Using Transfer Learning Between Games to Improve Deep Reinforcement Learning Performance and Stability

Chaitanya Asawa [*1] **Christopher Elamri** [*1] **David Pan** [*1]

[*]Equal contribution

Abstract

We explore transfer learning in the context of deep reinforcement learning to perform well on different OpenAI Gym games. Specifically, we use Deep Q-learning to play the games Snake and PuckWorld. We want to see if we can use a pre-trained Deep Q-Network from PuckWorld to play Snake, and with some additional training or layers, see if what is learned from one game can be adapted to another. We find that transfer learning not only boosts performance for the game Snake, but also leads to far greater performance stability while learning, potentially showing value of transfer learning for safe reinforcement learning.

1. Introduction

Reinforcement learning (RL) is a paradigm for learning sequential decision making tasks, where an agent seeks to maximize long-term rewards through experience in its environment. During the learning process the agent has to decide whether to look for new information (explore) or to use its current model to maximize reward (exploit). However, while significant progress has been made to improve single task learning, the use of transfer learning has only recently gained momentum. The idea behind transfer learning is that generalization can occur not only within tasks, but also across tasks. In this paper, we investigate the effectiveness of transfer learning across the Snake and PuckWorld game environments.

In particular, we use deep Q-learning, which tries to learn an optimal policy from its history of interaction with the environment. We primarily explore if the Deep Q-Networks (DQNs) that are trained for PuckWorld environments can be used to increase the relative performance of the game Snake. An increase in the relative performance could perhaps indicate the usefulness of transfer learning in reinforcement learning tasks.

The ability to successfully transfer learn has great importance, as it allows us to avoid having to train specific models for every task we may have (which takes much computational power and time), and rather achieve high performance quickly using what we have generally learned.

2. Related Work

Recently, researchers have achieved great success in games using deep reinforcement learning methods.

In their groundbreaking paper "Playing Atari with Deep Reinforcement learning" (Mnih et al., 2013), Mnih et al. developed a single Deep Q-Network (DQN) that is able to play multiple Atari games, in many cases surpassing human expert players. The model consists of convolutional layers followed by fully connected layers. The model takes in the raw image pixels of a game and outputs Q-values estimating future rewards. The model is trained with a variant of Q-learning with a target network and experience replay. The target network helps reduce oscillations or divergence of the policy, and experience replay removes correlation in the observations and prevents the model from getting stuck in bad local minima.

Realizing that being able to perform multiple tasks and use previous learnings is crucial for any intelligent agent, in the paper "Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning" (Parisotto et al., 2015), Parisotto et al. build on the work of Mnih et al. to explore transfer learning between Atari games. They developed the Actor-Mimic method that uses the guidance of many game-specific expert networks to train a single multitask policy network. For transfer learning, they treat the multitask network as a DQN, and they transfer all the weights except the final softmax layer to a new DQN. After transfer learning, the new DQN learns to complete a target task significantly faster than a DQN starting from a random weight initialization.

In our project, we will focus on trying different methods of transferring weights with deep Q-networks and Dueling Network architectures and measure the results.

3. Game Mechanics

3.1. Snake

Snake, the game popularized by Nokia, involves a single player who controls moving the direction of a snake and tries to eat randomly appearing items by running into them. Each item eaten makes the snake longer, and this makes the game progressively more difficult, as the player/agent loses when the snake runs into itself or the screen border. In the version of Snake we are using, there is a -5 reward for either running into the walls or into itself, and a reward of $+1$ for collecting a red square "food" (Sobrecueva, b).

3.2. PuckWorld

PuckWorld is a game in which the agent, a blue circle, must navigate to a green circle while avoiding a large red puck. The large red puck slowly tries to follow our agent, and the green circle randomly changes the location periodically.

The agent has up, down, left, and right thrusters, and its velocity decays over time. At each timestep, our rewards earned are a function of the distance to the green circle (where we want to be closer to the green circle) and negative reward proportional to our distance from the red puck's center if we are within the red puck's radius. The agent's goal is to get close to the green dot while avoiding the red puck.

The game is a continuous game and there are no terminal states (Sobrecueva, a).

3.3. Juxtaposition of Games

In both games, we are trying to move to some sort of target, while generally trying to avoid some regions.

