# COMP3004

# Designing Intelligent Agents

## EXPERIMENTING WITH TRANSFER LEARNING IN DEEP REINFORCEMENT LEARNING

20113536

E-SHEN GAN

hfyeg2@nottingham.ac.uk

10/5/2022

# Table of Contents

# 1.Introduction

For the past few decades, there have been many successful attempts at creating game Artificial Intelligence(AI) that can play games without human intervention at all. Deep Reinforcement Learning(DRL) has played a major role in successful massive projects done by OpenAI and DeepMind in defeating top human players with trained AI agents and an almost superhuman level of non-human players can be created [1] [2]. Inspired by the breakthroughs many have made in the field of Reinforcement Learning(RL), this project seeks to experiment with an existing deep reinforcement learning algorithm alongside a common approach in Deep Learning(DL) known as Transfer Learning(TL) in Atari 2600 games.

# 2.Aim

The aim of this project is to train an agent with DRL to play an Atari 2600 game through TL and compare its performance to another agent that is trained without TL.

# 3.Background & Related Works

In close relation to my project, Mnih et al. from DeepMind were the first to introduce a DRL model that can learn to control the agents by directly learning policies from raw high-dimensional pixel inputs in Atari 2600 games. It was made possible by training the model in a deep CNN with a Q-learning variant to produce a value function that estimates future rewards. Deep Artificial Neural Network(ANN) combined with Q- learning, a value-based RL algorithm, this approach is known as Deep Q-Network(DQN). Mnih et al. demonstrated that deep CNN can solve issues that are often present in RL tasks such as spare, noisy or delayed reward signals, state sequences that are usually correlated and inconsistent data distribution and successfully learn control policies from raw pixel inputs in complex RL environments. They experimented their approach on seven Atari 2600 games and it was proven to be capable of state-of-the-art performance in 6 out of the 7 games it was tested on without calibration in any hyperparameters and the network architecture[3].

TL is a technique in which a trained model produced from one training process is reused as the starting point of another training process, ergo a “transfer” of learning. In other words, TL allows the use of a trained model to be further trained and improved from one separate training process to another instead of having to start training from the ground up everytime a model is to be trained. TL is capable of retaining the knowledge gained from solving one task and applying that knowledge to a related but different task. TL is usually applied in deep ANN in which a pre-trained deep ANN is applied on a new task to achieve a reduced training period [5].

The paper that serves as the main inspiration for this project, Asawa et al. also experimented with TL in the realm of DRL by making use of networks that have performed well on PuckWorld[6] and adapted the networks into Snake[7] after training the networks a little more on the second game. They hypothesised that TL will increase the

convergence rate on Snake and maybe even produce better performance since PuckWorld and Snake have highly similar game mechanics. Their results showed that TL was indeed capable of improving performance and stability of DQN in playing Snake [8].

# 4.Design and implementation

## 4.1 Environment

A total of two game environments are used, Breakout and Space Invaders where the agents have 4 and 6 output actions size respectively. The game environment is provided by a python package known as *gym* [9] [10]. Both the game environments used are of version "v4" and type "Deterministic", "v4" is used because it has zero probability of action repetition so that the DQN agent can take full control of which action to take at any given moment. "Deterministic" type is used so that the game environments are generated based on a fixed frame skip of 4. Keeping the frame skip constant at 4 is important because the DQN agent is fixed at a training interval of 4, meaning that the network weights are only updated once every 4 frames [11]. The environments are preprocessed prior to training and the preprocessing process can be separated into 4 components:

1. Rewards clipping

   Rewards are clipped to a range of -1 to 1 to restrain the scale of error derivatives so that it is easier to use the same learning rate value on multiple different games. Besides, clipping rewards could also help an agent to cope better in different game environments since It is unable to distinguish between rewards of varying magnitudes [3].

2. Converting RGB images to grayscale

   The game environment is processed every 4 frames of images, so converting those images to grayscale can greatly reduce the amount of time needed for the convolutional layers to extract important information.

3. Resolution reduction

   The resolution of the game environment is resized to 84 by 84 from the original resolution of 210 by 160 so that it can be processed quicker.

4. Pixel Normalisation

   The pixel values of the images are converted to float values and divided by 255 so that all of it can be in the range of 0 to 1 because inputs of larger values will slow down the learning process of the agent.

