Perpetually Playing Physics

by

Chris Beeler

A thesis submitted to the School of
Graduate and Postdoctoral Studies in
partial fulfillment of the requirements for
the degree of

Master of Science

in

Modelling and Computational Science

Faculty of Science
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
August 2019

Thesis Examination Information

Submitted by: **Chris Beeler**

Master of Science in **Modelling and Computational Science**

Thesis Title: Perpetually Playing Physics

An oral defense of this thesis took place on July 29, 2019 in front of the following examining committee:

Examining Committee:

Chair of Exammining Commitee	Dr. Lennaert van Veen
Research Supervisor	Dr. Lennaert van Veen
Research Co-Supervisor	Dr. Isaac Tamblyn
Examining Committee Member	Dr. Hendrick de Haan
Thesis Examiner	Dr. Ken Pu
Thesis Examiner Affiliation	University of Ontario Institute of Technology

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

"Creativity is the ability
to introduce order into the
randomness of nature."

- Eric Hoffer

Abstract

Here we discuss ideas of reinforcement learning and the importance of various aspects of it. We show how reinforcement learning methods based on genetic algorithms can be used to reproduce thermodynamic cycles without prior knowledge of physics. To show this, we introduce an environment that models a simple heat engine. With this, we are able to optimize a neural network based policy to maximize the thermal efficiency for different cases. Using a series of restricted action sets in this environment, our policy was able to reproduce three known thermodynamic cycles. We also introduce an irreversible action, creating an unknown thermodynamic cycle that the agent helps discover, showing how reinforcement learning can find solutions to new problems. We also discuss shortcomings of the method used, the importance of understanding the class of problem being handled, and why some methods can only be used for certain classes of problems.

Author's Declaration

Chris Beeler

[illegible]

Statement of Contributions

The *Cart-Pole*, *Mountain-Car*, and *Pendulum* environments were used from the open-source package *gym* by OpenAI. The *Direct Control Grid-World* environment is not original to this work as it is one of the most common environments throughout Reinforcement Learning (RL), however the code base for it, and *Probabilistic Control Grid-World*, used in this work was written by Chris Beeler. All heat engine environments are original to this work and were primary written by Chris Beeler. The environments can each be found at https://github.com/CLEANit/heatenginegym and parts of the results in Chapter 4 can be found on the arXiv [1]. The *Water* environment is original to this work but was primarily written by NRC staff with the previously existing DFTB+ program used as the basis for all molecular simulation calculations. Lastly, all figures in each chapter and all results presented from Chapter 3 to the end of this thesis were produced or illustrated by the work of Chris Beeler.

Acknowledgements

I would like to start by thanking my advisor, Dr. Isaac Tamblyn, for giving me the opportunity and resources, through the National Research Council of Canada, to pursue reinforcement learning as a successful research topic. My fellow group members who started before me, Kyle Mills and Kevin Ryczko, for helping me learn and understand the basics of coding and machine learning when I first started and continuing to help long afterwards. My fellow group member who started with me, Rory Coles, for all the problems that he helped solve during the many nights, weekends, and holidays in the lab. Lastly I would like to thank all my fellow graduate students at UOIT who helped accommodate me during all my travels when working between Ottawa and Oshawa.

Contents

List of Figures

List of Tables

Abbreviations

A2C	Advantageous Actor Critic.
ANN	Artificial Neural Networks.
B3LYP	Becke, Three-parameter, Lee-Yang-Parr.
CPU	Central Processing Unit.
DFT	Density Functional Theory.
DFTB	Density Functional based Tight Binding.
DQN	Deep Q-Network.
ES	Evolutionary Strategies.
GA	Genetic Algorithm.
GGA	Generalized Gradient Approximation.
HMM	Hidden Markov Model.
LSDA	Local Spin Density Approximation.
MDP	Markov Decision Process.
MP	Møller-Plesset Perturbation Theory.
MP2	Second Order Møller-Plesset Perturbation Theory.
NEAT	NeuroEvolution of Augmenting Topology.
PBC	Periodic Boundary Condition.
POMDP	Partially Observable Markov Decision Process.

QM9 Quantum Machine 9.

RL Reinforcement Learning.

RS Reyleigh-Scrödinger Perturbation Theory.

TB Tight Binding.

TD Temporal Difference.

Chapter 1

Introduction

Recently machine learning with neural networks has been applied to solve difficult problems in physics and chemistry. Deep neural networks have been used to predict the nearest-neighbor energy of the 4×4 and 8×8 Ising models [2], recognize local atomic structure in high-resolution transmission electron microscopy images [3], and classify the symmetry of simulated crystal structures [4]. More recently RL [5] is starting to be applied to other problems in science. In chemistry, RL has been used for ribonucleic acid design [6], the shaping of polymer molecular weight distributions [7], and optimizing chemical reactions [8]. In physics, RL has been used for controlling rigid bodies with directed fluids [9], preparing physical systems with desired quantum states [10], and discovering complete quantum-error-correction strategies [11].

While RL has become more widely spread in the recent years, it is not a new topic and has had many successes in the past several decades with less advanced methods. One of the earliest successful examples of using machine learning for playing games was Samuel's checkers player [12, 13] first developed in 1952 but not demonstrated live until 1956. This algorithm performs a tree search for each move and uses a linear function approximation of the value function to evaluate each possible path in order to select which action to take. Samuel used a discounting technique for the tree searches where paths further along the tree search were weighted less than immediate paths and found this method essential. Instead of using a reward, Samuel's algorithm was designed to maximize the piece advantage feature which is how many pieces the player has relative to the opponent. Samuel's checkers player was able to play *checkers* at an above-average level, and the his 1967 improvement was even better but still not at an expert level. While this works for *checkers*, it does not generalize well for other problems but it is similar in many ways to the algorithm used by Tesauro. Between 1992 and 2002, Tesauro used various Temporal Difference (TD) methods for agents to learn how to play the popular game *backgammon* [14–17]. These studies showed that the TD-Gammon algorithms can beat world-champion level *backgammon* players by playing against itself without any prior knowledge of the game. However games like *chess* and *go* are still too complex to solve with these algorithms.

Many problems that have been too difficult to solve using traditional RL algorithms, like the ones mentioned earlier, have been made solvable by using a deep neural network [18] as the policy. Methods like Q-learning [19–23] with Deep Q-Network (DQN) [24] and Advantageous Actor Critic (A2C) [25] use neural networks, traditionally with a gradient descent learning algorithm, to approximate the value function that is used to estimate the future reward expected given a state-action pair. These methods have been used to achieve super-human level control [26] in Atari 2600 games [27] and more advanced Partially Observable Markov Decision Process (POMDP) [28] such as *Doom* [29, 30], *Rogue's Dungeon* [31], *Toribash* [32, 33] and more recently the very complex real-time strategy game *Starcraft II* [34, 35]. A2C has had other successes in more physically relevant tasks such as navigating random 3D mazes [25] and learning humanoid locomotive control [36] in MuJoCo [37] physics environments. Other interesting successes in RL include learning to play robot soccer [38, 39], mastering the physics-based games in the OpenAI gym [40, 41], and defeating world champions in *go* [42, 43]. These gradient descent algorithms update the neural network-based policy using back-propagation, where the loss is calculated from the expected and true reward for a given state-action pair. This is very useful for games that have feedback at each state, however it becomes a much more difficult problem for games with a sparse reward scheme. When games only return rewards between very large intervals of time, the policy must be able to estimate future reward very well in order to follow a path of little to no reward that leads to an eventual higher reward. There comes an even larger problem when not only is the reward sparse, but it cannot easily be attributed to a specific step, known as the credit-assignment problem [19]. This can be a significant problem in RL because an agent cannot possibly learn the optimal policy for a given environment if the reward cannot be accurately assigned. One class of methods that handles the credit-assignment problem quite well are Evolutionary Strategies (ES).

ES solve the credit-assignment problem by assigning a score to a policy using a fitness function. The fitness function assesses a policy's entire episode, or episodes, apposed to each step individually. This solves the credit-assignment problem as there is no need to assign rewards to specific steps taken by the agent. Another advantage of using ES is that they are extremely parralizable allowing for an efficient computational speed-up. Such et. al. [44] compared the ES used by Salimans et. al. [45] and a GA [46, 47], that is a subset of ES, with DQN, A2C, and a random search on several Atari games. While no one RL method in this study was better than all others, the GA was the only method that was consistently better than the random search on all 13 environments. The GA also out-performed DQN, A2C, and ES each on more than half of the environments, and was the best performing method for more environments all other methods. While the GA and ES performed quite well on average in comparison to DQN and A2C, the most important advantage of these methods are the lack of back-propagation. Without this need these methods require much less computational resources and reach maximum reward in a fraction of the time. DQN took approximately 7 to 10 days on average to reach maximum reward, A2C took approximately 4 days, while ES and GA took approximately 1 to 4 hours. Not only do the ES and GA out-perform the DQN and A2C methods on several environments, they reach this high performance much faster.

ES are not just used for optimizing the parameters in a policy but can also be used in optimizing the architecture of a policy as shown by Peng et. al. [48]. The NeuroEvolution of Augmenting Topology (NEAT) [49] algorithm described in that study starts with a the simplest neural network which directly links the input to the output. At each iteration the possible changes a network's architecture can undergo are adding a new node or adding connections between previously unconnected nodes. The goal with for this method is that the simplest network possible to solve the problem is found. In this study the NEAT algorithm was mainly used for learning important features of the state, while other back-propagation based methods were mainly used to optimize the policy network for mapping features to actions, although tests were done using NEAT for the policy network and the back-propagation based methods for feature learning. However in the cases when NEAT and back-propagation methods were combined, NEAT was always used for feature learning and the back-propagation based methods were always used for the policy network. When NEAT was used for both the feature learning and policy networks, it was the worst performing case for 5 out of 6 Atari games and still not quite comparable to the best methods tested in the sixth case. However when NEAT and a back-propagation based method were both used, it was the best performing method in 4 Atari games, comparable to the best performing methods in the fifth case, and only noticeably worse in the remaining case.

While it is commonly known in the RL community that no one method is better than every other one for all cases, this study shows that in some cases the combination of methods can also be beneficial.

While RL has been applied to board games, and more so recently video games almost to the point of exhaustion, there is still not a large amount of applications to scientifically relevant problems. With the recent breakthroughs in RL, many tasks that were once deemed too difficult to solve for anything other than the human mind have now been solved, it would seem it is unreasonable to put a limit on what can be done. In this study we aim to broaden the scope of what type of problems RL can be used for as well as show how it can be used for more than just playing games or solving problems that have been solved many times in the past.

Chapter 2

Reinforcement Learning

RL is a subsection of machine learning which focuses on how an agent should act on an environment to maximize some reward. An agent is something, or someone, that is performing actions in an environment. That environment is a game or system of some kind with a control scheme such as *chess*, an *Atari 2600* game, or even kicking a soccer ball and an action is a process that modifies the environment in some way, even if that modification is an identity process. The actions that an agent can perform are based on the environment rules. For example, you cannot kick the other players king when playing chess to score a goal just as you cannot pick up the soccer ball and move it three spaces to the right to capture a pawn. A very simple example of an action space is in the *Atari 2600.* The *Atari 2600* controller has a joystick which can move in 8 directions and a button that can be pressed for a total of 16 actions. When the agent is acting on an environment, it follows a policy, π, which describes how it should behave at a given state, $\vec{s}\in S$, by determining an action, $\vec{a}\in A$, and when that action is performed on the environment, a scalar reward, $r\in R$, is returned. The standard definition of a state in a physical sense is every possible variable there is to know about a certain environment, however in RL, the term state is used to describe the observation the agent can make on the environment. In certain problems these two definitions can mean the same thing, but we will be strictly using the term state in the RL sense. While the endgame in a RL problem is typically to maximize the reward, the ways of learning how to do this differ. Typically, although not always, this is done by learning the value function, $V(s)$. The value function evaluates a state of the environment and estimates the value of it, where the value is a measure of the future estimated reward. The two main classes of actions that agents take are exploitation and exploration actions. In exploitation, the agent performs actions at certain states that it is confident will yield the most positive reward. This pathway considered the "safe" path, where there is little risk involved because there is a lot of information known about it. This policy type is often called a greedy policy. However the main issue with exploitation is that unexplored pathways will remain unexplored. In exploration, the agent performs actions at certain states that it is not confident about reward that this pathway will yield. This pathway is considered the "dangerous" path, where there is high risk involved because there is little information known about it. This policy type is often called a random policy. Although there is a chance of getting less reward than in the exploitation path, there is also a chance of getting more reward. Examples of exploitation and exploration with *Grid-World*, an environment explained more in depth in Section 3.4.4, can be seen in Figures 2.1 and 2.2. In all paths shown with the exploitation policy, the agent repeats the same actions that it thinks are optimal, acquiring a best total reward of 0. However with the exploitation policy, the agent finds a path that gets the same total reward of 0 but in less steps, and it also finds a path that gets a higher total reward of 1 in the fewest amount of steps. Many algorithms exist to try find the optimal policy function and no single one is better than all the others. The collection of states, actions, and rewards from start to end is called an episode and these episodes, in one form or

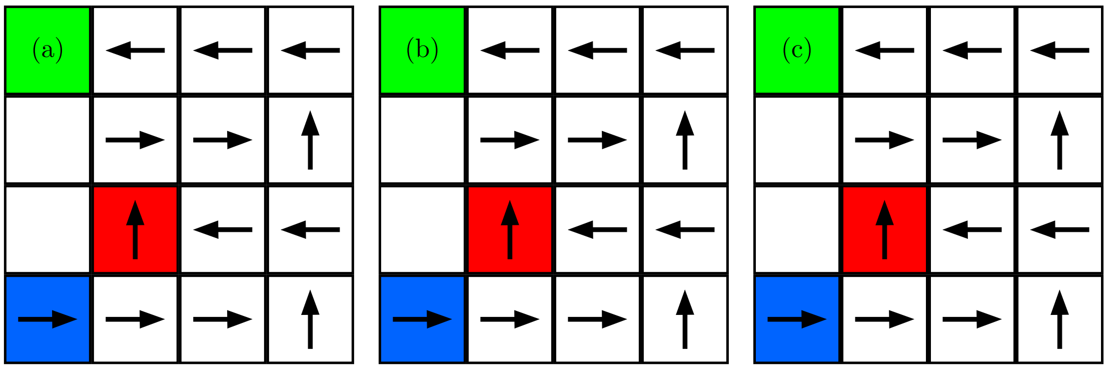


Figure 2.1: (a) An example of an agent acting on a *Grid-World* environment with an exploitation policy three separate times, each starting with the same initial conditions. The agent chooses a path and receives a total reward of 0 after 13 steps. (b) The agent exploits the knowledge of the first path taken and repeats it, receiving a total reward of 0 after 13 steps. (c) The agent exploits the known path once again and receives a total reward of 0 after 13 steps.

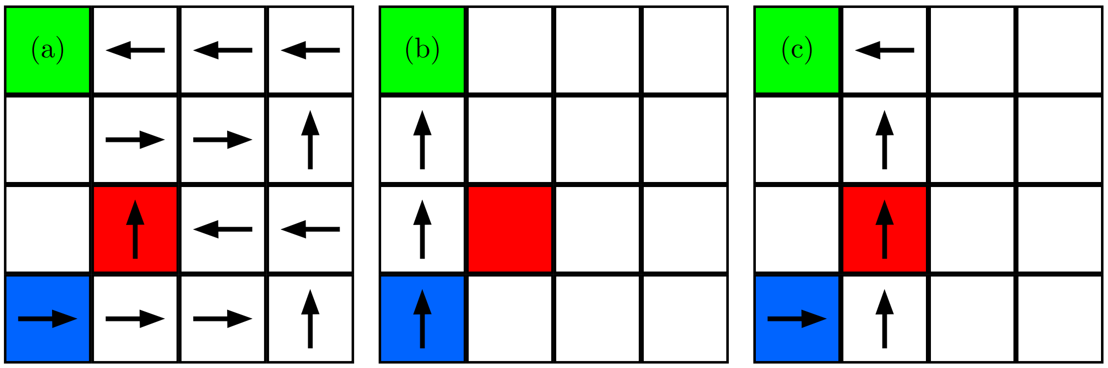


Figure 2.2: (a) An example of an agent acting on a *Grid-World* environment with an exploration policy three separate times, each starting with the same initial conditions. The agent chooses a path and receives a total reward of 0 after 13 steps. (b) The agent explores by choosing a different path than the first path and receives a total reward of 1 in 3 steps. (c) The agent explores again by choosing a different path than the first and second paths and receives a total reward of 0 in 5 steps.

another, are required by these algorithms to optimize a policy.

2.1 Markov Decision Processes

Just as there are more than one RL method, there are more than one class of RL problem. When the state of the environment contains all the information of the problem and the actions are deterministic, it is considered a Markov Decision Process (MDP) [28, 50–54]. While RL has become much more popular recently, the idea of MDPs have been around much longer [55–57]. When a problem is a MDP the agent does not need to know the previous states, actions, or rewards for that episode when making a decision because the current state has all the information the agent needs to know. Popular examples of MDPs are *chess* and *go* because the agent can observe the entire board configuration and can alter the state of the game directly. While the examples provided are considered challenging due to the enormous state and action space, MDPs are generally considered easier problems due to the amount of information contained in the state and the level of control of the actions. However when the state no longer contains all information in the environment but the actions are still deterministic, it is considered a POMDP. These problems are more difficult than MDPs because the agent now needs to have memory of previous states and actions to properly choose an action for the current state. Many video games can be considered POMDPs because the buttons on the controller that the agents presses directly map to the player in the game doing something but the image shown on the screen does not show everything. When playing *Super Mario Bros.* and looking at a frame of Mario jumping, the agent cannot tell if Mario is moving upwards or downwards without knowing what the previous frame was. Some environments that are MDPs and use position and velocity as the state can be turned into a POMDP by removing the velocity component. If we consider the opposite case where the problem has states that are completely observable but the actions are not deterministic, it is considered a Markov Chain. An example of this class of problem is playing *roulette* at a casino. The agent can see everything about the environment and knows the probabilities of the ball landing in any of the slots. However the agent can only bet on the ball landing in various categories or specific numbers but the agent's bet does not affect the outcome of the game, nor does the agent need to know the outcomes of previous rounds to know the probabilities of this one. The last scenario is if the problem has states that do not contain all the necessary information and the actions are also not deterministic, it is considered a Hidden Markov Model (HMM). An example of a HMM is another casino game *Twenty-One*. At the start of the game the deck has all 52 cards remaining and therefore each card has equal probability of being drawn. Just like with *roulette*, there is nothing that the agent can do to affect the outcome of the next turn, in this case which cards will be drawn. However unlike *roulette*, the more games of *Twenty-One* that are played, the more the probabilities of what the next state will be changes. Knowing how many cards are left in the deck does not give the agent enough information to determine the probabilities of a specific card being drawn. If the agent can remember the previous games played then it knows which cards are remaining in the deck. This policy is known as card counting and is generally frowned upon at casinos because it allows an agent to receive a much higher reward. It is important to know what class the problem falls into because it changes the requirements for the agent to solve it.

2.2 Q-Learning

Q-learning [19–23] is one of the earlier algorithms for maximizing reward in a RL problem and uses a quality function, or Q-function, to predict the value of an action at a specific state at time t with

$$Q^*\left(\vec{s}_t, \vec{a}_t\right) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ..., \tag{2.1}$$

where Q^* is the true Q-function, $\gamma \in [0, 1)$ is the discount factor. When $\gamma = 0$, only reward for the immediate step is considered, and as γ approaches 1, future reward is more heavily considered. If the environment has a maximum episode length that is guaranteed to always be finite, meaning it is inevitable that the agent will eventually reach a termination state, then γ is allowed to be 1. When $\gamma = 1$ all reward for the entire remainder of the episode is weighted equally. We can simplify Equation 2.1 by using

$$Q^*\left(\vec{s}_{t+1}, \vec{a}_{t+1}\right) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...,$$

for time $t+1$, and the assumption that we will take the action with the highest value, Q^* then becomes

$$Q^*\left(\vec{s}_t, \vec{a}_t\right) = r_t + \gamma \max_{\vec{a}} Q^*\left(\vec{s}_{t+1}, \vec{a}\right). \tag{2.2}$$

If we have the true Q-function then Equations 2.1 and 2.2 are equivalent. However, when trying to learn, or approximate, Q^* for an environment, the Q-function is updated by

$$Q'\left(\vec{s}_t, \vec{a}_t\right) = (1-\alpha)\, Q\left(\vec{s}_t, \vec{a}_t\right) + \alpha \left(r_t + \gamma \max_{\vec{a}} Q\left(\vec{s}_{t+1}, \vec{a}\right)\right), \tag{2.3}$$

a	s_0	s_1	s_2	s_3	...	s_{11}	s_{12}	s_{13}	s_{14}
0	0.7290	0.8100	0.7290	0.6561	...	-0.1900	0.8100	1.0000	0.8100
1	0.7290	0.8100	0.9000	0.8100	...	0.8100	0.8100	0.8100	1.0000
2	0.7290	0.8100	0.9000	0.8100	...	0.8100	1.0000	0.8100	0.8100
3	0.7290	0.6561	0.7290	0.8100	...	1.0000	0.8100	-0.1900	0.8100

Table 2.1: The true discrete Q-function used to create Figure 2.3 where s_0 is the starting state, s_{11}, s_{12}, s_{13} and s_{14} are the grid spaces which neighbor the goal square, either directly or due to the PBCs, and the rest are indexed arbitrarily.

where Q' is the new Q-function, Q is our approximation of Q^*, and $\alpha \in (0, 1]$ is the learning rate. This is done with the assumption that we do not possess all rewards from r_t to r_∞, because if we did, then there would be no need to approximate Q^*. Q-learning works best when the state and action spaces are discrete. When this is the case, the Q-function can be thought of as a table and is best visualized using the *Grid-World* example shown before. The true Q-function for an example game of *Grid-World*, shown in Figure 2.3(a), gives the values for each of the four actions for each state in this variant of the goal and obstacle layout. If the Q-function, or another variant of the value function, is not used to update our policy, the agent has no concept of which actions to take at each state, as shown in Figure 2.3(b). However if we take the current state shown as our initial state, the agent can then find the optimal path, or all the optimal paths in this case, by following the action with the maximum value at each state. In the case that two or more actions have the same value, and that value is the maximum at a specific state, the agent would choose at random between these actions. The values and corresponding actions for doing this are shown in Figures 2.3(c) and 2.3(d). A policy, like this one, that gives the actions with the highest value at each state is referred to as an optimal greedy policy. Up to this point we have been assuming that the policy the agent is using is one that is automatically following the value function, however this is not always the case. The Q-function used to create this example for *Grid-World* is shown in Table 2.1, however

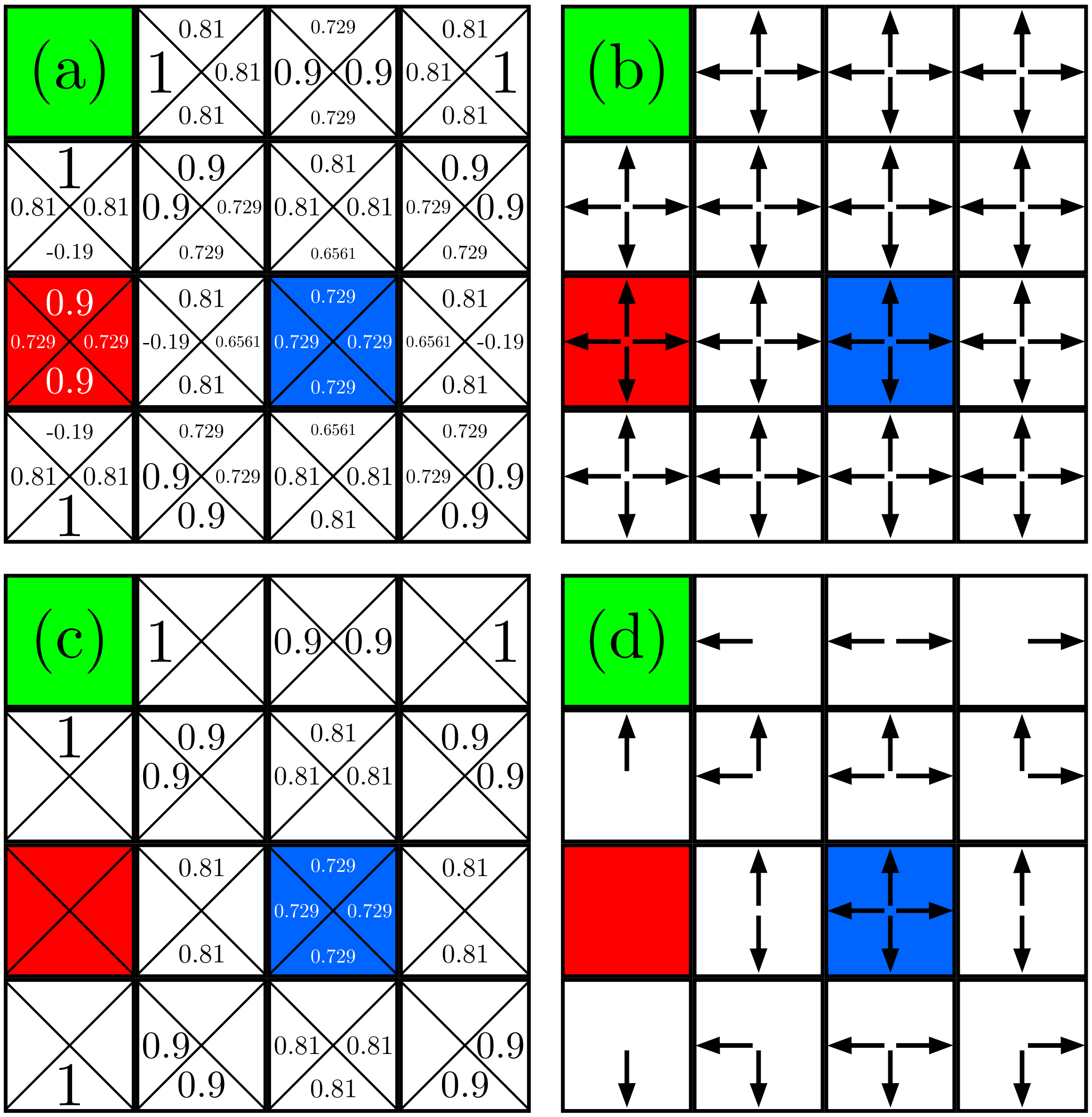


Figure 2.3: (a) The values from the true discrete Q-function for all states and actions for an example game of *Grid-World* overlaid on top of the *Grid-World* grid using $\gamma = 0.9$. (b) The directions for all grid spaces that the agent could move the blue square in using a random policy if the blue square was in that space. (c) The values from the same Q-function with only the highest values shown for the only states that any optimal policy would ever occupy. (d) The directions that the agent could move the blue square in using an optimal greedy policy if the blue square was in that space for the spaces that lie in the optimal path.