In the case of Snake, once we arrive at the target, we collect the reward and the target moves. In the case of PuckWorld, we want to be close to the reward, but the target moves periodically, not dependent on the agent's actions.

Additionally, in Snake we want to avoid walls and itself, while in PuckWorld we want to avoid the large red puck.

Movement is different as well. While in both games we can move in the same directions, in PuckWorld we have a velocity with our movement due to thrusters (this velocity decays over time) and in Snake we just move one square at a time.

Finally, the snake grows whereas the agent does not change size in PuckWorld. This increases the probability of reaching a terminal state in Snake, but PuckWorld has no terminal states.

We hope that, despite the many differences between the games, we can capture the idea of moving to an agent in both games through transfer learning, and also some sense of avoiding objects – whether that be avoiding the snake itself or the large red puck.

We note that we do not have good baselines for OpenAI games, as they were only released fairly recently and have few if any submissions (the best Snake submission, for example, has best 100-episode performance of -4.12, which is close to the worst possible score).

4. Approach

4.1. Deep Q-Learning

In Q-learning, we are trying to estimate for a given state and action pair, what is the expected discounted sum of future rewards is if we took that action from the state and followed the optimal policy after. Once we have these $Q(s, a)$ values, the action we can take from a given state s is $\text{argmax}_a Q(s, a)$.

When we have an incredibly large state space, however, it is hard with traditional update methods to determine these Q-values. Thus, neural networks have proven to, with training, be a great estimator for these Q-values. We use deep Q-learning convolutional neural network models with ϵ-greedy exploration (Mnih et al., 2013).

ϵ-greedy allows us to explore more states, and prevent getting stuck in exploring the same seemingly optimal path (exploration/exploitation problem). In particular, at each step the algorithm chooses a random action with probability ϵ or picks the best action following the Q estimate with probability $(1 - \epsilon)$. This ϵ decays over time, so initially we are exploring more and trying to understand the environment, but after some time we want to mostly use what we have learned with only a little bit of exploring.

We use experience replay, providing the model with signals throughout its training history, and target networks to avoid oscillations and divergences in policy, as mentioned in section 2.

4.2. Transfer Learning

Transfer learning is a technique in which we use networks that have proven to do well on some task and try to adapt what is learned from this task to a separate but potentially related task. In our case, we would like to use networks that have performed very well at estimating Q-values for one game and try to adapt them, with a little more training, to estimate Q-values for another game.

We want to mainly explore transfer learning from Snake to PuckWorld. Our hypothesis is that because the games have similar objectives of chasing an object, transfer learning will help speed up convergence and perhaps attain better

performing models than before.

We consider combinations of retraining layers and reinitializing layers. Retraining layers involves initializing layers with the weights of a pre-trained model and continuing to update these weights with backpropagation (we may choose to keep some layers fixed). Reinitializing layers involves simply randomly initializing the weights for a layer, rather than using the pre-trained weights. When we reinitialize layers, we are purposely choosing to almost "unlearn" some of what we learned from the previous model because perhaps it is very specific to the previous model and does not apply to our current problem.

5. Technical Details

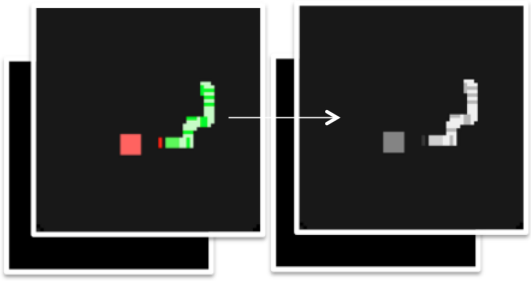

Figure 1. Preprocessing for Snake. We maintain a small state history and additional convert the images from RGB to grayscale to reduce the size of the input.

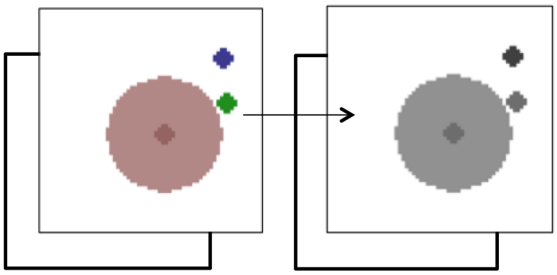

Figure 2. Preprocessing for PuckWorld. We maintain a small state history and additional convert the images from RGB to grayscale to reduce the size of the input.

