A Study of RL algorithms in Tetris, Pong and Snake

BTech Project, IIT Dharwad

Anudeep Tubati, 170010039
Dept. of Computer Science and Engineering
anudeep.tumati99@gmail.com

Mentored by
Prof. Prabuchandran K. J.
Dept. of Computer Science and Engineering
prabukj@iitdh.ac.in

Abstract

This report highlights the need for a type of learning algorithms that can interact with the environment, rather than those that require precisely labelled data to perform well. These algorithms are known to reside in the scope of reinforcement learning. This report further analyses the implementations of several such algorithms in environments like Tetris, Pong, and Snake. It briefly critiques each one of them based on their performance and resource requirements. Finally, this report concludes with challenges faced with the implementations, their mitigations and interesting insights derived from them.

Introduction

In the past few years, the field of AI has seen immense innovation in areas like computer vision and natural language processing. These innovations have given rise to several daily applications like, reading text from a photo (Google Lens), translating with near perfection and suggesting movies that fit one's choice. However, 2 straightforward problems ship with supervised learning,

a) Creating a dataset-specific to a problem might prove to be a costly affair, as it costs about $25 for a human to just classify 1000 images[1]
b) The long-term effect of a particular decision can only be ascertained by simulating all the possible paths until the end, which is an infeasible task for complex problems. Therefore, there might not be a "correct" label for every situation.

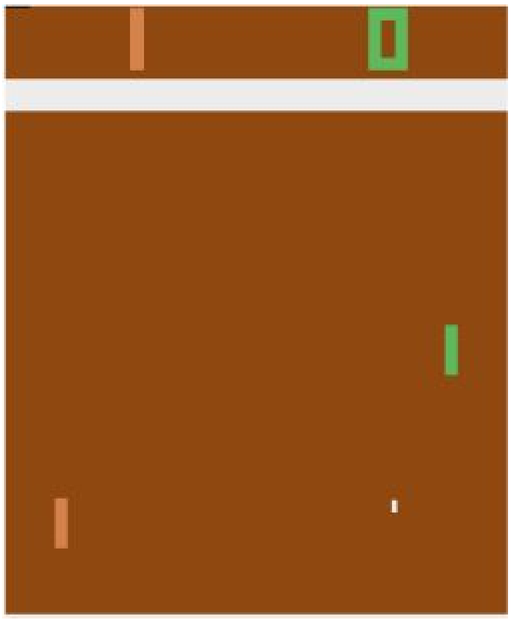
For example, while playing Chess, some actions are desirable (protecting one's high-valued pieces, sacrificing a low-valued piece to capture a high-valued piece of the opponent) and some actions aren't desirable. To ascertain that a particular action is correct, all possible game pathways have to be simulated. It is correct only if the game results in a victory.
Hence, having a dataset is not always an option. Without a dataset, supervised learning is almost set courseless. In such problems, the model can learn through interaction with the environment, to better solve more general and complex problems. That precise paradigm is dealt with by reinforcement learning, and the same is the focus of this project. It implements and analyses RL algorithms by utilising video-game environments like Tetris, Pong, and Snake.

RL problems essentially consist of an agent and an environment. The agent can observe the environment, partially or fully, and needs to take actions to obtain the highest reward in the long-term. The same construct is further elaborated in the following sections.
The implemented game-playing bots are based on various RL algorithms, involving evolutionary (gradient-free), Q-learning, planning-based, policy gradient and actor-critic methods.

Game Environments & Related Work

The 3 video-game environments majorly used in the project are elaborated in the following subsections. For each of them, there are a few previously-done works that serve as an inspiration. They are briefly discussed as well. It's worthwhile to note that 2 other games have been used too, but not as much as the aforementioned ones. Those games will be discussed in the next section.

1. Pong


PongDeterministic-v0 from OpenAI Gym[2]

Pong is a 2-player game (popular Atari 2600 game) wherein each player controls a paddle and their objective is to hit the ball in a way that the opponent isn't able to hit it back. **Each player gains 1 point for making the opponent lose the ball once.**

Default Attribute	Value
Observation	(210 x 160 x 3) shaped RGB image
Reward Scheme	+1 if the opponent loses the ball -1 if the player loses the ball
Termination Condition	Either of the players reaches 21 points
Objective	Keep hitting the ball to make the

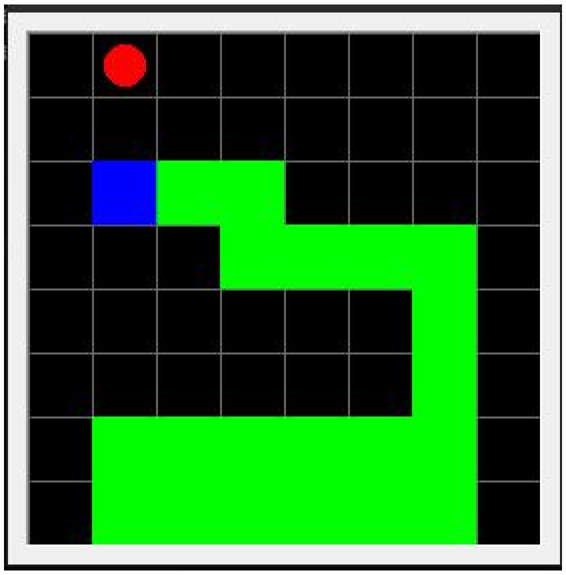
	opponent miss the ball. Get 21 such points.

Andrej Karpathy[3] uses a neural network described in Table 1 to develop a bot for this game. The bot performs slightly better than the default opponent, indicating that the model can learn significantly. A similar architecture is used to develop Pong-playing bots in the next section.

Table 1. NN Architecture used by Karpathy

Layer	Dimension	Activation
Input	6400	-
Hidden Layer 1	200	ReLU
Output	1	Sigmoid

2. Snake



Gym Snake by Telmo Correa[4]

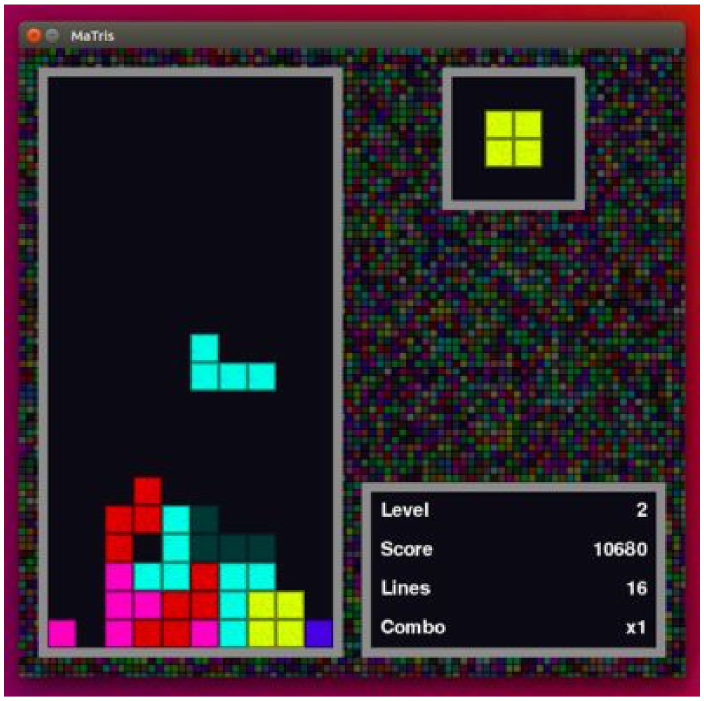
Snake is yet another popular game that involves only 1 player. The player controls a snake in a grid world. The objective is to make the snake eat apples that randomly spawn on the grid. The snake grows every time it eats an apple and it has to avoid running into itself or running into the boundary, as they result in game termination.