there is one obvious major flaw to this method. If the state space and/or the action space were to become too large, then it becomes unfeasible to store this Q-function as it exists presently. DQN is a form of Q-learning that uses an Artificial Neural Networks (ANN) to approximate Q^*. The input of the ANN is the state vector, and the output is a vector with the same length as the number actions. This output vector represents the relative value of each action, where the larger the value for an action, the higher the expected reward. Using an ANN as the Q-function means that we no longer need to store the value of every single action for every single state. Using Equations 2.2 and 2.3 we can construct a loss function similar to Equation 3.3 so that the ANN Q-function can be updated using the back-propagation method described in Section 3.1. By doing this we get

$$L_{\text{DQN}} = \Big(Q\left(\vec{s}_t, \vec{a}_t\right) - \Big(r_t + \gamma \max_{\vec{a}} Q\left(\vec{s}_{t+1}, \vec{a}\right)\Big)\Big)^2 \tag{2.4}$$

as the loss function for our DQN which we can then use the episodes that exist in our environment to update the Q-function. Ideally we would have all possible episodes for a given environment but this is unreasonable for many cases, and for some it is simply impossible. For the case of *chess*, it was estimated by Shannon [58, 59] that there are approximately 10^{120} possible episodes. Using an efficient storage method, each state for chess can be represented using 32 bytes, meaning to store every possible episode would require on the order of 10^{97} YB, where modern hard drives are still measured in TB with 1 YB = 10^{12} TB. For this reason most Q-learning algorithms, and most modern RL algorithms, use Monte Carlo methods to learn the value or Q-function for an environment. Monte Carlo methods for RL collect episodes from experiences by playing the game or controlling the environment through actions. Once an episode is completed, the agent can then use this collection of states, actions, and rewards to update the policy. This method is called offline learning because the agent only learns between episodes. However the agent can also learn during an episode, using the starting experiences to improve its behavior before the end of the episode. When an agent learns during an episode it is called online learning. While online learning is more similar to how humans learn, it can be tricky because the agent would be using a different policy at the start of an episode than at the end of an episode. Using Monte Carlo methods allows the agent to learn more from more probable states and explore other states around them. When these Monte Carlo methods are paired with RL algorithms like Q-learning, it is know as TD learning and is a central component of modern RL.

Chapter 3

Benchmarking the Genetic Algorithm

Before we can study RL problems of interest to us, we first must develop and test the desired method we wish to use. After the success Such et. al. [44] had with the GA on *Atari 2600* games, we decide to try this on other problems. Doing this requires knowledge of neural network-based policies and GAs.

3.1 Artificial Neural Networks

ANN can be used to approximate a function $y = f(x)$ with a set of parameters θ [60]. θ can be adjusted such that the difference between the approximation represented by an ANN, $g(x;\theta)$, and $f(x)$ is minimal. A standard ANN is a feed-forward network. A feed-forward neural network passes information forward through the network with no information being passed backwards to previous connections. These functions are called networks because they are usually several functions acting on each other.

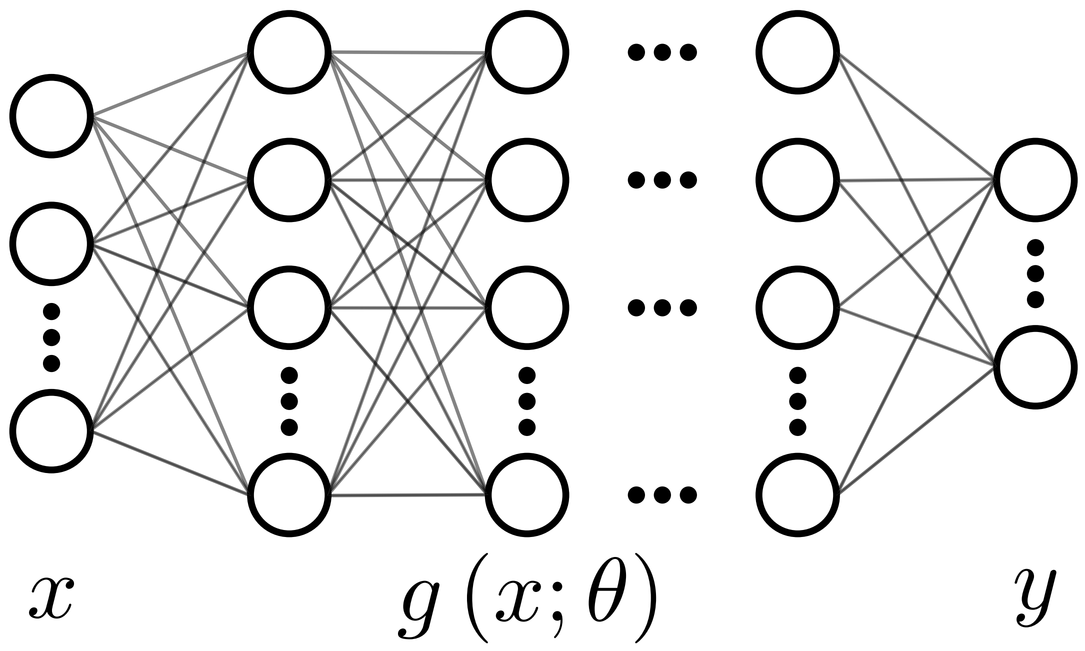


Figure 3.1: An ANN mapping an input, x, to an output, y, with an approximation, $g\left(x;\theta\right)$, of the true function, $f\left(x\right)$. Each circle represents a unit in a layer and each line represents a weight connecting a unit from layer i to layer $i+1$.

$g\left(x;\theta\right)$ can be represented as

$$g^{(n)}\left(...g^{(2)}\left(g^{(1)}\left(x;\theta^{(1)}\right);\theta^{(2)}\right)...;\theta^{(n)}\right), \tag{3.1}$$

where $g^{(i)}$ is the i^{th} layer of the network, and n is the depth of the ANN. The larger value of n used, the deeper the neural network is. The n^{th}, or final, layer of a neural network is the output layer. The final output of the network is meant to approximate the answer y, however the outputs of layers 1 to $n-1$ cannot be labeled with some answer y and are therefore referred to as hidden layers. The outputs of these hidden layers are typically a vector with each component in the vector called a unit. The number of units in each hidden layer determines the width of the neural network. The structure of an ANN can be seen in Figure 3.1. Shown in Equation 3.2, each

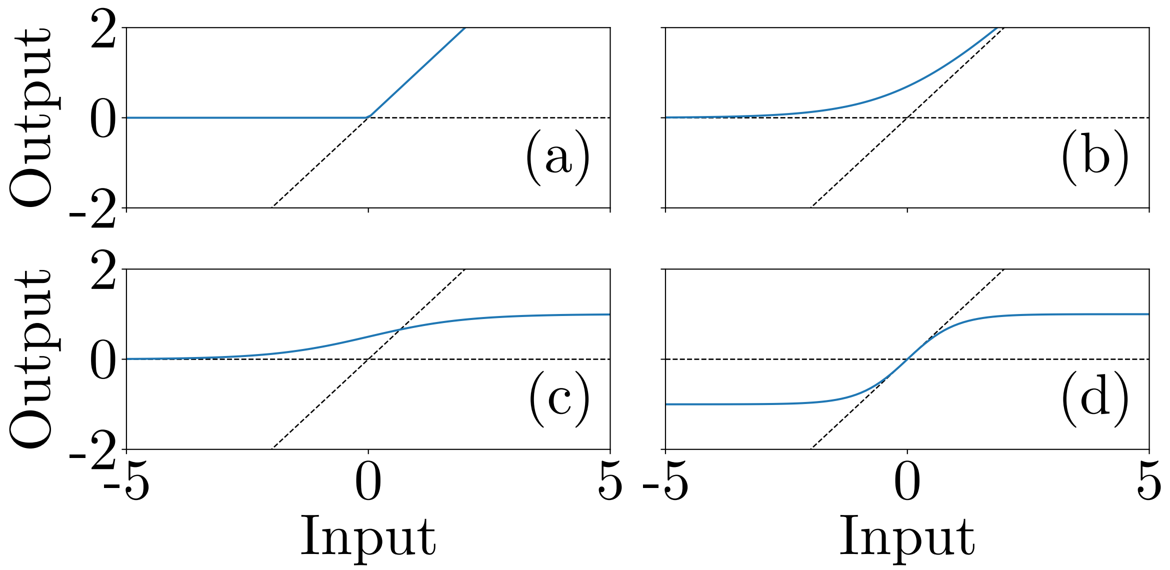


Figure 3.2: (a) Examples of the Rectified Linear Unit, (b) Softplus, (c) Sigmoid, and (d) Hyperbolic Tangent activation functions with $y = 0$ and $y = x$ lines shown for references.

$g^{(i)}$ typically consists of a set of weights in matrix form, w, a set of biases in a vector form, b, and an activation function, a, represented as

$$g^{(i)}\left(x\right) = a^{(i)}\left(x \times w^{(i)} + b^{(i)}\right) \approx y. \tag{3.2}$$

Examples of commonly used activation functions for an ANN are shown in Figure 3.2. This type of network is often referred to as fully connected because every unit in one layer is connected to every unit in the next layer. Typically in an ANN, y has less units than x, which has less units than the layers of $g\left(x;\theta\right)$, as depicted in Figure 3.1, however this is not a requirement and each layer of $g\left(x;\theta\right)$ does not need to be the same size as the rest. To determine the accuracy of the ANN, a measure of its error, or loss, is needed. One of the more common loss functions is mean squared error, defined as

$$L = \left(y_{\mathrm{t}} - y_{\mathrm{p}}\right)^2, \tag{3.3}$$

where y_t is the true label for x and y_p is the output of the ANN. The loss function, L, can then be used to optimize the parameters such that L is minimized. To do this the gradient of L is taken with respect to the parameters of the ANN, and then using gradient descent, the weights and biases can be modified to make a better approximation. First the loss needs to be propagated back through the network via back-propagation. Starting with the final layer, as it is the simplest due to its output being the output used to compute the loss, taking the gradient of L with respect to the n^{th} set of parameters, we get

$$\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial a^{(n)}} \frac{\partial a^{(n)}}{\partial g^{(n)}} \frac{\partial g^{(n)}}{\partial \theta^{(n)}}, \tag{3.4}$$

due to the chain rule, where $\theta^{(n)}$ is the set of parameters, $g^{(n)}$ is the output, and $a^{(n)}$ is the activation function, all for the final, n^{th}, layer of the network. Moving to the preceding layer, and taking the gradient of L with respect to the $n^{\mathrm{th}} - 1$ set of parameters, we get

$$\frac{\partial L}{\partial \theta^{(n-1)}} = \frac{\partial L}{\partial a^{(n)}} \frac{\partial a^{(n)}}{\partial g^{(n)}} \frac{\partial g^{(n)}}{\partial a^{(n-1)}} \frac{\partial a^{(n-1)}}{\partial g^{(n-1)}} \frac{\partial g^{(n-1)}}{\partial \theta^{(n-1)}}. \tag{3.5}$$

Repeatedly doing this for all layers we finally get

$$\frac{\partial L}{\partial \theta^{(1)}} = \frac{\partial L}{\partial a^{(n)}} \frac{\partial a^{(n)}}{\partial g^{(n)}} \frac{\partial g^{(n)}}{\partial a^{(n-1)}} \frac{\partial a^{(n-1)}}{\partial g^{(n-1)}} \frac{\partial g^{(n-1)}}{\partial \theta^{(n-1)}} \cdots \frac{\partial g^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial g^{(1)}} \frac{\partial g^{(1)}}{\partial \theta^{(1)}}, \tag{3.6}$$

for the first layer. Using Equations 3.4-3.6, and the gradients for layers 2 through $n - 2$ that are not explicitly shown here but can be easy interpolated, the set of parameters, θ, in $g(x; \theta)$ can be properly perturbed to minimize L.

3.2 Genetic Algorithms

A GA is an optimization procedure based on evolutionary theory. By itself, a GA is not RL, however any algorithm that is used for RL is a RL algorithm in that case. In the case of RL, the function being optimized is the policy, $\pi(\vec{S};\theta_n)$, where θ_n are the n parameters being changed. A population is created with N_P policies, each randomly initialized using the same distribution. This population is called generation 0. Each policy is then evaluated on the environment for at least one full episode and given a score, which is a function of the set of rewards acquired throughout the episode. The population is sorted by score and the top N_e policies are kept. These N_e policies are referred to as the elite population. The remaining $N_P - N_e$ policies to repopulate the population are created via crossover or mutation. In crossover, two policies, π_1 and π_2, are randomly selected from the elite population with uniform probability. The new policy is created by taking the first $\epsilon_c n$ parameters from π_1, where $\epsilon_c \in (0,1)$, and taking the last $(1-\epsilon_c)n$ parameters from π_2. In mutation, one policy, π_1, is randomly selected from the elite population with uniform probability. The new policy begins as a copy of π_1, then a set of random numbers, which are generated with the same size and distribution as the initialized parameters, are multiplied by some number $\epsilon_m \in (0,1]$, and then added to the new policy's parameters. The new population, generation 1, now consists of the elite, newly bred, and newly mutated policies. The evaluation, elimination, crossover, and mutation processes are repeated until the maximum score of the population converges.

3.3 Neural Network Based Policy

For our reinforcement learning methods we used a neural network based policy to process the state vectors. The neural network based policy takes the state vector from the environment as input, and fully connects it to a single hidden layer of 1024 units with a hyperbolic tangent activation function. The hidden layer is then fully connected to the output layer of n_a units where n_a is the total number of actions the agent can choose from. This ANN architecture was chosen because we found that it gave the best balance of low computational cost while still being able to solve trivial test cases reliably. The index of the maximal value in the output vector denotes the predicted optimal action of the policy. In the case of a continuous action space, the output vector itself denotes the predicted optimal action of the policy. Each parameter in the network is randomly initialized from a truncated normal distribution centered at 0, with a standard deviation of 1. An example of this network architecture for the *Carnot* environment is shown in Figure 3.3. A GA is used to optimize this neural network using a population of $N_P = 100$ policies, $\pi(\vec{S}; \theta_n)$, each with a set of parameters, θ_n, representing the weights and biases of the neural network, initialized randomly from a normal distribution with a mean of 0 and variance of 1. Each policy is evaluated on our target environment and sorted by increasing score. The top $N_e = 25$ elite policies are kept in the population, and the removed policies are replaced by 75 new policies created through either crossover with $\epsilon_c = 0.5$ or mutation with $\epsilon_m = 0.05$.

3.4 Benchmark Environments

To test any RL method, a series of well studied environments is required. To evaluate an agents performance on these tasks, we need to understand what the problem that each environment is designed to solve.

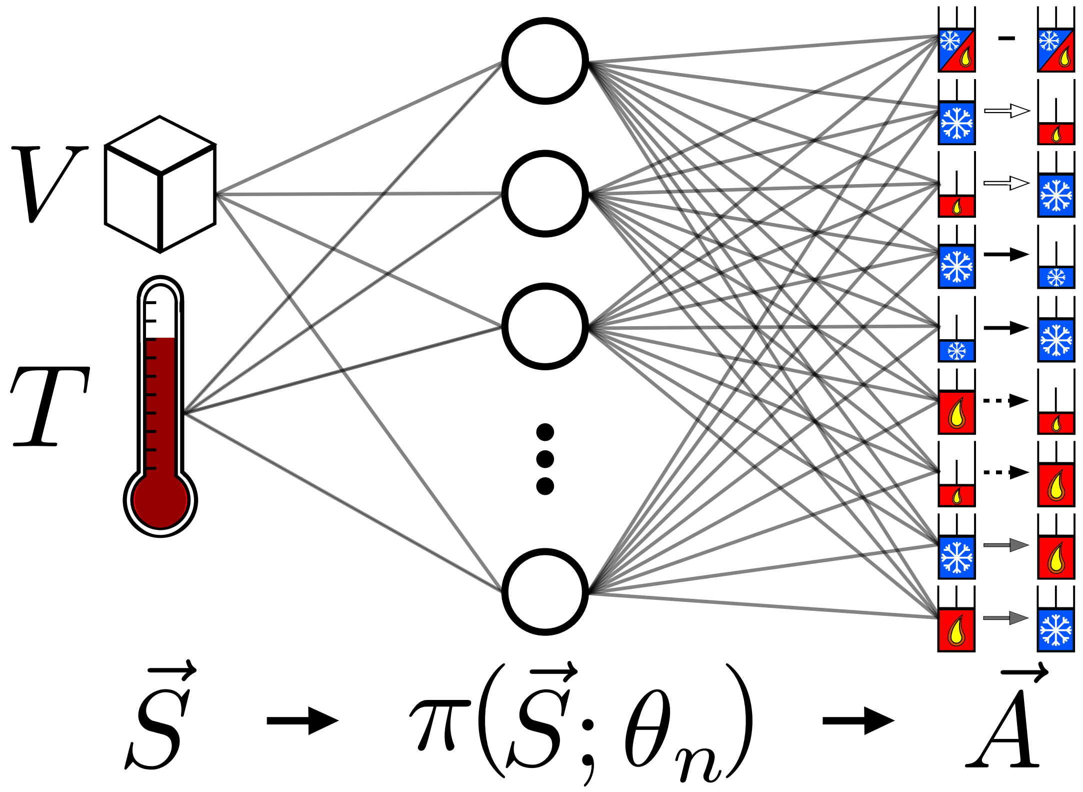

V
T
$\vec{S} \rightarrow \pi(\vec{S};\theta_n) \rightarrow \vec{A}$

Figure 3.3: A diagram of the network based policy $\pi(\vec{S};\theta_n)$ that takes the state vector $\vec{S} = [T, V]$ and through a neural network parametrized by weights θ_n, maps the state to a vector of actions. The position of the maximum value of this action vector, $\vec{A}$, denotes the policy's predicted optimal action. At any given iteration, there are N_P sets of parameters comprising the entire population of policies.

3.4.1 Cart-Pole

Cart-Pole is a game played on a linear track where the goal is to prevent a pendulum pole from falling over. *Cart-Pole* consists of a cart on a 1-dimensional frictionless track, with an non-actuated pole attached to the center. The available actions are: 1) Apply force to the cart in the negative x direction, and 2) Apply force to the cart in the positive x direction. At each step, the force of gravity and the moving cart are acting on the pole, causing it to rotate either clockwise or counter-clockwise. The state of *Cart-Pole* is a vector containing the x position and velocity of the cart, the angle the pole makes with the track, and the velocity of the tip of the pole. A graphical example of a state of *Cart-Pole* can be seen in Figure 3.4(a). The game is initialized with the value for each variable of the state vector being randomly selected from a uniform distribution in the range of [-0.05, 0.05]. The game is played until the cart reaches an x position of ± 2.4, the angle of the pole reaches ± 12.0 degrees, or 200 steps has been reached. The limit on the x for this problem is used to ensure the agent keeps control over the cart, the limit on the pole angle is to ensure the agent keeps the pole balanced upwards, and the limit on the number of steps is to ensure that the problem is guaranteed to converge. At each step, excluding the termination step, the environment returns a reward of 1. This environment is considered solved when an agent achieves an average total reward of 195 or greater for 100 episodes in a row. This environment was chosen for the lose condition which does not allow an agent to continue playing if it is performing too poorly, similar to a video game.

3.4.2 Mountain-Car

Mountain-Car is a game played on a valley style track where the goal is to push the car to the top of the hill. *Mountain-Car* consists of a car on a frictionless 2-dimensional

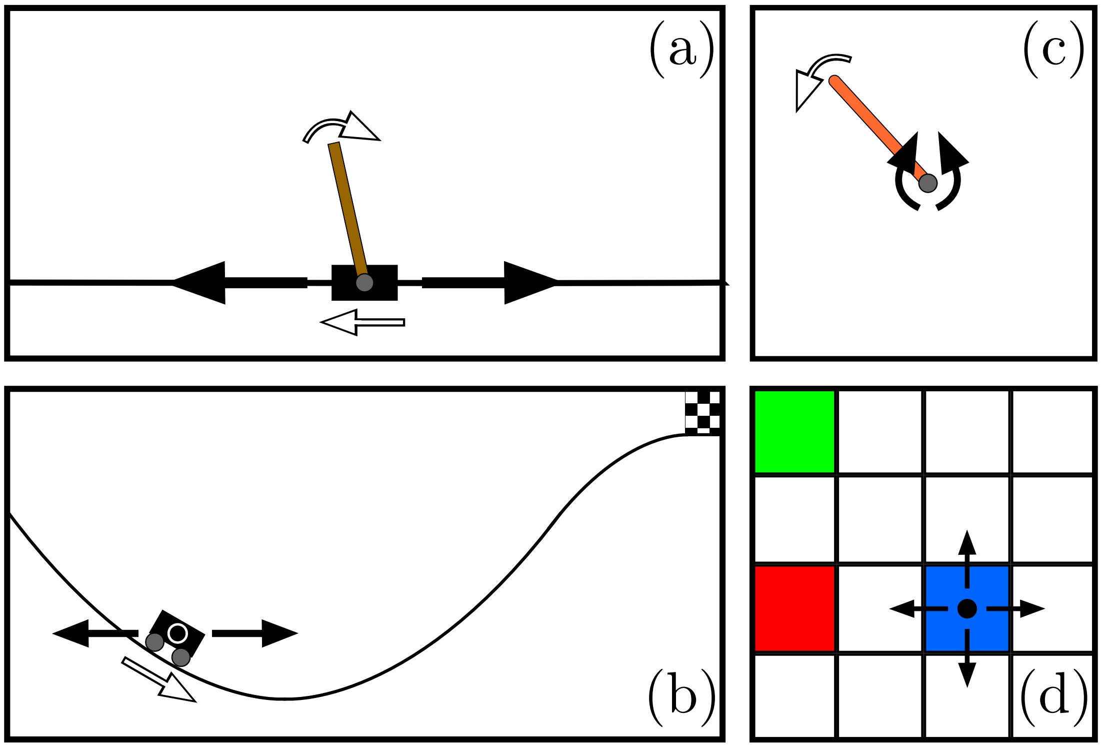


Figure 3.4: (a) A graphical example of what the *Cart-Pole* environment could look like during a given state. The black rectangle represents the cart and the brown rectangle represents the pole. The filled arrows represent the actions and the hollow arrows represent the velocity components of the state. (b) A graphical example of what the *Mountain-Car* environment could look like during a given state. The black rectangle represents the car. The filled circle represents action 2), the filled arrows represent actions 1) and 3) and the hollow arrow represents the velocity component of the state. (c) A graphical example of what the *Pendulum* environment could look like during a given state. The orange rod represents the pendulum. The filled arrows represent the positive and negative action space and the hollow arrow represents the angular velocity component of the state. (d) A graphical example of what the *Grid-World* environment could look like during a given state. The blue square represents the player square, the green square represents the goal square, and the red square represents the obstacle square. The circle represents action 1) and the arrows represent actions 2)-5).

track which slopes upwards in both the negative and positive x directions. The available actions are: 1) Apply force to the car in the negative x direction, 2) Apply no force to the car, and 3) Apply force to the car in the positive x direction. At each step, the force of gravity is acting on the car and is strong enough that the car cannot be pushed up the hill from rest. The state of *Mountain-Car* is a vector containing the x position and velocity of the car. A graphical example of a state of *Mountain-Car* can be seen in Figure 3.4(b). The game is initialized with a random x position of the car selected from a uniform distribution of [-0.6, -0.4], where the bottom of the valley is located at $x = -0.5$. If the car reaches the unmovable wall located at $x = -1.2$ the collision is perfectly inelastic and therefore the velocity of the car becomes 0. The hills in *Mountain-Car* are modelled by $y = \sin(3x)$ and therefore the bottom of the valley is at -$\frac{\pi}{6}$. The game is played until the car reaches the goal at $x = 0.5$ or 200 steps has been reached. At each step, the environment returns a reward of -1, until the goal is reached, in which case a reward of 0 is returned. This environment is considered solved when an agent achieves an average total reward of -110 or greater for 100 episodes in a row. This environment was chosen for the lack of feedback during an episode which does not allow the agent to learn anything until it has reached the goal, similar to games like *chess*.

