COMPARING DEEP Q-LEARNING, ASYNCHRONOUS ADVANTAGE ACTOR CRITIC, AND DEEP DETERMINISTIC POLICY GRADIENT

Khoi Nguyen, Durga Harish Dayapule, Matthew Meyn

ABSTRACT

With the advancement of Deep Learning, Reinforcement Learning has been given a new tool to solve more difficult problems. The input is not only the signal from the sensor but also a raw pixel format can be used directly without any pre-processing step. That gives the agent power to handle real-world problems with larger state spaces. In this report, we will demonstrate and compare the ability of the three most famous algorithms in Deep Reinforcement Learning up to now: Deep Q-Learning (DQN), Asynchronous Advantage Actor critic (A3C), and Deep Deterministic Policy Gradient (DDPG) on some several games. These games are various in terms of input (sensor signal or raw pixel) and output (discrete or continuous). We observed DQN and A3C had similar results in Flappy Bird (avg. Score: Approx. 264) and BiPedal Walker (Avg. Score: Approx. 185), DQN performed best in Lunar Lander (Avg. Score: 51.57) and Bipedal Walker: Hardcore (Avg. Score: 72.13), although A3C had faster training time.

1 INTRODUCTION

Asynchronous Advantage Actor critic (A3C) and Deep Deterministic Policy Gradient (DDPG) are two key deep reinforcement learning algorithms for discrete action spaces and continuous action space respectively. We compare them against a baseline of Deep Q-Learning (DQN) in the domain of three simple games: Flappy Bird, Lunar Lander, Bipedal Walker, and Bipedal Walker: Hardcore.

1.1 DEEP Q-LEARNING

DQN was introduced by DeepMind in 2014 Mnih et al. (2013). The paper was motivated by advancements in deep learning, which made it easy to extract high-level features from images. The three major changes compared to online-Q learning : (1) Convolution Layers: convolutional layers allow us to consider a whole image of current frame as input rather than the signal of agent's state, and extract spatial relationships between the objects on the current frame as we send information up to higher levels of the network. (2) Experience Replay: because learning directly from most recent samples has high variance, it makes the learning inefficient, due to the strong correlations between the consecutive sample data. Instead, randomizing the samples breaks these correlations and therefore reduces the variance of the parameter updates. (3) Separate Target Network: At every step of training the Q-Network gets updated (i.e) the value are updated. By constantly shifting set of values to adjust our network values, the value estimations can easily spiral out of control. In order to mitigate such risks, the target networks weights are fixed, and are updated periodically. So in general, the DQN makes the learning process more efficient by reducing the variance when training. Figure 1 shows the DQN architecture.

1.2 ASYNCHRONOUS ADVANTAGE ACTOR CRITIC

A3C, or Asynchronous Advantage Actor-Critic, was introduced last year (also by DeepMind, now part of Google)Mnih et al. (2016). Integrating aspects of both Q-learning and policy gradient, A3C uses multiple copies of an agent (worker agents) to asynchronously update the master agent's parameters for both speeding up, and making more effective, the process of learning the parameters of

