Using Options to Improve Learning in DQN

Dorin Drachsler, Kfir Goldberg, Tom Zahavy, Daniel J. Mankowitz, Shie Mannor
Electrical Engineering Department
The Technion - Israel Institute of Technology
Haifa 32000, Israel

Abstract

Learning in hierarchical environments, with complex goals and sparse feedback poses a major challenge for reinforcement learning algorithms. The problem stems from the fact that completing the complex sequence resulting in the reward is very unlikely for an untrained agent, and so the agent has nothing to learn from. We present two main modifications to the current existing state of the art algorithm. First, we introduce intrinsic rewards to the domain, compensating for the sparse and delayed nature of the domain reward. Second, we divide the long term goals to simpler subgoals, and use a compressed architecture to train the agent for each of the subgoals separately of the others. We introduce a controller which handles the long term planning needed for our domain bu initiating and terminating active subgoals according to an optimal long term strategy. We demonstrate the effectiveness of this learning technique on Seaquest domain in Atari2600 emulator, which provides an environment with very sparse and delayed reward.

1 Introduction

Learning in hierarchical environments, with complex goals, requires thorough understanding of the environment, and poses a challenge for artificial intelligence. The difficulty in learning these complex goals is the sparse nature of their feedback, which often is also delayed. Recently, Reinforcement Learning (RL) was combined with Deep Q-Learning to create the Vanilla DQN (Mnih et al. (2015)). The Vanilla DQN has significantly improved the learning capabilities of RL agents, but still shows difficulty in learning complex tasks. We propose to decompose complex goals into simpler sub-goals, each of them is simple enough to be learned using the Vanilla DQN.
In order to learn each of the subgoals, we introduce Intrinsic Rewards (Tejas Kulkarni (2016), Singh et al. (2010), Schmidhuber (2010), Szepesvari et al. (2014)). Intrinsic rewards are separate reward signals given for each of the different subgoals defined in our domain. Their purpose is to overcome the learning difficulties introduced by the sparse and delayed nature of the rewards in our domain.
In order to choose an active policy between the policies of each of the subgoals, we introduce a subgoal controller. The controller handles the long term planning of the agent and acts according to a static strategy, designed specifically for the domain. The controller examines the current game state and chooses which sub-goal to pursue, and initiates the policy corresponding to the selected subgoal.
Previous work (Rusu et al. (2016)) has shown that using distillation it is possible to teach a single Q-network student network the policies of different games, each learned by a separate Q-network teacher. We use this notion, and instead of learning each of the subgoals on a different Q-network, we use a single Q-network for all of the policies, with a separate output layer corresponding to each of the subgoals.
The domain we used for our implementation and experiments is the game Seaquest running on atari2600 emulator, as illustrated in Figure 1. In Seaquest, the agent is a submarine (sub) whose goal is to rescue 6 divers without losing all its air, and then surface, which levels up the agent. The agent starts the game with 3 extra subs, and loses one whenever it is either hit by enemies in the water, runs out of air or surfaces without any rescued divers. The agent is rewarded $(20 + 10 \times level)$ points for each enemy shot, and $(300 + 300 \times level + bounus\_for\_remaining\_air)$ for each time it surfaces with 6 agents. Shooting enemies is a simple task, and so the reward for it is small but frequent and immediate, while surfacing with 6 divers is a more complex task, resulting in a delayed and sparse reward of large value.
Many existing deep reinforcement algorithms have reported scores for Seaquest that are much lower than that of a human expert (Table 1)

Algorihtm	Score
Offline Monte-Carlo (Guo et al. (2014), Guo et al. (2016))	2023
Double Q-Learning (van Hasselt, Guez, and Silver (2015))	7995
Prioritized Experience Replay(Schaul et al. (2015))	39096
Dueling Networks (Wang, de Freitas, and Lanctot (2015))	37361
Asynchronous Methods (Mnih et al. (2016))	2355
Human Expert	>100K

Table 1: Seaqeust scores across different existing algorithms

To summarize, our main contributions are:

- Embedding intrinsic rewards in Seaquest domain
- Using a compressed network architecture to simultaneously teach the agents several policies
- Implementing a controller for long term planning of initiation and termination of active subgoals