## 4.2 Agents

DQN is essentially applying the Q-learning algorithm to a deep ANN ,ergo a Deep Q-Network. The reason for applying Q-learning to a deep ANN is because the computation of

Q- learning algorithm requires too much memory thus using it with a neural network would be more memory efficient [12]. Q-learning is a value-based, off-policy, model-free RL algorithm that aims to estimate the optimal actions to take given a state. It is known to be value-based because the Q-value is the primary component affecting the action selections, off-policy because the Q-learning function does not learn from actions that are inside of the current policy and therefore a policy is not required. In general, Q-learning aims to learn an optimal policy that maximises the total reward. The "Q" in Q-learning learning simply refers to quality, which represents the "goodness" of an action in terms of getting future rewards. When an RL agent carries out an action in its environment, the agent will in turn receive a reward based on the state of the environment. Therefore, the agent will constantly receive a state and action pair. With the state and action pair, Q-learning agent will interact with its environment in two ways, explore or exploit. The decision to either explore or exploit for Q-learning is usually done through a policy known as epsilon-greedy policy. Below is a pseudocode for the policy:

If random uniform num < epsilon:

Action = random action within action space

Else

Action = ArgMax(Q-value)

The agent explores simply by taking random actions that are within its action space and this is usually done at the earlier stages of the training process because it is important for the agent to be "adventurous" and discover different states and how its actions can affect those states. After some time of exploring, the agent exploits by selecting the action that would result in the highest future reward, essentially using the current state and action pair to predict and decide the next best action [13]. Below is the equation of Q-learning algorithm [14]:

$$Q(s,a) = \overbrace{Q(s,a)}^{\text{Old value}} + \alpha(\overbrace{r}^{\text{reward}} + \gamma \overbrace{maxQ(s'a')}^{\text{optimal future value}} - \overbrace{Q(s,a)}^{\text{Old value}})$$

$\alpha$: learning rate; $\gamma$: discount

Figure 1: Q-learning algorithm

Other than that, an Experience Replay mechanism is also present in DQN. Experience Replay fundamentally functions like memory that stores experiences including rewards, actions and state transitions which are then separated into mini batches of data to update the deep ANN by randomly replaying those "experiences" randomly [12]. Besides, DQN also utilises something called target network to stabilise the parameters of target function by overriding them with the latest network on a regular basis throughout training[15].

### 4.2.1 Network Architecture

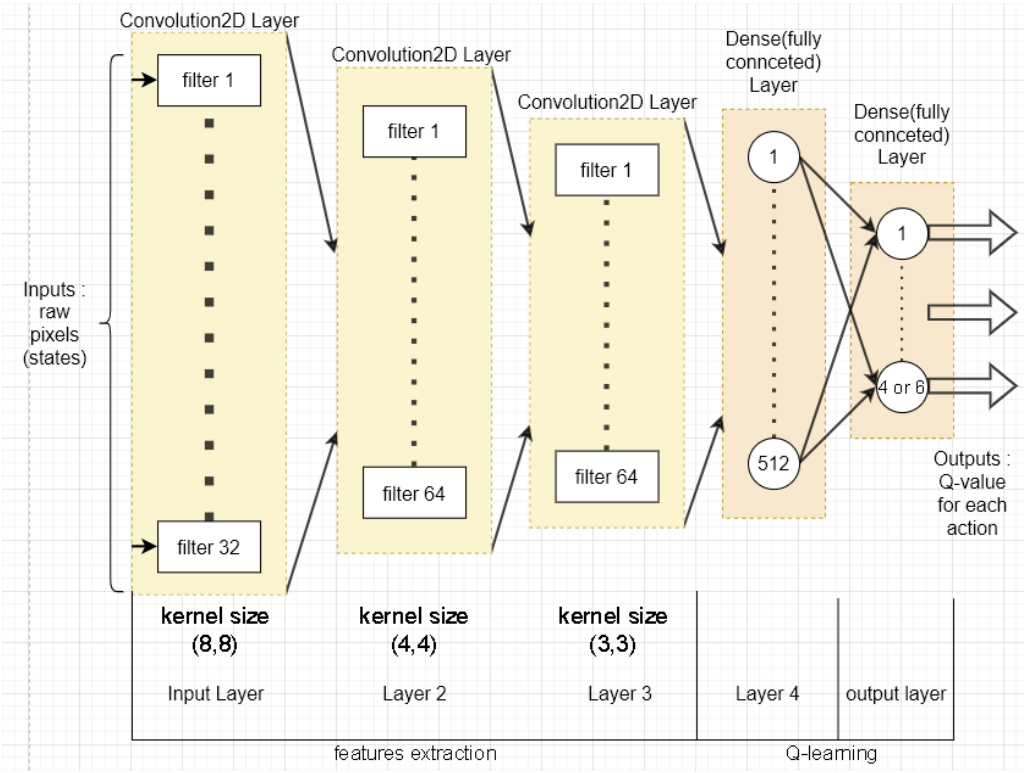


Figure 2: Networks architecture of DQN agents

Figure 2 is a basic illustration of the architecture of the model primarily used for all the DQN agents in this project. It is also the model that is described by Mnih et al. [4]. The first 3 layers of convolution layers are designed specifically to process the raw pixel inputs from the game environment to reduce feature dimensions and extract only the important features before passing those features over to the fully connected layers that are mainly in charge of learning and action selections. For the fully connected layers, there are only 2 layers, a layer of 512 neurons and the final output layer of 4 or 6 neurons, one for each action in the action space for the agents depending on the game environment. All of the layers made use of the rectifier linear activation function mainly to avoid a common deep ANN issue of vanishing gradient except for the output layer that uses a linear action function because the outputs are the actions for the agent so it should not be altered in any way.

