university of groningen / faculty of science and engineering

A Transfer Learning Analysis of the Deep Deterministic Policy Gradient Method

Bachelor's Project Thesis

Hamin Cheon, s3794644, h.cheon@student.rug.nl,
Supervisors: Dr. Matthia Sabatelli, m.sabatelli@rug.nl

Abstract: Deep Deterministic Policy Gradient (DDPG) is a type of Deep Reinforcement Learning (DRL) algorithm that is specialized in environments with continuous action space. One of the problems of DDPG, or Deep Learning (DL) in general, is that it is expensive to train. A possible solution to the problem is Transfer Learning (TL), a machine learning method that transfers the knowledge of a trained agent to another agent to give a quicker start on training. The paper investigates the effect of the memory buffer transfer and the model parameters transfer in training a DDPG agent. The three experiments, the regular Experience Replay (ER) buffer transfer, Prioritized Experience Replay (PER) buffer transfer, and model parameters transfer, were conducted to test the claim. The agents were trained in two OpenAI gym environments. When the first agent was trained in the first environment, its knowledge was collected to transfer to the second agent. The new agent was then trained in the second environment that was different but related to the first environment. The results showed that the memory buffer transfer hurts the training of the DDPG agent, while the model parameters transfer benefits the training.

1 Introduction

Deep Reinforcement Learning (DRL) combines artificial neural networks from Deep Learning (DL) with a framework of Reinforcement Learning (RL) (Arulkumaran et al., 2017). It is more advanced than a regular RL that can perform in more complicated environments. Its powerful learning skill is being widely researched and applied in different tasks such as in games, like Go (Arulkumaran et al., 2019) and StarCraft (Silver et al., 2018), and even in real-life problems, like self-driving cars (S. Wang et al., 2018).

Deep Deterministic Policy Gradient (DDPG) is one of the most popular DRL algorithms that specializes in the environment with continuous action spaces (Lillicrap et al., 2015). It is a model-free off-policy actor-critic algorithm that combines the idea of the Deep Q-network (DQN) (Mnih et al., 2015) algorithm with the Deterministic Policy Gradient (DPG) algorithm (Silver et al., 2014). DQN only works in a discrete space, but DDPG can perform in a high-dimensional, continuous space by learning a deterministic policy.

One of the characteristics of DDPG is that it uses two neural networks, an Actor-network and a Critic-network (Lillicrap et al., 2015). The Actor-network decides which action to take in a given state. The Critic-network then evaluates the Actor's choice in a given state and informs the Actor. This characteristic makes DDPG powerful because it can directly learn the policy with the Actor-network while applying Q-learning with the Critic-network in the continuous space.

The other characteristic of DDPG is that it uses Experience Replay (ER) to stabilize Q-learning. ER is a memory replay method that uses the agent's past experience to train its network. While training, the agent stores its experience in its memory buffer, which is a storage of past interactions. In a typical ER method, the agent samples a random batch from the memory buffer and calculates the loss. Then it uses the loss value to update the Q-network (Lin, 1992). In DDPG, however, it is proven that it shows a better performance with Prioritized Experience Replay (PER) (Cicek et al., 2021). PER is also a memory replay method similar to ER method, but instead of taking random

samples, it samples the experience that would give the greatest learning success to the agent (Schaul et al., 2015). By using PER for a DDPG agent, the agent can learn more efficiently and effectively from its experience.

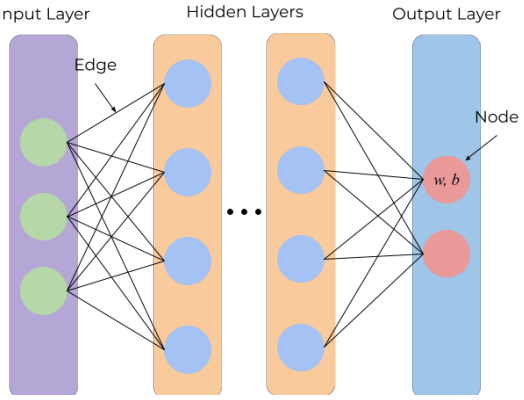


Figure 1.1: A representation of Artificial Neural Network

A problem of DDPG algorithm is that it is expensive to train. DDPG, or Deep Learning (DL) in general, uses deep neural networks (Figure 1.1). A deep neural network consists of an input layer, at least one hidden layer, and an output layer. Each layer has at least one node representing the neurons with threshold values (weight and bias). Then the layers are connected with the edges, which connect the nodes in each layer with the other layers. The computer has to calculate every node's value, the signal strength, by starting from the nodes in the input layers and calculating the following nodes that are connected with edges. As the number of nodes and the hidden layers of the network increases, the number of calculations increases, which also increases the amount of time and data needed for training.

