Chapter 3
Deep Reinforcement Learning Models

In this chapter, we will discuss about multiple DRL models in details.
General Functionalities of Deep Models

3.1 Value Function Approximation

Deep value function approximators are designed to learn value functions directly. Policies may be designed as the algorithm outputs as a side result from the learned value functions from neural networks.

3.1.1 Deep Q-learning

One of the main bottlenetwork of Q-learning is its incapability to deal with reinforcement learning problems with a large state and action space which needs to be represented by high-dimensional data. For instance, agent control from high-dimensional sensory inputs such as vision and speech was one of long-standing challenges of reinforcement learning before deep Q-learning was developed. Fig. **??** provides sample screenshots from five of the games that can be automated by reinforcement learning. Most successful reinforcement learning applications that operate on these domains have relied on carefully designed features combined with simple value functions or policy representations, e.g. linear and basis. The performance of such methods heavily relies on the quality of the feature representation and is hard to generalize.

Advances in deep representative learning through neural networks have made it possible to extract high-level features that can be generalized from raw high-dimensional data like sensory data. They are naturally integrated into reinforcement learning to improve representative power of the input features generated from raw high-dimensional data related to agents' observations and behaviors. Reinforcement learning presents several challenges before these deep learning techniques can be applied. Firstly, reinforcement learning algorithms must be able to learn from a scalar reward signal that is frequently sparse, noise, and delayed. The delay between ac-

tions and their immediate rewards can be thousands of timesteps long for a complex reinforcement learning problem. Successful deep learning networks have required large amounts of labelled training data from which direct association between inputs and targets are found. How to collect such labelled training data for reinforcement learning on an online or semi-online pattern needs to be resolved before applying deep learning to solve reinforcement learning problems. Secondly, most deep learning algorithms assume independent data samples while samples in reinforcement learning problems are mostly highly correlated sequential data. Finally, the sample distribution in reinforcement learning can change as new behaviors are learned, or along time as system dynamics change. This can be problematic for deep learning algorithms that perform good under the fixed data distribution.

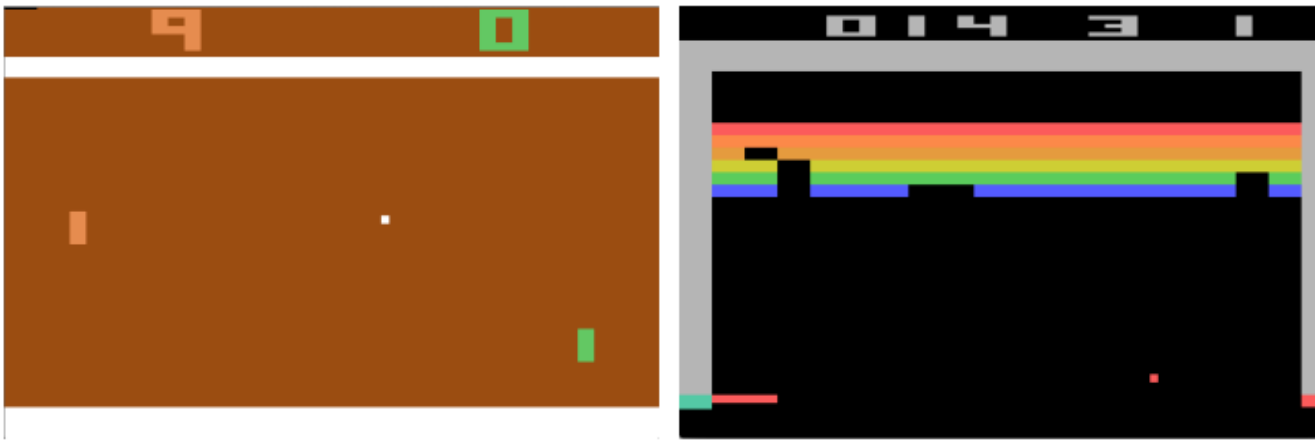

Fig. 3.1: Screen shots from five Atari 2600 Games: (Left-to-right) Pong, Breakout, Space Invaders, Seaquest, Beam Rider. The figure is copied from [5]

The deep DQN networks are one of the first deep reinforcement learning algorithms that handles above challenges successfully. Deep DQNs approximate the optimal (action, state) value function Q and some times action value function V in Q-learning **??** using convolutional neural networks as:

$$Q^*(s,a) = \max_{\pi} \mathbf{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi], \tag{3.1}$$

which aims to maximize the expected overall future return of the action sequence achievable by the policy π when making the action a in the state s at the time step t.

Q-learning algorithms are generally model free. The RL learning tasks are directly solved from real experiences or from samples generated by emulator of physical systems, without explicitly construct or learn the environment models. These algorithms are usually off-policy to allow behavior policies that ensure adequate exploration. The behavior policies are often constructed with a ϵ-greedy strategy that follows the target policy with probability $1 - \epsilon$ and selects a random action with probability ϵ.

