university of groningen / faculty of science and engineering

On the transferability of experience replay buffers in deep reinforcement learning

Bachelor's Project Thesis

Annika Moeller Chandiramani, s3585832, a.s.moller.chandiramani@student.rug.nl
Supervisor: Dr. Matthia Sabatelli

Abstract: Deep reinforcement learning (DRL) is a subfield of machine learning that combines deep learning and reinforcement learning. Oftentimes, training a DRL agent is highly computationally expensive, or there is not enough data to do so, which is where transfer learning may help. Transfer learning (TL) is an area of research that focuses on applying knowledge gained from solving one task to help solve another task. This paper investigates the feasibility of transfer learning in DRL by transferring experience replay buffers between different tasks that share the same state space. Experience replay buffers are used in DRL as part of a mechanism called experience replay (ER), in which an agent's trajectories (the information produced as it interacts with the environment) are stored in the aforementioned buffer, and a batch of these trajectories is then sampled and used for training. Three different types of RL tasks of varying levels of complexity are considered for ER buffer transfer in this paper. An agent is trained on two different versions of a task, namely task-A and task-B. The ER buffer from task-A is transferred to another agent that will be trained on task-B and vice versa. This process is then repeated using prioritized experience replay, in which trajectories that have more learning value are sampled more often. Results from all three chosen RL tasks show that this method of transfer learning is indeed successful, with various implications for this field of reinforcement learning. Prioritized experience replay does not seem to have an effect on the degree of transfer success.

1 Introduction

Deep reinforcement learning (DRL) is a sub-field of machine learning in which reinforcement learning and deep learning are combined in order to perform a wide range of tasks such as playing games Shao, Tang, Zhu, Li, and Zhao (2019), controlling a robot (Gu, Holly, Lillicrap, and Levine, 2017), advertising (Zhao, Xia, Tang, and Yin, 2019), etc. DRL stems from reinforcement learning, a method of training intelligent agents to solve a task based on a reward system that corresponds to the actions that the agent takes in a given state, similar to how humans learn to solve tasks in the real world.
There are several algorithms in the field of reinforcement learning, and this paper will focus on one such algorithm called Q-learning (Watkins, 1989). Q-learning is an off-policy, model-free algorithm that learns a function to estimate the value of a state-action pair $Q(s, a)$. This value represents the maximum discounted reward given action a is taken in state s. In standard Q-learning, these Q-values are stored in a lookup table. Despite the success of Q-learning in solving problems with small state and action spaces, this algorithm becomes less efficient or even impossible when the state space becomes too large or infinite. As a solution to this problem we can use an artificial neural network (ANN) to learn a parameterized value function $Q(s, a; \theta)$, which is called a deep q-network (DQN) (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller, 2013).
Although DRL is a significant advancement in the field of reinforcement learning, it comes with several challenges, one of which is a lack of sample efficiency. Sample efficiency refers to an algorithm making the most of a given sample, or in other words, the amount of samples (experiences) needed to perform a task up to a sufficient standard. The inefficiency lies in the fact that it takes a long time

to train an agent using DRL.
One mechanism developed to tackle this problem is experience replay (Lin, 1992). Throughout the process of training a DQN-style agent, its trajectories are collected and stored in a fixed-size buffer. This buffer is usually implemented as a queue, wherein the most recent trajectory is added and the oldest experience is removed at each training step. A batch of trajectories is sampled from the buffer at fixed intervals during training and used to update the neural network. This process improves sample efficiency because it allows for the trajectories to be reused and learnt from multiple times instead of being thrown away immediately after collection. Furthermore, a variant of ER known as prioritized experience replay (PER) will be considered in this paper. In PER, the most valuable trajectories (i.e those that an agent has the most to learn from) are replayed more often (Schaul, Quan, Antonoglou, and Silver, 2015). The way in which this prioritization works will be explained later on in this paper.
Another possible, more recent solution to the problem of sample inefficiency is transfer learning - exploiting knowledge gained in solving one task by applying it to help in the learning of another task. Transfer has proven to be successful in several areas of machine learning, such as natural language processing (Howard and Ruder, 2018) (Devlin, Chang, Lee, and Toutanova, 2018) and computer vision (Cui, Song, Sun, Howard, and Belongie, 2018) (Ge and Yu, 2017) (Gonthier, Gousseau, and Ladjal, 2020). In reinforcement learning, transfer learning has also shown success in model-based environments (Sasso, Sabatelli, and Wiering, 2021). However, in model-free environments where transfer learning is implemented via transferring model parameters, positive transfer appears to be more challenging to achieve (Sabatelli and Geurts, 2021). The distinction between model-based and model-free environments will be explained in more depth in further sections. For now it is important for the reader to know that this paper considers model-free environments.
Another approach to transfer learning in RL is to use the contents of the ER buffer of a DQN. In this paper, we will investigate whether transferring an ER buffer from one model-free agent to another agent trained on a similar task can improve its training process. Experiments are run in three different pairs of OpenAI Gym (Brockman, Cheung, Pettersson, Schneider, Schulman, Tang, and Zaremba, 2016) environments, which vary in task type and complexity. Each pair of tasks consists of a base task and a modified version of the task. In each experiment, agents are trained from scratch on both tasks, after which the most recent version of the ER buffer is stored. Another agent is then trained on the base task, having received the full ER buffer from the modified task, and vice versa. This full process is then repeated using PER instead of regular ER in order to assess whether this type of buffer transfer is more successful when more "valuable" trajectories are used for training. The training progress, measured as the reward per training episode, is then recorded for each agent and plotted for analysis. Finally, the area-ratio scores are calculated for each experiment.