Transfer Learning (TL) can address the problems. Transfer learning is a machine learning method that uses the knowledge, such as its network's weights and biases (Pratt, 1992), from a trained agent in a first task as the starting point for training a second agent in a related but different second task (Bozinovski, 2020). TL gives the second agent a faster start and could significantly reduce the number of resources required for training. Usually, TL is favored in computer vision (Gopalakrishnan et al., 2017) and natural language processing (D. Wang & Zheng, 2015), but recently, there have been successful implementations in DRL. The study by (Sabatelli & Geurts, 2021 and Sasso et al., 2021) proved that a deep model-based RL algorithm could gain a significant advantage in training with the use of TL. However, it has not been proven that model-free RL algorithms could expect the same result.

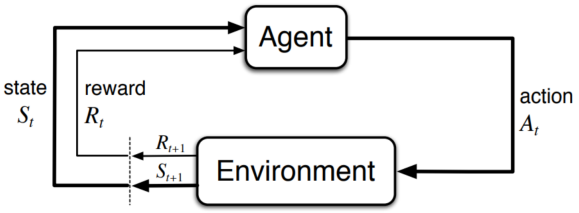


Figure 2.1: The interaction with the agent and the environment in Reinforcement Learning (Sutton & Barto, 2018).

The rest of the paper will investigate whether the transfer of the model parameters, such as weights and biases, or the memory buffer from the trained agent can improve the training of a DDPG agent in a similar but different environment. We will investigate the standard ER and PER buffer for the buffer transfer experiment. For the model parameters transfer experiment, we will be reusing the pre-trained network for the second task. We train the agent in two environments from the OpenAI gym. After training the first agent in one environment, we transfer its knowledge to the second agent and train it in the other environment. We test the agents regularly during the training and store the scores in an array. The performances of the agents were evaluated with four different comparison methods suggested by (Taylor & Stone, 2009).

2 Background

2.1 Reinforcement Learning

RL has two main features, an agent and an environment. An agent is a decision-maker that can interact with its surrounding environment. Its goal is to decide the best action based on its current state.

For each discrete time step t, the agent receives a state, $s \in \mathcal{S}$, where $\mathcal{S}$ is a set of possible states in the environment. Then based on the state, s,

the agent decides an action, $a \in \mathcal{A}(s)$, where $\mathcal{A}$ is the set of possible actions in the given state, s. The environment reacts to the action and returns a numerical reward, $r \in \mathbb{R}$. The agent uses the reward value as feedback and learns a policy, $\pi_t(a|s)$, which is a probability of choosing an action, a, from the state, s. Then the environment shifts to the next state, s', and continues the cycle until it achieves its goal or until it reaches the limited time steps. A sub-sequence interaction with the environment from the starting step to the final step, called the terminal state, is called an episode.

The goal is to learn the policy that maximizes the expected return, which is a cumulative discounted reward. The return, R, is computed with the sum of discounted future rewards,

$$R_t = \Sigma_{k=t}^{T} \gamma^{k-t} r_k \tag{2.1}$$

where T is the time step of the terminal state, and γ is the discount rate with $0 \leq \gamma < 1$. Then the algorithm uses the return value to calculate the action value (Q-value) function, Equation 2.2, which estimates the expected return of the given state and action pair.

$$Q(s, a) = \mathbb{E}[R|s, a] \tag{2.2}$$

The agent learns by trying to maximize the expected return using the Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon}[r(s, a) + \gamma \max_{a'} Q^*(s', a')] \tag{2.3}$$

where Q^* is the optimal Q-value function.

The ϵ is used in a near-greedy method, called the ϵ-greedy method. The method explores the space by uniformly sampling an action from $\mathcal{A}$ with a probability of ϵ. Then the agent acts independently with the highest Q-value estimation with a probability of $1 - \epsilon$.

2.2 DDPG

DDPG is a model-free off-policy DRL algorithm that makes a deterministic action on continuous action space. DDPG consists of two networks, the Actor-network, μ, and the Critic-network, Q. The Critic-network learns the Q-function, and the Actor-network uses the Q-value to learn the policy. This approach allows DDPG to apply Q-learning in continuous action spaces successfully. To stabilize Q-learning, similar to DQN, it uses a replay buffer and target network approach.

