Deep Reinforcement Learning on Atari Game Environments

Zachery Fogg
College of Engineering and Mathematical Sciences
University of Vermont
Burlington, Vermont, USA
zach.fogg@uvm.edu

I. INTRODUCTION

When we are presented with the challenge of learning a model that can act optimally in a sequential decision environment, we use reinforcement learning (RL) to train that model (also called an agent). There are many ways to implement RL to train such an agent, but as the environments become more complex, deep RL methods are some of the only ones capable of learning models that behave well (1).

Agents are essentially functions that take a game state as input and output some action, and because deep neural networks can learn to approximate any function, deep learning can be used to approximate some function of the agent. Deep RL models are 'deep' in the sense that they incorporate a neural network to approximate some function of the agent: whether that be the agent's policy, value function, or another function.

Atari environments are examples of environments that are complex enough to warrant deep RL. There are a variety of deep RL methods that have shown success in these environments in the relevant literature and in this report, I will make use of a few of them. And while these models are often able to achieve superhuman performance in some simpler Atari environments (i.e., Breakout or Pong), again, as the environments become more complex do does the task of learning to behave optimally (1). Maze style environments such as MsPacman, Alien, and Bank Heist require upwards of 10 million timesteps to see good performance (1).

An issue arises, in that these deep agents take many days of continual training on cutting edge GPUs to see success, and while they perform well on the environment they trained in, when evaluated in another environment, their knowledge does not typically transfer (6); this makes them only useful for one specific environment.

In this paper I will explore how a deep RL agent trained in one Atari maze environment is able to transfer its knowledge to a second Atari maze environment. This technique is referred to as transfer learning (TL) and has been shown to perform well in many image classifications tasks.

If transfer learning has truly taken place, then the pretrained agent should be able to noticeably outperform a second agent trained from scratch in the second environment, in the same amount of training timesteps.

When an agent is transferred to another environment and retrained, it is liable to experience a phenomenon coined the Catastrophic Interference (CI). Catastrophic Interference is when a model trained on a second set of data (or here, an agent trained in a new environment) 'forgets' what it learned about the first dataset and no longer performs well on the first dataset. Transfer learning and the Catastrophic Interference are therefore closely linked; thus, I will also be analyzing how

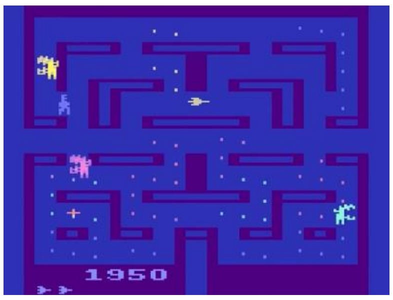


Figure 1 : Atari 2600 Alien environment

an agent, trained in a maze environment, and then transferred to a second maze environment, performs when evaluated back in its initial environment.

However, to be able to explore TL and CI, I must first have an agent that has learned something worth transferring; I first implement an agent that is able to achieve an impressive level of performance on a maze environment.

In this report, I perform five basic steps:

I. Implement a deep RL agent on an Atari environment.

II. Train that agent to a point where it can pass the initial level of the environment (If this can be achieved, it indicates that the agent has clearly learned something useful).

III. Train a second agent of the same successful architecture on the second environment; this agent will act as a baseline to determine how effective transfer learning is in this context.

IV. Transfer the first trained agent to the second environment and compare its learning curves and performance to that of the baseline agent: this is TL.

V. Evaluate the first agent on the first environment after it has trained on the second environment for some time: this is CI.

To summarize the results I achieve in this report: I successfully train a deep RL agent on the environments MsPacman and Alien; successful meaning that it consistently passes one or more levels per episode. I attempt to transfer the agent trained on Alien to the environment Bank Heist, where the transferred agent is expected to learn faster than an agent trained from scratch on Bank Heist. Lastly, I demonstrate that the transferred agent does indeed experience the CI phenomenon in regard to the Alien environment after training on Bank Heist.