2 Background

2.1 Reinforcement learning

Before diving into the specifics of our investigation, we will provide a general overview of the field of reinforcement learning and the related concepts necessary to understand this paper. At a fundamental level, reinforcement learning is learning through trial-and-error interaction with the environment. Similarly to the way a human might learn, a RL agent learns to adjust its behaviour in response to the outcomes of its own actions. An RL problem involves the autonomous agent, which observes a state $s_t \in \mathcal{S}$ at timestep t and performs an action $a_t \in \mathcal{A}$, after which the agent and environment transition to a new state s_{t+1}. Performing an action a_t also always results in a scalar reward value $r_t \in \mathbb{R}$ given as feedback to the agent on its chosen action. This process is repeated until the agent reaches its goal, or until a pre-determined number of time steps have elapsed. One such cycle is called an episode, which is typically repeated several times in the process of training a RL agent.
The aim of putting an RL agent through several training episodes is to use the information gathered as it interacts with its environment to discover a policy $\pi(s) \rightarrow a$ that tells it how to act in the best way by mapping states onto actions. More specifically, this policy should maximise the

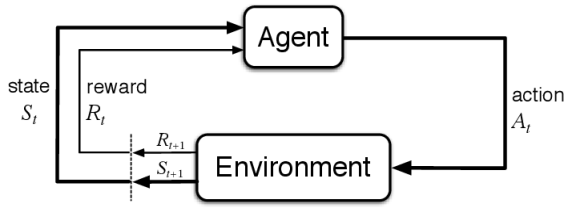

Figure 2.1: Reinforcement learning loop (Sutton and Barto, 1998)

expected sum of of the future, discounted rewards. At each time step t in an episode, the reward data generated by the chosen action is fed back into the agent and used to update its policy. This loop is illustrated in figure 2.1.

A RL problem can be framed as a Markov Decision Process (MDP), which is formalized as follows and consists of:

- $\mathcal{S}$, a set of states; the state space.
- $\mathcal{A}$ - a set of actions; the action space.
- $P(s_{t+1}|s_t, a_t)$, a transition model mapping a state-action pair at time t onto a distributions of states at time $t+1$.
- $R(s_t, a_t, s_{t+1}) \in \mathbb{R}$, a reward function.
- A discount factor $0 < \gamma < 1$, where lower values place more emphasis on immediate rewards and higher values place more emphasis on future rewards.

An important property of RL in the context of MDPs is the Markov Property, wherein the current state signal completely summarizes the history of the interaction with the environment, with no memory needed. For example, in a game of chess, a player only needs information about the current state of the board in order to be able to make their next move. The history of the actions is then effectively irrelevant. To formally define this, first consider how a non-MDP environment responds responds at time $t+1$ to an action taken at time t:

$$Pr\{r_{t+1} = r, s_{t+1} = s'|s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t\} \quad (2.1)$$

In equation 2.1, the environment's response depends on its entire history. However, if a state signal has the Markov property, the environment's response at $t+1$ depends only on the action taken at time t:

$$p(s', r|s, a) = Pr\{r_{t+1} = r, s_{t+1} = s'|s_t, a_t\} \quad (2.2)$$

The Markov property allows us to predict the next state and expected reward based purely on the current state and action, which will be useful in our discussion of Q-learning in the following sections.

2.2 Value functions

Up until now, we have reviewed some key concepts in RL and discussed how they are formulated in terms of MDPs. Now we discuss one of the methods of actually solving RL problems - value functions. Note that other methods exist, but that this is the only one we will consider in this paper. A value function $V(s)$ returns an estimate of the value (i.e expected return) of being in a given state s, when following a policy π, and can be formalized as follows:

$$V^\pi(s) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s, \pi] \quad (2.3)$$

The optimal policy would then return the maximum expected return when in state s:

$$V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in S \quad (2.4)$$

If $V^*(s)$ is known, the optimal policy would be to choose an action a from all available actions in a state s that maximises the expected return $E_{s_{t+1} \sim P(s_{t+1}|s_t, a_t)}[V^*(s_{t+1})]$ according to the transition model in the MDP. However, in RL problems the transition model $P(s_{t+1}|s_t, a_t)$ is not known so a state-action-value $Q^\pi(s, a)$ is used instead, which is calculated by 2.5. Since the action a is provided to the equation, the transition model is no longer needed. This is known as model-free learning.

$$Q^\pi(s, a) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s, a, \pi] \quad (2.5)$$

2.3 Q-Learning

A RL agent's policy Q^π is learned through bootstrapping, which means that the estimation of the

value for the next state is used to improve the policy's estimation of the current state's value. This is possible due to the Markov property; the next state depends only on the current state and action. From here we can define the Bellman equation (Bellman, 1957):

$$Q^{\pi}(s_t, a_t) = E_{s_{t+1}}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))] \tag{2.6}$$

This equation forms the basis of Q-learning (Watkins, 1989), which is a temporal-difference (TD) learning algorithm and can be generally defined as:

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha\delta \tag{2.7}$$

Here α is the learning rate and δ is the TD error given by $\delta = Y - Q^{\pi}(s_t, a_t)$, where Y is the target value that $Q^{\pi}(s_t, a_t)$ approaches. In Q-learning, $Y = r_{t+1} + \gamma \max_a Q^{\pi}(s_{t+1}, a)$. Q-learning is an off-policy algorithm, meaning that it is updated by actions not necessarily chosen by the policy Q^{π}. The on-policy version of this is the state-action-reward-state-action (SARSA) algorithm, in which the target value $Y = r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1})$, where value estimates are always updated based on the action chosen by the policy. In this paper, only Q-learning is used. Putting this all together, the update equation for Q-learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.8}$$

2.4 Exploration

In an off-policy algorithm such as Q-learning, it is common to use an exploration strategy to ensure that the agent learns from its actions while also exploring new actions and states. A well-known strategy is the decaying ϵ-greedy strategy, where ϵ is the probability of choosing a random action (bound between 0 and 1) and "greedy" means choosing the action with the highest expected return according to the current version of the agent's policy. "Decay" means that the value of ϵ is decreased over time such that the agent explores its environment more towards the beginning of training (where ϵ is close to 1), and exploits its learned knowledge more towards the end (where ϵ is close to 0).