### 4.2.2 Dueling Network Architecture

For this project, all DQN agents are trained with the dueling network architecture which was invented by Wang et al. that makes use of two separate neural networks, one which handles the state value function while the other deals with the state-dependent action advantage function. The main advantage of this approach is that it can generalise its knowledge across actions without interfering with the underlying DQN algorithm [16]. Based on the network architecture implemented for this project(Figure 2), the dueling networks architecture would be:

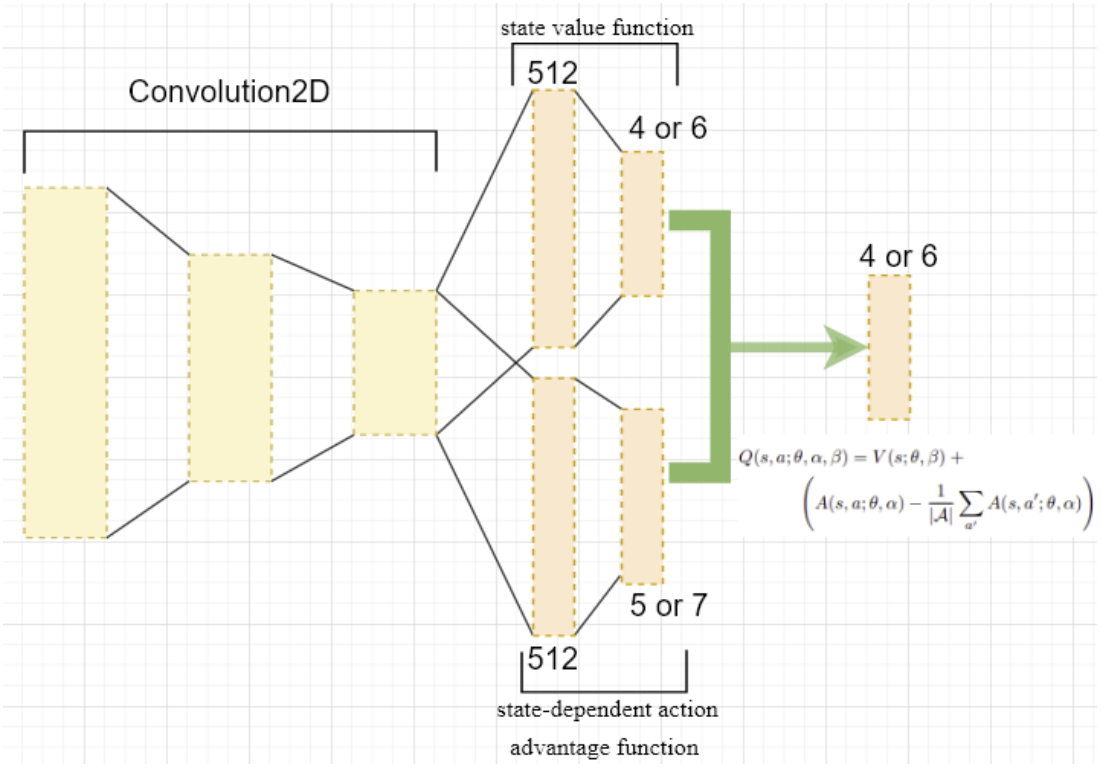


Figure 3: Basic illustration of Dueling Network Architecture [16]

All the DQN agents implemented in this project are done through the Keras-RL2 DQNAgent class [17].

### 4.2.3 Transfer Learning Implementation

Transfer learning can be done in multiple ways but the most common approach is to use a pre-trained network by “freezing” off the weights of all the layers after only taking the part of the network that contains the pre-trained convolutional layers. “Freezing” off weights refers to turning the weights to be not updatable. Usually the part that is responsible for feature extraction is the one that is transferred over to a different but related problem. With a pre-trained chunk full of convolutional layers, one would just need to add fully connected neural layers at the back of the network so that those layers can be trained to solve the problem at hand while the frozen convolutional layers remain unchanged since they are already pre-trained.

In the case of this project, an initial DQN agent network is trained on an Atari 2600 game before it is transferred over to another network to train on another different Atari 2600 game. After the first network is done training, the weights of its first 3 convolutional layers will be frozen off while the weights of its fully connected neural layers are reinitialised, essentially replaced by new ones but of the same architecture(Figure 2).

## 4.3 Logging of Network Weights and Training History

The network weights and training history are logged during training and all the files are uploaded to Google Drive right after the training is done. The final network weights and

training history are then taken from Google Drive(after google drive is mounted to Google Colab notebook) to be used for experiments.

