# Improving Action Branching for Deep Reinforcement Learning with A Multi-dimensional Hybrid Action Space

Laige Peng[1,a)] Yoshimasa Tsuruoka[1]

**Abstract:** Recent deep reinforcement learning methods address the complexity of state space and achieve great success in various video games. Deep Q-Network (DQN)-like algorithms show efficiency in environments with discrete action spaces while policy-based algorithms have good performance in environments with continuous action spaces. However, it is difficult to apply those algorithms in a complicated multi-dimensional hybrid action space in which both discrete and continuous action spaces exist. We propose to combine the action branching architecture proposed by Tavakoli et al. [1] with the proximal policy optimization (PPO) algorithm to address this problem. Our method keeps the continuous action space and achieves better performance than the dueling double DQN model which discretizes the continuous action space, and shows better compatibility with human demonstration data.

## 1. Introduction

In recent years, deep reinforcement learning (DRL) has led to striking success in several research fields such as game AI, robot control and navigation problems. The first success of DRL in the game of Go and the Atari games provide great confidence and a promising methodology for researchers to tackle complicated game problems. DRL has developed quickly and derived many extensions and variants for challenging games. Hasselt et al. [2] developed the Double Deep Q-Network (DDQN) to solve the overestimation problem in the traditional DQN algorithm and improved the generalization capability. Wang et al. [3] proposed Dueling DQN which outperforms the state-of-the-art on the Atari domain. Besides value-based algorithms, policy-based algorithms have also been widely used such as A3C [4], deterministic policy gradients (DPG) [5] and its deep version DDPG [6], and proximal policy optimization (PPO) algorithms [7].

As the development of DRL advances, many challenging games with much more complicated state spaces such as VizDoom have been overcome; however, there still remain challenges to solve games with both complex state spaces and action spaces such as StarCraft and Minecraft, which requires better sequence decisions. Previous value-based research mainly focuses on the state spaces and assumes that the action space is discrete and has only one dimension. For continuous action spaces, there are mainly two solutions: discretize the action space and utilize value-based algorithms or directly generate continuous output by applying policy-based algorithms such as DDPG. As for multi-dimensional action spaces such as the angles for different joints of a robot arm, Tavakoli et al. [1] proposed an action branching architecture to use a separate action branch for each action dimension while sharing the same state representation. They discretized the continuous value and utilized the dueling double DQN (DDDQN) as the reinforcement learning algorithm. Xiong et al. [8] proposed a parametrized DQN (P-DQN) method which can learn with a discrete-continuous hybrid action space by combining the DQN algorithm and the DDPG algorithm. Vinyals et al. [9] designed a parametrized action space for the game StarCraft II in which each action contains a function identifier and a sequence of arguments. They made a baseline with A3C as the learning algorithm and represented the policy in an auto-regressive manner by using the chain rule.

We take Minecraft as our platform for its challenging environment, high-dimension representation, complex action space and hierarchical item systems. We use the MineRL API and take a navigation task and a treechop task as our training domains. MineRL provides a real game environment in which the map is very large and contains all the possible entities in it. As a result, it is very hard for the agent to explore the environment with normal one dimension action space. What is more, besides discrete actions such as "forward" and "back", the agent should also take some continuous actions such as the direction and the sight-line angle. We consider it as a multi-dimensional discrete-continuous hybrid action space.

Considering the complexity of the action space, we assume that each action dimension is independent to each other and

[1] Department of Information and Communication Engineering, The School of Information Science and Technology, The University of Tokyo
[a)] penglaige@logos.t.u-tokyo.ac.jp

© 2019 Information Processing Society of Japan

utilize the action branching architecture to generate separate actions for each dimension. For a continuous action space, discretization may violate the mechanism of the action space, and will generate too many discrete points. Thus, in this paper, we improve the action branching architecture to deal with a multi-dimensional hybrid action space by applying PPO. We make use of non-pixel observations to augment the state representation as shown in Figure 1, categorial distributions and Gaussian distributions are used to sample discrete actions and continuous actions respectively. We also propose to use supervised learning during a pre-training phase by using human demonstration data to quickly start the training and accelerate training process.