2.5 Deep Q-Networks

For low-dimensional problems, Q-values for each state-action pair can be stored in a lookup table. However, this becomes unfeasible when MDPs become larger and more complex due to computational resource requirements. Here DRL becomes useful because we can use a neural network θ as a function approximator to estimate the value function $Q(s, a; \theta) \approx Q^*(s, a)$. Additionally, a target network θ' is used to calculate the expected reward values for the next state $Q(s_{t+1}, a)$ to reduce the correlation between target and the parameters θ that are being updated as the agent learns the policy. The TD error for the DQN is given by:

$$\delta_\theta = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta') - Q(s_t, a_t; \theta) \tag{2.9}$$

2.5.1 Loss functions

In this paper two types of loss function are used - mean squared error (MSE) (2.10) and Huber loss (Huber, 1964) (2.11). MSE is chosen for its emphasis on penalizing outliers (i.e very large TD-error δ_θ). It is used in two of the three chosen tasks, where it is useful to penalize incorrect actions heavily.

$$L_\theta = [\delta_\theta]^2 \tag{2.10}$$

Huber loss (2.11) is less sensitive to outliers than MSE, because the equation changes from quadratic to linear as the loss increases. It uses a pre-defined parameter ρ to control the range in which each function is applied. This loss function is used in the one remaining task, where it is not as necessary to heavily penalize errors.

$$L_{\theta_\rho} = \begin{cases} \frac{1}{2}(\delta_\theta)^2 & \text{if } |(\delta_\theta)| < \rho \\ \rho(\delta_\theta) - \frac{1}{2}\rho) & \text{otherwise} \end{cases} \tag{2.11}$$

2.6 Experience replay

Now that we have explained key concepts in more detail, we can revisit the topic of experience replay. As mentioned previously, experience replay involves storing trajectories in a buffer and sampling them to train an agent's network weights. These trajectories are produced at each time step t of a training cycle. A training *episode* consists of multiple time steps and therefore multiple trajectories. A trajectory consists of a state s_t, the chosen action a_t,

the resulting reward r_{t+1}, and the next state s_{t+1}. Each time a trajectory is produced, it is stored in the experience replay buffer D, which is a queue of a certain size N. Each time the network weights θ are updated, a batch b containing a number of trajectories is sampled at random from the buffer. For each experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in b, the target value $Y_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a_t; \theta_t)$ is calculated and gradient descent is performed on the TD error $Y_t - Q(s_t, a_t; \theta_t)$.

2.6.1 Prioritized experience replay

Prioritized experience replay (PER) is a variant of ER in which trajectories with high expected learning progress are replayed more frequently than others. The priority value of a trajectory is measured by the magnitude of the TD error:

$$\delta_i = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) - Q(s_t, a_t; \theta_t) \quad (2.12)$$

Given the TD error δ_i, we calculate a priority for each trajectory. The method used for ranking in this paper is proportional ranking, in which the priority p_i is calculated as:

$$p_i = |\delta_i| + \epsilon \quad (2.13)$$

where ϵ is a small offset value that serves to ensure that all trajectories have a non-zero probability of being drawn. The priorities are then converted to the probability of being chosen within the current batch. Because these priority values are based on an approximation, they may suffer from approximation errors. To dampen this effect, a priority scaling constant z is used. z is a value between 0 and 1, where $z = 1$ results in full priority sampling and $z = 0$ becomes random sampling just like normal ER.

$$P(i) = \frac{p_i^z}{\sum_k p_k^z} \quad (2.14)$$

The final adjustment made in PER is to the network's update step. Once we introduce sampling priorities to the replay buffer, this changes the distribution of trajectories that the RL agent's network is trained on, introducing bias and possible overfitting. This does not happen in regular ER as trajectories are sampled randomly, which preserves the original distribution of trajectories experienced in the environment. To account for this we introduce a weighting factor:

$$w_i = \left(\frac{1}{N} \times \frac{1}{P(i)}\right)^\beta \quad (2.15)$$

where N is the current size of the replay buffer, and β controls how much prioritization to apply. This weighting factor w_i is then used in the Q-network's update step by using $w_i \delta_i$ instead of only δ_i.

3 Methods

Three different categories of tasks were considered for the buffer transfer, including a simple toy-text task with a discrete vector state space (Frozen Lake), a control task with a continuous vector state space (Lunar Lander), and a pair of high-dimensional Atari tasks with pixel input (Breakout and Pong). All environments were sourced from OpenAI Gym (Brockman et al., 2016). These environments were chosen in order to investigate the transfer learning capabilities of the agents at different levels of complexity. All experiments involve four agents and two tasks which we will call `task-A` and `task-B`, and always follow the same structure:

1. Train agent 1 on `task-A` and save the last buffer, `buffer-A`.
2. Train agent 2 on `task-B` and save the last buffer, `buffer-B`.
3. Transfer `buffer-B` to agent 3 and train it on `task-A`.
4. Transfer `buffer-A` to agent 4 and train it on `task-B`.
5. Compare training progress of all four agents.