# 5.Experiments

The full list of parameters used in the experiments are as below:

1. Number of training steps

   It is set to 1 million steps threshold which takes about 7 or 8 hours to finish the training process so setting the training steps any high would be impractical for this project.

2. Window length, training interval

   Window length and training interval  are constantly kept at value of 4 in order to align with the fixed number of frames skipped in the game. The DQN agents will stack 4 frames worth of observations before input to the deep learning model while the model will update its weights once every 4 frames.

3. Memory limit

   The maximum size of memory that the agent can hold at any given moment during training is set at 10000 which is a lot smaller than the memory limit set by Mnih et al. because I am only training my agents for 1 million steps so it is necessary for the memory limit to be downscaled proportionally.

4. Policy

   The exploration and exploitation policies used are the epsilon-greedy Q policy and the Boltzmann policy.

5. Epsilon

   Epsilon is important because it is the hyperparameter that determines how the agent can explore or exploit its actions and states. The DQN agents always start training with an epsilon value of 1 and it is decayed to a minimum of 0.1 over the first 200k steps with linear annealing policy. The epsilon is then kept at 0.1 for the remaining 800k training steps For testing, the epsilon value is kept at 0.05.

6. Discount factor(gamma)

   The gamma value represents the magnitude of influence of future Q- values(future rewards) on the agent which is set constantly at 0.99.

7. Target network update

   The target network is updated regularly every 10000 steps.

8. Learning rate(alpha)

As opposed to the usual case of decaying learning rate in most machine learning models, the learning rate for the models described in Mnih et al.'s paper was kept at a constant value of 0.00025 which is also the case for this project.

There is total of 3 questions for which 2 different experiments are carried out to answer those questions:

a. How would the agents' training performance differ over 1 million training steps?
b. Which agent can achieve higher mean rewards over 10 testing episodes?
c. How would the agents' training performance differ if they were to have a different exploration policy?

## 5.1 Experiment 1

A DQN agent pre-trained on Breakout for 1 million steps and transferred to a new model network to train on Space Invaders for 1 million steps whereas another DQN agent is trained solely on Space Invaders for 1 million steps. All the parameters and network architecture for both the agents are exactly the same except for the use of TL. The exploration strategies used for both are the epsilon greedy policy. Their training and testing performance are compared and analysed to answer questions a and b.

## 5.2 Experiment 2

This is a supplementary experiment serving as an extension of experiment 1 where the setup is exactly the same but the exploration strategy is switched to an alternative policy - Boltzmann(Soft-max) policy with tau value of 1.0. This experiment is done to answer question c.

# 6.Results

To reduce complexity, from here onwards, the DQN agent that is trained without TL will be known as Agent 1 whereas the one trained with TL will be known as Agent 2.

## 6.1 Experiment 1

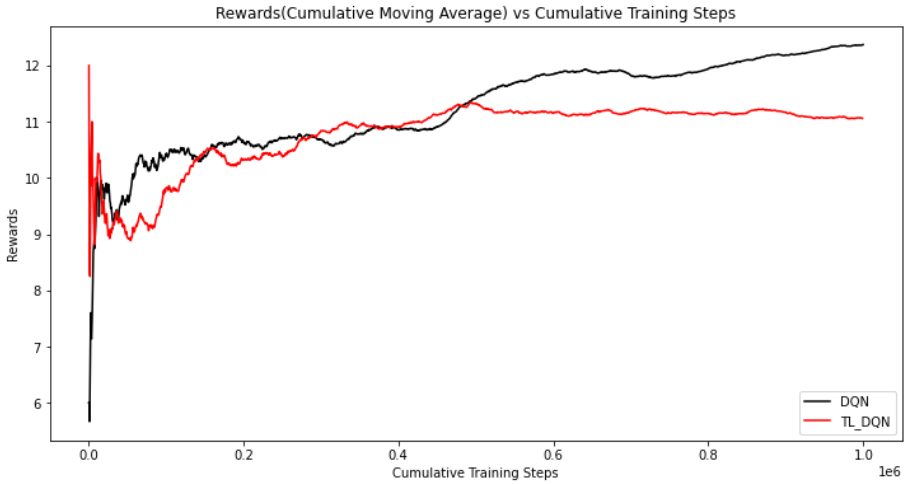


Figure 4: Rewards(Cumulative Moving Average) vs Cumulative Training Steps

Agent 1 did not manage to stabilise while its rewards are cumulatively higher than Agent 2 in around the last half portion of the training steps. Though Agent 2 has lower cumulative rewards but it did manage to stabilise.

