3-2022

Analysis of Generalized Artificial Intelligence Potential through Reinforcement and Deep Reinforcement Learning Approaches

Jonathan Turner

ANALYSIS OF GENERALIZED ARTIFICIAL INTELLIGENCE POTENTIAL THROUGH REINFORCEMENT AND DEEP REINFORCEMENT LEARNING APPROACHES

THESIS

Jonathan Turner, CAPTAIN, USAF

AFIT-ENS-MS-22-M-171

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENS-MS-22-M-171

ANALYSIS OF GENERALIZED ARTIFICIAL INTELLIGENCE POTENTIAL THROUGH REINFORCEMENT AND DEEP REINFORCEMENT LEARNING APPROACHES

THESIS

Presented to the Faculty

Department of Operations Research

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Operations Research

Jonathan Turner, B.S.

CAPTAIN, USAF

March 26, 2022

AFIT-ENS-MS-22-M-171

ANALYSIS OF GENERALIZED ARTIFICIAL INTELLIGENCE POTENTIAL THROUGH REINFORCEMENT AND DEEP REINFORCEMENT LEARNING APPROACHES

THESIS

Jonathan Turner, B.S.
CAPTAIN, USAF

Committee Membership:

Lance Champagne, Ph.D
Chair

Bruce Cox, Ph.D
Reader

Trevir Bihl, Ph.D
Member

AFIT-ENS-MS-22-M-171

Abstract

Artificial Intelligence is the next competitive domain; the first nation to develop human level artificial intelligence will have an impact similar to the development of the atomic bomb. To maintain the security of the United States and her people, the Department of Defense has funded research into the development of artificial intelligence and its applications. This research uses reinforcement learning and deep reinforcement learning methods as proxies for current and future artificial intelligence agents and to assess potential issues in development. Agent performance were compared across two games and one excursion: Cargo Loading, Tower of Hanoi, and Knapsack Problem, respectively. Deep reinforcement learning agents were observed to handle a wider range of problems, but behave inferior to specialized reinforcement learning algorithms.

Table of Contents

Page

List of Figures

List of Tables

ANALYSIS OF GENERALIZED ARTIFICIAL INTELLIGENCE POTENTIAL THROUGH REINFORCEMENT AND DEEP REINFORCEMENT LEARNING APPROACHES

I. Introduction

Artificial Intelligence (AI) is considered by many to be the next competitive domain; the first nation to develop truly autonomous AI will likely experience socio-economic effects similar to the advent of nuclear energy [1]. In concurrence with this belief, the United States' Department of Defense (DoD), has made the development of human-level AI a priority [2]. In alignment with this effort, this research aims to assess potential issues with the generalization of AI programs through the use of Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) agents as proxies. The remainder of this chapter includes the problem statement and research questions, with details regarding the games used to benchmark progress, the motivation of the problem, and background information surrounding reinforcement and deep reinforcement learning, and an outline of the remainder of the thesis.

1.1 Problem Statement and Research Questions

The objective of this research is to develop a baseline for the development of strong AI. As previously noted, strong AI agents do not currently exist despite ongoing efforts [3]. Additionally, there exists debate whether it will ever be possible for AI to achieve capabilities similar to human's due to the absence of human functions and biological functions [4]. While this may appear a superficial concern, research in other domains has shown that the nature of the environment can have unexpected

impacts on the development of organisms [5], so it is reasonable to assume that there may be some aspects of human nature that are not easily captured or measured through artificial means. Therefore, this research intends to capture issues in the generalization of problem solving capabilities of weak AI agents. Specifically, DRL agents are used because they are arguably the most similar in capability to strong AI while RL are used to baseline performance as a relatively weaker AI. To compare progress, two similar but increasingly difficult problems are evaluated and compared and one excursion is conducted: Tower of Hanoi (ToH), Cargo Loading (CL), and Knapsack (KP) as the excursion. The agents' performance in the various games will be used to evaluate differences in their respective strengths and weaknesses as well as potential costs in terms of time to train and develop each agent to accomplish different games.

1.1.1 Tower of Hanoi

The standard ToH game consists of three pegs with rings of increasing size stacked upon one peg, typically the left side peg. The objective of the game is to transfer all rings from this side to the right side. Each ring moves by first removing it from its current peg and then placing it unto the target peg. Larger rings are not permitted to stack upon smaller rings and preference is given to completing the task in the least number of moves. While the action space for the agent is very small relative to the CL, the small space and weights of the objects creates its own unique challenges. However, the ToH has a well-known pattern that once identified facilitates solving all ToH problems, and the game can be solved optimally in $2^N - 1$ moves where N is the number of rings. In other words, this game helps serve as a measure of pattern recognition. Increasing difficult instances of this problem would include increasing the number of rings. A visual representation of this problem can be found in **Figure 1**.

Figure 1: Visual representation of the Tower of Hanoi (ToH) game initial condition: The rings must move from the left peg to the right in as few moves as possible. Larger rings cannot be placed on smaller rings

1.1.2 Cargo Loading

CL involves the storage of three-dimensional objects into a three-dimensional space. Each object has a unique weight and heavier objects cannot be placed upon lighter objects. Furthermore, objects are not allowed to hang over other objects and must be stacked "neatly", (see Figure 2). For the purposes of this research, both cargo items and storage areas are restricted to rectangular prisms. Testing for agents in this problem set include expanding the dimensions of the storage space, expanding the dimensions of the storage items, and changing the weights of the storage items. This problem set is unique in the higher number of actions the agent is required to evaluate and the potential for alternative optimal solutions. This problem set is also inspired by previous research sponsored by Air Force Research Laboratory (AFRL)[6].

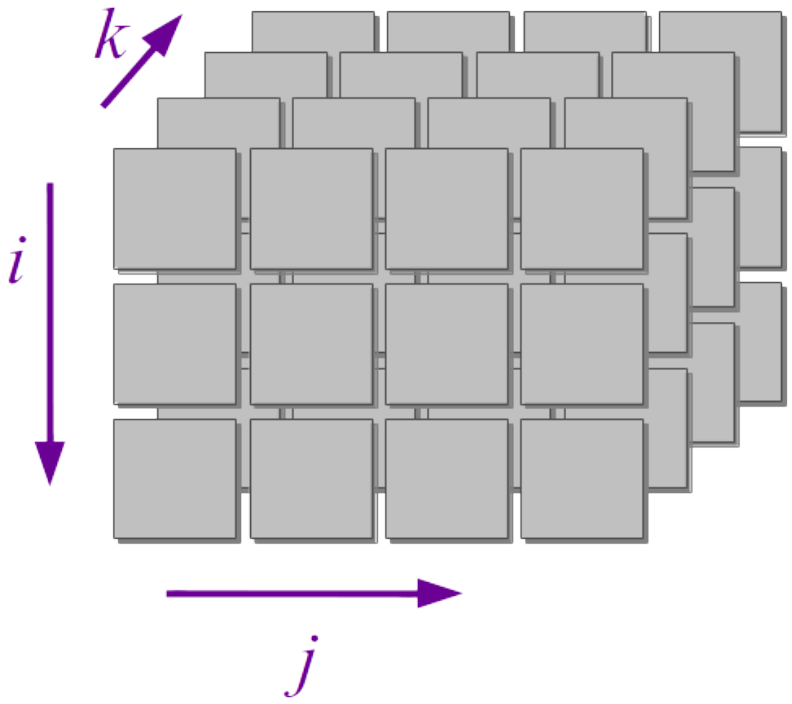


Figure 2: Dimensional representation of the cargo loading problem: The cargo loading space is shown as rectangular prism. Items are stored one by one into the space provided there is enough space for the object. Objects also have their own weights. Larger and heavier objects cannot be placed upon smaller objects.

1.1.3 Knapsack

This problem is a well-known in integer programming: the agent has list of items of differing values and a limited space to store them and attempts to maximize value given a size constraint. The knapsack problem has generally seen the most success using heuristics and evolutionary algorithms due to the computational complexity inherent to the problem. Increasing difficult of this problem would consist of increasing the number of items for the agent to differentiate from when selecting the optimal solution.

1.2 Motivation

While the creation of strong AI is considered a prized achievement, its only relatively recently that computational power has reached a point where the pursuit of human-level AI seems attainable. Since 1948, the concept of self-learning machinery has been on the forefront of the scientific conscience [7]. As of 2022, modern AI algorithms have already served to enhance accuracy and precision in the medical field [8]. However, all of these programs have been examples of weak AI. These programs are often limited to specific problems or require human input or assistance in order to operate. Commonly used examples of weak AI include virtual assistants like Cortana which can follow limited voice commands such as, "solve 3+3" but would not be able to follow the voice command, "solve my math homework."

A strong AI attempts to solve large varieties of problems with little to no human input. As of 2022, strong AI programs do not exist. However, Of these weak AI programs, arguably the closest to resembling strong AI is deep reinforcement learning. Deep reinforcement learning algorithms are a form of unsupervised machine learning algorithms, a form of computer program capable of adapting and learning from its environement through trial and error. It is also a Neural Network (NN) abstraction of RL and is a form of unsupervised Machine Learning (ML). Thus, to baseline AI development research, we focus on solving a series of games with reinforcement and DRL.