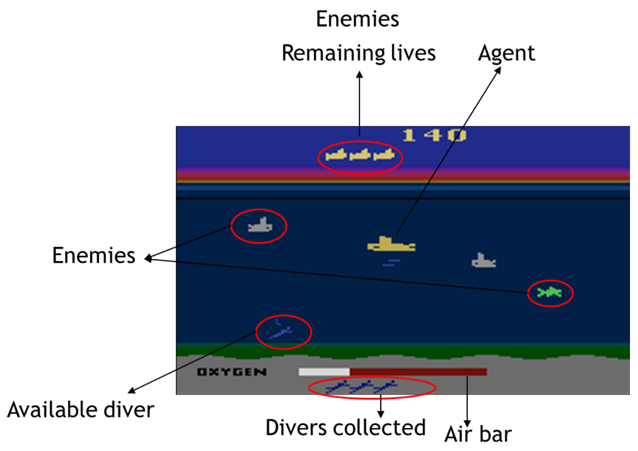


Figure 1: Seaquest in Atari2600 environment. Status indications (remaining lives, remaining air, collected divers, available divers and enemies) along with the agent, are marked

2 Background

Reinforcement Learning: The goal of an RL agent is to maximize its expected future reward. To do so, it learns a policy $\pi : S \mapsto \Delta_A$ which maps the state-space S to the action-space A, giving each action at each state some probability. The process is discretized in time, and at time t the agent observes state $s_t \in S$, selects an action $a_t \in A$ which is rewarded by some reward $r_t \in [0, R_{max}]$. The action a_t chosen in s_t leads to $s_{t+1} \in S$ at the next time step. The cumulative reward at time t is given by $R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_t$ where $\gamma \in [0, 1]$ is the discount factor, determining the importance of immediate reward over future reward. The action-value function Q^π gives the expected accumulated reward when taking action a from state s under a given policy π. The action-value function for the optimal policy obeys a recursive law known as Bellman equation

$$Q^* = \mathbb{E}\left[r_t + \gamma \underset{a'}{max} Q^*\left(s_{t+1}, a'\right)\right]$$

Deep Q Networks: The DQN algorithm [Mnih et al. (2015)] uses a Convolutional Neural Network (CNN) to approximate the optimal Q function by optimizing the network weights such that the expected Temporal Difference (TD) of the optimal Bellman equation (Equation 1) is minimized:

$$\mathbb{E}_{s_t, a_t, r_t, s_{t+1}} \| Q_\theta\left(s_t, a_t\right) - y_t \|_2^2, \qquad (1)$$

where

$$y_t = \begin{cases} r_t, & \text{if} s_{t+1} \text{is terminal} \\ r_t + \gamma \underset{a'}{max} Q_{\theta_{target}}\left(s_{t+1}, a'\right), & \text{otherwise} \end{cases}$$

In DQN, the learning is performed offline, using tuples $\{s_t, a_t, r_t, s_{t+1}, \gamma\}$ that are collected during the agent's activity and stored in an Experience Replay (ER) memory [Lin (1993)]. ER is a buffer storing previous states of the agent. Randomly selected states are retrieved from the ER for the purpose of training the DQN. DQN implements two Q-networks: current Q-network with parameters θ and target Q-network with parameters $\theta_{trarget}$. Every fixed number of iterations, the parameters θ_{target} are updated to θ. DQN defines a state as a few consecutive frames from the environment.

Skills, Options: [Sutton, Precup, and Singh (1999)] A skill is a sequence of primitive actions, defined by the tuple $\langle I, \pi, \beta \rangle$ where I is the set of states in which the skill can be initiated, π is the skill policy, determining what action the skill uses in each state, and β is the set of termination probabilities, determining when a skill stops executing. β is typically defined as a function of the state s or time t.

Policy Distillation: [Rusu et al. (2016), Emilio Parisotto (2016)] Distillation is a way of compressing accumulated knowledge. Policy distillation uses several different deep reinforcement learning (DRL) teacher agents to train a DRL student agent. Each of the teachers learns a different domain using the Vanilla DQN, and passes sets of input states and actions chosen to the student agent. The student uses some conventional supervised learning algorithm to learn from the sets passed from all of the teachers, eventually learning the domains of all the teachers without performance degradation.

