Atari Skiing: Applying DQN, DDQN, and A2C

Andrew Pozzuoli - 6017735

1. Introduction

Skiing for the Atari 2600 is one of the reinforcement learning environments in the OpenAI Gym library (Brockman et al., 2016). The task is to get a pink skiing agent to the bottom of the hill, marked by two red flags, in the shortest time possible. Along the way there are 20 gates marked in blue flags. If the agent misses a gate then 5 seconds are added to the elapsed time. If the agent runs into a flag or one of the many trees dotting the trail, then they crash resulting in a time loss.

The state space is comprised of the RGB pixel information of the current screen. This is represented as a three dimensional array 210 pixels wide by 160 pixels in height, each pixel having 3 colour values for red, green, and blue colour information.

The action space has three discrete values of left, right, and no action. Left and right turn the tips of skis towards the direction specified and no action keeps the skier moving downward. These three actions result in eight different ski positions which vary how far the agent is turned. Hard left and hard right result in the agent not moving at all as they are perpendicular to the direction of the hill. Straight down results in the fastest descent but turning is still necessary to avoid obstacles and pass through gates.

The reward signal is a negative integer returned once the agent has reached the bottom of the hill. This signal is measured as the negative number of centiseconds that elapsed. For example, a time of 32.72 seconds results in a reward of -3272. This allows for reward maximization as a number closer to zero will be a better time for the agent.

This task is notoriously difficult among the OpenAI Gym Atari environments due to the sparsity of the reward signal. Since the agent only receives a reward once it has reached the bottom, it is difficult to learn which specific actions along a trajectory had any effect on the reward signal.

This project compares the performance of three different reinforcement learning algorithms (DQN, DDQN+PER, and A2C) in their ability to learn how to effectively tackle this task.

2. Methods

The algorithms compared in this project were *Deep Q-Network* (DQN), *Double DQN with Prioritized Experience Replay* (DDQN+PER), and *Advantage Actor-Critic* (A2C). These specific algorithms were chosen since they gave the top three best performances according to the Atari Benchmark Leaderboard located on SLM lab's Gitbook (Keng & Graesser, 2017).

2.1. Deep Q-Network N

DQN is an off-policy TD algorithm that approximates the Q-function (Mnih et al., 2013). It calculates the Q-function by using the maximum Q-value for the next state. The equation for approximating this is given in equation 1 This results in DQN learning the optimal Q-function.

$$y = r + \gamma max_{a'} Q^{\pi_\theta}(x', a') \tag{1}$$

DQN also uses an experience replay memory which uses batches of experiences that are randomly sampled for training, improving sample efficiency. The memory is often much larger than the batch size allowing experiences to be sampled more than once. When the memory is full, the oldest one is discarded (Keng & Graesser, 2019).

The pseudocode for DQN can be seen in algorithm 1.

2.2. Double DQN with Prioritized Experience Replay

DDQN is a method that improves upon DQN by modifying its learning techniques. One modification is the use of a *target network*. In the original DQN algorithm, Qpi_{tar} changes constantly between training steps. Due to the constant changing, it is difficult to adjust the parameters to minimize the difference between $Qpi_\theta(s, a)$ and Qpi_{tar}. DDQN uses a target network which is a copy of the Q-network with lagged parameter updates. This lagged network is used to calculate Qpi_{tar}. This helps to stabilize the training (Keng & Graesser, 2019).

The next modification is *double estimation* which addresses DQN's issue of overestimating Q-values. DDQN uses two Q-function estimates with two different networks. The first network selects the action which maximizes Q then the second network uses the action selected by the first network

Algorithm 1 DQN Algorithm (Keng & Graesser, 2019)

Input: initial learning rate α, initial Boltzmann policy τ, number of batches per training step B, updates per batch U, batch size N, experience replay memory size K
Initialize network parameters θ randomly.
for $m = 1$ **to** MAX_STEPS **do**
  Gather and store h experiences (s_i, a_i, r_i, s'_i) using current policy
  for $b = 1$ **to** B **do**
    Sample batch of b experiences from experience replay memory
    for $u = 1$ **to** U **do**
      for $i = 1$ **to** N **do**
        $y_i = r_i + \delta_{s'_i} \gamma max_{a'_i} Q^{\pi_\theta}(s'_i, a'_a)$ where $\delta_{s_i} =$ 0 if s'_i is terminal, 1 otherwise
      end for
      $L(\theta) = \frac{1}{N} \Sigma_i (y_i - Q^{\pi_\theta}(s_i, a_i))^2$
      $\theta = \theta - \alpha \nabla_\theta L(\theta)$
    end for
  end for
  Decay τ
end for