The first formal Deep Q-learning work [5] developed a variant Q-learning algorithm using a convolutional neural network to learn the optimum policy. The deep Q-learning algorithm targets to learn the optimum value function $Q^*(s, a)$ as:

$$Q^*(s, a) = \mathbf{E}_{s' \sim \mathcal{S}}[r + \gamma \max_{a'} Q^*(s', a')|s, a]. \tag{3.2}$$

Instead of iteratively updating value function by using the Bellman equation for each experience sequence, deep Q-learning using a neural network approximator to directly estimate the parameterized optimium action value function $Q(s, a|\Theta) \approx Q^*(s, a)$. This makes it possible the generalization of learned value functions with respect to unknown experience or partially observable environment. The neural network function approximator with weights Θ is referred as a Q-network. Denotes the target for the ith iteration of the optimization process as y_i:

$$y_i = \mathbf{E}_{s' \sim \mathcal{S}}[r + \gamma \max_{a'} Q(s', a'|Theta_{i-1}|s, a)], \tag{3.3}$$

the Q-network can be trained minimizing a loss function $L(\Theta)$ over training samples:

$$L_i(\Theta_i) = \mathbf{E}_{s,a \sim \rho(s,a)}[(y_i - Q(s, a|\Theta_i))^2], \tag{3.4}$$

where $\rho(s, a)$ is the behavior distribution over state-action pairs. Then, the parameter gradient is calculated as:

$$\nabla_{\Theta_i} L_i(\Theta_i) = \mathbf{E}_{s,a \sim \rho(s,a); s' \sim \mathcal{S}}[(r + \gamma \max_{a'} Q(s', a'|\Theta_{i-1}) - Q(s, a|\Theta_i)) \nabla_{\Theta_i} Q(s, a|\Theta_i)]. \tag{3.5}$$

Stochastic gradient descent is used for parameter update with the gradients calculated. Generally, any neural network optimization algorithms that minimizing a loss function may be used for the parameter update. When the weights are updated every time-step of experience, we arrive at the basic Q-learning algorithm. N-step DQN [9] does the value update every multiple steps for faster convergence, but this may increase the chance of learning divergence due to omitting the value maximization at the intermediate steps resulting in optimal policy mismatch. The complete deep Q-learning algorithm is defined as below:

Special handling of the training samples are needed to make Deep Q-learning works well. Firstly, most neural network optimization algorithms requires independently distributed samples. However, samples generated by the exploitation and exploring process in an enviornment, this requirement is not satisfied. Replay buffer is used to address the issue. The replay buffer is a finite-sized cache that stores transition tuples (s_t, a_t, r_t, s_{t+1}). The process to sample the replay buffer for network training is called as experience replay. Experience replay randomly samples pre-

Algorithm 8 Deep Q-learning with Experience Replay

Initialize experience replay buffer with capacity N
Initialize action-value network Q with random weights
Algorithm parameter: ϵ for probability of exploration experience
for each episode **do**
Initialize state $s_1 = x_1$
for each time step t **do**
with probability ϵ, select a random action a_t
otherwise, $a_t = \max_a Q^*(\phi(s_t), a|\Theta)$
execute a_t and observe reward r_{t+1} and state image x_{t+1}
set $s_{t+1} = x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
store transition $(\phi_t, a_t, r_{t+1}, \phi_{t+1})$ in the replay buffer
sample random minibatch of transitions $(\phi_i, a_i, r_{i+1}, \phi_{i+1})$ from the replay buffer
set

$$y_i = \begin{cases} r_{i+1} & \text{for terminal state} \\ r_{i+1} + \gamma \max_{a'} Q(\phi_{i+1}, a'|\Theta) & \text{for non-terminal state} \end{cases} \tag{3.6}$$

perform parameter update based on gradient descent on $(y_i - Q(\phi_i, a_i|\Theta))^2$
end for
end for
Return $\pi = \pi^*$ and $V = v^*$

vious experience in the replay buffer to construct additional training, this removes the correlations in the observed experiences and stabalize the data distribution used for training. To perform the experience replay, the agents' experiences are stored in e.g. cache. The stored experiences are randomly sampled, batched, and used as the training data to update the Q-network in experience replay stages.

Seondly, the input features of the physical system vary a lot in range and distribution properties. This makes the networks difficult to learn effectivaly and converge to generalized hyper-parameters across environment with different scales and dynamics. Two approaches may be applied to alleviate this problem: 1) scale the features in similar ranges across dimensions and environments; 2) batch normalization which normalize each dimension, in an online way, across samples in a minibatch, e.g. so that each dimension has unit mean and variance. Batch normalization is more computationally expensive but is more generalized. The batch normalization layer can be used to normalize the inputs of each selected network layers.