II. PROBLEM DEFIITION AND ALGORITHM

A. Task Definition

The environments used are sourced from OpenAI gym. These Atari environments are distinct from many other game environments in the state that they give to the agent. They only provide the raw pixel output of the environment as state and nothing more. The agent does not have access to internal game state or any other information. This provides a large challenge for the agent as initially the agent does not understand basic concepts such as: where its sprite is located or what it looks like, where the enemies are located, the constraints of walls, and the list goes on. In fact, the agent does not receive any rules of the environment and does not even understand that its action effect what it is seeing.

The agent receives images and rewards and must learn to decern everything relevant about the environment and how its actions effect the environment. This makes the task especially interesting as the agent is learning in a similar way we would – purely visually.

To elaborate, there are two things about the input to the agent that are important to specify: state and reward. In terms of state, the environment initially outputs consecutive 240 x 160 x 3 matrices (a 240 x 160 color image), but a wrapper around the environment stacks 4 frames together, converts them to grayscale, and down samples them to 84 x 84. Thus, the actual state that the agent receives is an 84 x 84 x 4 array representing 4 consecutive frames. The agent must receive four frames stacked together so that it is able to determine the trajectories of sprites, flashing objects, and movement (broadly speaking). This implies that the agent acts every four frames.

The reward that the agent receives is based on the change of score per timestep: +0 for no score increase or +X where X is the increase in score. A negative reward will not be observed in these Atari environments. Some agent implementations choose to truncate this to +0 and +1 respectively.

The agents I implemented all use the same architecture of convolutional neural network (CNN). This CNN takes the 84 x 84 x 4 input and outputs values to an output layer of the size of the action space: 9 for MsPacman and 18 for Alien or Bank Heist. The activation of the output layer depends on the agent being implemented; some expect probabilities, and some expect raw values.

Before moving forward, it is useful to briefly explain the environments that are going to be used in this experiment, as they serve as the 'dataset':

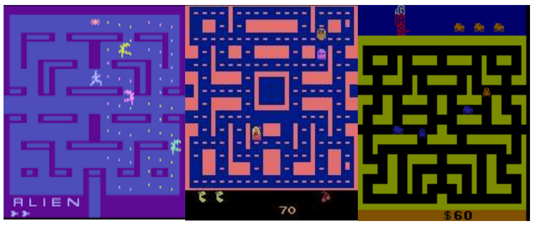


Figure 2 : Alien MsPacman Bank Heist

In Alien, the agent traverses a maze colleting pellets for reward and avoiding enemies that can kill it. If the agent picks up a power pellet (higher reward), then it can kill the enemies and earn a very high reward; otherwise, the agent must avoid enemies. The agent is equipped with a weapon that it can fire when in close proximity to enemies that will immobilize them temporarily. If an agent collects all pellets in a maze it will advance to the next level; it has three lives per episode.

In MsPacman, the agent traverses a maze collecting pellets for reward and avoiding enemies that can kill it. If the agent picks up a power pellet (higher reward), then it can kill the enemies and earn a very high reward. The agent has no weapon and can only move about the maze. If the agent collects all pellets in a level, then it will advance to a next level. The agent has three lives per episode.

In Bank Heist, the agent moves about the maze collecting sparsely placed 'banks' that spawn. When the agent collects a bank, a cop will spawn that can catch and kill the agent. The agent can drop dynamite behind it to kill the enemy. This dynamite is on a slight delay and thus can kill the player if it backtracks after dispatching one. The agent has a gas tank that slowly runs out as it moves, if the gas tank empties then the agent dies. The agent can collect banks to earn more gas and collect all the banks in a given level to earn much more gas. The agent can exit a level, and escape the cops that have spawned through multiple exits along the wall of the maze. The agent has 4 lives per episode.