`task-A` and `task-B` have the same state and action space, but are modified in terms of the environment. This whole process is repeated with an agent that uses PER, and is otherwise exactly the same. From now on in this paper, each transfer experiment where `buffer-B` is transferred to `task-A` shall be referred to as `task-A[buffer-B]`. A more detailed description of each environment, agent, and how the transfer is performed is presented in the following sections.

3.1 Frozen Lake

3.1.1 Environment description

Frozen lake is a game in which the aim is to walk across a grid (3.1) from a starting point (top left square) to reach the goal (bottom right square), without falling into any holes (squares with blue circles). The agent can move one square at a time and has a choice of four actions: left, right, up, or down. The state space is either a 4×4 (3.1) or 8×8 (3.2) grid with holes on certain squares, the locations of which are randomly generated whenever the environment is reset. The agent's state is represented by the number of the square that it is currently on, which is encoded as a one-hot vector. The game resets when the agent falls in a hole or reaches the goal, and it is considered solved when the agent reaches the goal. The agent receives a reward of +1 for reaching the goal, and for all other actions the reward is 0.

Figure 3.1: 4x4 Frozen Lake

Figure 3.2: 8x8 Frozen Lake

3.1.2 Transfer

The environmental parameter that was modified for this experiment was the `slip` parameter. Setting this value to `True` makes the environment stochastic. There is a 2/3 chance of the agent 'slipping' i.e moving to a square other than the intended one. This will be referred to as the `slip` environment from now on. Setting the parameter to `False` means that the environment is entirely deterministic. The agent will always move to the square it decides to move to in this `no-slip` environment. Hence, the two environments are `slip-4×4` and `no-slip-4×4`. The buffer from `slip-4×4` was transferred to an agent training on `no-slip-4×4`, and vice versa.
Since Frozen Lake can also be played on an 8×8 grid, the same experiment was performed on this version. Using a larger grid means that there are 64 possible states instead of 16, which this makes the task more difficult to solve. Hence, a second experiment was performed on this 8×8 frozen lake grid to investigate if the results from the simpler task scale up to a more difficult task. The two resulting environments are `slip-8×8`, and `no-slip-8×8`. The buffer from `slip-8×8` was transferred to an agent training on `no-slip-8×8`, and vice versa.
All four of these experiments were then repeated using PER. There was no transfer between agents

trained on 4×4 and 8×8 grids as the state space is different.

3.1.3 Agent

This agent is a DQN (Table 3.1). The loss is calculated at each training step from a minibatch of 50 trajectories sampled from the ER buffer, which contains a maximum of 1000 trajectories. The ϵ-greedy strategy is used for exploration, where ϵ is decayed from 1.0 to 0.1 exponentially. The decay factor is 0.95 for the 4×4 grid and 0.99 for the 8×8 grid. This was done to encourage more exploration in the 8×8 grid since it is larger and therefore more difficult to find the path to the goal. The loss function is mean-squared error (2.10,) and an Adam optimizer (Kingma and Ba, 2014) with a learning rate of 0.001 is used. Each agent is trained for 250 episodes, and all buffers used for transfer are obtained after these 250 episodes of gameplay have been completed.

Table 3.1: ANN architecture for Frozen Lake DQN agent

Layer	Nodes	Activation Function
Input layer	16 or 64	Relu
Hidden layer	24	Relu
Output layer	4	Linear

3.2 Lunar Lander

3.2.1 Environment description

Lunar Lander is a control task that involves landing a rocket on a moon surface in between two flags on the screen. The rocket is controlled by two thrusters situated on the left and right respectively. The agent has 4 possible actions available: firing the left thruster, the right thruster, firing both, or doing nothing. The state space (3.3) is a mixture of continuous and discrete and consists of a vector with 8 dimensions. It represents the current status of the rocket with the following variables: x position, y position, v_x velocity in x-direction, v_y velocity in y-direction, θ orientation in space, $\dot{\theta}$ angular velocity (how fast the rocket is spinning around its central point), right leg ground contact (discrete boolean variable), and left leg ground contact (discrete boolean variable). The total reward of an episode is the sum of all reward for all steps within the episode. Moving from the top of the screen to the landing pad gives between 100 and 140 points. Moving away from the landing pad results in losing points (corresponding to the distance from the landing pad). Crashing gives in an additional -100 points, and landing smoothly gives an additional +100 points. Ground contact with each leg gives 10 points, and firing the main engine gives -0.3 points per time step. The game is considered solved with an episode score of 200 or more.

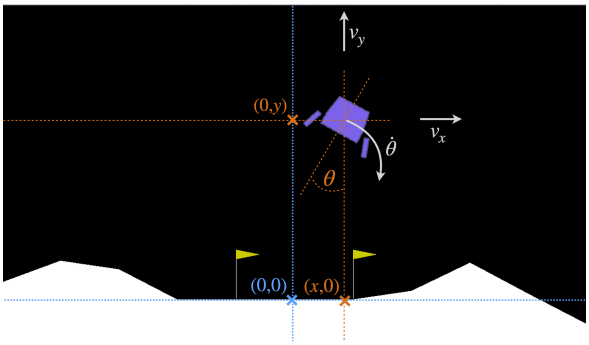

Figure 3.3: Lunar Lander environment (own image)

3.2.2 Transfer

The gravitational pull of this environment was modified for this transfer experiment. The gravity value is bounded between 0 and -12 in this Gym environment, so the values -10 and -5 were used. Hence, the transfer experiments are `gravity-5[gravity-10]` and `gravity-5[gravity-10]`. The same experiment was performed again using PER instead of regular ER.