Algorithm 2 Double DQN + PER Algorithm (Keng & Graesser, 2019)

Input: initial learning rate α, initial Boltzmann policy τ, number of batches per training step B, updates per batch U, batch size N, experience replay memory size K, target update frequency F, maximum priority P
Initialize network parameters θ randomly.
Initialize target network parameters $\varphi = \theta$
for $m = 1$ **to** MAX_STEPS **do**
  Gather and store h experiences $(s_i, a_i, r_i, s'_i, p_i)$ using current policy where $p_i = P$
  for $b = 1$ **to** B **do**
    Sample a prioritized batch of b experiences from experience replay memory
    for $u = 1$ **to** U **do**
      for $i = 1$ **to** N **do**
        $y_i = r_i + \delta_{s'_i} \gamma Q^{\pi_\varphi}(s'_i, max_{a'_i} Q^{\pi_\theta}(s'_i, a'_i))$ where $\delta_{s_i} = 0$ if s'_i is terminal, 1 otherwise
        $\omega_i = |y_i - Q^{\pi_\theta}(s_i, a_i)|$
      end for
      $L(\theta) = \frac{1}{N} \Sigma_i (y_i - Q^{\pi_\theta}(s_i, a_i))^2$
      $\theta = \theta - \alpha \nabla_\theta L(\theta)$
      $p_i = \frac{(|\omega_i| + \varepsilon)^\eta}{\Sigma_j (|\omega_j| + \varepsilon)^\eta}$
    end for
  end for
  Decay τ
  if $(m \ mod \ F) == 0$ **then**
    $\varphi = \theta$
  end if
end for

to calculate $Q^p i_{tar}(s, a)$. Using the second estimate which trains on different experiences removes the overestimation (Keng & Graesser, 2019).

The third modification is *prioritized experience replay* (PER) which alters the random uniform sampling of the memory of DQN. PER comes from the intuitive understanding that not all experiences are as valuable to an agent's learning as others. Each experience is measured in terms of the absolute TD error between $\hat{Q}^p i(s, a)$ and $Q^p i_{tar}$. Experiences are then prioritized by this measure which improves sample efficiency (Keng & Graesser, 2019).

The pseudocode for DDQN + PER can be seen in algorithm 2.

2.3. Advantage Actor-Critic

Advantage Actor-Critic (A2C) is an on-policy algorithm with two main components – an *actor* which uses a parameterized policy and a *critic* which learns a value function to assign to state-action pairs. The main idea behind this algorithm is to learn a dense reinforcing signal when the environment's signals are more sparse (Keng & Graesser, 2019).

The reinforcing signal is learned via an *advantage function* which selects actions based on their relative performance to actions available in that state. Equation 2 shows how the reinforcing signal is calculated.

$$A^\pi(s, a) = Q^\pi - V^p i(s) \tag{2}$$

The advantage function measures how much better or worse an action is than the average action in a given state (Keng & Graesser, 2019).

In order to calculate the advantage, V^π is estimated using *n-step* returns. With this we use a trajectory of returns for n-steps $(r_1, ..., r_n)$ and then use $\hat{V}^\pi(s)$ learned by the critic. Equation 3 demonstrates this.

The pseudocode for A2C can be seen in algorithm 3.

$$A^\pi_{NSTEP} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + \gamma^{n+1} \hat{V}^\pi(s_{t+n+1}) + \hat{V}(s_t) \tag{3}$$

All agents were run using SLM lab's atari benchmark spec file (Keng & Graesser, 2017) with slight modifications. For DQN, the memory's max size was reduced to 1000 due to hardware constraints. A2C's n-step size was increased

Algorithm 3 A2C Algorithm (Keng & Graesser, 2019)