Default Attribute	**Value**
Observation	(S x S x 4) shaped image, where S is grid size Each grid can contain the snake's head, snake's body, food or be empty
Reward Scheme	+1 for eating food -1 for crashing
Termination Condition	Snake running into itself or the boundary
Objective	Eat all apples till snake covers the entire grid

Jack of Some[5] uses an actor-critic method to complete the game of snake in a 6x6 grid. This materialises a base for understanding and implementing actor-critic methods in the next section.

3. Tetris

a. Full Tetris



MaTris by Uglemat[6]

Tetris[7] is an iconic game from the 80s. It takes place in a grid environment that involves a set of blocks that fall one-by-one, in random order. The objective is to avoid running out of space in the grid box and overflowing the Tetris blocks out of the grid. The blocks fall by a unit at each timestep and they can be rotated to be arranged in a "tight" manner, without any holes.

Line clear - A line clear happens when a horizontal row is full of Tetris blocks without any gaps. It results in the whole row vanishing and all the blocks above falling by one unit each. This is the main technique used to survive longer in the game.

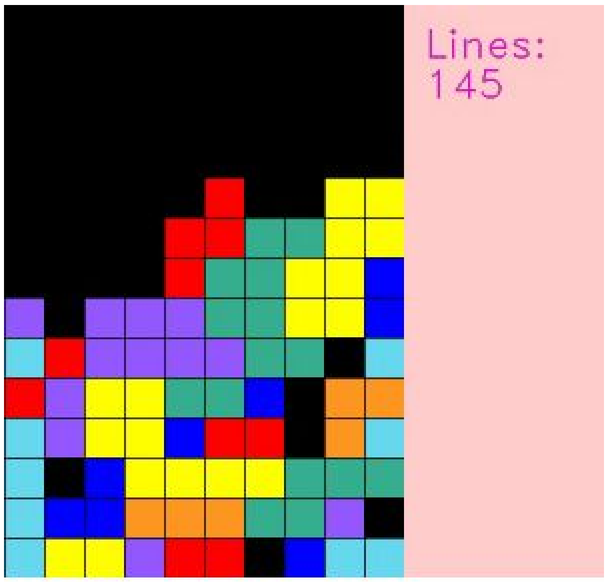
Default Attribute	**Value**

Observation	(H x W x 3) shaped image
	H = Height of grid
	W = Width of grid
	All pixels containing a block are 255 and rest are 0

Reward Scheme	No default scheme (to be shown in the Algorithms section)
Termination Condition	Tetris block overflowing out of the grid
Objective	Clear the most number of lines (or) survive the longest time

Daniel Gordon[8] uses an actor-critic method (GPU A3C) to design an efficient bot that aims to clear 4 lines at once. The same bot is analysed and tested in the next section.

b. Simple Tetris



Simple Tetris by uvipen[9]

A game that is very similar to Full Tetris, except for the action space. Instead of a block falling down every timestep and being rotated, it waits in an imaginary space above the environment grid for an action that specifies the row to place it in and the number of times to rotate it. The action is in the form (x, rotations) where *rotations* specifies the number of times to rotate the block and *x* is the index of the column to

place the rotated Tetris block (in a way that the left-most piece of the Tetris block is in *x-column*).
Such a straightforward version allows faster training and works better with evolutionary algorithms.

Default Attribute	**Value**
Observation	(20 x 10 x 1) shaped image All pixels containing a block are 255 and rest are 0
Reward Scheme	No default scheme (to be shown in the Algorithms section)
Termination Condition	Tetris block overflowing out of the grid
Objective	Clear the most number of lines (or) survive the longest time

C. Thiery *et al.*[10] use the Cross Entropy Method to play Simple Tetris. Though it is a straightforward and intuitive algorithm, it plays extremely well for such a simple algorithm. It is elaborated in the next section, along with an actor-critic bot as well.

Algorithms

It is worthwhile to note that this report assumes significant knowledge about Markov Decision Processes[11]. It is also briefly discussed in the following subsection.

Markov Decision Process

MDP is a mathematical framework for representing a problem and its characteristics in a way that is intuitive to formulate for reinforcement learning problems.
MDP consists of 4 sets, namely, States, Actions, Rewards and Transition Probabilities.

- States - The state the agent exists in, at a given point of time
- Actions - The set of all possible actions an agent can take (only a subset of Actions might be possible for a given state)
- Transition Probabilities - The probability mapping from a (state, action) pair to a “next state”
- Reward - A function mapping from (state, action, next state) to real numbers.

After taking an action *a* in a state *s*, the transition probability mapping provides a stochastic distribution of possible next states. In some problems, it might be deterministic. That is, after taking an action *a* in a state *s*, the agent surely goes to a particular next state *s’*.

The algorithms discussed in the following subsections use certain keywords stemming from the MDP formulation. They are elaborated in the following list.

a. Episode - The simulation of the agent from the start of the game to termination of the game. It is defined by the trajectory of (*s*, *a*, *s’*, *r*) at each time step
b. Discount Rate (γ) - The rate by that future rewards are deemed to contribute to the present value. A reward *r* after *k* steps is only worth $\gamma^k r$

c. Cumulative Reward - The sum of rewards at each time step, till the end of the episode. ("Discounted Cumulative Reward", if discounting is applied)

$$G_t = r_{t+1} + r_{t+2} + \cdots = \sum_{k=0}^{\infty} r_{t+k+1}$$

Eqn 1. Cumulative Reward

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Eqn 2. Discounted Cumulative Reward

d. Policy - A probability mapping that produces the distribution of actions given a state *s*

e. V-value - The expected cumulative reward given a current state *s*, obtained by following a policy π

f. Q-value - The expected cumulative reward given a current state *s* and action *a* and then following a policy π

Snake

Naïve Baseline

Input Vector is transformed into the following vector (Note that the algorithm also keeps track of the snake's direction)

Danger on Left BOOL	Danger on Straight BOOL	Danger on Right BOOL	Food Left BOOL	Food Straight BOOL	Food Right BOOL	Food Back BOOL

Examples - [0, 0, 0, 1, 0, 0, 0], [0, 1, 0, 1, 1, 0, 0], [1, 1, 1, 0, 1, 1, 0]

```
Algorithm 1: Naïve Baseline for Snake
    Input: $observation$ (7D vector)
    Result: Best action
    $A \leftarrow$ set of all actions;
    $possibleActions \leftarrow A$ - $\{a \mid a$ results in an immediate crash, $a \in A\}$;
    $desiredActions \leftarrow \{\}$;
    if $food$ $is$ $on$ $left$ then
        $desiredActions \leftarrow desiredActions \cup \{left\}$
    else if $food$ $is$ $on$ $right$ then
        $desiredActions \leftarrow desiredActions \cup \{right\}$
    end
    if $food$ $is$ $straight$ then
        $desiredActions \leftarrow desiredActions \cup \{straight\}$
    else if $food$ $is$ $back$ then
        $desiredActions \leftarrow desiredActions \cup \{left\}$
        $desiredActions \leftarrow desiredActions \cup \{right\}$
    end
    if $possibleActions = \emptyset$ then
        $bestAction \sim A$;
    end
    if $desiredActions \cap possibleActions = \emptyset$ then
        $bestAction \sim possibleActions$
    else
        $bestAction \sim desiredActions \cap possibleActions$
    end
```

This baseline is a one-step lookahead algorithm that gives priority to avoid crashing. Out of all the moves that will not result in an immediate crash, it prefers a move that takes it closer to the apple, if any.
Such a straightforward algorithm serves as a good baseline while comparing with other learning-based algorithms.