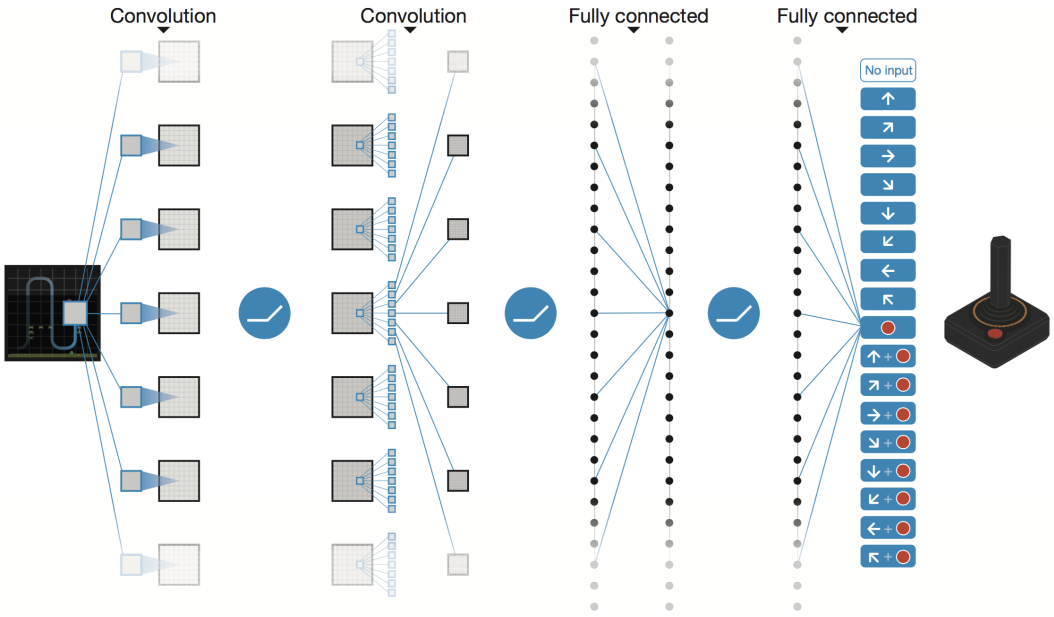

Figure 1: DQN Architecture

the problem's state space and reward function. It uses a Critic module to determine the actual advantage of an agent's actions over what was expected. A3C offers excellent scaling with the number of threads used. The Figure 2 illustrates the architecture of A3C model.

1.3 Deep Deterministic Policy Gradient

While Deep Reinforcement Learning algorithms like A3C are powerful methods for discrete action spaces, they are not suitable for continuous action spaces. Deep Deterministic Policy Gradient (DDPG)Lillicrap et al. (2015) addresses this space. Like A3C, DDPG uses an actor and critic , but with different architecture from A3C. With A3C the actor and critic are two output of the same network, but in DDPG, they are two different networks with different update mechanisms. The actor network estimates the policy to produce the best action from a set of continuous parameters, then the critic estimates the Q value of the current state and the action given by the actor to estimate the loss with the current reward. Finally, the algorithm updates the actor network based on the given loss propagated from the critic network. To know more about the DDPG's architecture, you can see Figure 3

2 Related work

Mnih and the rest of the DeepMind team demostrated the first successful deep-Q-learning approach, Deep Q-Network (DQN), which outperformed existing methods in six out of seven games tested. Mnih et al. (2013) This seems to have been a transformational development, as it solved the problem of stability when approximating Q functions with neural networks. It was also remarkable for being applicable to a wide variety of games, using only the games' visual output as percept data.

In 2014, Silver et al. introduced Deep Deteministic Policy Gradient Lillicrap et al. (2015), which extends the power of deep Q-learning to domains with continuous action spaces.

In 2016, Mnih et al. published a new improvement for deep Q-learning, A3C Mnih et al. (2016). Remarkably, this algorithm could run on a multi-core CPU, not even requiring a GPU. Babaeizadeh