3 Method

In this section we provide an in-depth description of our agent and the algorithm used for training it. Our goal is to break down the hierarchical structure of the domain and define some simplified subgoals. We define different skills, each aiming at completing a different subgoal. Each skill has initiation and termination conditions, and a policy according to which it plays while active. Skills' policies are learned on separate network, using intrinsic reward signals we defined. The intrinsic rewards aim at replacing the sparse and delayed rewards given by the game. We define an intrinsic reward for each of the subgoals, and design it to be frequent and immediate. We use a subgoal controller to examine the current game state and choose which subgoal to pursue. By using the controller, we manually provide the agent with long-term planning according to the strategy programmed into the controller for the domain.

3.1 Subgoal Controller

The domain we use for implementing and testing the Multi Subgoal DQN (MSDQN) is the game Seaquest. This domain presents two types of rewards - one is small, but frequent and immediate, while the other one is large, and also increase future rewards, but sparse and delayed. In order to face this challenge, we define intrinsic rewards, aiming at providing immediate feedback for some simplified subgoals we defined. Each of the subgoals is represented by a different Markov Decision Process (MDP) and policy $\pi_{subgoal}$ and taught using a different Q-network. At any given time, the subgoal controller activates one of the subgoals according to their set of initiation states I and termination states β, and the policy $\pi_{subgoal}$ corresponding to that subgoal is used to determine the action. The selected subgoal is used for a random number of steps $\tau \in [\tau_{min}, \tau_{max}]$. This randomization helps improve exploration for our agent. **The controller is programmed in a way that it provides the long term planning** needed for the agent to pursue the large and sparse reward.

3.2 Understanding Domain's Hierarchy

As presented in (Mnih et al. (2015)), Vanilla DQN reaches scores much lower than that of a human expert, even though it was used for a the parametric grid search for hyper parameters in Vanilla DQN. We suggest that the reason is the hierarchical structure of the game, which is easily conceived by humans but not by the agent. Collecting a single diver has no reward, but collecting 6 divers and surfacing provides a big reward and "levels up" the agent, increasing future reward. The Vanilla DQN agent excels at killing enemies, which provides a small but frequent reward, without perceiving the potential for a big delayed reward after completing the sequence explained above. The approach we take is to divide the sequence into smaller "subgoals", each of them alone is not sufficient to fully exploit the game's reward system, but when combined together and activated correctly, it has the potential to beat human experts.
This hierarchical approach is what allows us to use the DQN with some modifications in order to learn each of the subgoals, similarly to the Vanilla DQN.

3.3 Subgoal Learning

In our environment, different subgoals are learned concurrently using the same network (Figure 2). We design a reward signal fitting the target of each subgoal, and use it to train the corresponding subgoal. The DQN algorithm is composed of two stages: learning and playing.
During the testing stage, the subgoal controller (section 3.1) activates one of the subgoals. Using the subgoal σ we can select which part of the Q-networks is used to select the primitive action a and get the reward r for the selected action. We record the active subgoal σ_t.
Learning is performed by sampling random tuples from the ER memory and feeding them to the Q-network corresponding to the active skill in each tuple. When sampling tuples from the ER memory we only sample tuples for which σ_t is used for both s_t and s_{t+1}, i.e $\sigma_t = \sigma_{t+1}$. Since different subgoals approximate different Q functions, choosing a tuple with $\sigma_t \neq \sigma_{t+1}$ can provide an incorrect gradient when forwarded, and disrupt the learning of each of the splits in the output layer. By approximating the different Q-functions separately, we managed to eliminate the need for reward clipping, and we use different reward signals, in different scales, for each of the subgoals.

3.4 Reward Shaping

As explained in section 3.1, each subgoal has a different reward signal. Due to the hierarchical structure of our environment (Section 3.2) and the delayed rewards, we need to break down each of the complex goals required in the game to simpler subgoals. This break down is achieved by treating each sub goal as a different option and rewarding it.
The rewards given for each of the subgoals are frequent and immediate compared to the sparse and delayed reward typical in our environment, providing the agent necessary feedback in order to learn.