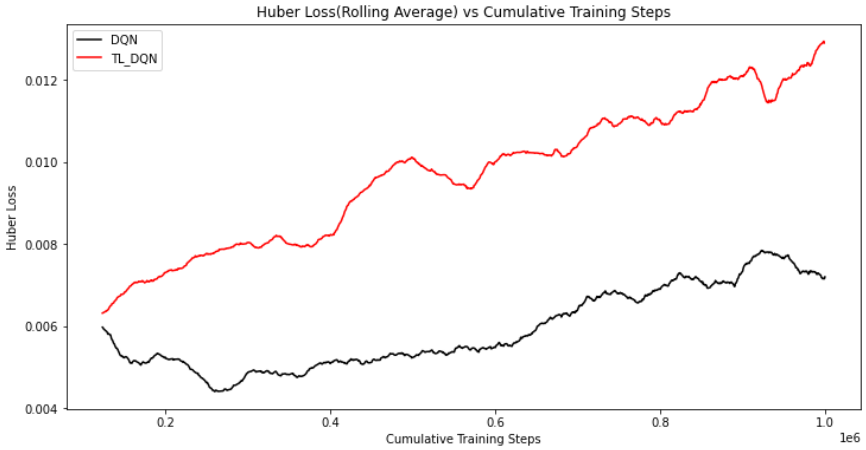


Figure 5: Huber Loss(Rolling Average) vs Cumulative Training Steps

Huber loss is calculated automatically by Keras-RL2 DQNAgent [17] based on the function below:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Figure 6: Huber loss Function [18]

Thus, for any loss that is below the delta clip value set at 1.0 for both agents , the Mean Squared Error will be computed whereas if any loss is above 1.0, the Mean Absolute Error will be computed.  Based on the graph above, the Huber loss for Agent 2 is higher than Agent 1 as the training progresses. This explains why agent 1 was capable of achieving more rewards than Agent 2 as Agent 2 consistently has more Q- value prediction errors than Agent 1.

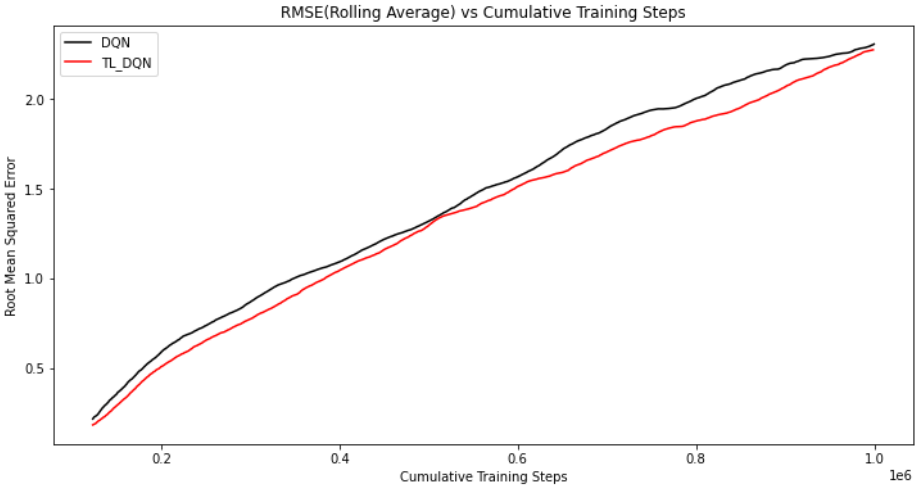


Figure 7: RMSE(Rolling Average) vs Cumulative Training Steps

Although the Huber loss graph shows that Agent 1 has smaller loss than agent 2, the rolling average of Root Mean Squared Error(RMSE) for Agent 1 is generally higher than agent 2. Since RMSE is responsible for magnifying huge prediction error outliers [18], the graph above indicates that Agent 2 generally has lower prediction errors which leads to Agent 2's ability to stabilise its cumulative rewards while Agent 1 struggles to do so albeit having higher cumulative rewards than Agent 2.

Both Figure 5 and 7 align well with Figure 4, showing that lower Huber loss but higher RMSE causes Agent 1 to achieve higher cumulative rewards but less stability while Agent 2 does the opposite.