B. Algorithm Definition

I implement multiple styles of agents in this report to ulimately find a method that performs well on these environments, but the basic algorithm structure remains the same and approximates this diagram:

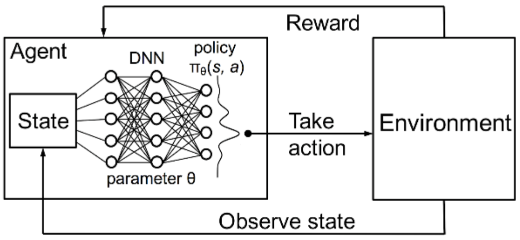
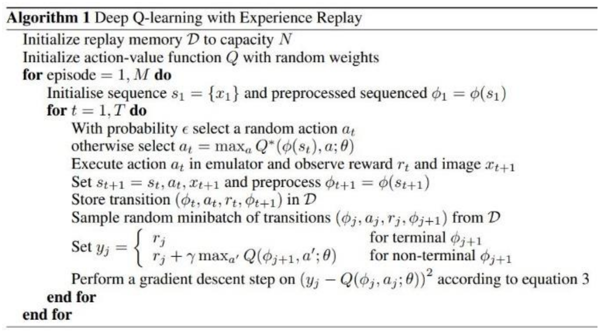


Figure 3 : Diagram displaying how a Deep RL agent would interact an environment. The agent shown here is a policy gradient agent that uses a CNN that outputs the probabilities of taking each action

The basic algorithm loop follows: the agent observes some state and reward from the environment → It passes the state through a CNN and obtains some output → The agent will interpret these outputs in some fashion (depending on the method) and then choose an action to step in the environment with; this action selection will depend on whether or not the agent is training or testing. If the agent is training, then the agent will use this reward and state returned to train and update its policy → The agent steps in the environment with that action → the Agent receives a new reward and a new state back from the environment → The loop continues.

I implement two styles of agents in this report. The first agent is a Q-Learning agent called a DQN; this agent was initially proposed by DeepMind in 2013 (3) for Atari games. The pseudocode for the algorithm is as such:



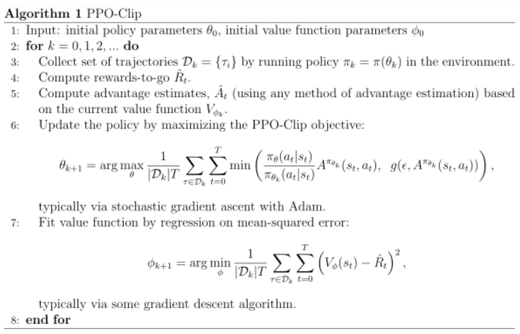
```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Figure 4 : DQN pseudocode as proposed by DeepMind (3)

To summarize the method: The agent learns a parametrized function Q(s,a), called a Q function, that maps state and action pairs to Q values. A Q value indicates how much discounted, long term reward the agent expects to earn if it takes action a in state s and then continues to follow its current policy to a terminal state. Then, assuming that we have an optimal Q function, the agent can act optimally just by greedily choosing the highest valued action given any state. The task is learning an optimal Q function for an environment by interacting with that environment repeatedly. The DQN uses two CNNs that share the same weights: a policy network, and a target network (the target network derives weights from the policy network every certain number of steps). The agent learns, as show above, by choosing an action with its policy network, evaluating that action with the target network which generates a target value, and then moving towards that target value by back propagating on the loss squared error between the target and the actual return (calculated by observing the reward for its action).

DQNs have been shown to perform well in some Atari environments (3) and so seemed promising for this task.

I also implement an actor-critic method called Proximal Policy Optimization (PPO):



```
Algorithm 1 PPO-Clip
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

Figure 5 : Pseudocode of PPO algorithm (4)

PPO optimizes a parametrized policy directly instead of learning a Q function and then inferring some optimal policy from it. PPO optimizes the actor network, while a separate critic network optimizes a value function used in the objective function. PPO optimizes the above objective function by performing gradient ascent on the objective function. PPO uses the same CNN.

What sets PPO apart from other actor-critic methods is its focus on stability. PPO uses a previous iteration of its policy in the objective function to ensure that the updated policy is only marginally different from the previous policy; this helps with learning stability. PPO also clips updates to within a certain range to, again, ensure stability in training.

