067.8	1,710.8	2,250.8	**5,643.1**	3,204.5	3,560.3
space_invaders	1692.3	2525.5	7,696.9	6,427.3	6,869.1	2,145.5	**18,789.0**
star_gunner	54282.0	60142.0	56,641.0	89,238.0	69,306.5	34,504.5	**127,029.0**
surround	-5.6	-2.9	2.1	4.4	6.2	-3.3	**9.7**
tennis	12.2	-22.8	0.0	5.1	**23.6**	0.0	-0.0
time_pilot	4870.0	8339.0	11,448.0	11,666.0	7,875.0	6,157.0	**12,926.0**
tutankham	68.1	218.4	87.2	211.4	**249.4**	231.6	241.0
venture	163.0	98.0	863.0	497.0	**1,107.0**	0.0	5.5
video_pinball	196760.4	309941.9	406,420.4	98,209.5	478,646.7	270,444.6	**533,936.5**
wizard_of_wor	2704.0	7492.0	10,373.0	7,855.0	15,994.5	5,432.0	**17,862.5**
yars_revenge	18089.9	11712.6	16,451.7	49,622.1	16,608.6	9,570.1	**102,557.0**
zaxxon	5363.0	10163.0	13,490.0	12,944.0	18,347.5	9,390.0	**22,209.5**

Table 6: **No-op starts** evaluation regime: Raw scores across all games, averaged over 200 testing episodes, from the agent snapshot that obtained the highest score during training. We report the published scores for DQN, DDQN, Dueling DDQN, and Prioritized DDQN. For Distributional DQN and Rainbow we report our own evaluations of the agents. A3C is not listed since the paper did not report the scores for the no-ops regime.

4. RAINBOW - CODE

PONG



1. Import Libs

In [1]:
```
from common.wrappers import make_atari, wrap_deepmind, wrap_pytorch
from common.replay_buffer import ReplayBuffer
from common.layers import NoisyLinear

from collections import deque
import os
import math
import random
import gym
from gym import wrappers
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.autograd as autograd

from IPython.display import clear_output
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

# plt.style.use('ggplot')
# plt.style.use('fivethirtyeight')
%matplotlib inline

USE_CUDA = torch.cuda.is_available() # False
Variable = lambda *args, **kwargs: autograd.Variable(*args, **kwargs).cuda() if USE_CUDA else autograd.Variable(*args, **kwargs)
print(USE_CUDA)
```

```
True
```

2. Setting Env

In [2]:
```
env_id = "PongNoFrameskip-v4"
env = make_atari(env_id)
env = wrap_deepmind(env)
env = wrap_pytorch(env)

print("env's action space : ", env.action_space)
print("env's observation space : ", env.observation_space)
print("env's reward range : ", env.reward_range)
print("env's number of action : ", env.action_space.n)
```

```
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.uint8'>. Please provide explicit dtype.
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>. Please provide explicit dtype.
env's action space :  Discrete(6)
env's observation space :  Box(1, 84, 84)
env's reward range :  (-inf, inf)
env's number of action :  6
```

4. RAINBOW - CODE

NOISYLINEAR

Factorised Gaussian Noise

- $\mu_{i,j} : u\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}\right]$
- p : number of corresponding linear layer
- $\sigma_{i,j} : \frac{\sigma_0}{\sqrt{p}}$, σ_0 is 0.5
- $\epsilon : \epsilon^w_{i,j} = f(\epsilon_i)f(\epsilon_j),\ \epsilon^b_j = f(\epsilon_j)$
- f : real-valued function, $f(x) = sign(x)\sqrt{|x|}$