3.2.3 Agent

The Lunar Lander agent also makes use of a DQN (3.2), and uses the mean-squared error loss function (2.10). The ER buffer size is 100000, and minibatches of 128 trajectories are sampled for training the network. The ϵ-greedy exploration strategy is used again, where ϵ is decayed exponentially from

Table 3.2: ANN architecture for Lunar Lander DQN agent

Layer	Nodes	Activation Function
Input layer	8	ReLu
Hidden layer	64	ReLu
Hidden	64	ReLu
Output layer	4	Linear

1.0 to 0.1 at a rate of 0.995. Experiments are run for 500 episodes on this task.

3.3 Breakout/Pong

3.3.1 Environment description

Breakout and Pong are both classic Atari 2600 games. Breakout involves moving a paddle across the bottom of a screen to deflect a ball, with the goal of hitting, and thereby breaking, all the bricks at the top of the screen (3.4). Missing the ball results in losing a life, of which the player starts each game with 3. There are three possible actions: move the paddle left, right, or keep it still.
Pong also involves moving a paddle to deflect a ball, but in this game the paddle is on the right of the screen and the ball is deflected towards another player (the computer) on the left side of the screen (3.5). The goal is to deflect the ball such that the second player misses it, which results in +1 reward. The agent itself missing the ball results in -1 reward. The game ends when either player reaches a score of 21 (so either +21 or -21 total reward for the agent). The three possible actions here are to move to paddle up, down, or keep it still.

3.3.2 Transfer

This experiment is slightly different from the previous two in the sense that the transfer is not between two different versions of the same environment, but rather two separate tasks entirely. This is possible because the state space for both tasks is an 84×84 grid of pixels. Additionally, both tasks share their action space. In addition to sharing the state and action space, these two tasks were chosen because of the similarity in the behaviour that needs to be learnt by both agents i.e moving a paddle across the screen to deflect a moving ball towards the other side. The experiments are then `Breakout[Pong]` and `Pong[Breakout]`.

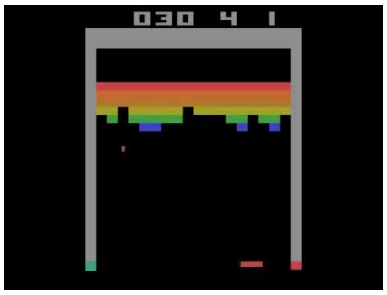

Figure 3.4: Atari Breakout environment

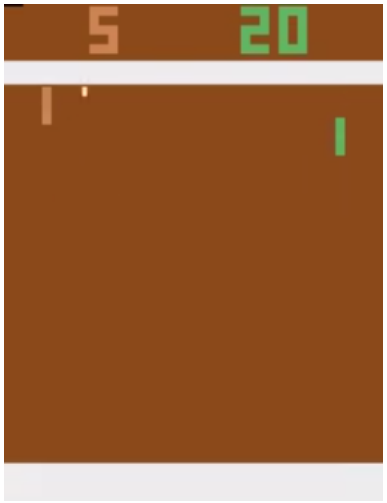

Figure 3.5: Atari Pong environment

3.3.3 Agent

This task uses a DQN modelled after DeepMind's Atari agent (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, Petersen, Beattie, Sadik, Antonoglou, King, Kumaran, Wierstra, Legg, and Hassabis, 2015). The pre-processing steps and CNN architecture used (3.3) remain the same in this experiment, along with the batch size, discount factor, learning rate, and replay start size (actions are chosen randomly for 50000 frames before training commences in order to fill the buffer). However, given time constraints, adjustments have been made in accor-

dance with a modified implementation by Chapman and Lechner (2020). The agents in this paper are trained for 10 million frames, although the original paper trained for 50 million frames. Due to the reduced number of training frames, an Adam optimizer (Kingma and Ba, 2014) was used instead of the original RMSProp; according to Chapman and Lechner (2020), the Adam optimizer improves training time. The buffer size used is 190000 instead of the original 1000000 in order to avoid memory issues. Additionally, the Huber loss function is used with $\rho = 1.0$ (see 2.11) to improve training stability. The agent is trained every 20 steps instead of the original 4 in order to reduce training time. We use a variation on the decaying epsilon-greedy exploration strategy: epsilon is decayed linearly from 1.0 to 0.2 over 200000 steps, from 0.2 to 0.1 over another 200000 steps, and from 0.1 to 0.02 over another 200000 steps. This makes for an epsilon value that decays at a relatively faster rate at the beginning and at a slower rate towards the end. Note that the units for training steps for all other experiments in this paper are in episodes, which comprise of multiple frames. 10 million frames equates to roughly 40000 episodes of `Breakout` (4.7), and 4000 episodes of `Pong` (4.8).

Table 3.3: CNN architecture for Breakout/Pong task

Layer	Filters	Kernel size	Strides	Activation Function	Output shape
Input	-	-	-	-	$84 \times 84 \times 4$
Conv2d	32	8×8	4	Relu	$20 \times 20 \times 32$
Conv2d	64	4×4	2	Relu	$9 \times 9 \times 64$
Conv2d	64	3×3	1	Relu	$7 \times 7 \times 64$
Flatten	-	-	-	-	3136
Dense	-	-	-	Relu	512
Dense (output)	-	-	-	Linear	4

3.4 Experimental setup

3.4.1 Performance evaluation

To compare the performance of the different agents, the training progress per episode of all four agents involved in an experiment is plotted on a graph. For the Frozen Lake task, the progress is measured as cumulative reward because the agent in this task receives a fixed reward for reaching the goal. Hence, the progress cannot be measured by the reward value alone, but rather by the rate at which the task is solved by the agent. Lunar Lander and Breakout/Pong are measured using non-cumulative episode reward because the reward value in these games changes depending on how well the agent is performing. Each graph is analysed according to the three criteria described by (Lazaric, 2012), which include:

- Learning speed: reduction in the amount of the experience needed to learn the solution of the task.
- Jumpstart: improvement in the initial performance of an agent.
- Asymptotic performance: improvement in the final learned performance of an agent
- Transfer ratio: the ratio of the total reward accumulated by the transfer learner and the total reward accumulated by the non-transfer learner, given by 3.1.

$$T = \frac{\text{area with transfer - area without transfer}}{\text{area without transfer}} \tag{3.1}$$

4 Results

4.1 Frozen Lake

Table 4.1: Area ratio scores for Frozen Lake experiments

Task	4×4	8×8	4×4 PER	8×8 PER
`slip [no-slip]`	0.22	-0.41	-0.12	0.44
`no-slip [slip]`	0.23	2.32	0.13	2.13

4.1.1 Findings

Regular ER: All experiments showed positive transfer, except for `slip[no-slip]` in the 8×8 environment 3.1. The area-ratio score T is higher for the `no-slip[slip]` transfer in both environments than for the opposite transfer direction.

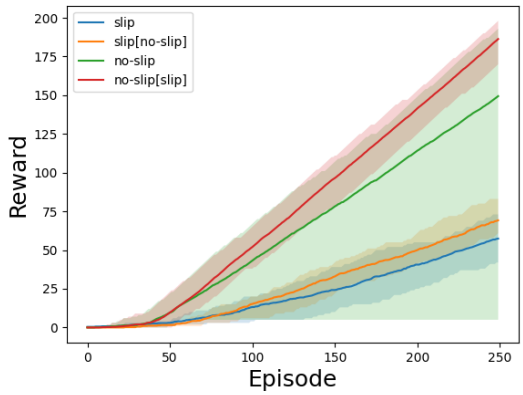

Figure 4.1: Frozen lake 4×4

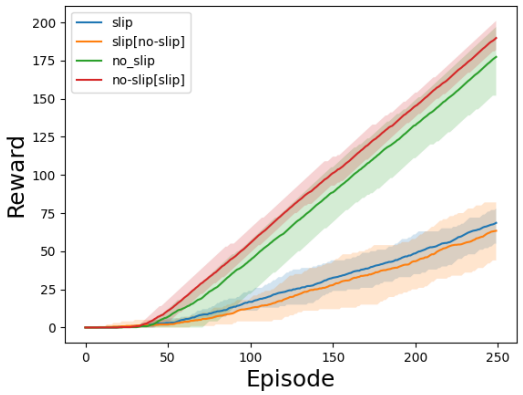

Figure 4.3: Frozen lake 4×4 with PER

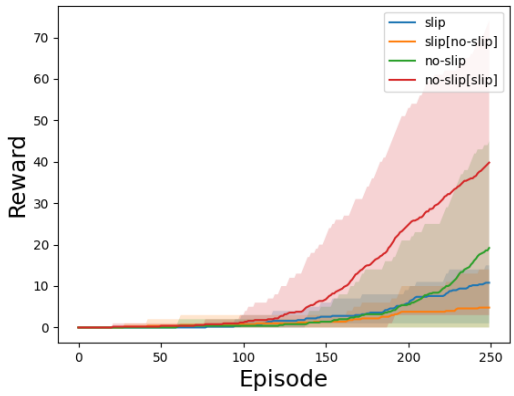

Figure 4.2: Frozen lake 8×8

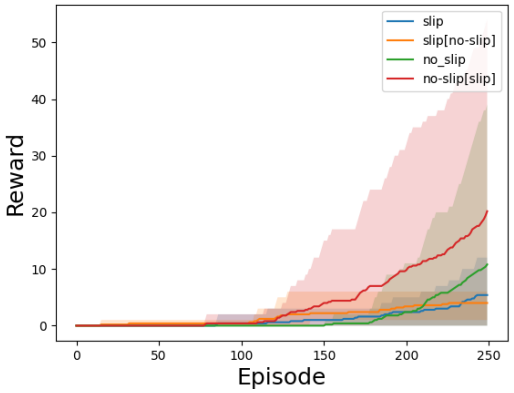

Figure 4.4: Frozen lake 8×8 with PER

Both transfer directions in the 4×4 environment show an increase in learning speed, but no jumpstart or asymptotic improvement (4.1). In the 8×8 environment, the `no-slip[slip]` experiment also shows learning speed improvement, but no jumpstart or asymptotic improvement (4.2).

PER: All experiments showed positive transfer, except for `slip[no-slip]` in the 4×4 environment. Similarly to the regular ER experiments, T was also higher for both `no-slip[slip]` transfer experiments than for the other transfer direction.

Regular ER vs PER: Area-ratio scores for PER experiments are not significantly different to the area-ratio scores for regular ER.

4.2 Lunar Lander

Table 4.2: Area-ratio scores for Lunar Lander experiments

Task	Regular ER	PER
`gravity-10 [gravity-5]`	0.78	-0.18
`gravity-5 [gravity-10]`	0.14	0.51