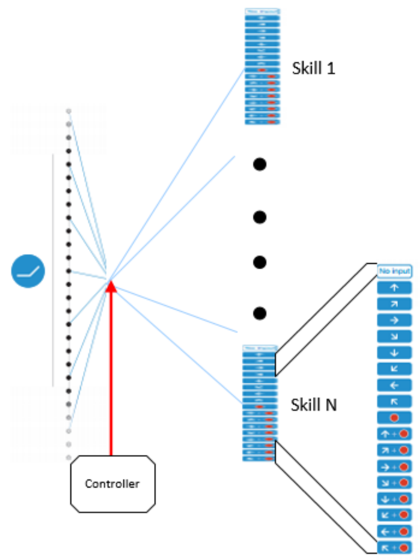


Figure 2: Last output layer of the MSDQN architecture, with separate set of primitive actions for each subgoal

3.5 Multi Subgoal Neural Network Architecture

For each of the subgoal, a different Q-network should be used to approximate its Q-function. Instead, according to Rusu et al. (2016), we use a single Q-network to learn the policies of each of the subgoals. The network used for learning multiple subgoals is based on the Vanilla DQN architecture, but instead of the last fully connected layer, with one neuron representing each primitive action, we implement a split last layer, where each part is a fully connected layer representing each subgoal. Separate subgoals learn different values for each of the primitive actions, and the executed action is chosen according to the active subgoal. The architecture of the Multi Subgoal DQN (MSDQN) is described in Figure 2.

4 Experiments

Our domain for research was Seaquest. This game is suitable for our purposes since there are two ways to gain rewards: one is frequent and grants small rewards, while the other grants large rewards but requires a long sequence of actions. We designed new reward signals and a controller that match the objectives above. In table 2 we summarize the details of our design. By setting the intrinsic rewards as described in the table, we provide a more frequent and immediate reward for both surfacing for air and collecting a diver (the reward for shooting enemies remains the same).
We use a controller programmed specifically for Seaquest, which receives an image of the current game state as its input and then choose which subgoal to pursue according to the control sequence defined in Figure 3.

4.1 Experiments Setup

In order to evaluate the effects of the techniques we used, we isolate the 2 key components of our implementation - intrinsic rewards and policy distillation. For this purpose, we ran 4 different simulations. The details of each of the configurations are given in Table 3. Running without policy distillation and intrinsic rewards is equivalent to the simulations presented in Mnih et al. (2015), and was used as our baseline. In order to measure the success in learning each of the skills, we tracked the number of times the agent has completed each subgoal in the evaluation episodes, and averaged it over the number of games played. We also recorded the average score for each trajectory in the evaluation episodes, in order to evaluate the success of the long term strategy using the learned skills.

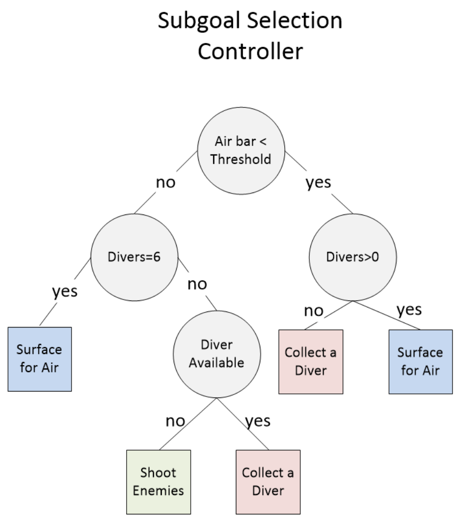


Figure 3: Decision tree used to choose the active subgoal.

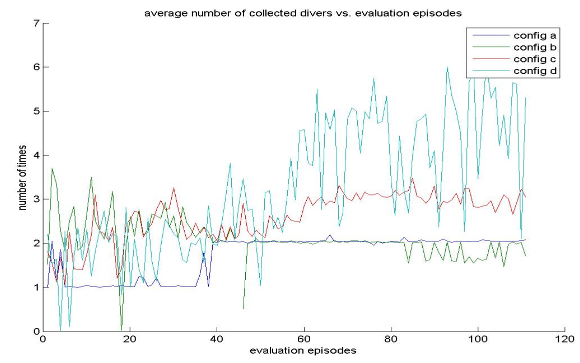
Subgoal	Intrinsic reward	Initiation set	Termination set
Collecting a diver	100× the number of currently collected divers	1. Diver available on screen & less than 6 currently collected 2. Air bar is below some threshold and there are 0 diver collected	Collected a diver or diver is not available on screen
Surfacing for air	1000	1. Air bar is below some threshold and there's at least 1 diver collected 2. There are 6 divers collected - surfacing in order to level up	Surfaced for air
Shooting enemies	Same as the game reward for this objective	Conditions for the other two are not met	Enemy shot or air bar is below some threshold or a diver became available on screen

