DEPARTMENT OF COMPUTER SCIENCE

Evaluation of Actor-Critic Models with Cart Pole

Jerry Wang (sw16437)

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Friday 22nd May, 2020

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of BSc in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Jerry Wang (sw16437), Friday 22nd May, 2020

Table of Contents

List of Figures

List of Tables

Chapter One

Introduction

1.1 Artificial Intelligence and Machine Learning

Machine learning is an important branch of artificial intelligence that provides a system the ability to carry out intelligent tasks such as decision making, data classification and prediction. It automatically learns and improves from experience without explicit programming. Reinforcement learning is a category of machine learning, it creates virtual agents that learn by interacting with the environment and finding the solution through maximising reward. The agent is rewarded when performing and beneficial action and punished otherwise. Reinforcement learning differs from the above methods in that the system is not given labelled input and output pairs, and undesired actions are automatically corrected.

There are many developing reinforcement learning algorithms available. However, each has its own strengths and weaknesses and each is suited to excel in different environment. Therefore, it could be useful to compare these popular models in the same settings and find out about their respective performances. Policy gradient methods and Q-Learning methods are two popular classes of algorithms in reinforcement learning. Although both types of models have been constantly improved over time, there is a third type of hybrid models that have been created in recent years, attempting to capture the benefit of both categories in one algorithm. These algorithms are often known as actor-critic algorithms and have both policy-based as well as value-based logic built in. They are praised for being able to solve certain problems faster but more detailed analysis is rarely found. In this experiment, several reinforcement learning algorithms will be implemented and detailed analysis on different aspects of their performances will be carried out in order to find out the real advantages and drawbacks of actor-critic algorithms.

1.2 Terminology

- action: An action is chosen by the agent using the policy. The agent uses an action to perform transitions between states in an environment.
- agent: A virtual entity that uses a policy to maximize expected return gained from

transitioning between states in an environment.

- critic: The value network component in an actor-critic network.
- environment: The world that the agent travels in. The agents interacts with the environment and receives feedback from the environment in order to improve.
- episode: A complete iteration of an agent's attempt to learning an environment.
- policy: An agent's probabilistic mapping from states to actions.
- reward: The numerical result of taking an action given a state. The rewards are defined by the environment.
- state: The parameter that describes the current configuration of the environment which the agent uses to sample an action.
- trajectory: a sequence of tuples that represent a sequence of state transitions of the agent, where each tuple corresponds to the state, action, reward, and next state for a given state transition.

1.3 Objective and Outline

The aim of this project is to evaluate the merits of actor-critic algorithms in detail by comparing the performance of popular actor-critic models with "non-hybrid" models such as the classic REINFORCE and DQN.

The remainder of the paper will introduce the concept of reinforcement learning, policy gradient, Q-Learning and actor-critic models, as well as the implementation of some popular models from the above categories and the evaluation and comparison of their performances.

Chapter Two

Background

2.1 Overview

This chapter will introduce the key concepts of various reinforcement learning algorithms that will be implemented in the next chapter, including REINFORCE, A2C, A3C, and DQN.

2.2 Reinforcement Learning

Reinforcement learning is a subset of machine learning that learns through enabling a virtual agent to interact with the environment in order to maximise a notion of cumulative reward. A plain reinforcement learning model is implemented as a Markov decision process (MDP) shown in Fig. 2.1, which is specified by a 5-tuple $< S, A, P_a, R_a, \gamma >$, where

- S is a set of environment states.
- A is the set of agent actions.
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of transitioning to state s' from state s under action a.
- $R_a(s, s')$ is the expected immediate reward by transitioning to state s' from state s under action a.
- γ is the discount factor for future rewards. This constant is used to adjust an agent's focus on current and future rewards. When γ is set to a low value, the agent prioritises short-term returns. When γ is set to a high value, the agent takes more long-term rewards into consideration.

A reinforcement learning agent interacts with the above given environment in discrete time steps. For each time step t, the agents arrives at state s_t and receives the observations of the environment including the reward r_t, and the available set of actions at state s_t. By choosing action a and performing transition (s_t, a, s_{t+1}), the agent receives the corresponding reward r_{t+1}, and the environment moves to the resulting state s_{t+1}. A policy π is defined as the probability distribution of actions given a state. The aim of the agent is find the policy π that collects the maximum amount of total reward expressed in equation 2.1

$$R = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.1}$$

The agent walks through the virtual environment, producing a trail of states, actions, and rewards. This is known as a trajectory. Sub-optimal trajectories will result in the notion of regret. Thus, the agent must learn to consider the long-term consequences and expected returns when choosing its actions even if the immediate return may be negative.

With every element in an MDP present, the optimum MDP policy and its value can be found by executing classic planning algorithms such as value iteration and policy iteration [4]. This is a very sensible framework that closely resembles many real world scenarios, such as the training of pets, bonus for outstanding employees and etc. All of which make use of the notion of rewards and regret.

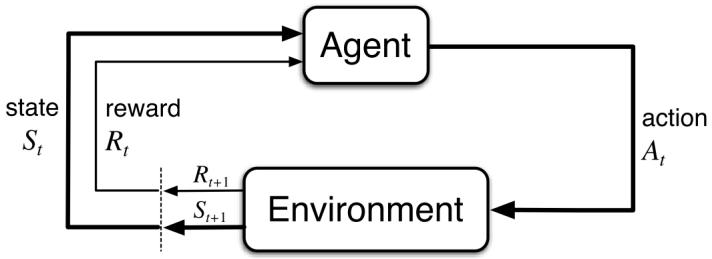

Figure 2.1 The agent-environment interaction in a Markov decision process

2.3 Policy Gradient Algorithm

Policy gradient methods are a popular type of reinforcement learning techniques that rely on optimising parametrised policies to solve a given task. Policy determines the probability distribution of actions according to a given state. And gradient refers to the use of gradient ascent or gradient descend for updating the parameters of the current policy. Each policy controlled by a set of parameters is associated with an expected reward. This is the amount of reward that an agent is likely to receive if it interacts with the environment following the given policy. The aim of policy gradient algorithms is trying to find an optimum policy and set of parameters that allows the agent to achieve maximum rewards by repeatedly nudging the policy with the gradient. The quality or value of a set of parameter θ is defined as the expected reward following trajectory τ under policy π as shown in 2.2.

$$J(\theta) = \mathbb{E}_{\pi}[r(\tau)] \tag{2.2}$$

The updating rule for the parameters is defined in equation 2.3, where $\alpha \in \mathbb{R}_+$ is a small scaling factor. The term $\alpha \nabla J(\theta_t)$ is to be deducted from the current parameters (or

added in the case of gradient ascent), making it move against the gradient towards the local minimum. This step is repeated forming a monotonic sequence until the parameters converge at an adequate local minimum.

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t) \tag{2.3}$$

To calculate the gradient $\alpha \nabla J(\theta)$, Policy Gradient Theorem is required. Policy Gradient Theorem states that the gradient of the expectation of the reward equals to the expectation of the reward multiplied by the gradient of the log of the policy π. Therefore, we can rewrite the term $\nabla J(\theta)$ as follow in equation 2.4.

$$\nabla J(\theta_t) = \nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}[r(\tau) \nabla \log \pi_\theta(\tau)] \tag{2.4}$$

The term $\pi_\theta(\tau)$ can be expanded into 2.5. Where $\mathcal{P}$ represents the distribution starting from state s_0. From this onward, each new action probability is independent and can be multiplied with the previous. This is due to the nature of First-Order Markov.

$$\pi_\theta(\tau) = \mathcal{P}(s_0) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}, r_{t+1}|s_t, a_t) \tag{2.5}$$

By applying log on both sides of the equation, the product can be decoupled into sums of probability s_0, likelihood of actions, and likelihood of transitions in equation 2.6. Finally by finding the gradient we arrive at the desired equation 2.8.

$$\log \pi_\theta(\tau) = \log P(s_0) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \sum_{t=1}^{T} \log p(s_{t+1}, r_{t+1}|s_t, a_t) \tag{2.6}$$

$$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \tag{2.7}$$

$$\nabla \mathbb{E}_{\pi_\theta}\left[r(\tau)\right] = \mathbb{E}_{\pi_\theta}\left[r(\tau)\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)\right)\right] \tag{2.8}$$

This is an important step because it removes the state probability distribution P from the process, which is a variable that is often very hard to model in a practical situation. Policy gradient methods using this framework are classified as model-free implementations.

The remaining expectation term is removed with Markov Chain Monte-Carlo method (MCMC). MCMC samples a very large number of trajectories and takes the average. This produces an approximation of the policy. Although it is an unbiased process, the characteristics of random sampling often leads to high variance in the policy.

2.3.1 REINFORCE

REINFORCE is used as the basis of many advanced policy gradient algorithms. It is a Monte Carlo variant of the plain policy gradient method, originally proposed by Ronald J. Williams [14].

$$G_t = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \tag{2.9}$$

$$\nabla \mathbb{E}_{\pi_\theta} [r(\tau)] = \mathbb{E}_{\pi_\theta} \left[\left(\sum_{t=1}^{T} G_t \nabla \log \pi_\theta(a_t|s_t) \right) \right] \tag{2.10}$$

In REINFORCE, the agent performs a full trajectory until the end of the game using its current policy. It then reflects on the probability of the policy and rewards. Then importantly, it calculates the expected future rewards G_t from each time step with a discount factor γ until the end of game (equation 2.9). And finally it calculates the policy gradient and updates the parameters with G_t using formula 2.10. The steps of REINFORCE can be concluded as in graph 2.2.

Algorithm 1 REINFORCE Algorithm

Initialise θ arbitrarily;
for each episode $s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T$ **do**
  for $t = 1$ to $T - 1$ **do**
    $\theta \longleftarrow \theta + \alpha \nabla_\theta \log \pi_\theta (s_t, a_t) v_t$
  end for
end for
return θ

2.3.2 REINFORCE with Baseline

In maximum likelihood estimate (MLE), data overwhelms the prior. This means no matter how incorrect the initial estimates are, the model can converge to true parameters given enough data. However, if the data samples contain high variance, it will be more difficult for the model to be stabilised.