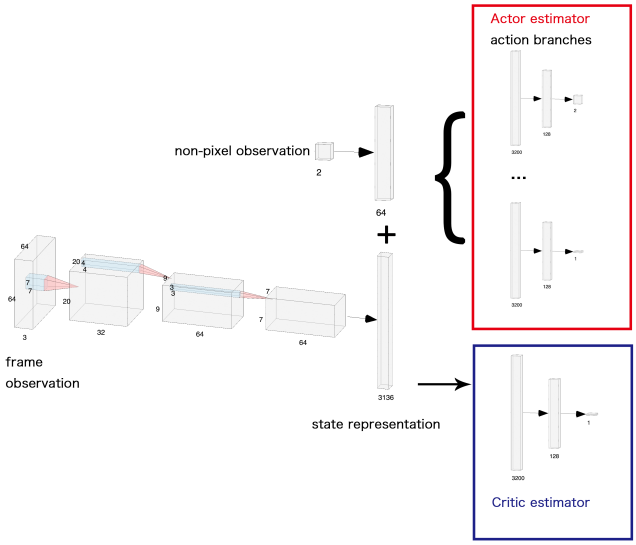


**Figure 1.** Proposed method

We compared our proposed method with DDDQN in two environments. In the navigation task with dense rewards, PPO learns faster than DDDQN and achieves better performance. In the treechop task with extremely sparse rewards, PPO shows better compatibility with human demonstration data and achieves higher scores than DDDQN. The experiments show efficiency of our proposed method.

## 2. Background

### 2.1 Reinforcement learning basics

Reinforcement learning [10] focuses on the interaction between the agent and the environment and is usually modeled as a Markov Decision Process (MDP) with definitions $\{S, A, R, P(s\prime \mid s, a), \gamma\}$. The agent observes current state $s_t \in S$ from the environment $\epsilon$, chooses an action $a_t \in A$ following the policy $\pi(a \mid s) = P[A_t = a \mid S_t = s]$ and receives a reward $r_t \in R$ from the environment. The environment makes a transition to another state $s_{t+1} \in S$ with the transition probability $p(s_{t+1} \mid s_t, a_t)$, and then the agent repeats choosing a new action.

Reinforcement learning tries to maximize the expectation of the cumulative rewards $V_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$, for all $s \in S$ with a discount factor $\gamma$ from time $t$ given state $s$. $V$-value can be written as $V_\pi(s) = \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s]$ according to the Bellman equation.

Value-based reinforcement learning algorithms such as Q-learning maximize the action value function $q_\pi(s, a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$ to approximate the optimal policy $\pi$. Empirically, the Bellman equation $q(s_t, a_t) = r_t + \gamma \max_{a\prime} q(s_{t+1}, a\prime)$ is widely used in value-based algorithms.

Policy-based reinforcement learning algorithms such as REINFORCE maximize the objective function $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau) r(\tau) d\tau$ by directly optimize the policy $\pi(s|a)$ with policy gradient $\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$, where $\tau$ is a trajectory.

### 2.2 Deep Q-Network (DQN) families

Deep Q-Network [11] uses neural networks to build an approximate function of the action value function $Q_\pi(s, a)$ to solve high-dimensional observation problems such as pixel frame inputs, and is trained by the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s_{t+1})\sim U(D)}[(r_t + \gamma \max_{a\prime} Q(s_{t+1}, a\prime; \theta_i^-) - Q(s_t, a_t; \theta_i))^2],$$

in which $\theta$ is the action network parameter and $\theta_i^-$ is the target network parameter at iteration $i$. Here action network is used to choose actions. And the target value of $Q(s, a)$ is

$$\hat{y}_t = r_t + \gamma \max_{a\prime} Q(s_{t+1}, a\prime; \theta_i^-).$$

The target network parameter $\theta_i^-$ updates with $\theta_i$ only every $C$ steps in order to eliminate the correlation between the network $Q(\theta)$ and the target network $Q(\theta^-)$.