$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b$$

where $\zeta = (\mu, \sigma)$, learnable parameters

ϵ, vector of zero-mean noise with statics

In [3]:
```
class NoisyLinear(nn.Module):
    def __init__(self, in_features, out_features, std_init=0.4): # 논문기준 : std_init=0.5
        super(NoisyLinear, self).__init__()

        self.in_features = in_features
        self.out_features = out_features
        self.std_init = std_init

        self.weight_mu = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.weight_sigma = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.register_buffer('weight_epsilon', torch.FloatTensor(out_features, in_features))

        self.bias_mu = nn.Parameter(torch.FloatTensor(out_features))
        self.bias_sigma = nn.Parameter(torch.FloatTensor(out_features))
        self.register_buffer('bias_epsilon', torch.FloatTensor(out_features))

        self.reset_parameters()
        self.reset_noise()

    def forward(self, x):
        if self.training:
            weight = self.weight_mu + self.weight_sigma.mul(Variable(self.weight_epsilon))
            bias = self.bias_mu + self.bias_sigma.mul(Variable(self.bias_epsilon))
        else:
            weight = self.weight_mu
            bias = self.bias_mu

        return F.linear(x, weight, bias)
    def reset_parameters(self):
        mu_range = 1 / math.sqrt(self.weight_mu.size(1))

        self.weight_mu.data.uniform_(-mu_range, mu_range)
        self.weight_sigma.data.fill_(self.std_init / math.sqrt(self.weight_sigma.size(1)))

        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(self.std_init / math.sqrt(self.bias_sigma.size(0)))

    def reset_noise(self):
        epsilon_in = self._scale_noise(self.in_features)
        epsilon_out = self._scale_noise(self.out_features)

        self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
        self.bias_epsilon.copy_(self._scale_noise(self.out_features))

    def _scale_noise(self, size):
        x = torch.randn(size)
        x = x.sign().mul(x.abs().sqrt())
        return x
```