Table 2: Intrinsic rewards and initiation and terminations sets in Seaquest

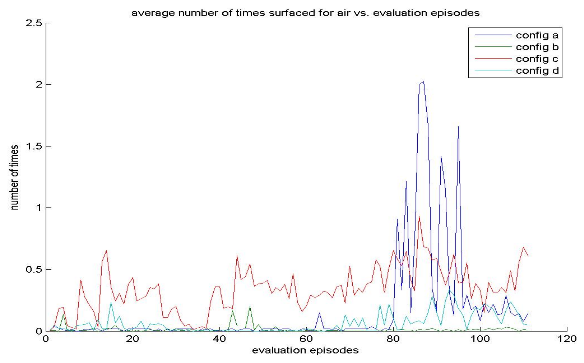
	Intrinsic Rewards	No Intrinsic Rewards
Policy Distillation	Config. a: MSDQN architecture and dedicated reward signal for each skill	Config. b: MSDQN architecture, using Seaquest's reward signal
No Policy Distillation	Config. c: Vanilla DQN architecture, dedicated reward signal for each skill	Config. d: Vanilla DQN architecture, using Seaquest's reward signal

Table 3: Simulations configurations

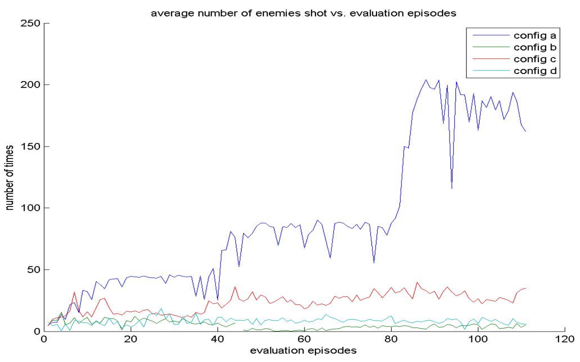
Figure 4: Results for simulations using all 4 configurations



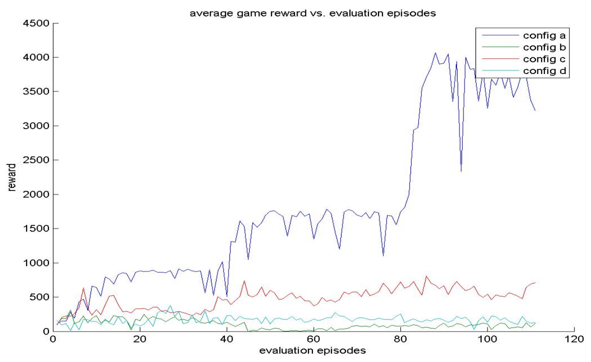
(a) Average number of collected divers for trajectory vs. number of evaluation episode



(b) Average number of times the agent surfaced for air for trajectory vs. number of evaluation episode



(c) Average number of enemies shot for trajectory vs. number of evaluation episode



(d) Average reward for trajectory vs. number of evaluation episode

4.2 Results

In figure 4 we present the results for simulations using all the configurations described in table 3. Each sub-figure includes 4 lines, representing the results for all the configuration from table 3.

The results from our experiments, shown in figs. 5(a) to 5(c) show notable progress in the performance for each of the subgoals our agent is trained for. For example, comparing the average number of divers collected (figure 5(a)) for Configs. (a) (Vanilla DQN) and (d) shows that using both intrinsic rewards and policy distillation provided around 5x improvement (from 2 to 10). Figure 5(a) also shows that Config. (b) (policy distillation and no intrinsic rewards) provides similar results to that of Vanilla DQN, while Config. (c) (intrinsic rewards with no policy distillation) provides some improvement compared to Vanilla DQN, but not as much as Config (d). Similar conclusions can be drawn regarding the ”surface for air” subgoal. The exception is the ”shoot enemies” subgoal, for which Vanilla DQN notably outperforms all other configurations, since Seaquest’s reward for this subgoal is frequent and immediate. This is the original problem we set out to solve - the agent focusing on the small reward for shooting enemies without comprehending the more complex layers of the game - collecting 6 divers and surfacing. In terms of overall score (figure 5(d)), Vanilla DQN outperforms all other configurations. This is owed to the fact that in order for configs. (b)-(d) to compete with Vanilla DQN, the agent must surface with 6 divers at least twice each episode. Our agent has not been able to do so in the simulations, since it only rescues about 10 divers at most, and the subgoal controller initiates the ”collect a diver” option for long periods of time, preventing the agent from shooting enemies and gaining smaller rewards.