Experience replay is another key idea to break the correlation between sequence trajectories. A replay buffer stores the $(s, a, r, s\prime)$ tuples as the agent interacts with the environment and randomly samples data from the replay buffer to train the model in fixed time intervals, reducing the influence of the correlation.

Some important and widely used extensions of DQN are introduced as follows:

**Double DQN (DDQN)** Hasselt et al. [2] developed DDQN in order to solve the overestimation problem in some games and the algorithm is shown to be able to be generalized to work with large-scale function approximation. The target value of DDQN is written as:

$$\hat{y_t}^{DDQN} = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_i); \theta_i^-).$$

Unlike DQN that directly chooses action which maximizes $Q(s_{t+1}, a; \theta_i^-)$, DDQN uses the action network $Q_{\theta_i}$ to select an action that maximizes $Q(s_{t+1}, a; \theta_i)$ and uses this action to calculate the target value $\hat{y_t}^{DDQN}$.

© 2019 Information Processing Society of Japan

**Dueling DQN** Wang et al. [3] proposed Dueling DQN which can generalize learning across actions without changing current reinforcement learning algorithms. Dueling DQN also provides better evaluation of policy and outperforms the state-of-the-art on the Atari domain. In value-based reinforcement learning algorithms, action advantage $A(s,a)$ is defined as $Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a)$. DQN uses neural network to directly approximate the $Q$-value; however, dueling DQN separates it into two parts, one for estimating the $V$-value, the other for estimating the action advantage while the two parts sharing the convolutional neural networks. However, an unidentifiable problem of calculating $Q$-value by adding $V$-value and $A$-value is when given $Q$ we can not recover $V$ and $A$ uniquely. To solve this issue of identifiability, the solution is to force the advantage function estimator to be zero at the chosen action. Then the dueling DQN is implemented with:

$$Q_\pi(s,a) = V_\pi(s) + (A_\pi(s,a) - \max_{a\prime} A(s,a\prime)).$$

An alternative way is to replace the max operator by an average operator as:

$$Q_\pi(s,a) = V_\pi(s) + (A_\pi(s,a) - \tfrac{1}{|\mathcal{A}|}\textstyle\sum_{a\prime} A(s,a\prime)),$$

which may provide better performance.

**Prioritized replay buffer** Prioritized replay buffer [12] is developed to solve a sample efficiency problem. Generally, we want the agent to pay more attention to some transitions than others because it can learn more from those transitions; however, the original replay buffer randomly samples from the whole buffer. Priority replay buffer tends to more frequently sample transitions with high expected learning progress, which is measured by their temporal-difference (TD) error $\delta$. The probability of sampling a transition $i$ is defined as:

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

in which $p_i > 0$ is the priority of transition $i$, the parameter $\alpha$ indicates how much prioritization will be used. There are mainly two ways to calculate the priority for each transition. One direct way is a proportional prioritization where $p_i = |\delta_i| + \epsilon$, $\epsilon$ is a small positive value to prevent that a transition not being resampled once when the TD loss is zero. The other way is an indirect rank-based prioritization by calculating $p_i = \frac{1}{rank(i)}$, where $rank(i)$ for transition $i$ is decided when it is stored into the replay buffer with TD loss $|\delta_i|$.

### 2.3 Proximal Policy Optimization Algorithms (PPO)

As mentioned in section 2.1, policy gradient is represented as $\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$, a sample of a trajectory is usually used to calculate the expectation; however, directly applying the $r(\tau)$ into this equation causes unstable performance because of the huge variance. One common method is to subtract a baseline $r_b$ from $r(\tau)$ to make it more robust. One most commonly used gradient estimator is represented as:

$$\hat{g} = \mathbb{E}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t],$$

where $\hat{g}$ is a gradient estimator and $\hat{A}_t$ is an estimator of the advantage function at timestep $t$. Here the policy gradient estimator $\hat{g}$ is obtained from the objective loss function:

$$L^{PG}(\theta) = \mathbb{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t].$$