Input: initial actor learning rate α_A, initial critic learning rate α_C, entropy regularization weight β
Initialize actor and critic network parameters θ_A and θ_C randomly.
for $episode = 0$ **to** $MAX\_EPISODE$ **do**
  Gather and store data (s_t, a_t, r_t, s'_t) by acting in environment under the current policy
  for $t = 0$ **to** T **do**
    Calculate predicted V-value $\hat{V}^{\pi}(s_t)$ using the critic network θ_C
    Calculate the advantage $\hat{A}^{p}i(s_t, a_t)$ using the critic network θ_C
    Calculate $V^{\pi}_{tar}(s_t)$ using the critic network θ_C and/or trajectory data
    Optionally, calculate entropy H_t of the policy distribution, using the actor network θ_A, otherwise $\beta = 0$
  end for
  Calculate value loss: $L_{val}(\theta_C) = \frac{1}{T}\Sigma_{t=0}^{T}(\hat{V}^{p}i(s_t) - V^{\pi}_{tar}(st))^2$
  Calculate policy loss: $L_{pol}(\theta_A) = \frac{1}{T}\Sigma_{t=0}^{T}(-\hat{A}^{\pi}(s_t, a_t)log\pi_{\theta_A}(a_t|s_t) - \beta H_t)$
  Update critic parameters: $\theta_C = \theta_C + \alpha_C \nabla_{\theta_C} L_{val}(\theta_C)$
  Update actor parameters: $\theta_A = \theta_A + \alpha_A \nabla_{\theta_A} L_{pol}(\theta_A)$
end for

to 100 to try to improve the advantage function due to the sparse reward signal of the skiing environment. Another change to A2C was to include more entropy especially earlier on to encourage exploration. A linear-decay beginning at 1.0 and settling at 0.5 after 100000 steps was used.

3. Results

Consistent with the Atari Benchmark Leaderboard (Keng & Graesser, 2017), DDQN+PER performed the best out of the selected methods, settling into an average return of -16000 (160 seconds) early on in training. Figure 1 shows how the agent performed over one million frames and figure 2 shows the agent's performance compared to a random policy. The graph shows that the agent started with an average return of -15000 and suffered performance collapse to -16000 early on in the run. However, it still performed better than a random policy and was the only method of the three tested to do so.

A2C performed the second best on average but with high variation across sessions. On average, the performance began at -30000 (300 seconds) which is the maximum time an agent can spend before restarting at the top of the hill and managed to improve to an average score of -25000. Figure 3 shows the performance over one million frames of training and figure 4 shows how it fares against a random policy. Despite A2C having been the only method to improve during training, it still performed much worse than a random policy on average.

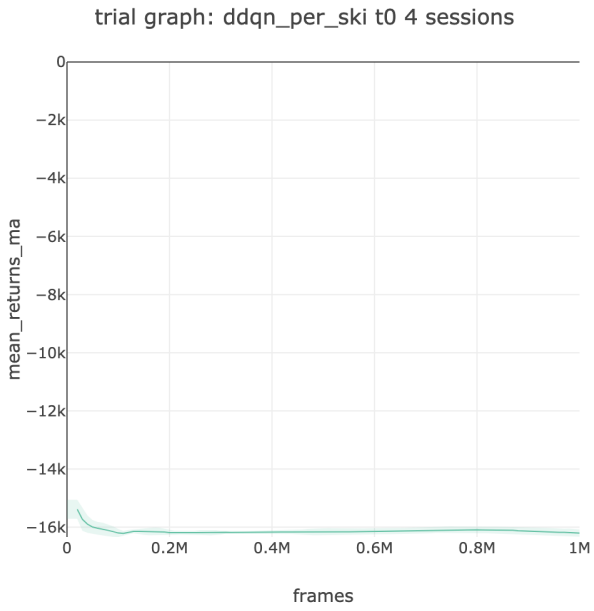

Figure 1. Agent returns over 1 million frames of training under DDQN+PER averaged over 4 sessions (moving average).

Table 1. Atari Benchmark Leaderboard Skiing

ENVIRONMENT	DQN	DDQN	A2C	BEST HUMAN
SKIING	-14094	**-12883**	-14234	-3272

DQN performed the worst, having suffered heavy performance collapse very early in its run. It began with an average return of -15500 (155 seconds) before falling sharply towards -30000 (300 seconds), the maximum time an agent can spend in the environment. Figure 5 shows the returns over one million frames and figure 6 shows the performance compared to a random policy.

For comparison, table 1 includes the benchmark from SLM lab's gitbook (Keng & Graesser, 2017) including the best human player as recorded by Twin Galaxies (Galaxies).