1.3 Background

To understand deep reinforcement learning, we first must discuss reinforcement learning. Reinforcement learning is an unsupervised machine learning method where the entire state-space of the problem is generated, and the agent must navigate to a specific goal solution. In other words, without human input, the computer agent

attempts to map out all the possibilities of the problem and determine a policy with regards to which actions need to occur to reach the goal from any state. An example would be a draw-by-numbers picture where the final state is the completed picture, the policy would be to connect all the numbers starting from one in sequential order and each action would be the agent drawing a line connecting any two dots. In this scenario, the agent would learn by taking a certain percentage of random actions to explore the states and slowly identify the sequence of states that most reliably lead to the end state. For the purpose of this research, this is done by the Bellman equation which will be better explained in Chapter 2.

While powerful, there are two major weaknesses of the reinforcement learning approach. First, as the learning process involves the use of the entire state space, changes to the problem can change the state-space and require the agent to relearn the entire problem, hampering generalizability. Referring to the draw by numbers, an example would be inserting a random point inside the picture. The agent will now have to identify how this point fits in relation to all other points in the problem. For a simple 10-point draw-by-numbers group, the problem difficulty increases by a factor of 10. Furthermore, even if the agent has learned to handle draw by numbers, this does not mean the agent has learned to draw the same picture without numbers or even a free drawing. The second major issue is related to the first. Because the learning process requires the entire state-space, computer memory can quickly become a limitation. Under the same assumptions for the draw-by-numbers, assuming the agent always starts from the same spot there are 362,880 potential states, adding an additional point increases the number of states to 3,628,800. Hence, there exists an incentive to solve these problems without relying on the entire state space, the main strength of deep reinforcement learning.

Deep reinforcement learning addresses reinforcement learning's reliance on the

state-space mapping through deep neural networks. A neural network acts to collect inputs which are processed through a layer of mathematical functions simulating human brain functions called neurons to generate an output. Each neuron has its own activation function and weight on the output. A deep neural network expands this idea by stacking layers of neural networks upon one another with the outputs of one layer serving as inputs to the following layer. Through the processes of feed-forward and back-propagation during the training of the neural network, the weights of the neural network are slowly updated, eventually increasing model accuracy. In this case, the neural network is trained on the action-space as opposed to the state-space. As the action-space consists only of what actions are possible, the size of the space is generally much smaller than the state space. Through this method, the deep reinforcement learning approximates the Bellman-equation, the mathematical equations that create the policy that a standard reinforcement learning model follows. In an ideal scenario, this would enable the DRL problem to solve more types of problems than the reinforcement learning agent and be more computationally efficient.

1.4 Methodology

Python was used as the main platform for development of the models, agents and training, however the common reinforcement learning package OpenAI was not used due to incompatibilities with NengoSPA, a cognitive architecture that was considered more critical for future research. Code for the deep reinforcement learning environment is provided in the appendices. Much, but not all, of the environments used for training share a common internal logic for the placement of items and checks for achievement of the goal state. Details on these environments and the differences between them are found in Chapter III.

Two types of agents were trained and challenged to complete the problem sets de-

scribed in Section 1.1: RL agents and DRL agents. These agents followed Q-Learning and Deep Q-Learning respectively for better performance comparison. Explanations for the history and purpose of Q-Learning and Deep Q-Learning are found in Chapter II, while details for their specific implementation in this research are explained in Chapter III.

1.5 Assumptions and Limitations

Considering the variety of problems to solve, hyper-parameter tuning is not applied to any model throughout the process and was considered outside the scope of this research. This is to avoid biasing the model performance to any specific problem. As a result, model performance may fluctuate depending on the architecture developed in practice. Furthermore, due to the amount of time required for a neural network to train, weights from one model are used as a baseline for training more difficult versions of the same problem. Additionally, each of the Deep Q-Learning models use the same environment so that an identical input structure can be maintained.

As discussed in Section 1.4, OpenAI Gym, a popular reinforcement and deep reinforcement learning package is not used. This is to facilitate follow on research with cognitive neural architectures, specifically NengoSPA, in future research. As these languages currently lack inter-connectivity with OpenAI Gym, efforts have been made to ensure the code developed is as reproducible in NengoSPA as possible.

1.6 Organization of Thesis

This document is organized as follows. Chapter II provides an overview of relevant literature. Chapter III details the process of developing RL and DRL agents and their environments, including the Artificial Neural Network (ANN) architecture. Chapter IV presents the results of evaluating the various agents' performance. Finally,

Chapter V discusses the conclusions drawn from the results.

II. Background and Literature Review

The intent of this chapter is to explain the history and current state of Artificial Intelligence (AI) algorithms as well as to provide details surrounding the origin and application of the techniques used. As Machine Learning (ML) is directly linked to the development of artificial intelligence and pertinent to this research, a brief history of ML will be reviewed in Section 2.1 . Because the application of an Artificial Neural Network (ANN) with Reinforcement Learning (RL) methods serves as a focus for much of this research, there will be further elaboration on each of these technique in Section 2.2 and Section 2.3. Information surrounding Deep Reinforcement Learning (DRL) will also be included in section 2.3. In section 2.4, the interconnection between ML and AI will be expanded. Since future research is intended to leverage cognitive architecture, an overview of this topic is provided in section 2.5. Finally, recent relevant achievements in AI will be recounted in section 2.6

2.1 Overview of Machine Learning

Machine learning is an area of study focused on the design of programs capable of taking raw data and determining patterns or creating predictions. The first modern machine learning algorithm can be traced to Frank Rosenblatt in 1957, who created a prototypal ANN he dubbed the "perceptron" [9]. From this first step, pioneers in control theory, computer science and stochastic science all began developing their own approaches to problem sets they faced. This convergence of statistical and computer science efforts became what we call machine learning today and gave rise to many statistical techniques we use today, such as ANNs and RLs.

While the range and utility of machine learning techniques has increased with time, the key feature machine learning algorithms share is the capability to define

their own parameters from the data they process [10]. Machine learning algorithms can be broadly divided into one of four learning categories, one of two learning methods, and one of two main purposes. Potential learning categories are supervised, unsupervised, semi-supervised, and reinforcement learning. Learning methods include batch learning and online learning. The purpose of a machine learning algorithm can be described as data point comparison or pattern detection [10].

In supervised machine learning algorithms, the program is given a labelled response for training, the process of tuning parameters to maximize prediction or classification accuracy. The program uses this defined response as a reference to calculate its loss, or how much the predicted data differs from the true measurement. This loss is propagated throughout the weights in a which enables the model or agent to converge on a correct prediction or classification.

In unsupervised learning, algorithms lack a response feature and instead attempt to minimize or maximize some other metric, such as the variance within a certain number of groups. These algorithms are often used to analyze clusters or detect similarities among data points. K-means clustering is an example of an unsupervised machine learning method that attempts to identify groups within the data by minimizing the variance within identified groups. As a brief aside, its important to note that this requires the use of a hyper-parameter term to specify the number of groups to identify, another common facet of ML. While ML programs can control how they estimate the data, hyper-parameters serve to modify how the model updates the parameters, such as specifying a learning rate to impact how strongly the model is influenced by its losses.

Semi-supervised learning is a mixture of the two previous learning methods, where some, but not all, data points in the training set are labelled. These points are used as references for the unlabelled data during training and can be used for such tasks

as content and topic mapping in text analysis.

In reinforcement learning, an agent attempts to navigate an environment with its own internal rules and logic and is rewarded and punished according to its ability to accomplish its objective. The agent repeatedly takes actions that influence its environment and observes the results. Through trial and error the agent learns how these actions impact the environment and then develops a policy to maximize its reward for problem completion. Reinforcement learning is expounded upon in Section 2.3.

As for the learning types, the main distinction is whether the agent or algorithm requires the whole data set to begin training or if the agent works through the steady assimilation of more data points. Batch learning refers to the former practice and is very common for ANN. For the purpose of this research, mini-batching, the partitioning of data for training in a batch-like method, will also considered as the same category. The alternative method is on-line learning, where the algorithm learns and gradually increments its predictions and inferences with data given to it over time.

Data point comparison can be when an algorithm classifies a data point based on its most similar data point and pattern detection occurs when an algorithm attempts to predict or classify data points using trends within the data. The K-Means algorithm previously mentioned is an example of of data point comparison, whereas regression methods and policy development in some reinforcement learning methods can be considered a pattern detection. The focus point for most of this research consists of the application of ANN a historically supervised learning method algorithm with RL techniques.

2.2 Overview of Artificial Neural Networks

As the name implies, ANNs are a form of mathematical models designed to make predictions using a process imitating how humans think, albeit simplified. At the most

fundamental level, the network is composed of singular units called called neurons. Each unit is composed of a bias, an activation function and weights to apply to the input vector, as shown in fig. 3. For early neural networks, the activation function typically used was the threshold logic unit postulated by McCulloch and Pitts which simply output a value of one if a certain level of stimulus was achieved or nothing otherwise[11]. For simple neural networks, such as the original perceptron, the model would simply receive an input, multiply these inputs by the weights, add a bias to this value and if this value exceeds a certain threshold, the activation function would output a value of 1.