Table 2. Performance of Naive Agent on different-sized envs (200 games)

Env	Avg	Max	Min	Std
8x8	13.14	31	1	5.2
4x4	7.72	13	1	2.8

Note - 13 is the maximum possible score on a 4x4 grid, as the snake already starts with a size of 3. Similarly, 59 is the maximum score on an 8x8 grid.

Table-based Q-learning using TD(0)

```
Algorithm 2: Table-based Q-learning using TD(0)
Input: $\alpha$ (learning rate), $\gamma$ (discount rate), $\epsilon$ (exploration rate)
Result: Table Mapping from (s, a) to their Q-values
Initialise: $Table(s,a) \leftarrow 0 \; \forall s \in S, a \in A$
while train do
    Initialise s;
    while s is non-terminal do
        $a \sim A$, with probability $\epsilon$
        $a \leftarrow \text{argmax}_a\, Table(s,a)$, otherwise
        Take action $a$, observe $r, s'$
        $Table(s,a) \leftarrow (1-\alpha)Table(s,a) + \alpha[r + \gamma \max_a Table(s',a)]$
        $s \leftarrow s'$
    end
    Update $\epsilon$ according to schedule
end
```

This algorithm maintains a table that maps from all possible (state, action) pairs to a Q-value. Though initialised to zeroes, the Q-values are updated with a Temporal-Difference update[12]. The model also gives significant importance to exploration, controlled by the ϵ value.

It works due to the possible exploration that allows the model to register rewards (Q-value) for each state/action pair. As more state/action pairs are explored, their Q-values initially shape towards the immediate reward, as the next state Q-values are close to zero as well. However, after sufficient exploration, the model starts to follow a high Q-value path, known as exploitation. That way, the Q-value of one state is indirectly propagated back to its previous states and the model can gradually converge to near-optimal Q-values for each state/action pair.

Table 3. Performance of Q-learning Agent on different-sized envs (200 games)

Env	Avg	Max	Min	Std
8x8	14.45	30	3	5.8
4x4	8.11	13	1	2.6

However, the table construct makes it infeasible for problems with very high state and action spaces. This is due to 2 reasons,

a) High memory usage - More state/action pairs result in very high table entries that may not be supported by the available memory
b) Non-generalisability - Since the table has a defined mapping for each state/action pair, its performance is random on un-explored states. Due to a huge state space, exploring the whole space might not be feasible.

Upper Confidence Trees

UCT is a planning algorithm that constructs a game tree from the current state and picks the action resulting in the highest score state. It is a similar strategy to best-first search. It iteratively expands the tree, considering both exploitation and exploration.

The value of each state is based on the random simulations conducted from that state as they prove the desirability of the state. For eg., a state with more possibilities for a positive reward is valued higher than a state that may run into troubles more often.

Algorithm 1. The UCT Algorithm, partial pseudocode from Cameron Browne's lecture[13]

```
function UCTSEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP($v_l, \Delta$)
    return $a$(BESTCHILD($v_0, 0$))

function TREEPOLICY($v$)
    while $v$ is nonterminal do
        if $v$ not fully expanded then
            return EXPAND($v$)
        else
            $v \leftarrow$ BESTCHILD($v, Cp$)
    return $v$
```

```
function EXPAND($v$)
    choose $a \in$ untried actions from $A(s(v))$
    add a new child $v'$ to $v$
        with $s(v') = f(s(v), a)$
        and $a(v') = a$
    return $v'$

function BACKUP($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of $v$
```

```
function BESTCHILD($v$, $c$)
    $node \leftarrow \underset{v' \in children\ of\ v}{\operatorname{argmax}} \frac{Q(v')}{N(v')} + c\sqrt{\frac{log(N(v))}{N(v')}}$
    return $node$
```

Eqn 3. The scoring function of UCT, with exploitation and exploration components

```
function DEFAULTPOLICY($s$)
    Initialise $G = 0$
    while $Simulations < max\_playouts$ do
        Initialise $k = 0$
        while $s$ is non-terminal do
            choose $a \in A$ uniformly at random
            $s, r \leftarrow f(s, a)$
            $G \leftarrow G + \gamma^k r$
            $k \leftarrow k + 1$
        end
    end
    return $G$
```

Note - The function *f* in the algorithm refers to the step in the environment, *s(v)* refers to the state represented by node *v* and *a(v)* refers to the action taken by the parent of *v* to select *v*

The main component that enables the algorithm is the simulations. Unlike other AI algorithms like A* search, UCT does not completely depend on a heuristic function to assess a game state. It estimates the value by conducting simulations from the state. Hence, it provides a more "true" estimate, whereas a heuristic function is hand-made. The exploration component helps the algorithm to be fair to all states as well.

Table 4. Results of UCT on different-sized envs (200 games)

Env	Avg	Max	Min	Std
4x4 Time per move = 2s Playouts = 1	11.95	13	7	1.6
8x8 Time per move = 2s Playouts = 1	7.23	17	0	3.9

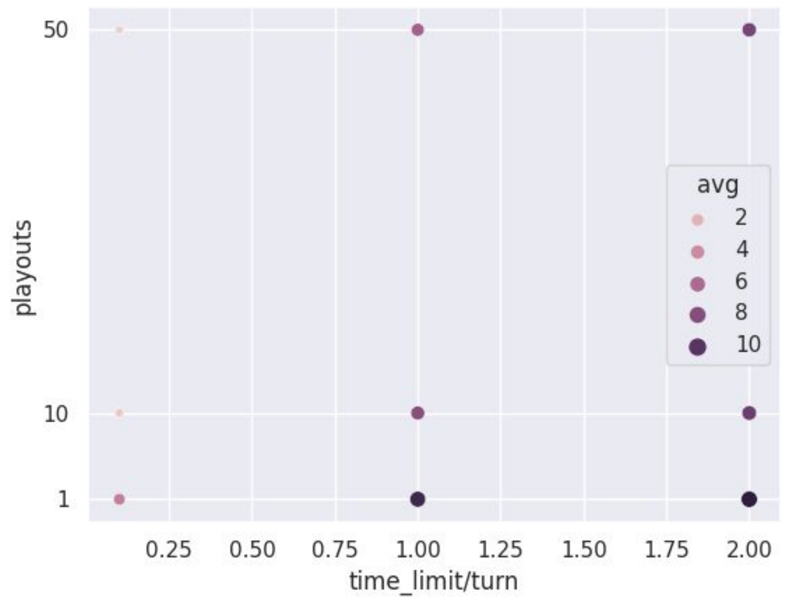


Fig. Avg Score of UCT agent on a 4x4 grid with various playouts and time limits

It is evident from the graph that more time makes the agent play much better. Whereas, increasing the number of simulations proves worse, as the agent does not get enough time to search deeper. In the same way, increasing both simulations and time limit results in a better performance.