3.4.3 Pendulum

Pendulum is a game played on a fixed position peg where the goal is to keep a pendulum in the upright position with minimal effort. *Pendulum* consists of a frictionless actuated pendulum with a joint at one end fixed in position but not rotation. Unlike the rest of the environments in Section 3.4, the action space for *Pendulum* is not discrete, but rather a continuous single action in the range of [-2.0, 2.0]. This number corresponds to the amount torque applied on the pendulum by the joint, with torque in the counter-clockwise direction being positive. At each step, the force of gravity is acting on the pendulum and strong enough that the pendulum cannot be rotated to the upright position from rest at the downward facing position. The state of *Pendulum* is a vector containing $\cos(\theta)$, $\sin(\theta)$, and $d\theta$, where $\theta \in [-\pi, \pi]$ is the angle in radians from the upright position. A graphical example of a state of *Pendulum* can be seen in Figure 3.4(c). The game is initialized with a random θ selected from a uniform distribution in the range of $[-\pi, \pi]$ and a random $d\theta$ selected from a uniform distribution in the range of [-1, 1]. The game is played until 200 steps has been reached. The reward returned by the environment at each step is calculated using

$$r = -\left(\theta^2 + \frac{d\theta^2}{10} + \frac{\vec{a}^2}{1000}\right), \tag{3.7}$$

where $\vec{a}$ is the current action that was taken. This reward function is an example of how to shape the requirements on the solution the agent will find. The first term is minimized by having the pendulum in the upright position, the second term is minimized by keeping the pendulum still, and the third term is minimized by applying no force to the pendulum.This environment was chosen for the continuous action space because the agent must learn to solve a problem which is more similar to regression than classification.

3.4.4 Grid-World

Grid-World is a game played on a $n \times m$ grid with Periodic Boundary Condition (PBC)s where the goal is to move the player square to same space as the goal square while avoiding the obstacle square(s). The available actions are: 1) Don't move the player square, 2) Move the player square up, 3) Move the player square left, 4) Move the player square down, and 5) Move the player square right. The state of *Grid-World* at any time is a $n{\times}m$ array representation of the grid where each grid space corresponds to an element of the state. The element of the state vector that corresponds to the blue square's position is represented by some number x, the element that corresponds to the goal square is represented by some number $y{\neq}x$, and the element that corresponds to the obstacle square is represented by -y. A graphical example of a state of Grid-World can be seen in Figure 3.4(d). The game is initialized with the player squared being placed randomly on the grid with uniform probability, followed by the goal square being placed randomly on the grid with uniform probability, excluding the position with the player square. Grid-World can be played with between 0 to $n{\times}m-2$ obstacle squares, where each obstacle squared is placed randomly on the grid with uniform probability of the remaining positions. If the player square is moved to the same position as the goal square, there is a reward of +1 and the environment is reinitialized. If the player square is moved to the same position as the obstacle square, there is a reward of -1 and the obstacle square is removed from the grid. *Grid-World* is played until N steps are reached and the maximum total reward is acquired by moving the player square along the shortest L_1-norm path to the goal square that passes through the least number of obstacle squares possible. This environment is considered solved when an agent achieves an average total reward of

$$R_{GW} = \frac{2N}{n+m-2} - x_{Obs} \tag{3.8}$$

or greater for 100 episodes in a row where R_{GW} is the sum of all reward in an episode of *Grid-World* and x_{Obs} is the number of obstacle squares if x_{Obs} is much less than N. This solve condition comes from the fact that $n+m-2$ is the maximum L_1 distance that can be between the player and goal squares, however due to the PBCs, the maximum L_1 distance is actually half of what it would be without PBCs. The ratio of the number of steps per episode and the maximum L_1 distance gives the best possible return given the worst case scenario subject to the initial conditions. Lastly, the x_{Obs} takes into account possible cases that the agent might have to go through an obstacle square to reach the goal. This environment was chosen for the image-like state so the agent would have to learn the importance of neighboring pixels.

A second version of *Grid-World* was created to add an element of randomness, other than the initialization stage. This version is an copy of the original *Grid-World*, however the way actions are performed has been altered. When an action is taken in the original *Grid-World*, the probability of that action occurring is 100%, as seen in Figure 3.5(a), and therefore it will be referred to as *Direct Control Grid-World.* When an action is taken in this new *Grid-World*, the probability of that action occurring is 50% with a 12.5% probability each that one of the other four remaining actions occur instead, as seen in Figure 3.5(b), and therefore it will be referred to as *Probabilistic Control Grid-World.* The purpose of this second version of *Grid-World* is to test how an agent handles random elements when it comes to actions.

3.5 Methods

To evaluate the GA algorithm it was used to optimize a neural network based policy on each of the environments described in Section 3.4. While optimizing a neural network based policy, each policy in the population was evaluated on the environment 10 times to acquire an average score due to the randomness of the initial conditions. The score for each environment is the sum of the reward for each step. OpenAI's *CartPole-v0*, *MountainCar-v0*, and *Pendulum-v0* gym environments were used for the *Cart-Pole*, *Mountain-Car*, and *Pendulum* environments respectively. A grid size of 5×5 with 1 obstacle square was used for the *Grid-World* environment with 1, 0.5, and -0.5 for

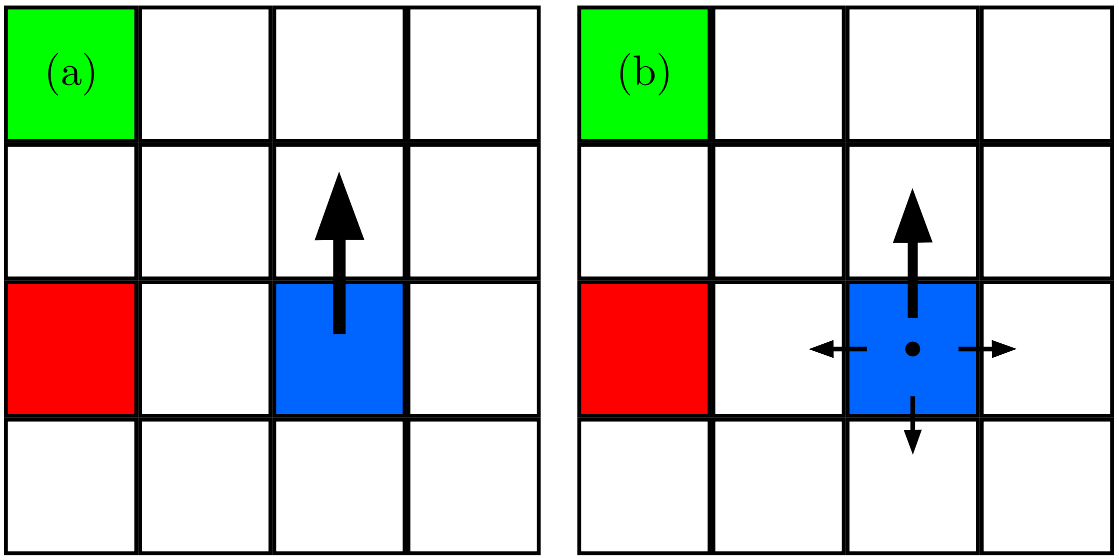


Figure 3.5: (a) The probabilities, represented by size, of each action occurring on the example state after choosing action 2) with the *Direct Control Grid-World* environment and (b) the *Probabilistic Control Grid-World* environment.

the player, goal, and obstacle squares respectively. Due to *Grid-World* having a 2-dimensional state, the output layer of the policy is $n \times n_a$ where n is the width of the grid and n_a is the total number of actions. To resolve this the output layer is reduced to $1 \times n_a$ by summing along the first dimension. The GA optimization was run on each environment until a policy's average score was the maximum possible score or, in the cases that the maximum score or solve requirements are not well defined, the population's average and maximum scores converge.

3.6 Results

The first environment our GA optimization process was tested on was the *Cart-Pole* environment as it is simplest of the ones listed in Section 3.4 due to the low complexity and the size of both the state vector and action space. For this reason, the *Cart-Pole* environment was also used to test different ratios of $\epsilon_c = 0.5$ crossover and

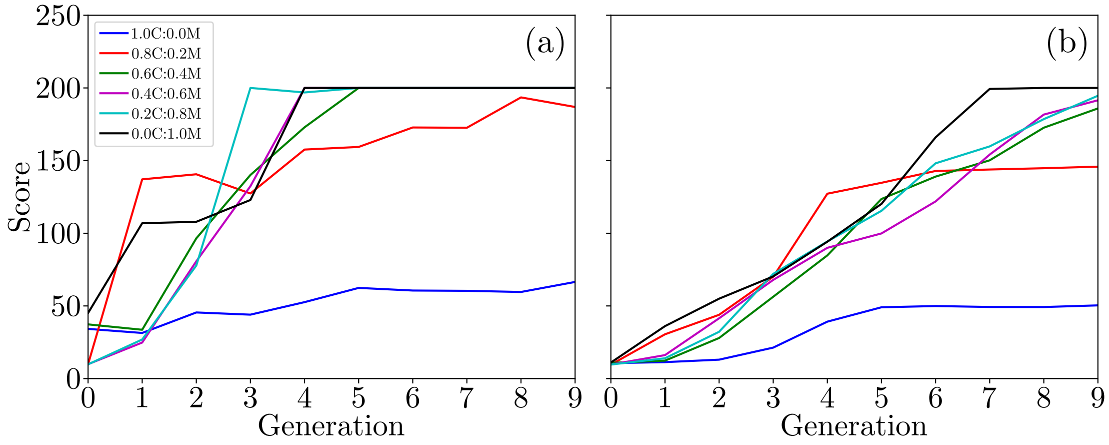


Figure 3.6: (a) The maximum average score of the population (b) average score of the elite population per generation for various ratios of policies created through crossover and mutation during the optimization process for the *Cart-Pole* environment.

$\epsilon_m = 0.05$ mutation for repopulating the remaining 75 necessary policies spaces in the population. The maximum average score of the population per generation is shown in Figure 3.6(a) and the average score of the elite population per generation is shown in Figure 3.6(b). When comparing the maximums and averages across crossover and mutation ratios, it is apparent that using only mutation yields the best results and therefore all other tests with the GA will only use mutation. This is likely due to the nature of this ANN architecture. The mutation operation applies small perturbations to each parameter in the network, however the crossover operation is taking half of the parameters in one network and appending them with half the parameters of another network. As our ANN only contains a single hidden layer, crossover is splitting it in half, where one half is designed to work with a different set of parameters than the ones that are appended to them. The best performing neural network based policy was able to achieve an average score of 200.00 over 100 consecutive tests, which is the maximum possible score in the *Cart-Pole* environment.

The second environment our GA optimization process was tested on was the

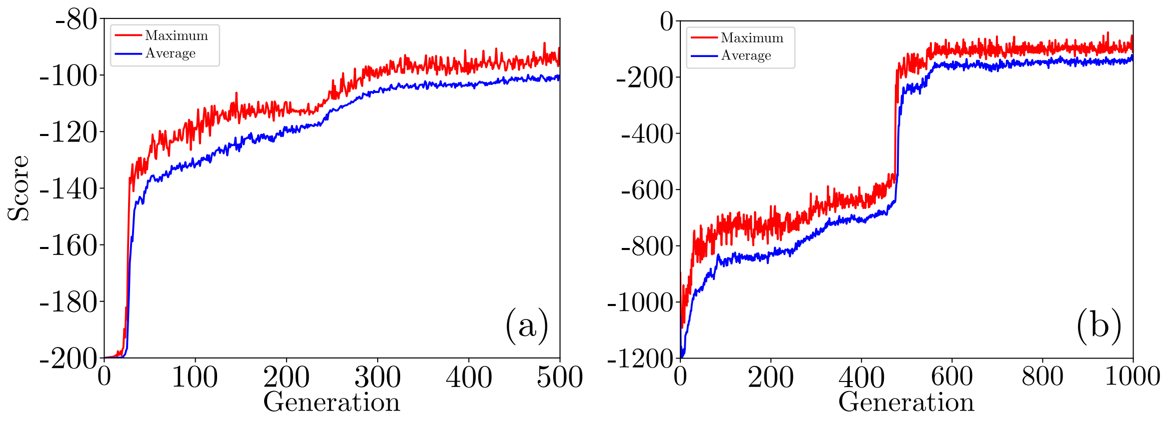


Figure 3.7: (a) The maximum average score of the population and average score of the elite population per generation during the optimization processes for the *Mountain-Car* environment and (b) the *Pendulum* environment.

Mountain-Car environment. Unlike the *Cart-Pole* environment, the *Mountain-Car* environment is a delayed reward problem, meaning that the agent has to perform many actions before there is any increase in reward. This can be a problem for gradient based RL algorithms because it is very difficult to get any reward, and as a consequence, a gradient to optimize the policy. The maximum average score of the population per generation and the average score of the elite population per generation are shown in Figure 3.7(a). Unlike with the *Cart-Pole* environment, the population's scores have a sharp increase when the agent learns how to reach the goal. The best performing neural network based policy was able to achieve an average score of -99.57 over 100 consecutive tests, which is well above the *Mountain-Car* solve conditions. The agents behavior can be seen on a position vs velocity phase plot for 11 different initial conditions in Figure 3.8. The key difference in the initial positions is that the first set, [-0.60, -0.48], and the second set, [-0.46, -0.4], adopt different strategies. In the first case, the agent must first push the car partially up the right hill and then back up the left before being able to push it all the way up the right hill to reach the goal. However in the second case, the agent just has to push the car up the

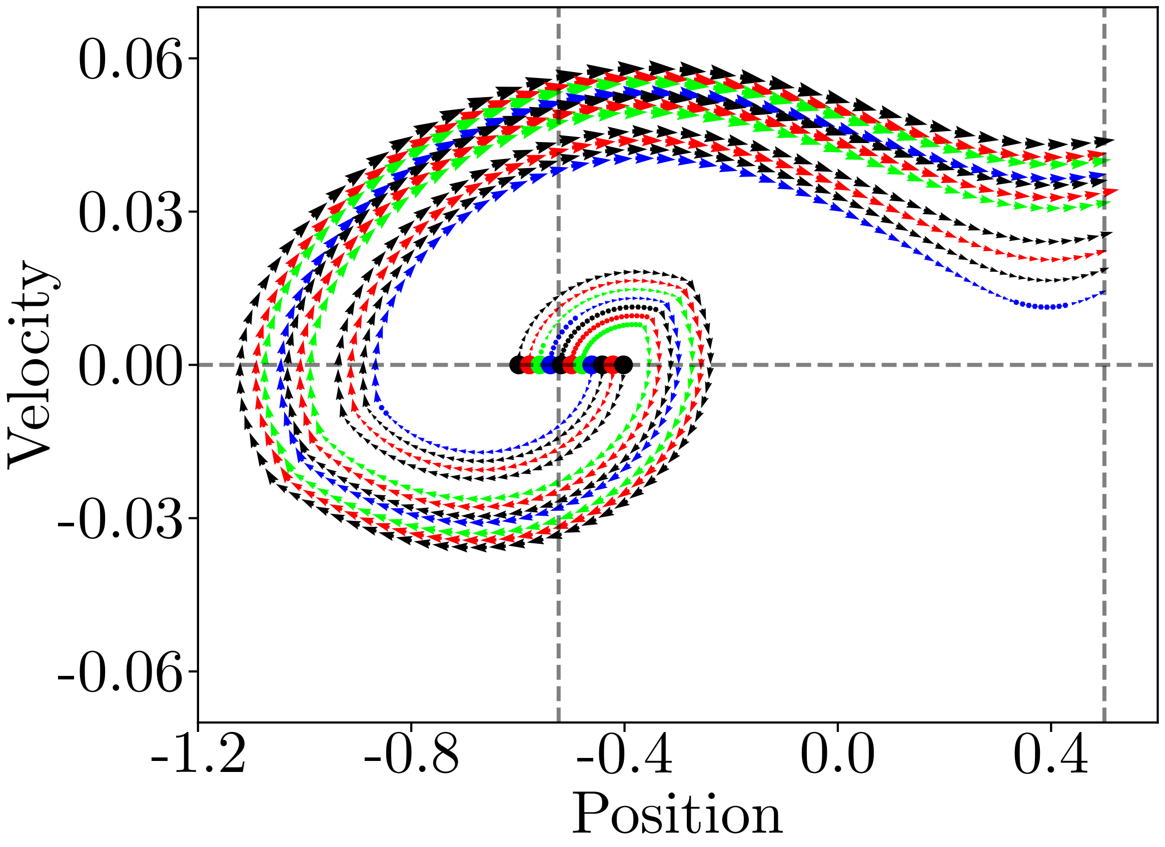


Figure 3.8: The trajectories produced by the best performing neural network based policy on a position vs velocity phase plot for 11 initial positions between -0.6 and -0.4 all with an initial velocity of 0.0. The vertical dashed grey line at -$\frac{\pi}{6}$ denotes the bottom of the valley as that is when the derivative of $\sin(3x) = 0$. The vertical dashed grey line at 0.5 denotes the goal as reaching that x position with any velocity will terminate the game. The horizontal dashed grey line denotes 0 velocity to see when the car starts moving in the opposite direction.

left hill before being able to push it all the way up the right hill to reach the goal. Optimal strategies could be interpolated for initial x positions inside of the range shown here but not specifically shown themselves. While it would be less accurate and slightly more difficult, optimal strategies for initial x positions outside of this range could be extrapolated by simply expanding the vector field generated by the agents other trajectories. The main issue that would arise however would be once this extrapolation reaches the boundary condition of $x =$ -1.2. At this point it would be expected that many trajectories that began unique would all converge due to the instantaneous change to 0 in velocity when the reaching the negative wall.

The third environment our GA optimization process was tested on was the *Pendulum* environment. Unlike the *Cart-Pole* and *Mountain-Car* environments, the *Pendulum* environment has a continuous action space of one action. This changes the RL problem from being analogous to a classification problem to being analogous to a regression problem. The maximum average score of the population per generation and the average score of the elite population per generation are shown in Figure 3.7(b). Unlike the *Cart-Pole* and *Mountain-Car* environments, the *Pendulum* environment has a continuous reward space when using Equation 3.7 to calculate the reward at each step, allowing for a much larger spread in scores. The best performing neural network based policy was able to achieve an average score of -152.01 over 100 consecutive tests. There is no specified solve requirements for the *Pendulum* environment, however when examining the θ vs $d\theta$ phase plot for 20 different initial conditions shown in Figure 3.9, it is apparent the agent achieves the goal stated in Section 3.4.3. The agent gets the pendulum to a θ very near 0 and keeps it there with minimal oscillation. While Figure 3.9 may appear symmetric at initially, due to the stable steady point of the agent's solution not being exactly at a θ of 0 radians, it is not in fact symmetric. Using this phase plot we can see the changes in strategy based

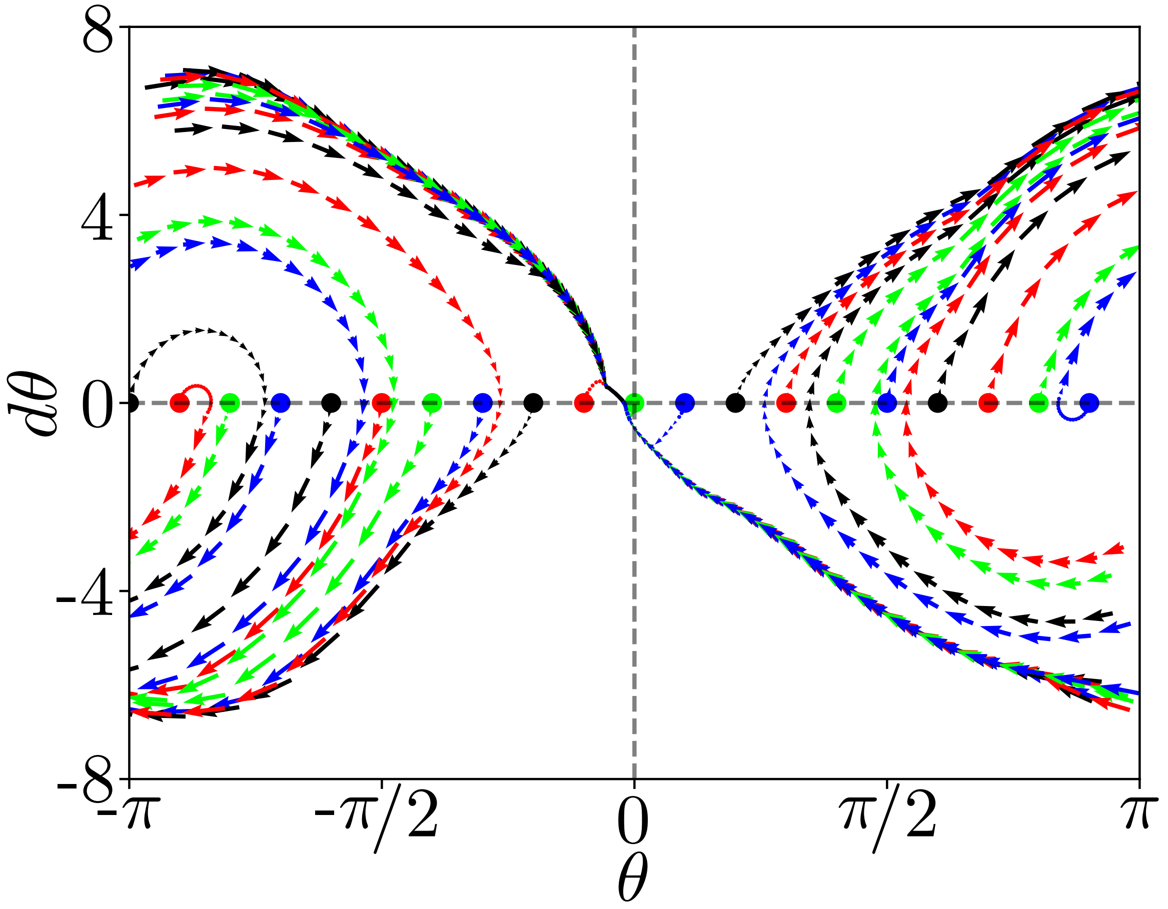


Figure 3.9: The trajectories produced by the best performing neural network based policy on a θ vs $d\theta$ phase plot for 20 equally spaced initial θ between -π and π all with an initial $d\theta$ of 0.0. The vertical grey line denotes an angle of 0 which corresponds to the pendulum being upright. The horizontal dashed grey line denotes 0 radial velocity to see when the pendulum starts moving in the opposite direction.

on initial conditions. While the *Pendulum* environment allows for initial $d\theta$ not at 0, the majority of the trajectories for those initial conditions can be interpolated from the trajectories with $d\theta = 0.0$. Unlike *Mountain-Car*, the agent must try to maintain a certain position opposed to just reaching a specific position with arbitrary velocity. Due to this requirement we see that the trajectories produced by the agent get funneled into one of two streams, approaching the near upright position from either the clockwise or counter-clockwise directions. The starting angles in the range [-0.1π, 0.1π] are close enough to 0 that the agent can immediately push the pendulum to the optimal position. The starting angles outside of this range however begin rocking the pendulum back and forth until it has enough momentum to swing into the near upright position, being funneled into one of those streams. While the pendulum is being rotated in both the clockwise and counter-clockwise directions, all trajectories in Figure 3.9 rotate in the clockwise direction. This rotating motion is due to the fact that the pendulum is often being rocked back and forth around $\theta = \pm\pi$ until it can finally reach the upward position at $\theta = 0$. However no matter what the behavior of the agent is, the phase plot will always appear clockwise in some regard. This is due to the fact that we have $d\theta$ on the y-axis and θ on the x-axis. Whenever the pendulum is in the bottom half of this phase plot, it will be moving towards the left, and whenever the pendulum is the top half of this phase plot, it will be moving towards the right, creating the appearance of clockwise rotation in θ-$d\theta$ phase space.