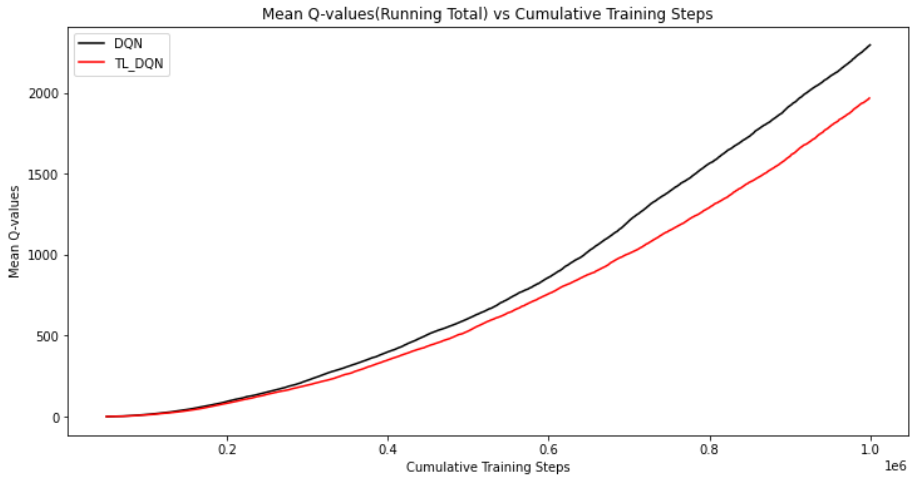


Figure 8: Mean Q-values(Running Total) vs Cumulative Training Steps

Q-value is essentially an estimation of the maximum expected future reward by taking a given action in a state [15]. The graph above is a correct indication of why Agent 1 is able to achieve higher cumulative rewards than Agent 2 because the running total of the mean Q-values for Agent 1 is gradually higher than Agent 2 as the training progresses.

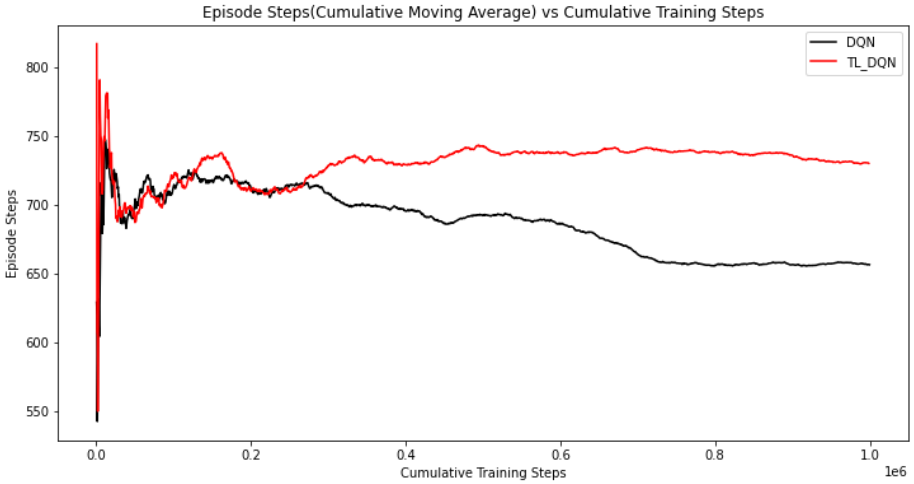


Figure 9: Episode Steps(Cumulative Moving Average) vs Cumulative Training Steps

Based on Figure 4 and 8, Agent 1 is capable of achieving higher cumulative rewards within a lower number of action steps per episode which is usually the desirable performance while Agent 2 obtains lower but more stable cumulative rewards without taking lower number of action steps. This proves that Agent 1 is able to exploit better than Agent 2 since it is able to achieve more with fewer steps.

```
Testing for 10 episodes ...
Episode 1: reward: 9.000, steps: 380
Episode 2: reward: 9.000, steps: 380
Episode 3: reward: 9.000, steps: 380
Episode 4: reward: 9.000, steps: 380
Episode 5: reward: 9.000, steps: 380
Episode 6: reward: 9.000, steps: 380
Episode 7: reward: 9.000, steps: 380
Episode 8: reward: 9.000, steps: 380
Episode 9: reward: 9.000, steps: 380
Episode 10: reward: 9.000, steps: 380
```

Figure 9: Agent 1 testing performance

```
Testing for 10 episodes ...
Episode 1: reward: 5.000, steps: 642
Episode 2: reward: 5.000, steps: 642
Episode 3: reward: 5.000, steps: 642
Episode 4: reward: 5.000, steps: 642
Episode 5: reward: 5.000, steps: 642
Episode 6: reward: 5.000, steps: 642
Episode 7: reward: 5.000, steps: 642
Episode 8: reward: 5.000, steps: 642
Episode 9: reward: 5.000, steps: 642
Episode 10: reward: 5.000, steps: 642
```

Figure 10: Agent 2 testing performance

Figure 9 serves as a good inference for the testing performance of both agents shown above. Within 10 testing episodes, Agent 1 is able to achieve 80% more average rewards than Agent 2 with around 40% fewer action steps per episode.