One problem with this objective function is that empirically it often results in large policy updates and affect the robustness. Trust region methods such as TRPO [13] maximizes another objective loss function with a constraint in order to address the large policy updates problem by the following form:

$$\max_\theta \mathbb{E}_t[\tfrac{\log \pi_\theta(a_t|s_t)}{\log \pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{old}}(\dot{|}s_t), \pi_\theta(\dot{|}s_t)]].$$

PPO [7] proposed a clipped surrogate objective loss function to penalize changes to the policy as the following form:

$$L^{CLIP}(\theta) =$$

$$\mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)],$$

where the probability ratio $r_t(\theta)$ is defined as $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)}$ and $\epsilon$ is a hyperparameter.

In order to reduce variance, PPO also makes use of a state-value estimator $V(s)$, and the objective loss function is a combination of the policy loss and the value function error term. The objective function can further be augmented by combining an entropy bonus to ensure sufficient exploration. By adding all these terms together, PPO proposes the following objective function:

$$L_t^{CLIP+VF+S}(\theta) =$$

$$\hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)],$$

where $c_1$, $c_2$ are coefficients, $S$ represents the entropy bonus, and $L_t^{VF}$ is a squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$.

### 2.4 Branching Dueling Q-Network (BDQ)

In order to solve a multi-dimensional action space problem, Tavakoli et al. [1] proposed an action branching architecture (Figure 2) to address this issue. As indicated as the name of this methodology, action branching architecture separates several network branches for each action dimension, while each dimension shares the common convolutional neural networks to obtain the same state representation vector. State representation vector is then fed into different action branches and calculate action advantages for each dimension. A common value function estimator is also used to approximate state value, and then is used to calculate $Q$-value for each dimension. Dueling Double DQN (DDDQN) and prioritized replay buffer are used in this architecture.

BDQ proposed several TD error candidates. One simplest way is to calculate branch-separated TD target as:

$$y_d = r + \gamma Q_d^-(s\prime, \arg\max_{a\prime_d \in A_d} Q_d(s\prime, a\prime_d)),$$

where $d$ represents the $d$th branch. Alternatively, a single

© 2019 Information Processing Society of Japan

global TD target can be used for all branches by using the maximum TD target over all branches with the following form:

$$y = r + \gamma \max_d Q_{\bar{d}}(s\prime, \arg\max_{a\prime_d \in A_d} Q_d(s\prime, a\prime_d)).$$

The maximum operation can be replaced by a mean operation:

$$y = r + \gamma \frac{1}{N} \sum_d Q_{\bar{d}}(s\prime, \arg\max_{a\prime_d \in A_d} Q_d(s\prime, a\prime_d)),$$

and this TD target shows better performance.

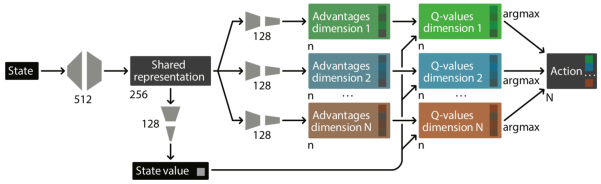


**Figure 2.** Branching Dueling Q-Network

The final loss function is calculated by the mean squared TD error across all the action branches:

$$L = \mathbb{E}_{(s,a,r,s')\sim D}[\frac{1}{N} \sum_d (y_d - Q_d(s, a_d))^2].$$

Gradient rescaling is applied because all branches back-propagate gradients through the shared convolutional neural networks.

### 2.5 Deep reinforcement learning with demonstrations

For games with sparse rewards, it is very difficult for a random agent to get some rewards from the environment, thus it is very hard to learn an efficient policy. Hester et al. [14] proposed Deep Q-learning from Demonstrations (DQfD) to apply existing demonstrations to accelerate the learning process and provide good starting policy instead of a random policy. In such case, DQfD can be used to deal with environments with extremely sparse rewards when provided demonstration data.