The last environments our GA optimization process was tested on were the two *Grid-World* environments. These environments are the only ones which are reinitialized in the middle of testing an agent on a single play. The maximum and average score of the population per generation and the average score of the elite population per generation for the *Direct Control Grid-World* are shown in Figure 3.10(a). The best performing neural network based policy was able to achieve an average score

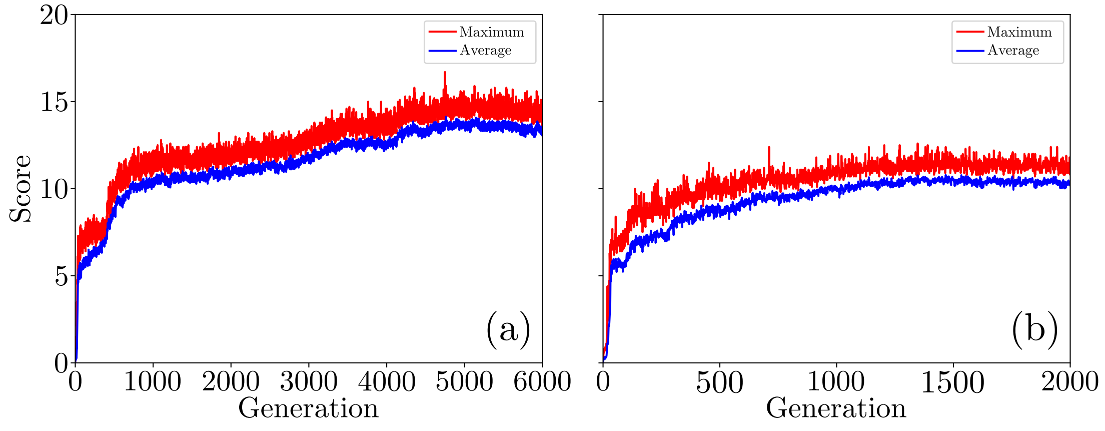


Figure 3.10: (a) The maximum average score of the population and average score of the elite population per generation during the optimization process for the *Direct Control Grid-World* and (b) the *Probabilistic Control Grid-World* environments.

of 13.28 over 100 consecutive tests. Using Equation 3.8, the solve requirements for the *Direct Control Grid-World* environment with our conditions is an average total reward of 11.50 or greater for 100 episodes in a row. An example for what this agent would do in each grid spacing is shown in Figure 3.11(a). This agent learned to always move left until in the same row as the goal square and then move down if it is above the it or up if it is below it. While this result is not the same perfect policy we saw in Figure 2.3(d), it still performs quite well despite the odd behavior of ignoring the move right action. Moving onto the last case, the maximum and average score of the population per generation and the average score of the elite population per generation for the *Probabilistic Control Grid-World* are shown in Figure 3.10(b). The best performing neural network based policy was able to achieve an average score of 10.53 over 100 consecutive tests. Using a probability of 0.5 for a random action being taken instead of the desired one, it is expected that the agent should be able to achieve at least half of the solve condition for *Direct Control Grid-World*. Using Equation 3.8, and multiplying it by the random action probability, the solve requirements for the *Probabilistic Control Grid-World* environment with out conditions is an average total

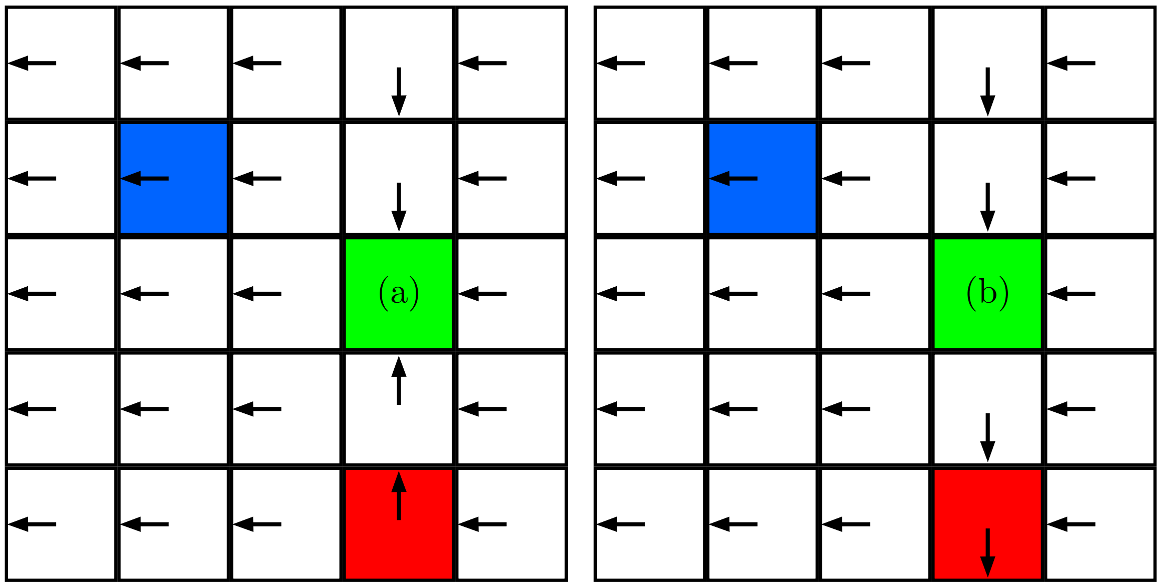


Figure 3.11: (a) The action selected by the agent using the best performing network-based policies if the blue square were in each grid spacing for an example of *Direct Control Grid-World* and (b) *Probabilistic Control Grid-World* environments.

reward of 5.75 or greater for 100 episodes in a row. The same example as seen before, but in *Probabilistic Control Grid-World*, for what this agent would do in each grid spacing is shown in Figure 3.5(b). This agent behaves very similarly to the one in used in *Direct Control Grid-World* however the main difference is that this agent has learned to ignore the move up action as well as the move right action. Again, despite this odd result, the agent still performs quite well. With this happening in both the *Direct Control Grid-World* and *Probabilistic Control Grid-World* environments, the result is worth noting for the future in case it arises again in a more meaningful environment but poses no major problems now.

Chapter 4

Learning to Maximize Thermal Efficiency

Now that it has been shown that our GA optimized neural network-based policy method works on well known RL problems, we decided to use this method on another well known problem, but one that has not been considered in RL before. Heat engines have been used throughout the world for centuries, and still are today. Operating one of these devices is naturally a control problem, making it an appealing candidate for RL. As with the cases before, we do not encode or provide any human knowledge to our agent beforehand. The goal is to allow the agent to learn any physics it requires during the learning process, deciding for itself what is and is not important.

4.1 Heat Engines

A heat engine is a device that transforms thermal energy into work, and an example of one is shown in Figure 4.1. The heat engine we are considering can compress or expand the working substance, as well as be connected to a hot or cold reservoir. To

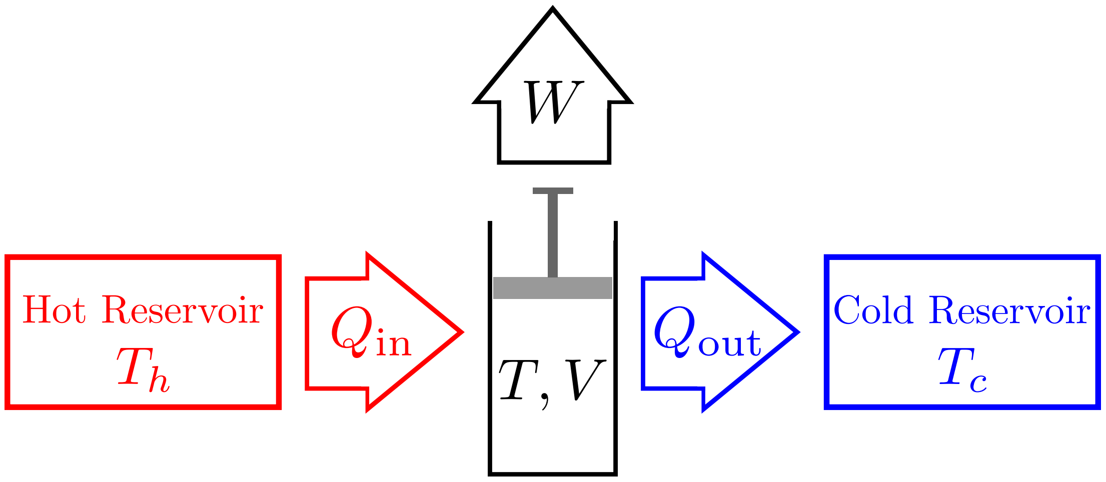


Figure 4.1: A simple example of a cylindrical heat engine showing the energy flow in from the hot reservoir and out to the cold reservoir, producing work.

maintain temperature when connected to a reservoir, the thermal energy gained or lost by compression or expansion is transferred from or to that reservoir. By using the cold reservoir to absorb the thermal energy produced by compressing the working substance and the hot reservoir to provide the thermal energy lost by expanding the working substance, work is done on the surroundings. If the heat engine is perfectly insulated, compression and expansion while disconnected from both reservoirs is adiabatic. However if the heat engine is not perfectly insulated, the effects of compression and expansion while disconnected from both reservoirs are irreversible. The possible actions that can be taken on the heat engine are shown in Figure 4.2(a) with their behavior on a pressure vs volume plot in Figure 4.2(b). Assuming the working substance of the heat engine is an ideal gas, the necessary properties can be modelled using the ideal gas law [61] defined as

$$PV = Nk_BT, \tag{4.1}$$

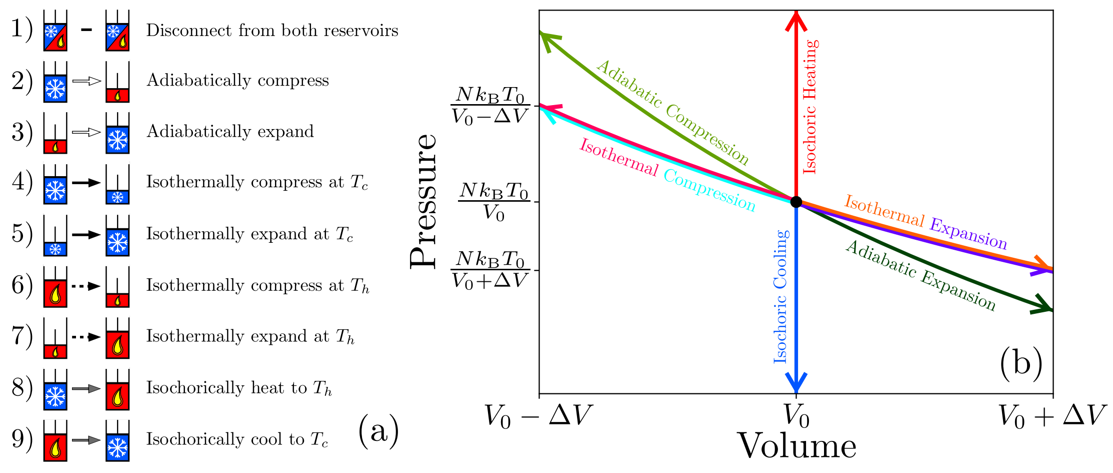

1) Disconnect from both reservoirs
2) Adiabatically compress
3) Adiabatically expand
4) Isothermally compress at T_c
5) Isothermally expand at T_c
6) Isothermally compress at T_h
7) Isothermally expand at T_h
8) Isochorically heat to T_h
9) Isochorically cool to T_c
(a)
Pressure
$\frac{Nk_B T_0}{V_0-\Delta V}$
$\frac{Nk_B T_0}{V_0}$
$\frac{Nk_B T_0}{V_0+\Delta V}$
Adiabatic Compression
Isochoric Heating
Isothermal Compression
Isothermal Expansion
Adiabatic Expansion
Isochoric Cooling
(b)
$V_0 - \Delta V$
V_0
$V_0 + \Delta V$
Volume

Figure 4.2: (a) Graphical representations and legend of all the possible actions that can be taken on the simple heat engine described in this section with (b) the behavior of all the actions in P-V space.

where P is the pressure of the system, V is the volume of the system, N is the number of molecules of the working substance, k_B is the Boltzmann constant, and T is the temperature of the system. For the isothermal and isochoric processes, the behavior of Equation 4.1 is trivial as V is the independent variable and T is constant or linear. However for the adiabatic processes, the behavior is less trivial, with

$$TV^{\gamma-1} = C \tag{4.2}$$

holding true during the process, where C is a constant, and $\gamma = \frac{C_P}{C_V}$ where $C_P = \frac{5}{2}Nk_B$ and $C_V = \frac{3}{2}Nk_B$ are the isobaric and isochoric heat capacities for a monatomic ideal gas, respectively. The thermal efficiency of these heat engines is calculated by

$$\eta = \frac{W}{Q_{\text{in}}}, \tag{4.3}$$

where W is the total work, and Q_{in} is the amount of energy put into the system from the hot reservoir. For an ideal gas W and Q_{in} can be calculated exactly and are

Action	ΔW	ΔQ_{in}
Adiabatic Compression	$\frac{3}{2}Nk_{\text{B}}T_{\text{i}}\left(1-\left(\frac{V_{\text{i}}}{V_{\text{f}}}\right)^{\frac{2}{3}}\right)$	0
Adiabatic Expansion	$\frac{3}{2}Nk_{\text{B}}T_{\text{i}}\left(1-\left(\frac{V_{\text{i}}}{V_{\text{f}}}\right)^{\frac{2}{3}}\right)$	0
Isothermal Compression (T_{h})	$Nk_{\text{B}}T_{\text{h}}\log\left(\frac{V_{\text{f}}}{V_{\text{i}}}\right)$	0
Isothermal Expansion (T_{h})	$Nk_{\text{B}}T_{\text{h}}\log\left(\frac{V_{\text{f}}}{V_{\text{i}}}\right)$	$Nk_{\text{B}}T_{\text{h}}\log\left(\frac{V_{\text{f}}}{V_{\text{i}}}\right)$
Isothermal Compression (T_{c})	$Nk_{\text{B}}T_{\text{c}}\log\left(\frac{V_{\text{f}}}{V_{\text{i}}}\right)$	0
Isothermal Expansion (T_{c})	$Nk_{\text{B}}T_{\text{c}}\log\left(\frac{V_{\text{f}}}{V_{\text{i}}}\right)$	0
Isochoric Heating	0	$\frac{3}{2}Nk_{\text{B}}\left(T_{\text{h}}-T_{\text{i}}\right)$
Isochoric Cooling	0	0

Table 4.1: The W and Q_{in} equations for each action that can be taken on a simple heat engine that is shown in Figure 4.2.

shown for each process with a discrete change in volume in Table 4.1.

4.1.1 Carnot Cycle

In 1824, Carnot's theorem [1] was developed, which states that the maximum thermal efficiency, η_{max}, of any heat engine is dependent on the temperatures of the reservoirs and using Equation 4.3 and Table 4.1 it can be derived as follows;

$$\begin{aligned}\eta_{\text{max}} = & \frac{Nk_{\text{B}}T_{\text{c}}\log\left(\frac{V_y}{V_{\text{max}}}\right)+\frac{3}{2}Nk_{\text{B}}T_{\text{c}}\left(1-\left(\frac{V_y}{V_{\text{min}}}\right)^{\frac{2}{3}}\right)}{Nk_{\text{B}}T_{\text{h}}\log\frac{V_x}{V_{\text{min}}}} \\ & + \frac{Nk_{\text{B}}T_{\text{h}}\log\left(\frac{V_x}{V_{\text{min}}}\right)+\frac{3}{2}Nk_{\text{B}}T_{\text{h}}\left(1-\left(\frac{V_x}{V_{\text{max}}}\right)^{\frac{2}{3}}\right)}{Nk_{\text{B}}T_{\text{h}}\log\frac{V_x}{V_{\text{min}}}},\end{aligned}$$

where

$$V_y = \left(\frac{T_{\text{h}}}{T_{\text{c}}}\right)^{\frac{3}{2}} V_{\text{min}}, \tag{4.4}$$

and

$$V_x = \left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}} V_\mathrm{max}, \tag{4.5}$$

are the volumes that the adiabatic compression and expansion processes are started at respectively. Plugging those in and simplifying, we get

$$\eta_\mathrm{max} = \frac{T_\mathrm{c}\log\left(\left(\frac{T_\mathrm{h}}{T_\mathrm{c}}\right)^{\frac{3}{2}}\frac{V_\mathrm{min}}{V_\mathrm{max}}\right) + \frac{3}{2}T_\mathrm{c}\left(1-\frac{T_\mathrm{h}}{T_\mathrm{c}}\right) + T_\mathrm{h}\log\left(\left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}}\frac{V_\mathrm{max}}{V_\mathrm{min}}\right) + \frac{3}{2}T_\mathrm{h}\left(1-\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)}{T_\mathrm{h}\log\left(\left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}}\frac{V_\mathrm{max}}{V_\mathrm{min}}\right)}.$$

Rearranging terms using the following log identity

$$\begin{aligned}\log\left(\left(\frac{T_\mathrm{h}}{T_\mathrm{c}}\right)^{\frac{3}{2}}\frac{V_\mathrm{min}}{V_\mathrm{max}}\right) &= -\log\left(\left(\frac{T_\mathrm{h}}{T_\mathrm{c}}\right)^{\frac{3}{2}}\frac{V_\mathrm{min}}{V_\mathrm{max}}\right)^{-1} \\ &= -\log\left(\left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}}\frac{V_\mathrm{max}}{V_\mathrm{min}}\right),\end{aligned}$$

and using the fact that

$$T_\mathrm{c}\left(1-\frac{T_\mathrm{h}}{T_\mathrm{c}}\right) = -T_\mathrm{h}\left(1-\frac{T_\mathrm{c}}{T_\mathrm{h}}\right),$$

we then get

$$\eta_\mathrm{max} = \frac{T_\mathrm{h}\log\left(\left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}}\frac{V_\mathrm{max}}{V_\mathrm{min}}\right) - T_\mathrm{c}\log\left(\left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}}\frac{V_\mathrm{max}}{V_\mathrm{min}}\right)}{T_\mathrm{h}\log\left(\left(\frac{T_\mathrm{c}}{T_\mathrm{h}}\right)^{\frac{3}{2}}\frac{V_\mathrm{max}}{V_\mathrm{min}}\right)},$$

then simplifying we end up with

$$\eta_\mathrm{max} = \frac{T_\mathrm{h}-T_\mathrm{c}}{T_\mathrm{h}}. \tag{4.6}$$

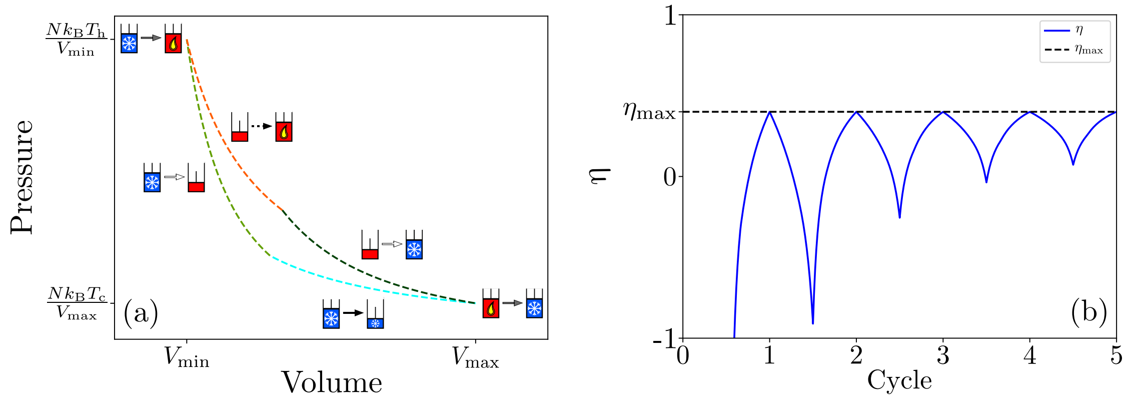


Figure 4.3: (a) The phase plot of a heat engine as it performs the Carnot cycle with each action being taken labeled with its graphical representation and a different colour. (b) The thermal efficiency of a heat engine as it performs the Carnot cycle several times with $\eta_{\max}$ for reference.

$\eta_{\max}$ can only be achieved by performing a specific set of actions on the heat engine, which creates a cycle known as the Carnot cycle. This cycle is shown in Figure 4.3(a), with the thermal efficiency for several cycles shown in Figure 4.3(b). Starting at the maximum volume, $V_{\max}$, the heat engine is compressed isothermally while connected to the cold reservoir until the engine approaches V_y. Next the engine is adiabatically compressed until the temperature reaches T_h and the volume reaches the minimum, $V_{\min}$. During the compression steps the heat engine is performing work on the working substance, therefore the thermal efficiency decreases during this part of the cycle. The engine, at $V_{\min}$, is then expanded isothermally while connected to the hot reservoir until the engine approaches V_x. Finally the engine is adiabatically expanded until the temperature reaches T_c and the volume reaches $V_{\max}$, ending at the starting point of the cycle, extracting the most possible W given a fixed Q_{in}. During the expansion steps the working substance is performing work on the heat engine, therefore the thermal efficiency increases during this part of the cycle, explaining the oscillating behavior seen in Figure 4.3(b).

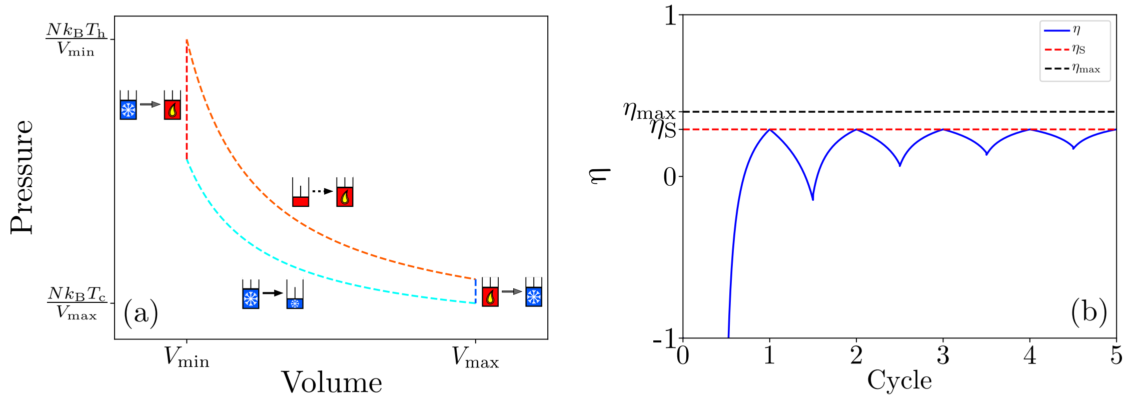


Figure 4.4: (a) The phase plot of a heat engine as it performs the Stirling cycle with each action being taken labeled with their graphical representation. (b) The thermal efficiency of a heat engine as it performs the Stirling cycle several times with η_{max} and η_S for reference.