A series of works extend the DQN to increase the learning performance, including Double DQN [6] and [10], Dueling DQN, and Categorical DQN [2].

3.1.2 Double DQN

In DQN algorithms, a small update to Q may significantly change the current optimum behaviors and thus the correlations between the optimum behavior values, represented by the network parameters or the target optimum values $Q^*(s, a)$. Two variations are usually used to alleviate the problem: minibatch with experience replay and double Q-networks. The minibatch training samples are generated from the buffer by uniform sampling, which can greately decrease the correlation across batched samples.

To stabilize the Q-value updates and thus improve the learning convergence, Double DQN stabalizes the training by using two networks: a target netowrk which learns from experience and is periodically update the target network, and a behavior network which generates the experience for training and udpates its parameters continuously during the learning. The Double DQN methods can be considered as a simplified actor-critic version without explicitly specify the stochastic policies.

The loss function is usually a function of expected distance between the target values and the estimated values:

$$L_t(\Theta_t) = \mathbf{E}_{(s,a,r_{t+1},s')\sim\mathcal{U}(D)}, \tag{3.7}$$

where r_{t+1} is the immeidate reward of a_t observed at the next time step $t+1$, γ is the discount factor. There are multiple ways to parameterizing the value function using CNNs. Basically, Q-netowrk maps previous experience, which can be represented by the current state according to MDP or semi-MDP, to estimates of their Q-values. Any information about the history actions and current states can be used as the model inputs. For instance, history-action pairs have been used as the inputs to Q-netowrks such as multi-layer perceptron [7]. The main disadvantage of this type of network architecture is that a separate forward path for each action is required to compute the Q value of each action, because with respect to the reward and long-term value of actions, actions are generally different from each other and may not be generallized well.

Most Q-networks with a CNN architecture use the state representation as the inputs for efficient information sharing among actions. Then, a seprate output unit or set of output units are constructed for each possible action, corresponding to the Q-values of the action under the input state. For these architectures, only a single path through the network is sufficient because action Q-values can be generalized well conditioned on the input state, according to the property of MDP and semi-MDP. Experimental experiments and simulations also demonstrate that the generalization is common even when the constraints of MDP and semi-MDP are relaxed.

Fig. **??** shows the exact architecture used in the first formal publication on DQN [6]. Each hidden layer is followed by a rectifier layer. The number of actions in problems studied is between 4 and 18, which is comparatively small. During the training, the rewards are usually normalized or clipped to a smaller range, e.g. $[-1, 1]$ to limit the scale of error derivatives which further stabalize the training.

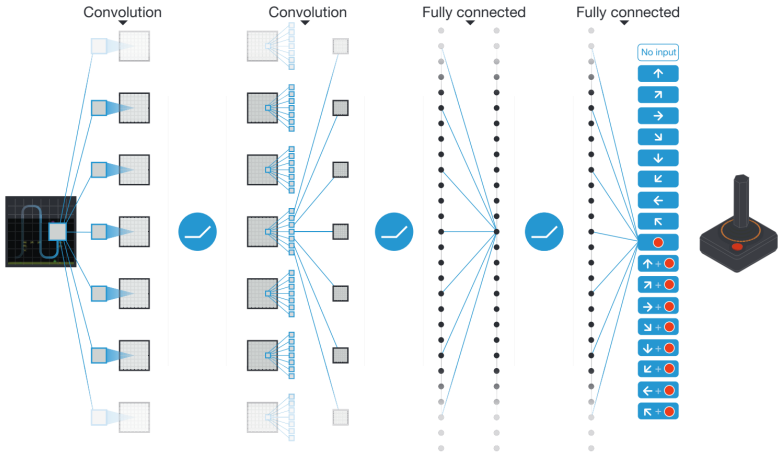

Fig. 3.2: Sample architecture of a CNN deep Q-learning network. The inputs to the network are images, each represents the state associated with the target Q-value. The image is preprocessed followed by three convolutional layers and then two fully connected layers with a single output for each valid action. The figure is copied from [6].

Algorithm-wise, this version of Double DQN is the minimal possible change to DQN in Algorithm **??** towards using two Q-networks to improve DQN methods, with only the target at the ith iteration y_i be changed to

$$y_i = r_{i+1} + \gamma \max_{a'} Q(\phi_{i+1}, a'|\Theta^-). \tag{3.8}$$

We can see that instead of using the target value generated by a continuously updated Q-network, the Double DQN uses the target value generated by the comparatively stable target network. Notes that the index of r_{i+1} does not have a numeric meaning but simply denotes that r is the immediate reward of the associated action in the ith iteration.