DQfD separates the learning process into a pre-training phase and an interacting phase, and uses a variant of replay buffer with fixed demonstration data. During the pre-training phase, the agent samples min-batches from the demonstration data and updates the network with a combination loss: a 1-step double Q-learning loss, an n-step double Q-learning loss, a supervised large margin classification loss, and an L2 regularization loss. The large margin classification loss is defined as:

$$J_E(Q) = \max_{a \in A}[Q(s, a) + l(a_E, a)] - Q(s, a_E),$$

where $a_E$ is the demonstration action and $l(a_E, a)$ is a margin function that equals to 0 when $a = a_E$ and a positive value otherwise.

The combination loss is represented as:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q),$$

where $\lambda$s are used to control the weights of the losses.

Once the pre-training phase ends, the agent begins to interact with the environment as normal DQN methods do. Two things different from normal methods are replay buffer and proportional prioritized sampling. The replay buffer in DQfD fixes the demonstration data and never replace those data. DQfD also uses positive constants $\epsilon_a$ and $\epsilon_d$ as priority bonus for agent data and demonstration data respectively to control the relative sampling between demonstration and agent data.

## 3. Proposed Approach

To deal with multi-dimensional hybrid action spaces, we propose to combine action branching architecture and the policy gradient algorithm PPO. Figure 1 shows our proposed architecture. In this section we will introduce several methods used in our proposed approach.

**Adding non-pixel observation** We not only take the pixel-based frames as observation, but also the non-pixel observations provided by the experiment environment such as compass angles and inventory items. To deal with those non-pixel observations, we feed the observation vector into extra fully connected layers and get the feature vector, which is then added to the output of the convolutional layers. We take the combination of non-pixel feature vector and frame feature vector as the final state representation and then feed it into the value function estimator and action branches.

**TD loss** In the multi-dimensional hybrid action space, we use categorical distributions for discrete action spaces and Gaussian distributions for continuous action spaces respectively. For each dimension, branch loss is calculated by $L_d^{CLIP+VF+S}(\theta)$ mentioned in section 2.3, where $d$ means the $d$th branch.

**Loss function** One most direct and simplest way to define the loss function is the averaged TD loss across all action branches as:

$$L = \mathbb{E}_{\tau\sim\pi(\theta)}[\frac{1}{N} \sum_d L_d^{CLIP+VF+S}(\theta)],$$

where $N$ is the number of total branches.

**Human demonstration** For DQN-like value-based algorithms, we can apply DQfD to utilize human demonstration data to improve sample efficiency, provide the agent with a good starting policy, and accelerate the learning process. However, in policy-based algorithms such as PPO, the actor estimator directly approximates the policy $\pi(a|s)$, it is convenient to directly apply supervised learning to update the actor estimator. We use negative log likelihood loss for discrete actions and MSE loss for continuous actions respectively, and the final loss function is defined as the averaged loss. We also separate the learning process into a pre-training phase and an interacting phase. During the pre-training phase, only the actor estimator is updated with supervised learning. During the interacting phase, the agent interacts with the environment for several episodes with

© 2019 Information Processing Society of Japan

PPO, then updates the actor estimator for a fixed number of times with supervised learning by using demonstration data, and continues this loop.

**Multi-process simulation** In order to reduce the variance of the policy gradient, a common method for policy-based algorithms is to use multiple environments to sample multiple trajectories and take the average. However, our experiment MineRL environments do not support multi-processing. As a result, we use only one game process to simulate multiple processes to reduce variance. Generally, PPO creates $N$ processes, steps $T$ timesteps for each update, and samples mini-batches from those $NT$ timesteps. In our experiments, we create 1 process, step $NT$ timesteps and treat them as $N$ trajectories with length $T$ and do the same sampling as the original PPO.

# 4. Experiment

## 4.1 Environments

**Navigation task** In this task, the goal is to find a diamond block in a random generated Minecraft world. Accessible observations are frames, the number of item "dirt" in the inventory, and a "compassAngle" observation which points near the goal location, 64 meters from the start location. Actions include "forward", "back", "left", "right", "attack", "place", "jump", "sneak", "sprint", "direction", "sightline angle", where "direction" and "sightline angle" are continuous values. The agent is given a +100 reward upon touching the diamond block; however, we use a dense reward variant in which the agent is given a reward every tick for how much closer (or a negative reward for farther) the agent gets to the target. Episode terminates when agent reaching the goal block or using up maximum 6000 steps.