4.1.2 Stirling Cycle

The Stirling cycle is similar to the Carnot cycle; the major difference comes from replacing the adiabatic processes with isochoric processes. This cycle is shown in Figure 4.4(a), with the thermal efficiency for several cycles shown in Figure 4.4(b). Staring at V_{max}, the engine is compressed isothermally while connected to the cold reservoir until it reaches V_{min}. Next the engine is connected to the hot reservoir, allowing the body to heat isochorically until the temperature reaches T_h. The engine is then expanded isothermally until the volume reaches V_{max}. Finally the engine is connected to the cold reservoir, allowing the body to cool isochorically until the temperature reaches T_c. If a regenerative device is used to exchange internal heat which would otherwise be lost during the isochoric cooling, the Stirling thermal efficiency, η_S, is the same as η_{max}. However, without such device, using Equation 4.3 and Table 4.1, η_S can be derived as follows;

$$\eta_S = \frac{Nk_B T_c \log\left(\frac{V_{min}}{V_{max}}\right) + Nk_B T_h \log\left(\frac{V_{max}}{V_{min}}\right)}{\frac{3}{2}Nk_B\left(T_h - T_c\right) + Nk_B T_h \log\left(\frac{V_{max}}{V_{min}}\right)}.$$

Using the same log identity from deriving Equation 4.6 and simplifying we then get

$$\eta_{\mathrm{S}} = \frac{T_{\mathrm{h}} - T_{\mathrm{c}}}{T_{\mathrm{h}} + \frac{3(T_{\mathrm{h}} - T_{\mathrm{c}})}{2\log\left(\frac{V_{\max}}{V_{\min}}\right)}},$$

and for a cleaner equation we can replace some terms with already defined thermodynamic terms we end up with

$$\eta_{\mathrm{S}} = \frac{T_{\mathrm{h}} - T_{\mathrm{c}}}{T_{\mathrm{h}} + \frac{\Delta U_V}{\Delta S_T}}, \tag{4.7}$$

where ΔU_V is the change in internal energy for an isochoric process, and ΔS_T is the change in entropy for an isothermal process, defined respectively as

$$\Delta U_V = \frac{3}{2} N k_{\mathrm{B}} \left(T_{\mathrm{h}} - T_{\mathrm{c}}\right), \tag{4.8}$$

and

$$\Delta S_T = N k_{\mathrm{B}} \log\left(\frac{V_{\max}}{V_{\min}}\right). \tag{4.9}$$

4.1.3 Otto Cycle

The Otto cycle was designed in 1861 to be used on four-stroke engines. The Otto cycle is similar to the Stirling cycle; the major difference comes from replacing the isothermal processes with adiabatic processes. In the case of a four-stroke engine, there is also air intake and outtake processes, however we will only be considering the Otto cycle for the simple heat engine described before. With the air intake and outtake steps omitted, the Otto cycle forms a closed single directional cycle on a pressure vs volume plot shown in Figure 4.5(a), with the thermal efficiency for several cycles shown in Figure 4.5(b). Starting at $V_{\max}$, the engine is compressed

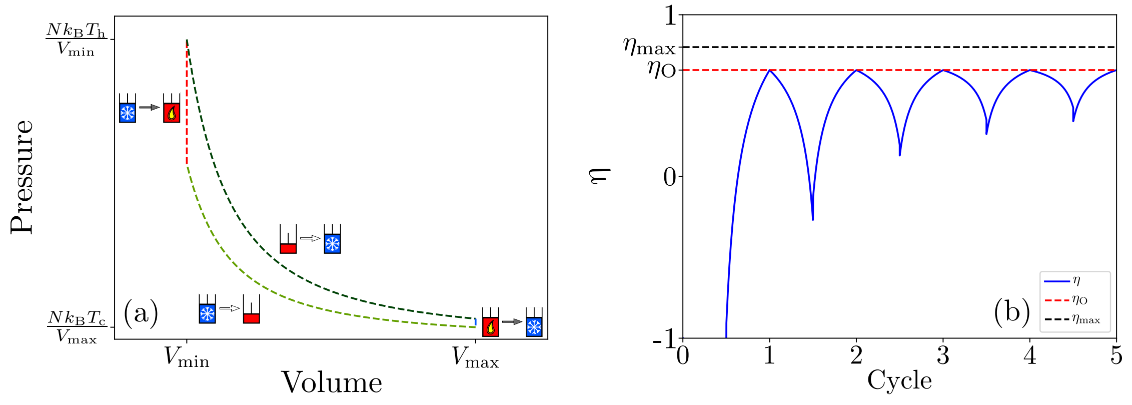


Figure 4.5: (a) The phase plot of a heat engine as it performs the Otto cycle with each action being taken labeled with their graphical representation. (b) The thermal efficiency of a heat engine as it performs the Otto cycle several times with η_max and η_S for reference.

adiabatically until the volume reaches V_min. The engine is then connected to the hot reservoir, allowing the body to heat isochorically until the temperature reaches T_h. Next the engine is expanded adiabatically until the volume reaches V_max. Lastly, the engine is connected to the cold reservoir, allowing the body to cool isochorically until the temperature reaches T_c, ending at the beginning of the cycle. When performing the Otto cycle on a simple heat engine the Otto efficiency, η_O, using Equation 4.3 and Table 4.1, η_O can be derived as follows;

$$\eta_\mathrm{O} = \frac{\frac{3}{2}Nk_BT_\mathrm{c}\left(1-\left(\frac{V_\mathrm{max}}{V_\mathrm{min}}\right)^{\frac{2}{3}}\right)+\frac{3}{2}Nk_BT_\mathrm{h}\left(1-\left(\frac{V_\mathrm{min}}{V_\mathrm{max}}\right)^{\frac{2}{3}}\right)}{\frac{3}{2}Nk_\mathrm{B}\left(T_\mathrm{h}-T_i\right)},$$

where T_i is the temperature of the system when the isothermal heating process is initially performed and can be determined using Equation 4.2 knowing that the adiabatic compression process was performed from $V_{\max}$ to $V_{\min}$ starting at T_{c} by

$$\begin{aligned}
T_i V_{\min}^{\frac{2}{3}} &= C \\
T_{\text{c}} V_{\max}^{\frac{2}{3}} &= C \\
T_i V_{\min}^{\frac{2}{3}} &= T_{\text{c}} V_{\max}^{\frac{2}{3}} \\
T_i &= T_{\text{c}} \left(\frac{V_{\max}}{V_{\min}}\right)^{\frac{2}{3}}.
\end{aligned} \tag{4.10}$$

Plugging this in for T_i and simplifying we end up with

$$\eta_{\text{O}} = \frac{T_h\left(1-\left(\frac{V_{\min}}{V_{\max}}\right)^{\frac{2}{3}}\right) + T_c\left(1-\left(\frac{V_{\max}}{V_{\min}}\right)^{\frac{2}{3}}\right)}{T_h - T_c\left(\frac{V_{\max}}{V_{\min}}\right)^{\frac{2}{3}}}. \tag{4.11}$$

4.2 Heat Engine Environment

To model a heat engine, we created a simple environment that an agent can interact with. The environment is initialized with an engine at a volume of $V_{\max}$, and a temperature of T_c. The state of this environment is the current temperature, T, and the current volume, V, of the system. The current pressure, P, could also have been used in the state with both T and V or in lieu of either of them, however it is not necessary due to their relation through Equation 4.1. T was chosen due to the control from the isothermal and isochoric processes, and V was chosen due to the control with compression and expansion processes, while P is the dependant variable for all the processes in this heat engine. The actions the agent can take are the ones shown in Figure 4.2 where all compression and expansion actions are done using a fixed ΔV, unless otherwise stated. If an action is taken that would increase V above $V_{\max}$ or decrease V below $V_{\min}$, the state remains unchanged. After a fixed number of steps, the maximum η is used as the score of the game. To ensure the engine is usable for more than one cycle, a penalty is applied to the score of any policy that causes the engine to get stuck at a constant V. For this study, we used $V_{\text{min}} = 2 \times 10^{-4}$ m^3, $V_{\text{max}} = 1 \times 10^{-3}$ m^3, $T_c = 300$ K, $T_h = 500$ K, and ΔV values of 5×10^{-5} m^3, 1×10^{-4} m^3, and 2×10^{-4} m^3. The environment is always initialized at V_{max} and T_c because if it is initialized at any other volume it is possible to achieve a thermal efficiency of ∞. With all actions available, the most efficient cycle possible is the Carnot cycle, therefore this first environment will be referred to as the *Carnot* environment. Using Equation 4.6 with these parameters, $\eta_{\text{max}} = 0.4$.

A second heat engine environment was created, which is identical to the original one, except the adiabatic actions are unavailable. This second environment will be referred to as the *Stirling* environment as the Stirling cycle is the most efficient cycle possible with this reduced action space. Using Equation 4.7 with these parameters, $\eta_{\text{S}} = 0.291$.

A third heat engine environment was created, which is another copy of the original one, except the isothermal actions are unavailable. This environment will be referred to as the *Otto* environment as the Otto cycle is the most efficient cycle possible in this environment. T_h had to be increased to 1500 K for this environment due to the high temperatures that can be reached through adiabatic compression. Using Equation 4.11 with these parameters, $\eta_{\text{O}} = 0.658$, and using Equation 4.6 for comparison, $\eta_{\text{max}} = 0.8$ in this environment.

A fourth heat engine environment was created, which includes the full action set and the agent additionally chooses which ΔV to use from: 1x10^{-4} m^3, 1x10^{-5} m^3, 1x10^{-6} m^3, 1x10^{-7} m^3, or 1x10^{-8} m^3. This heat engine environment will be referred to as the *Variable* ΔV *Carnot* environment.

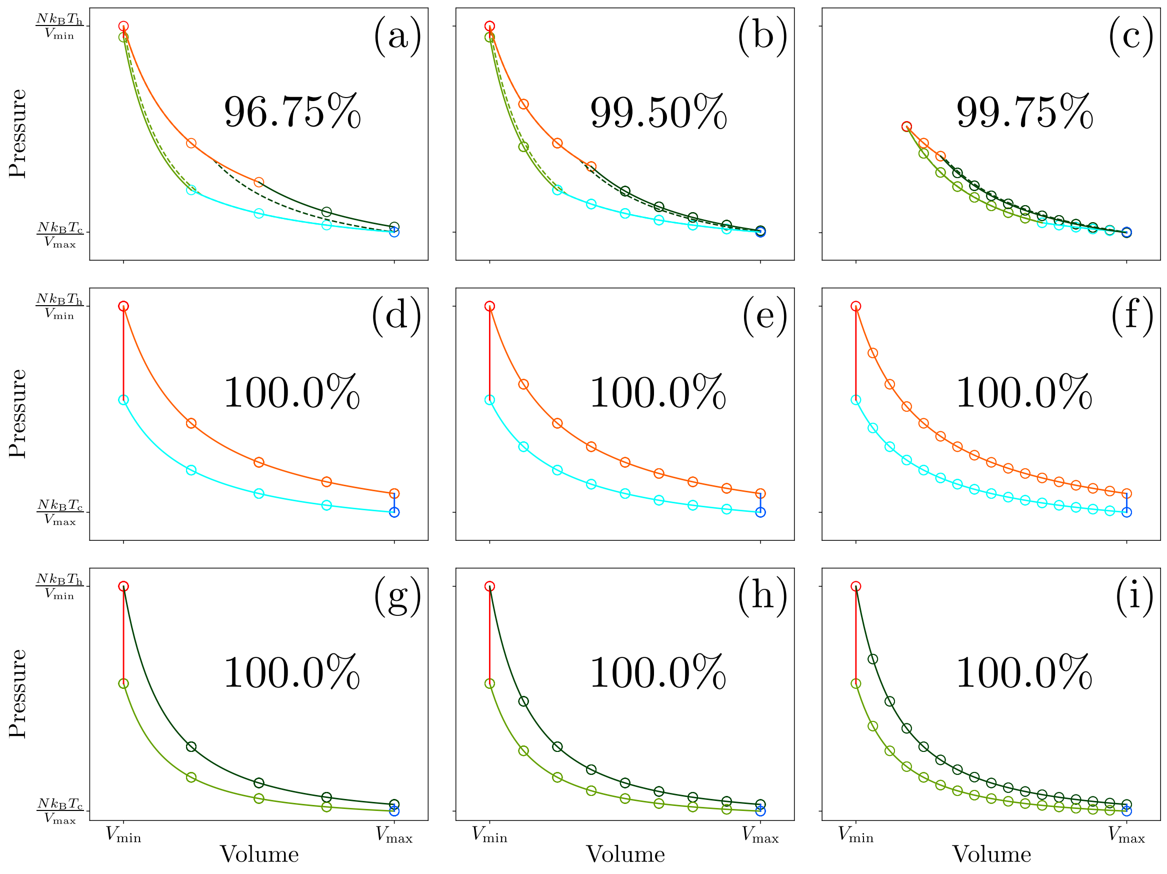


Figure 4.6: (a) Our trained network based policy agents as they act on the *Carnot* environment with the exact Carnot cycle for reference and a ΔV of 2x10^{-4} m^3, (b) 1x10^{-4} m^3, and (c) 5x10^{-5} m^3, (d) the *Stirling* environment with a ΔV of 2x10^{-4} m^3, (e) 1x10^{-4} m^3, and (f) 5x10^{-5} m^3, and (g) the *Otto* environment with a ΔV of 2x10^{-4} m^3, (h) 1x10^{-4} m^3, and (i) 5x10^{-5} m^3 each labeled with their percent accuracy.

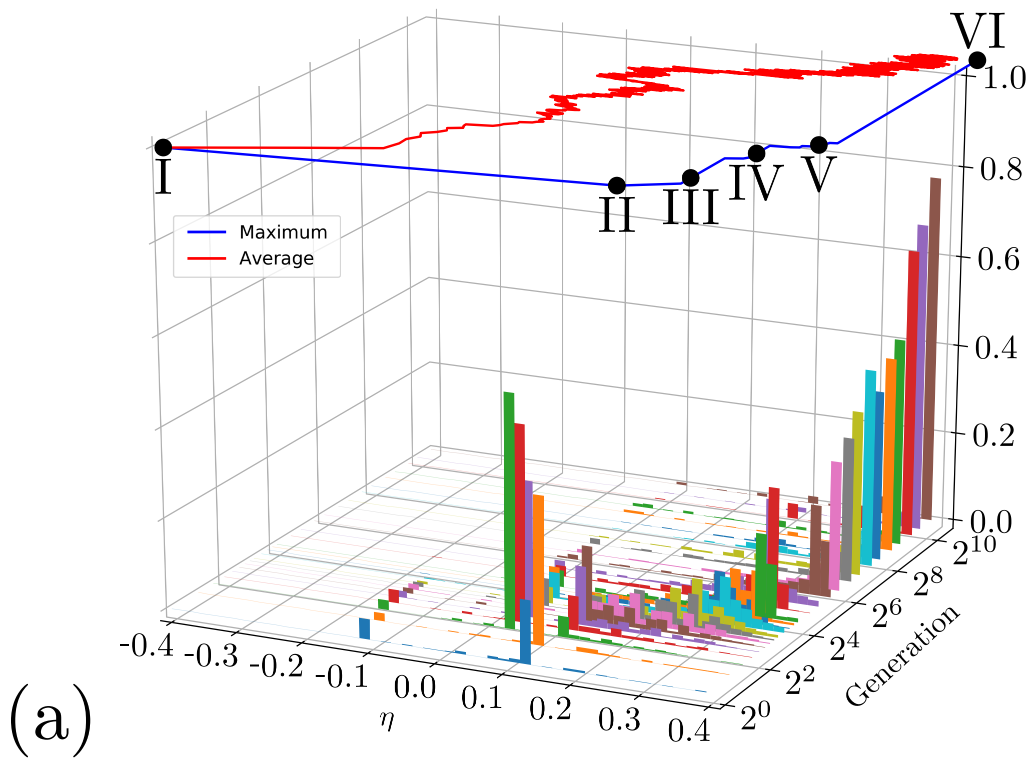
4.3 Results

With the capability of our GA optimization method shown on the benchmark environments, we tested it on the heat engine environments from Section 4.2 [62]. For this test we first ran our GA algorithm on the *Carnot* environment using the large ΔV of 2x10^{-4} m^3 to contain the number of possible states. The network based policy was able to achieve a maximum thermal efficiency of 0.387, less than $\eta_{\max} = 0.4$. As seen in Figure 4.6(a), the network based policy learned a similar cycle as the one seen in Figure 4.3(a). While T and V are used as inputs to the neural network-based policy being used in this study, the Carnot cycle is most commonly shown in P-V space, so we chose to follow this convention for plotting purposes. Using the same network architecture, this process was repeated using the *Stirling* and *Otto* environments, yielding a maximum thermal efficiency of 0.291 and 0.658 respectively, the exact efficiency values as η_{S} and η_{O}. Unlike with the *Carnot* environment, as seen in Figures 4.6(d) and 4.6(g), the network based policy was able to reproduce the exact cycles on the *Stirling* and *Otto* environments shown in Figures 4.4(a) and 4.5(a) respectively.

Now that we have shown that the network based policy performs well at a ΔV of 2x10^{-4} m^3, we reduced ΔV to more useful values of 1x10^{-4} m^3 and 5x10^{-5} m^3, then trained the network based policy GA again on each of the three environments already tested. As the network based policy was able to achieve a maximum thermal efficiency of η_{S} in the *Stirling* environment with a large ΔV, it should be able to achieve a maximum thermal efficiency η_{S} on any ΔV which 2x10^{-4} m^3 is a integer multiple of. As seen in Figures 4.6(e) and 4.6(f), the network based policy was able to produce the exact Stirling cycle in the *Stirling* environment, with a maximum thermal efficiency of $\eta_{\text{S}} = 0.291$ as expected. Similarly, the same should be expected from the *Otto* environment. As seen in Figures 4.6(h) and 4.6(i), again the network based policy was able to produce the Otto cycle in the *Otto* environment, with a maximum thermal efficiency of $\eta_{\text{O}} = 0.658$ for both additional ΔV values.

Unlike the *Stirling* and *Otto* environments, it was not possible to achieve the maximum thermal efficiency of η_{max} in the *Carnot* environment using the large ΔV of 2x10^{-4} m^3. The main difference between achieving η_{max} and achieving η_{S} or η_{O} comes from the specific volumes at which certain actions need to be taken. With the Stirling and Otto cycles, actions are only ever changed at V_{min} and V_{max}, where the Carnot cycle requires adiabatic actions starting at other V values, therefore it is expected that as we decrease ΔV, the maximum thermal efficiency our agent can achieve in the *Carnot* environment will increase. Using a ΔV of 1x10^{-4} m^3, the network based policy was trained on the *Carnot* environment again, yielding a maximum thermal efficiency of 0.398, higher than the maximum thermal efficiency found when using a ΔV of 2x10^{-4} m^3. As seen in Figure 4.6(b), the cycle produced by our agent using this smaller ΔV more closely resembles the actual Carnot cycle shown in Figure 4.3(a), however it is still not the exact cycle, therefore ΔV was decreased again to 5x10^{-5} m^3. With this even further decreased ΔV on the *Carnot* environment, the network based policy was able to achieve a maximum thermal efficiency of 0.3993, even closer to $\eta_{\max}$ than with the previous ΔV. As seen in Figure 4.6(c), unlike every other case seen so far, the optimal cycle with this ΔV does not use the full available set of volumes. This shows that, unlike the Stirling and Otto cycles, the volume of our system is not important for thermal efficiency. This can be seen when comparing Equation 4.6 to Equations 4.7 and 4.11. What is important for maximizing thermal efficiency in our *Carnot* environment is being able to go from T_c to T_h without isochoric actions after isothermally compressing the system by some amount while connected to the cold reservoir, and being able to go from T_h to T_c without isochoric actions after isothermally expanding the system by some amount while connected to the hot reservoir. For this reason, to achieve a maximum thermal efficiency of $\eta_{\max}$, ΔV must be small enough that the system can reach the exact V values required for the adiabatic actions to be started.

After training our GA algorithm on the *Variable* ΔV *Carnot* environment, the network based policy was able to achieve a maximum thermal efficiency of 0.39995. Not only does the best performing policy learn to do this, but the population learns and shifts from performing inefficient cycles to performing highly efficient cycles as seen in Figure 4.7(a). This case is with 1000 policies, instead of 100, for statistical purposes. Any policies that have undefined efficiency's or became stuck at some point



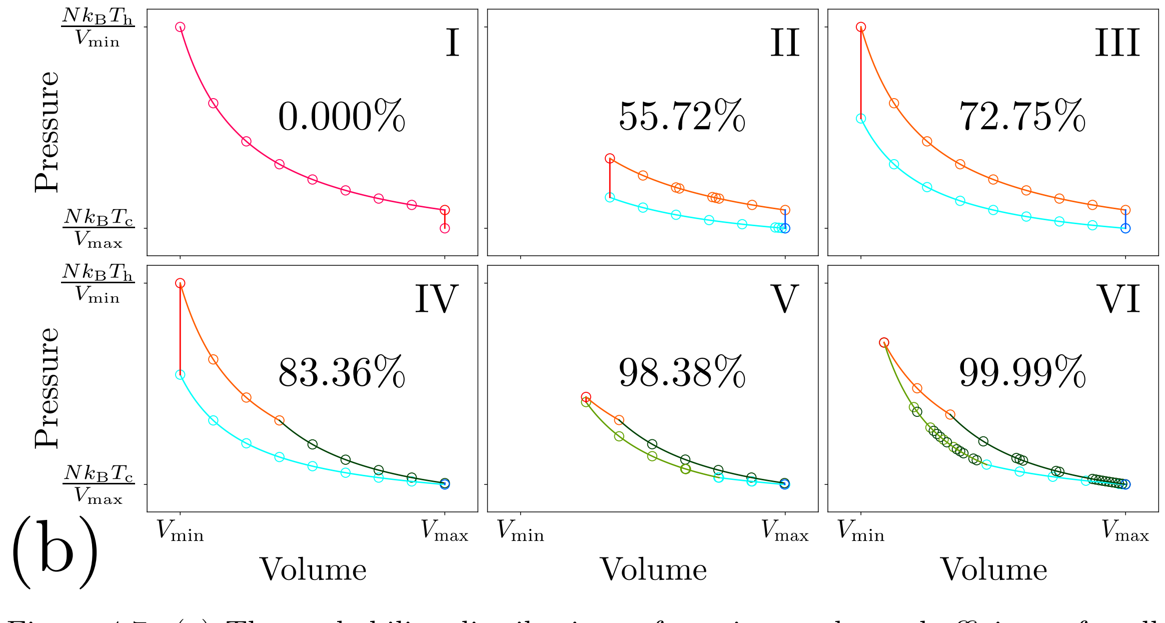


Figure 4.7: (a) The probability distributions of maximum thermal efficiency for all 1000 policies of a population as they act on the *Variable* ΔV *Carnot* environment over several generations. (b) The results of the best performing policy for generations 2^0, 2^1, 2^2, 2^4, 2^5, and 2^{12} each labeled with their percent accuracy.

do not appear in the distribution, however they are still taken into account. As the generations progress, the more of the population appears in the distribution as they start producing cycles. Once a few policies reach high maximum thermal efficiency, the population distribution starts to shift towards the maximum. Towards the later generations the population converges on the highest maximum thermal efficiency with the average of the elite population being equal to the best performing policy, showing that the training procedure is concluding. In Figure 4.7(b), the agent can be seen learning, first producing no cycle, then smaller low efficiency cycles, and eventually performing a near perfect Carnot cycle in this environment, with a maximum thermal efficiency with less than 0.02% error.

Having seen how the distributions of scores and elite population policies evolve over generations, shown in Figure 4.7, we investigated how the entire population changes and how the initial policies propagate. To do this, we arbitrarily assign each of the initial 100 policies an ID number from 0 to 99. At each generation when a new policy is created through mutation, it is assigned the same ID as the policy it was mutated from. Once every policy has the same ID number, we reassign IDs between 100 and 199 and continue, increasing the IDs by 100 every time we reassign them. These population dynamics are shown in Figure 4.8. Starting at generation 0, there are 100 policies with unique IDs 0 through 99 as they are the initial population. At generation 1, 75 of the unique IDs are lost and each is replaced by one of the remaining 25 as these are the IDs of the initial 25 best performing policies. After generation 3 every policy is one that was created by mutating policy 61 one or more times and thus the 100 policies with the ID number of 61 are reassigned unique IDs 100 through 199. After generation 16 this happens once again with policy 168, however it took four times as many generations for this second ID convergence to occur. This is likely due to the fact that policies 100 through 199 are much more similar than policies 0

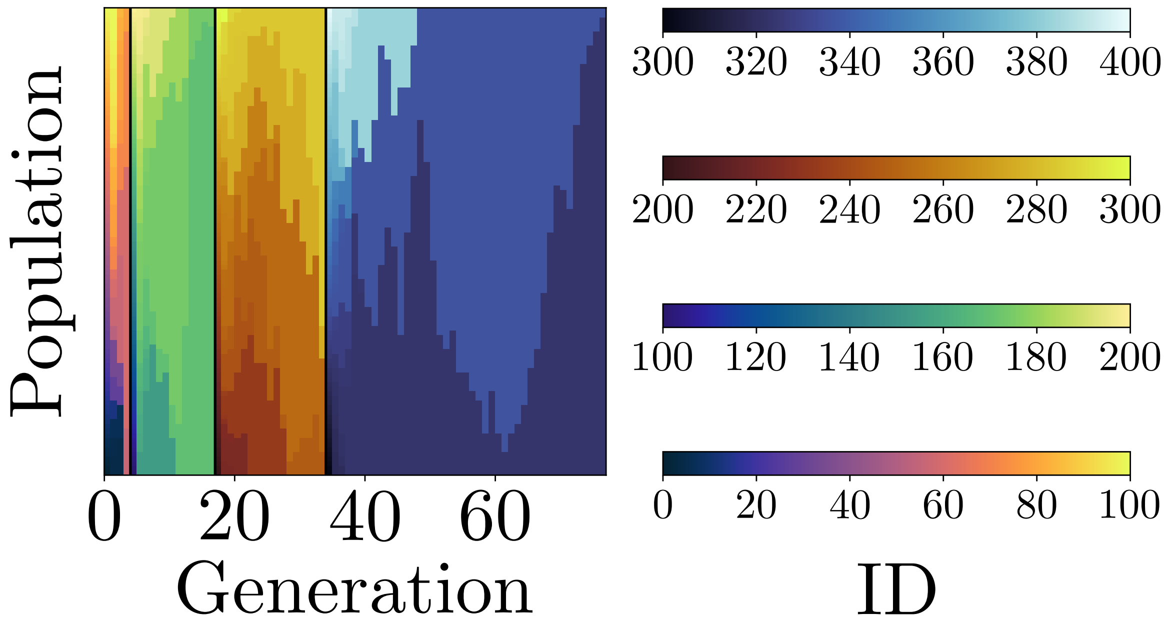


Figure 4.8: The ID numbers of each policy in the population as a function of generation. A policy created through mutation is assigned the ID number of the policy it was mutated from. The ID numbers are reassigned at any generation that all policies have the same ID and a vertical black line denotes this reassignment.

through 99 were. After reassigning unique IDs 200 through 299, this trend is seen again with the third ID convergence not occurring until after generation 33 with policy 284 and even more so in the forth ID convergence not occurring until after generation 76 with policy 322.