One of the problem of the original REINFORCE method is that the sampled trajectories contain high variance which is also magnified by the scale of rewards. And the variance in these trajectories can cause a shift in the policy, causing it to drift further away from the optimal solution.

One method available to reduce the variance is by introducing a new variable called baseline b. This changes the policy gradient to the following (equation 2.11) and update rule (equation 2.12).The baseline is a state-value mapping function that is independent of

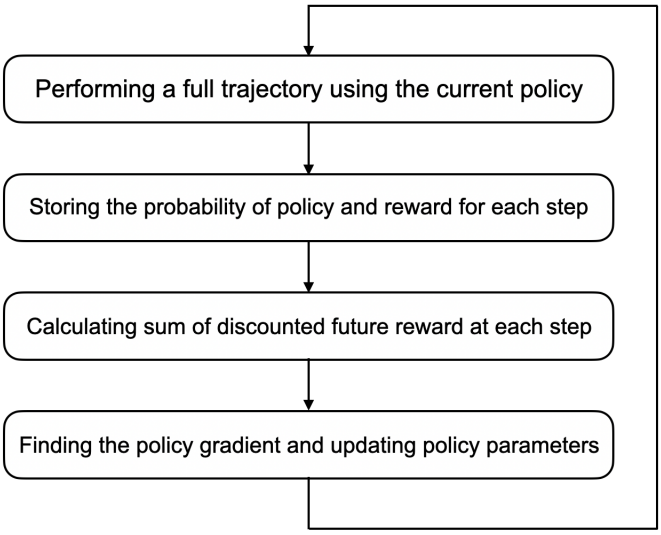

Figure 2.2 Flow diagram of REINFORCE

the policy parameters in order to keep the policy gradient unbiased. It is used in various state-of-the-art policy gradient algorithms such as Asynchronous Advantage Actor Critic (A3C). By subtracting the expected future rewards G_t with baseline b, what remains is the advantage reward. This is the difference between the actual reward, and the average amount of reward the agent usually receive at the given state. Because the advantages are smaller values compared to the original rewards, they will introduce lower variances.

$$\nabla \mathbb{E}_{\pi_\theta}\left[r(\tau)\right] = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T}(G_t - b)\nabla \log \pi_\theta(a_t|s_t)\right)\right] \tag{2.11}$$

$$\theta_{t+1} = \theta_t + \alpha\left(G_t - b\right)\nabla \log \pi_\theta(a_t|s_t) \tag{2.12}$$

With the addition of baseline, the gradient remains unchanged. This can be proved with the following (2.13).

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} b\nabla \log \pi_\theta(a_t|s_t)\right] &= \int \left[\sum_{t=1}^{T} \pi_\theta(a_t|s_t) b\nabla \log \pi_\theta(a_t|s_t)\right] d\tau \\
&= \int \left[\sum_{t=1}^{T} \nabla b\pi_\theta(a_t|s_t)\right] d\tau \\
&= \int \left[\nabla b\pi_\theta(\tau)\right] d\tau \\
&= b\nabla \int \pi_\theta(\tau)\, d\tau \\
&= b\nabla 1 \\
&= 0
\end{aligned} \tag{2.13}$$

This shows that using a baseline properly can both lower variance as well as keeping the policy gradient unbiased.

2.4 Q-Learning Algorithm

Q-Learning is a class of off-policy reinforcement learning algorithms[7]. It is the basis of many real life reinforcement learning applications. Such as the DeepMind AlphaGo that defeated Lee Sedol[2]. Like policy gradient methods, Q-Learning is also based Markov Decision Process. The agent traverses through states and receive a reward after performing an action and transitioning to the next state. Unlike policy gradient algorithms which are policy-based, Q-Learning algorithms are value-based. Because Q-Learning agents learns from all actions, including the those which are outside its current policy and random actions, a policy is not necessary.

$$Q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...\right] \tag{2.14}$$

The "Q" in Q-Learning refers to the Q-value, defined in equation 2.14. It can be interpreted as "quality" as the value determines how useful or good an action is at gaining future rewards. These Q-values are stored in an entity called Q-table (example 2.1). The Q-table has the shape of $[state * action]$, with each pair of state and action assigned with their respective Q-value. After each step of the game, Q-values are updated using rule 2.15. Where α is the learning rate, γ is the reward discount factor, and r_t is the reward received at step t. These updated Q-values then become the reference for action selection for the agent in the next episode. A diagram of Q-Learning is shown in 2.3.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a_t) - Q(s_t,a_t)\right] \tag{2.15}$$

REWARD	ACTION 0	ACTION 1	ACTION 2
state 0	1	0	0
state 1	0	0	2
state 2	1	0	0
state 3	0	1	0

Table 2.1 Example of a Q-table

In Q-Learning, starting from state s_0, the agent interacts with the environment in two ways. Either exploit using the Q-table as a reference, or explore taking random actions. In exploitation, the agent views the Q-table for all possible actions in its current state. It then selects an action with the maximum Q-value. In exploration, instead of choosing an action with the maximum future reward based on the Q-table, the agent performs a random action. This is equally as important and useful since the agent needs to attempt new trajectories that are able to solve the game that may otherwise never be explored. The rate of exploration is usually referred as epsilon (ϵ). The larger the value, the more exploration the agent performs. The smaller the value, the more exploitation the agent performs.

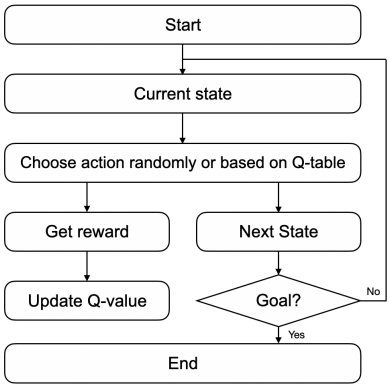

Figure 2.3 Flow diagram of Q-Learning

2.4.1 Deep-Q Network (DQN)

Although with the most basic form of Q-Learning, the agent is able to navigate through the game easily with the help of Q-table, the limitation of this framework manifests when as dimension increases. When there is a very large number of states and action combinations, it will be difficult to contain all the Q-values in a table. Learning and exploring all the combinations will also take a tremendously long time. The amount of memory required will

also increase exponentially. For example, in chess, just the state space alone holds more than 10^{120} entries, which is impossible to store. Therefore a more capable alternative is needed.

This is the idea behind Deep-Q Network (DQN). In this framework, the Q-value function can be approximated by a neural network instead of being remembered. This neural network serves as an approximation function. Denoted as $Q(s, a; \theta)$, where θ represents the weights of the network. Given an input state, the network will return all the possible actions with their Q-values. This enables a new type of Q-Learning algorithms that are better in solving higher-dimension problems. The difference between Q-Learning and Deep-Q Network is shown in figure 2.4.

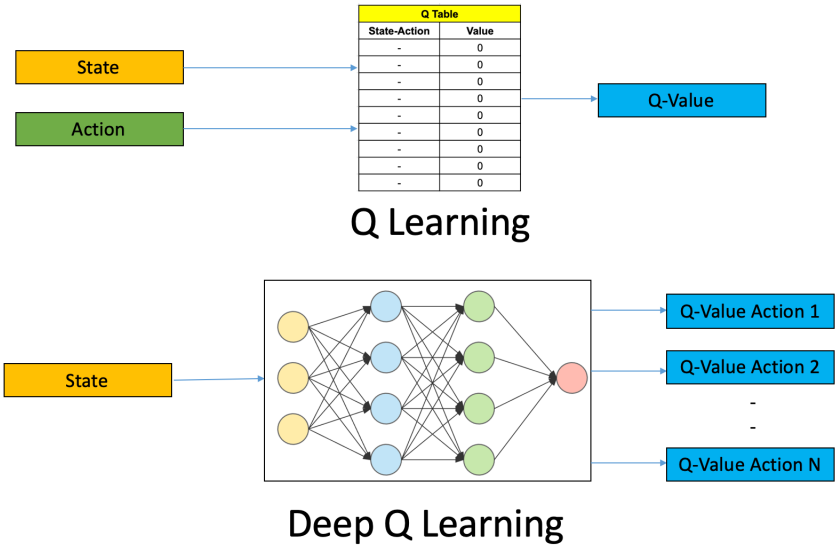

Figure 2.4 Comparison between Q-Learning and Deep-Q Network

DQN agent interacts with the environment in the exact same way. Updating the Q-values until they converge to the final values. The goal of DQN is to minimise the difference between the current predicted Q-value and the target Q-value. This is known as the cost, and is calculated by a cost function (2.16).

$$Cost = \left[Q\left(s, a; \theta\right) - \left(r\left(s, a\right) + \gamma \max Q\left(s', a; \theta\right)\right)\right]^2 \tag{2.16}$$

When this function is minimised and the neural network is stable. The agent will have then solved the environment.

2.4.2 Double Deep-Q Network (DDQN)

In Deep-Q Network, the cost function (2.16) behaves like the Mean Square Error (MSE) function where the predicted value is compared with the target value. However, the target values are not constant and can vary after every iteration as the agent updates its belief.

This may result in the over-estimation of Q-values and make it difficult to stabilise the network. Therefore, the concept of Double Deep-Q Network is introduced. This is done by keeping two copies of the Q network. One is referred to as "online network", it selects the best action to take for the next state. The other network is responsible for calculating the target Q-value of taking such an action at the next state. This design decouples the action selection process from the Q-value generation process. Therefore, DDQN is less likely to over-estimate Q-values and trains faster as well as being more stable[6].