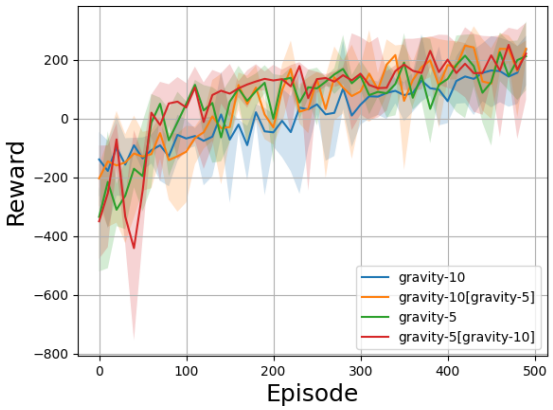

Figure 4.5: Lunar Lander results with regular ER

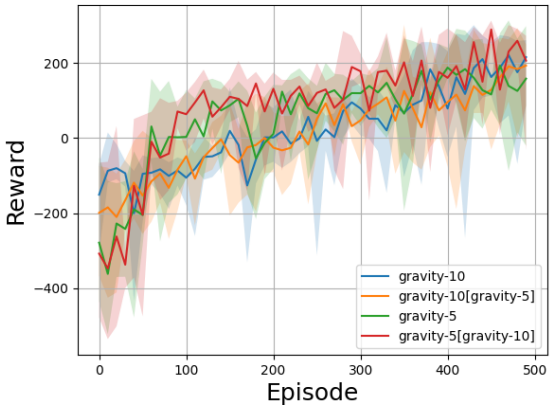

Figure 4.6: Lunar Lander results with PER

4.2.1 Findings

Regular ER: Both transfer directions showed positive transfer, although T was higher for `gravity-10[gravity-5]` than for the other way around. Both transfer experiments show marginal improvement in learning speed, although it looks to be more consistent in `gravity-10[gravity-5]` (4.5), which is reflected in the area-ratio score (4.2). No jumpstart or asymptotic improvement can be seen.

PER: Transfer was negative for `gravity-10[gravity-5]`, while for the other direction it was positive. For the positive transfer, `gravity-5[gravity-10]`, there was some learning speed improvement, and again no jumpstart or asymptotic improvement.

Regular ER vs PER: Similarly to the Frozen Lake experiments, PER showed no significant improvement in transfer when compared to regular ER. T was higher for `gravity-5[gravity-10]` using PER instead of regular ER, but not by a large amount. In fact, transfer was negative for the other experiment, `gravity-10[gravity-5]`, while with regular ER, this same experiment showed the highest transfer value of all.

4.3 Breakout/Pong

4.3.1 Findings

Both transfer directions in this experiment resulted in positive transfer (4.3). However, the area-ratio score for `Pong[Breakout]` was far larger than that of `Breakout[Pong]` (0.59 and 0.02 respectively). Due to computational resource and time limits, prioritized experience replay was not implemented for this task. In `Breakout[Pong]`, no jumpstart or asympototic improvement can be observed, whereas in `Pong[Breakout]`, there is clear improvement asymptotic performance (4.8).

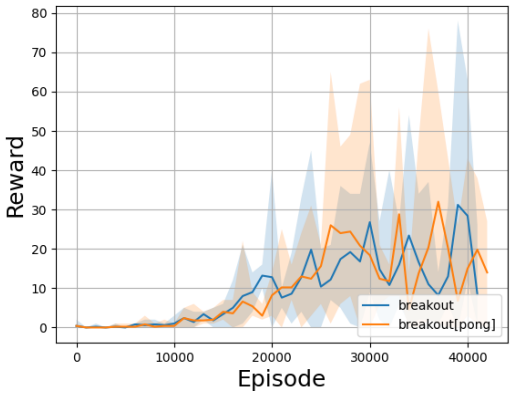

Figure 4.7: Breakout transfer experiment

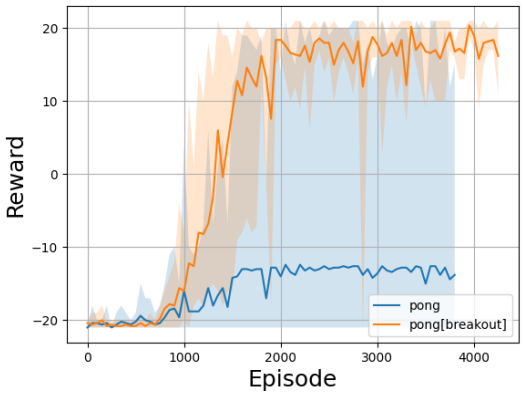

Figure 4.8: Pong transfer experiment

Table 4.3: Area-ratio scores for Breakout/Pong transfer experiments

Task	Regular ER
`Pong[Breakout]`	0.59
`Breakout[Pong]`	0.02

5 Discussion

When considering the Frozen Lake tasks, the investigated buffer transfer technique was generally successful. Furthermore, it is clear that in all experiments, transfer was more successful in the `no-slip[slip buffer]` direction than vice versa. `slip` is a more difficult task to solve than `no-slip` due to its stochasticity, so perhaps the trajectories from this task are more valuable than those from the deterministic `no-slip` task. This is a possible explanation for the consistently positive and higher area-ratio scores when the `slip` buffer is the one being transferred. Interestingly, the highest area-ratio scores were seen in the 8×8 environment for both types of ER. These were 2.32 and 2.13 for 8×8 environments, compared to 0.23 and 0.13 for 4×4 environments, which are roughly 10-fold and 20-fold increases in respective area-ratio scores. The 8×8 environment is more difficult to solve when compared to the 4×4 environment due to its larger state space. This observation indicates that transfer learning may be the more beneficial in challenging environments than in simple environments. As for the comparison between regular ER and PER, it does not seem as if the PER buffer contains more valuable experiences for learning than the regular ER buffer as hypothesised.

For the Lunar Lander task, the degree of transfer was also largely positive. There was no significant difference between the area ratio scores achieved in Lunar Lander experiments and Frozen Lake experiments. The use of buffer transfer in both experiments resulted improvements in learning speed only. Prioritized experience replay showed worse area-ratio scores for the `gravity-10[gravity-5]` task, and better scores for the `gravity-5[gravity-10]` task. Similarly to the Frozen Lake tasks, no strong conclusions can be drawn about any advantage that PER may provide over regular ER in buffer transfer experiments.