The Critic-network uses the Q-function to calculate the expected reward for the action, a, in the state, s. The network calculates the mean-squared Bellman error (MSBE) loss, $L(\phi, \mathcal{D})$, while $\mathcal{D}$ is the collected set of experiences, $\mathcal{D} = (s, a, r, s', d)$. The value d is to check whether the next state is terminal. The MSBE loss tells the agent how much the network satisfies the Bellman equation, y, which is also the target network. The θ and ϕ are the parameters of the Actor-network and the Critic-network. The θ' and ϕ' are the parameters of the networks' target networks.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}}[(Q_\phi(s, a) - y)^2] \tag{2.4}$$

$$y = r + \gamma(1 - d) Q_{\phi'}(s', \mu_{\theta'}(s')) \tag{2.5}$$

One difference between y, Equation 2.5, and the original Bellman equation, Equation 2.3, is that y is not recursive. Instead, y make use of the approximation, $\max_a Q^*(s, a) \approx Q(s, \mu(s))$, where μ is a gradient-based learning policy. The policy μ uses the fact that the function $Q(s, a)$ is presumed to be differentiable for the action argument and allows to make the assumption. Eliminating the recursive property of the original Bellman equation allows a successful implementation of Q-learning in a continuous action space because the recursion takes a redundant time to terminate in a continuous action space.

The goal of the Critic-network is to minimize the MSBE loss value. DDPG algorithm uses target networks to address the goal. To reduce the MSBE loss, the $Q_\phi(s, a)$ in Equation 2.4 has to be more like the target, y. The target networks are the copy of the Actor-network, $\mu'_{\theta'}(s)$, and the Critic-network, $Q'_{\phi'}(a, s)$, that are used to calculate the target values. The target network parameters are softly updated (slowly track the learned network) with respect to the Actor and the Critic parameters.

$$\theta' \leftarrow \tau\theta' + (1 - \tau)\theta, 0 < \tau < 1 \tag{2.6}$$

$$\phi' \leftarrow \tau\phi' + (1 - \tau)\phi, 0 < \tau < 1 \tag{2.7}$$

The target values are constrained to change slowly, which greatly stabilizes the learning progress.

The Actor-network is rather simple compared to the Critic-network. It learns a deterministic policy $\mu_\theta(s)$ that aims to give the action that maximizes $Q_\phi(s, a)$.

$$\max_\theta \underset{s \sim \mathcal{D}}{\mathrm{E}}[Q_\phi(s, \mu_\theta(s)] \tag{2.8}$$

The Actor-network is updated by applying the chain rule to the objective function, J, with respect to the Actor's parameters.

$$\nabla_\phi J \approx \mathop{\mathrm{E}}_{s\sim\rho}[\nabla_a Q_\phi(s,a)|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)] \quad (2.9)$$

The ρ is a discounted state distribution. The equation, Equation 2.9, is also known as the policy gradient, which is the gradient of the policy's performance (Silver et al., 2014).

2.3 PER

PER is a type of memory replay method that replays the transitions with high expected learning progress more frequently to update the Q-network. While the agent interacts with the environment, it stores the transition, $\mathcal{D}$, in its memory buffer. Then, the agent takes a sample batch from the memory buffer based on the expected learning progress and updates the Q-network by replaying the batch on the network. The expected learning progress is measured by the magnitude of temporal-difference (TD) error of each transition.

However, greedy prioritization could lead to a problem of over-fitting due to the lack of diversity in the sampled batch. To overcome the problem, it uses a stochastic sampling method, which is the midpoint of pure greedy prioritization and random sampling. The probability of sampling experience i in the stochastic sampling method is

$$P(i) = \frac{p_i^\alpha}{\Sigma_k p_k^\alpha} \quad (2.10)$$

where p_i is the positive numerical priority of experience i. The α determines how much it uses the prioritization, where $\alpha = 0$ means the uniform sampling case.

2.4 Transfer Learning

TL is a machine learning method that reduces the time and resources needed to train an agent. It does so by transferring the knowledge of the trained agent in one environment to the other agent in a different but related environment. It is mostly preferred in computer vision or natural language processing, but recently, there have been successful implementations in DRL (Sabatelli, 2022 and Zhu et al., 2020).

The approaches of the TL experiment differ depending on the answers to the following questions:

1. What knowledge is transferred?
2. What RL frameworks are compatible with the TL approach?

The knowledge to transfer can be any information that a trained agent learned, i.e., replay memory buffer, reward shaping, policy, or model parameters. One of the most common knowledge used for TL is the model parameters, such as weights and biases of the artificial neural network. Since there are different types of knowledge, the suitable types of knowledge could differ by the RL frameworks.

3 Method

Three main experiments were conducted to test the research question. The three experiments are 1) a memory buffer transfer experiment with both ER and 2) PER, and 3) a model parameters transfer experiment. For each buffer transfer experiment, we had another three separate experiments based on different memory buffer sizes, 10,000 (10^4), 100,000 (10^5), and 1,000,000 (10^6).

3.1 OpenAI Gym Environment

The agents were trained in two different environments, `HopperBulletEnv-v0` and `Walker2DBulletEnv-v0`, from the OpenAI gym. These are the two environments that the first DDPG paper, (Lillicrap et al., 2015), recommended. The experiment transfers the knowledge from one environment to the other and vice versa.