Both methods use a CNN of the following architecture:

- Conv2d (32 filters, kernel = (8,8), stride = (4,4)
- Conv2d (64 filters, kernel = (4,4), stride = (2,2)
- Conv2d (64 filters, kernel = (3,3), stride = (1,1)
- Dense(X)
- Dense(Y)

The number of hidden units X and output units Y depend on the environment. Relu activations are used for all layers accept the output: in the output layer a Softmax or Linear activation will be applied depending on the method.

III. Experimental Evaluation

A. Methodology

To evaluate any given agent's performance, there is only one numerical metric that is relevant: the mean episode reward that an agent is achieving. Measures like loss, or MSE do not hold any real information as they are generated by the agent for the agent (not in relation to any ground truth values) and don't correspond to performance in the environment. In addition, the only goal the agent has is to maximize long term reward (long-term implies total reward per episode), thus any other metric would be largely irrelevant.

In addition, watching how an agent is actually acting in an environment is an important, although not strictly measurable, way of evaluating its performance.

To reiterate, unlike traditional supervised learning tasks, we do not have a dataset; the agent learns in an environment and then is later evaluated in that same environment. I use OpenAI gym's implementation of the Atari 2600 games Alien, MsPacman, and Bank Heist. To evaluate an agent, I will be recording per episode reward as the agent trains. Once an agent finishes training, I initialize a new environment and save the frame output as the agent acts in the environment to get a video of its performance. I also record the reward collected in these test runs; these three aspects – reward curve during training, a video of the agent acting, and the testing reward – give us all the performance data we need to evaluate an agent. Agents train for tens of millions of timesteps in a given environment.

When graphing the training curves (reward per episode over the training period) because the episode rewards are so sporadic, the graphs can be somewhat indiscernible. Therefore, I take the mean over some number of consecutive rewards and graph this mean reward instead of each episode reward. Moving forward when I refer to an agent's 'performance', I am referring to the mean episode reward that they achieve in an environment.

My initial hypothesis for this project – with the goal of testing how effective TL is in these Atari environments – was that an agent trained in MsPacman to complete one or more levels, should achieve the same performance in Alien, when transferred, as an agent trained from scratch in Alein, in half the number of timesteps. That is to say that the transferred agent should achieve the same performance in x timesteps, that the agent trained from scratch achieves in 2x timesteps. I did not specify the number of timesteps as I was not sure how long these agents would need to train to achieve appreciable results.

My intuition when choosing this hypothesis was the following: The agents make use of a CNN to extract useful features from the environment and build up a feature representation; this can be thought of as forming a representation of state, where that representation is made up of features that the agent has learned are useful. It seems reasonable that, like in image classification tasks, agents training on similar maze environments with similar mechanics, should learn some of the same filters as one another. An agent starting off with many useful (or close to useful) filters should enjoy a speed up in training as it can more quickly form a useful state representation.

During the project I changed the start environment from MsPacman to Alien and switched the transfer environment from Alien to Bank Heist. I will still report on both as the MsPacman experiments were instrumental in the final choice of agent.

B. *Results*

I originally implemented a DQN on the MsPacman environment. After 1.5 million timesteps (about 24 hours on a Tesla V100) the agent was receiving the following results:

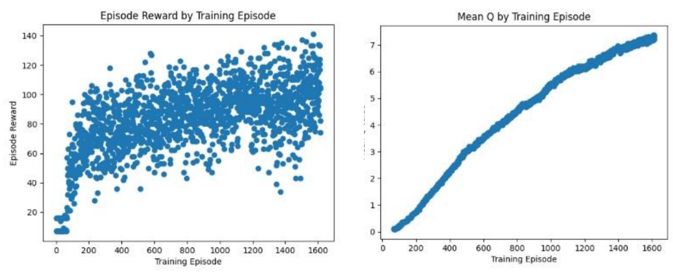


Figure 6 : DQN on MsPacman after 1.5 million learning timesteps. Episode reward is scaled by 1/10th. An episode reward here of 60 indicates a score of 600.