It gets infeasible mainly owing to its compute-intensive simulations. The accuracy of the value obtained for a particular state is related to the number of simulations held, from that state. Therefore, as the environment complexity increases, the computation complexity skyrockets. The same is evident from Table 4 as the agent performs well in a 4x4 grid but fails with the same parameters in an 8x8 grid. Additionally, the algorithm needs 2 seconds for each step to consistently complete the 4x4 snake game. 2 seconds/turn is a huge amount of time, considering the simplicity of the game.

It is interesting to note that the AlphaGo family[14] uses MCTS at its core. MCTS is a more general version of the UCT. To cut down on the compute-intensive simulations, it uses neural networks to calculate the value of a state and the action probabilities. Those neural networks are trained on a massive dataset consisting of self-play moves. Hence, this serves as a motivation to explore neural network-based RL algorithms.

Deep Q-Networks

This algorithm[15] is indeed very similar to the table-based Q-learning method, the only major difference being the usage of a neural network instead of a table. It also employs an experience replay buffer that stores experiences to later learn from a shuffled subset of them. Therefore, it does not learn every step like the table-based method, rather maintains a batch size. Doing so gives better convergence to the neural networks, as a batch size of 1 generally proves to be a noisy update.

DQNs mitigate the 2 flaws discussed for the latter method by replacing the table with a neural network. This allows the algorithm to generalize better for unexplored states and it does not require a huge amount of memory as compared to tables. Learning from shuffled previous experiences also mitigates the problem of the model learning sequential patterns in the game.

The disadvantages of using DQNs are as follows,

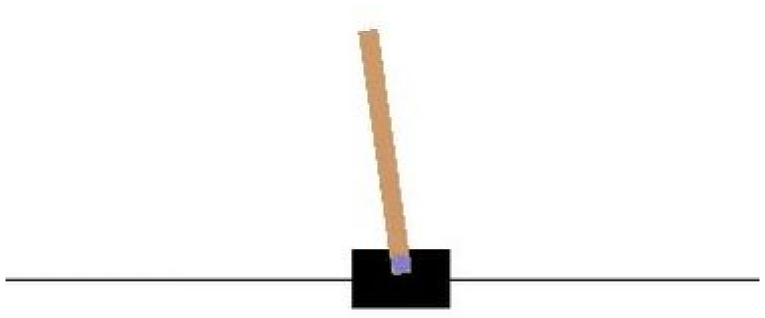
a) Value for each Action - Predicting a Q-value for each state/action pair is not viable with a huge action space or even in a continuous action space
b) Stochastic Environments - DQNs choose actions with the highest Q-value, given a state. Hence, they cannot model a stochastic policy for a state. That does not perform so well in environments having implicit stochasticity, as the model cannot learn it.
c) Exploration schedule is generally hand-made. Therefore, it may induce some bias. A related insight is explored in the *Challenges and Insights* section.

Therefore, algorithms that directly deal with the policy are analysed in the next subsections.

The DQN consisting of 2 hidden layers, directly fed with the game image as it is just 64-D (4x4x4), did not learn anything significant after 12 hrs of training. The possible problem is highlighted in the following paragraph.
The Snake game has a fundamental issue that makes it harder for the algorithms to learn. It involves sparse positive rewards as the food is randomly spawned in a huge space. Hence, a beginner agent needs to tremendously explore before even finding out that eating food is the objective. Additionally, completing the game requires the agent to play in a zig-zag fashion[16]. Finding out such a style of play again requires even more exploration and planning. This calls for analysing much simpler games, as various agents can be tested for learning in a faster manner. Therefore, the following subsections deal with games like CartPole and Pong.

CartPole

Description

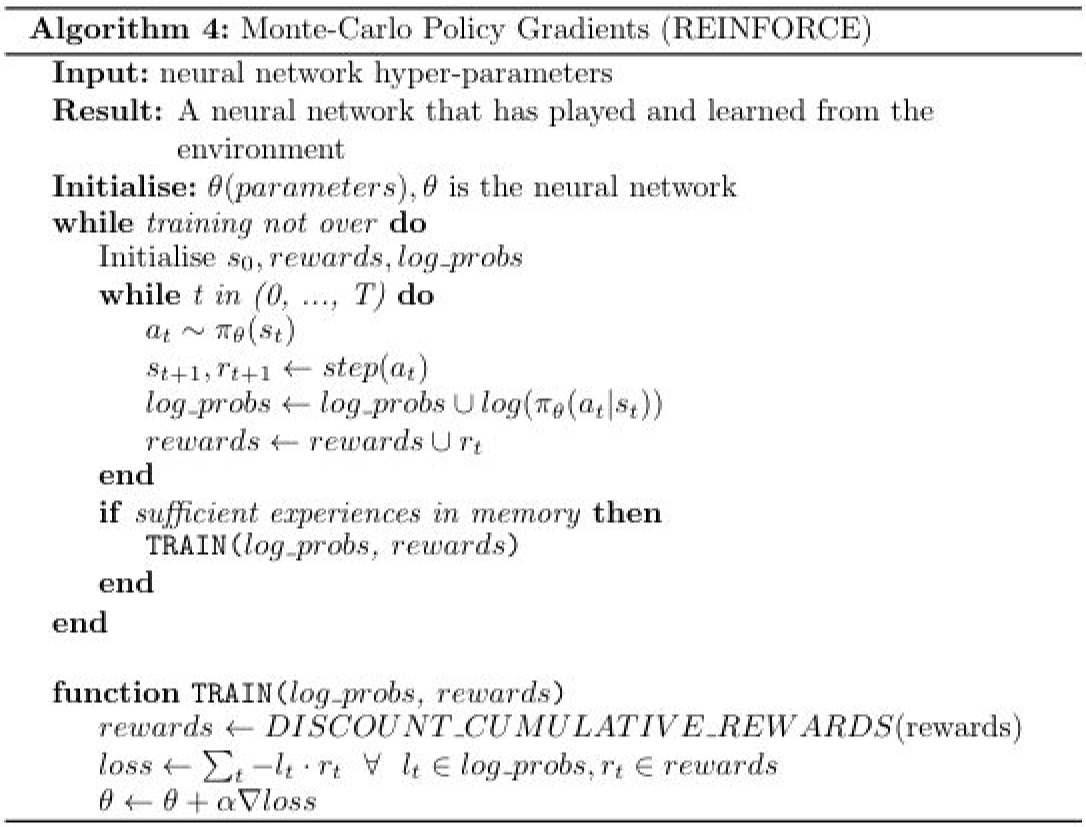


CartPole-v0 from OpenAI Gym[2]

Default Attribute	Value
Observation	4 values [position of cart, the velocity of the cart, angle of the pole, the rotation rate of pole]
Reward Scheme	+1 every timestep
Termination Condition	Pole moves more than 15° from vertical (or) Cart moves more than 2.4 units from the start position (or) 200 timesteps are elapsed
Objective	Survive the longest (Average more than 195 points in the past 100 games to “solve” the environment)

Monte-Carlo Policy Gradients (also known as *REINFORCE*)