3.1.1 HopperBulletEnv-v0

Hopper is a two-dimensional one-legged feature with four body parts, a torso, a thigh, a leg, and a foot, and they are all connected with three hinges. Its goal is to move forward (right) by controlling its three hinges. The three hinges are also the action space of the environment. The feature can decide a numerical torque in the range [-1, 1] to apply on each hinge. States are 15 tuples containing the position and velocity of the feature and the angle and angular velocity of each hinge.

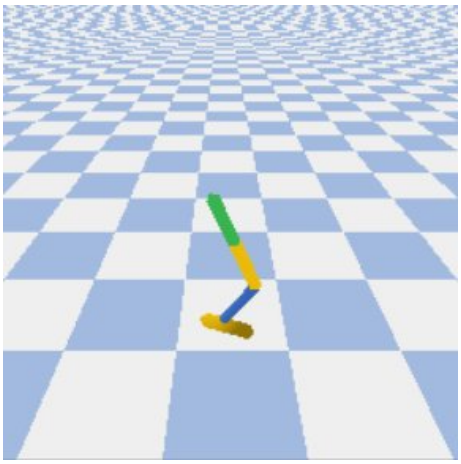


Figure 3.1: OpenAI gym Hopper Environment

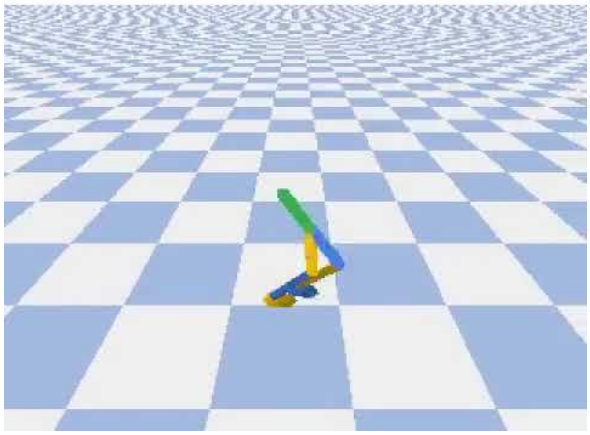


Figure 3.2: OpenAI gym Walker2D Environment

The episode ends if the duration reaches 1000 time steps, if the feature hopped too high, or if it has fallen. In each time step, the feature gets the `alive_reward`, which rewards 1 for being alive. It also gets the `reward_forward` for moving forward, and `reward_control` as a penalty for its actions being too large. Therefore, the cumulative reward is the sum of `alive_reward`, `reward_forward`, and `reward_control`.

3.1.2 Walker2DBulletEnv-v0

Walker2D is similar to Hopper, but instead of having a single leg, it has two legs. An additional leg increases the number of hinges to six, and their mechanisms are the same as the leg of Hopper. Its goal is also to walk forward as far as it can from its initial position. Since there are six hinges, the action space for Walker2D is six. The additional hinges also increase the state space to 22 by combining the original states from Hopper with additional information about the hinges of the other leg. The episode termination condition and the cumulative reward calculation are the same as Hopper.

3.2 Hyper-parameters

Here are the hyperparameters of the DDPG agent (Pearce, 2020):

- Replay buffer size = 10^6, 10^5, 10^4
- Mini-batch size = 64
- Gamma (γ) = 0.99
- Tau (τ) = 0.001
- Actor's learning rate = 0.0001
- Critic's learning rate = 0.001
- Weight decay = 0.01
- Sigma (σ) = 0.05

3.3 Experiment Setup

3.3.1 Training Procedure

First, we created a DDPG agent with random weights. The agent was trained in one of the first environments, $\mathcal{M}_s$, for 100,000 time steps. For every 2000 time steps in training, the agent gets tested, and its score were saved in an array. To test the agent, we ran the agent in the environment for ten episodes and collected their total rewards. We calculated the test score by finding the mean average of the total rewards.

Then, we prepared for the TL by getting a filled-up buffer from the *Section 3.3.2* for a memory buffer transfer experiment or a modified network from *Section 3.3.3* for a model parameter transfer experiment.

Lastly, the TL agent was trained. In the buffer transfer experiments, a new DDPG agent with random weights was created and it was given with the

filled-up buffer from *Section 3.3.2*. In the model parameters transfer experiment, the modified network from *Section 3.3.3* for the second agent was used and it was given with a new empty replay buffer. Then we trained the new agents in the second environment, $\mathcal{M}_t$, for 100,000 time steps and saved the test scores for every 2000 time steps.