Algorithm 2 Double Deep-Q Network (Hasselt, 2010)

Initialise network Q^A, Q^B
for each iteration **do**
  Choose a, based on $Q^A(s,)$ and $Q^B(s,)$
  Observe r, s'
  Choose (random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = argmax_a Q^A(s', a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s', a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = argmax_a Q^B(s', a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)\left(r + \gamma Q^A(s', b^*) - Q^B(s,a)\right)$
  end if
  $s \leftarrow s'$
end for

2.5 Actor Critic Methods

In the above sections, two types of algorithms were introduced:

- Policy-based algorithms (such as REINFORCE) where a policy is directly optimised without a value function. This is ideal when the action space is continuous or stochastic. The quality of the policy is measured by the amount of total reward received at the end of the episode. However, policy-based methods do not guarantee convergence. It is also not as simple as value-based methods since there is no tabular version of policy gradient methods. In some cases, policy gradient methods are slower than value-based methods because the latter use temporal difference (TD) learning methods with bootstrapping.

- Value-based algorithms (such as Q-Learning) where the agent learns a value function (or table) that maps each state-action combination to a "quality" value. Then it chooses the action that has the biggest action-value at each state. This suits environments with limited number of states and actions. However this type of methods are

not able to solve environments with continuous action spaces without approximation (such as sampling into a discrete action space), nor can it do well in environments where the optimal policy is stochastic due to the lack to trainable parameters.

Therefore, a new category of hybrid methods attempt to take the advantage of both worlds, actor-critic methods. These algorithms make use of two networks:

- A policy-based actor that controls how the agent behaves in the environment.
- A value-based critic that evaluates how good an action is.

This fusion architecture is used in many of state-of-the-art algorithms including Proximal Policy Optimisation, Advantage Actor Critic, etc.

One of the major drawbacks of policy gradient methods comes from the use of Monte Carlo. A REINFORCE agent for example would wait until the end of the episode to reflect on its actions based on the amount of reward it has received. This creates a problem in a situation where most of the trajectories are made up of good actions but with a fraction of it made up of very ad actions. A Monte Carlo agent would consider all the actions in the trajectory good because it leads to a high reward. However, this focus on total reward means the agent fails to learn that not all actions are good individually simply because they produce better results combined. Some agents attempt to overcome this by sampling very large amount of trajectories, this is effective to an extent, but often the learning process and trajectory convergence will take significantly longer.

This is what actor-critic algorithms are trying to address. Instead of waiting until the end of the episode, an actor critic agent makes an update after each step. In the update function for policy gradient (2.3), total reward G_t is used for updating the policy. However since updates are performed frequently, the approximated value provided by the trained critic function is used. This changes the update formula to the following (2.17).

$$\Delta\theta = \alpha\nabla \log \pi_\theta (s_t, a_t)\, Q (s_t, a_t) \tag{2.17}$$

In actor critic, the actor is a policy controlled by parameters θ expressed as $\pi(s, a, \theta)$, and the critic is a value function expressed as $\hat{Q}(s, a, w)$. Both of these networks run in parallel and updates separately. The policy network is updated with rule 2.18 where α is the learning rate of the actor and $\hat{q}_w (s_t, a_t)$ is the estimated action value. The value network is updated through rule2.19, where β is the learning rate of the critic and $\nabla\hat{q}_w (s_t, a_t)$ is the gradient of the value function.

$$\Delta\theta = \alpha\nabla \log \pi_\theta (s_t, a_t)\, \hat{q}_w (s_t, a_t) \tag{2.18}$$

$$\Delta w = \beta \left(R (s, a) + \gamma\hat{q}_w (s_{t+1}, a_{t+1}) - \hat{q}_w (s_t, a_t)\right) \nabla\hat{q}_w (s_t, a_t) \tag{2.19}$$

The process of actor critic method is shown in graph 2.5. At each time step t, the actor and the critic are given the current state as input. The actor receives the state s_t, takes action a, proceeds to new state s_{t+1} and receives reward r_{t+1}. The critic observes the action by the actor, calculates the value of the action and gives the feedback to actor (TD error). The actor is then able to update its policy parameters using this new-found Q-value. With this updated policy, the actor is able to choose an action at the next state s_{t+1} by incorporating what it has learned in the previous state. This is the benefit of bootstrapping learning, where the agent evaluates the actions individually instead of as a batch.

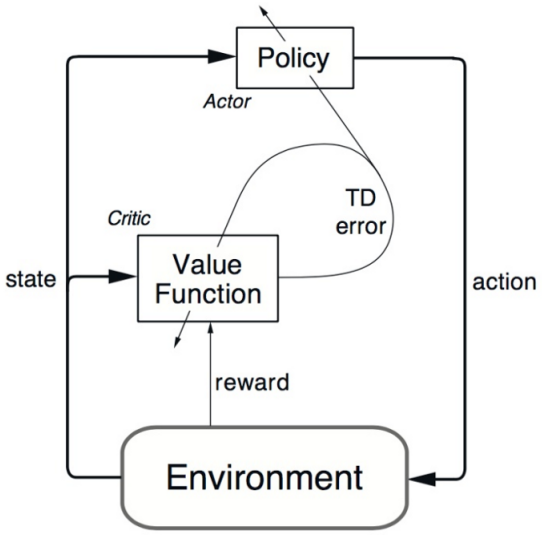

Figure 2.5 Flow diagram of Actor-Critic Method

2.5.1 Advantage Actor Critic (A2C)

While non-advantage actor critic methods use $Q(s_t, a_t)$, advantage actor critic (A2C) uses the "advantage" $A(s_t, a_t) = V(s_t) - Q(s_t, a_t)$ as its bootstrapping value. This value captures how much better the action is compared to it usually is. Instead of Q-values, the critic in A2C learns the advantage values. This means an A2C has a better chance of making the best out of a bad situation. It not only understands how good an action is, but also how much better it can be.

2.5.2 Asynchronous Advantage Actor Critic (A3C)

A3C is an asynchronous version of A2C. Instead of having a single agent that interacts with the environment, A3C has multiple independent agents interacting with parallel versions of the environment. All agents have their own weights and are trained in parallel. The

Algorithm 3 Advantage Actor Critic (A2C)

Initialise parameters θ, s
for each iteration **do**
  Sample $a \leftarrow \pi_\theta$
  Perform action a, receive reward r_t and next state s'
  Compute loss $\delta = r(s_t, a_t) + \gamma V(s_{t+1}) - V(s_t)$
  Update policy network
  $\theta \leftarrow \theta + \alpha \left(1/N \sum_{i=1}^{N} \left(\sum_{t=0}^{T} \gamma^t \nabla_\theta \log \pi_\theta \left(a_{i,t}|s_{i,t}\right) \left(r\left(s_t, a_t\right) + \gamma V\left(s_{t+1}\right) - V\left(s_t\right)\right)\right)\right)$
  Update $V\left(s_t\right)$ using target $r\left(s_t, a_t\right) + \gamma V\left(s_{t+1}\right)$
end for

update are carried out periodically to a global network which stores the shared parameters and behaves as a coordinator. Each agent pushes the update at different time, then copies the updated parameters back from the global network and continue its exploration. It then carries on for another n iterations and repeat the process. Using this framework as shown in figure 2.6 (adapted from [5]), the agents not only provides information to the global network, but also communicates with each other by resetting their parameters to global values during each update.

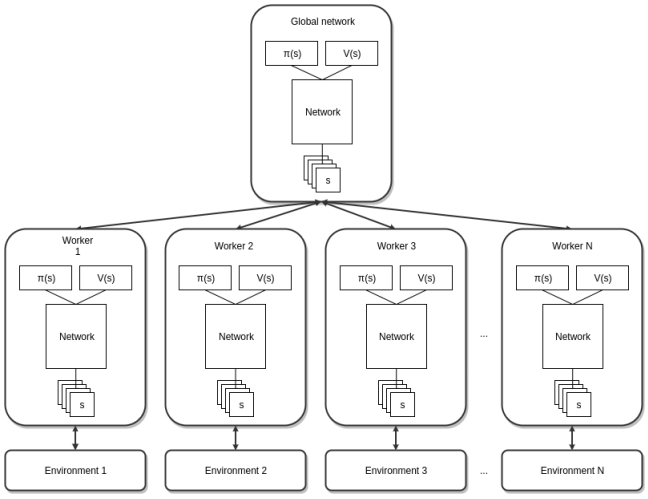

Figure 2.6 Flow diagram of Asynchronous Advantage Actor Critic

2.6 Neural Network

In machine learning, neural networks are built on the idea of simulating the logic of a human brain. When the brain receives the input from the body, some neurons are activated as a response to the incoming signal. The result from multiple neurons are then collected by the

following neurons, combined and propagated to the next. An artificial network mimics a similar behaviour.

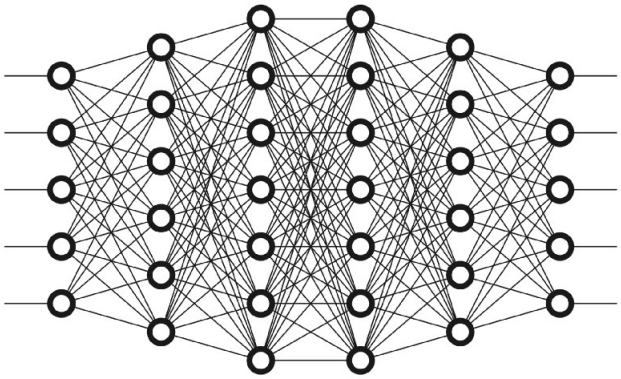

Figure 2.7 Diagram of a neural network

Each neuron is associated with a weight and bias. An linear transformation is performed when the input enters the neuron (2.20). This new found value is then entered into an activation function which determines its value moving onto the next layer (2.21).

$$x = (weight * input) + bias \tag{2.20}$$

$$f(x) = Activation\left(\sum (weight * input) + bias\right) \tag{2.21}$$

2.6.1 Activation Function

Activation functions in neural networks are functions that take the values from the neurons of the previous layer and determine and outputs an answer for the neurons in the next layer. It is important to choose the correct activation function as different activation functions are designed to be used in different parts of neural networks. There are mainly two types of activation functions:

- Linear activation function
- Non-linear activation function

A very simple example of a linear activation function is $f(x) = x$. Such functions don't have a range limitation in terms of its output value. However it is rarely used because it is not able to respond to complex data input.