As opposed to learning Q-values of the states, this algorithm learns a probability distribution for actions for a given state. The gradient update for the model uses log-probability rather than the actual probability, as it makes the proof more intuitive to understand and doesn't affect the theory in any way since the log function is monotonic. The proof follows from the Policy Gradient Theorem[17].



```
Algorithm 4: Monte-Carlo Policy Gradients (REINFORCE)
Input: neural network hyper-parameters
Result: A neural network that has played and learned from the
        environment
Initialise: $\theta(parameters), \theta$ is the neural network
while training not over do
    Initialise $s_0, rewards, log\_probs$
    while t in (0, ..., T) do
        $a_t \sim \pi_\theta(s_t)$
        $s_{t+1}, r_{t+1} \leftarrow step(a_t)$
        $log\_probs \leftarrow log\_probs \cup log(\pi_\theta(a_t|s_t))$
        $rewards \leftarrow rewards \cup r_t$
    end
    if sufficient experiences in memory then
        TRAIN(log_probs, rewards)
    end
end

function TRAIN(log_probs, rewards)
    $rewards \leftarrow DISCOUNT\_CUMULATIVE\_REWARDS$(rewards)
    $loss \leftarrow \sum_t -l_t \cdot r_t \quad \forall \quad l_t \in log\_probs, r_t \in rewards$
    $\theta \leftarrow \theta + \alpha \nabla loss$
```

Table 5. NN Architecture

Layer	Dimension	Activation
Input	4	-
Hidden Layer 1	16	ReLU
Output	2	Softmax

Table 6. NN Properties

Optimiser	RMSProp (lr = 0.01)
Loss	$-\sum_{t=0}^{T} log(\pi(a_t\|s_t)) * G_t$
Batch	Train every 1 episode
Gamma	0.99

The gradient, that gives the direction to move to increase the probability of an action taken, is multiplied by the cumulative reward after taking the action. This multiplication helps the model learn “good” and “bad” actions, as the cumulative reward is higher in the former case.

The only major problem that slows down learning is that the cumulative rewards used to adjust the gradients are usually of high variance. This is because even a small deviation in the trajectory of an episode might result in very different cumulative rewards. For eg., an action results in the game termination within 10 steps whereas another action might have resulted in the game ending in 50 steps. The cumulative reward in both cases is starkly different. Hence, learning from such a high variance loss function produces noisy gradients.

Advantage Actor-Critic

Actor-Critic algorithms combine the previously described methods of Q-learning and Policy Gradients. Therefore, they are aimed to bring in the best of both worlds. The actor is the model that produces a probability distribution of actions given a state. The critic is the model that learns the V-values of states and it can be used to calculate Advantage-value of an action. By doing so, the actor’s loss function, that was previously based on high variance cumulative reward, is now based on a more stable

Advantage-value predicted by the critic. Hence, it learns faster[18] owing to the less-noisy gradients.

Table 7. NN Properties

Actor Optimiser	RMSProp (lr = 0.01)
Critic Optimiser	RMSProp (lr = 0.1)
Actor Loss	$-\sum_{t=0}^{T} log(\pi(a_t \vert s_t)) \; * \; A_t$ $A_t \; = \; r_t + \gamma v(s_{t+1}) \; - \; v(s_t)$
Critic Loss	$-\sum_{t=0}^{T} (\; v(s_t) \; - \; (r_t + \gamma v(s_{t+1})) \;)^2$ $v(s_{t+1}) \; = \; 0 \; if \; t = T$
Batch	Train every 1 episode
Gamma	0.99

The table listed below shows the same. It measures the number of games taken by MCPG and A2C to solve CartPole, that is, achieve an average of 195 in the last 100 games. 15 such evaluations are made in total.
Each evaluation starts with a randomly initialised agent and the agent needs to train till it solves the game.

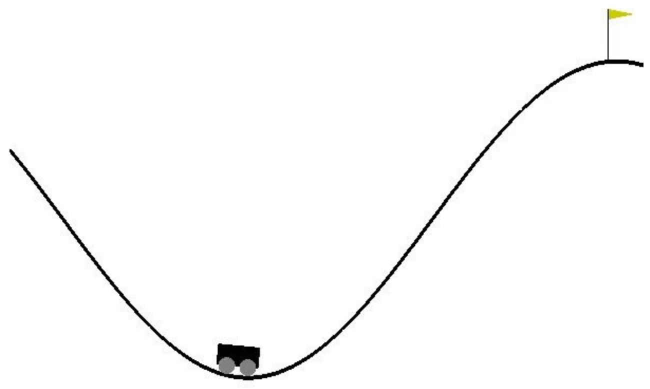
Table 8. Games took to solve, both the agents on CartPole (15 evaluations)

Agent	Avg	Max	Min	Std
MCPG	576.2	966	242	212.1
A2C	343.13	706	149	136.15

It is evident that A2C learns more consistently, as the average and standard deviation of the number of episodes taken to solve the game is significantly lesser than those of MCPG.

MountainCar

Description



MountainCarContinuous-v0 from OpenAI Gym[2]

A car is to be pushed to the goal on right by oscillating it and gaining momentum, as the car is not powerful enough to cross the slope just by its acceleration.

Default Attribute	**Value**
Observation	2 values, [x coordinate of car, the velocity of the car]
Reward Scheme	-1 every timestep +90 on reaching the goal
Termination Condition	Car reaches goal (or) 1000 timesteps are elapsed
Objective	Reach the goal located at the x coordinate +0.06 (Average more than -700 points in the past 100 games to “solve” the environment)

As discussed earlier, value-based methods can’t be used in an environment with a continuous action space. MCC is a simple, yet continuous game environment. The actor network produces a distribution that is used to sample an action. This particular implementation uses a Gaussian distribution for actions. Therefore, the actor network outputs *O1* and *O2* such that *O1* is the mean of the distribution and e^{O2} is the variance of the distribution.

Table 9. NN Architecture

Layer	**Dimension**	**Activation**
Input	2	-
Hidden Layer 1	16	ReLU
Output	2	Linear

Table 10. NN Properties

Optimiser	RMSProp (lr = 0.003)
Loss	$- \sum_{t=0}^{T} log(\pi(a_t \vert s_t)) \ * \ G_t$
Batch	Train every 1 episode
Gamma	0.99

Table 11. Games took to solve, both the agents on MCC (15 evaluations)

Agent	**Avg**	**Max**	**Min**	**Std**
MCPG	11866.67	36,681	3,029	9282.2
A2C	10539.11	33,734	1,523	8856.1

Pong

Input Preprocessing

The observation from Pong is preprocessed and flattened into a 6000-D vector. The difference of the current 6000-D vector and the previous one is returned as input to the neural network.

Monte-Carlo Policy Gradients

A similar implementation as mentioned in the previous subsection, with a few hyperparameter changes.

Table 12. NN Architecture

Layer	Dimension	Activation
Input	6000	-
Hidden Layer 1	200	ReLU
Output	1	Sigmoid

Table 13. NN Properties

Optimiser	RMSProp (lr = 0.001, L2 Regularisation Coeff=0.99)
Loss	$-\sum_{t=0}^{T} log(\pi(a_t \mid s_t)) \, * \, G_t$
Batch	Train every 10 episodes
Gamma	0.99

As proved by Karpathy[3], this bot can perform slightly better than the default opponent and achieves an average of +3 (beats the opponent

21-18 on an average). It trains for about 8 hrs on a CPU machine (AWS c5.large).