For $\mathcal{M}_s$ and $\mathcal{M}_t$,
$\mathcal{M}_s, \mathcal{M}_t \in \{\text{Hopper, Walker2D}\}$, but $\mathcal{M}_s \neq \mathcal{M}_t$.

3.3.2 Preparation for Buffer Transfer

To prepare for buffer transfer for ER and PER, we first created a new empty memory buffer. Then we collect the transaction of the trained agent with $\mathcal{M}_s$ until the new replay buffer is full.

However, because the environments, Walker2D and Hopper, have different state and action spaces, we had to modify the experience data. An experience/transition, $\mathcal{D}$, consists of five tuples, current state, action, reward, next state, and done boolean. The current state, next state, and the action are an array with the size of the action and state space of the environment. Therefore, we need to modify the experience to make it usable in the second environment.

There were two cases for the modification. The first case was when $\mathcal{M}_s$ is Hopper, and $\mathcal{M}_t$ is Walker2D, and the second case was the opposite case, where $\mathcal{M}_s$ is Walker2D, and $\mathcal{M}_t$ is Hopper. For the first case, because Hopper has 15 states and Walker2D has 22 states, we added seven zeros at the end of the arrays of the states to increase their size to 22. For the action array, we added three zeros at the end of the array to increase its size to six. For the second case, we used the first 15 items of the state arrays and the first three items of the action array to fit into the Hopper environment.

3.3.3 Preparation for Model Parameters Transfer

In the model parameters transfer, we reused the trained agent network for the second environment, $\mathcal{M}_t$. Since the DDPG algorithm has two networks, Actor-network and Critic-network, we transferred both networks and their target networks.

However, similar to buffer transfer, the networks also needed modifications to fit into $\mathcal{M}_t$, and the modification method changed for the same two cases from the buffer transfer preparation. For every layer in the network, which its size varies with the state space, we added seven more nodes with weight zero if $\mathcal{M}_s$ is Hopper and removed the last seven nodes if $\mathcal{M}_s$ is Walker2D. Similarly, for every layer that size varies with the action space, we added three more nodes with weight zero if $\mathcal{M}_s$ is Hopper and removed the last sever nodes if $\mathcal{M}_s$ is Walker2D.

4 Results

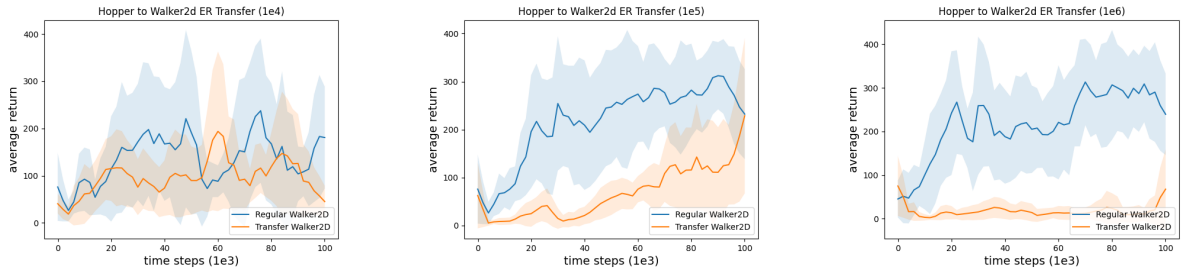
4.1 Plots and Analysis

The agent gets tested for every 2000 time steps while training, and its score, or the averaged total reward, is plotted. Since we have six experiments (two cases × three buffer sizes) for both ER transfer and PER transfer experiments and two experiments for model parameters transfer experiment, we have 14 plots. The plots were analyzed by four comparison methods, initial jumpstart, asymptotic performance, best performance, and the ratio of areas defined by Equation 4.1 (Taylor & Stone, 2009). The initial jumpstart comparison compares the initial performances of the agents, while the asymptotic performance comparison compares the final performances of the agents. The best performance comparison compares the best scores of the agents, and the ratio of areas calculates the ratios between the areas under the agents' learning curves.

$$R = \frac{area\ with\ transfer - area\ without\ transfer}{area\ without\ transfer} \tag{4.1}$$

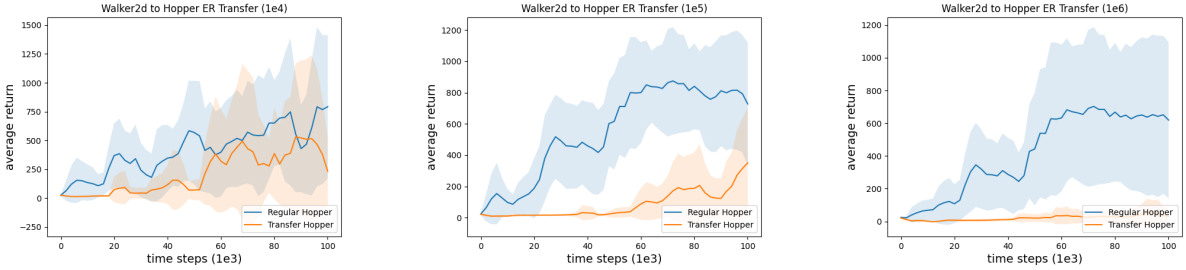
Figures 4.1 to 4.5 shows the plots of the learning curves. The x-axis is the time steps with 1000 units, and the y-axis represents the mean average return. The blue curves are plots of the default agent, and the orange curves are the TL agent. The curves represent the mean average of the five training data sets, and the shaded areas show the standard deviation.