Another version of Double DQN [10] is inspired by the double Q-learning algorithm in section **??**. It decouples policy evaluation and action selection by using two networks in order to prevent overselect of actions with overestimated values: a behavior network from which its Q values are used to generate experience for network updates and a target network which is in charge of policy evaluation over the experience generated by the behavior network. The target network is updated every N step with the trained behavior network. Different from [6], the target value for each time step is calculated using the same value update equation but with the optimum action selected by the behavior network instead of by the target network. Assuming the target network is parameterized with Θ^- and the behavior network is parameterized with Θ, the target value at step t is:

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a|\Theta_t)|\Theta^-{}_t). \tag{3.9}$$

We can see that the behavior network is used for optimum action selection in the policy evaluation process by the target network parameterized with Θ^-. This scheme effectively decouples action seelction and policy evaluation for network updates and thus effectively prevents value overestimation of the target network.

3.1.3 More Advanced DQN Methods

One of the main weakness of DQNs is their adaptation to RL problems with a continuous action space. For those problems, discretization of the action space is proceeded to make the networks compatible to the target outputs. However, the obvious limitation is the curse of dimensionality, specifically, the number of actions increases exponentially with the degrees of freedom. For instance, a 7 degree of freedom with 5 different action in each degree results in $5^7 = 78125$. The situation is even worse for tasks that require finer grained discretization in multiple degrees, leading to an explosion of the number of discrete actions and thus of the number of network outputs. How to discretize the action space is important for the model performance. Simple discretization of the action space such as uniform sampling throw away important information about the structure of the action space. More complex algorithms like deep actor-critic algorithm section **??** is proposed to better solve RL problems with a continuous action space.

Dueling DQN [8] decomposes the Q-value into the summation of the state value that measures the value of a state and state-action value specific to a particular action in the state. The proposed network improves the training stability and convergence speed. Categorical DQN [1] predict the discretized distributions of values/actions instead of a single distribution or probability vector of values/actions. While the more accurate value modeling is demonstrated to improve the learning performance of more than half of the games experimented on, the complexity of the approach increases linearly with the dimension of the value/action probability that can be large in real-world RL problems.

Mathematically, the action advantage function is defined as the distance between Q value function and the corresponding state value function:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s). \tag{3.10}$$

It's easy to conclude that $\mathbf{E}_{a\sim\pi(s)} = 0$. Iuitively, the action advantage function substracts the state value from Q value to obtain a measure of relative importance of each action. Then the target output of the aggregating module can be defined as:

$$Q(s, a|\Theta, \alpha, \beta) = V(s|\Theta, \beta) + A(s, a|\Theta, \alpha). \tag{3.11}$$

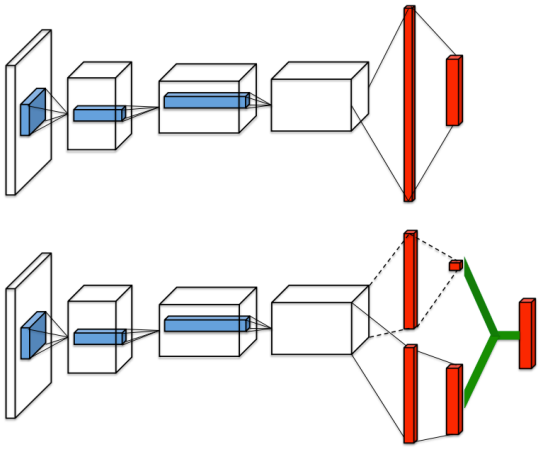

Fig. 3.3: Sample architecture of a Dueling DQN network and that of its DQN counterpart. The inputs to the network are images, each represents the state associated with the target Q-value. In the DQN couterpart, the image is preprocessed followed by three convolutional layers and then two fully connected layers with a single output for each valid action. The Dueling DQN introduces two intermediate output layers before the final Q function output, representing value function output and action advantage function output respectively. And an aggregating layer is added in the end to output the Q values. The figure is copied from [8].

The mapping from Q to (V, A) is not unique, as a result we cannot recover unique V and A from a given Q. Furthermore, both V and A can be large values that are not negligible. These make the learning unstable. To address this issue, we force the input from the action advantage function the last module to be close to zero. Then, the output layer of the network is expressed as:

$$Q(s,a|\Theta,\alpha,\beta) = V(s|\Theta,\beta) + (A(s,a|\Theta,\alpha) - \max_{a'\in\mathcal{A}} A(s,a'|\Theta,\alpha)). \tag{3.12}$$

We can see that for the optimum action $a^* = argmax_{a'\in\mathcal{A}} Q(s,a'|\Theta,\alpha,\beta) = argmax_{a'\in\mathcal{A}} A(s,a'|\Theta,\alpha)$, and $Q(s,a^*|\Theta,\alpha,\beta) = V(s|\Theta,\beta)$. This is consistent to the original design that the two intermediate output layers $\dot{A}()$ and $\dot{V}()$ provide the estimate of action advantage function and state-value function, respectively. It's easy to notice that we can consider $(A(s,a|\Theta,\alpha) - \max_{a'\in\mathcal{A}} A(s,a'|\Theta,\alpha))$ as a constrained action advantage function that is of small values. Practically, this modification increases the learning stability obviously.