The Snake and PuckWorld environments from OpenAI Gym return images of size 64 x 64 x 3, which means 64 pixels width, 64 pixels height, and the last dimension corresponding to the RGB channels which have values between 0 and 255. We convert the image to grayscale to reduce the input dimension to 64 x 64 x 1. Also, each time the agent makes an action, we repeat it for some time steps, and we return a pixel-wise max-pooling of these consecutive frames. This allows us to play many times more games while training by reducing the frequency of computing new actions. We also experiment with maintaining a state history, since the state here might not totally capture all information, such as the direction an object is traveling with. This is especially important in a game such as PuckWorld where the agent is moving with thrusters.

Note that, throughout this paper, we say an epoch is 50,000 training iterations.

6. Vanilla DQN Experiments

6.1. DQN Architecture

Our DQNs have an initial input of 64 x 64 x s where s is the size of the state history. After this initial layer, in both DQNs, we apply 3 convolutional layers consisting of 32 filters of size 3 x 3 and stride 1. For both DQNs, our final hidden layer is a fully-connected layer that results in 256 hidden units, and then we have another fully connected layer that outputs a vector of size number of actions $|A|$.

6.2. Snake

Initially, we trained a model for 3 million iterations with a state history of 1 and skip frame of 4. For this experiment, we chose not to maintain a state history because for Snake there is no concept of speed and we theoretically know direction by the snake's head.

We then tried another experiment as well, for 5 million iterations, with a state history of 4 (because perhaps the direction was hard to determine from a grayscale head of the snake), and not skipping frames since for snake each frame is important (difference between reward as well as life and death). Figure 3 shows the results of these two experiments – in particular, the larger state history did not help and actually made it harder for the snake to learn, and it achieves almost consistently -5 reward across 5 million iterations. Hence, we will focus our analysis on experiment with 3 million iterations, a state history of 4, and skip frame of 4.

After training in the first experiment, we achieved a final average reward of 14.94 +/- 1.01. However, at an earlier point in training, we achieved an average reward of 21.68 +/- 1.15. Figure 3 shows the change in average evaluation reward over our training epochs.

The performance over training time as well as actually watching the algorithm behave reveals some interesting insights:

- Initially, for the first 15 epochs or so, our average reward remains around -5 (meaning no points were collected). This seems to indicate at this point the snake is just crashing into walls, and has not yet learned to characterize and avoid such states. The snake is not

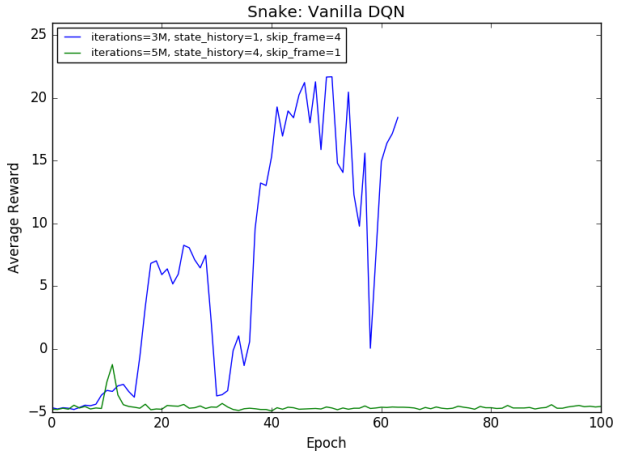

Figure 3. Average evaluation reward of the Snake agent on 50 test episodes over training epochs with vanilla DQNs.