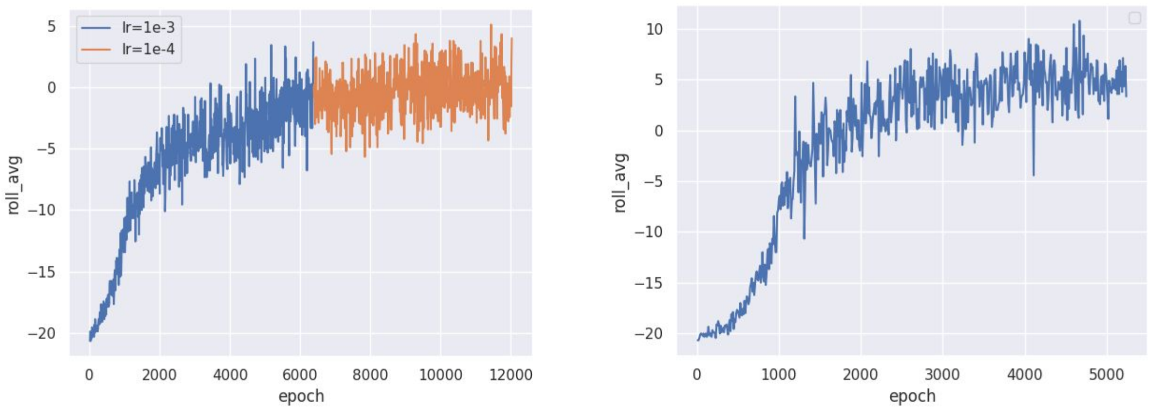
Advantage Actor-Critic

The similar implementation as mentioned in the previous subsection, with a few hyperparameter changes. Uses another neural network with the same architecture (except for the output as it contains only 1 unit) for the critic.

Table 14. NN Properties

Actor Optimiser	RMSProp (lr = 0.001, L2 Regularisation Coeff=0.99) lr = 0.0005 after 1000 epochs
Critic Optimiser	RMSProp (lr = 0.001, L2 Regularisation Coeff=0.99)
Actor Loss	Epoch < 1000 $-\sum_{t=0}^{T} log(\pi(a_t \vert s_t)) \ * \ G_t$ Epoch >= 1000 $-\sum_{t=0}^{T} log(\pi(a_t \vert s_t)) \ * \ A_t$ $A_t \ = \ G_t \ - \ v(s_t)$
Critic Loss	Epoch < 1000 $-\sum_{t=0}^{T}(\ v(s_t) \ - \ G_t \ )^2$ Epoch >= 1000 $-\sum_{t=0}^{T}(\ v(s_t) \ - \ (r_t + \gamma v(s_{t+1})) \ )^2$ $v(s_{t+1}) \ = \ 0 \ if \ t = T$
Batch	Train every 10 episodes
Gamma	0.99

Its performance reinforces the statement that A2C learns faster than the REINFORCE algorithm. The model is able to achieve an average of +6 points (beats the default opponent 21-15 on an average) and it learns to do so within 50% of the epochs taken by MCPG. The same can be seen in the following learning graph of both the models.



MCPG vs. A2C Pong Learning Graph (Rolling Average of 10 eps vs. Epoch)

Full Tetris (Analysis on Pre-trained model)

The pre-trained agent discussed in the Related Work section is based on the GPU Asynchronous Advantage Actor-Critic algorithm[19]. It uses a massive CNN of 1.3 million parameters to achieve efficient performance on the Full Tetris environment. It also requires a week of training on an Nvidia Titan X[8]. The A3C algorithm[20] is a distributed version of the A2C algorithm wherein several workers contribute to the gradient. It works better than A2C in some cases. However, it mostly fails due to its nature of making multiple copies of the same model, as that makes it inefficient in some situations. The GA3C algorithm works more efficiently, as it uses only one copy of a model and uses queueing algorithms to infer and train the model.

The model's output layers (logits for action distribution and V-values) are retrained, in two different ways, to check the training of the model and the resources it requires. Firstly, by perturbing the weights a little and then completely re-initialising them. Additionally, the whole network is re-initialised for training as well.

The model with perturbed weights learns to perform as well as the original network within 3000 episodes of training. Whereas, the model with re-initialised output layers requires about 15000 episodes to do the same. This verifies that the algorithm works and gives an insight into the resources required to train such a massive network.

Table 15. Re-training on Full Tetris

Model	Episodes to Match Original Performance	Resources Consumed
Output Layer Weights Perturbed	3000	3 hrs CPU (AWS t2.medium)
Output Layer Weights Re-initialised	15000	17 hrs CPU (AWS t2.medium)
Total Network Re-initialised	N/A	6 days on a 1080 Ti without any improvement

Since the resource requirements were not satisfiable, a smaller model was trained on a simpler version of Tetris.

Simple Tetris

Cross Entropy Method

Unlike the previously mentioned methods that involve learning values or modelling the environment, CEM is a gradient-free evolutionary method that learns to score each state to clear most lines. The score is calculated by taking the inner product of a weight vector and extracted features from a state. The optimal weight vector is assumed to belong to a distribution G. This algorithm tries to iteratively improve a current distribution F_t from a family of distributions (Gaussian in this case), to be closest to G.

```
**Algorithm 5:** Cross Entropy Method
  **Input:** $mean$, $var$, $Z$ (Noise), $N$ (Population Size), $T$ (Epochs)
  **Result:** A Gaussian distribution closest to the optimal weight vector
              distribution and the best performing weight vector
  $F_0$ = Gaussian($mean$, $var$);
  **while** $t$ in (0, ..., T) **do**
      $w_k \sim F_t$ for k in (0, ..., N);
      $L_k$ := play($w_k$) for k in (0, ..., N);
      $elite\_weights$ := Top $rho$ performing weight vectors;
      $mean$ := avg($elite\_weights$);
      $var$ := variance($elite\_weights$) + Z;
      $F_{t+1}$ := Gaussian(mean, var);
  **end**
```

It works very well for the case of Simple Tetris. This is mainly because of the features it uses to evaluate a state. The features used by this implementation are the following,

a) cleared_lines - The additional lines that will be cleared upon entering the particular next state from the current state
b) holes - The number of cells that have at least one square covered above it, in the same column

c) bumpiness - Sum of absolute height difference between each pair of adjacent columns
d) total_height - Sum of the height of each column

Intuitively, these features give a good idea of a state. Therefore, CEM, as an evolutionary algorithm, simply does it by selecting weight vectors from the top-performing in the population and narrowing down upon the distribution. It is interesting to note that there is some noise added in every step to avoid reaching a local optimum.

Though the algorithm seems to perform very well for its complexity, it is not as scalable as gradient-based algorithms due to the following reason. There is no future planning involved, as the algorithm selects action based on a greedy objective to maximise next state score. Hence, the algorithm is only as good as the scoring weight vector, that is in turn relying on hand-made features. Therefore, it paves way for bias.

It achieved a maximum score of 617,112 and an average of 214,704 (50 games) after training for 2 days on 4 CPUs.

Advantage Actor-Critic and Supervised Learning

The A2C algorithm is similar to the one described in the previous subsections. However, the input, architectural, and reward system changes are implemented to better suit this game. As using a CNN proved to be extremely resource-intensive in the Full Tetris case, this model uses a much simpler neural network. The input vector is (w+6)-dimensional with w being the width of the grid environment.