This performance originally seemed promising, and I continued training the agent for an additional 8.5 million steps for a total of 10 million timesteps. I graphed Mean Q values here, which is a measure specific to Q – learning methods like the DQN. This Mean Q value tells us how much discounted, long term, reward the agent expects to get given the state-action pairs it is experiencing. I originally interpreted this steady increase as a sign of good performance as I thought it meant the agent was getting more reward on average; this is not the case however.

The agent is merely predicting that it will get higher and higher rewards. If the agent is predicting that it will get higher and higher rewards but is not actually achieving those predicted higher rewards, then this becomes a problem; DeepMind researchers call an agent in this situation 'delusional' (5). The agent thinks it is going to experience rewards that it will not in reality. The agent begins bootstrapping off of its inaccurate estimates to form more inaccurate estimates that cause it to choose increasingly sub optimal actions. This usually starts a downward spiral of performance that the agent is not able to recover from. This is caused by a maximization bias in the way that the DQN updates where it is using the target network to both choose an action in the next state and assign a value to that selection (the network will obviously assign a high value to an action that it chose because it was the network to choose that action in the first place) (5). When examining the performance of the DQN after 10 million timesteps, it is evident that this likely occurred:

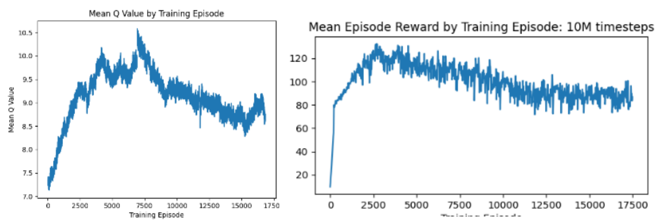


Figure 7 : DQN Mean Q and Mean episode reward after 10 million timesteps. In the right figure the x axis is training episodes and the y axis is Mean episode rewards – the axis labels were somehow lost

Examining this figure, we see that the Mean Q value becomes erratic, causing a steady decline in reward that the agent does not pull out of. The video of this agent's performance can be seen here. The agent only manages to complete about two thirds of a level and its movement is very sporadic. In addition, it often gets stuck in corners.

In theory, Double Q Learning should remove this maximization bias by using two different networks in the target calculation (5). In practice it did not yield any better performance. Unfortunately, the graphs of training performance were lost, but the training curves were very similar to that of the DQN. The differences between Q-learning and Double Q-learning are displayed in this graphic for reference:

Basic Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Double Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \boxed{Q'(s_{t+1}, \boxed{a})} - Q(s_t, a_t))$$

estimated/expected Q-value

$$\boxed{a} = \max_a Q(s_{t+1}, a)$$

$$q_{estimated} = \boxed{Q'(s_{t+1}, \boxed{a})}$$

One important note here is that training must be stopped every 3 million timesteps to comply with Deep Green's (the computing cluster that this agent was trained on) 48-hour time restriction. Because a DQN makes use of an experience replay buffer, when the agent is trained in these increments it must begin filling this buffer anew every time. If I were to perform this experiment again, I would have stored the replay buffer's data and reloaded it upon training continuation. This likely did not impact training however as the buffer is only 50,000 experiences long which fills after less than an hour, and the buffer only needs 1000 experiences for the agent to begin learning (which it acquires in a matter of minutes).

After this failure with Q-learning approaches, I switched learning methods to the PPO agent described earlier. PPO is able to make use of multiple workers (multiple agents in different environments all gathering experience to learn with) and thus training was much faster with the PPO agent. After 50 million timesteps (2 days of training) the agent was performing very well in the environment:

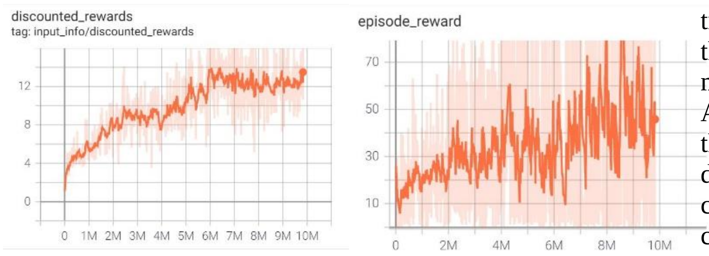


Figure 8 : PPO agent on MsPacman after 50M timesteps. Results displayed in Tensorboard instead of matplotlib, hence the distinct style