In both ER and PER buffer transfer results, there are three different plots per TL for three different buffer sizes (10^4, 10^5, and 10^6). Tables 4.1 to 4.3 shows the scores of the agents' best performances, and Tables 4.4 to 4.6 shows the ratio of areas of each plot.



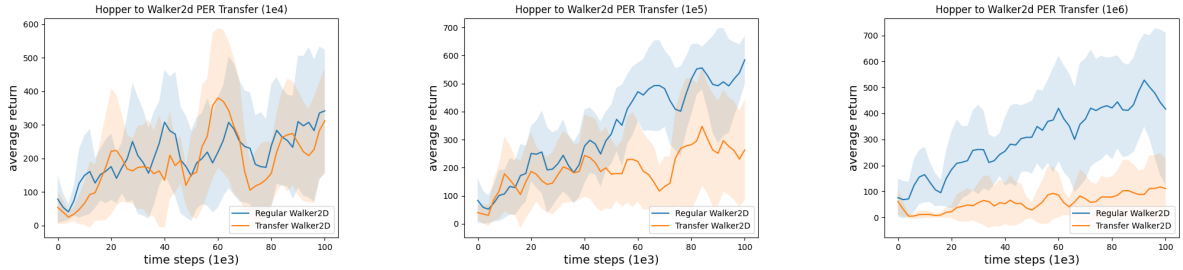
(a) **buffer size:** 10^4 (b) **buffer size:** 10^5 (c) **buffer size:** 10^6

Figure 4.1: Plots of ER Buffer Transfer from Hopper to Walker2D



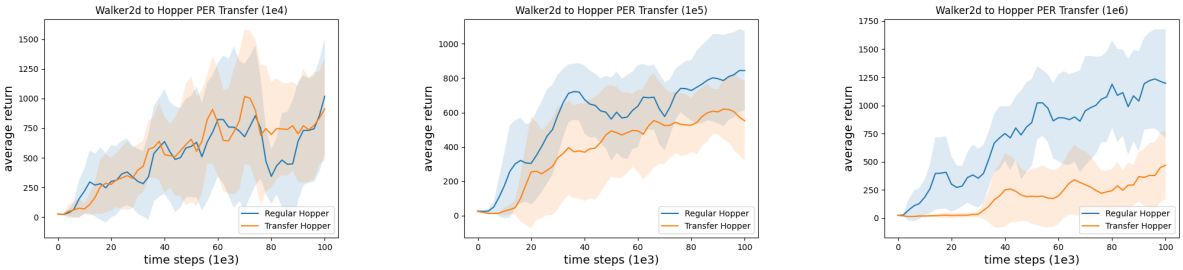
(a) **buffer size:** 10^4 (b) **buffer size:** 10^5 (c) **buffer size:** 10^6

Figure 4.2: Plots of ER Buffer Transfer from Walker2D to Hopper



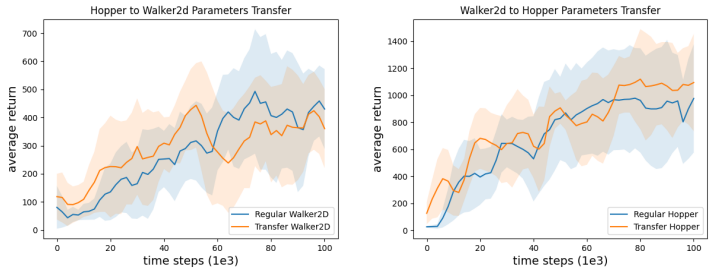
(a) **buffer size:** 10^4 (b) **buffer size:** 10^5 (c) **buffer size:** 10^6

Figure 4.3: Plots of PER Buffer Transfer from Hopper to Walker2D



(a) **buffer size:** 10^4 (b) **buffer size:** 10^5 (c) **buffer size:** 10^6

Figure 4.4: Plots of PER Buffer Transfer from Walker2D to Hopper



(a) Hopper to Walker2D **(b) Walker2D to Hopper**

Figure 4.5: Plots of Model Parameters Transfer

4.2 ER Transfer

The ER transfer experiments show that the TL agent performed worse than the default agent considering Taylor's comparison methods. If we look into Figures 4.1 and 4.2, in most of the cases of TL agents, there was no initial jumpstart, and the asymptotic performances were worse compared to the default agent. Not only that, Table 4.1 shows that the best performances of the TL agent were lower than the default agent. Lastly, the ratios of areas in Table 4.4 are negative, meaning that the area under the default agent learning curves is greater than the TL agent.