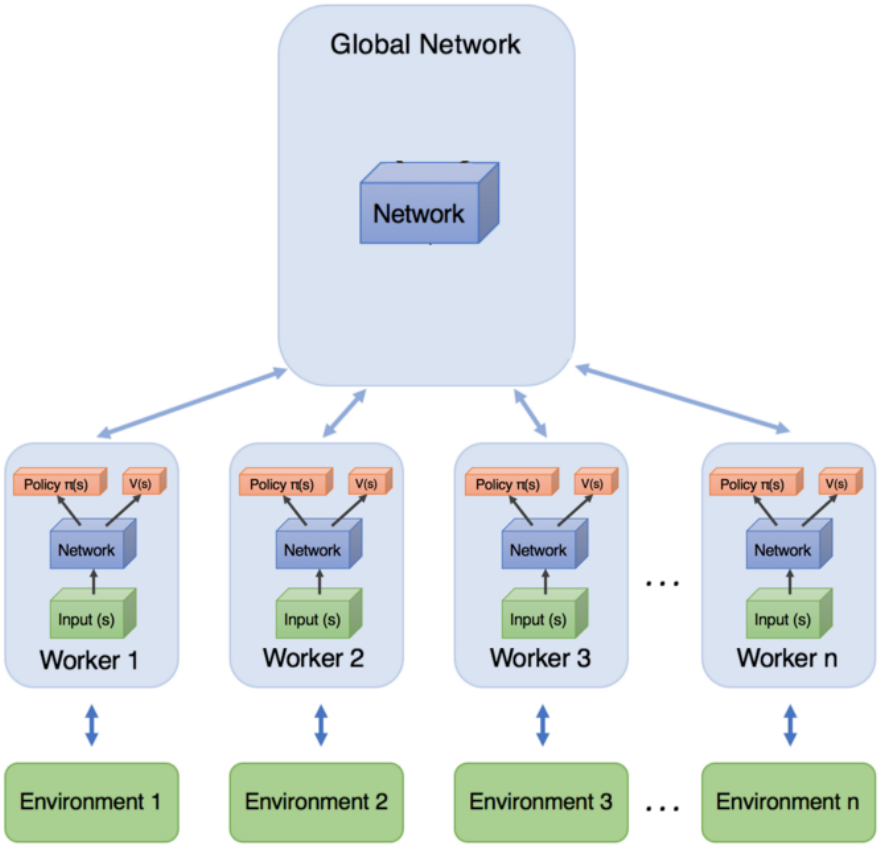

Figure 2: DDPG Architecture

recently introduced an implementation of A3C that uses TensorFlow on GPU to accelerate training Babaeizadeh et al. (2016).

3 Algorithms

3.1 Deep Q-Learning (DQN)

We implemented DQN as introduced by Mnih et al. The experience replay memory is a limited length queue of tuple (state, action, next state, reward). The details of the DQN algorithm can be found in Figure 4.

3.2 A3C

The A3C Algorithm begins with constructing a global network. Next, a set of worker agents with their own environments are created. Each worker has an actor and a critic module. The actor module is trying to learn the policy; the critic is trying to approximate the Q function. In the beginning of learning iteration, each worker agent begins by setting its network parameters to those of the global network parameters. Each worker then interacts with its own copy of the environment and collects experience. We use the experience to calculate value and policy losses. The worker then uses these losses to obtain gradients with respect to its own network parameters. A worker then uses the gradients to update the global network parameters. In this way, the global network is

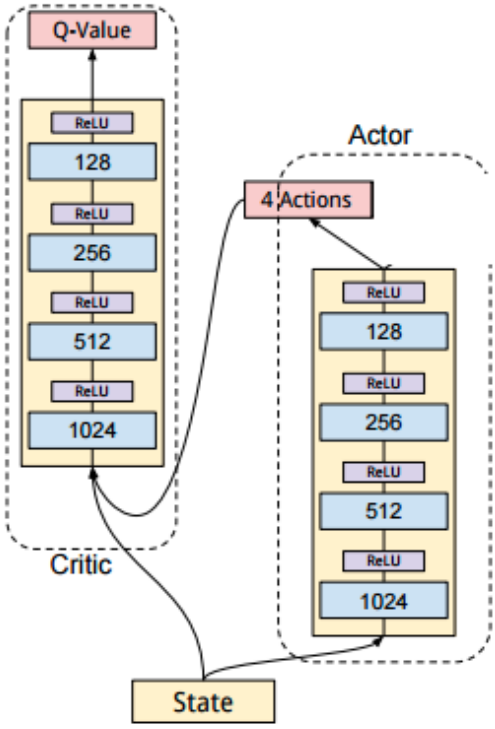

Figure 3: DDPG Architecture

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 ϵ 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 from Mnih et al. (2013)

constantly being updated by each of the agents, as they interact with their environment. So, this kind of architecture allows the agent to learn quickly compared to DQN. The pseudocode for A3C is shown in Figure 5

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 5: A3C PseudoCode from Mnih et al. (2016)

3.3 DDPG

Although A3C and DDPG have the same actor-critic style of learning, they have two completely different networks (i.e different parameters to learn). With A3C, the actor and critic are in the same network, they are just two outputs. In DDPG, the actor and critic are two different networks with two different update mechanisms. The actor will try to learn the best policy by using policy gradient, the critic will try to learn the Q function to critic the action given by the actor. The detail of DDPG algorithm can be found in Figure 6

4 Comparison setups

We tested our implementation of DQN and A3C on the games Flappy Bird, Lunar Lander, Bipedal Walker and Bipdeal Walker Hardcore environments as shown in Figure 7; we tested DDPG on these games with the exception of Flappy Bird, since this game does not support continuous action spaces.