With the capability of our network based GA policy on the *Variable* ΔV *Carnot* environment shown, we wanted to test this method on an unknown problem. Therefore a fifth heat engine environment was created, which is a copy of the *Variable* ΔV *Carnot* environment, however the adiabatic processes are altered so they are irreversible. Modifying Equation 4.2 such that the heat engine would lose a fraction of its thermal energy over the fully available volume, the new irreversible process is modelled with

$$TV^{\gamma-1} = C\left(1-K\right)^{f(V;V_i)}, \tag{4.12}$$

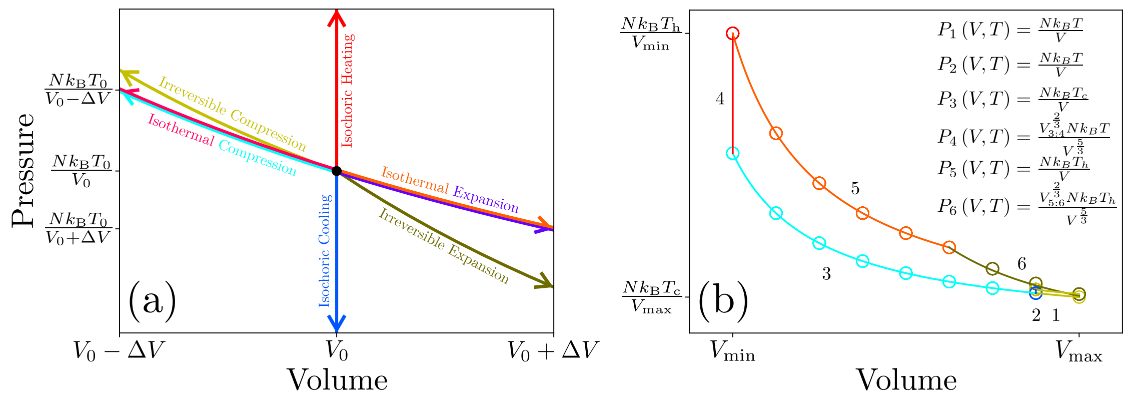


Figure 4.9: (a) The behavior all the actions that can be taken on the simple heat engine with the irreversible processes instead of the adiabatic processes in P-V space. (b) Our trained network based policy agent acting on the *Heat Loss Carnot* environment with each action being taken labeled with its learned equation where $V_{x:y}$ is the volume at which segments x and y intersect.

where $K \in [0,1)$ is the heat loss constant, $f(V;V_i)$ is the relative absolute change in volume defined as

$$f(V;V_i) = \frac{|V-V_i|}{V_{\max}-V_{\min}}, \tag{4.13}$$

and V_i is the starting volume for this process. This fifth heat engine environment will be referred to as the *Heat Loss Carnot* environment. Unlike the other four environments, this fifth one contains a function of our own design, and therefore the most efficient cycle possible is not yet known to us. The behavior of the possible actions that can be taken on the heat engine in the *Heat Loss Carnot* environment are shown on a pressure vs volume plot in Figure 4.9(a). We trained our network based policy on our *Heat Loss Carnot* environment using $K = 0.4$ for the irreversible processes described in Equation 4.12 to determine what the resulting cycle will be. After training our network based policy on the *Heat Loss Carnot* environment it was able to achieve a maximum thermal efficiency of 0.311. This new cycle shown in Figure 4.9(b) that we have learned from our agent is a hybrid of the Stirling and Carnot cycles, where the first half represents Stirling, and the second half represents

Carnot. The likely reason for this combination is due to the increased difficulty of using compression to increase T from T_c to T_h with the irreversible compression process. Hence the first half representing the Stirling cycle, as it does not use the irreversible compression. However, the difficulty of using expansion to decrease T of the system from T_h to T_c with the irreversible expansion has decreased, hence the second half representing the Carnot cycle. Using a combination of grid searching and a least squares method we were able to fit the same flexible function with different parameters to each segment of this new cycle. This shows that if our agent were to act on a non-ideal system, much like the *Heat Loss Carnot*, we can also use this function fitting method to learn the details about this system. To fit the data produced by the agent acting on the heat engine environment we used a function flexible enough that it can be used to fit both ideal and van der Waals gases for, isothermal, adiabatic, and irreversible compression and expansion as well as isochoric heating and cooling. The flexible function used is

$$P\left(V,T\right)=\frac{Nk_{\mathrm{B}}CT^{x_1}\left(1-K\right)^{f(V;V_i)}}{V^{x_2}-nb}-\frac{an^2}{V^2}, \tag{4.14}$$

where C is a general constant, x_1 is the Boolean exponent which determines if T is used in the equation, x_2 is the volume exponent which is either 1 or γ for the gas, a is a constant specific to the gas, and b is the volume per mole that is occupied by the molecules. To optimize this equation for a specific segment of the a thermodynamic cycle, x_1, x_2, a, and b are discretized and iterated over while C and K are fit using a least squares method for each x_1, x_2, a, and b grouping.

Using the results obtained by our agent, we can define this new cycle that will be referred to as the Heat Loss cycle. The Heat Loss efficiency, η_{H}, using Equations 4.3

and 4.12 and Table 4.1, can be derived as follows;

$$\eta_{\mathrm{H}} = \frac{Nk_{\mathrm{B}}T_{\mathrm{c}}\log\left(\frac{V_{\min}}{V_{\max}}\right) + Nk_{\mathrm{B}}T_{\mathrm{h}}\log\left(\frac{V_{x'}}{V_{\min}}\right) + Nk_{\mathrm{B}}C_2\int_{V_{x'}}^{V_{\max}}(1-K)^{f\left(V;V_{x'}\right)}V^{-\frac{2}{3}}dV}{\frac{3}{2}Nk_{\mathrm{B}}\left(T_{\mathrm{h}}-T_{\mathrm{c}}\right) + Nk_{\mathrm{B}}T_{\mathrm{h}}\log\left(\frac{V_{x'}}{V_{\min}}\right)},$$

where

$$V_{x'} = \left(\frac{T_{\mathrm{c}}}{T_{\mathrm{h}}}\right)^{\frac{3}{2}}\frac{V_{\min}}{(1-K)^{\frac{3}{2}f\left(V_{\min};V_{x'}\right)}}, \tag{4.15}$$

is the volume that the irreversible expansion process is started at and C_2 can be solved for with T_{h} and $V_{x'}$ using Equation 4.2. Plugging $V_{x'}$ in and simplifying, we get

$$\eta_{\mathrm{H}} = \frac{T_{\mathrm{c}}\log\left(\frac{V_{\min}}{V_{\max}}\right) + \frac{3}{2}T_{\mathrm{h}}\log\left(\frac{T_{\mathrm{c}}}{T_{\mathrm{h}}(1-K)^{f\left(V_{\min};V_{x'}\right)}}\right) + C_2\int_{V_{x'}}^{V_{\max}}(1-K)^{f\left(V;V_{x'}\right)}V^{-\frac{2}{3}}dV}{\frac{3}{2}\left(T_{\mathrm{h}}\left(1+\log\left(\frac{T_{\mathrm{c}}}{T_{\mathrm{h}}(1-K)^{f\left(V_{\min};V_{x'}\right)}}\right)\right)-T_{\mathrm{c}}\right)}. \tag{4.16}$$

Starting at $V_{\max}$, the engine is compressed isothermally while connected to the cold reservoir until it reaches $V_{\min}$. Next the engine is connected to the hot reservoir, allowing the working substance to heat isochorically to T_{h}. The engine, at $V_{\min}$, is then expanded isothermally while connected to the hot reservoir until the engine approaches $V_{x'}$. Finally the engine is irreversibly expanded until the temperature reaches T_{c} and the volume reaches $V_{\max}$, ending at the starting point of the cycle. In this environment we can also redefine the Carnot cycle by replacing the adiabatic processes with irreversible processes. The new Carnot efficiency, $\eta_{\mathrm{C}'}$, using Equations

4.3 and 4.12 and Table 4.1, can be derived as follows;

$$\eta_{\mathrm{C}'} = \frac{Nk_{\mathrm{B}}T_{\mathrm{c}}\log\left(\frac{V_{y'}}{V_{\max}}\right) + Nk_{\mathrm{B}}C_1\int_{V_{y'}}^{V_{\min}}(1-K)^{f\left(V;V_{y'}\right)}V^{-\frac{2}{3}}dV}{Nk_{\mathrm{B}}T_{\mathrm{h}}\log\left(\frac{V_{x'}}{V_{\min}}\right)} + \frac{Nk_{\mathrm{B}}T_{\mathrm{h}}\log\left(\frac{V_{x'}}{V_{\min}}\right) + Nk_{\mathrm{B}}C_2\int_{V_{x'}}^{V_{\max}}(1-K)^{f\left(V;V_{x'}\right)}V^{-\frac{2}{3}}dV}{Nk_{\mathrm{B}}T_{\mathrm{h}}\log\left(\frac{V_{x'}}{V_{\min}}\right)}$$

where

$$V_{y'} = \left(\frac{T_{\mathrm{h}}}{T_{\mathrm{c}}}\right)^{\frac{3}{2}}\frac{V_{\max}}{(1-K)^{\frac{3}{2}f\left(V_{\max};V_{x'}\right)}}, \tag{4.17}$$

is the volume that the irreversible compression process is started at and C_1 can be solved for with T_{c} and $V_{y'}$ using Equation 4.2. Plugging $V_{y'}$ and $V_{x'}$ in and simplifying, we get

$$\eta_{\mathrm{C}'} = \frac{T_{\mathrm{c}}\log\left(\frac{T_{\mathrm{h}}}{T_{\mathrm{c}}(1-K)^{f\left(V_{\max};V_{y'}\right)}}\right) + T_{\mathrm{h}}\log\left(\frac{T_{\mathrm{c}}}{T_{\mathrm{h}}(1-K)^{f\left(V_{\min};V_{x'}\right)}}\right)}{T_{\mathrm{h}}\log\left(\frac{T_{\mathrm{c}}}{T_{\mathrm{h}}(1-K)^{f\left(V_{\min};V_{x'}\right)}}\right)} + \frac{C_1\int_{V_{y'}}^{V_{\min}}(1-K)^{f\left(V;V_{y'}\right)}V^{-\frac{2}{3}}dV + C_2\int_{V_{x'}}^{V_{\max}}(1-K)^{f\left(V;V_{x'}\right)}V^{-\frac{2}{3}}dV}{\frac{3}{2}T_{\mathrm{h}}\log\left(\frac{T_{\mathrm{c}}}{T_{\mathrm{h}}(1-K)^{f\left(V_{\min};V_{x'}\right)}}\right)}.$$

To perform this new Carnot cycle, the same control is used as described in Section 4.1.1, however because the adiabatic processes are replaced with irreversible ones, V_y and V_x are replaced with $V_{y'}$ and $V_{x'}$. Using this we can compare the new Carnot, Heat Loss, and Stirling cycles as we change K. To solve for the required $V_{y'}$ and $V_{x'}$ for each K value we used a binary search with a convergence threshold error of 10^{-9} when checking Equations 4.15 and 4.17. Starting with $V_{\min}^{'} = V_{\min}$ and $V_{\max}^{'} = V_{\max}$ as the minimum and maximum bounds of the search, we first test $V_t = \frac{1}{2}\left(V_{\max}^{'} + V_{\min}^{'}\right)$.

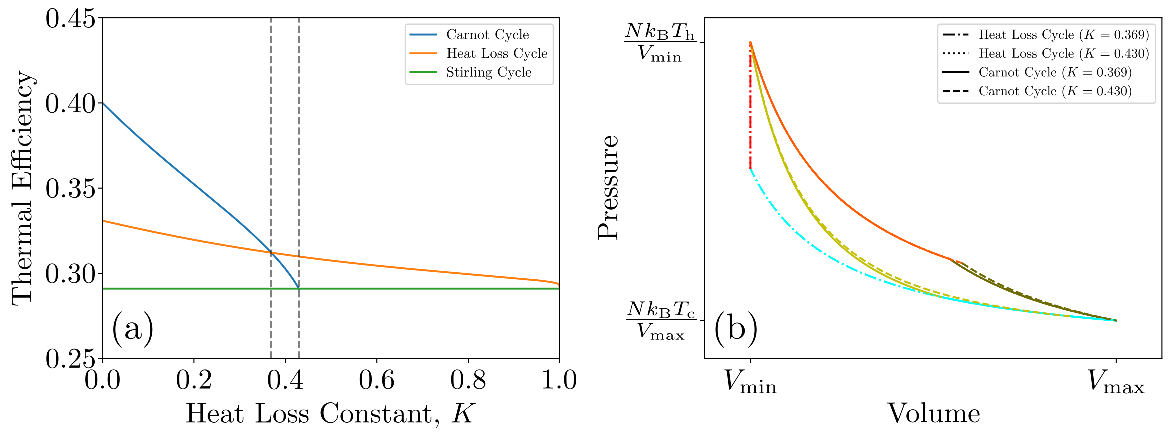


Figure 4.10: (a) The thermal efficiency for the new Carnot, Heat Loss, and Stirling cycles as a function of the heat loss constant, K. The vertical grey line at $K = 0.369$ represents where the new Carnot and Heat Loss cycles have the same thermal efficiency and the vertical grey line at $K = 0.430$ represents where the new Carnot Cycle becomes almost impossible to perform. (b) The new Carnot and Heat Loss cycles for $K = 0.369$ and $K = 0.430$.

If the left side of the equation is less than the right side, we then set $V'_{\text{min}} = V_t$ and recompute V_t with the new bounds. If the left side of the equation is greater than the right side, we then set $V'_{\text{max}} = V_t$ and recompute V_t with the new bounds. Once $V_{y'}$ and $V_{x'}$ are found, the integrals required to calculate the change in work from the irreversible actions are solved numerically. The thermal efficiency for each of these three cycles as a function of K is shown in Figure 4.10(a). When $K = 0$ the irreversible processes are actually adiabatic and therefore $\eta_{\text{C}'} = \eta_{\text{max}} = 0.400$ and $\eta_{\text{H}} = 0.331$. This variant of the new Carnot cycle is identical to the original and can be seen in Figure 4.3(a) and this variant of the Heat Loss cycle is almost identical to Cycle IV in Figure 4.7(b). As we increase K, both the new Carnot and Heat Loss thermal efficiency begin to decrease, however the new Carnot thermal efficiency decreases at a much faster rate. The original Stirling cycle does not use the adiabatic processes and therefore the new Stirling cycle will have the same thermal efficiency no matter what K value is used. At $K = 0.369$ the new Carnot and the Heat Loss cycle now have approximately the same thermal efficiency, $\eta_{\mathrm{C}'} \approx \eta_{\mathrm{H}} = 0.312$. When K is increased beyond this, the Heat Loss cycle becomes more efficient than the new Carnot cycle. At $K = 0.430$, the irreversible compression action has become so inefficient that it must be started almost at V_{max} in order to reach T_{h} in the specified volume range. For this reason once K is increase beyond this, the new Carnot cycle becomes impossible to perform. The new Carnot and Heat Loss cycles for these two K values are shown in Figure 4.10(b). The new Carnot cycle has a thermal efficiency slight below the Stirling cycle with $\eta_{\mathrm{C}'} = 0.290$, while the Heat Loss cycle is still more thermally efficient with $\eta_{\mathrm{H}} = 0.310$. As K approaches 1, the Heat Loss thermal efficiency approaches the Stirling thermal efficiency. Although the equations for the irreversible processes we introduced do not allow $K = 1$, we can still determine what would happen. The irreversible processes at this point become identical to isothermal processes while connected to the cold reservoir. When the irreversible expansion process is performed while the temperature of the engine is at T_{h} and the volume is at V_{max}, it is identical to connecting to the cold reservoir and allowing the temperature to decrease isochorically to T_{c}. Therefore the Heat Loss cycle has the exact same efficiency as the Stirling cycle for this last case.

The choice of ΔV in the *Variable* ΔV *Carnot* environment allowed the agent to reach a much higher efficiency while not being stuck using an extremely small ΔV. While this is beneficial over the fixed ΔV models our other environments used, it still limits the agent by discretizing the volume space. To try to overcome this problem we created a sixth and final heat engine environment with a continuous action space called the *Continuous Carnot* environment. The action space in this environment consists of two inputs. The first action is ΔV in the range of $[0, V_{max} - V_{min}]$ and the second is an integer in the range of [0, 8] that corresponds to which process shown in Figure 4.2 to do with the specified ΔV. As there are no more approximations due

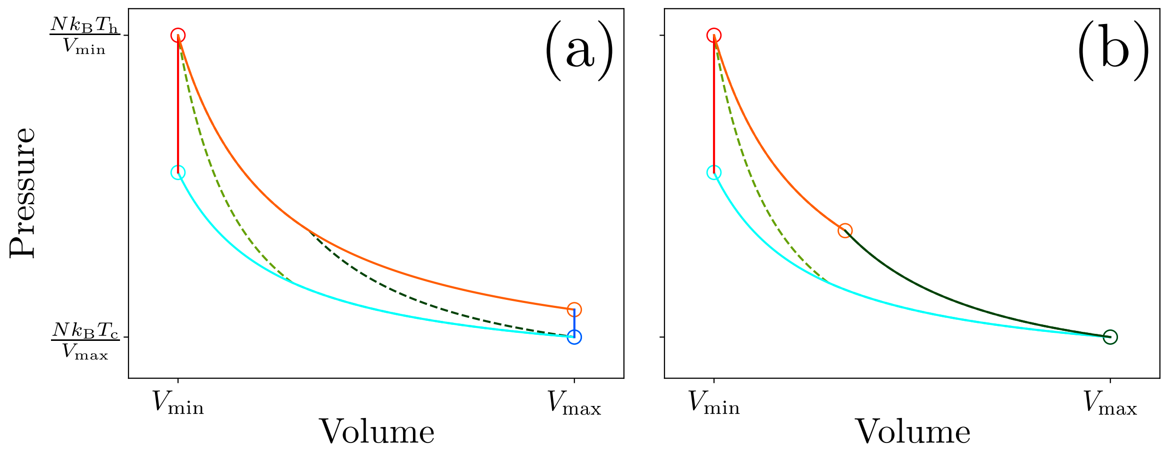


Figure 4.11: (a) Our partially trained and (b) fully trained network based policy agent acting on the *Continuous Carnot* environment with the exact Carnot cycle for reference.

to the volume step size of each, theoretically it is possible for the agent to learn to produce the exact Carnot cycle of any size. One of the noticeable results produced while training an agent in this environment, shown in Figure 4.11, is the Stirling cycle. In this environment the agent performs the exact Stirling cycle in only four steps achieving a maximum thermal efficiency of η_S=0.291, which is the minimum required. While the agent's performance improved with more training, it was not able to produce the exact Carnot cycle or any of the approximations similar to the ones seen in Figure 4.6. However the cycle that the agent ended up producing, shown in Figure 4.11(b) is similar to the Heat Loss cycle seen in Figure 4.10(b) from the *Heat Loss Carnot* Environment. At the end of this agent's training it achieved a maximum thermal efficiency of 0.331, the same as the Heat Loss cycle when $K = 0$.

A problem that we have considered is having an environment which could in fact be solved using a back-propagation based method like DQN. While we did not have time to explore this scenario, it is still useful to discuss. If the agent was provided with only a limited supply of energy to use for Q_{in}, the amount of W produced could be used as the reward. Due to Equation 4.3, if Q_{in} is fixed, then maximizing W will also maximize thermal efficiency as well, still solving the task we initially set out to do. Including both the current total W and Q_{in} in the state vector with T and V should also be enough to then distinguish the same T-V pairs of different cycles.

Chapter 5

Learning to Produce Water

Now that our GA optimized network based policy has been shown learning previously unknown results, it is ready to be tested on a completely new problem, creating water by controlling simulations. However due to the computation cost of these simulations we used the server-based GA method in order to broaden the resources that we can be utilized.

5.1 Density Functional based Tight Binding

In Tight Binding (TB) each atom is represented as a function and the sum of these representing functions is used as the wave-function, $\Psi\left(\vec{r}\right)$. The wave-function can then be represented as a linear sum of basis functions, $\phi\left(\vec{r}\right)$, also known as a linear combination of atomic orbitals. Substituting this into Equation 7.1 yields

$$\hat{H}\sum_i c_i\phi_i\left(\vec{r}-\vec{r}_i\right) = E\sum_i c_i\phi_i\left(\vec{r}-\vec{r}_i\right), \tag{5.1}$$

where c_i is the coefficient for the i^{th} basis function. Multiplying from the left by the complex conjugate of $\Psi(\vec{r})$ and integrating over all space gives

$$\begin{gathered}\int \sum_{i'} \phi^*_{i'}(\vec{r}-\vec{r}_{i'})\,\hat{H} \sum_{i} c_i \phi_i(\vec{r}-\vec{r}_i)\,d\vec{r} = \int \sum_{i'} \phi^*_{i'}(\vec{r}-\vec{r}_{i'})\,E \sum_{i} c_i \phi_i(\vec{r}-\vec{r}_i) \\ \sum_{i'} \sum_{i} c_i \langle \phi_i(\vec{r}-\vec{r}_{i'})\,|\hat{H}|\phi_i(\vec{r}-\vec{r}_i)\,\rangle = E \sum_{i'} \sum_{i} c_i \langle \phi_i(\vec{r}-\vec{r}_{i'})\,|\phi_i(\vec{r}-\vec{r}_i)\,\rangle.\end{gathered} \tag{5.2}$$

If ϕ_i is an orthonormal function then $\langle \phi_i(\vec{r}-\vec{r}_{i'})\,|\phi_i(\vec{r}-\vec{r}_i)\,\rangle = \delta_{i'}\delta_i$, otherwise it results in a S matrix. In Density Functional based Tight Binding (DFTB) $\hat{H}$, S, and the repulsive energies are computed using Density Functional Theory (DFT) and stored in Slater-Koster files.

5.2 Water Environment

To model a simple chemical reaction, we created an environment that an agent can interact with. The basis of this environment is a DFTB simulation containing two H_2 molecules and one O_2 molecule where the goal is to make H_2O, and therefore will be referred to as the *Water* environment. The initial set of atomic positions is randomly selected from a collection of equilibrium states at 300 K with uniform probability. The system uses a Nose Hoover thermostat with the range of [100 K, 20000 K], initially at 300 K and the Velocity Verlet algorithm to update the positions and velocities at each time step of 0.2 fs, where the interatomic forces are determined with DFTB. This simulation is run inside a 20 Å^3 box with PBCs. Similar to the *Pendulum* environment, the *Water* environment has a continuous action space in the range of [-1000, 1000]. This number corresponds to the change in the thermostat temperature in Kelvin. At each environment step the thermostat is changed at a constant rate for 500 time steps. After the DFTB simulation has evolved for 500 time steps, the

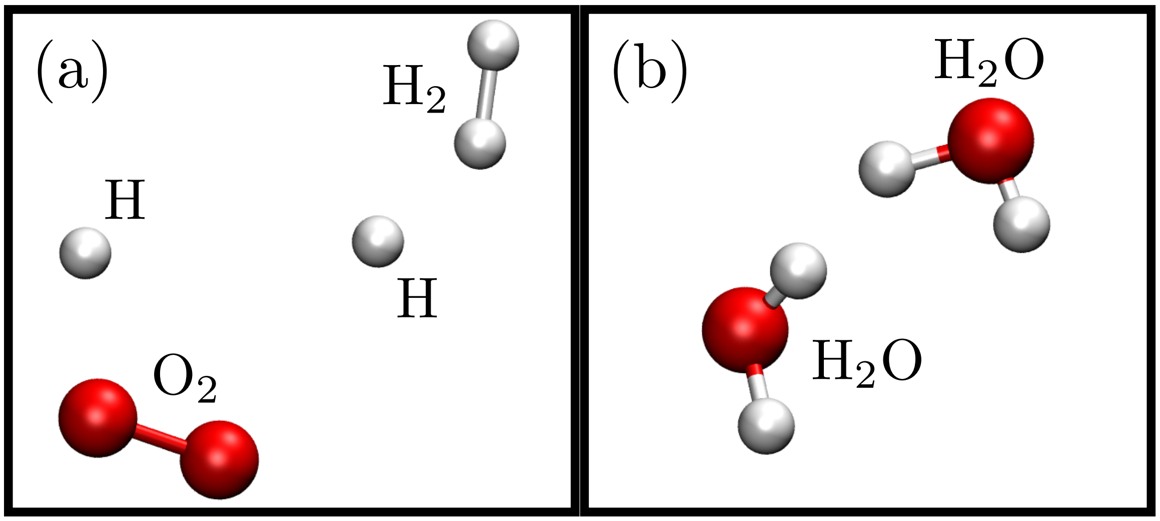


Figure 5.1: (a) A graphical example of what the *Water* environment could look like during a given state with molecular fractions for H, O_2 and H_2 being 0.5, 0.25, and 0.25 respectively with all other molecular fractions being 0. (b) A graphical example of what the *Water* environment could look like during a state that would give the agent positive reward with molecular fractions for H_2O being 1.0 and all other molecular fractions being 0.

environment returns the state vector and reward. This 500th time step will be referred to as a key frame. The state of this environment is a vector containing the observed temperature and pressure with the weighted molecular fractions by time of O, H, O_2, H_2, OH, H_2O, HO_2, H_2O_2, and other between each key frame, where other is the molecular fraction of everything not specifically listed. If a species is formed halfway between key frames, then the weighted molecular fraction is the molecular fraction multiplied by 0.5 and similarly if a species dissociates halfway between key frames. A graphical example of a state of the *Water* environment can be seen in Figure 5.1. The game is played until 100 steps has been reached. At each step the environment returns the weighted molecular fraction of H_2O in the range of [0, 1] for the reward. A reward of 0 means that no H_2O molecules existed between key frames for any amount of time and a reward of 1 means that H_2O was the only molecule that existed between key frames.