More often, non-linear functions are used as activation functions in neural networks. This type of functions enable the model to generalise and adapt to a variety of data. Two of the most important activation functions are Rectified Linear Unit (ReLU) and softmax. While ReLU is used to simulate the reaction of biological neurons, Softmax is used for calculating the probability of actions in the action space.

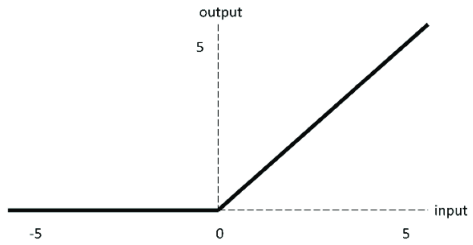

Figure 2.8 Rectified Linear Unit (ReLU)

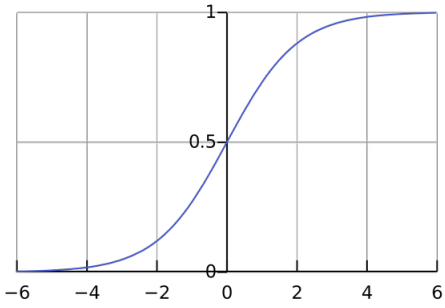

Figure 2.9 Softmax Activation Function

ReLU Activation

ReLU is one of the most popular activation functions that is often used in hidden layers. The main advantage of ReLU over other activation functions is that not all neurons are activated at the same time. And all the neurons with a negative output will return 0 and deactivated. The function can be expressed as $f(x) = \max(0, x)$.

Softmax Activation

Softmax is another popular activation function commonly used at the output layer (2.22). It behaves like a combination of multiple sigmoid functions, thus having the ability to perform multi-class classification problem. The function returns the probability of an input belonging to an individual class. The sum of these values will always be 1.

$$\sigma(z)_j = \frac{e_j^z}{\sum_{k=1}^{K} e_k^z} \; for \; j = 1, ..., K \tag{2.22}$$

Chapter Three

Implementation

3.1 Overview

Following the concepts introduced in the previous chapter, this chapter will introduce the process as well as the challenges of implementing the models listed above. This includes the hardware and software libraries used for implementation and testing, the reasoning for choices of hyper-parameters and the method used for data collection.

The strategy is to approach REINFORCE first as it is the most basic model with a policy network containing plenty of components that can be reused in later models. A2C is the second model to be implemented as it makes use of the same policy network, followed by A3C which is the multi-threaded version of A2C. A diagram of implementation hierarchy is shown below.

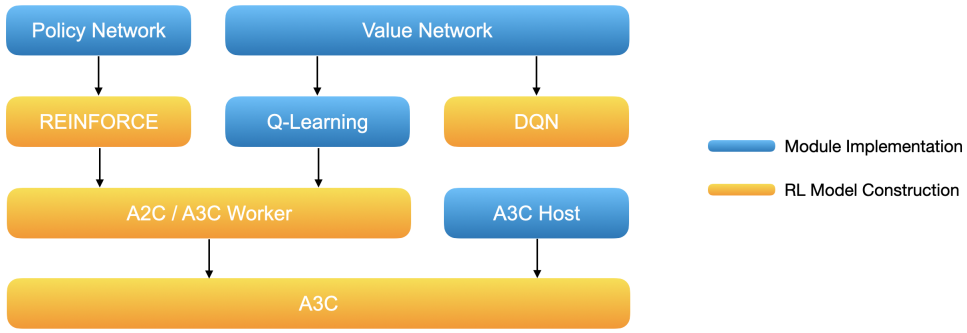

Figure 3.1 Implementation Flow Diagram

3.2 Testing Hardware

The data generated by the model were trained on the following hardware:

- Intel Core i7-7820HQ (4-core, 8-thread)

- AMD Radeon Pro 560

3.3 Software Libraries

3.3.1 PyTorch

PyTorch is an open source machine learning library used for developing and training neural network and deep learning models. It can be adapted to both C++ and Python code. The Syntax of PyTorch is very similar to python, making it much easier to read and understand as well as supporting plenty of the existing Python libraries. While there are other existing machine learning frameworks such as scikit-learn, PyTorch focuses more on deep learning. While numpy arrays can be used both on CPUs and GPUs, PyTorch tensors can be used on GPUs specifically with CUDA support for faster training and learning.

In this paper, the computer used for training does not have an NVIDIA graphics card with CUDA support. Therefore, the training process is entirely dependant on the performance of the CPU. It is unclear what is the advantage of using an adequate NVIDIA graphics card over a CPU. This may affect the judgement and comparisons of the performance between the Asynchronous Advantage Actor Critic (A3C) algorithm which utilises PyTorch multi-threading and its closely related counterpart Advantage Actor Critic (A2C).

Version used in this implementation: `torch 1.4.0`, `Python 3.7.2`
Modules used in this implementation:

- `time`
- `random`
- `numpy`
- `collections.deque`
- `matplotlib.pyplot`

3.3.2 Keras

Keras is another open source machine library written in Python. Unlike PyTorch that uses `torch` back-end, Keras uses `tensorflow` back-end. They also differ in terms of the level of abstraction provided. Both frameworks provide adequate performance when training and GPU acceleration support. However Keras excels in terms of ease of use with convenient features such as automatic back-propagation, which makes it ideal for fast prototyping and experimentation. When implementing relatively standard models such as DQN, Keras is a much more efficient choice compared to PyTorch. At the same time, PyTorch is the more preferred library for A2C and A3C because it allows for a greater degree of customisation.

Version used in this implementation: `Keras 2.3.1`

3.3.3 OpenAI Gym

OpenAI Gym is a library for developing and comparing reinforcement learning algorithms. It supports many classic reinforcement learning environment as Cart Pole, Pendulum, Mountain Car, etc. Since there are plenty of implementations that utilises Gym environments, it provides a standard for comparing the implementations of an algorithm in this paper to an existing one.

Every Gym environment comes with an action space and an observation space. Both spaces can either be continuous or discrete.

Version used in this implementation: `gym 0.15.4`

3.3.4 Cart Pole

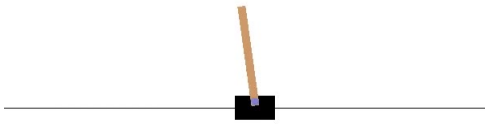

Figure 3.2 Cart Pole Environment

The Cart Pole is a classic reinforcement learning environment. It corresponds to the version of the cart-pole problem that was originally proposed by Barto, Sutton and Anderson. It consists of a pole which can move along a horizontal track with no friction. The cart is controlled by an action space of horizontal movement of 1 to either the left side or the right side. The pole starts upright. The goal of the game is to make the pole to remain upright for as long as possible. For every time step that the pole stays upright, the agent receives a reward of +1. The game ends when the pole is more than 12 degrees from vertical or the cart is more than 2.4 unit away from the centre starting point or the episode length is greater than 200. In cart pole, both the action space and the observation space are discrete.

It can be observed from the source code of Cart Pole[1] that the environment has an observation space of size 4 and an action space of size 2.

Version used in this implementation: `CartPole-v0`

[1] https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

3.4 Hyper-parameter

3.4.1 Hidden Layer

Choosing the right number and size of hidden layers is very important. The more hidden layers, the better the model is at collapsing data. While the model is able to achieve a 95% performance, the number of neurons and hidden layers should be kept as low as possible. This is because an excess number of nodes and layers will lead to over-fitting. It will also enable the network to memorise the training set to perfection. This means the network becomes a memory bank and when it will perform poorly in any environment that is not part of the training set.

It has been shown that in both linear and quadratic cases, the learning rate of single hidden layer networks is more flexible than that of multiple hidden layer networks. It is also observed that single hidden layer networks converge quicker to linear target functions compared to multiple hidden layer networks. [10]

Therefore only one hidden layer is used in all the implementations. While the appropriate number of neurons in this layer still requires further experimentation.

3.4.2 Choice of Optimiser

PyTorch provides a variety of back propagation optimiser, including Adagrad, RMSprop, SGD/ASGD, etc. Adam[2] (A Method for Stochastic Optimisation) is currently a very popular choice of optimiser in reinforcement learning. The advantage of using Adam includes producing lower training error and higher learning rate, making it performing significantly better than the other methods according to the original paper. [3] Therefore, Adam is used as the default optimiser for all the implementation in this paper.

3.4.3 Choice of Learning Rate

Learning rate is a hyper-parameter that determines how much the weight of the network is adjusted with respect to the loss gradient. The value can range from 0 to 1. In gradient descent, the lower the learning rate, the slower the parameters will travel down the gradient slope in its search of a minimum. The higher the value, the larger the stride.

Because the convergence of policy is crucial to the performance of models in policy gradient algorithms, it is important to select the optimal learning rate during the training process. While an overly high learning rate can cause the model to miss a local minimum or to diverge instead of converging, a learning rate that is extremely low means that the algorithm will requires a much larger number of episodes to converge, thus costing more time. This is demonstrated in the graph below[3] (3.3).