4. RAINBOW - CODE

DUELING + NOISY + C51

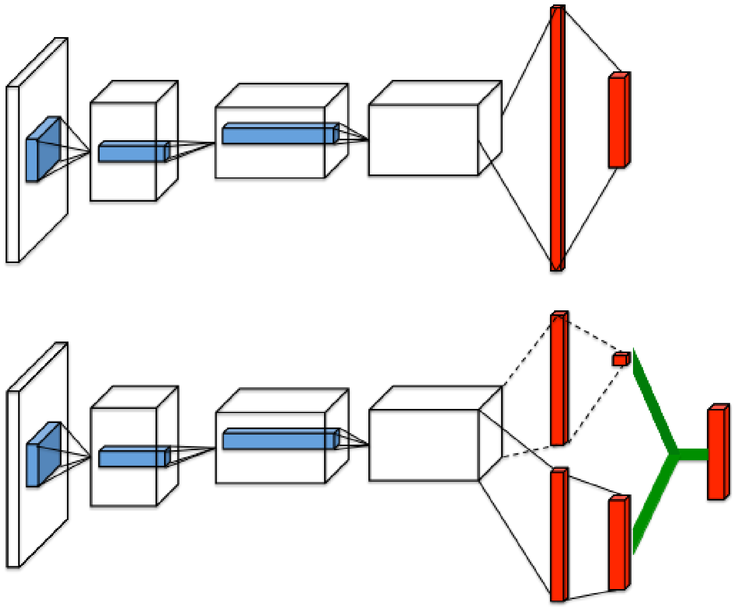


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

3. Build Model

In [3]:

```
class RainbowDQN(nn.Module):
    def __init__(self, input_shape, num_actions, num_atoms, Vmin, Vmax):
        super(RainbowDQN, self).__init__()

        self.input_shape = input_shape
        self.num_actions = num_actions
        self.num_atoms = num_atoms
        self.Vmin = Vmin
        self.Vmax = Vmax

        self.features = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        self.noisy_value1 = NoisyLinear(self.feature_size(), 512, use_cuda=USE_CUDA)
        self.noisy_value2 = NoisyLinear(512, self.num_atoms, use_cuda=USE_CUDA)

        self.noisy_advantage1 = NoisyLinear(self.feature_size(), 512, use_cuda=USE_CUDA)
        self.noisy_advantage2 = NoisyLinear(512, self.num_atoms * self.num_actions, use_cuda=USE_CUDA)

    def forward(self, x):
        batch_size = x.size(0)

        x = x / 255.
        x = self.features(x)
        x = x.view(batch_size, -1)

        value = F.relu(self.noisy_value1(x))
        value = self.noisy_value2(value)

        advantage = F.relu(self.noisy_advantage1(x))
        advantage = self.noisy_advantage2(advantage)

        value = value.view(batch_size, 1, self.num_atoms)
        advantage = advantage.view(batch_size, self.num_actions, self.num_atoms)

        x = value + advantage - advantage.mean(1, keepdim=True)
        x = F.softmax(x.view(-1, self.num_atoms)).view(-1, self.num_actions, self.num_atoms)

        return x

    def reset_noise(self):
        self.noisy_value1.reset_noise()
        self.noisy_value2.reset_noise()
        self.noisy_advantage1.reset_noise()
        self.noisy_advantage2.reset_noise()

    def feature_size(self):
        return self.features(autograd.Variable(torch.zeros(1, *self.input_shape))).view(1, -1).size(1)

    def act(self, state):
        state = Variable(torch.FloatTensor(np.float32(state)).unsqueeze(0), volatile=True)
        dist = self.forward(state).data.cpu()
        dist = dist * torch.linspace(self.Vmin, self.Vmax, self.num_atoms)
        action = dist.sum(2).max(1)[1].numpy()[0]
        return action
```

4. RAINBOW - CODE

PROJECTION STEP

Algorithm 1 Categorical Algorithm

input A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0,1]$
$Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$
$a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$
$m_i = 0, \quad i \in 0, \dots, N-1$
for $j \in 0, \dots, N-1$ **do**
# Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$
$\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$
$b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z \quad \# \, b_j \in [0, N-1]$
$l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
Distribute probability of $\hat{\mathcal{T}} z_j$
$m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
$m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
end for
output $-\sum_i m_i \log p_i(x_t, a_t) \quad$ # Cross-entropy loss

Projection step

$$(\Phi \hat{T} Z_\theta(x,a))_i = \sum_{j=0}^{N-1} clip\left[1 - \frac{|clip[\hat{T} z_j]_{V_{MIN}}^{V_{MAX}} - z_i|}{\triangle z}\right] p_j(x', \pi(x')),$$

where $z_i = V_{MIN} + i\triangle z, \quad \triangle z = \frac{V_{MAX} - V_{MIN}}{N-1},$

$\hat{T} z_j = r + \gamma z_j, \quad p_j$ is probability of component j

In [4]:
```
def projection_distribution(next_state, rewards, dones):
    batch_size = next_state.size(0)

    delta_z = float(Vmax - Vmin) / (num_atoms - 1)
    support = torch.linspace(Vmin, Vmax, num_atoms)

    next_dist = target_model(next_state).data.cpu() * support
    next_action = next_dist.sum(2).max(1)[1]
    next_action = next_action.unsqueeze(1).unsqueeze(1)₩
                    .expand(next_dist.size(0), 1, next_dist.size(2))
    next_dist = next_dist.gather(1, next_action).squeeze(1)

    rewards = rewards.unsqueeze(1).expand_as(next_dist)
    dones = dones.unsqueeze(1).expand_as(next_dist)
    support = support.unsqueeze(0).expand_as(next_dist)

    Tz = rewards + (1 - dones) * 0.99 * support
    Tz = Tz.clamp(min=Vmin, max=Vmax)
    b = (Tz - Vmin) / delta_z
    l = b.floor().long()
    u = b.ceil().long()

    offset = torch.linspace(0, (batch_size - 1)₩
        * num_atoms, batch_size).long().unsqueeze(1).expand(batch_size, num_atoms)

    proj_dist = torch.zeros(next_dist.size())
    proj_dist.view(-1).index_add_(0, (l + offset).view(-1),₩
                                  (next_dist * (u.float() - b)).view(-1))
    proj_dist.view(-1).index_add_(0, (u + offset).view(-1),₩
                                  (next_dist * (b - l.float())).view(-1))

    return proj_dist
```