The PPO agent had achieved a mean episode reward of 4651. An agent will usually collect about 2500 reward per level so a mean reward of 4651 indicated that agent was, on average, completing one level and most of a second every episode. This video shows the agents testing performance, at the end of the 50 million timesteps of training, in the environment. Observing the agent, we see very good performance. The agent moves smoothly about the maze targeting pellets. The agent has no trouble chasing lone pellets to complete a level. The top agent in the video beats two levels in one episode, which more than satisfies the criteria to begin transferring the agent to another environment.

MsPacman has an action space of 9 while Alien has an action space of 18. Switching environments should have been as simple as swapping the output layer of the agent for one with 18 output neurons instead of 9. However, when attempting to do this, I was experiencing an endless number of shape mismatch errors between the tensors flowing through the network. After repeated failure, I instead decided to simply switch the initial learning environment to Alien (action space 18) and the transfer environment to Bank Heist (action space 18) to avoid this issue. This required a PPO agent to be trained from scratch on Alien first:

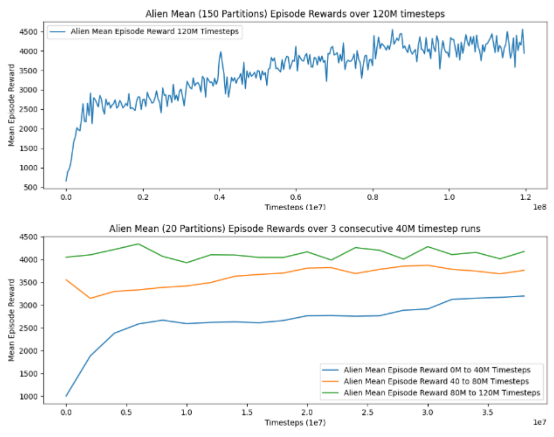


Figure 9 : PPO agent Alien learning curve after 120 million timesteps

After 120 million timesteps of training, the agent was performing very well in the Alien environment and was consistently passing the first level; sometimes the second. In the above figure I first display the mean episode reward over the entire training period. The learning curve is very steady which is where the PPO algorithm excels. The bottom figure displays the agents mean reward over the three consecutive 40 million timesteps blocks of training. I think this is an interesting comparison as it shows how the agents learns less and less as training progresses. The final 80 – 120 million training block shows a final mean reward at 120 million steps that is only marginally higher than the mean reward at 80 million steps. This video displays the performance of the Alien PPO agent after 120 million training steps. Similar to the MsPacman agent, the Alien agent moves smoothly and decisively throughout the maze, avoiding enemies and collecting all pellets. This bottom agent beats two consecutive levels in the first episode displayed in the evaluation.

Having this trained agent in hand, I then trained a PPO agent in Bank Heist for 80 million timesteps to act as the baseline agent. I then transferred the trained Alien agent to Bank Heist and trained it for 80 million timesteps as well. At 40 million timesteps I evaluated it back in the Alien environment to observe the Catastrophic Interference (CI). Below are the results of the transfer learning experiment:

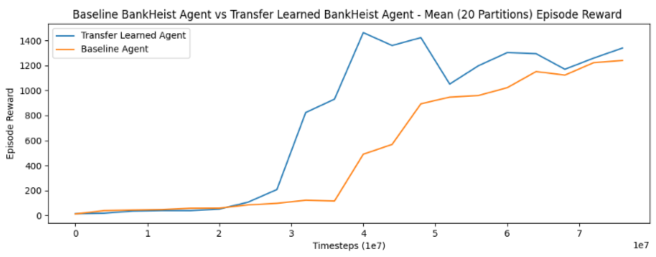


Figure 10 : Transfer learning experiment between Alien and Bank Heist comparing learning curves of the transferred agent and baseline agent after 80 million timesteps.