[2] https://pytorch.org/docs/stable/optim.html

[3] https://www.deeplearningwizard.com/deep_learning/boosting_models_pytorch/lr_scheduling/

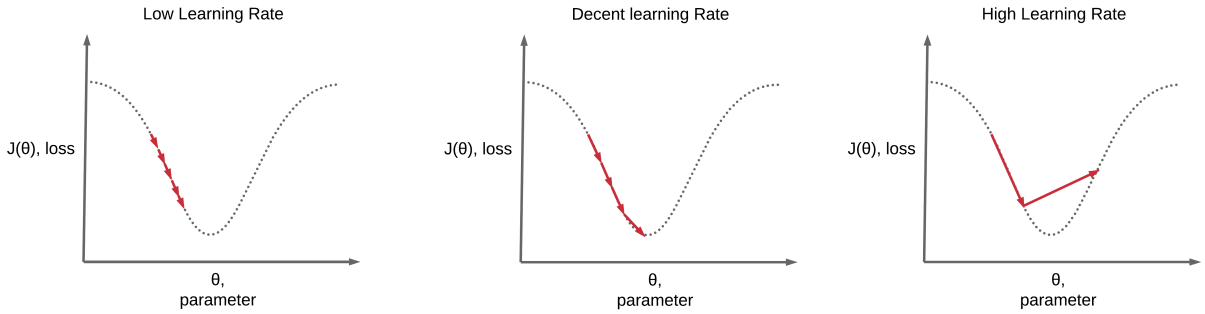

Figure 3.3 Learning Rate in Gradient Descent

Therefore, all models are trained initially with a relatively low learning rate to ensure accuracy. The learning rate is then gradually increased until a optimal value is found.

3.4.4 Choice of Discount Factor

The discount factor is also another important factor as it determines how much the agent values current reward over the ones in the future. The value can range from 0 to 1. Although in many researches, it is common to use a discount factor of 0.9. Because balancing a pole requires the agent to consider how much a current action can affect the balance of the pole in the future, this seems like a reasonable value to use. Therefore, 0.9 is used as the default value for all implementations.

3.5 Data Collection

3.5.1 Performance Plotting

In order to analyse the performance of the algorithms, various information will be collected on-the-fly. These include the reward value for each episode, the average reward value, the runtime of an individual episode, and the average runtime of episodes. The time consumed by each episode is acquired through the `time` module in `Python`. By enclosing each episode with a pair of `time.time()`, we are able to measure the runtime of each episode in the most accurate manner.

Using `matplotlib.pyplot.ion()`, the program is able to showcase these values and training progress in real time. This is a beneficial implementation as the training process can be extremely time-consuming. Therefore real-time plotting requires a user less time to determine if the algorithm is working correctly, providing more flexibility in testing and debugging.

```
class ReinforceNetwork(nn.Module):
    def __init__(self):
        super(ReinforceNetwork, self).__init__()
        self.fc1 = nn.Linear(4, 128) #observation space to hidden layer
        self.fc2 = nn.Linear(128, 2) #hidden layer to action space

    def forward(self, x):
        x = F.relu(self.fc1(x))
        y = self.fc2(x)
```

Listing 3.1 REINFORCE Network Structure

3.5.2 Script Profiling

To measure the hardware usage of each script, a separate profiler is required. Memory Profiler[4] is a python library that is able to track and trace and memory usage of entire scripts and code snippets, in single-threaded as well as multi-threaded applications. Using the `mprof` module in the `memory-profiler` package enables the script to generate time-based memory usage graphs. For example, `mprof run <executable>` is used for profiling single-threaded algorithms such as REINFORCE and A2C, and `mprof run -multiprocess <script>` is used for profiling multi-threaded algorithm such as A3C.

Version used in this implementation: `memory-profiler 0.57.0`

3.6 Models

3.6.1 REINFORCE

As a start, the most basic policy gradient algorithm REINFORCE is used as an attempt to solve the environment.

The simple REINFORCE agent contains two fully connected layers. No additional layers are used. This is to avoid over-fitting. The first layer maps the input observation space to the hidden layer of size 128 with ReLU activation, and the second maps the hidden layer to the output action space with softmax activation for classification.

The model is trained for 1000 episodes with a maximum running step of 200 and max reward of 200 and learning rate of 0.001. The agent considers the environment solved and terminates learning when the reward value reaches 195 for 100 consecutive trials.

[4] https://pypi.org/project/memory-profiler/

```
class A2CNetwork(nn.Module):
    def __init__(self):
        super(A2CNetwork, self).__init__()
        self.policy1 = nn.Linear(4, 256) #observation space to hidden layer
        self.policy2 = nn.Linear(256, 2) #hidden layer to action space

        self.value1 = nn.Linear(4, 256) #observation space to hidden layer
        self.value2 = nn.Linear(256, 1) #hidden layer to value output

    def forward(self, state):
        policy = F.relu(self.policy1(state))
        policy = self.policy2(policy)

        value = F.relu(self.value1(state))
        value = self.value2(value)
```

Listing 3.2 A2C Network Structure

3.6.2 REINFORCE with Baseline

The baseline model is constructed using mostly the same components as the basic REINFORCE. It uses the same size of hidden layer, the same activation functions, optimisers and learning rate. However during each episode, once the initial rewards have been calculated, the baseline will be deducted from those values. These will then go on to be used for updating the policy.

3.6.3 Advantage Actor Critic (A2C)

The Actor-Critic model contains two network that needs to be defined and optimised separately.

The actor network contains two fully connected layer, with the hidden layer of size 256. The first layer maps the input observation space to the hidden layer with ReLU activation, and the second maps the hidden layer to the output action space with softmax activation for classification.

The critic network contains two fully connected layer, with the hidden layer of size 256. The first layer maps the input observation space to the hidden layer with ReLU activation, and the second maps the hidden layer to a single output that represents the critics Q-value.

The model is trained for 1000 episodes with a learning rate 0.001.

```
with self.global_episode.get_lock():
    self.global_episode.value += 1
```

Listing 3.3 MultiProcessing Mutex Lock

3.6.4 Asynchronous Advantage Actor Critic (A3C)

The A3C algorithm requires multiple workers to train in parallel. Therefore a worker class is implemented. The worker contains all the functions included in the A2C model, with an addition function that is responsible for synchronising with the global parameters. Both the global network and the worker network are constructed in the same way as the A2C network, they are also equipped with the same optimiser and learning rate.

At the beginning of execution, based on the number of threads available on the host CPU, the driver program will spawn the same number of workers. Using `torch.multiprocessing` module, the workers are able to achieve real parallel processing.

Challenges

There are two major challenges when implementing Asynchronous Advantage Actor Critic. Ensuring that the program does not deadlock, and the collection and plotting of data from parallel worker agents. The common cause of deadlocks when using `torch.multiprocessing` is the thread starting method. The default starting method in `torch.multiprocessing` is `fork`. This is dangerous in the context of multi-threading programs. If there's any thread that holds a lock or imports a module, and fork is called, it's very likely that the child process will be in a corrupted state and will deadlock or fail in a different way. [11]

Therefore in the driver program before the workers are created, `mp.set_start_method('spawn', force=True)` is used to ensure that all workers are spawned instead of forked. It is also important to be careful every time when the worker reads or write the global variables in the driver program. One of such examples is the number of episode. To ensure the same episode is not executed twice, the `get_lock()` method is used. Each worker is only allowed to modify the global episode number while holding the mutex lock (Listing 3.3). After the worker has finished executing the episode, the global number is incremented by 1 and the global parameters are updated. This addressed the deadlocking issue in my very first implementation of A3C algorithm.

The other challenge is the collection of reward data. Because the reward for each episode is calculated locally on each worker. These information needs to be retrieved for plotting. Therefore, a new storage structure is needed inside the global driver program. Using `torch.multiprocessing.Manager().dict()`, it is possible to create a shared dictionary structure that all agents can read and write. This is also the ideal structure compared to `list` or `int` because a dictionary ensures all the keys are unique, which means the reward value for each episode is only stored once. Unlike a list of rewards, which requires that the

```
class ValueNetwork(nn.Module):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(4, 256) #observation space to hidden layer
        self.fc2 = nn.Linear(256, 1) #hidden layer to action space

    def forward(self, state):
        value = F.relu(self.fc1(state))
        value = self.fc2(value)
```

Listing 3.4 Deep-Q Network Structure

data to be in sequential order so that it remains valid for plotting, a dictionary does not require the reward values to be inserted in a specific order. This is beneficial in A3C since it is difficult to determine if the workers will finish the episodes in the original order upon completion. A similar implementation was done for runtime information.

Because the host Python code will be paused after workers have started execution and will keep waiting until the worker threads terminate, it is difficult to plot the real-time data gathered in the host program in an A3C algorithm implementation. Therefore the figure will only be plotted after all training episodes have finished.

The model is trained for 1000 episodes with a learning rate 0.001.

3.6.5 Deep-Q Network

In Deep-Q Network, a single value network is constructed with three layers of neurons. The first layer maps the input space to the hidden layer. And the second layer maps the hidden layer to the output value of size 1. The middle layer is hidden with size of 128. Both the second layer and final layer uses ReLU activation function.

Experience replay is also implemented in this version of DQN. Because DQN stores the information discovered throughout the training process, a mini batch is sampled from these experiences. To implement this, a buffer of size 10000 is created within the agent, and each set of state, action, reward is pushed to the memory after each step taken by the agent. The newly updated buffer is then used for sampling in the next step.

In addition, DQN balances exploration and exploitation through a hyper-parameter ϵ as well as a decay rate. ϵ is set to 1 at the beginning of the game. This is because the agent is not aware of the environment at first, and it can only increase experience through exploring. However, as more episodes are performed and the agent has gained more experience, the agent should make better use of the information it has already gathered. This is done by decaying the exploration rate. The decay rate is set to 0.95. This is not an assumed value however, as a parameter such as decay rate requires many trials and errors to tune and

stabilise.

The model is trained for 1000 episodes with a learning rate 0.001, an initial epsilon of 1, and a decay rate of 0.99.