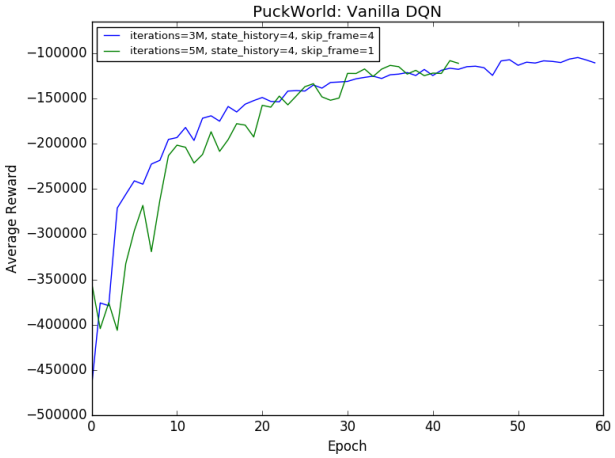

Figure 4. Average evaluation reward of the PuckWorld agent on 50 test episodes over training epochs with vanilla DQNs. (Note that the second experiment was cut short of 5M iterations because of resource constraints and 4x training time due to skip_frame=1.)

long enough, with no points collected, to run into itself.

- However, after the 15th epoch, we see a quick dramatic improvement in the performance (to an average of +5), as we can now substantially delay the terminal state of a wall.
- At the 30th epoch though, performance dips back down to negative reward. It rises significantly starting from the 35th epoch, but then we see a huge decline at the 58th epoch again. Near the last few epochs though, we are once again on the rise.

We observe some sort of oscillation in Snake performance. From these initial experiments with vanilla DQNs, we hypothesize that this is either caused by difficulty generalizing with both a longer snake and the snake crashing into itself, or in the mechanics of the learning process of the Deep Q-Network.

6.3. PuckWorld

First, we trained the model for PuckWorld with 5 million iterations, state history of 4, and skip frame of 4. The agent learned to follow the green dot. As shown in Figure 4, the agent continuously improves at the game with little oscillation. We made the following observations:

- The agent oscillates significantly around the green dot instead of staying on it. We think the reason is that we apply the same action for four frames in a row, but the agent can travel a great distance in four frames.
- The agent does not attempt to avoid the red puck. We noticed in the OpenAI Gym implementation of PuckWorld, the reward contributed by the red puck ranges from -2 to 0 while the reward contributed by the green dot ranges from $-64\sqrt{2}$ to 0. Because the agent oscillates so much, the reward contributed by the red puck is relatively negligible.
- When the green dot is close to the red puck, the agent will sometimes appear to oscillate around the red puck instead of green dot. We think the reason is that after the grayscale preprocessing, the green dot and the red puck only differ by 1 grayscale value (the values range from 0 to 255), so the agent gets confused between the green dot and red puck.

Second, we trained the model for PuckWorld with 5 million iterations, state history of 4, and skip frame of 1. The agent was able to land on the green dot with little oscillation because skip frame is now 1 instead of 4. The agent appeared to avoid the red puck sometimes, but most of the time it just follows the green dot even in the vicinity of the red puck. Like the previous agent, the agent continuously improves at the game with little oscillation, and we see very similar results.

7. Dueling Network Experiments

We then experimented in seeing if we could both increase performance and reduce oscillation using a Dueling Network architecture instead of vanilla DQNs. The rationale for this was that the Dueling Network architecture was shown to be successful for a wide variety of Atari games, but also because it separates how valuable a state is with what the advantage of different actions are from a state by using different streams (Wang et al., 2015). We thought this separation could help better capture the idea of grow-

ing longer is valuable, and taking actions to avoid one's self and collect rewards are advantageous – perhaps helping with the oscillation. Figure 5 shows the Dueling Network architecture.

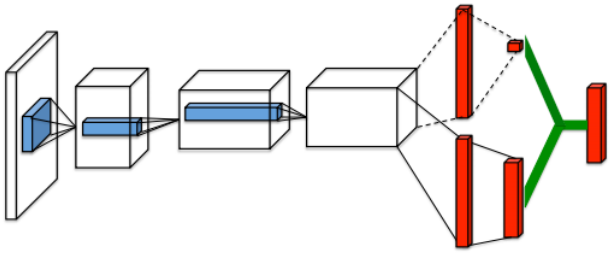

Figure 5. Dueling Network Architecture (Wang et al., 2015)

We show below the results of the Dueling Network architecture for our two games, PuckWorld and Snake. We use a state history of 2 and skip every 4 frames in both games.