5.3 Server-Based Genetic Algorithm

One of the great benefits of GAs is that they are extremely parallelizable. As the evaluation of one policy does not depend on the evaluation of another, this parallelization can be done with multi-processing. A processes is a task that can be run on a Central Processing Unit (CPU), and multi-processing runs several processes co-currently on multiple CPUs, with each process being run independently of the others. This is opposed to multi-threading which runs one process on several CPUs and in this case the CPUs need to communicate which slows down computation but is a necessary cost sometimes. Currently the standard implementations of a GA are limited to the number of processes that can be run on one machine. A solution to this problem is to use a method we call server-based GA. This method involves three key components: the master, the workers, and the database that consists of an unfinished, in-progress, and completed table. The master initializes the population of N_P policy seeds, each with an ID, and adds them to the unfinished table. A policy seed is a number used as the seed in the random number generator to generate the parameters, θ, for the policy. If the environment we are evaluating the policies on has stochastic elements we need to evaluate each policy multiple times. In that case the policy is added the unfinished table multiple times so that the scores can be averaged over. The workers then check the unfinished table for any policies that have not yet been evaluated. When a worker finds an unfinished policy, it moves the policy to the in-progress table, marks it with a time stamp, constructs the parameters using the policy seed, and then starts evaluating it on the environment. When the worker finishes evaluating a policy, it moves the policy to the completed table, assigns it a score, and then waits for another policy to appear in the unfinished table. A maximum process time, $t_{\max}$, is used in the case that a worker has failed to complete the evaluation of a policy. Any policy that has

1st Policy ID	0	1	2	...	N_P
2nd Policy ID	0	0	0	...	n_{avg}
Policy Seeds	[9108, 6251]	[1937]	[4374, 2784]	...	[6736, 0023, 5901]
Time (s)	1561323273	1561323284	1561323286	...	1561323426
Score	0.98	0.23	0.01	..	0.75

Table 5.1: An example of the unfinished, in-progress, and completed tables for the database component used by the server-based GA represented as one table.

a time stamp from more than $t_{\max}$ seconds ago is removed from the in-progress table and placed back in the unfinished table by the master. During this the master also checks the completed table to see if all policies have been evaluated yet. Once the completed table is filled, the master averages the n_{avg} scores for each policy with the same ID and then sorts them. The master then continues the normal GA process by keeping the best N_e policies and then creating new ones through mutation. However when using policy seeds, mutation is performed by simply adding another number to the policy seed, turning it into a list of seeds. Using a list of policy seeds we can then define a policy as

$$\pi = \theta^{(X_0)} + \epsilon_m \sum_{i=1}^{n} \theta^{(X_i)}, \tag{5.3}$$

where $\theta^{(X_i)}$ is the set of parameters generated using the random number generator seed X_i, ϵ_m is the mutation rate from Section 3.2, and n is the number of seeds for that policy. For simplicity, we are only considering the mutation operation for this method and not the crossover operation. Once the population is filled back up to N_P policies, the master adds them to the unfinished table and the process is repeated. A simplified example of the three database tables is shown in Table 5.1 and a visual example of this algorithm is shown in Figure 5.2.

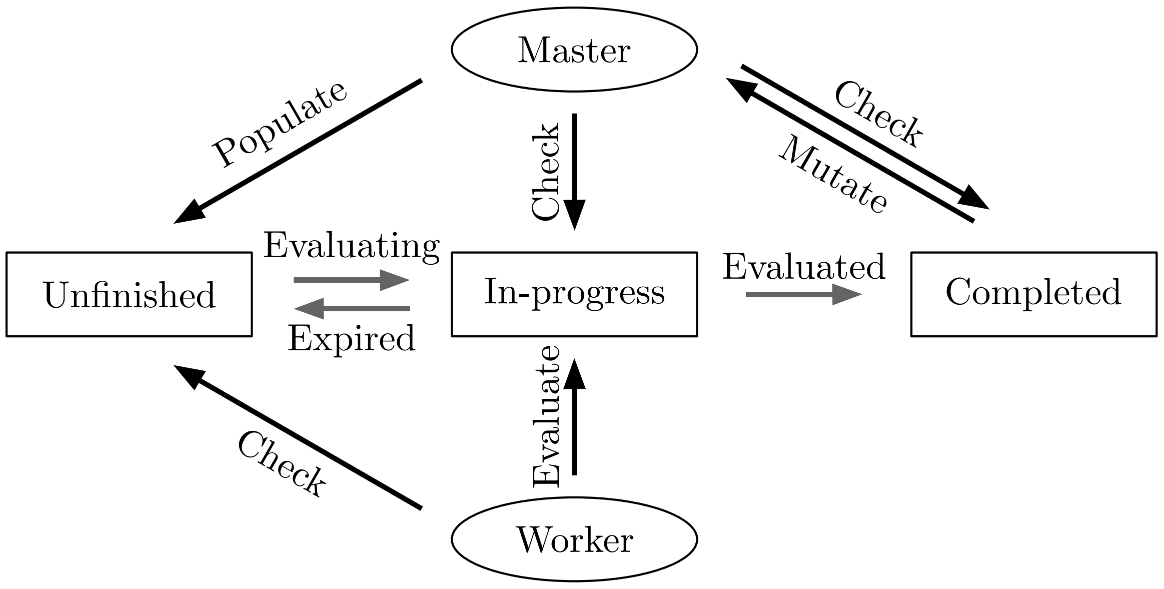


Figure 5.2: A visual example of the server-based GA showing the flow of policies between database tables. The black arrows represent a process being done by the master or worker and the grey arrows represent policies moving between tables.

5.4 Results

Unlike every other previous environment used in this study, except the *Heat Loss Carnot*, the optimal solution to maximize reward is not known. The closest to a solution that is known is that the system must be heated to a temperature at which both the hydrogen-hydrogen and oxygen-oxygen bonds can break, and then cooled down at a rate which hydrogen-oxygen bonds can form, producing water. To explore the unknown solution space a series of manual policies were tested on the *Water* environment. Each policy has the same form which is to heat the system to some higher temperature over n_{heat} steps, denoted as T_{max} as it will be the maximum temperature the system reaches with that policy, hold the thermostat temperature constant for n_{hold} steps, cool the system back down to the initial temperature of 300 K at a constant rate over n_{cool}, and then hold the thermostat temperature constant at 300 K. A general example of the thermostat temperature over an episode for one of these

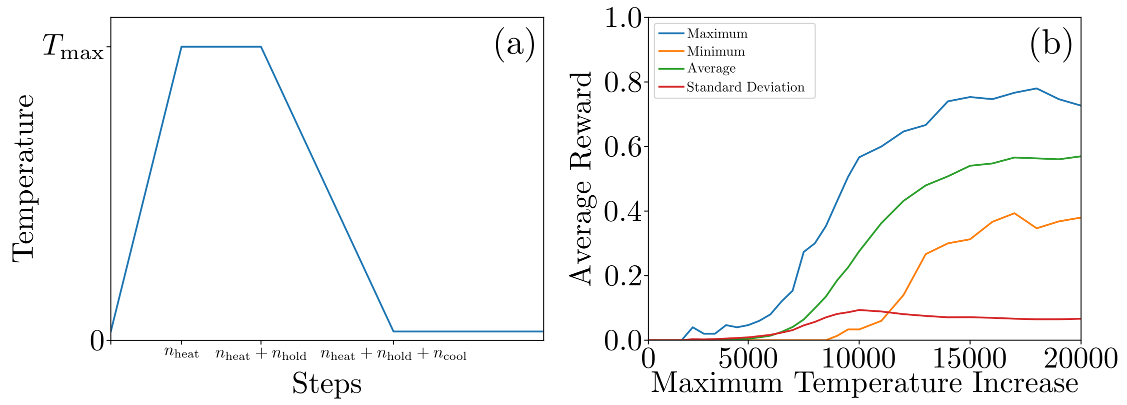


Figure 5.3: (a) The thermostat temperature over the course of an episode as a general policy in the manual testing set acts on the *Water* environment. (b) The maximum, minimum, average, and standard deviation of the average final rewards of each policy for each $T_{\max}$ tested.

policies is shown in Figure 5.3(a). The maximum thermostat temperature was varied from 500 K to 20000 K above the initial temperature, n_{hold} was varied from 1 to 20 steps, and n_{cool} was varied 0 to 19 steps above the minimum number of steps required to decrease the thermostat temperature from $T_{\max}$ to 300 K. Each combination of $T_{\max}$, n_{hold}, and n_{cool} making up a manual test policy was evaluated over 50 episodes to get both an average total reward and average final reward. The total episodic reward represents how long water existed during the episode and the final episodic reward represents whether or not water existed at the end of the episode. The best performing policy would have both a high average and final reward with low variation. The results of these tests for a $T_{\max}$ of 20000 K above the initial thermostat temperature are shown in Figure 5.4. With $T_{\max}$ at 500 K above the initial thermostat temperature no water is produced in any episode for any policy. This shows that there is not enough energy at 800 K to break both the hydrogen-hydrogen or the oxygen-oxygen bonds. With $T_{\max}$ at 2500 K above the initial thermostat temperature there are some episodes where water is made. However it is only a few episodes where this occurs meaning that although there is enough energy at 2800 K to break

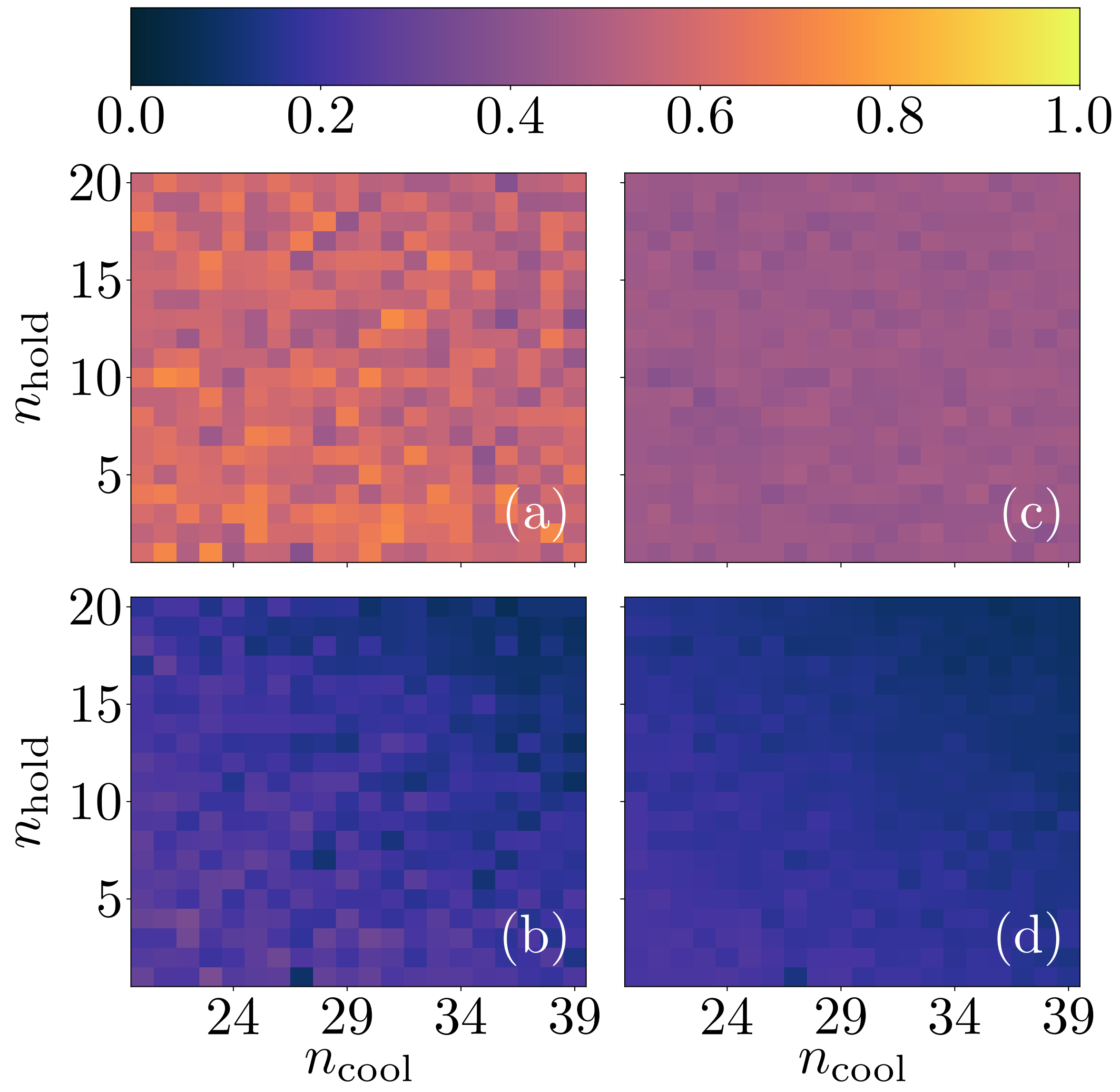


Figure 5.4: (a) The average final rewards with (b) the standard deviation and (c) the average rewards with (d) the standard deviation over 50 episodes of all manual test policies with T_{max} 20000 K above the initial thermostat temperature.

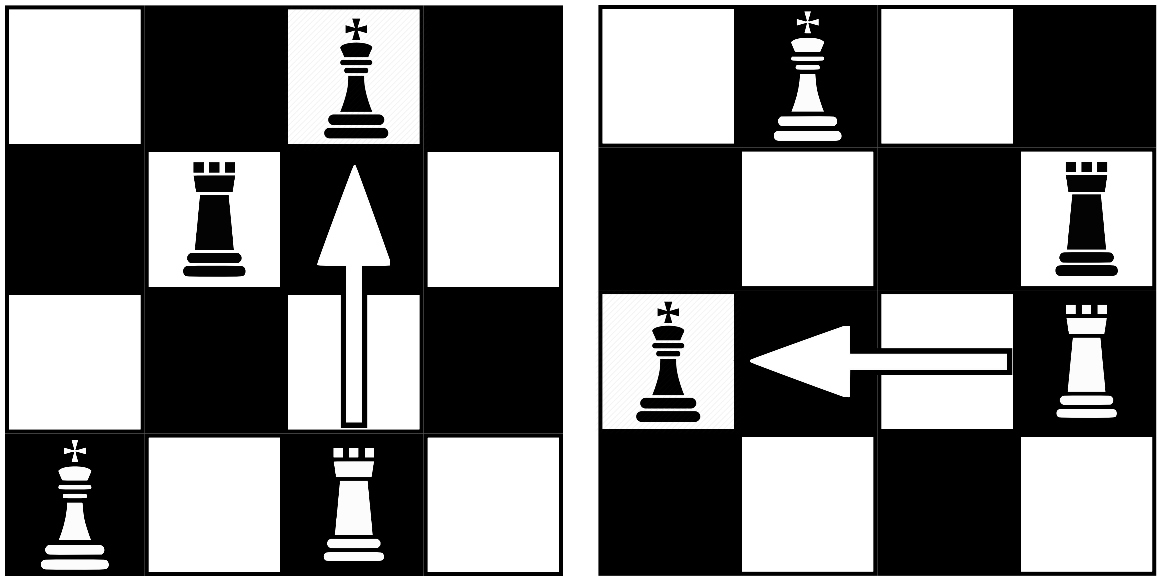


Figure 5.5: Two examples of a state-action pair in a game of chess on a quarter sized board that can be assigned a positive reward. The player is moving any white piece, in this case a rook, onto the same square as the opponent's king, thereby ending the game and giving a positive reward for the player.

the necessary bonds, there is not enough energy to break them consistently. Increasing T_{max} to 10000 K above the initial thermostat temperature water is produced by most policies in several episodes. The most apparent trend follows increasing T_{max}, regardless of what n_{cool} and n_{hold} are, and can be seen in Figure 5.3(b). With this policy structure shown to work, it should possible for a RL algorithm to learn how to produce water. However the GA optimization was not able to reliably produce water. This is likely due to the extremely high randomness of the *Water* environment. Another reason this environment is unlike all the others tested previously is there is a random element at every step, not just random initial conditions. This means that there is a high chance that if a policy happens to produce water, it will not be able to repeat this process consistently. Although there was no promising results using the GA for optimizing the neural network based policy, that does not mean this problem is impossible to learn. The GA seemed like the best choice for the heat engine environments as it solves the credit assignment problem that arises when trying to assign a state-action pair to the thermal efficiency. Even problems like chess, that are considered delayed reward problems where it is not clear or obvious what a good move is, can still have the reward of winning assigned to a state-action pair. An example of such pair is shown in Figure 5.5. For this reason, standard back propagation based RL methods can learn to win the game of chess. There is no one state-action pair in the heat engine environments in which thermal efficiency can be assigned to, but rather a set of state-action pairs. For this reason, our maximizing thermal efficiency problem is best handled with a GA optimization method. However like chess, and many other delayed reward problems like it, the rewards in the *Water* environment can be assigned to state-action pairs. These being the state-action pair at which water is produced. This means that perhaps more traditional back propagation based RL methods could learn to produce water in the *Water* environment. While no meaningful results were obtained from attempting to train a RL agent on the *Water* environment, an important lesson was learned. Having an agent learn to control an environment like this where it cannot directly control what is happening would be significant, however there are too many components in the *Water* environment to know what does and does not make it possible to learn.

Chapter 6

Conclusions

While we discussed traditional RL techniques, and more modern ones, the focus of the work shown here has been on GAs and their use for the optimization of neural network-based policies. While we did not use methods like DQN or A2C to get any meaningful results, that does not discredit the importance of them for other problems. Just as these methods optimize policies in different ways, they succeed at different problems than each other and no single method is better than all others in RL. Such et. al. [44] showed this quite well by comparing GAs with these other methods, inspiring us to pursue GAs as a viable method to solve our series of heat engine environments. The GA worked well when applied to the heat engine environments but failed on the *Water* environment. While there are many reasons for this, the main one attributed the the GA is that individual feedback was not used and therefore specific learning at certain states could not take place. Another issue could be the ANN architecture used, which if modified in a way that increases the parameter space, would increase the search space for the GA. Once the search space becomes too large, using a GA would become unreasonable.

We have demonstrated the ability of a neural network-based policy optimized using a GA to reproduce classic control problems in thermodynamics, mainly heat engine thermodynamic cycles. They are particularly challenging as RL problems because they do not provide meaningful feedback, but rather depend on a final score. RL finds a policy which steers a system toward an optimal condition, doing so without any guidance from the user. It explores parameter space on its own, and finds the best sequence of decisions to achieve a goal. Here a neural network-based policy trained in a heat engine environment without any prior knowledge of physics can reproduce the Carnot cycle with over 99% accuracy, nearly achieving $\eta_{\max}$. The Stirling and Otto cycles were similarly obtained with 100% accuracy when adiabatic or isothermal processes are removed from the action space. The same algorithm naturally learned irreversible processes, again learning an optimal control scheme given the restrictions of the environment. RL offers the ability to both seek data and learn the optimal, and likely dynamical, control scheme for physical systems. The heat engine environments contained the temperature and volume of the system as the state, but these variables alone were not enough to know everything about the environment but the agent did have control over the transition between states. This is why they can be considered POMDPs and required more than the standard RL methods for solving. While not studied here, another alternate method of solving this problem would be to use a policy or agent with the concept of memory so it could remember previous states and therefore know the full trajectory of the thermodynamic cycle being performed. Constructing the problem how we did made it a more difficult problem because of the partially hidden states of the POMDP but turning it into a MDP could take away from what we are trying to show here, depending on how it is done. One of the most impressive breakthroughs in modern RL is the number and magnitude of successes that these agents achieve without any human precursor knowledge. AlphaGo was able to beat *go* masters, an achievement thought of as impossible beforehand, but it was trained using examples of humans playing *go*. AlphaGo Zero however was trained by playing a copy of itself and by doing so AlphaGo Zero learned from its own mistakes. Through this process AlphaGo Zero was able to even out-play the best versions of AlphaGo, all without any prior knowledge or human examples of the game. Had we designed the heat engine environments as a MDP then we would have needed to include work, energy, pressure, thermal reservoir connections, and more in the state. On top of this we would have to manually shape the reward function to build in components of the problem such as what a cycle is, when the agent should receive the reward to properly define the problem, and other human knowledge aspects. We made a harder problem for the agent to solve by not teaching it thermodynamics or how to operate a heat engine beforehand and because of this the results achieved from this are more meaningful.

Although we were not able to further demonstrate the ability of a neural network-based policy optimized using a GA by producing water from hydrogen and oxygen gas molecules, this example helps emphasize that not one RL algorithm is better than all others. A GA was extremely well suited for learning to reproduce known thermodynamic cycles by maximizing thermal efficiency in the heat engine environments as it deals with the credit assignment problem wherever it arises. However it seems that when the environment involves a large number of random elements the GA had difficulty learning the objective. While we were not yet able to test other RL algorithms on the *Water* environment in this study, we were able to partially explore the policy space, learning more about the environment. An important distinction between the heat engine environments and the *Water* environment is that they could be considered POMDPs where this one is more accurately classified as a HMM. While in a true HMM the agent has no control over state transition, it is not very far off here. The agent had control over the thermostat temperature that changed the probability of certain state transitions to occur, however it is still impossible for the agent to direct this transition any more than this like it did with the heat engine environments.

Future work that could continue from here includes, but is not limited to, studying how the information available from the environment impacts the learning process for the agent, how different policies structures modify the solvability of the problem, and how the knowledge learned by an agent on one problem can be transferred to help or speed up the learning process for another. With the abundance of RL methods yet to test, it is not safe to assume the *Water* environment is impossible to learn as each method succeeds at different tasks. With that being said, another aspect of trying to solve the problem could be to modify how it is presented. We originally chose to use the averaged measured temperature of the system as part of the state because we thought it would be a more accurate representation than the thermostat temperature. However even when averaging the temperature over many time steps, there are still large fluctuations due to the small sample size of molecules. To conclude, we hope that the work shown here helps shed light on the uses of RL and its possibilities for scientific problems.

Chapter 7

Appendix

This section contains work that was done during this degree but does not necessarily fit into the overall story of the main body of the thesis itself. Along with extra work, there is sample code provided for various components of the projects discussed in previous sections along with a description of the contributions that were made towards each task by the primary author.

7.1 Updating Quantum Machine 9

The Quantum Machine 9 (QM9) database is a database consisting of 133,885 stable small organic molecules. Each molecule is made of up to 9 non-hydrogen atoms limited to carbon, oxygen, nitrogen, and fluorine. QM9 contains the geometries relaxed at the Becke, Three-parameter, Lee-Yang-Parr (B3LYP) 6-311+G(3df,2p) level labeled with energies calculated at the same level. 6,095 constitutional isomers of $C_7H_{10}O_2$ are included in QM9 and these have energies calculated at the Second Order Møller-Plesset Perturbation Theory (MP2) level. As machine learning methods can only be as accurate as the data they were trained on, we set out to increase the level of theory the QM9 molecule energies were calculated at by calculating the energy for rest of the molecules at the MP2 level.

7.1.1 Density Functional Theory

The energy of an atom or molecule is calculated using the Schrödinger equation

$$\hat{H}\Psi = \left[\sum_{i}^{N}\left(-\frac{\hbar}{2m_e}\nabla_i^2\right) - \sum_{i<k}^{N}\frac{1}{\vec{r}_{ik}} + \sum_{i<j}^{N}\frac{1}{\vec{r}_{ij}} + \sum_{i<k}^{M}\frac{1}{R_{ik}}\right]\Psi = E\Psi \tag{7.1}$$

where $\hat{H}$ is the Hamiltonian operator, Ψ is the wave-function for a system with N electrons, M is the number of nuclei in the systems, $\hbar$ is the reduced Planck constant, m_e is the mass of an electron, ∇ is the Laplacian operator, $\vec{r}_{ik}$ is the distance between the i^{th} electron and the k^{th} nucleus, $\vec{r}_{ij}$ is the distance between the i^{th} and j^{th} electrons, R_{ik} is the potential energy from the i^{th} and and k^{th} nucleus interacting, and E is the total energy of the system. As the number of electrons in the system increases, the Schrödinger equation becomes more complicated. The $\vec{r}_{ij}$ term makes the equation difficult to solve because it is not separable into components consisting of only electron i or j. DFT does not solve the Schrödinger equation with an N-electron wave-function. DFT uses the electron-density as it is easier to solve a 3-D function than a $3N$-D wave-function. DFT calculates the total energy as a function of electron density with

$$E_{\text{DFT}} \equiv E\left[\rho\right] = -\sum_{i}\int \psi_i \nabla^2 \psi_i d\boldsymbol{r} + \int \rho \nu_{\text{ext}} d\boldsymbol{r} + J\left[\rho\right] + E_{\text{xc}}\left[\rho\right], \tag{7.2}$$

where ρ is the electron density, ψ is the minimum energy orbital, ν_{ext} is the external potential, $J\left[\rho\right]$ is the Coulomb repulsion between the electrons, and $E_{\text{xc}}\left[\rho\right]$ is the exchange-correlation functional. The exchange-correlation functional is the part of the DFT energy equation that is not known exactly and varies depending on which type of calculations are being performed. Since DFT uses electron density to calculate energy, the electron density is needed, however the electron density is also not known. Self-consistent field calculations use a initial electron density guess to construct an effective potential to find minimum energy orbitals. These minimum energy orbitals are used to create a more accurate electron density function through

$$\rho = \sum_{i}^{N} |\psi_i|^2. \tag{7.3}$$

The newly calculated electron density is used as the initial guess and this process is repeated until a convergence of the energy is reached.