With Bipedal Walker and Bipedal Walker Hardcore, the games action space is continuous, so we discretized action space for running A3C and DQN Algorithms. The continuous action space is on the interval [-1, 1]; we discretized it into 3 values -1, 0 and 1.

With the exception of DDPG on Flappy Bird, we trained each algorithm on each game for one day. The wall-clock training times may not be particularly meaningful for comparison, however, as we performed training on a few different machines. The neural networks for all three algorithms were trained on GPU.

4.1 Flappy Bird

In Flappy Bird, the agent's input is raw pixel data of four consecutive frames, so we applied 3 convolutional layers before applying normal DQN or A3C. We do the following to pre-process the frame:

1. Crop the lower part
2. Scale to 84 x 84 size

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 6: DDPG pseudocode from Lillicrap et al. (2015)

3. Convert to Gray Scale
4. Subtract the background
5. Concatenate 4 consecutive frames

More information about the game is available here.

4.2 Lunar Lander, Bipedal Walker, and Bipedal Walker Hardcore

The agent's input from Lunar Lander is the physical signal state from the game. Our input is information from 4 consecutive steps. Lunar Lander has two versions: Discrete and Continuous, so we applied all of three algorithms to test their performance. (They use the same reward system.). The input for Bipedal Walker (both versions) is again the physical information from the game. The game description is available here.

For each environment and each algorithm, once the network has been trained, we store the parameters and run the game for 100 times. Table 1 shows the score statistics for each agent in each (compatible) test environment.

4.3 Agent parameters

DQN

For the Deep-Q learning algorithm, we used a learning rate of $1e^{-4}$, a minibatch size of 32, a discount factor of 0.99, a replay memory holding 50,000 episodes, and an epsilon starting at 0.1 and decaying to $1e^{-4}$.