These results are in support of the initial hypothesis but do not appear as I expected them to. The transfer agent does indeed achieve the performance in 40 million timesteps that the baseline agent achieved in 80 million timesteps. The learning curves are however different then I had envisioned them being. I expected the transfer agent to initially diverge upward from the baseline agent using its learned filters from Alien; this was not the case. The transfer agent and baseline agent have the same performance for the first 20 million timesteps before the transfer agent diverges in performance.

The transfer agent then reaches a mean episode reward that it stays at for the rest of training. The baseline agent also converges to the same mean episode reward, but at a much slower pace. These results suggest that TL may have had provided a benefit for the agent, but I question this. Observing both the transfer and baseline agent acting in the environment we get an insight to what is actually occurring. This video shows the transfer agent and this video shows the baseline agent.

Observing either of these videos initially, a graphical glitch appears to be manifesting, but upon slowing the video to .25 speed we see that it is not. The agent learned that a bank will sometimes spawn at the start or end of a level, and so if the agent rapidly enters and exits levels it will collect banks. This strategy prevents cops from ever spawning to chase the agent. This strategy is not optimal however as the agent does not collect enough banks per level to refill its gas tank fast enough; this results in it slowly and predictably dying every episode as gas dwindles down. Both agents learn the same exact strategy. Because of the nature of PPO making very small policy updates and the profundity of this local optima, the agent is almost certainly never going to progress past this behavior. Reexamining the learning curves, it is plausible that the agents only started achieving higher rewards when they adopted this strategy.

This indicates that this environment may be too complex and too exploitable for a PPO agent to learn how to behave even semi optimally, unlike Alien or MsPacman where the agents learned a strategy that is similar to a human. Bank Heist may require too much strategy and long-term planning for the agent.

This strategy would also suggest that transfer learning was not a key factor in the transfer agent achieving a higher reward faster than the baseline. The transfer agent likely stumbled upon this strategy earlier than the baseline agent by 'luck' and not due to its previous knowledge.

This is not to say that transfer learning is not effective with deep RL on Atari environments; I believe that it still may be, just that Bank Heist was not an environment that was conducive to its success.

Unfortunately, after I had observed theses videos of the agent and reached these conclusions, I had been training these agents for weeks and restarting with a different environment from scratch would have taken longer than the amount of time left to complete the project.

Expectedly, the transfer agent, when transferred from Bank Heist back to Alien, had forgotten how to perform well in the environment to any degree at all. While I expected this would have likely happened regardless of the environments involved, the CI was doubt exacerbated by the unconventional strategy learned for Bank Heist where the agent did not need to learn how to navigate a maze, avoid enemies, or chase pellets; only to blindly move back and forth rapidly. This video shows the transfer agent transferred back to Alien after 40 million steps in Bank Heist (mean reward 321); this video shows the results after 80 million steps (mean reward 200).

C. *Discussion*

Empirically, the hypothesis is supported: the transfer agent achieved the same mean episode reward in 40 million steps that the baseline agent achieved in 80 million steps. However, upon further examination I do not believe that transfer learning was useful for a PPO agent learning to act in Bank Heist having learned on Alien. I do not think that this failure indicates that transfer learning would not be applicable in other environments but is instead due to Bank Heist being too complex to easily learn and easily exploitable, removing the need to learn. I believe that given a different pair of environments that are not so easily exploitable, transfer learning would show some value.

Unfortunately, after receiving these results and drawing the conclusion that Bank Heist should not be used in the experiment, I did not have time to rerun the experiment. To get these results took weeks of continual training and swapping out the environments to experiment again would have taken more time than was left in the semester.