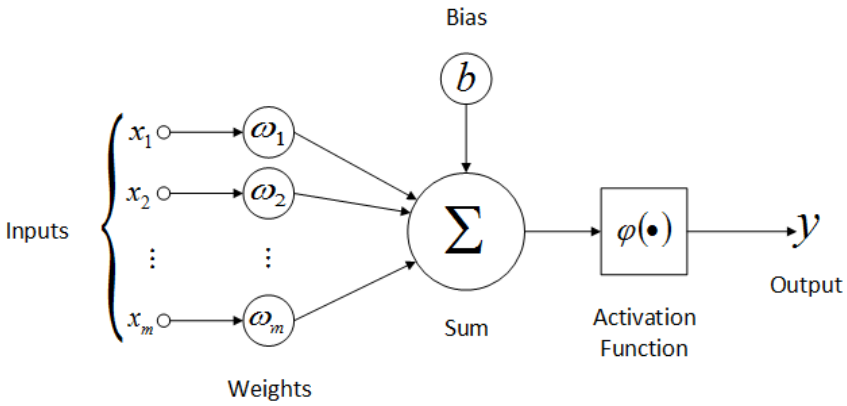


Figure 3: Visual representation of a single neuron from an Artificial Neural Network, figure courtesy of Pinjare 2013 [12]

.

Unfortunately, these models do not naturally provide the correct solution upon initiation, in fact it must learn the problem by training on the data set. While aspects of the training method have evolved over time, Algorithm 1 has served as a foundation for much of this development. Let $(\mathbf{v}, \mathbf{t})$ be a vector mapping of training inputs upon training outputs while $\mathbf{y}$ is a vector representing the true values of those inputs and $\mathbf{w}$ be the vector representing weights on the neural network input layer.

The model trains by following the algorithm depicted in algorithm 1. As is implied by the algorithm, many of these early ANN followed supervised learning methods.

Algorithm 1 Perceptron Training Rule Algorithm as depicted in Gurnery 2018 [13].

1: **while** $\mathbf{y} \neq \mathbf{t}$ **do**
2: $\quad \mathbf{d} = (\mathbf{t} - \mathbf{y})\mathbf{v}$
3: $\quad \mathbf{w}' = \mathbf{w} + \mathbf{ad}$ ▷ $\mathbf{a}$ represents the learning rate
4: $\quad \mathbf{w} = \mathbf{w}'$
5: **end while**

As research into ANN progressed, it was discovered that stacking layers of neurons upon one another creates better performance than simply adding more neurons to a single layer, in a manner similar to the one depicted in Figure 4; the first working model using this method was created by Ivahenko and Lapa in 1967 [14]. These more modern ANNs consisted of layers of neurons working in tandem to model and predict outputs. Layers between the output and input function were termed hidden layers and a neural network with at least one hidden layer was classified as a Deep Neural Network (DNN), also known as a Multilayer Perceptron (MLP) [13]. Because each unit is its own function, the model is extremely flexible. In fact, ANN are known universal approximaters, meaning provided sufficient neurons and time ANNs can approximate any Borel measurable function [15]. Unfortunately, these MLPs were limited by the computational resources of the time. While the operation of each neuron is simple, the nature of the network meant that the weight matrix rapidly grew unwieldy with increases in neurons and layers. In other words, despite the fact the majority of the calculations were quite simple, the sheer quantity of calculations rendered ANN impractical to use.

Ultimately, ANN research entered a few periods of dormancy termed "AI Winters" and a new spring would not be seen until the development of better Graphics Pro-

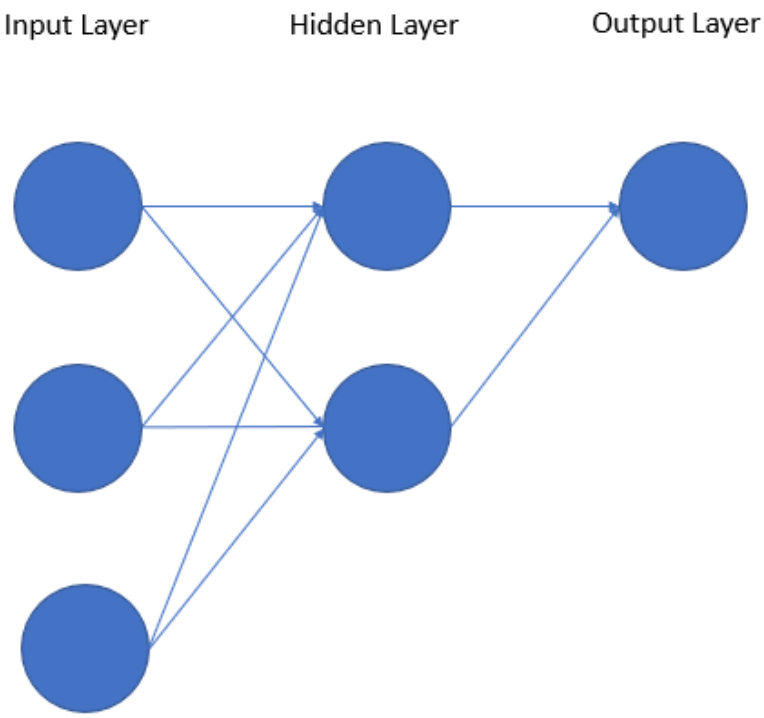


Figure 4: Visual representation of deep neural network. Each circle represents an individual neuron with arrows directing outputs to following layer

.

cessing Units (GPU) and parallel computing methods [9]. Since this new awakening, nearly every facet of ANN has been incremented upon. More computationally efficient activation functions capable of handling nonlinear data have been established, such as the Leaky Rectified Linear Unit (LeakyReLU). The development of Stochastic Gradient Descent (SGD) further enhanced the flexibility of ANN improving on the original perceptron training rule to allow the model to learn convex nonlinear functions computationally efficiently. Even the concept of learning rate was improved, with optimizers such as AdaMax surfacing, decreasing the time required for ANNs to train by modifying the learning rate during training to quickly converge towards the optimal point.

As new techniques emerged to optimized and enhance neural networks, new struc-

tures of neural networks emerged as well. In the image classification problem set, convolutional neural networks emerged, which used filtering layers on image data to enhance classification. Other structures of neural networks emerged as well, such as recurrent neural networks. However, one of the more recent structures developed were DRL, which will be further explained in section 2.3.

2.3 Overview of Reinforcement Learning and Deep Reinforcement Learning

The origins of reinforcement learning are markedly more murky than the history of ANN. What we know today as reinforcement learning is actually a merger of two different concepts each with their own unique origins. The earlier concept was rooted in psychology and animal learning. The other thread was known as optimal control and originated in the 1950s. Optimal control was concerned with the optimization of dynamical systems[16] and led to many of the mathematical methods informing modern reinforcement learning. These threads and a few other extemporaneous threads merged in the 1980s creating the concept of reinforcement learning we have today.

The origin of optimal control is closely linked to Dr. Bellman's work in 1957: the Bellman Equation [17]. Let s be a current state in the problem, such that, $\mathbf{V}(\mathbf{s})$ is the value of said state. Then $\mathbf{R}(\mathbf{s}, \mathbf{a})$ will be the reward associated with taking an action a at the current state. The state resulting from this action will be s' and γ will be the discount factor for this next step, or a sort of expectation for this state. For a deterministic environment the Bellman equation takes the following form:

$$V(s) = \max(R(s, a) + \gamma V(s')) \tag{1}$$

While for a stochastic environment, the following equation holds true, but with $P(s, a, s')$ representing the probability of transitioning to the next state, s', following

action, a, in state s:

$$V(s) = \max(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')) \tag{2}$$

From these equations spawned the disciplines of dynamic programming and with Ronald Howard's policy iteration method[18], Markov Decision Processes. In this manner, we can see there were mathematical attempts to quantify and model the consequences of actions in both deterministic and stochastic environments. This analytic bedrock is what helped with implementation of ideas developed in the second thread.

The second discipline from which modern reinforcement learning got its start originates from the fields of psychology and animal learning. While some of these concepts existed since the 1800s, the first clear codification of these concepts originated with Edward Thorndike's 1911 "Law of Effect":

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connection with that situation weakened, so that, when it recurs, they will less likely to occur. The greatest satisfaction or discomfort, the greater the strengthening or weakening of the bond.[19]

In essence, the "Law of Effect" states that an animal will seek out circumstances that have historically given it pleasure and avoid circumstances that have historically given it pain. In 1948, Alan Turing discovered that machines could do something similar [7]. Thus, the error of machines learning through trial and error had begun, but it would not yet blossom until the 1980s. The reason for this period of dormancy is largely unclear, but in retrospect there was much confusion between what exactly

RL was versus ML in general and with respect to ANN [16]. In any case, the efforts of Harry Klopf in 1982 at least in some part revived the field by clarifying the distinction between reinforcement and supervised learning [20].

Currently, reinforcement learning is a thriving discipline closely linked with machine learning with identifiable and distinct approaches. In summary, these approaches are value-based, policy-based and model-based [21]. One of the most popular methods used in this research is Q-Learning, a value-based method pioneered by Watkins in 1989 [22]. Let $q(s, a)$ be a function which returns the value of an action, a, in state, s. Therefore, the approximation of this function is $Q(s_t, a_t)$, where t represents the number of steps Q has spent updating. Thus, $Q(S_t, A_t)$ is the value approximation of all state-actions after t steps, (S_t, A_t). Thus Q-Learning seeks to maximize the reward R according to Algorithm 2.