<u>Reward Structure</u>

Every step = +0.1

Every line cleared = +2

Game over = -10

Table 16. NN Architecture

Layer	Dimension	Activation
Input	W + 6	-
Hidden Layer 1	64	ReLU
Hidden Layer 2	64	ReLU
Logits (Output)	len(possible_actions)	Linear
Q-values (Output)	len(possible_actions)	Linear

Table 17. NN Properties

Optimiser	RMSProp (lr = 0.0005)
Loss	Policy Loss $- \sum_{t=0}^{T} log(\pi(a_t \vert s_t)) \ * \ A_t$ $A_t = r_t + \gamma v(s_{t+1}) - v(s_t)$ $v(s_t) = \sum_i \pi(a_i \vert s_t) \cdot q(s_t, \ a_i)$ Value Loss $- \sum_{t=0}^{T} (\ q(s_t, a_t) - (r_t + \gamma v(s_{t+1})) \ )^2$ $v(s_{t+1}) = 0 \ if \ t = T$
Batch	Train every 1 episode
Gamma	0.99

Table 18. Input to the Neural Network, (W+6)D vector

First W values W is the Width of the Grid	$\frac{Height \ of \ each \ column \ in \ the \ grid}{Height \ of \ Grid}$
Next 6 values	One-hot encoding of the 7 Tetris blocks

Two neural networks are trained on differently sized environments, 20x6 and 20x10. In addition to A2C that learns from a reward system, another method of training is employed as well. The CEM agent supervises the neural network. This is known as bootstrapping. It lets the neural network quickly learn the fundamentals that can optionally be trained further with a reward system to avoid the inherent bias from the supervision.

Table 19. Scores (Lines cleared) of agents on Simple Tetris (50 games)

Model	**Avg**	**Max**	**Min**	**Training**
CEM	214704.05	617,112	14,291	4 days CPU (4 threads, Azure Standard_D4_v4)
CEM Bootstrap NN (20x10) Policy loss only	60.72	566	8	2 days CPU (AWS c5.large)
A2C (20x10)	5.41	27	0	1 day CPU (AWS c5.large)
A2C (20x6)	11.35	82	3	1 day CPU (AWS c5.large)

It is interesting to note that the A2C agents learn to some extent, given their input (just heights and one-hot vector). However, larger networks are not tested due to resource constraints.

Challenges and Insights

Invalid Actions

In environments like Snake and Pong, each action from the Action Space (A) can be taken at every step. Though some of such actions may result in the termination of the game due to undesirable play, the action can still perform such actions.

Now, consider a Simple Tetris environment of size 20x10. An "I" block spans 4 columns without any rotation. Hence, it can only be placed in column 6 at max. Placing it in column 7 or higher causes it to horizontally overflow, which is not allowed. Therefore, there are some invalid actions in each step, depending on the current state.

In huge action spaces, like the one in Simple Tetris, this leads to a bit of a problem. If a valid action is taken at time=t and it is found to produce a negative advantage, the loss motivates the model to decrease the probability of such an action. Hence, the model ultimately increases the probabilities of all other actions, as the probabilities have to sum to 1. This increases the probabilities of invalid actions too. Therefore, the model does not learn efficiently.

To solve this, an approach known as Invalid Action Masking[21] is used. This approach masks the invalid actions in such a way that their gradients are zeroed out while backpropagating. Therefore, only the probabilities of valid actions are changed.

Reward Shaping

While designing a reward system, it's extremely important to keep in mind the final appearance of it to the model, that is, its values after being discounted (if applicable). A reward scheme of (+1 every step, +5 every line

clear, -50 game end) and gamma=0.9 for Tetris seems plausible from a human perspective. The same reward forces the A2C agent to learn nothing, owing to its massive penalty that roughly penalises the last 30-45 moves. Since the beginner agent hardly plays 10 moves and that results in all moves being penalised. Therefore, the agent learns that even line clears are negative actions.
In Snake, a reward scheme of (+0.1 every step) seems like it will help the snake survive longer but it makes the agent learn to take longer-than-required paths to reach the apple. That may be the desired behaviour in the later stages of the game, but it leads to poor learning in the start phases.

It is best to keep the reward scheme as general as supported by the learning algorithm. Doing so helps reduce bias from the hand-made parameters.

Architecture and Learning Parameters

Deciding upon an architecture and its hyperparameters is a substantial task by itself, as the algorithm may be super effective but a poor implementation proves it otherwise. For small input sizes, a rule of thumb can be to try 2^x or 2^{x+1} units in one hidden layer, where x is the dimension of the input vector. The reason being, an x dimension column vector has 2^x possible column combinations. Each combination can be thought of as a feature.
CartPole and MCC implementations in this report follow the same.
It is better to start with a larger than expected architecture with a small step size. If training that for a significant amount of time proves that the model learns, the step size and architecture can be further altered to speed it up.

The MCPG Pong bot has a high bias (not very generalisable), as it scores in only selected situations (video in GitHub repo). In such cases, enlarging the network often helps. Whereas, if the network has a lot of variance, training further with a small step size helps it stabilise.

Resource Scarcity

One of the main challenges with RL algorithms is procuring the resources required to debug and train them. A model has to be tested several times, before it is set up for training for long durations, to avoid wasting resources. Due to such constraints, this project did not analyse real-world problems or larger models like AlphaGo.
The main resources used for this project are as follows,

I. AWS Educate Pack ($100)
II. Azure Student Pack
III. DeepNote (forever free CPU)
IV. Linode Free Trial ($100)
V. DigitalOcean Free Trial ($100)
VI. Vast.ai (Affordable GPUs)

The following subsections consist of selected intriguing discussions on various paradigms. They can be considered as a motivation to explore more advanced and intensive algorithms.

Humans vs. RL Agents

The Pong agent significantly learns to beat the opponent. Although, it still seems to behave quirkily. It gains points in the same way almost every time (gameplay in GitHub repo) and loses points in a certain way too. Q-learning and Policy Gradient methods come under the branch of Model-Free algorithms. They do not model the environment. Instead, they only aim to learn the actions that maximise reward. To do so, they must get lucky in the beginning to even notice a reward, which is then followed by the model increasing Q-value or probability of a high-rewarding action.

Now, consider the characteristics of human learning. Humans model the dynamics of the environment. They try to predict the future state from a current action to foresee rewards. Model-free agents can generally beat humans in games like Pong, owing to the simplicity of such environments. It is fairly easy to stumble upon a reward and the model-free algorithms encash that.

However, in games like Chess and Shogi, it is far too difficult to just “stumble” upon a victory. More meticulous planning is required to perform well in those games. Model-based algorithms are known to perform well in such environments, as they learn as humans do.

Consider an analogy to 2 robots in a maze. First one has very limited memory. It can only learn what to do in a particular situation. Whereas, the second one has more memory. It can learn the resultant state of a particular action in the current state. The first robot will have to explore almost every state and action before knowing to solve the maze. However, the second robot can construct a top-view of the maze by modelling it. Therefore, it can solve the maze by a planning algorithm once the model is accurate enough. This is the major difference between model-free and model-based algorithms.