To account for the changes to the electron density with respect to position, $\boldsymbol{r}$, the exchange correlation functional is expanded in terms of electron density gradient, opposed to having a constant electron density throughout the system. The Local Spin Density Approximation (LSDA) functional with the form

$$E_{\text{xc}}^{\text{LSDA}} \left[n^{\uparrow}, n^{\downarrow}\right] = \int n\left(\boldsymbol{r}\right) \epsilon_{\text{xc}} \left(n^{\uparrow}\left(\boldsymbol{r}\right), n^{\downarrow}\left(\boldsymbol{r}\right)\right) d\boldsymbol{r}, \tag{7.4}$$

where ϵ_{xc} is the exchange correlation energy density, $n^{\uparrow}$ is a spin up electron, and $n^{\downarrow}$ is a spin down electron, is expanded to a Generalized Gradient Approximation (GGA) functional with the form

$$E_{\text{xc}}^{\text{GGA}} \left[n^{\uparrow}, n^{\downarrow}\right] = \int n\left(\boldsymbol{r}\right) \epsilon_{\text{xc}} \left(n^{\uparrow}\left(\boldsymbol{r}\right), n^{\downarrow}\left(\boldsymbol{r}\right), \nabla n^{\uparrow}\left(\boldsymbol{r}\right), \nabla n^{\downarrow}\left(\boldsymbol{r}\right)\right) d\boldsymbol{r}, \tag{7.5}$$

which now includes the gradient of electron density. Hybrid exchange correlation functionals linearly combine these LSDA and GGA functionals, and sometimes others, with the Hartree-Fock exact exchange functional

$$E_{\mathrm{x}}^{\mathrm{HF}} = -\frac{1}{2}\sum_{i,j}\int\int \psi_i^*\left(\boldsymbol{r_1}\right)\psi_j^*\left(\boldsymbol{r_2}\right)\frac{1}{r_{12}}\psi_i^*\left(\boldsymbol{r_1}\right)\psi_j^*\left(\boldsymbol{r_2}\right)d\boldsymbol{r_1}d\boldsymbol{r_2} \tag{7.6}$$

with coefficients experimental or other accurately calculated data. A very popular example of these hybrid functionals is the B3LYP exchange-correlation functional

$$E_{\mathrm{xc}}^{\mathrm{B3LYP}} = E_{\mathrm{x}}^{\mathrm{LDA}} + a_0\left(E_{\mathrm{x}}^{\mathrm{HF}} - E_{\mathrm{x}}^{\mathrm{LDA}}\right) + a_{\mathrm{x}}\left(E_{\mathrm{x}}^{\mathrm{GGA}} - E_{\mathrm{x}}^{\mathrm{LDA}}\right) + E_{\mathrm{c}}^{\mathrm{LDA}} + a_{\mathrm{c}}\left(E_{\mathrm{c}}^{\mathrm{GGA}} - E_{\mathrm{c}}^{\mathrm{LDA}}\right), \tag{7.7}$$

where

$$E_{\mathrm{xc}} = E_{\mathrm{x}} + E_{\mathrm{c}}, \tag{7.8}$$

$a_0 = 0.20$, $a_{\mathrm{x}} = 0.72$, $a_{\mathrm{c}} = 0.81$, $E_{\mathrm{x}}^{\mathrm{LDA}}$ is defined as

$$E_{\mathrm{x}}^{\mathrm{LDA}} = -\frac{3}{4}\left(\frac{3}{\pi}\right)^{\frac{1}{3}}\int \rho\left(\boldsymbol{r}\right)^{\frac{4}{3}} d\boldsymbol{r}, \tag{7.9}$$

$E_{\mathrm{c}}^{\mathrm{LDA}}$ is the Vosko, Wilk, Nusair correlation functional, $E_{\mathrm{x}}^{\mathrm{GGA}}$ is the Becke 88 exchange functional, and $E_{\mathrm{c}}^{\mathrm{GGA}}$ is the Lee, Yang, and Parr correlation functional. While DFT reduces the computational cost by replacing all individual electrons with a single electron density, it does so at a loss in accuracy. For this reason more computational expensive methods still exist.

7.1.2 Møller-Plesset Perturbation Theory

In Reyleigh-Scrödinger Perturbation Theory (RS) the unperturbed Hamiltonian, $\hat{H}_0$, is taken and some perturbation, $\hat{V}$, is added to it so the Hamiltonian becomes

$$\hat{H} = \hat{H}_0 + \lambda\hat{V}, \tag{7.10}$$

with $0 < \lambda << 1$. The wave-function, Ψ, is then perturbed using a power series by

$$\Psi = \sum_{i=0}^{\infty} \lambda^i \Psi_i. \tag{7.11}$$

With these perturbations, Equation 7.1 becomes

$$\left(\hat{H}_0 + \lambda \hat{V}\right)\left(\sum_{i=0}^{\infty} \lambda^i \Psi_i\right) = \left(\sum_{i=0}^{\infty} \lambda^i E_i\right)\left(\sum_{i=0}^{\infty} \lambda^i \Psi_i\right). \tag{7.12}$$

When the sums in Equation 7.12 are expanded, the i^{th} order terms are $O\left(\lambda^i\right)$. In Møller-Plesset Perturbation Theory (MP), it is assumed that the zeroth energy, E_0, is the Hartree-Fock energy, meaning that the unperturbed Hamiltonian, $\hat{H}_0$ becomes the shifted Fock operator

$$\hat{H}_0 = \hat{F} + \langle \Phi_0 | \hat{H}_0 - \hat{F} | \Phi_0 \rangle, \tag{7.13}$$

where $\hat{F}$ is the Fock operator and Φ_0 is the normalized Slater determinant. Substituting Equation 7.13 into Equation 7.12 gives the MP Schrödinger equation

$$\left(\hat{F} + \langle \Phi_0 | \hat{H}_0 - \hat{F} | \Phi_0 \rangle + \lambda \hat{V}\right)\left(\sum_{i=0}^{\infty} \lambda^i \Psi_i\right) = \left(\sum_{i=0}^{\infty} \lambda^i E_i\right)\left(\sum_{i=0}^{\infty} \lambda^i \Psi_i\right). \tag{7.14}$$

7.1.3 Results

One of the biggest limitations of machine learning, especially supervised learning, is that your approximation of the true function can only be as accurate as your training set. Therefore if all of your data has an average error of 10% because your model is not exact, you cannot hope for machine learning methods to achieve better results than that. This is also true with using machine learning to determine certain properties of molecules. When using a specific computational method to calculate these initial

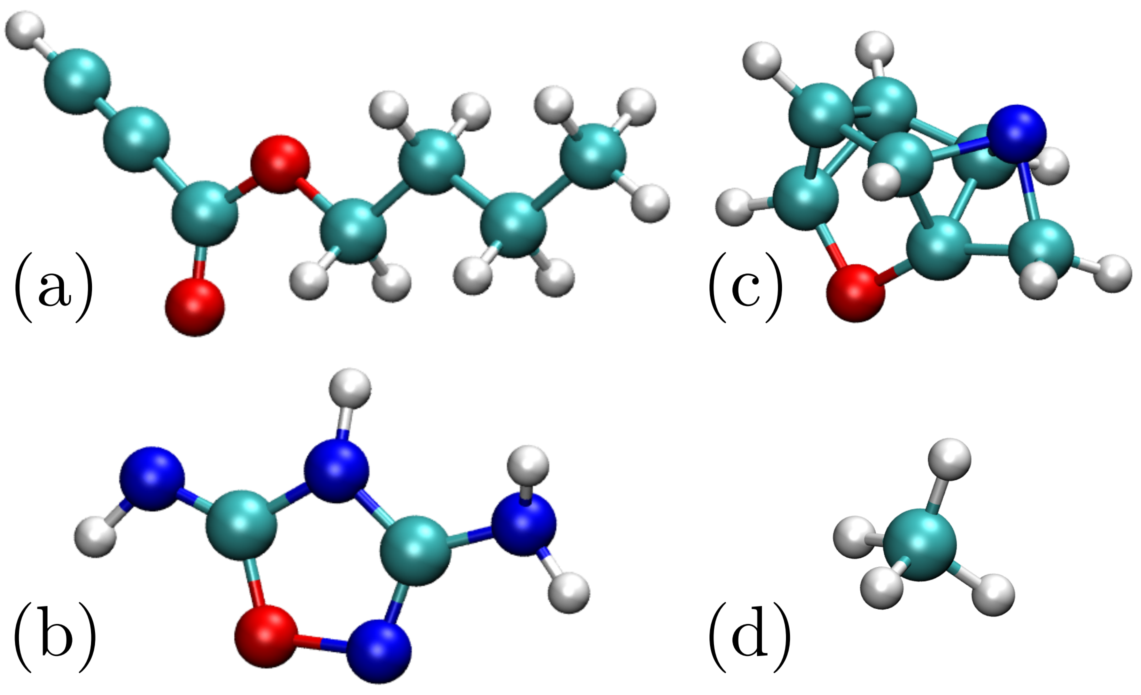


Figure 7.1: Examples of molecules in the QM9 dataset. (a) One of the $C_7H_{10}O_2$ constitutional isomers, (b) one of the more complex molecules, C_7H_7NO, (c) one of the more planar molecules, $C_2H_5N_4O$, and (d) the simplest molecule in QM9, CH_4.

properties to produce the dataset, even the best machine learning method can only reach the accuracy of that computational method. For the case of the QM9 dataset this method is B3LYP. Some examples of what the molecules in the QM9 database look like are shown in Figure 7.1. While all the molecules in QM9 have 9 or less non-hydrogen atoms limited to carbon, oxygen, nitrogen, and fluorine, there is still a range of different size and shaped molecules. This diversity is what makes QM9 a good choice for machine learning as another limitation is not just the accuracy but also the range of the training data. Now that we have energies calculated at the MP2 level for QM9, they can be used for supervised learning. Comparing the B3LYP energies that were already included in QM9 with the new MP2 energies, we get a non-trivial change in the the distribution shown in Figure 7.2(a). While some bins have negligible differences between the B3LYP and MP2 distributions, there are some

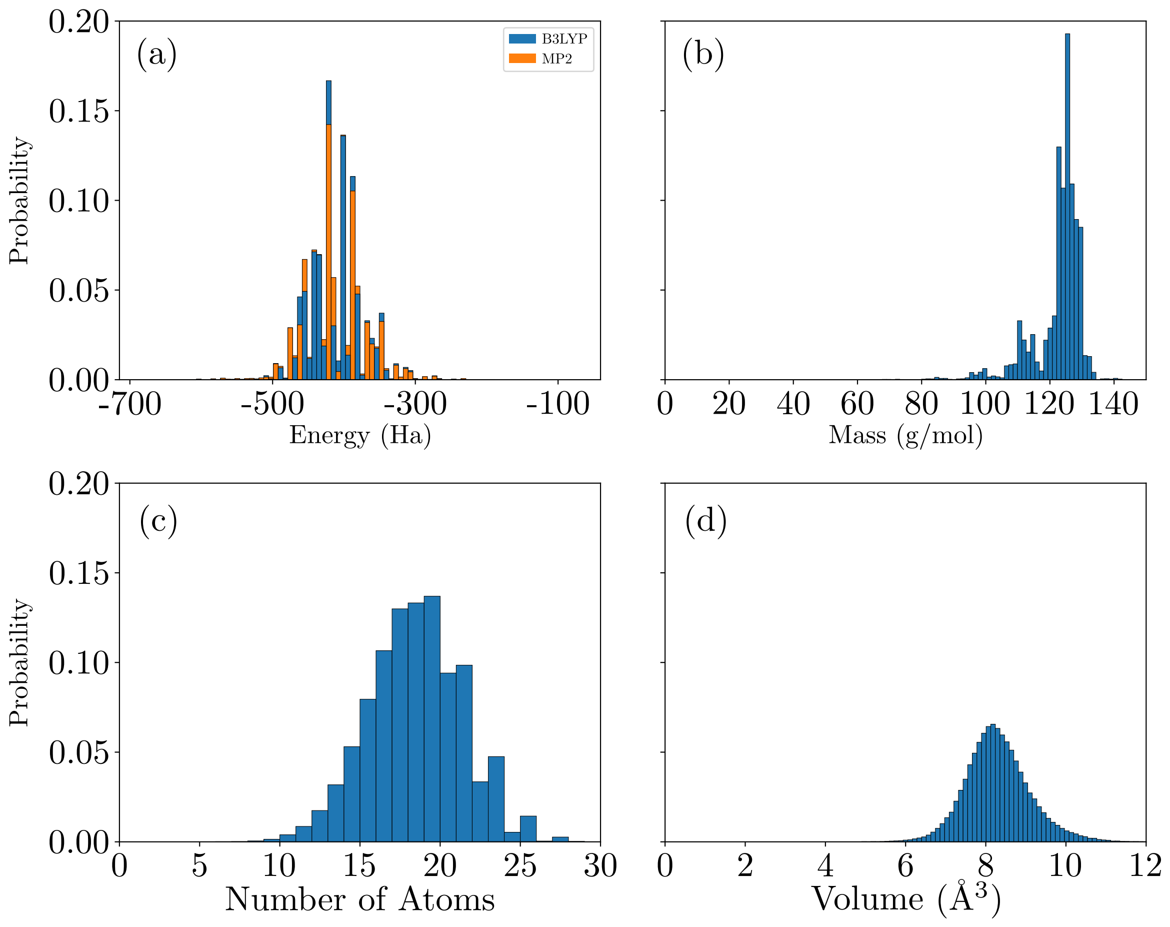


Figure 7.2: (a) The histograms of energies for the molecules in the QM9 database calculated at the B3LYP and MP2 levels. Bins that appear to be entirely blue represent a very small change in the distribution with the orange bar being slightly higher and the same is true for the opposite case. (b) The histogram of molecular masses, (c) number of atoms, and (d) volume for the molecules in the QM9 database.

changes of up 50%, ignoring the extremely small bins. While the molecular mass of each molecule does not change with the method being used it is still important to know the distribution that is shown in Figure 7.2(b). Where the energy distributions had a single node that was not very skewed, the mass distribution is noticeably left skewed. Despite the mass distributions skew, the number of atoms distributions shown in Figure 7.2(c) is less noticeably skewed. This is likely due the majority of the atoms being hydrogen and therefore not largely contributing to the mass of the molecule. Lastly, the distributions of volumes shown in Figure 7.2(d) is the most normal distribution out of the five. The volume was calculated as

$$V = \sqrt{(x_{\max} - x_{\min})^2 + (y_{\max} - y_{\min})^2 + (z_{\max} - z_{\min})^2}, \tag{7.15}$$

where $x_{\min}$, $y_{\min}$, and $z_{\min}$ are the minimum xyz coordinates of the molecule and $x_{\max}$, $y_{\max}$, and $z_{\max}$ are the maximum xyz coordinates of the molecule. While using this method for volume calculation does not give a completely accurate volume for the molecule itself, it does represent how much space it spans in each of the three dimensions. This is important because if the molecule is represented as an image, the image will need enough voxels, in either quantity or size, to contain the molecule. These distributions are important for supervised learning methods because they tend to perform better on the areas of the distribution that has more data and poorly on areas that have low amounts of data.

7.2 Sample Code

All neural network-based policy code was written by Chris Beeler which is provided below:

```
class Policy():

    def __init__(self, shape, hidden_units, num_actions, game):

        self.shape = shape

        self.game = game

        self.hidden_units = hidden_units

        self.num_actions = num_actions

        self.win = 0

        self.W = []

        self.B = []


    def gen_random(self, seed=None):

        np.random.seed(seed)

        self.W = []

        self.B = []

        w, b = layer(self.shape, self.hidden_units[0])

        self.W.append(w)

        self.B.append(b)

        for i in range(1, len(self.hidden_units)):

            w, b = layer(self.hidden_units[i-1], self.hidden_units[i])

            self.W.append(w)

            self.B.append(b)

        w, b = layer(self.hidden_units[-1], self.num_actions)

        self.W.append(w)

        self.B.append(b)


    def evaluate(self, state):
        Y = np.tanh(np.matmul(state, self.W[0]) + self.B[0])
        for i in range(1, len(self.W)):
            Y = np.tanh(np.matmul(Y, self.W[i]) + self.B[i])
        return np.argmax(Y)


def layer(num_in, num_out):
    w = np.random.normal(scale = 1.0/(num_in*num_out),
                         size = (num_in, num_out))
    b = np.random.normal(scale = 1.0/(num_in*num_out), size = num_out)
    return w, b
```

where the function *gen_random* is used to generate a random set of initial parameters for the policy, and the function *evaluate* is used to evaluate a state and return an action for the discrete action space case.

All code used for the GA was written by Chris Beeler which is shown below in parts:

```
def selection(population):
    i = np.random.choice(range(len(population)))
    j = i
    while i == j:
        j = np.random.choice(range(len(population)))
    return population[i], population[j]


def crossover(cross_pop, p = 0.5):
    policy1 = cross_pop[0]
    policy2 = cross_pop[1]
    new_policy = Policy(policy1.shape, policy1.hidden_units,
                        policy1.num_actions, policy1.game)
    for i in range(int(len(policy1.W) * p)):
        w = np.zeros((policy1.W[i].shape[0], policy1.W[i].shape[1]))
        b = np.zeros(policy1.B[i].shape[0])
        for j in range(len(policy1.W[i])):
            for k in range(len(policy1.W[i][j])):
                w[j][k] = policy1.W[i][j][k]
        for j in range(len(policy1.B[i])):
            b[j] = policy1.B[i][j]
        new_policy.W.append(w)
        new_policy.B.append(b)
    for i in range(int(len(policy2.W) * (1 - p)), len(policy2.W)):
        w = np.zeros((policy1.W[i].shape[0], policy1.W[i].shape[1]))
        b = np.zeros(policy1.B[i].shape[0])
        for j in range(len(policy2.W[i])):
            for k in range(len(policy2.W[i][j])):
                w[j][k] = policy2.W[i][j][k]
        for j in range(len(policy2.B[i])):
            b[j] = policy2.B[i][j]
        new_policy.W.append(w)
        new_policy.B.append(b)
    return new_policy

def mutation(mutate_pop, p = 0.05):
    policy = mutate_pop[0]
    new_policy = Policy(policy.shape,
                        policy.hidden_units, policy.num_actions,
                        policy.game)
    for i in range(len(policy.W)):
        w = np.zeros((policy.W[i].shape[0], policy.W[i].shape[1]))
        b = np.zeros(policy.B[i].shape[0])
        for j in range(len(policy.W[i])):
            for k in range(len(policy.W[i][j])):
                w[j][k] = policy.W[i][j][k] + p * np.random.normal()
        for j in range(len(policy.B[i])):
            b[j] = policy.B[i][j] + p * np.random.normal()
        new_policy.W.append(w)
        new_policy.B.append(b)
    return new_policy


def evaluate_policy(policy):
    game = policy.game
    env = gym.make(game)
    reward = 0
    for i in range(10):
        s = env.reset()
        d = False
        while not d:
            a = policy.evaluate(s)
            s, r, d, _ = env.step(a)
            reward += r
    return reward / 10.0
```

```
def generation(population):
    scores = pool.map(evaluate_policy, population)
    scores = np.array(scores)
    l1, l2 = zip(*sorted(zip(scores, population),
                 key = lambda x: x[0]))
    scores = np.array(l1[n_sacrifice:])
    population = list(l2[n_sacrifice:])
    for i in range(n_breed):
        policy1, policy2 = selection(population)
        cross_pop.append([policy1, policy2])
    for i in range(n_mutate):
        policy1, policy2 = selection(population)
        mutate_pop.append([policy1])
    breeds = pool.map(crossover, cross_pop)
    mutants = pool.map(mutation, mutate_pop)
    population += breeds
    population += mutants
    return population
```

where the function *selection* is used to randomly select policies for either crossover or mutation, the function *crossover* is used for crossover operations, the function *mutation* is used for mutation operations, the function *evaluate_policy* is used to evaluate a policy on the chosen environment, and the function *generation* is used to perform a single generation of the GA.

Bibliography

[1] S Carnot. Reflections on the motive power of fire and on machines fitted to develop that power. reprinted in: Mendoza, e.(ed).(1960). *Reflections on the motive power of fire by Sadi Carnot and other papers on the Second Law of Thermodynamics by E. Clapeyron and R*, 1824.

[2] Kyle Mills and Isaac Tamblyn. Deep neural networks for direct, featureless learning through observation: The case of two-dimensional spin models. *Physical Review E*, 97(3):032119, 2018.

[3] Jacob Madsen, Pei Liu, Jens Kling, Jakob Birkedal Wagner, Thomas Willum Hansen, Ole Winther, and Jakob Schiøtz. A deep learning approach to identify local structures in atomic-resolution transmission electron microscopy images. *Advanced Theory and Simulations*, 1(8), 2018.

[4] Angelo Ziletti, Devinder Kumar, Matthias Scheffler, and Luca M Ghiringhelli. Insightful classification of crystal structures using deep learning. *Nature communications*, 9(1):2775, 2018.

[5] Richard S Sutton, Andrew G Barto, Francis Bach, et al. Reinforcement learning: An introduction, 1998.

[6] Peter Eastman, Jade Shi, Bharath Ramsundar, and Vijay S Pande. Solving the rna design problem with reinforcement learning. *PLoS computational biology*, 14(6):e1006176, 2018.

[7] Haichen Li, Christopher R Collins, Thomas G Ribelli, Krzysztof Matyjaszewski, Geoffrey J Gordon, Tomasz Kowalewski, and David J Yaron. Tuning the molecular weight distribution from atom transfer radical polymerization using deep reinforcement learning. *Molecular Systems Design & Engineering*, 2018.

[8] Zhenpeng Zhou, Xiaocheng Li, and Richard N Zare. Optimizing chemical reactions with deep reinforcement learning. *ACS central science*, 3(12):1337–1344, 2017.

[9] Pingchuan Ma, Yunsheng Tian, Zherong Pan, Bo Ren, and Dinesh Manocha. Fluid directed rigid body control using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 37(4):96, 2018.

[10] Marin Bukov, Alexandre GR Day, Dries Sels, Phillip Weinberg, Anatoli Polkovnikov, and Pankaj Mehta. Reinforcement learning in different phases of quantum control. *Physical Review X*, 8(3):031086, 2018.

[11] Thomas Fösel, Petru Tighineanu, Talitha Weiss, and Florian Marquardt. Reinforcement learning with neural networks for quantum feedback. *arXiv preprint arXiv:1802.05267*, 2018.

[12] AL Samuel. Some studies in machine learning using the game of checkers. reprinted in ea feigenbaum & j. feldman (eds.)(1963). computers and thought, 1959.

[13] Arthur L Samuel. Some studies in machine learning using the game of checkers. ii—recent progress. *IBM Journal of research and development*, 11(6):601–617, 1967.

[14] Gerald Tesauro. Practical issues in temporal difference learning. In *Advances in neural information processing systems*, pages 259–266, 1992.

[15] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[16] Gerald Tesauro. Td-gammon: A self-teaching backgammon program. In *Applications of neural networks*, pages 267–285. Springer, 1995.

[17] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[18] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[19] Richard Stuart Sutton. Temporal credit assignment in reinforcement learning. *ProQuest Dissertations and Theses*, 1984.

[20] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[21] MAL Thathachar and P Shanti Sastry. A new approach to the design of reinforcement schemes for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(1):168–175, 1985.

[22] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[23] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

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

[25] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[27] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[28] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[29] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[30] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Vizdoom competitions: Playing doom from pixels. *arXiv preprint arXiv:1809.03470*, 2018.

[31] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. Crawling in rogue's dungeons with (partitioned) a3c. *arXiv preprint arXiv:1804.08685*, 2018.

[32] Anssi Kanervisto and Ville Hautamäki. Torille: Learning environment for hand-to-hand combat. In *2019 IEEE Conference on Games (COG)*. IEEE, 2019.

[33] Janne Karttunen, Anssi Kanervisto, Ville Hautamäki, and Ville Kyrki. From video game to real robot: The transfer between action spaces. *arXiv preprint arXiv:1905.00741*, 2019.

[34] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[35] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. *arXiv preprint arXiv:1902.01724*, 2019.

[36] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[37] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[38] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[39] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[40] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[42] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[43] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[44] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[45] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[46] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[47] David B Fogel and Lauren C Stayton. On the effectiveness of crossover in simulated evolutionary optimization. *BioSystems*, 32(3):171–182, 1994.

[48] Yiming Peng, Gang Chen, Harman Singh, and Mengjie Zhang. Neat for large-scale reinforcement learning through evolutionary feature learning and policy gradient search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 490–497. ACM, 2018.

[49] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[50] Douglas John White. *Dynamic programming*, volume 1. Oliver & Boyd Edinburgh, 1969.

[51] Dimitri P Bertsekas. Dynamic programming and suboptimal control: A survey from adp to mpc. *European Journal of Control*, 11(4-5):310–334, 2005.

[52] Langford B White. A new policy evaluation algorithm for markov decision processes with quasi birth-death structure. *Stochastic Models*, 21(2-3):785–797, 2005.

[53] Douglas J White. Real applications of markov decision processes. *Interfaces*, 15(6):73–83, 1985.

[54] Douglas J White. Further real applications of markov decision processes. *Interfaces*, 18(5):55–61, 1988.

[55] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[56] WR Thompson. On tetradite psi-function with application to apportionment theory. *Bull. Am. Math. Soc*, 40:42, 1934.

[57] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[58] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[59] Claude E Shannon. A chess-playing machine. *Scientific American*, 182(2):48–51, 1950.

[60] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[61] Martin Stuart Silberberg. *Principles of general chemistry.* McGraw-Hill Higher Education New York, 2007.

[62] Chris Beeler, Uladzimir Yahorau, Rory Coles, Kyle Mills, Stephen Whitelam, and Isaac Tamblyn. Optimizing thermodynamic trajectories using evolutionary reinforcement learning. *arXiv preprint arXiv:1903.08543*, 2019.