4.3 PER Transfer

The results of PER transfer experiments show a similar characteristic to ER transfer, but better. There was no jumpstart at the start, and the asymptotic performances were worse than the default agent. Table 4.2 shows that the best performance of the TL agent in buffer 10^4 was higher than the default agent, but the other cases were worse. Table 4.5 shows that the ratio of the areas under the curve in Figure 4.4a is positive, but the rest of the plots have negative values.

4.4 Model Parameters Transfer

Unlike buffer transfer experiments, the agent has significantly improved its performance during the training after model parameters transfer. In both plots in Figure 4.5, we can see that the TL agent had an initial jumpstart, and its learning speed is faster than the default agent. In Figure 4.5a, the asymptotic performance of the TL agent is lower than the default agent, but it is the opposite in Figure 4.5b. In both cases, the scores of the best performances of the TL agent were higher than the default agent. The ratio of the area in Table 4.6 is all positive numbers indicating that the learning curve of the TL agent has a larger area than the learning curve of the default agent.

5 Discussion

5.1 Findings

The three experiments, ER buffer transfer, PER buffer transfer, and the model parameters transfer in DDPG, were conducted to test the research question. In buffer transfer experiments, we experimented with three different buffer sizes, namely 10^4, 10^5, and 10^6. The agents were trained in two different OpenAI gym environments, the `HopperBulletEnv-v0` and `Walker2DBulletEnv-v0`.

The results of the ER transfer experiments show that TL of memory buffer harms training DDPG experiment. The TL agent showed a lower performance at the start and end of the training than the default agent. Not only that, its scores of best performances were lower than the default agent. All the ratios of areas are negative, supporting the claim that ER transfer has a negative effect on DDPG training.

In the PER transfer experiment, the TL agents performed better than the TL agent in ER transfer experiment but still worse than the default agent. In most cases, the learning curves of the TL agent are under the default agent, and its performance was lower than the default agent at the start and

	Hopper to Walker2D		Hopper to Walker2D	
Buffer Size	Default Agent	TL Agent	Default Agent	TL Agent
10^4	430.08 ± 69.25	329.02 ± 108.08	1213.23 ± 450.73	886.52 ± 823.02
10^5	456.03 ± 80.15	363.98 ± 76.18	951.18 ± 298.51	528.03 ± 420.78
10^6	466.06 ± 93.85	177.79 ± 111.77	780.06 ± 476.88	76.71 ± 94.87

Table 4.1: Best performance for ER Transfer

	Hopper to Walker2D		Hopper to Walker2D	
Buffer Size	Default Agent	TL Agent	Default Agent	TL Agent
10^4	608.31 ± 25.36	615.84 ± 123.05	1383.61 ± 427.68	1451.69 ± 476.76
10^5	671.06 ± 45.54	562.53 ± 138.00	904.33 ± 228.84	789.88 ± 78.26
10^6	637.54 ± 166.34	241.82 ± 162.38	1432.31 ± 356.73	572.04 ± 335.77

Table 4.2: Best performance for PER Transfer

	Default Agent	TL Agent
Hopper to Walker2D	637.63 ± 97.97	672.42 ± 46.63
Walker2D to Hopper	1245.80 ± 417.24	1316.40 ± 247.74

Table 4.3: Best performance for Model Parameters Transfer

Buffer Size	Hopper to Walker2D	Walker2D to Hopper
10^4	-0.279	-0.489
10^5	-0.696	-0.854
10^6	-0.924	-0.951

Table 4.4: Ratio of Area for ER Transfer

Buffer Size	Hopper to Walker2D	Walker2D to Hopper
10^4	-0.106	0.105
10^5	-0.418	-0.308
10^6	-0.809	-0.747

Table 4.5: Ratio of Area for PER Transfer

Hopper to Walker2D	Walker2D to Hopper
0.043	0.119

Table 4.6: Ratio of Area for Model Parameters Transfer

the end of the training. The scores of the ratio of area are mainly negative, indicating that the buffer transfer in DDPG was unsuccessful. The PER buffer transfer is better than ER buffer transfer but still has a negative effect on training the DDPG agent when it has a large buffer size.