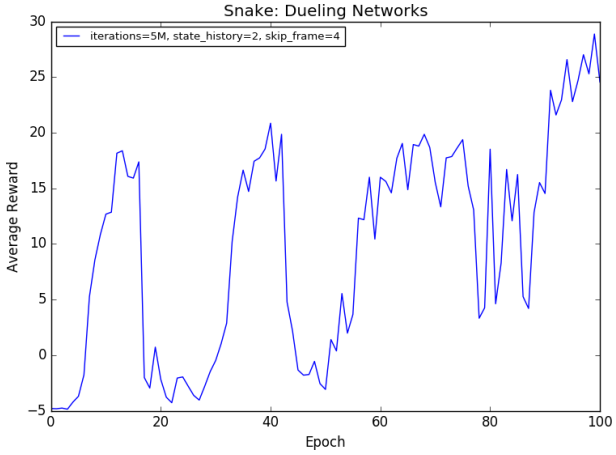

Figure 6. Average evaluation reward of the Snake agent on 50 test episodes over training epochs with a Dueling Network architecture.

We find that we have increased performance for Snake and we reach this quicker as well – on the 16th epoch we have a reward of around +17 with the Dueling Network architecture, and we do not even have positive reward with vanilla DQNs at this point. However, we find that the oscillations are still maintained.

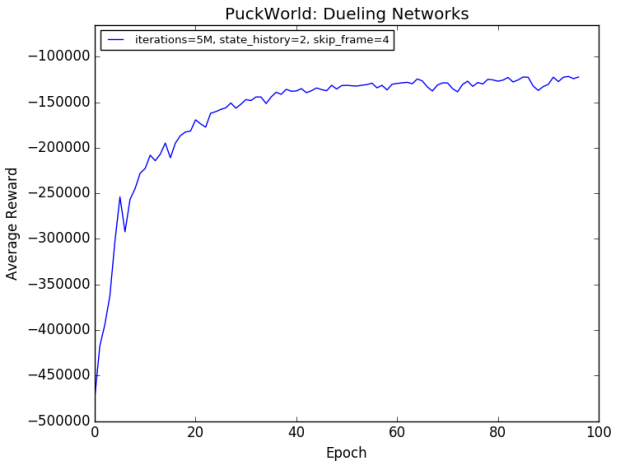

Figure 7. Average evaluation reward of the PuckWorld agent on 50 test episodes over training epochs with a Dueling Network architecture.

The PuckWorld performance, for the most part, seems very similar to that achieved by vanilla DQNs (converges to same value, if not only slightly worse).

8. Oscillations

We then tried to better understand why we saw these oscillations in performance for Snake – in the real world, we would never want to deploy a system which we could not trust to have reliable performance across training. We look at the behavior of the models when performance drastically drops after peaks.

With Vanilla DQNs, we find that the Snake, close to a wall, simply moves in a very tight circular fashion, with the target being far away.

With the Dueling Network architecture, we don't see the circling behavior, but instead the snake almost hugs the wall and moves back and forth.

After seeing both how the Vanilla DQNs and the Dueling Network architecture perform over time, we infer that the agent, as it grows longer, needs to solve a new problem: initially, it needed to learn to avoid walls and go to targets, but now, however, it has grown long enough that the reason we reach a terminal state is because of the snake running into itself. As it tries to update to avoid this terminal state, with the twist and turning behavior we do see that it learns, we believe the updates override past learnings that help it find and move towards the target.

We provide videos to visualize this behavior here: `https://goo.gl/M0aBgu`

9. Transfer Learning Experiments

As PuckWorld seems fairly stable, but Snake is not, we wanted to see if transfer learning from PuckWorld to Snake could both somehow reduce oscillation and improve the average reward for Snake.

We explored various ways of transfer learning from PuckWorld to Snake. After transferring all the learned weights from PuckWorld, we chose which layers to reinitialize and which layers to retrain.

At first, we tried retraining all layers without reinitializing any layers, but we found that the reward just stays around -5. The reason was that the PuckWorld weights, in particular those with the affine layers, were very large because PuckWorld has a reward function with large magnitude. Hence, these weights were on a very different scale, and due to gradient clipping in the update step, the weights updated stayed this way.