Algorithm 2 Q-Learning Algorithm as shown in Sutton 2018

1: Set hyper-parameters: step size, $a \in (0, 1]$ and small $\epsilon > 0$
2: Initialize arbitrarily $Q(s, a) \ \ \forall S, A$
3: Set $Q(finalstate,) = 0$
4: **while** S is not final state **do**
5: Choose A from S from policy derived from Q
6: Take action A, observe R, S'
7: Update $Q : Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
8: Set $S = S'$
9: **end while**

This algorithm serves as the mechanism for training RL agents and is shown to converge reliably even in stochastic cases. However, it is also computationally intensive as the entire state space must be mapped and updated. For large state spaces this process quickly become unwieldy. This is where the DRL method Deep

Q-Learning is beneficial.

As discussed in Section 2.2, ANNs are universal approximaters; a relatively less computationally demanding neural network can be designed to approximate the Q function making problems with large state-spaces solvable. This is done by training the neural network on the state-action pair as inputs with the next state and reward as the labelled training outputs. In this manner, the model is able to reduce its exploration of the state-space to an exploration of the action space. Under this method however, a few structural modifications are required to be made to the agent and algorithm to facilitate the procedure.

Because the neural network is constantly updating, stable predictions are a potential issue. To solve this issue, upon initializing the agent, a primary and target network are generated, the primary network updates in regular batches while the target network updates after a set number of moves. Because the target network is not constantly updating it can create more stable predictions, and an additional benefit is that the number of computations are reduced. To generate the batches used, as the model progresses a record of states, actions, next states and the rewards associated is recorded. As memory is not infinite, a limit is defined for the memory and upon reaching the limit, the memory is wiped until new data points are gathered. To avoid biasing the information fed into the model towards early or irrelevant data points, a random sample from this memory of the specified batch size is taken. This is possible because the state space for Q-Learning models is assumed Markovian, meaning the current state provides sufficient information for the next state. However, as the agent will naturally prefer to follow what it assumes is the best option each step, it is likely to fail to discover better methods. As a result, a higher percentage of random actions are considered in the early training period and decrease as training continues. This method is called Epsilon Greedy Action Selection [23], shown in Algorithm 3.

Algorithm 3 Deep Reinforcement Learning General Algorithm

1: Initialize the starting state for environment.
2: Initialize Policy Neural Network
3: Clone Neural Network for use as Target Neural Network
4: **while** Optimal Output has not been reached **do**
5: **for** each time-step **do**
6: Choose between exploration or exploitation randomly
7: Execute Action
8: Observe Reward and Next State
9: Store experience in replay memory
10: Sample random batch from replay memory
11: Preprocess states from batch (normalization)
12: Pass batch of preprocessed states to policy network.
13: Calculate loss between output Q-values and target Q-values using target network
14: Gradient descent updates weights in the policy network to minimize loss.
15: After x time steps, update Target Neural Network weights with the Policy Neural Network.
16: **end for**
17: **end while**

2.4 Machine Learning and Artificial Intelligence

From the same fusion of ideas that birthed the concept of machine learning, another idea began to arise: can humans make computers think? Where machine learning proved that humans could teach computers to learn, the issue is the specificity of the learning. A Support Vector Machine (SVM) could potentially learn to classify

several businesses based on their probability of success, but the same SVM will struggle to predict the probability an individual will default on their loan despite some similarities in metrics. One of the first attempts to solve this issue of generality was reported by Allen Newell and Herbert Simon in 1958, where the "General Problem Solver" was created [24]. Incredibly, this first prototypical AI could solve multiple different problems, however it had shortcomings in addressing large state spaces and complex environments.

This attempt signaled a shift in history. John McCarthy developed the AI programming language Lisp in 1958 and with his transfer to Stanford and subsequent foundation of the AI Laboratory, AI development began in earnest [25]. Unfortunately, in less than a decade AI programs began to run into serious roadblocks, such as computational limitations at the time, the curse of dimensionality, and the black box nature of many early algorithms. Thus, AI research slowed to a crawl and true continuous and steady support of AI research did not resume until the late 1980s with the rediscovery of backpropagation, a technique for training feedforward neural networks, and advances in computer technology. In the early 2000s, the development of big data and statistical techniques have reawakened a public interest in Artificial General Intelligence (AGI)) and Human-level Artificial Intelligence (HLAI), which will eventually be know as "strong AI".

Of note, not all current AI systems are what humans would traditionally consider as intelligent. For instance, the preprogrammed enemies in any number of video games are AI but lack true cognition. From this discrepancy rose the terms weak and strong AI, weak referring to the more limited machine learning algorithms and strong AI referring to machine intelligence comparable to humans. In the words of Mark Bishop "weak AI focuses on epistemic issues relating to engineering a simulation of human intelligent behavior, whereas strong AI, in seeking to engineer a computational

system with all the causal power of a mind, focuses on the ontological" [26].

2.5 Cognitive Architecture

This research leverages the current advances in neural networks with developments in cognitive architecture to lay the foundations of further development. As the concept of artificial intelligence grew, so did the underlying science regarding cognition and awareness. To put it succinctly, many people tried to model the mind only to find gaps between programming behaviors and creating thoughts. However, what has emerged from this struggle is the understanding that to some degree a cognitive creature must be able to simultaneously reason and interact with its environment [27]. From this understanding, the notion of cognitive architectures begun to develop. Cognitive architectures serve as the framework to translate human-like awareness understanding into machine functions and processes [28]. Broadly speaking, there exist three main approaches towards developing a cognitive architecture: dynamical, connectionism and the symbolic approach.

The dynamical hypothesis in cognitive science posits that cognitive agents exist as dynamic systems. A dynamical system is one in which at least one of the following statements are true: the states are quantitative, states and time are interdependent, or rate of change is dependent [29]. The dynamical hypothesis exists in two parts. First, it claims that cognitive agents are dynamic systems with the previously described properties. Secondly, it proposes that the best way to understand cognitive processes is through a dynamical approach. This approach emphasizes that dynamical frameworks are used in many scientific applications and have been shown to be empirically successful; cognition naturally changes with time, a core feature of dynamical systems; cognitive agents will interact with a dynamical environment, encouraging dynamical approaches; and lastly, just like natural cognitive systems,

dynamical systems tend toward a natural stability.

Connectionism is a very different approach to dynamical models. The general premise of connectionism is that neural nodes can be modelled to capture specific cognitive ideas, which can range from letters to abstract patterns and concepts. These nodes are connected to other nodes by differing degrees and can be activated in the presence of their relevant stimuli and by the activation of related nodes depending on their relation to this node [30]. The quantity of nodes activated in accordance with a stimulus can be aggregated to an appropriate function which in turn defines a new state. Layers of these nodes can provide greater levels of abstraction, and backpropagation enables policy updates on the connections between nodes through each layer. Thus, a connectionist model focuses on relationships between cognitive ideas.

The symbolic method is so far the most dominant method in the current studies. The symbolic methods can generally trace their lineage to the original General Problem Solver developed by Newell and Simon in the late 1950's. A relatively recent symbolic cognitive model is ACT-R, developed by John Robert Anderson, Christian Lebiere and Michael Matessa [31]. Symbolic approaches attempt to quantify entities into computable numeric values that are used for computations and focuses on syntax as opposed to connectionism which focuses on semantics.

Each of these three paradigms offers its own unique benefits and drawbacks. For the purpose of this research, a holistic view incorporating each paradigm is adopted. Specifically, we adopt the Semantic Pointer Architecture (SPA) framework as described in Chris Eliasmith's How to Build a Brain: A Neural Architecture for Biological Cognition [27]. SPA is an architecture that attempts to bring forth the best attributes of existing cognitive architectures, while simultaneously rooting itself in biological and psychological understanding of cognition as much as possible. SPA uses

data compression on semantic pointers to enact operations similar to how ACT-R operates but providing a better opportunity to glean insight by valuing representational meaning more than syntax.

2.6 Recent Achievements in Artificial Intelligence

Notably, while cognitive insights have been a core interest in AI research, progress along this domain have been scant. AGI is something that has not been created yet and some experts doubt it will be seen until the 2040s at the earliest [32]. Despite strong AI's poor performance, weak AI has made dramatic progress [33]. As a result, practical measures of AI capability have been sought. One of the most common benchmarks for AI progress is competition, both with humans and other AI engines in games, preferably those with large state spaces. Because traditional computer systems struggled with large state spaces and human performance can act as a reasonable reference for capability, games make perfect sense for comparing AI utility with humans in a practical and repeatable manner. One of the most promising of these enterprises was Deepmind's AlphaStar.

AlphaStar was a multi-agent reinforcement learning algorithm trained to play Starcraft II at a level beyond 99.8% of the player base [34]. In the development of AlphaStar, potential key features were identified using a supervised learning algorithm. These key features were then used to inform statistics for the reinforcement learning algorithm. The significance of this methodology is Starcraft II is a real time strategy game combining concepts such as economy management, strategic control of units, navigation of terrain and intelligent response to an adversary in an environment with imperfect knowledge. As such, Deepmind's AlphaStar represents a major step forward in practical use of weak AI systems. One key insight of Deepmind's work was the use of neural networks to act as key features in the development of hyper-parameters.

Deepmind used population-based training in an effort to efficiently explore high dimensional feature spaces [35]. This method involves using several parallel agents to explore the feature space simultaneously while each agent simultaneously informs the others of potential successful policies. In essence, by aggregating weak AI it has been shown that a stronger AI can be developed. It remains to be seen whether or not this is the only or best way for AI development ot progress.