4. Discussion

Both DQN and DDQN suffered from performance collapse with DQN having a more severe collapse than DDQN. It is possible that changing the memory size may have caused DQN to become sample inefficient, leading to the collapse seen. Once the agent begins to perform worse, which can happen at anytime since the reward signal is so sparse that it

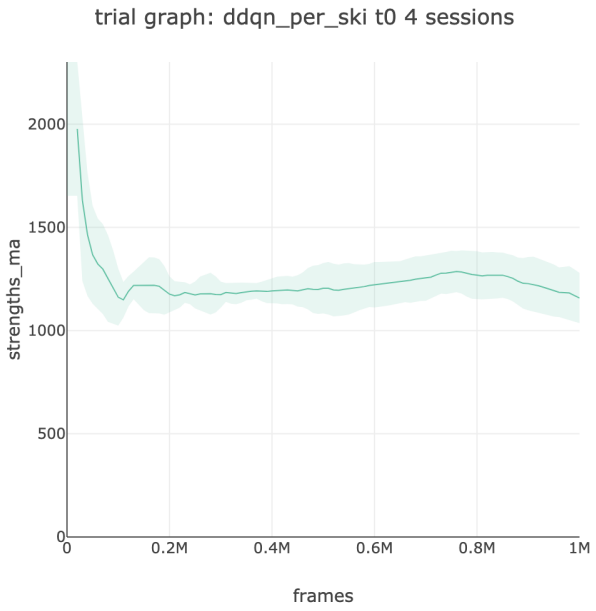

Figure 2. Comparison of agent vs random policy returns over 1 million frames of training under DDQN+PER averaged over 4 sessions (moving average).

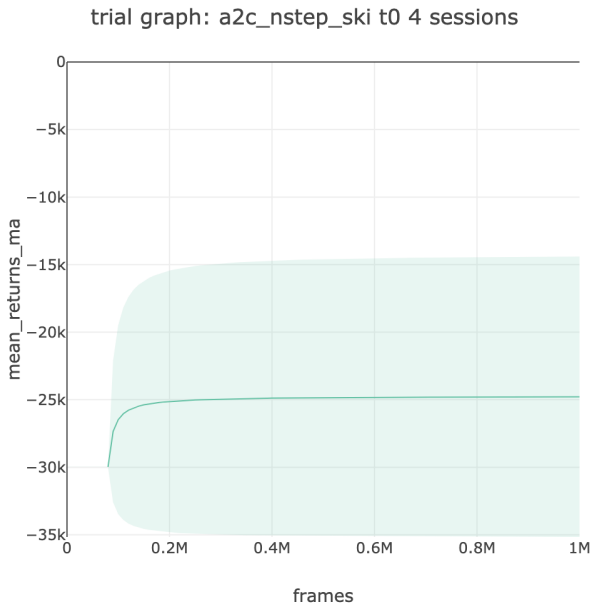

Figure 3. Agent returns over 1 million frames of training under A2C averaged over 4 sessions (moving average).

becomes hard to determine which actions are meaningful, a feedback loop of performance loss perpetuates toward total failure of agent learning. With DDQN, a prioritized experience replay may have remedied this collapse over DQN since it can isolate trajectories that may have contributed to better rewards early on. It is also possible that training two networks to curb overestimation of Q-values led to better performance.

A2C had many more problems than DQN and DDQN in terms of training. While it did manage to improve over time and perform better than DQN on average, it is important to note that this performance was highly variable across the four sessions run. The skiing environment only gives out its reward at the end of the run when the agent has made it to the bottom of the hill unless it times out. Until then, there is no reward signal and what ends up happening is that after n-steps of zero reward, the agent finds no difference between selecting any of the actions and simply chooses one until the end of the run. At best, the agent selects no action and moves straight down the hill, occasionally running into flags or trees but eventually reaching the bottom in about 160 seconds. At worst, the agent selects left or right and gets stuck on the side of the screen with no downward movement until it times out. This was found to be especially troubling in preliminary runs with the default parameters supplied by SLM lab. In an attempt to fix this, n was increased to 100. The thinking being that, in 100 steps, if the agent happened to pick no action and happened to make it to the bottom of the hill, the reward would be in the trajectory and used to calculate a better advantage. Another change was to raise the entropy to encourage exploration so that even if the agent found no difference between actions, the higher entropy would allow it two at least randomly make it to the bottom of the hill rather than get stuck on the side. These changes at least allowed the agent to learn and improve whereas in preliminary runs, the agent would settle into one return and never change over the entire run.

5. Conclusion

As demonstrated by the results of the run, and consistent with the Atari Benchmark Leaderboard, DDQN + PER still performs the best compared to A2C and DQN. Both A2C and DQN performed worse than a random policy on average. DQN and DDQN suffered performance collapse early in their respective runs. A2C was the only method to improve over the course of its run however this was subject to high variance across sessions.