**Figure 3.** Navigation task environment. The goal is to find a diamond block in a random environment, the diamond may be slightly below surface level.

**Figure 4.** Treechop task environment. The goal is to obtain 64 log units from a forest biome.

**Treechop task** As logs are necessary and significant to craft a wide range of items, it is a very important skill to get logs from the environment. In this task, the agent needs to collect 64 logs in a forest biome given an iron axe for cutting trees. Accessible observations are frames only, while actions are "attack", "back", "forward", "jump", "left", "right", "sneak", "sprint", "direction" and "sightline". The agent is given a +1 reward when it obtains a unit of wood. Episode terminates when agent obtains 64 units or uses up 8000 steps.

## 4.2 Navigation

We trained four different models in the navigation task: (1) Dueling double DQN without human demonstration data; (2) DDDQN with human demonstration data while the pre-training loss is $J(Q) = J_{DQ}(Q)$; (3) DDDQN with human demonstration data while the pre-training loss is $J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_3 J_{L2}(Q)$; (4) PPO without demonstration data. For all DDDQN models, we applied the prioritized replay buffer and discretized the continuous action spaces. All models used the same network architecture shown in Figure 1; however, for DDDQN, the output layers of continuous action dimensions are different from PPO model. In DDDQN demo models, mini-batch size is 32, the number of pre-training step is 100000. Demonstration data replay $\epsilon$ is 0.01 and interaction data replay $\epsilon$ is 0.0001.


**Figure 5.** Training results in navigation task for 4 models. DDDQN represents model 1, DDDQN_V0 for model 2, DDDQN_V1 for model 3 and PPO for model 4.

| Episode rewards analysis for 4 models in navigate | | | | |
|---|---|---|---|---|
| Model | Mean reward | Median reward | Min reward | Max reward |
| DDDQN | 34.90 | 33.36 | -18.76 | 175.18 |
| DDDQN_V0 | 39.46 | 43.06 | -44.33 | 175.99 |
| DDDQN_V1 | 41.96 | 46.08 | -34.72 | 177.49 |
| PPO | **49.27** | **53.15** | **-14.64** | 172.28 |

**Table 1.**

Training results are shown in Figure 5 and Table 1. PPO learns faster and achieves higher average rewards than other three models, even obtains better performance than

© 2019 Information Processing Society of Japan

DDDQN with human demonstration data. From Table 1, we can see that PPO has the highest mean and median reward over 1000 episodes. The min reward of PPO comes from the first episode in the training process and episode reward gets higher gradually.

### 4.3 Treechop

We trained four models for the treechop task: (1) DDDQN without human demonstration; (2) DDDQN with human demonstration and the pre-training loss is $J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_3 J_{L2}(Q)$; (3) PPO without human demonstration; (4) PPO with human demonstration. All these four models used the same network shown in Figure 1, except that DDDQN algorithms had different output layers with PPO algorithms. In our proposed PPO demo model, mini-batch size is 32, the number of pre-training step is 50000. During the interacting phase, we trained the agent with demonstration data for 500 steps every 2 episodes.

| Episode rewards analysis for 4 models in treechop | | | | |
|---|---|---|---|---|
| Model | Mean reward | Median reward | Min reward | Max reward |
| DDDQN | 0.0 | 0.0 | 0.0 | 0.0 |
| DDDQN_DEMO | 8.52 | 8.0 | 0.0 | 37.0 |
| PPO | 0.0 | 0.0 | 0.0 | 0.0 |
| PPO_DEMO | **36.62** | **38.0** | 0.0 | **63.0** |

**Table 2.** In treechop task, episode terminates when agent obtains 64 log units, thus the highest score in one episode is 63.