DQN Infeasibility in Snake

As discussed earlier, the explicit exploration schedule in DQN learning makes it quite inefficient in some situations. One of such scenarios is the Snake game. When the game is about 50% complete, the snake has to start playing in a zig-zag manner to fully complete the game. Every step has danger in approximately one direction out of 3 (left, right or straight). Considering the exploration factor to be ε, the probability of ending the game within *k* steps due to an exploration mistake roughly is,

$$\sum_{i=0}^{k}(1-\varepsilon)^{i-1} \cdot \varepsilon \cdot (1 \div 3)$$

It translates to a value of 0.14 for even a minute value of epsilon=0.025, within 20 steps (0.25 for k=50). Even 20 steps is a small number in comparison with the number of steps required to complete the game after achieving the zig-zag style. Therefore, it is evident that training such an agent is infeasible.

The Emphasis of Exploration in UCT

The exploration factor plays an important role in UCT. It gives higher value to states whose parents have been visited often and lower value to those who have been visited often themselves. Without any exploration, the agent performs very poorly. This is because just one simulation is not enough to guarantee a good or bad value. However, if the exploration is much more than required, the algorithm only explores the breadth and doesn't go deep on any node. That results in less accurate estimates, and in turn, poorer performance.

Table 20. Performance of UCT with different exploration factors (200 games)

Time per move = 1s, Playouts = 1

Env	Avg	Max	Min	Std
4x4 c = 1 (ideal)	11.95	13	7	1.6
4x4 c = 0	0.37	5	0	0.9
4x4 c = 10	3.63	13	0	3.6

Q-learning vs. Policy Gradients

As discussed in the DQN disadvantages, Q-learning faces challenges of non-stochastic policy, the cardinality of action space and bias from exploration.

However, they are better than policy gradient methods in some areas too. Firstly, it is easier to implement as the crux is a TD learning equation as

compared to policy gradients that have an involved derivation. Secondly, they are faster in some sense, as they bootstrap from the rewards. This makes them have a higher data efficiency (measured as the improvement in performance per sample consumed).
Every problem might have certain characteristics that make DQNs more suitable than policy gradients. It's best to analyse the problem in detail before fixating on a particular family of algorithms. Actor-Critic methods sort of give the best of both worlds.

Snake Naïve Agent vs. Q-learning Agent

The evaluated policy of Naive Agent and Q-learning agent match in most scenarios, except the ones with more than 1 possible action, as the Naive Agent picks randomly in such scenarios. This proves that the Q-learning agent learns the best possible policy, given just a 1-step lookahead input. It even beats the Naive Agent as it learns to better handle multiple-action-possible situations and maximises future reward.

Cross Entropy Method

The final learned weights are of the 4 features are

[9.42, -64.86, -12.06, -0.79]

Evidently, major emphasis is placed on the hole count. This owes to the fact that holes block a line until the blocks on top of the holes are cleared. Even bumpiness is considered to be important as a bumpy frontier may lead to more holes. The cleared lines are important as well since they're the main tool used to survive longer. However, aggregate height seems to be of little to no importance. This may owe to the agg. height automatically decreasing when lines are cleared and increasing by default at every step. Therefore it can be considered as a function of lines cleared.

Conclusion

After analysing and implementing RL algorithms from various branches, this project concludes that each algorithm ships with its leverages and liabilities. The same is highlighted in the following table,

Table 21. Perks and Liabilities of RL algorithms

Algorithm	Perks	Liabilities
Cross Entropy Method (Evolutionary)	Simplicity	Data efficiency and bias
Policy Gradients	Stochasticity, action space and guaranteed convergence	Data efficiency
Q-learning	Simplicity and data efficiency	Non-stochasticity and limited action space
Actor-Critic	A fine mix of both the above	Non-generalisable to huge, analytical problems (Model-free nature)
Monte-Carlo Tree Search	Perform well in complex environments (with sufficient time)	High resource requirements, needs perfect simulator

Video-games are a suitable play-area for exploring RL algorithms, as they offer games that vary in complexity and also have a robust simulator. They can be used as testbeds before applying algorithms in the real-world. Just using a particular algorithm might not achieve desirable results in real-world applications, which is the ultimate goal. It is best to analyse the algorithms and applications in a way to compose a panacea for unique applications.

References

1. Data Labeling Pricing, Google Cloud, https://cloud.google.com/ai-platform/data-labeling/pricing, accessed 03-12-2020
2. Greg Brockman, Vicki Cheung *et al.*, “OpenAI Gym”, 2016, arXiv:1606.01540
3. Andrej Karpathy, “Deep Reinforcement Learning: Pong from Pixels”, 2016, GitHub. http://karpathy.github.io/2016/05/31/rl/, accessed 03-12-2020
4. Correa, Telmo, “Snake Environment for OpenAI Gym”, 2019, GitHub, https://github.com/telmo-correa/gym-snake
5. Jack of Some, “Neural Network Learns to Play Snake using Deep Reinforcement Learning”, YouTube, https://www.youtube.com/watch?v=i0Pkgtbh1xw
6. Uglemat, “MaTris”, 2018, GitHub, https://github.com/Uglemat/MaTris
7. Tetris, Wikipedia, https://en.wikipedia.org/wiki/Tetris, accessed 04-12-2020
8. Daniel Gordon, “Learning to Perform a Tetris with Deep Reinforcement Learning”, 2018, GitHub Pages, https://danielgordon10.github.io/pdfs/deep_rl_for_tetris.pdf
9. uvipen, “Tetris deep Q-learning pytorch”, 2020, GitHub, https://github.com/uvipen/Tetris-deep-Q-learning-pytorch
10. Christophe Thiery, Bruno Scherrer. ”Improvements on Learning Tetris with Cross Entropy”. International Computer Games Association Journal, ICGA, 2009, 32 inria-00418930
11. Richard Sutton, Andrew Barto. “Reinforcement Learning, An Introduction. 2nd Edition”, Chapter 3, Finite Markov Decision Processes, http://incompleteideas.net/book/bookdraft2018jan1.pdf
12. Richard Sutton, Andrew Barto. “Reinforcement Learning, An Introduction. 2nd Edition”, Chapter 6, Temporal-Difference Learning
13. Cameron Browne, “Monte Carlo Tree Search”, 2012, Lecture at Imperial College London, http://ccg.doc.gold.ac.uk/ccg_old/teaching/ludic_computing/ludic16.pdf
14. Silver, D., Huang, A., Maddison, C. et al., ”Mastering the game of Go with deep neural networks and tree search”. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961
15. Volodymyr Mnih et al., “Playing Atari with Deep Reinforcement Learning”, 2013, CoRR. arXiv:1312.5602
16. Omar Ghatasheh, “The Perfect Snake Game”, YouTube, https://www.youtube.com/watch?v=kZr8sR9Gwag

17. Richard Sutton, Andrew Barto. “Reinforcement Learning, An Introduction. 2nd Edition”, Chapter 13, Policy Gradient Methods
18. Daniel Seita, “Going Deeper Into Reinforcement Learning: Fundamentals of Policy Gradients”, 2017, GitHub. https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/, accessed 05-12-2020
19. Mohammad Babaeizadeh et al., “GA3C: GPU-based A3C for Deep Reinforcement Learning”, 2016, CoRR. arXiv:1611.06256
20. Volodymyr Mnih et al., “Asynchronous Methods for Deep Reinforcement Learning”, 2016, CoRR. arXiv:1602.01783
21. Costa Huang, “A Closer Look at Invalid Action Masking in Policy Gradient Algorithms”, 2020, Website-Blog, https://costa.sh/blog-a-closer-look-at-invalid-action-masking-in-policy-gradient-algorithms.html