4.3 Debugging and Monitoring

Since fully training a DQN agent is a tedious task, it was important for us to determine whether a simulation is on the right track or not as soon as possible, and to locate bugs. In order to do so, we used 2 main methods:

Baseline Comparison: We used the graphs from the Vanilla DQN (the weights and gradients of all the layers, along with the score for evaluation episodes) as a reference and compared graphs from our simulations to it. We didn’t expect the graphs to be the same, but it was a good sanity check to see that the weights are in the same order of magnitude and that the trend seems similar.

Coverage: We monitored a specific set of states that is interesting for our algorithm, and verified that the agent behaved correctly in these situations. For example, we wanted to see that the agent always chooses the air subgoal when it gets the 6th diver, and that it tries to collect divers rather than kill enemies when it is possible. using these techniques we discovered bugs in our code such as one of the output layers never being accessed (due to a typo when accessing some variable), which the coverage check immediately pointed out, and bugs like not forwarding states correctly in the Q-network when in the evaluation epochs, that were revealed by comparison to the baseline simulation values, and realizing there's a bug.

5 Discussion

In this work, we presented two main contributions -

- Embedding intrinsic rewards in a hierarchical domain (Seaquest)
- Applying a compressed architecture in order to learn skills needed for the agent to play according to a long term strategy

Simulation we ran in order to evaluate the effect of each of the techniques showed that they both have a significant effect on the policies learned by the agent for each skill. When using both intrinsic rewards and policy distillation, the agent has succeeded in learning complex tasks not learned in Vanilla DQN (collecting 6 divers and surfacing for air), but a rate not sufficient for it to beat the score set by Vanilla DQN. The solution we presented in this paper is limited to the Seaquest domain since it requires a lot of manual feature extraction (number of diver collected, tracking the air bar, etc...), and programming a controller to invoke a strategy we defined is ideal (collect 6 divers, surface and repeat). This of course limits the generality of our solution, and so we propose as future work to learn the controller's policy for choosing skills. i.e. having the controller rewarded for its choices of skills and learn according to the reward. Another idea for future work we propose is to re-examine the model of the MDPs represented by each of the subgoals, and try to pass different termination signals to each of the skills in order to simplify the MDPs.

References

Emilio Parisotto, Jimmy Ba, R. S. 2016. actor-mimic: deep multitask and transfer reinforcement learning.

Guo, X.; Singh, S.; Lee, H.; Lewis, R. L.; and Wang, X. 2014. Deep learning for real-time atari game play using offline monte-carlo tree search planning. 3338–3346.

Guo, X.; Singh, S. P.; Lewis, R. L.; and Lee, H. 2016. Deep learning for reward design to improve monte carlo tree search in ATARI games. *CoRR* abs/1604.07095.

Lin, L.-J. 1993. Reinforcement learning for robots using neural networks. Technical report, DTIC Document.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. *CoRR* abs/1602.01783.

Rusu, A. A.; Colmenarejo, S. G.; Desjardins, G.; Kirkpatrick, J.; Pascanu, R.; Mnih, V.; Kavukcuoglu, K.; and Hadsell, R. 2016. policy distillation.

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

Schmidhuber, J. 2010. Formal theory of creativity, fun, and intrinsic motivation (19902010). *Autonomous Mental Development, IEEE Transactions*.

Singh, S.; Lewis, R.; Barto, L.; Andrew, G.; and Sorg, J. 2010. Intrinsically motivated reinforcement learning: An evolutionary perspective. *Autonomous Mental Development, IEEE Transactions*.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112(1):181–211.

Szepesvari, C.; Sutton, R. S.; Modayil, J.; and Bhatnagar, Shalabh, e. a. 2014. Universal option models. *Advances in Neural Information Processing Systems*.

Tejas Kulkarni, Karthik Narasimhan, A. S. J. T. 2016. Deep reinforcement learning with temporal abstraction and intrinsic motivation. 48.

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

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