4. RAINBOW - CODE

CROSS-ENTROPY LOSS

Algorithm 1 Categorical Algorithm

input A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0, 1]$

$Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$

$a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$

$m_i = 0, \quad i \in 0, \dots, N-1$

for $j \in 0, \dots, N-1$ **do**

Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$

$\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$

$b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z \quad \# \, b_j \in [0, N-1]$

$l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$

Distribute probability of $\hat{\mathcal{T}} z_j$

$m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$

$m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$

end for

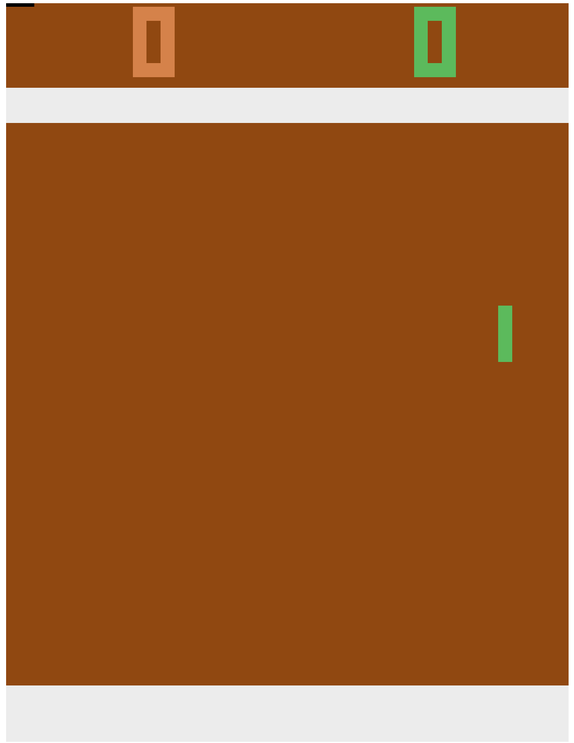
output $-\sum_i m_i \log p_i(x_t, a_t)$ # Cross-entropy loss

In [7]:

```
def compute_td_loss(batch_size):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)), volatile=True)
    action = Variable(torch.LongTensor(action))
    reward = torch.FloatTensor(reward)
    done = torch.FloatTensor(np.float32(done))

    proj_dist = projection_distribution(next_state, reward, done)

    dist = current_model(state)
    action = action.unsqueeze(1).unsqueeze(1).expand(batch_size, 1, num_atoms)
    dist = dist.gather(1, action).squeeze(1)
    dist.data.clamp_(0.01, 0.99)
    loss = -(Variable(proj_dist) * dist.log()).sum(1).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    current_model.reset_noise()
    target_model.reset_noise()

    return loss
```

4. RAINBOW - CODE

TEST



```
current_model = RainbowDQN(env.observation_space.shape, env.action_space.n, num_atoms, Vmin, Vmax)
current_model.load_state_dict(torch.load('./save_model/PongNoFrameskip-v4/999398-BreakoutNoFrameskip-v4_RainbowDQN.pkl'))

if USE_CUDA:
    current_model = current_model.cuda()

EPISODE = 5
batch_size = 32
gamma = 0.99

losses = []
scores = []

global_steps = 1
rendering = True
```

```
for episode in range(1, EPISODE + 1):
    frames = []
    done = False
    score = 0
    state = env.reset()

    # for gif
    obs = env.render(mode='rgb_array')
    frames.append(obs)

    while True:
        if rendering:
            env.render()

        frames.append(env.render(mode='rgb_array'))
        action = current_model.act(state)

        next_state, reward, done, _ = env.step(action)

        state = next_state
        score += reward
        global_steps += 1

        if done:
            break

string = '{}_{}_{}.gif'
imageio.mimsave('./save_gif/' + string.format(env_id, "RainbowDQN", episode), frames, duration=0.0286)

env.close()
```

PYCON KOREA 2018

Thank you

RAINBOW

김예찬