Chapter Four

Evaluation

4.1 Overview

The code for the above algorithms are trained for 1000 episodes with a maximum of 200 time steps. The learning rate is set to 0.001, and the discount factor is set to 0.99. Each model outputs a figure as the report, containing:

- Reward value achieved by individual episodes
- The average reward value over the period of the training
- The runtime of each training episode

Each of the above criteria will be compared across all models in the sections below.

4.2 Runtime Evaluation

The runtime of each algorithm is compared based on the amount of time it takes for a model to finish executing 1000 episodes. From the figures, the runtime of each algorithm is listed in the table below. From the table, it can be observed that A3C is able to finish the training significantly faster than all the rest of the models, using only 1.46 seconds. A2C and REINFORCE finished execution using relatively similar time. While DQN finished training last, using 477.95 seconds.

Another phenomenon that can be observed in the above figures is the runtime of models are similar in shape with the episode rewards, except for A3C model.

This can be explained since A3C takes advantage of multi-threading performance while the rest of the models only make use of a single thread. Therefore, it is much easier for A3C to collect more samples in the limited time, thus faster training speed. The CPU usage display while training A2C and A3C is shown below in figure 4.5.

It is also possible that because single-threaded algorithms run for a significantly longer period of time, the CPU on the training device would decrease its clock speed, thus offering less performance.