Next, we tried the following: retraining the affine layers, reinitializing and retraining the affine layers, and reinitializing and retraining the last convolutional layer (conv3) and the affine layers. We found that the reward barely improves. (Figure 8. Note that training for the "Reset & train affine layers" model was cut short due to resource constraints. We conclude that this is fine because its reward over epoch should be approximately bounded by the reward over epoch for the "Reset & train conv3, affine layers" model.) Overall, this shows that the convolutional layers from PuckWorld are not directly applicable to Snake and needs more tuning before achieving success.

Finally, we tried reinitializing the affine layers and retraining all layers, and reinitializing the last convolutional layer and the affine layers and retraining all layers. We found that reward over epoch increases approximately linearly with smaller oscillations compared to the dueling network without transfer learning. In addition, the final reward was higher than that of the dueling network without transfer learning (Figure 9). These results validate our hypothesis that transfer learning from PuckWorld reduces oscillation and improves average reward.

Note that we started reinitializing layers from the top of the network (closest to the output) instead of from the bottom layers (closest to the input). The reasoning is that top layers tend to have specific features that are usually only applicable to the game trained on, while bottom layers have more general features that could be applicable to other games. In addition, the bottom layers are convolutional layers which extract visual features, so they are important to keep.

We hypothesize that transfer learning from PuckWorld helps to both reduce oscillation and improve average reward in Snake because the visual features in the convolutional layers of the PuckWorld model have deeply ingrained the concept of going towards an object. Without transfer learning, the snake learns to avoid walls and go towards the red dot, but as the snake gets longer, it gets in an unfamiliar state. It has to learn to avoid running into its longer body, and it may have to relearn going towards the red dot because the model may be unsure what to do in this unfamiliar state where the snake is longer (as we discussed in Section 8). With transfer learning, the snake is conditioned to go towards the red dot no matter what its length is, therefore bypassing this dilemma and reducing oscillation.

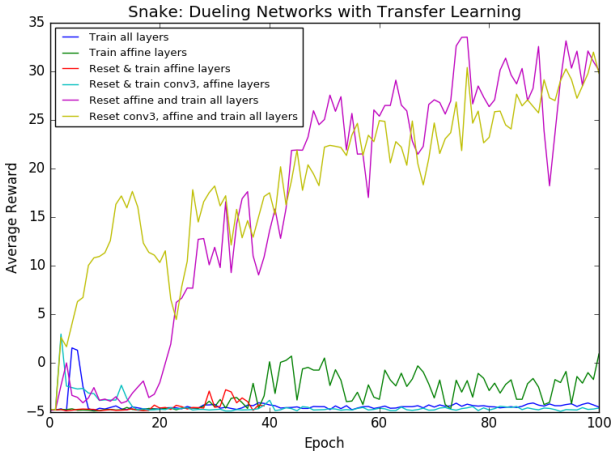

Figure 8. Average evaluation reward of the Snake agent on 50 test episodes over training epochs with different variants of transfer learning.

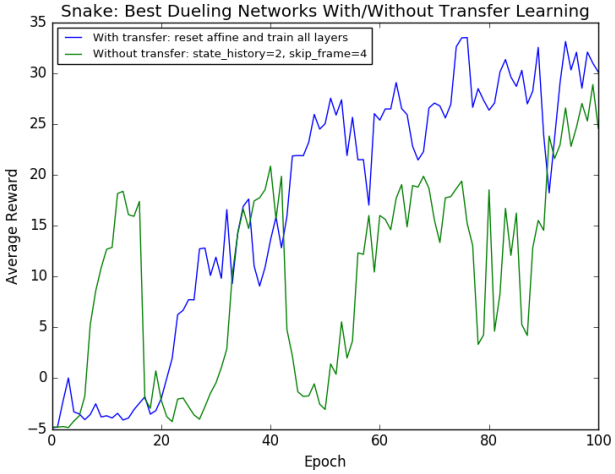

Figure 9. Average evaluation reward of the Snake Dueling Network agent on 50 test episodes over training epochs, with and without transfer learning.