A3C

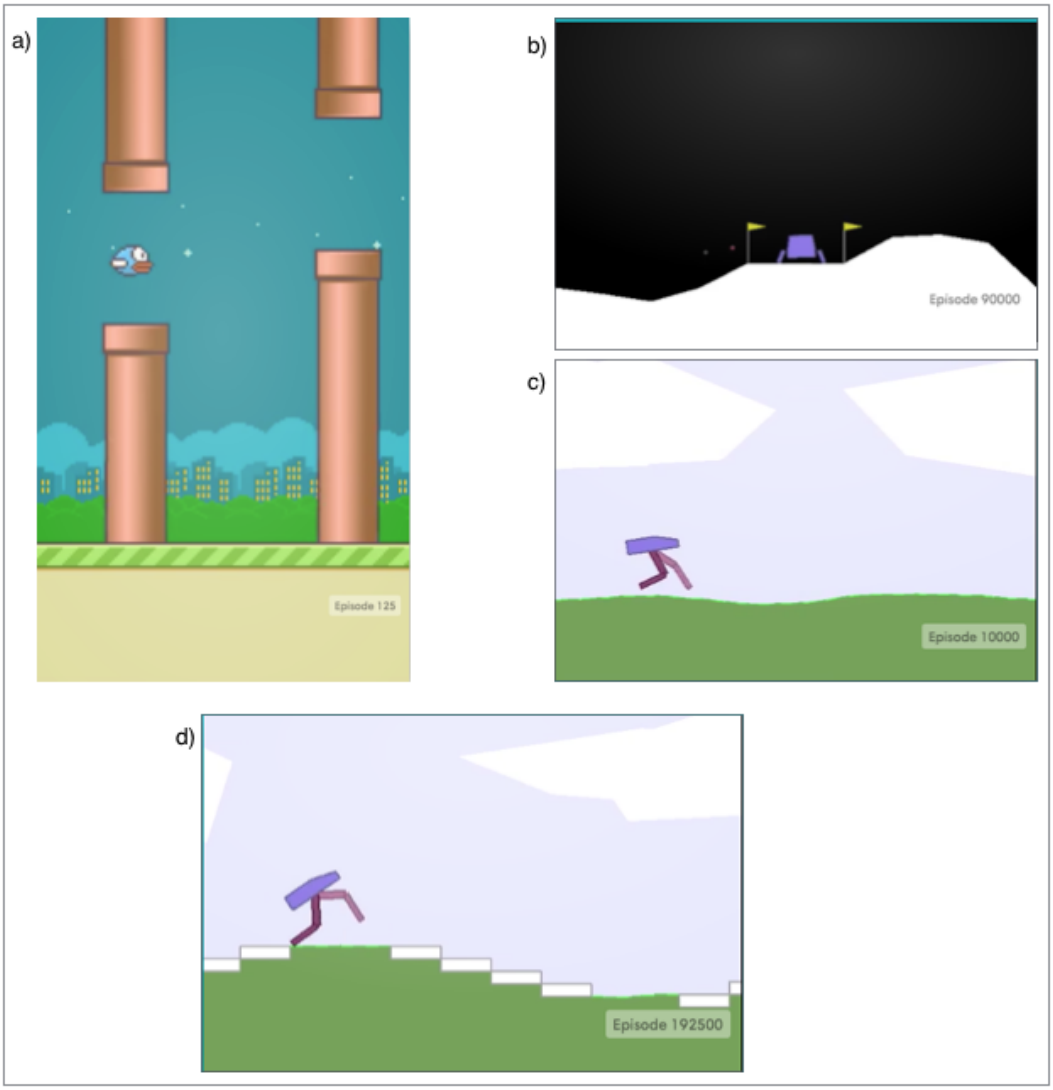

Figure 7: Figure depicts different environments the Algorithms been tested a) Flappy Bird b) Lunar Lander c)Bipedal Walker d) Bipedal walker:Hardcore

Our A3C implementation used a learning rate of $1e^{-4}$ and discount factor of 0.99. It used four worker agents. The actor and critic networks had a hidden layer size of 512. The number of step to update with the shared network a time is 32. When calculating final loss, the value loss is weighted by 0.5, the policy loss by 1.0.

DDPG

For DDPG, we used a Q learning rate of 0.001 and a policy learning rate of 0.0001. Epsilon started at 1.0 and decayed to 0.1. We used a replay memory of 1,000,000 episodes, and a batch size of 32. The networks had hidden layers of size 10 and 200.

5 RESULTS

As can be seen from the table of results 1, A3C and DQN were able to learn an effective policy in most of the environments to which they were applied. (The exception was A3C in Lunar Lander.)

Table 1: Results of different algorithms on different games

		Envirnoment			
	Score Statistics	**Flappy Bird**	**Lunar Lander**	**Bi. Walker**	**Bi. Wa. Hardcore**
A3C	*Max*	264.00	47.63	318.45	114.84
	Mean	109.78	-18.23	185.33	24.47
	St.d	87.46	28.70	130.53	32.48
DQN	*Max*	264.00	81.61	287.71	281.09
	Mean	109.54	51.57	184.08	72.13
	St.d	82.58	13.29	92.31	104.74
DDPG	*Max*	N/A	-7.76	3.83	6.57
	Mean	N/A	-71.41	-19.34	-25.61
	St.d	N/A	19.03	40.83	31.30

Our DDPG implementation, on the other hand, was unsuccessful. At this point, it is unclear to us why our DDPG agent was unable to learn in any of the three environments in which it was trained.

5.1 Flappy Bird

The Deep-Q learning agent required about 9,000 episodes of training (as shown in Figure 8) to achieve a peak score of 264 points, with scores in the 80-120 range being more common. Its mean score was 109.5. Our A3C implementation seems to have converged after about 4,000 training episodes. Its peak performance in a single episode was a score of 264 points. Its average, 109.8, was nearly the same as DQN. As we would expect, DDPG does not work on Flappy Bird because the action space in this game is discrete.