Alternative output modules are possible, applying similar constraint on the action advantage function. For instance, we can replace the max operator with an average. And the output module is then:

$$Q(s,a|\Theta,\alpha,\beta) = V(s|\Theta,\beta) + (A(s,a|\Theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'\in\mathcal{A}} A(s,a'|\Theta,\alpha)). \quad (3.13)$$

This version of modified action advantage function further improves the learning stability as the advantages have a zero mean and change slowly along the change of the advantage mean, instead of that of the optimal action's advantage. To further control the learning accuracy and statbility of action advantages and/or state-values, partial losses over action advantages and/or state-values may be added to the total loss.

3.2 Policy Approximation

The majority of modern policy network models can learn policies end-to-end directly from raw representations, e.g. raw pixels, of the environment.

3.2.1 Deep Policy Gradient Method

Policy gradient methods are widely used for RL problems with a continuous action space. The basic idea is to parameterize the stochastic policy $\pi_\theta(a|s) = P[a|s|\theta]$ that selects actions in state any particular s, so that the model parameters determine the way of action selection. During the model optimization, the parameterized policy is sampled and the parameters are adjusted in the direction of greater accumulated rewards or values.

3.2.2 Deep Actor-Crtitic Method

Deep Actor-Critic algorithms that use deep networks as the function approximators are able to find policies whose performance is as good as those found by a delicatedly designed planning algorithm with full access to the system dymanics. Some of the algorithms operate over continuous action spaces based on deterministic policy gradients.

The Deep DPG (DDPG) approach [3] is developed based on the actor-critic method with a deterministic actor and a critic learned through Q-learning as described in chapter **??**. The algorithm can learn competitive policies using low-dimensional observations. The approach requires only a actor-critic architecture and a straight-forward learning algorithm with very few other parts, which makes the approach easy to implement and scale to more difficult and complex prolems.

One major learning component in continuous spaces is the exploration scheme. For off-policy algorithms, the exploration can be conducted independently from the learning from exploitation. In DDPG, an exploration policy π' is constructed as a noised version of the actor policy:

$$\pi'(s_t) = \pi(s_t|\Theta_\pi) + \mathcal{N} \tag{3.14}$$

where $\mathcal{N}$ is the noise process, which can be chosen to fit the noise distribution of the environment dynamics. The original DDPG uses an Ornstein-Uhlenbeck process [3] to generate temporally correlated efficient exploration for physical control problems with intertia. The complete DDPG algorithm is defined in Algorithm **??**.

Algorithm 9 Deep Deterministc Policy Gradient Algorithm

Initialize the actor network $\pi(s|\Theta_\pi)$ and the critic network $Q(s, a|\Theta_Q)$ with network weights Θ_{pi} and Θ_Q.
Initialize target network Q' and π': $\Theta_{Q'} \leftarrow \Theta_Q$ and $\Theta_{\pi'} \leftarrow \Theta_\pi$
Initialize replay buffer R and minibatch size N

for each episode **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Initialize state as s_1
  for each time step **do**
    Select the action with exploration $a_t = \pi(s_t|\Theta\pi) + \mathcal{N}$ under policy π
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store experience (s_t, a_t, r_t, s_{t+1}) in R
    Sample a minibatch of experiences in R
    $y_t \leftarrow r_t + \gamma Q'(s_{t+1}, \pi'(s_{t+1}|\Theta_{\pi'})|\Theta_{Q'})$
    Update the critic by minimizing the loss $L = \frac{1}{N}\sum_t (y_t - Q(s_t, a_t|\Theta_Q))^2$
    Update the actor policy using the sampled policy gradient:
    $\nabla_{\Theta_\pi} J \approx \frac{1}{N}\sum_t \nabla_a Q(s, a|\Theta_Q)|_{s=s_t, a=\pi(s)} \nabla_{\Theta_\pi} \pi(s|\Theta_\pi)|_{s_t}$
    Update the target networks:
    $\theta_{Q'} \leftarrow \tau\Theta_Q + (1-\tau)\Theta_{Q'}$
    $\theta_{\pi'} \leftarrow \tau\Theta_\pi + (1-\tau)\Theta_{\pi'}$
  end for
end for

In the implementations of the algorithm, different exploration method like the ϵ-greedy may be used for action exploration. To further stabalize the learning process, the target networks may be updated every M step. Special handling of the training samples is needed to make deep actor-critic method works well, beyond the one used for deep Q-networks.