One of the intriguing characteristics of the results of buffer transfer experiments is that the TL agent's performance decreases as the buffer size increases. The only case that the buffer transfer experiment that was successful (Figure 4.4a) also had the smallest buffer size, 10^4. The possible cause could be overfitting. As the memory buffer size increases, the percentage of changed memory in the whole memory buffer for each time step reduces. Therefore, it increases the possibility of choosing the experience from the other environment for a longer time step, resulting in a bad experience in the new environment. Then the bad experiences fill the memory buffer again and create a negative cycle while training. Therefore, the buffer transfer has a negative effect on training DDPG agents, and the effect increases when there are more memories or knowledge to transfer.

In model parameters transfer experiments, the transfer of model parameters showed a positive effect on training the DDPG agent. Not only that, the TL agent had a higher performance at the start of the experiment, and it had higher scores for the best performances than the default agent. The scores for the ratio of area are all positive, supporting the claim that the model parameters transfer is beneficial in training DDPG agents.

Overall, the results show that the replay buffer transfer affects the training of DDPG negatively, while the model parameters transfer helps with the training. However, the results are not conclusive because 1) the experiment took place in two environments only; 2) there were overlaps of standard deviation areas in Figures 4.1 to 4.5.

The experiments have to investigate each transferring method in various environments and get a generalized conclusion about the DDPG knowledge transferability. The experiment should also be repeated several times to reduce the standard deviation values for each means and eliminate the overlaps of the shaded areas in Figures 4.1 to 4.5 to have a significant conclusion.

5.2 Future research

The results and conclusions of the experiments are hard to generalize to all the cases of TL in DDPG. Therefore, it has to be investigated in various ways.

One possible future work is to investigate how to modify the transferring knowledge not only to fit in the new environment but also to help the training of the new agent. During the experiment, we had to modify the sizes of arrays in the memory buffer and the number of nodes in the neural networks. The memory buffer was modified by adding zeros to the states and action arrays, and the neural network was modified by adding nodes with zero weights. It was ambiguous if the modification affected the training positively. Therefore, future research could investigate the methods and effects of knowledge modifications.

6 Conclusions

The paper investigated the effect of transferring the memory buffer and the model parameters in the DDPG algorithm. We trained the agent in the first task, collected its knowledge (memory buffer and the model parameters), modified the knowledge to fit into the new similar but different task, and trained the second agent in the second task with the transferred knowledge. We experimented with a regular ER buffer and PER buffer to test the transferability of the memory buffer. For each, we also experimented with three different buffer sizes. All the memory buffer transfers (except for one case, Figure 4.4(a)) harmed the training, and its effect increased as the amount of knowledge, or buffer size, increased. On the other hand, the model parameters transfer helped with the training in both cases.

References

Arulkumaran, K., Cully, A., & Togelius, J. (2019). Alphastar: An evolutionary computation perspective. In *Proceedings of the genetic and evolutionary computation conference companion* (pp. 314–315).

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey

of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

Bozinovski, S. (2020). Reminder of the first paper on transfer learning in neural networks, 1976. *Informatica*, *44*(3).

Cicek, D. C., Duran, E., Saglam, B., Mutlu, F. B., & Kozat, S. S. (2021). Off-policy correction for deep deterministic policy gradient algorithms via batch prioritized experience replay. In *2021 ieee 33rd international conference on tools with artificial intelligence (ictai)* (pp. 1255–1262).

Gopalakrishnan, K., Khaitan, S. K., Choudhary, A., & Agrawal, A. (2017). Deep convolutional neural networks with transfer learning for computer vision-based data-driven pavement distress detection. *Construction and building materials*, *157*, 322–330.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, *8*(3), 293–321.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... others (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Pearce, J. (2020). *Continuous control with deep reinforcement learning - ddpg with prioritized experience replay.* Retrieved from `https://github.com/Jonathan-Pearce/DDPG_PER.git`

Pratt, L. Y. (1992). Discriminability-based transfer between neural networks. *Advances in neural information processing systems*, *5*.

Sabatelli, M. (2022). Contributions to deep transfer learning: from supervised to reinforcement learning.

Sabatelli, M., & Geurts, P. (2021). On the transferability of deep-q networks. *arXiv preprint arXiv:2110.02639*.

Sasso, R., Sabatelli, M., & Wiering, M. A. (2021). Fractional transfer learning for deep model-based reinforcement learning. *arXiv preprint arXiv:2108.06526*.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... others (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning* (pp. 387–395).

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Taylor, M. E., & Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, *10*(7).

Wang, D., & Zheng, T. F. (2015). Transfer learning for speech and language processing. In *2015 asia-pacific signal and information processing association annual summit and conference (apsipa)* (pp. 1225–1237).

Wang, S., Jia, D., & Weng, X. (2018). Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*.

Zhu, Z., Lin, K., & Zhou, J. (2020). Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888*.