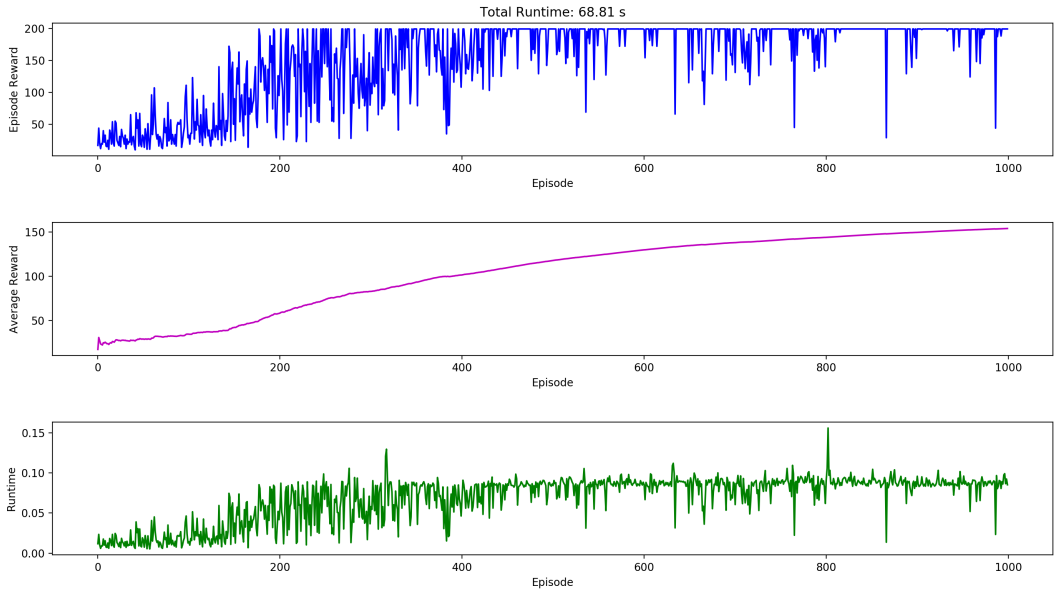

Figure 4.1 Reward received by REINFORCE agent

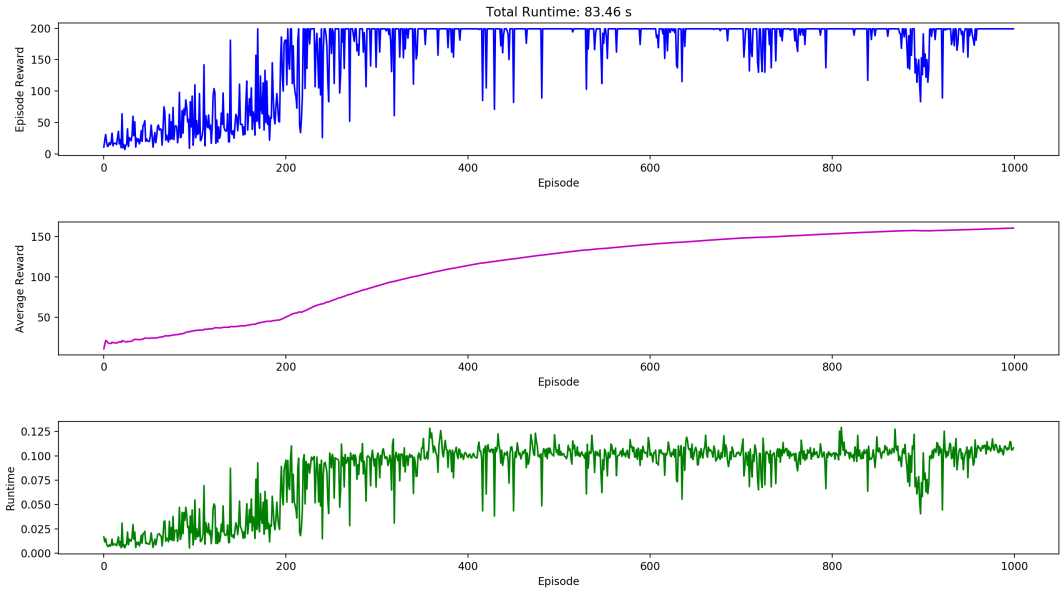

Figure 4.2 Reward received by A2C agent

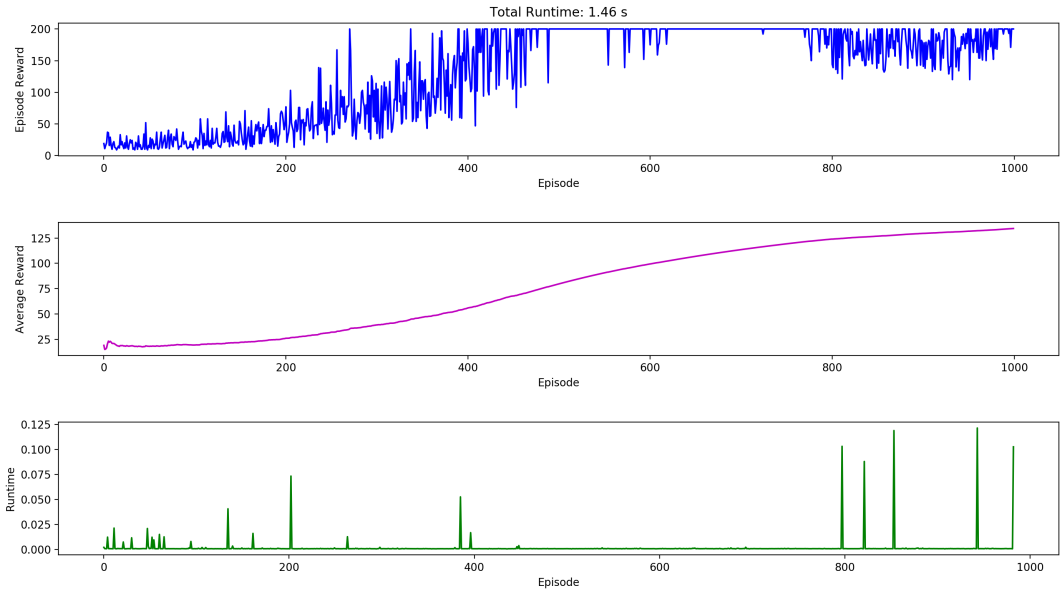

Figure 4.3 Reward received by A3C agent

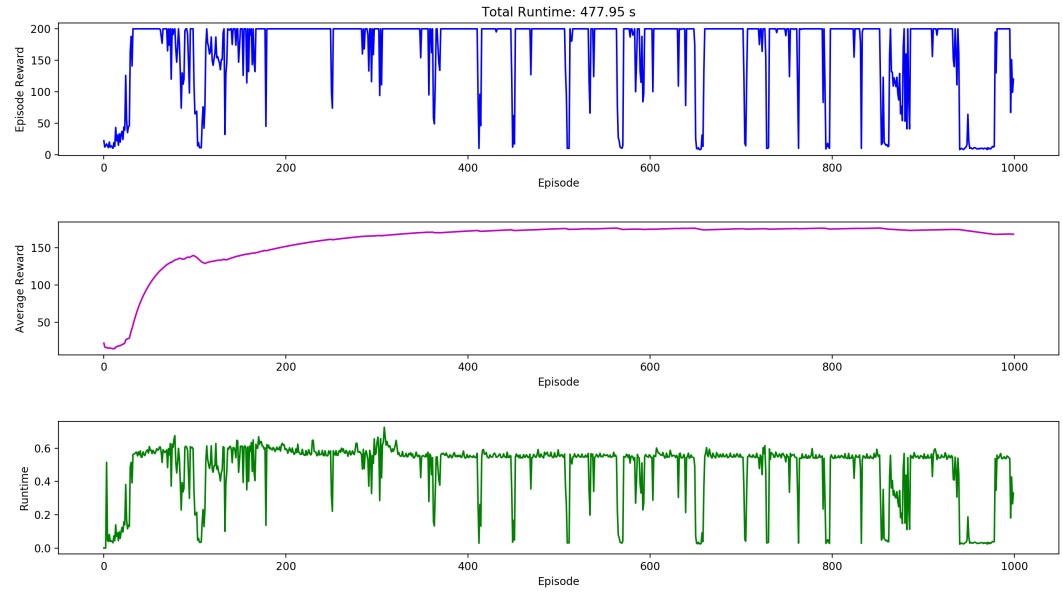

Figure 4.4 Reward received by DQN agent

Model	Runtime (s)
REINFORCE	68.81
Baseline	-
A2C	83.46
A3C	1.46
DQN	477.95

Table 4.1 Runtime of all models

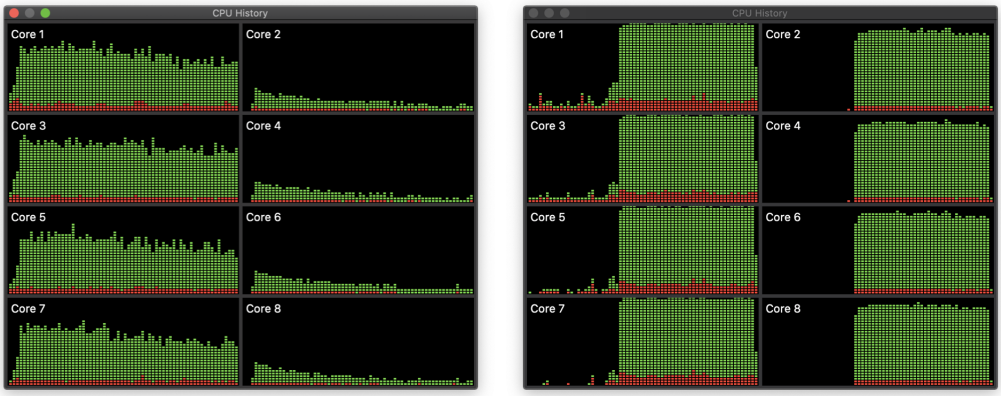

Figure 4.5 CPU usage of A2C(Left) and A3C(Right) during training

4.3 Reward Evaluation

Due to limited time and hardware resource, only 1000 episodes are trained for each algorithm during each test. From the above figures, it can be observed that although all algorithms converge quickly after the initial 200 episodes, non of the models could completely converge to true minimum within the limited episodes. However, the trend in each graph is still clearly noticeable. DQN has the fastest growth in reward, possibly due to its aggressive exploration strategy initially. REINFORCE has the second reward growth rate, then followed by A2C and A3C. It clearly demonstrates the strength of Deep-Q Networks. In an environment such as Cart Pole with discrete action space, a value network converges much quicker than a policy-based network with the same hyper-parameters.

But the problem remains about how to evaluate the performance of these models with reward when no model was able to completely solve the environment in the limited number of episodes, an alternative method to evaluate the ability of a model to gain rewards is by comparing the average reward acquired.

The average reward graphs show that all models are able to continuously improve over the process of training. However, DQN seems to have the fastest rate of improvement,

Model	Highest Average Reward
REINFORCE	154.6
Baseline	-
A2C	156.9
A3C	140.1
DQN	171.2

Table 4.2 Average Reward of all models

followed by REINFORCE, A2C, A3C, which improved at a similar pace. At the end of the training, it is also DQN that achieved the highest average reward. REINFORCE, A2C and A3C received similar results.

This is surprising since A3C should in theory have high sampling efficiency and achieve higher rewards over the same period of time. However, the fluctuation in episode reward values is noteworthy. While REINFORCE, A2C, A3C were all able to experience an upward trajectory and maintain the reward values relatively well, DQN struggles to maintain a high reward value after a certain number of episodes.

4.4 Stability Analysis

In the previous section, it was mentioned that although DQN was able to solve the game relatively quicker than all the other models, it struggled to maintain a high level of reward and periodically displays performance drop-offs. Having investigated further, this was proven to be a known issue in deep neural networks referred to as "catastrophic forgetting" [9]. This happens when the agent received an input that it has not encountered in a long time. In the case of DQN that utilises a replay buffer, when the entire buffer is filled with positive experiences, the agent will struggle when it arrives in a negative state again. Often DQN behaves poorly for a few episodes and returns to a optimal behaviour, but in some test cases the model also took more episodes to recover.

This can be confirmed by observing the gaps between the drop-offs. In the DQN performance graph (figure 4.4), the distance between each set of drop-offs is about 50 episodes, which is the size of the replay buffer used to generate the result.

In Cart Pole, This issue can be addressed by constantly keeping a number of samples of bad experiences in the replay buffer. Therefore, the agent will less likely to forget the action to take when arriving in a bad state again. This is known as prioritised experience replay[12].

Other models do not seem to suffer the same disadvantage. Because the rest of the models do not rely on replay buffer to sample from. The policy gradient methods simply sample from the policy distribution. This shows the advantage of on-policy algorithms over the off-policy

Model	Reward fluctuation
REINFORCE	180.3
Baseline	-
A2C	140.5
A3C	129.1
DQN	264.5

Table 4.3 Reward fluctuation in all models

models. To measure this variability in policy reward, a variance is calculated based on the reward values received after the first time the model reached 95% performance (190 reward) in an episode. This method disregards the initial episodes used for convergence of models and measures how well the models maintain a certain level of reward in the remaining of the training process. The result shows that DQN clearly has the highest variance, as it can be also observed from the performance graph. The variance of REINFORCE is the second highest. And the variance of A2C and A3C are the lowest of all.

This is unsurprising since DQN experiences very severe catastrophic forgetting, which resulted in a very unsteady training, therefore producing the least stable model. REINFORCE easily outperforms DQN since in stability since as it does not suffer the same issue.

4.5 Memory Usage

The figures below were generated by python memory profiler while the models ran the same configuration as the the above.

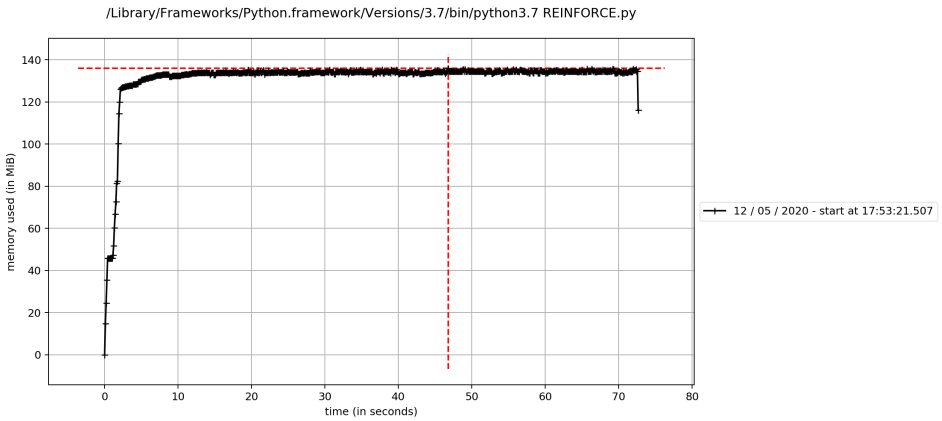

Figure 4.6 Memory Usage of REINFORCE

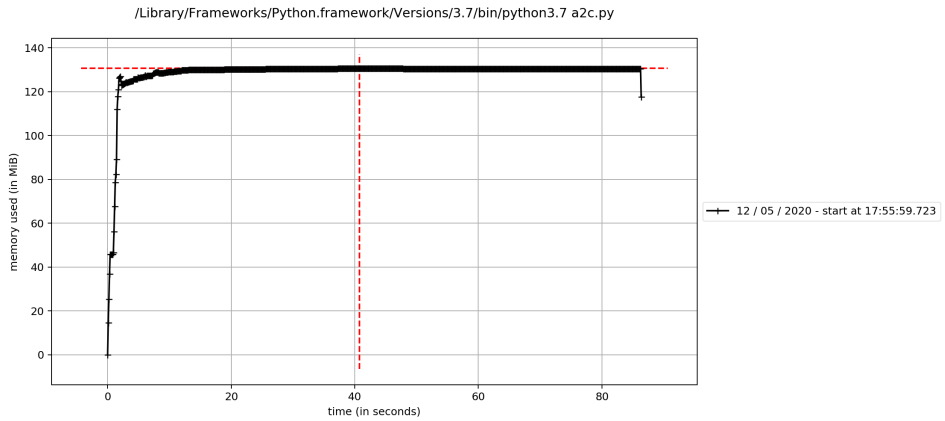

Figure 4.7 Memory Usage of A2C

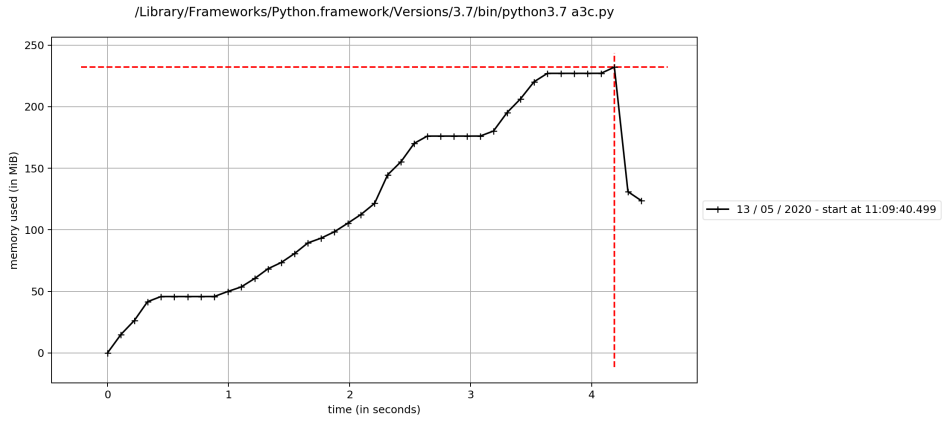

Figure 4.8 Memory Usage of A3C

It can be seen that all models occupies very different amount of memory while operating. In single-threaded models, A2C consumes the least memory (4.7), of around 130MB. REINFORCE (4.6) performs similarly, consuming about 135MB. Finally, DQN (4.9) is the most memory-intensive model, consuming just below 300MB. All graphs seem to level quickly and remain steady. Thus, it is reasonable to assume that these values are final and will remain constant regardless of the number of episode, or runtime. The memory usage of DQN however can be affected by the batch size.