9.1. Transferring from Snake to PuckWorld

Finally, we were curious to see, despite stable performance of PuckWorld, whether transferring the learned weights in Snake with a Dueling Network architecture could improve PuckWorld in terms of either convergence time or performance. We used the most successful transfer learning strategy we found above; retraining the entire network, but reinitializing the final affine layers. We show the results below:

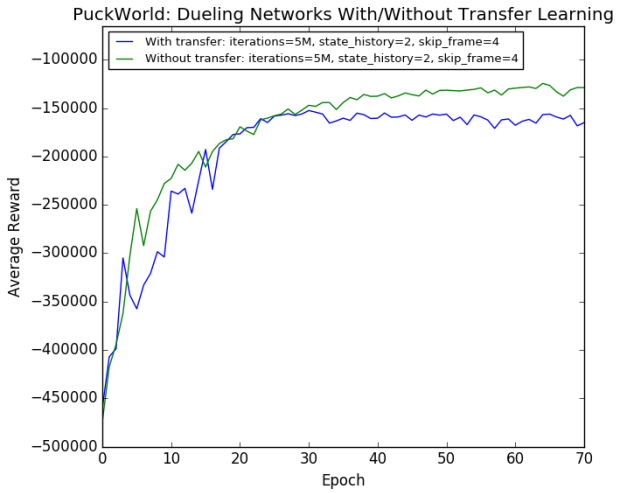

Figure 10. Average evaluation reward of the PuckWorld Dueling Network agent on 50 test episodes over training epochs, with and without transfer learning.

It seems that the performances with and without transfer learning are roughly the same – if anything, the transfer learning performance is slightly worse.

We noticed however that a decent number of gradients for PuckWorld are getting clipped – this is most likely due to having a reward function that has a high magnitude. It is possible we can see improved performance if we set a much higher threshold for gradient clipping (both in previous experiments and this one).

10. Conclusion

We find that we were able to successfully use transfer learning to improve performance and stability in using deep Q-learning to play the Snake game. Specifically, we found best results when we retrained the entire weights of a previously trained Dueling Network model and then reinitialized the layers at the end. We are excited about these results, since they not only show that transfer learning can improve deep reinforcement learning performance, but also that transfer has implications in Safe Reinforcement Learning (which, to our knowledge, is the first of its kind with transfer learning) – as of course, we would not want to trust an unstable agent in the real world.

11. Future Work

In the future, we are interested in seeing if we can successfully transfer learn weights to other games to improve performance and stability – this would more definitively show that transfer learning is viable more broadly in deep reinforcement learning in the context of games.

Specifically for the models we explored, one way to improve performance that could be explored is using the full RGB image as the input instead of the preprocessed grayscale image. The reason is that in PuckWorld, grayscale preprocessing makes it difficult to distinguish between the green dot and the red puck, and in many instances it appeared that the agent started following the red puck instead of the green dot. We would also like to allow the gradients for PuckWorld to be larger, clipping at a more appropriate scale for the game.

While there is still much left to explore in terms of our deep Q-network architectures to optimize performance for these games, ultimately it may be more interesting to focus on whether we successfully apply transfer learning to improve the relative performance and stability of the algorithms across a variety of applications that are not just limited to games.

Acknowledgements

Thank you to Barak Oshri for being our project mentor, and also the CS 234 DQN assignment.

References

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

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

Sobrecueva, Luis. PuckWorld-v0 (experimental) (by @lusob) - OpenAI Gym, a. URL https://gym.openai.com/envs/PuckWorld-v0.

Sobrecueva, Luis. Snake-v0 (experimental) (by @lusob) - OpenAI Gym, b. URL https://gym.openai.com/envs/Snake-v0.

Wang, Ziyu, Schaul, Tom, Hessel, Matteo, van Hasselt, Hado, Lanctot, Marc, and de Freitas, Nando. Dueling

network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

12. Supplemental Material

(Also in submitted zip file.)

We wanted to show how, over time, our Snake is able to learn various aspects of the game to perform better (such as avoiding walls, collecting target, and avoiding itself). Please check out the following video of our final Snake agent's learning process (using transfer learning and Dueling Networks):

https://www.youtube.com/watch?v=-7UZHERm2bA

Our source code is here:

https://drive.google.com/file/d/0B4pM4ssJKA2MTkltVmFzemZPRzQ/view?usp=sharing

Contributions

All three members contributed to the analysis and contributed different sections of the code; for example, among other things, David did most of the transfer learning code, Chaitanya did most of the DQN architecture code, and Chris worked on the general infrastructure and visualizations.