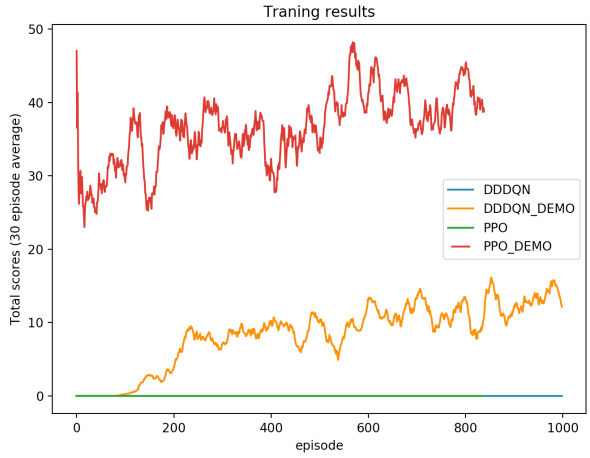


**Figure 6.** Training results in treechop task for 4 models.

Training results of these models are shown in Figure 6 and Table 2. The results of PPO and DDDQN without demonstration data indicate that in the treechop task, which has an environment with extremely sparse reward and is very hard to get rewards with random actions from, pre-training with demonstration data is crucial for the agent to have a quick start. With demonstration data, PPO achieves much more higher scores than DDDQN, shows better compatibility with data and sample efficiency.

## 5. Conlusion

In this work, we propose to combine the action branching architecture and PPO to solve a multi-dimensional hybrid action space problem in the game of Minecraft. We compare our methods with BDQ in two different tasks. Our method greatly accelerates the learning process and achieves better performance in both environments. In the treechop task, our method shows better compatibility with human demonstration data.

We think there are still some ways to improve this method. In this work, we only apply the simplest way to combine branch losses into a final loss function, it could be improved to use other kinds of aggregation. The second point is that it is able to reduce the action dimension, such as combining "forward" and "back" into one dimension. Also when we used human demonstration data in our proposed method, we found that it was difficult to make a balance between supervised learning and environment interaction. Overfitting occurred when the frequency of supervised learning was high during the interacting phase while lower frequency decreased the performance.

## References

[1] Tavakoli, A. et al.: Action Branching Architectures for Deep Reinforcement Learning, *AAAI Conference on Artificial Intelligence*, pp. 4131–4138 (2018).
[2] van Hasselt, H., Guez, A. and Silver, D.: Deep Reinforcement Learning with Double Q-learning, *ArXiv e-prints* (2015).
[3] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. and de Freitas, N.: Dueling Network Architectures for Deep Reinforcement Learning, *ArXiv e-prints* (2015).
[4] Mnih, V. et al.: Asynchronous methods for deep reinforcement learning, *ICML 2016*, pp. 1928–1937 (2016).
[5] Silver, D. et al.: Deterministic policy gradient algorithms, *ICML 2014* (2014).
[6] Lillicrap, T. P. et al.: Continuous control with deep reinforcement learning, *arXiv preprint arXiv:1509.02971* (2015).
[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O.: Proximal policy optimization algorithms, *arXiv preprint arXiv:1707.06347* (2017).
[8] Xiong, J. et al.: Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space, *arXiv preprint arXiv:1810.06394* (2018).
[9] Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J. et al.: Starcraft ii: A new challenge for reinforcement learning, *arXiv preprint arXiv:1708.04782* (2017).
[10] Sutton, R., Barto, A., Barto, R., Barto, C. and Bach, F.: *Reinforcement Learning: An Introduction*, A Bradford book, Bradford Book (1998).
[11] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D.: Human-level control through deep reinforcement learning, *Nature*, Vol. 518, pp. 529 EP – (online), available from ⟨http://dx.doi.org/10.1038/nature14236⟩ (2015).
[12] Schaul, T., Quan, J., Antonoglou, I. and Silver, D.: Prioritized experience replay, *arXiv preprint arXiv:1511.05952* (2015).
[13] Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P.: Trust region policy optimization, *International conference on machine learning*, pp. 1889–1897 (2015).
[14] Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I. et al.: Deep q-learning from demonstrations, *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

© 2019 Information Processing Society of Japan