III. Methodology

The purpose of this chapter is to describe the design of environments and the neural networks used. The first half of this chapter will discuss the programs used, environment development and other aspects not particular to any of the problems solved. The logic used for move verification will be described, which is imperative for both the Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) environments. Following this section, the model development and training for the RL and DRL agents will be described. For the DRL, there will also be a discussion regarding the development of the neural network. The second half of the chapter will consist of specific details of how each agent was trained for the specific problems presented. The DRL models were designed with a Deep Q-Learning approach for better comparison to the Q-Learning reinforcement models.

3.1 Programming Platform

The Python programming language was used with multiple specialized libraries throughout this work mainly for its compatibility with the TensorFlow and Keras. Google Colab Pro Plus was used in lieu of dedicated Graphics Processing Units (GPU)s. Furthermore, hyper-parameter optimization was not conducted on the DRL agents to maintain the Artificial Neural Network (ANN)s internal structure and avoid specializing the neural networks to any particular problem. However, the maximum steps permitted in episode training were modified in testing of the DRL models. Helpful libraries for the reinforcement learning development were itertools, pandas and numpy.

3.2 Move Verification Logic

In order to establish the state space for the reinforcement models and in order to verify action validity for the deep reinforcement models, an internal validity check had to be conducted for each state and action respectively. For most of the agents, this proces was the same, with the notable exception being the Tower of Hanoi reinforcement learning agent. For this reason, the following subsection will detail how the logic works in the general case and the subsection after will explain how the Tower of Hanoi (ToH) RL model works specifically.

3.2.1 General Rules for Internal Validity

To lay the foundation of the environment, two object classes were established: the storage space and the cargo object. The simpler of the two classes was the cargo class. which served as a representation for cargo in the Knapsack (KP) and Cargo Loading (CL) problems and rings for the ToH. Upon initialization this class was given spatial dimensions as well as a weight, an individual value and a boolean status of being stored. In anticipation of working in three dimensional space, these objects also had three methods associated with flipping the cargo along any of the three main axes, so long as the object was not stored. These methods are shown in Figure 5.

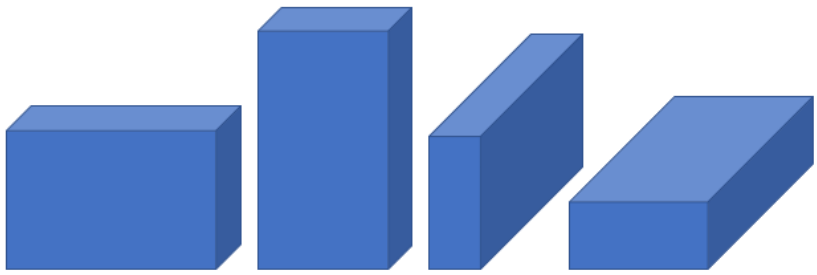


Figure 5: Depiction of ways cargo can flip for visual clarification

The more complicated of the two classes was the storage space class. Similar to the cargo class, the storage space also instances with dimensions, but it also receives a list of cargo to store as a packing list. This class controls nearly all of the internal physics and logic within the environment; the storage space records the initial parameters of all of the cargo within the packing list in the case of removal, restarts and other action requirements. Upon storing or moving an object within the storage space, the storage space conducts a check on the coordinates given to determine if the cargo fits in the bounds of the storage space if placed at the target location. Provided that the cargo has sufficient space within the container, additional checks are made to prevent cargo from existing in overlapping spaces, as illustrated in Figure 6.

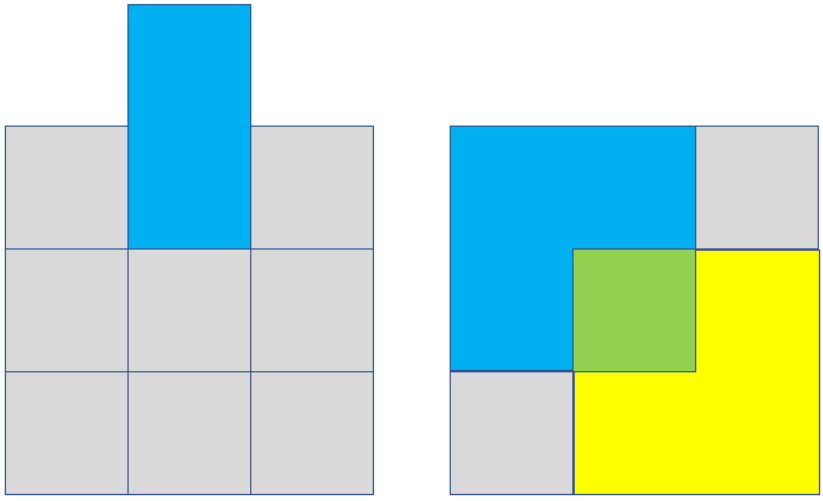


Figure 6: Left: visual representation of storage logic check for cargo dimensions. Prior to placement, the storage space (grey) verifies cargo (blue) dimensions do not continue outside the space. Right: visual representation of cargo collision issues. Blue and yellow each represent unique cargo items. The storage space checks to avoid the green overlap for each item in the space.

To better simulate real life environments, cargo are only allowed to be placed on

the floor of the storage space or on top of another cargo with dimensions greater than or equal to its self, i.e cargo could not float in the middle of the storage area. Additionally, heavier cargo are not allowed to be placed on top of lighter cargo. These rules are illustrated in Figure 7. This effectively reduced the three dimensional coordinate space to two dimensions, helping to reduce the action space.

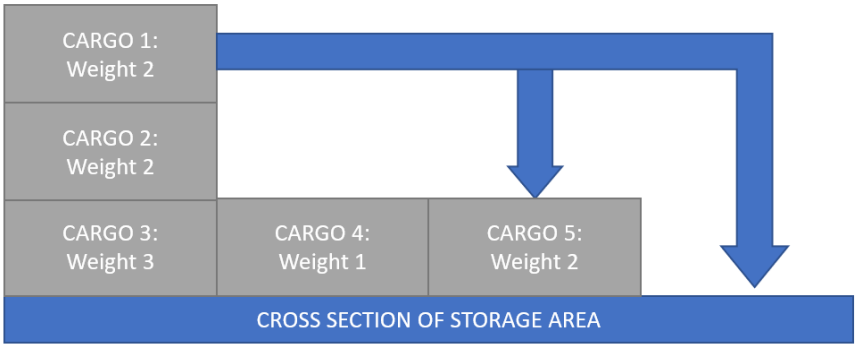


Figure 7: Each grey square represents a stored item while the blue rectangle represents the floor of the cargo space if viewed along the yz plane. The blue arrows indicate locations cargo 1 could potentially move. Please note that weight metrics are not cumulative such that the weights of cargo 1 and cargo 2 are valid for cargo 3.

To represent and record changes in storage space in three dimensional space, two arrays are generated and maintained: the occupational array, which recorded the location of each cargo item in three dimensional space and the loading array which recorded the weights at each location in the same space. These arrays served as references for the previously mentioned checks and as some of outputs for the agent's observations, as will be discussed in further detail in section 3.5.

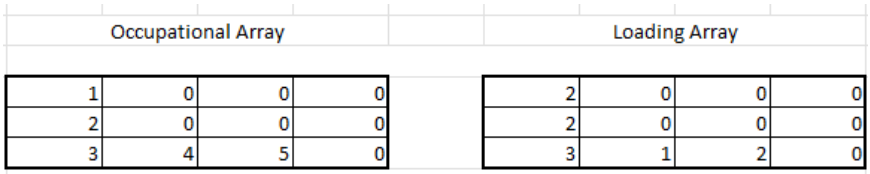


Figure 8: These arrays serve as a visual reference for how the occupancy and loading arrays work. The depicted arrays are the same cross-section as in Figure 7

Between the reinforcement and deep reinforcement versions of the CL problem, the main difference was the output for failed actions by the agent. In the RL version of the problem, as will be discussed in section 3.5, the state space was elucidated in advance and impossible states were given a penalty of negative infinity. Therefore, the agent would be prevented from pursuing an illegal action. The state space development will be discussed further in section 3.3. In the DRL case, however, depending on whether the agent was exploring or exploiting the state-space, the agent would either receive a penalty upon attempting an illegal action or would continue to randomly select moves until a legal move was chosen, respectively.

3.2.2 Internal Validity in Reinforcement Learning Tower of Hanoi

The basis for the reinforcement learning methodology directly follows Kurt Peeks [36] attempt to follow the Watkins and Dayan's methodology [37] and will be discussed further in section 3.3. The current state of the ToH is indicated by a tuple of length N, where N represents the number of rings. In this tuple, the value of the ith entry is an integer, j, which serves to indicate the position of the ring on one of the three towers. This organization also means that the weight of the rings can be interpreted as i as well. Moves are annotated as a tuple of length two, (a, b), which represents the movement of the smallest ring from peg a to peg b. A helper function, $z(a, b)$ is also established to return the minimum index of the ring, i, on peg a and compare its size

to the minimum index, i, on peg b. Thus, when given a current state, the function will receive a move as input, the helper function will verify the moves legality, and if legal the state will transition.

$$\min z(a, b) \tag{3}$$

3.3 Reinforcement Learning

For each of the reinforcement models a similar method was conducted. First, a transition diagram was created. In this diagram, the row would represent the current state, i, and the column would represent the value of transitioning to the target state, j. For each invalid state, the value of transitioning was recorded as negative infinity. This served the purpose of dissuading the agent from every choosing illegal moves. For every transitional state, the value was recorded as 0, indicating that the move was legal, however. Lastly, the goal states were all recorded with a score of 100. A discount factor of 0.8 was used for all models.