In the multi-threaded A3C (4.8), the memory usage is noticeably higher, peaking at around 230MB. Although, A3C demonstrated very different behaviour compared to the other models. Instead of a quick increase and then stabilised memory usage, A3C continuous consumes more memory as the training progresses. This was thought to be caused by the

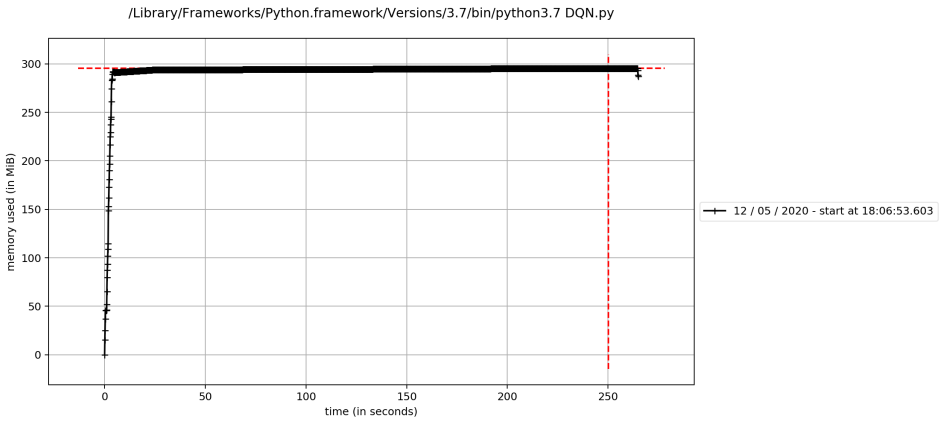

Figure 4.9 Memory Usage of DQN

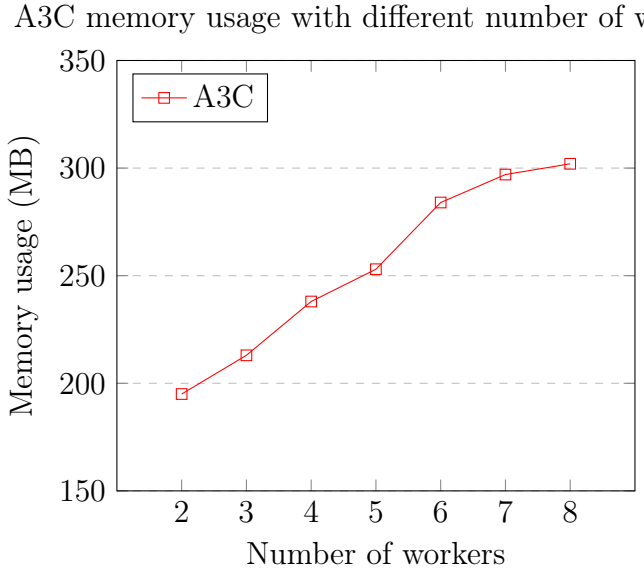

Figure 4.10 Memory usage of A3C when using different number of workers

number of cores used for training since `torch.multiprocessing` spawns processes that may requires individual memory allocation. By experimenting with different thread numbers, it can be observed from the graph (4.10) that configurations with fewer number of workers require less memory, whereas increasing the number of worker can result in a almost linear increase in memory usage. This shows although A3C is able to complete the same number of training in a much shorter time, it does have higher demand for memory bandwidth. This trade-off between computation power and memory bandwidth is an essential factor when the models run on different platforms.

Chapter Five

Discussion

It can be seen from the above evaluation that when solving the Cart Pole problem, Asynchronous Advantage Actor Critic (A3C) has the fastest runtime, DQN has the highest average reward, A3C has the highest stability and A2C least memory usage. However, as DQN suffers from "catastrophic forgetting", it has major disadvantages in stability. Therefore, the high reward results of DQN should not be considered as a contributing factor when choosing the best algorithm to solve cart pole or similar problems. Therefore overall, the actor-critic family algorithm appears to be the best algorithms in solving cart pole type problems with discrete action and observation spaces, having the advantage of high training speed, high sampling efficiency and high stability.

REINFORCE also performed reasonably well. The model demonstrated similar and faster runtime as A2C, a matching average reward value over 1000 training episodes, very close memory usage and slightly higher variance. The difference in performance between the basic REINFORCE and the actor-critic algorithms is much smaller than expected. Such relatively small margin could be the result of the testing environment since cart pole is a simple and discrete space. It is possible that the margins could expand with more training episodes.

Between the two actor-critic algorithms implemented in this paper, A3C seems to have a slight advantage specifically in the this cart pole environment. It provides much faster training speed despite consuming more device memory. However, it is worth noting that A3C is usually run on CPUs while A2C is often run on GPUs. Since all the models in this paper were tested on CPUs, it is unclear whether the results will still stand when the models are tested on CUDA-supported GPUs.

Given the typical number of cores in modern processors and size of of on-board memory, it is unlikely that any A3C configuration will experience memory shortage while training Cart Pole. Therefore, A3C should be the preferred method assuming the device is capable of performing multi-threading tasks and has enough memory. When the available memory is limited, A2C is the preferred method instead.

Chapter Six

Conclusion and Future Work

Through evaluating and comparing the speed, efficiency, stability and hardware usage of several popular reinforcement learning algorithms, it was shown that A3C was the best performing algorithm when solving the classic Cart Pole environment. The actor-critic family algorithms including A2C and A3C that attempt to combine the benefits of both policy gradient algorithms and Q-Learning algorithms demonstrated faster runtime, higher average reward value, higher stability over the other basic algorithms that only implements either on of the above categories. The result also highlighted the weakness at the same time. Policy gradient algorithms such as REINFORCE evidently suffered from higher variance, while Q-Learning methods such as DQN was not able to stabilise its performance due to inherit flaws such as "catastrophic forgetting".

Only DQN with memory replay was implemented in this paper. As the aim of the paper is to demonstrate the advantage of actor-critic algorithms, it is important to also verify other Q-Learning based algorithms such as DDQN in order to achieve a better comparison. It may also be beneficial to explore the other less common members of the actor-critic family such as the novel Proximal Policy Optimization(PPO), Deep Deterministic Policy Gradients(DDPG) and Soft Actor Critic(SAC). These algorithms represent the state-of-the-art of the actor-critic class and may demonstrate an even greater advantage over pure policy gradient or Q-Learning methods.

Furthermore, testing the current implementations of the models in different environments could provide more comprehensive results. Because PyTorch supports GPU acceleration, a properly optimised version of some models may receive some extent of performance boost. It may also be worthwhile to port the above implementations to TensorFlow as different machine learning framework may deliver different performance.

Finally, to investigate if the conclusion from this paper derived from the Cart Pole environment is universal and consistent. Other environments with discrete action and observation space need to be implemented. Although Cart Pole is a simple environment and only limited testings were carried out in this paper, the results may imply the advantages and characteristics of the actor-critic models in similar settings. Similar environments provided by OpenAI includes `Acrobot-v1` which has a state size of 6 and action space size of 3, as well as `MountainCar-v0`. It will be very interesting to observe if different environments will

lead to different conclusions as neural nets appear to perform more poorly with an increased number of dimensions. [13]

The combination of pure policy-based models and value-based models lead to the creation of a new class of hybrid models such as the actor-critic algorithms. These novel models prompts the possibility of creating better learning algorithms by mixing different types of the available implementations.

6.1 Mean Actor Critic

Mean actor critic is a recently proposed algorithm[1]. As the name suggests, the algorithm is also a member of the actor-critic family. Unlike the majority of the actor-critic models whose actors compute the sample means using only the sampled actions, Mean Actor Critic explicitly calculates the weighed average Q-value over all actions. This reduces the mean variance due to sampling to 0 as it does not directly utilise random sampling. This potentially leads to a few advantages. Using the calculated average may result in lower variance, making it more suitable for real world applications as real world data often contain high level of variance. The Q-values also acts as a proxy between the environment and the actor policy, decoupling the policy from the environment dynamics. The inclusion of Q-Learning brings the benefit of off-policy learning. It is considered as a discrete version of Deep Deterministic Policy Gradient (DDPG)[8].

This is a very promising and recent research as the algorithm seems to address some major weaknesses in state-of-the-art techniques. As there is no source code available for Mean Actor Critic, an original implementation and performance evaluation of this model could be very valuable.

APPENDICES

References

[1] Kavosh Asadi Cameron Allen. "Mean Actor Critic". In: (2017).

[2] DeepMins. *AlphaGo - Our approach*. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far#our_approach.

[3] Jimmy Ba Diederik P. Kingma. "Adam: A Method for Stochastic Optimization". In: (2014).

[4] Hector Geffner. *Planning with MDPs*. 2007. URL: http://www.inf.ed.ac.uk/teaching/courses/plan/slides/MDP-Planning-HG.pdf.

[5] Daniele Grattarola. "Deep Feature Extraction for Sample-Efficient Reinforcement Learning". PhD thesis. Oct. 2017. DOI: 10.13140/RG.2.2.30267.31527.

[6] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[7] Michael Herrmann. *On-policy and off-policy algorithms*. 2015. URL: http://www.inf.ed.ac.uk/teaching/courses/rl/slides15/rl05.pdf.

[8] Shimon Whiteson Kamil Ciosek. "Expected Policy Gradient". In: (2018).

[9] James Kirkpatrick. "Overcoming catastrophic forgetting in neural networks". In: (2017).

[10] Takehiko Nakama. "Comparisons of Single- and Multiple-Hidden-Layer Neural Networks". In: (2011).

[11] PyTorch.org. *MULTIPROCESSING BEST PRACTICES*. URL: https://pytorch.org/docs/stable/notes/multiprocessing.html.

[12] Tom Schaul. "Prioritized Experience Replay". In: (2016).

[13] Walter F. Bischof Terry Caelli. "Machine Learning and Image Interpretation". In: (1997).

[14] Ronald J. Williams. ""Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: 229–256 (1992).