Firstly, the DDPG was designed as an off-policy algorithm, the replay buffer can be very large; and with a large replay buffer the sampled minibatch data are uncorrelated transitions which are suitable for neural network learning. Secondly, directly deep Q-networks were proved to be unstable in many RL enviornments. The same networks used to learn the target vlaues as the ones used to generate experiences are updated continuously, which leads the critic prone to diverge. The

DDPG algorithm uses target updates with momentum. Particularly, a copy of actor and critic networks, $Q'(s, a|\Theta_{Q'})$ and $\pi(s|\Theta_{\pi'})$, are used to calculate the target values. Then, the weights of the two target networks are updated with the learned networks: $\Theta' \leftarrow \tau\Theta + (1-\tau)\Theta'$ with $\tau \ll 1$. This soft way of network weights update changes the weights slowly and thus greatly improves the stability of the learning procedure. The soft weights update sometime seems to slow down the learning by delaying the value updates. In practice, the improved learning stability outweights the local slowing down of updated value propagation which generally improves the overall learning efficiency.

3.2.3 Asynchronous Advantage Actor-Critic

A3C [4] developed a deep reinforcement learning framework trained with asynchronous gradient descent for the optimization of deep neural network controllers. The idea is to asynchronously execute multiple agents in parallel on multiple instances of the environment, so that an agent does not affect the policies of other agents. This scheme provides similar functionalities of experience replay, such as decorrelates the agents' experience and makes the agents' data more stationary locally in time, since in this way the parallel agents will experience a variety of different states. It enables deep learning techniques to be applied robustly and effectively in a much larger spectrum of fundamental on-policy reinforcement learning algorithms, such as Sarsa, N-step methods, and actor-critic methods, as well as off-policy reinforcement learning algorithms such as Q-learning. Deep asynchronous advantage actor-critic (A3C) outperforms the others on 2D and 3D games with discrete and continuous action spaces. Its ability to train feedforward and recurrent agents makes it the most general and successful reinforcement learning agent to date.

Asysnchronous reinforcement learning trains agents asynchronously. Infrastructure wise, there are generally two ways to deploy the agents: 1) use multiple learner machines and a parameter server machine. 2) use multiple threads in a single machine. Keeping multiple learners and parameter server on a single machine remvoes the communication costs of sending gradients and distributing parameters. Multiple actor learners running in paprallel are likely to exploring different part of the action and state spaces. Furthermore, one can explicitly use different exploration policies and even slightly different exploitation policies in each actor-learner to maximize the diversity and thus improve the convergence as a whole. Furthermore, by running different exploration policies parallelly, the online parameter changes being made by multiple actor-learners are likely to be less correlated in time than updates made by a single agent. Thus, instead of using experience replay, A3C relies on parallel actor-learners employing different exploration policies to stabalize the learning process. Since experience replay is no longer in need to stabilize the training, we are able to use The on-policy actor-critic methods.

In A3C, asysnchronous actor-learners are designed to learn the optimum policy $\pi(\Theta)^*$. Each paired with a critic to estimate the optimum value function $V(\Theta_v)^*$. It

operates in the forward view and update both policy and the value function every N steps or when a terminal state is reached. Define the advantage function as

$$\mathcal{A}(s_t, a_t; \Theta, \Theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k|\Theta_v}) - V(s_t|\Theta_v), \tag{3.15}$$

where k is upper bounded by N and may vary state by state. The gradient update is:

$$\nabla_{\Theta'} log\pi(a_t|s_t, \Theta')A(s_t, a_t|\Theta, \Theta_v). \tag{3.16}$$

The complete A3C algorithm is presented in Algorithm **??**.

Algorithm 10 Asynchronous Advantage Actor-Critic Algorithm (operations for a single actor-learner)

Denote globally shared parameter vectors Θ and Θ_v, and globally shared counter $T = 0$.
Denote thread-specific parameter vectors Θ' and ${\Theta'}_v$.
Initialize thread step counter $t \leftarrow 1$.

while $T > N$ **do**
  Reset gradients: $d\Theta \leftarrow 0$ and $d\Theta_v \leftarrow 0$.
  Synchronize thread-specific parameters: $\Theta' = \Theta$ and ${\Theta'}_v = \Theta_v$
  $s_t \leftarrow s_{t+1}$
  $t_{start} = t$
  while $t - t_{start} < N$ or s_t is not a terminal state **do**
    Perform a_t according to policy $\pi(a_t|s_t, \Theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  end while

$$R = \begin{cases} 0 & \text{for terminal state s\_t} \\ V(s_t, {\Theta'}_v) & \text{for non-terminal s\_t; bootstrap from last state} \end{cases} \tag{3.17}$$

  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt Θ': $d\Theta \leftarrow d\Theta + \nabla_{\Theta'} log\pi(a_i|s_i|\Theta')(R - V(s_i|{\Theta'}_v))$
    Accumulate gradients wrt ${\Theta'}_v$: $d\Theta_v \leftarrow + \frac{\partial (R - V(s_i|{\Theta'}_v))^2}{\partial {\Theta'}_v}$
  end for Perform asynchronous update of Θ and Θ_v using $d\Theta$ and $d\Theta_v$
end while

The optimization method can be general gradient-based neural network optimization methods with minor changes. In the original work of A3C, three optimization

methods are introduced: SGD with momentum, RMSProp without shared statistics, and RMSProp with shared statistics.