3.3.1 Tower of Hanoi

To initialize the problem, each possible state is created resulting in 3^N possible states with N rings, as visualized in Figure 9. Simultaneously, a general list of possible moves is also generated (e.g. move ring from peg 1 to peg 2). With the general case of rules established and moves established, the program iterates through each move and state according to the internal validity rules that were described in Section 3.2. The results of this process are recorded in a reward table, R. An example of this matrix can be seen in Table 1. In this table, transitional states provide a reward of 0 and the goal state provides a score of 100. With the reward matrix established and states initialized, the model then proceeds to follow the Q-Learning algorithm as described

in Chapter III.

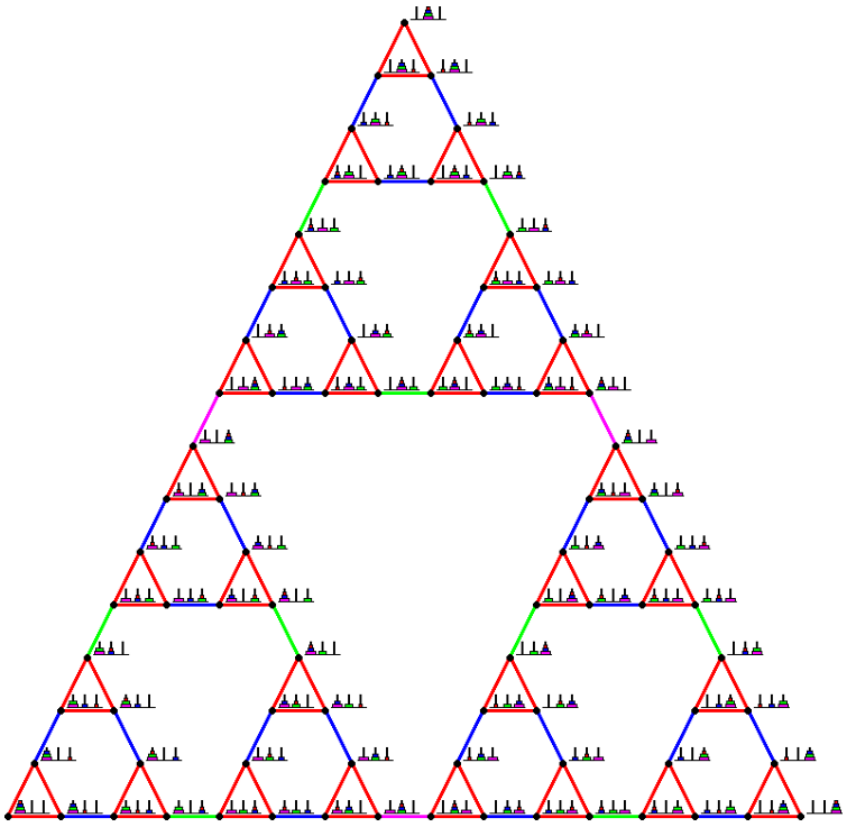


Figure 9: Enumeration of states for Tower of Hanoi game with four rings. The goal state is in the bottom right. Note that all arcs are bidirectional. Image found on Jaap's Puzzle Page[38]

Using R as a foundation, the agent creates its Q Table, which records the expected reward of choosing an action in each state: moving the smallest ring from peg 0 to 1, moving the smallest ring from peg 0 to 2, moving the smallest ring from peg 1 to 2 and etc. When the agent takes an action, the agent transitions from its current state to the state corresponding to the action it chose. For each episode, the agent begins in a random state and chooses a state to transition to randomly. Using the maximum reward possible in the selected state and the value of transitioning to the selected state, the agent updates its Q-Table according to the update equation shown

in step 7 of Algorithm 2. To determine a policy, the best moves for the agent from any state, the agent iterates through each state following the actions with the maximum expected return in the Q Table. As it follows these actions, the sequence of states it moves through are recorded.

Table 1: Rewards Matrix for 2 Ring Tower of Hanoi

State(i,j)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
State 0	-	0.000	0.000	-	-	-	-	-	-
State 1	0.000	-	0.000	0.000	-	-	-	-	-
State 2	0.000	0.000	-	-	0.000	-	-	-	-
State 3	-	0.000	-	-	-	0.000	100.000	-	-
State 4	-	-	0.000	-	-	-	-	0.000	0.000
State 5	-	-	-	0.000	-	-	100.000	-	-
State 6	-	-	-	0.000	-	0.000	-	-	-
State 7	-	-	-	-	0.000	-	-	-	0.000
State 8	-	-	-	-	0.000	-	-	0.000	-

3.3.2 Cargo Loading

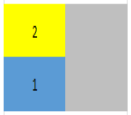
The development of the Q Table for the cargo loading problem followed the same methodology as that for the Tower of Hanoi problem; the delineation of the two exists in the establishment of the state space. As shown in Figure 10, the order of moves has a unique effect on the state space. As a result, the state-space for the model was iterated using brute force methods using the internal validity rules described in section 3.2.1. The occupancy matrix from the environment was used as a reference and all states with legal transitions to and from that matrix were updated as 0 in the R. Finally, all states with N items, where N is the number of cargo to store, were

annotated with a positive value of 100. Notably, the state-space is impacted both by the dimensions of the cargo and the dimensions of the storage space and many alternative optimal solutions exist.

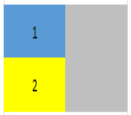
Start	Turn 1	State Enocding	Turn 2	State Encoding	Final Move Encoding	Unique Move Enumeration
			(2,0,0)	9	(0,(1,0,0),(2,0,0))	1
	(1,0,0)	1	(2,0,1)	10	(0,(1,0,0),(2,0,1))	2
			(2,1,0)	11	(0,(1,0,0),(2,1,0))	3
			(2,1,1)	12	(0,(1,0,0),(2,1,1))	4
			(2,0,0)	13	(0,(1,0,1),(2,0,0))	5
	(1,0,1)	2	(2,0,1)	14	(0,(1,0,1),(2,0,1))	6
			(2,1,0)	15	(0,(1,0,1),(2,1,0))	7
			(2,1,1)	16	(0,(1,0,1),(2,1,1))	8
			(2,0,0)	17	(0,(1,1,0),(2,0,0))	9
	(1,1,0)	3	(2,0,1)	18	(0,(1,1,0),(2,0,1))	10
			(2,1,0)	19	(0,(1,1,0),(2,1,0))	11
			(2,1,1)	20	(0,(1,1,0),(2,1,1))	12
	(1,1,1)	4	(2,0,0)	21	(0,(1,1,1),(2,0,0))	13
			(2,0,1)	22	(0,(1,1,1),(2,0,1))	14
			(2,1,0)	23	(0,(1,1,1),(2,1,0))	15
			(2,1,1)	24	(0,(1,1,1),(2,1,1))	16
START STATE						
			(1,0,0)	25	(0,(2,0,0),(1,0,0))	17
	(2,0,0)	5	(1,0,1)	13	(0,(2,0,0),(1,0,1))	18
			(1,1,0)	17	(0,(2,0,0),(1,1,0))	19
			(1,1,1)	21	(0,(2,0,0),(1,1,1))	20
			(1,0,0)	10	(0,(2,0,1),(1,0,0))	21
	(2,0,1)	6	(1,0,1)	26	(0,(2,0,1),(1,0,1))	22
			(1,1,0)	18	(0,(2,0,1),(1,1,0))	23
			(1,1,1)	22	(0,(2,0,1),(1,1,1))	24
			(1,0,0)	11	(0,(2,1,0),(1,0,0))	25
	(2,1,0)	7	(1,0,1)	15	(0,(2,1,0),(1,0,1))	26
			(1,1,0)	27	(0,(2,1,0),(1,1,0))	27
			(1,1,1)	23	(0,(2,1,0),(1,1,1))	28
			(1,0,0)	12	(0,(2,1,1),(1,0,0))	29
	(2,1,1)	8	(1,0,1)	16	(0,(2,1,1),(1,0,1))	30
			(1,1,0)	20	(0,(2,1,1),(1,1,0))	31
			(1,1,1)	28	(0,(2,1,1),(1,1,1))	32

State Visualization for Selected Samples

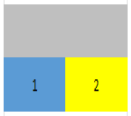
State 9: Unique Move = 1



State 25: Unique Move = 17



State 11: Unique Move = 3



State 11: Unique Move = 25

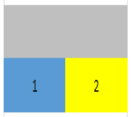


Figure 10: Depiction of possible states in the CL game. Note that this depiction assumes each cargo is a cube with dimensions and weight of one and the storage area is a cube with dimensions of 2. Transitions within the state space may or may not be valid depending on the dimensions of the cargo items. Left: Enumeration of possible moves in simple cargo loading game with two items and storage area with dimensions 2x2x2. Right: Visualization of select states to highlight properties of CL problem. Note, that moves are not necessarily commutative (comparison of states 9 and 25) although some are (comparison of state 11 potential moves)

3.4 Deep Reinforcement Learning Problem Modifications

Unlike the reinforcement learning model, the DRL agent was intended to approximate a generalized Artificial Intelligence (AI). Therefore, it was considered imperative that the same architecture was maintained across problems. However, for training and solving requirements, modifications had to be made for problem initialization, goal attainment and training verification. The same neural network was intended to train on as close to a similar environment as possible. These differences are annotated in the following subsections while the model architecture and training methods are documented in section 3.5.