In the available time span, A3C completed more episodes of training than DQN (approximately 13,000 compared to 10,000). We suspect this may be because A3C does not use replay memory; thus the re-updating of parameters based on previous episodes does not happen. As we would expect, A3C converged much faster than DQN (about 4,000 compared to 9,000 episodes). It is well known that A3C scales well with multiple threads, because of (1) the parallelism of exploring the environment simultaneously with multiple worker agents, (2) the stabilizing effect of not being confined to a single trajectory at a time, and (3) the further stability gained by separating the value and policy networks into actor and critic modules.

Although A3C seems to have converged in about half as many episodes as DQN, both algorithms scored similarly in testing once the network weights had been learned, which suggests to us that both algorithms converged to a competitive policy. This video shows an example run of DQN trained on Flappy Bird.

5.2 Lunar Lander

As shown in Figure 9. After about 15,000 training episodes, DQN appears to have stabilized at an average score of approximately 51.6. Its peak score was 81.6. A3C, despite training for nearly 25,000 episodes, never performed higher than 47.6 points, and its average score in testing was -18.23. It is unclear to us why A3C, despite converging to a similar result in Flappy Bird as DQN (and in less time), so drastically underperformed A3C in this game. Throughout training and testing, Deep Deterministic Policy Gradient also scored consistently poorly, not improving or worsening much. It scored an average of -71.4 points.

When compared to A3C and DQN. The DQN was able to reach terminal state. Whereas, the A3C agent continuously applied a thrust in East direction, due to this reason the evaluation rewards where less than zero. Once the agent in that state, it always tries to exploit rather than exploring a new state.

5.3 Bipedal Walker

Our DQN agent approached its best score of a little under 300 after approximately 6,000 episodes, reaching peak performance after about 10,000 episodes. Its average score was about 184 points. A3C reached its peak performance quickly, within 2,000 or so episodes. It had a mean score of

185.3, and a maximum of about 318.5 points. Our DDPG agent did not improve performance after a few hundred episodes. It scored a mean of -19.3, and a maximum of 3.8. This can be seen in Figure 10

A3C and DQN again scored similar averages in testing, but the A3C algorithm had more variance in test scores. It is hard to interpret this result, but looking at the training data may be of some help here. Within the first thousand training episodes, A3C was demonstrating the ability to score well, and continued to do so through the first 5,000 episodes. Afterwards, however, the training scores dropped considerably on average, before improving somewhat but with much higher variance. A sort of cyclic behavior seems to have appeared at some point between ten and fifteen thousand episodes. This may have resulted from explore-exploit policy.

5.4 Bipedal Walker Hardcore

On this game, DQN actually outperformed A3C, reaching training scores in the mid-200s after about 7,500 episodes. Its average test score was 71.1 points. A3C peaked quickly in this game but typically only reached scores around 100, earning an average score in testing of about 24.5. After 4,000 or so episodes, it stopped improving performance in training. This can be seen in Figure 11

We saw a significant difference in rewards achieved by DQN and A3C Agents in this environment. From Table 1, the standard deviation of DQN is high compared to A3C. We suspect may be due to over-fitting of neural networks. This kind of effect can be reduced by (1) early stopping, (2) carefully tuning the hyper-parameters of the network using validation step.

It may be informative to visually observe the noticeably differing behavior of the trained DQN and A3C agents. The A3C agent takes small, regular steps, whereas the DQN agent frequently takes wildly awkward large steps. But the DQN agent is often able to compensate when it throws itself into a poorly balanced state; the more conservative A3C agent seems often unable to recover from a misstep. We speculate that the critic module in A3C may be inhibiting the actor module from taking obviously "bad" decisions–leading to the lower variance in evaluation results–but that this inhibition also makes the agent excessively cautious. If the DQN agent's steps are too large and reckless, the A3C agent's steps are too small, for instance, to get it over the obstacles it encounters in the game. Thus, the critic module's input may be counter-productively influencing the agent's ability to explore. In future work, we would like to experiment with different weights for the value loss versus policy loss to see if we can come up with better results for A3C in this domain.