Momentum SGD: denote the parameter vector that is shared across all threads as Θ and the accumulated gradients of the loss with respect to parameters Θ computed by thread i as $\Delta\Theta_i$, each thread i independently maintains its standard momentum SGD update and parameter update without any locks:

$$m_i = \alpha m_i + (1 - \alpha)\Delta\Theta_i, \tag{3.18}$$

$$\Theta \leftarrow \Theta - \eta m_i, \tag{3.19}$$

where α is the momentum and η is the learning rate. The main process can, or another thread may be created to, act as the parameter server to receive and distribute the accumulated moments and/or updated parameters, usually it can be gauranteed that parameters sent out from the virtual parameter server are globally-updated.

The gradient update in **RMSProp** methods is the standard non-centered RMSProp update given by:

$$g = \alpha g + (1 - \alpha)\Delta\Theta^2, \tag{3.20}$$

$$\Theta \leftarrow \Theta - \eta \frac{\Delta\Theta}{\sqrt{(g + \epsilon)}}, \tag{3.21}$$

where the operations are performed elementwise. The moving average of elementwise squared gradients g is shared when RMSProp with shared statistics is used for optimization and is per thread when RMSProp with shared statistics is used. In RMSProp with shared statistics, the vector $\mathbf{g}$ is shared among threads and is updated asynchronously without locking. Sharing statistics also improve memory efficiency by using one fewer copy of parameter vector per thread.

RMSProp methods are generally more stable and outperform SGD with a median increase in computation complexity. Experiments show that a variant of RMSProp with shared statistics across threads is considerably more robust then the other two methods. The performance of A3C with these three asynchronous optimization algorithms is compared in terms of their sensitivity to random network initialization and different learning rates. Fig. **??** shows the comparison on four different games (Breakout, Beamrider, Seaquest and Space Invaders) with async n-step Q as the reference reinforcement learning method.

Each curve shows the scores for 50 experiments with different learning rates and initializations. The x-axis shows the rank of the model in descending order by final average score. The y-axis shows the final average scores achieved. The more robust the algorithm is, the closer its slope to horizontal thus maximizing the area under the curve (AUC). The algorithm that performs better would get more rewards and thus final average score on the y-axis. We can see that RMSProp with shared statistics tends to be more robust than the other two optimization methods.

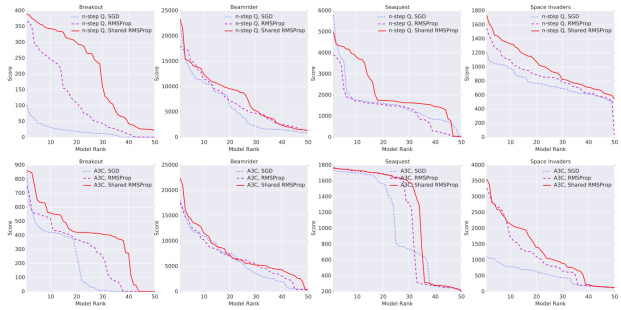

Fig. 3.4: Comparison of three different optimization methods (Momentum SGD, RMSProp, Shared RMSProp). Each individual graph shows results for one of the four games and three different optimization methods.

References

[1] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[2] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[3] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[4] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[7] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine learning: ECML 2005: 16th European conference on machine learning, Porto, Portugal, October 3-7, 2005. proceedings 16*, pages 317–328. Springer, 2005.

[8] Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[9] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[10] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30-1, 2016.

Glossary

Use the template *glossary.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your glossary in the Springer layout.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

References

[1] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[2] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Ng. An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems*, 19, 2006.

[3] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

[4] Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

[5] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13:341–379, 2003.

[6] Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021.

[7] Jacob Beck, Risto Vuorio, Evan Zheran Liu, Zheng Xiong, Luisa Zintgraf, Chelsea Finn, and Shimon Whiteson. A survey of meta-reinforcement learning. *arXiv preprint arXiv:2301.08028*, 2023.

[8] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[9] Léon Bottou. Online algorithms and stochastic approximations. *Online learning in neural networks*, 1998.

[10] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[11] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[12] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[13] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. *Advances in neural information processing systems*, 29, 2016.

[14] G Dulac-Arnold, N Levine, DJ Mankowitz, and etc. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110:2419–2468, 2021.

[15] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala:

Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[16] Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 29, 2016.