3.4.1 Modifications for Tower of Hanoi

Tower of Hanoi starts with cargo in the environment state space. Therefore additional initialization logic was created in order to sort the rings by weight and store them in the appropriate location within the environment. The agent was not allowed to store or remove objects. Instead, the agent interacted with the environment through moving rings from one peg to another. The success of the game is declared once all the rings are stored in appropriate x and y index for the storage array, where x and y represent the length and width in the 2 dimensional plane. The reward given upon success is penalized according to how many moves were made before achieving the target result, with a game completed in zero moves receiving a score of 1000. For clarity, in a game with two rings, the maximum possible score would be 997.

3.4.2 Modifications for Cargo Loading

In the cargo loading problem the environment begins absent of any items. The agent must learn which cargo can be placed and what locations they can be placed. This problem of the generalized neural network was the first to be developed and

planned and helped to serve as a first baseline of the neural networks effectiveness. Similar to the ToH architecture, there exists a movement penalty in these games. For the CL problem a maximum score was instead indicated by maximum reward less the number of items to store. Thus, a perfect game for a storage problem of two objects would be 998. Initially, this problem type permitted the agent to store, remove and rotate objects according to the methods described previously, however the action space was scoped to strictly consist of the ability to store and remove items (i.e., no rotation). Additionally, some initial configurations were vetted to ensure the problem was solvable. An example of an unsolvable problem for the agent under these conditions can be seen in Figure 11.

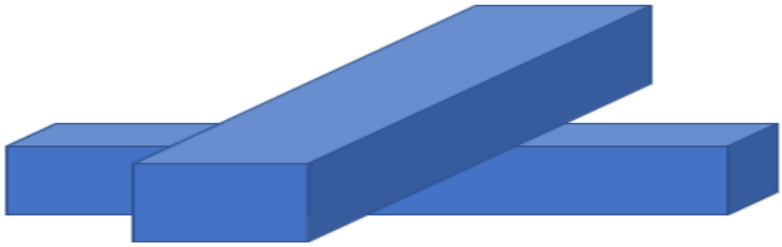


Figure 11: Due to the agent's inability to rotate items, the following cargo would render the problem unsolvable. This is because one cargo would overhang the other violating internal validity logic

3.4.3 Modifications for Knapsack

Unlike the other agents, the knapsack problem does not penalize moves. This environment follows a similar logic to the cargo loading model. However, the weight, length, and width dimensions of all cargo as well as the length and width of the storage space are set to 1. In this manner, the height of the cargo serves to act as what would traditionally be called the weight, and the height of the storage space serves as the carrying capacity. The only variability in the problem is the value of

each specific cargo item and the storage space. Once the storage area fills such that there is insufficient space for any other item, the agent would then be rewarded the value of all items stored.

3.5 Deep Reinforcement Agent Architecture and Training

Initially, the model consisted of three convolutional neural networks which would concatenate their outputs into a back-end model which would then process and output a final observation. The front end models would process the parameter space, occupational matrix and the loading space respectively, and their combined output would result in an action space consisting of all possible actions. Ultimately, this model proved to be too unwieldy and slow to train, and a simplified model was pursued instead. This simplified model consists of 5 dense hidden layers with 200 neurons each interspersed with normalization layers. The number of output neurons for this model is variable and depends on the action space for the problem, as outlined in section 3.4. Additionally, as the input of model depends on the size of the storage space, the inputs for the first layer also depend on the size of the storage space, a cube storage area of dimensions 3 results in an input tensor of $[none, 3, 3, 3]$. Table 2 is an example of the model for a 3x3x3 storage space cargo loading problem with 3 items. Adam was used as the optimizer as it has been shown to enhance convergence in DRL models while the Huber function was used to calculate loss as it has been shown to be less sensitive to outliers [39] [40].

Table 2: Architecture for Deep Reinforcement Learning Model

Layer(Type)	**Output Shape**	**Number of Parameters**
OCC INPUT(InputLayer)	none, 3,3,3	0
OCC NORM 1(Batch Normalization)	none, 3,3,3	12
OCC L1 (Dense)	none, 3,3,3	800
OCC NORM 2(Batch Normalization)	none, 3,3,3	800
OCC L2 (Dense)	none, 3,3,3	40200
OCC NORM 3(Batch Normalization)	none, 3,3,3	800
OCC L3 (Dense)	none, 3,3,3	40200
OCC NORM 4(Batch Normalization)	none, 3,3,3	800
OCC L4 (Dense)	none, 3,3,3	40200
OCC NORM 5(Batch Normalization)	none, 3,3,3	800
OCC L5 (Dense)	none, 3,3,3	800
OCC Flatten (Dense)	none, 1800	0
Output (Dense)	none, 30	54030

For each of these problems the training cycle for the agent proceeds according to the same subroutine. The actual process of training follows an adaptation of Algorithm 2. First, the agent initializes with its own primary neural network and target neural network and a memory length limit. This target neural network initializes as a clone of the primary neural network. Throughout the training, the primary neural network updates once a batch of 32 steps have been taken. While it proceeds through these steps it uses the target network for predictions on future states. Every four steps, the target network updates its weights to those of the primary neural network. For each episode, the network chooses to explore or exploit its knowledge. During

the exploration operation, the model chooses a random action, if the action is invalid, the model will choose another random action until a valid action is presented. This addition is made to enhance convergence as the DRL model will still have a probability to choose an illegal move. However, if the model chooses to exploit its own knowledge and chooses an invalid action, the model will receive a penalty. Otherwise, the agent will complete the move and return the reward in the given state as well as the occupational matrix of this state for training.

The agent would then proceed to attempt to solve the selected problem according to Algorithm 3. In the case in which the algorithm reached a solution state or 10,000 steps were conducted without reaching the target state, the model would terminate the current episode and proceed to the next. Because time was a major limitation in training the deep reinforcement agents, multiple iterations could not be conducted for analysis of average training time or verification of model accuracy. Instead, two statistics were taken. The first was a record of the number of episodes it took the model to reach the optimal answer. The second was a record of the number of runs until the model could optimally complete the problem five times in the row. Five times were chosen because in the event of random chance the model would accomplish this task approximately 4% of the time.

IV. Results and Analysis

This chapter is explains the results of testing on the reinforcement and deep reinforcement learning methods on the problems tested. The first section delineates the results of conventional reinforcement learning and analyzes why the agent failed or succeeded in each case. The second phase discusses the deep reinforcement algorithm and its results, as well as document challenges in its development. In general the Reinforcement Learning (RL) models converged on results quicker than the Deep Reinforcement Learning (DRL) in terms of time when they did converge, however they also required many more training episodes before convergence and were less capable of solving a variety of problems. This is likely due to the fact the reinforcement learning algorithm is recursive in nature. In more complicated state spaces the calculations can become intractable.

4.1 Cargo Loading Results

In the case of the Cargo Loading (CL) problem, the reinforcement learning algorithm completely failed to process the data in the base case: 2 1x1x1 cargo items and a storage space of 3x3x3. While the exact cause of failure is uncertain, there are two major factors that likely contributed. First, the cargo loading problem is one in which many alternatively optimal solutions are possible. As a result, the algorithm may struggle to identify a best action when no such case exists. Secondly, the possible moves in each part of the state space are extensive: from start there exist 8 alternatives to consider and each step can be similarly complicated.

The deep reinforcement learning agent performed surprisingly well in comparison to the RL model. Notably, the deep reinforcement learning agent was able to solve the base case in one episode consistently. Because the Reinforcement Learning method

failed, a useful baseline for comparison could not be established. For confirmation of results, the agent's performance in a higher difficulty problem was tested, and the results are shown in Figure 12.

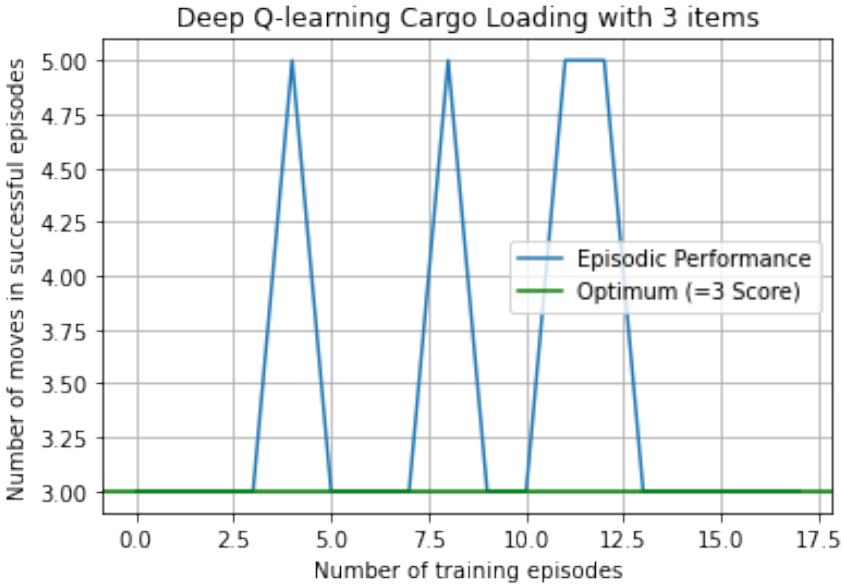


Figure 12: CL problem Deep Q-Learning results with 3 1x1x1 cubes in a 3x3x3 space

4.2 Tower of Hanoi

In the Tower of Hanoi (ToH) problem the reinforcement learning agent performed remarkably well, usually processing within minutes and under 1000 episodes for problems with 3 rings or less. Comparatively, the neural networks performed worse than the reinforcement learning method in the training time required but discovered the optimal move order in a fewer number of episodes. Notably, while the 2 rings and 3 ring models converged in less than an hour for the DRL models, the remaining models would take hours or even days. Much of this delay can be attributed to the Greedy Epsilon Search algorithm. By initially focusing on exploration, the agent is more likely to arrive at an optimal solution however it is less likely to do so quickly. Table 3 summarizes the results of the RL agents, while the following plots show the

average number of moves for the RL agents in dark blue and the standard deviation of the performance in light purple. Note, that due to the large number of episodes in latter problems, a logarithmic scale is used for the x-axis. As mentioned in chapter III, this methodology follows and confirms Kurt Peek's results in the Tower of Hanoi problem.