## 6.2 Experiment 2

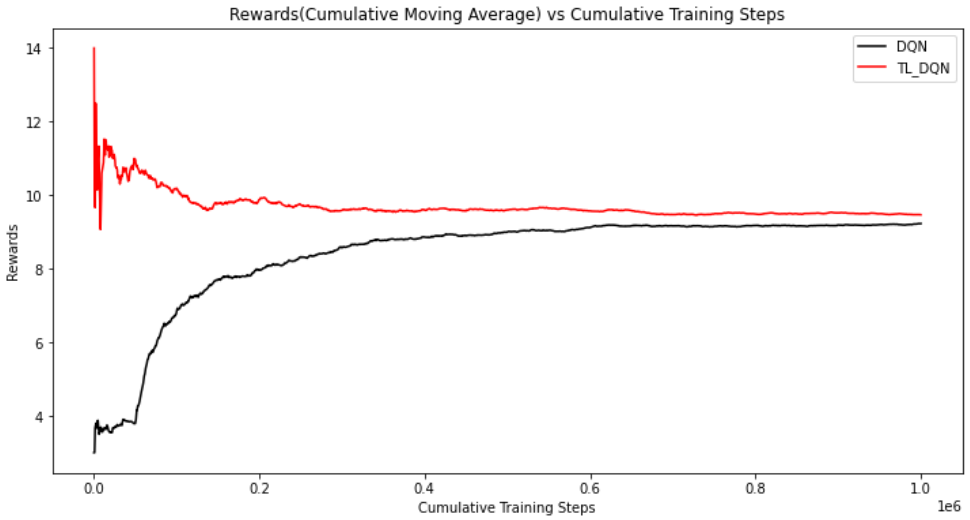


Figure 11: Rewards(Cumulative Moving Average) vs Cumulative Training Steps{Boltzmann}

In contrast to training the agents in epsilon greedy policy, Boltzmann policy greatly improves the performance of Agent 2, surpassing Agent 1 from the very beginning. It is apparent that although Agent 1 performance is slightly poorer, it is able to increase its cumulative rewards asymptotically before plateauing while Agent 2 is not able to increase its cumulative rewards but only able to plateau at a tight range slightly higher than Agent 1.

# 7.Discussion

The expectation for the results of experiment 1 is that Agent 2 is able to have not just a more stable but also better performance in terms of rewards as compared to Agent 1 like how it is concluded in Asawa et al. 's paper [8]. Unfortunately, the statistical analyses proved otherwise. Though it is worth noting that Agent 2 is more stable than Agent 1 if one were to disregard the rewards, further proving the robustness of TL in terms of learning stability. Agent 2 is not able to achieve the expected results due to multiple reasons. First, there is an extremely high possibility that both agents are severely underfitted for Atari game environments that are known to be complex and challenging testbeds for RL [3]. The agents presented in the paper by Mnih et al. [4] were trained for a total of 50 million steps each whereas the agents in this project are trained for only 1 million steps. Therefore, if more training were given to both agents, Agent 2 might have been able to surpass or at least perform roughly at the same level as Agent 1 but with higher stability. Second, Asawa et al.[8] were able to demonstrate that TL is a good approach in improving agent performance and stability by experimenting with two game environments with highly similar mechanics while this project is experimenting TL on two Atari games, Breakout and Space Invaders, both have different action space size as well as slightly different game mechanics. Breakout requires the agent to move a paddle around to bounce a ball to hit static objects while Space Invaders requires the agent to move around while shooting at other moving objects. TL is justifiably applicable because both

games still have sufficient fundamental similarities to be considered as related but different problems. However, it might be due to the small differences that both the games have that causes the agent not able to transfer its knowledge from Breakout to Space Invaders effectively.

As evident from experiment 2 that exploration strategy plays an important role. Finding a good balance between exploration and exploitation in DRL is still an ongoing research so there is no one strategy that can be universally robust. As for the reason why the Boltzmann policy enables Agent 2 to achieve higher rewards than Agent 1 while epsilon-greedy policy does the opposite, this can be supported by Koroveshi and Ktona's paper which concluded that Boltzmann policy has a slightly better performance than epsilon-greedy policy [19]. However, epsilon-greedy is still the common approach to DQN for Atari games because it substantially yields better results when training scratch without TL [3] [4].

# 8.Conclusion

## 8.1 Limitation

The biggest issue faced in this project is the time constraints. It is quite difficult to be even more rigorous in terms of training, testing and experimenting with the agents because it took a lot longer than expected to finish the training process of a DQN agent model although the environments were preprocessed to reduce the amount of raw pixel data to the least possible without losing too much important information.

## 8.2 Future work

If there were more time, a more rigorous experiment can be done in terms of TL approach. Fine tuning can be done by unfreezing the convolutional layers to retrain the entire model with a lower learning rate. Eventually, the entire flow would be :