Buffer transfer was successful again in the Pong/Breakout transfer task. However, `Breakout[Pong]` showed negligible improvement in any sense, while `Pong[Breakout]` showed clear asymptotic improvement. When inspecting the individual results of each of the five repetitions of the experiment, it was clear that in the base `Pong` task with no buffer transfer, only one of the five repetitions resulted in the agent being able to solve the task. After ten million frames, it had achieved a rolling average of 17.6 points (out of 21 possible points). The other four runs saw the agent being stuck at -21 points i.e not learning at all.

A possible reason for `Pong` being unable to learn so often is that the same DQN architecture and hyperparameters were used for both `Pong` and `Breakout`, so while it may have worked for `Breakout`, it may not have been optimal for `Pong`. For future experiments, one could run this same experiment after tuning the hyperparameters or finding a better architecture such that `Pong` is able to learn more stably. Nevertheless, the results achieved from this experiment were unexpectedly intriguing. Interestingly, when the buffer from the `Breakout` task was transferred to the `Pong` agent, it was able to learn and achieve a score of more than 17 points in all five repetitions of the experiment. Given this result, it seems that buffer transfer can improve the learning stability of an agent, giving it a kick-start and making it more likely to find a solution to a task, and perhaps preventing it from becoming stuck in a sub-optimal behaviour.

6 Conclusions

The experiments detailed in this paper have been largely successful. Buffer transfer generally results in higher area-ratio scores in all three types of OpenAI Gym environments chosen; a simple toy-text task with discrete vector state-space (Frozen Lake), a control task with a mixed discrete/continuous vector state space (Lunar Lander), and high-dimensional Atari tasks with pixel input (Breakout and Pong). Key takeaways from this investigation are:

1. Transferring trajectories from more difficult tasks to a simpler task, rather than vice-versa, results in more successful transfer (as seen when transferring `slip` trajectories to `no-slip` environment).

2. Buffer transfer may be more beneficial in more challenging environments (as seen when comparing area-ratio scores from 8×8 with those from 4×4 Frozen Lake tasks).

3. Buffer transfer can improve the training stability of an agent (as seen in the improvement of the `Pong` agent when `Breakout` trajectories are transferred into it).

4. Prioritized experience replay does not have an effect on the success of buffer transfer.

This investigation has shown that buffer transfer can improve the training progress of agents in environments with the same state and action space, so the next step would be to investigate whether similar outcomes can be observed in environments that differ in their state and/or action space.

References

Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN 9780486428093.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Jacob Chapman and Mathias Lechner. Deep Q-Learning for Atari Breakout, 05 2020. URL `https://keras.io/examples/rl/deep_q_network_breakout/`.

Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge J. Belongie. Large scale fine-grained categorization and domain-specific transfer learning. *CoRR*, abs/1806.06193, 2018. URL `http://arxiv.org/abs/1806.06193`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

Weifeng Ge and Yizhou Yu. Borrowing treasures from the wealthy: Deep transfer learning through selective joint fine-tuning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10–19, 2017. doi: 10.1109/CVPR.2017.9.

Nicolas Gonthier, Yann Gousseau, and Saïd Ladjal. An analysis of the transfer learning of convolutional neural networks for artistic images. *CoRR*, abs/2011.02727, 2020. URL `https://arxiv.org/abs/2011.02727`.

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

Jeremy Howard and Sebastian Ruder. Fine-tuned language models for text classification. *CoRR*, abs/1801.06146, 2018. URL `http://arxiv.org/abs/1801.06146`.

Peter J. Huber. Robust estimation of a location parameter. *Annals of Mathematical Statistics*, 35:492–518, 1964.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

Alessandro Lazaric. Transfer in reinforcement learning: A framework and a survey. In *Reinforcement Learning*, 2012.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*,

abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

Matthia Sabatelli and Pierre Geurts. On the transferability of deep-q networks. *CoRR*, abs/2110.02639, 2021. URL https://arxiv.org/abs/2110.02639.

Remo Sasso, Matthia Sabatelli, and Marco A. Wiering. Fractional transfer learning for deep model-based reinforcement learning. *CoRR*, abs/2108.06526, 2021. URL https://arxiv.org/abs/2108.06526.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *CoRR*, abs/1912.10944, 2019. URL http://arxiv.org/abs/1912.10944.

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

Xiangyu Zhao, Long Xia, Jiliang Tang, and Dawei Yin. "deep reinforcement learning for search, recommendation, and online advertising: A survey" by xiangyu zhao, long xia, jiliang tang, and dawei yin with martin vesely as coordinator. *SIGWEB Newsl.*, (Spring), jul 2019. ISSN 1931-1745. doi: 10.1145/3320496.3320500. URL https://doi.org/10.1145/3320496.3320500.

A Appendix

	Frozen Lake	**Lunar Lander**	**Breakout\ Pong**
DQN			
Learning rate α	0.001	0.001	0.00025
Discount rate γ	0.97	0.99	0.99
Maximum epsilon ϵ	1.0	1.0	1.0
Minimum epsilon ϵ	0.05	0.1	0.02
ϵ- decay rate	0.95 (4×4) 0.99 (8×8)	0.995	See section 3.3.3
ER Buffer			
Buffer size	1000	100000	190000
Batch size	50	128	32
Target model update frequency (time steps)	500	1000	10000
PER			
z	1.0	0.6	NA
β	1-ϵ	0.95	NA
Experiment Runs			
Training start (time steps)	50	256	32
Training frequency (time steps)	1	4	20
Episodes	250	500	10 million frames
Max steps per episode	100	500	10000
Repetitions	5	5	5

Table A.1: All experiment hyperparameters