IV. Related Work

Many papers regarding Deep Reinforcement Learning do not delve into the topic of transfer learning. Most of the papers that do explore the topic do not explore Atari environments except one that I read. My experiments differ from theirs in the complexity of environments used; they use the environments Pong and Breakout (6). Initially this may not seem to be that much of a difference however, observing results obtained by DeepMind (5) we can see that deep RL agents achieve 1500% human level performance on environments like Breakout and 20% of human level performance on maze environments like MsPacman. The difference between Pong, where the agent simply must align its paddle with the same singular ball, and Alien, where the agent must learn long term strategies to navigate a maze, avoiding enemies and seeking reward (in an action space of 18), is astronomical. I do not believe they are comparable.

Other papers exploring the topic, do so on other non Atari environments. As explained at the beginning of this report, Atari environments are very different from other game environments (the state that the agent receives being purely visual) and my theory of useful convolutional filters being transferred would not apply.

A very popular paper investigating transfer learning in deep reinforcement learning does not explore environment to environment learning (as I have), but instead human expertise to agent learning. An example of this style of transfer learning is how in DeepMind's original GO playing agents, the agents were given expert human games to learn from. To summarize, there is not much relevant work that addresses this same problem.

V. Code and Dataset

The code for this project can be found in this repository. Here you will find the PPO and the DQN agent implementations. The agents went through many iterations as different environments were swapped out, so the code shows how agents can be trained on an environment for only 1000 timesteps; except the colab notebook where most of the later training and experimenting was performed.

When I was experiencing the DQN becoming 'delusional' I used an 'out-of-the-box' agent to make sure that my DQN implementation was not the reason that the agent was not learning. It was not my implementation, but the DQN method. These stand-in agents were sourced from keras-rl and were only used to prove that the dqn and double dqn methods were not applicable for these environments after I had implemented the DQN from scratch on my own.
The environments for this project can be found in OpenAI's gym repository.

I implemented a toy version of PPO on a simple OpenAI gym environment to learn the ins and outs of how it works. However, when implementing the PPO agent on these larger Atari environments I choose to use the stable baselines library to do so. I did this because the stable baselines implementation provides numerous optimizations and functionality for multi-processing to greatly speed up training. Since my project was focused on deep transfer learning, I deemed this large training speed up as a worthwhile trade off to lose fine grain control over the agent.

The initial proposal papers for the DQN and PPO agents are cited below.

VI. Conclusion

In conclusion, this report did not prove that transfer learning was useful in Atari environments. This failure was not due to the shortcomings of transfer learning but instead the easily exploitable, but complex, Bank Heist environment. Transfer learning may have been applicable if the environments were different, but time did not permit this to be tested. The agent's acting in Bank Heist did not end up traversing the maze and likely didn't use visual data as they merely paced rapidly back and forth off of the spawn point; making any filters transferred useless.

These results do highlight some shortcomings of many deep RL agents, however. Agents of many types can get trapped in local minima or optimal that they never are able to escape from. In the example of Bank Heist, this happened because the environment was too complex for the agents to easily learn and thus exploited the environment. I would be interested to perform these experiments again with less complex, similar environments to see if my hypothesis for this project would hold.

VII. References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wiestra, D., & Reidmiller, M. (2013). Playing atari with deep reinforcement learning. axXiv preprint arXiv: 1312.5602

[2] Zhuangdi Zhu, Kaixiang Lin, Jiayu Zhou: "Transfer Learning in Deep Reinforcement Learning: A Survey", 2020; [http://arxiv.org/abs/2009.07888 arXiv:2009.07888]

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller: "Playing Atari with Deep Reinforcement Learning", 2013; [http://arxiv.org/abs/1312.5602 arXiv:1312.5602].

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov: "Proximal Policy Optimization Algorithms", 2017; [http://arxiv.org/abs/1707.06347 arXiv:1707.06347].R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[5] Hado van Hasselt, Arthur Guez, David Silver: "Deep Reinforcement Learning with Double Q-learning", 2015; [http://arxiv.org/abs/1509.06461 arXiv:1509.06461]

[6] Akshita Mittel, Sowmya Munukutla, Himanshi Yadav: "Visual Transfer between Atari Games using Competitive Reinforcement Learning", 2018; [http://arxiv.org/abs/1809.00397 arXiv:1809.00397].