Future work would be to run the experiments for the benchmark's ten million frames rather than only for one million frames but this was cut down for this project due to hardware limitations. Atari environments are highly complex problems that need long training times in order to be somewhat proficient at the game. Another area to explore is parameter optimization. The spec files for these benchmarks are

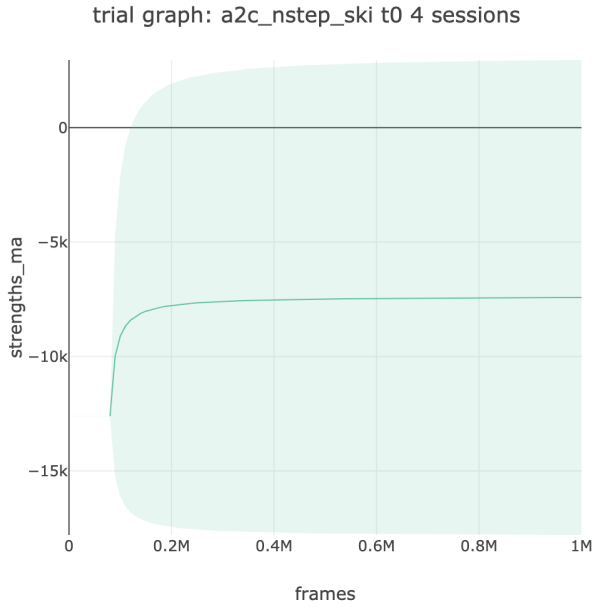

Figure 4. Comparison of agent vs random policy returns over 1 million frames of training under A2C averaged over 4 sessions (moving average).

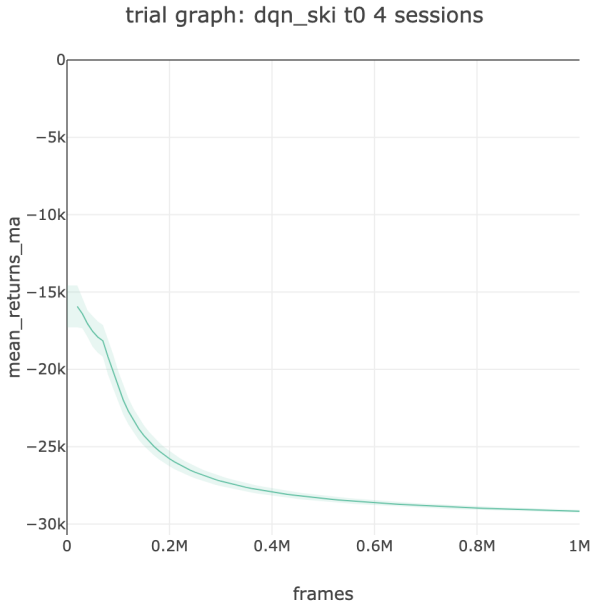

Figure 5. Agent returns over 1 million frames of training under DQN averaged over 4 sessions (moving average).

enormous with many different parameters to be optimized. Additionally, only one set of parameters is used for all Atari environments whether or not those parameters would be better or worse in each specific environment. Skiing is a very different game when compared to some of the other Atari games in terms of its sparse reward signal and finding a good set of parameters would help it to learn well.

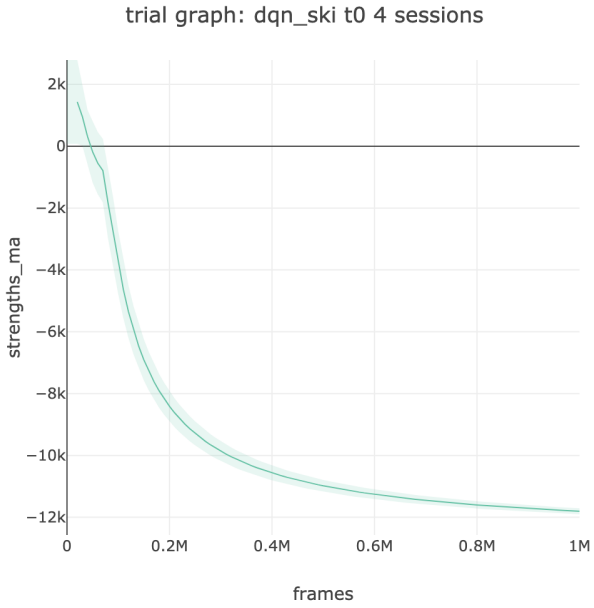

Figure 6. Comparison of agent vs random policy returns over 1 million frames of training under DQN averaged over 4 sessions (moving average).

References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Galaxies, T. Twin galaxies, atari 2600/vcs skiing. URL https://www.twingalaxies.com/game/skiing/atari-2600-vcs.

Keng, W. L. and Graesser, L. Slm lab. https://github.com/kengz/SLM-Lab, 2017.

Keng, W. L. and Graesser, L. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Professional, 2019.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.