DDPG achieved a maximum score of just under 6.6, with a mean of -25.6.

6 Figures

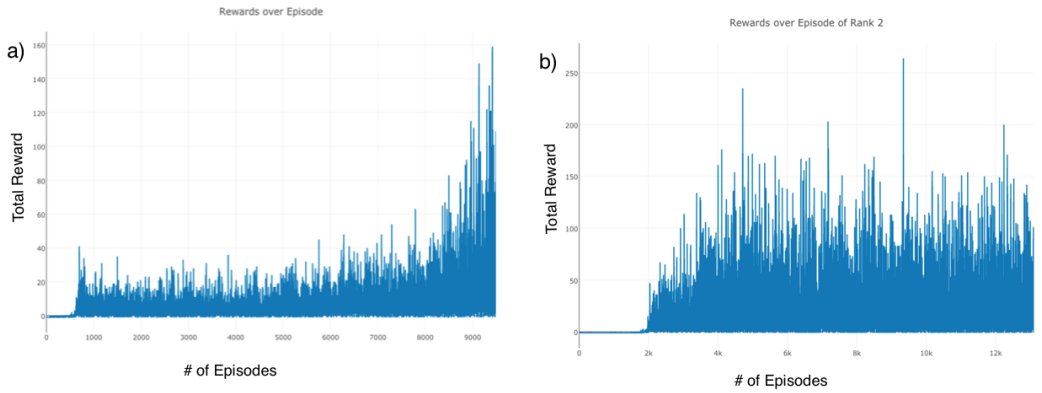

Figure 8: Performance of a) DQN b) A3C on training data for Flappy Bird Environment

Figure 9: Performance of a) DQN b) A3C c) DDPG on training data for Lunar Lander Environment

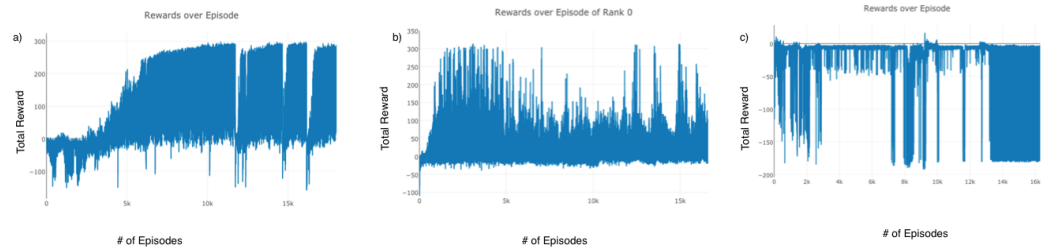

Figure 10: Performance of a) DQN b) A3C c) DDPG on training data for Bipedal Walker Environment

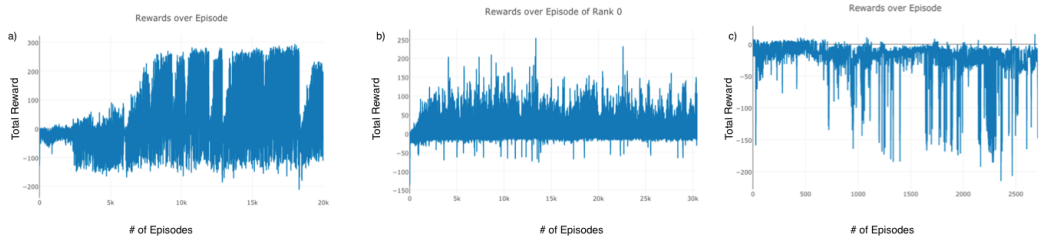

Figure 11: Performance of a) DQN b) A3C c) DDPG on training data for Bipedal Walker: hardcore Environment

REFERENCES

Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Ga3c: Gpu-based a3c for deep reinforcement learning. *CoRR*, abs/1611.06256, 2016.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

7 TEAM RESPONSIBILITIES

Khoi Nguyen was responsible for implementing the A3C algorithm. Durga Harish was responsible for the DDPG algorithm. Matt Meyn was responsible for the DQN implementation. We all contributed to testing and interpreting results and writing the report.