[17] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 1146–1155. PMLR, 2017.

[18] Justin Fu, Katie Luo, and Sergey Levine. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248*, 2017.

[19] Maor Gaon and Ronen Brafman. Reinforcement learning with non-markovian rewards. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3980–3987, 2020.

[20] Adam Gleave, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. Adversarial policies: Attacking deep reinforcement learning. *arXiv preprint arXiv:1905.10615*, 2019.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[22] Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Meta-reinforcement learning of structured exploration strategies. *Advances in neural information processing systems*, 31, 2018.

[23] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *Autonomous Agents and Multiagent Systems: AAMAS 2017 Workshops, Best Papers*, pages 66–83. Springer, 2017.

[24] Eric A Hansen, Daniel S Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715, 2004.

[25] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado Van Hasselt. Multi-task deep reinforcement learning with popart. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3796–3803, 2019.

[26] Todd Andrew Hester, Evan Jarman Fisher, and Piyush Khandelwal. Predictively controlling an environmental control system, January 16 2018. US Patent 9,869,484.

[27] Rein Houthooft, Yuhua Chen, Phillip Isola, Bradly Stadie, Filip Wolski, OpenAI Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. *Advances in Neural Information Processing Systems*, 31, 2018.

[28] Wenyi Huang and Jack W Stokes. Mtnet: a multi-task neural network for dynamic malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San*

Sebastián, Spain, July 7-8, 2016, Proceedings 13, pages 399–418. Springer, 2016.

[29] Mahdi Imani and Seyede Fatemeh Ghoreishi. Scalable inverse reinforcement learning through multifidelity bayesian optimization. *IEEE transactions on neural networks and learning systems*, 33(8):4125–4132, 2021.

[30] Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. *ICLR*, 2019.

[31] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[32] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[33] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[34] Frank L Lewis and Kyriakos G Vamvoudakis. Reinforcement learning for partially observable dynamic processes: Adaptive dynamic programming using measured output data. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 41(1):14–25, 2010.

[35] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[36] Robert Loftin, Aadirupa Saha, Sam Devlin, and Katja Hofmann. Strategically efficient exploration in competitive multi-agent reinforcement learning. In *Uncertainty in Artificial Intelligence*, pages 1587–1596. PMLR, 2021.

[37] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

[38] IA Luchnikov, SV Vintskevich, DA Grigoriev, and SN Filippov. Machine learning non-markovian quantum dynamics. *Physical review letters*, 124(14):140502, 2020.

[39] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[40] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[42] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis: proceedings of the biennial Conference held at Dundee, June 28–July 1, 1977*, pages 105–116. Springer, 2006.

[43] Daniel Neider, Jean-Raphael Gaglione, Ivan Gavran, Ufuk Topcu, Bo Wu, and Zhe Xu. Advice-guided reinforcement learning in a non-markovian environment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9073–9080, 2021.

[44] Andrew Y Ng, Stuart Russell, et al. Algorithms for inverse reinforcement learning. In *ICML*, volume 1, page 2, 2000.

[45] Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33:1060–1070, 2020.

[46] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[47] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

[48] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *International conference on machine learning*, pages 2817–2826. PMLR, 2017.

[49] Pascal Poupart, Aarti Malhotra, Pei Pei, Kee-Eung Kim, Bongseok Goh, and Michael Bowling. Approximate linear programming for constrained partially observable markov decision processes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[50] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21(178):1–51, 2020.

[51] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine learning: ECML 2005: 16th European conference on machine learning, Porto, Portugal, October 3-7, 2005. proceedings 16*, pages 317–328. Springer, 2005.

[52] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, 1998.

[53] Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[54] Nicolas Schweighofer and Kenji Doya. Meta-learning in reinforcement learning. *Neural Networks*, 16(1):5–9, 2003.

[55] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[56] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents. *arXiv preprint arXiv:1903.00784*, 2019.

[57] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[58] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[59] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[60] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[61] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.

[62] Yee Teh, Victor Bapst, Wojciech M Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

[63] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[64] Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. Androidenv: A reinforcement learning platform for android. *arXiv preprint arXiv:2105.13231*, 2021.

[65] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30-1, 2016.

[66] Wikipedia contributors. Stochastic gradient descent — Wikipedia, the free encyclopedia, 2024. [Online].

[67] Haiyan Yin and Sinno Pan. Knowledge transfer for deep reinforcement learning with hierarchical experience replay. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[68] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on robot learning*, pages 1094–1095. PMLR, 2020.

[69] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control*, pages 321–384, 2021.

[70] Zeyu Zheng, Junhyuk Oh, Matteo Hessel, Zhongwen Xu, Manuel Kroiss, Hado Van Hasselt, David Silver, and Satinder Singh. What can learned intrinsic

rewards capture? In *International Conference on Machine Learning*, pages 11436–11446. PMLR, 2020.