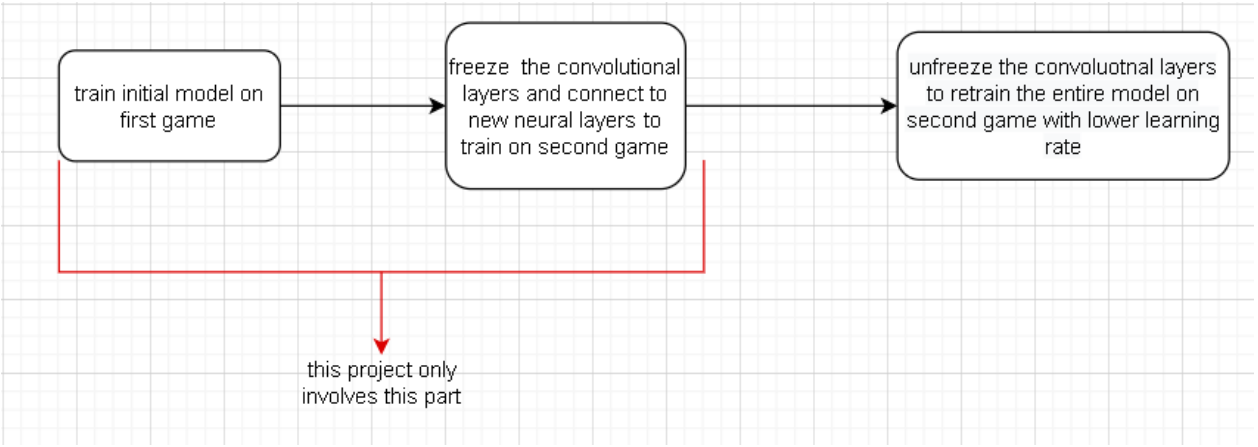


Figure 12: Future work TL workflow

# 9. References


[1] C. Berner *et al.*, 'Dota 2 with Large Scale Deep Reinforcement Learning', *ArXiv191206680 Cs Stat*, Dec. 2019, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1912.06680

[2] D. Silver *et al.*, 'Mastering the game of Go with deep neural networks and tree search', *Nature*, vol. 529, no. 7587, Art. no. 7587, Jan. 2016, doi: 10.1038/nature16961.

[3] V. Mnih *et al.*, 'Playing Atari with Deep Reinforcement Learning', *ArXiv13125602 Cs*, Dec. 2013, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1312.5602

[4] V. Mnih *et al.*, 'Human-level control through deep reinforcement learning', *Nature*, vol. 518, no. 7540, Art. no. 7540, Feb. 2015, doi: 10.1038/nature14236.

[5] S. Bozinovski, 'Reminder of the First Paper on Transfer Learning in Neural Networks, 1976', *Informatica*, vol. 44, no. 3, Art. no. 3, Sep. 2020, doi: 10.31449/inf.v44i3.2828.

[6] 'PuckWorld — PyGame Learning Environment 0.1.dev1 documentation'. https://pygame-learning-environment.readthedocs.io/en/latest/user/games/puckworld.html (accessed May 10, 2022).

[7] 'snake-gym · PyPI'. https://pypi.org/project/snake-gym/ (accessed May 10, 2022).

[8] C. Asawa, C. Elamri, and D. Pan, 'Using Transfer Learning Between Games to Improve Deep Reinforcement Learning Performance and Stability', p. 8.

[9] Gym Community, *gym: Gym: A universal API for reinforcement learning environments*. Accessed: May 10, 2022. [Online]. Available: https://www.gymlibrary.ml/

[10] G. Brockman *et al.*, 'OpenAI Gym', *ArXiv160601540 Cs*, Jun. 2016, Accessed: May 10, 2022. [Online]. Available: http://arxiv.org/abs/1606.01540

[11] 'Difference between Breakout-v0, Breakout-v4 and BreakoutDeterministic-v4? · Issue #1280 · openai/gym', *GitHub*. https://github.com/openai/gym/issues/1280 (accessed May 10, 2022).

[12] 'What Are DQN Reinforcement Learning Models', *Analytics India Magazine*, Jun. 10, 2021. https://analyticsindiamag.com/what-are-dqn-reinforcement-learning-models/ (accessed May 10, 2022).

[13] A. Violante, 'Simple Reinforcement Learning: Q-learning', *Medium*, Jul. 01, 2019. https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56 (accessed May 10, 2022).

[14] M. Buchholz, 'Deep Reinforcement Learning. Introduction. Deep Q Network (DQN) algorithm.', *Medium*, Mar. 16, 2019. https://markus-x-buchholz.medium.com/deep-reinforcement-learning-introduction-deep-q-network-dqn-algorithm-fb74bf4d6862 (accessed May 10, 2022).

[15] 'Welcome to Deep Reinforcement Learning Part 1 : DQN | by Takuma Seno | Towards Data Science'. https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b (accessed May 10, 2022).

[16] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, 'Dueling Network Architectures for Deep Reinforcement Learning', *ArXiv151106581 Cs*, Apr. 2016, Accessed: May 10, 2022. [Online]. Available: http://arxiv.org/abs/1511.06581

[17] 'taylormcnally/keras-rl2: Reinforcement learning with tensorflow 2 keras'. https://github.com/taylormcnally/keras-rl2 (accessed May 10, 2022).

[18] G. Seif, 'Understanding the 3 most common loss functions for Machine Learning Regression', *Medium*, Feb. 11, 2022. https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3 (accessed May 10, 2022).

[19] J. Koroveshi and A. Ktona, 'A comparison of exploration strategies used in reinforcement learning for building an intelligent tutoring system', p. 7.