Table 3: Results for the Tower of Hanoi with a Reinforcement Learning Approach

Number of Rings	**Approximate Number of Episodes to Converge**
2	**300**
3	**900**
4	**6000**
5	**30000**
6	**Did Not Converge**

Figure 13 shows the results of the reinforcement learning algorithm with two discs truncated at 120 episodes for clarity. After every four episodes, 100 runs were conducted from that episodes Q-Table and the number of moves required to complete the problem was recorded for each run. From these runs, the mean and standard deviation of the moves were taken for that particular episode. The reinforcement learning method is shown to converge consistently upon the solution in accordance with Watkins and Dayan's analysis [37].

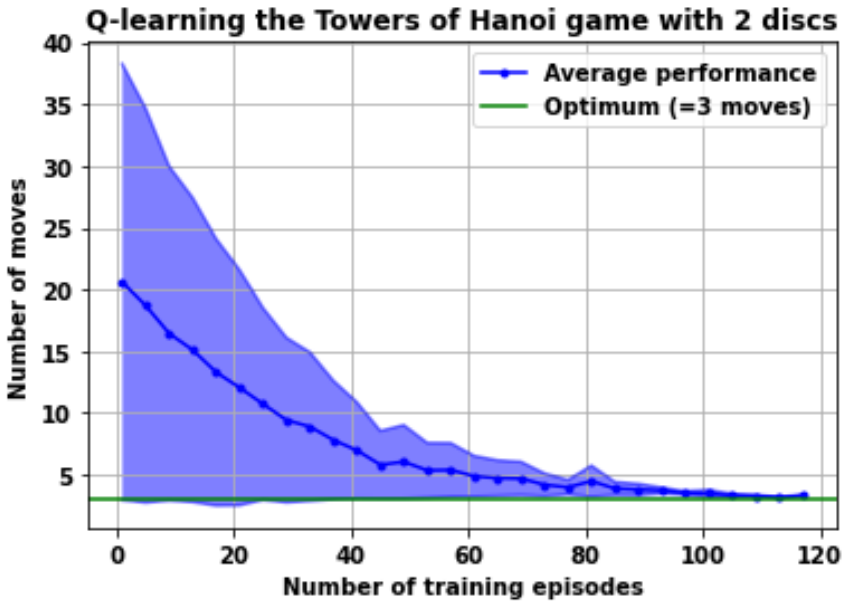


Figure 13: ToH problem with two rings. The purple region represents the standard deviation of the agent's runs while the blue line indicates the average of the agent's runs. While very good results are obtained at approximately 120 episodes, consistently optimal results are not found until at least an additional 150 runs.

It is worth noting that in terms of episodes, the DRL agents seem to be slightly more efficient. While both the reinforcement learning and the deep reinforcement learning agents achieved good results in approximately 80 episodes, only the deep reinforcement learning model was able to achieve the optimal result in under 70 episodes. This implies that the DRL agents may be more efficient with the information obtained. It is interesting to note however, that in both the deep and standard reinforcement learning methods, while the agent does steadily converge, there are moments where the agent's performance decreases temporarily. This is particularly evident in the three ring Tower of Hanoi model, Figure 14.

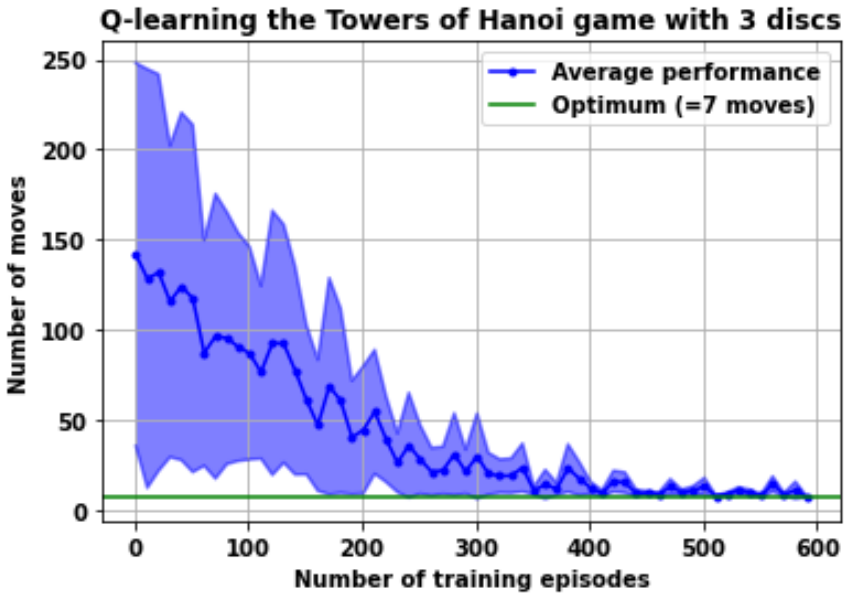


Figure 14: ToH problem Q-learning results with three rings. The purple region represents the standard deviation of the agent's runs while the blue line indicates the average of the agent's runs.

While not a serious concern in the context of these problems, in the grander scheme of AI research, this indicates a broader issue. While human supervision can discern optimal solutions from a set of alternatives, the agent itself must be able to communicate its confidence in its solution. While strides have been made in this regard, such as Bayesian Neural Networks, these methods also require more training time to return results as well [41]. This points to another major issue with DRL agents: they are time intensive.

While the RL agent was not able to provide results past 6 rings, the results it was able to provide were relatively good quality and quick, Figure 15. In the DRL case, the agent was unable to finish training for the four ring variation of the game. In fact, even confidence intervals for the three ring Tower of Hanoi game could not be obtained in the allotted time. This failure was only more pronounced in observing

the five run streak for the DRL agent; the agent was only able to accomplish the most basic one ring Tower of Hanoi game. It is highly probable that the method of conducting exploration in the Deep Q-Learning models contributes to this discrepancy as well.

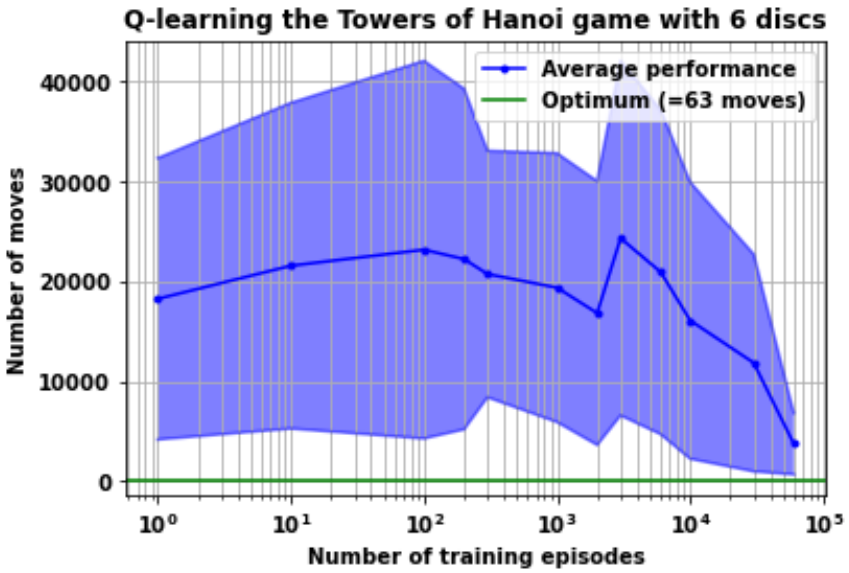


Figure 15: ToH problem Q-Learning results with 6 rings

Under the Epsilon Greedy Method of exploration, while the probability of exploring as opposed to exploiting steadily decreases, it never reaches zero. Therefore, as the number of moves required to complete the task increase, the likelihood of the model completing the task successful per the completion criterion specified decreases. While the RL agents also have random moves, the action space of the DRL is larger to account for the more generalized approach, resulting in a higher likelihood for non-beneficial actions to be selected. In short, this points to a need for artificial intelligence agents to identify and scope their own actions within the environment. The effects of the Epsilon Greedy Search algorithm can be seen in Figure 16. While the agent quickly comes close to optimality, random actions inhibit success until epsilon decays sufficiently. A more intelligent search algorithm could facilitate state-space

exploration.

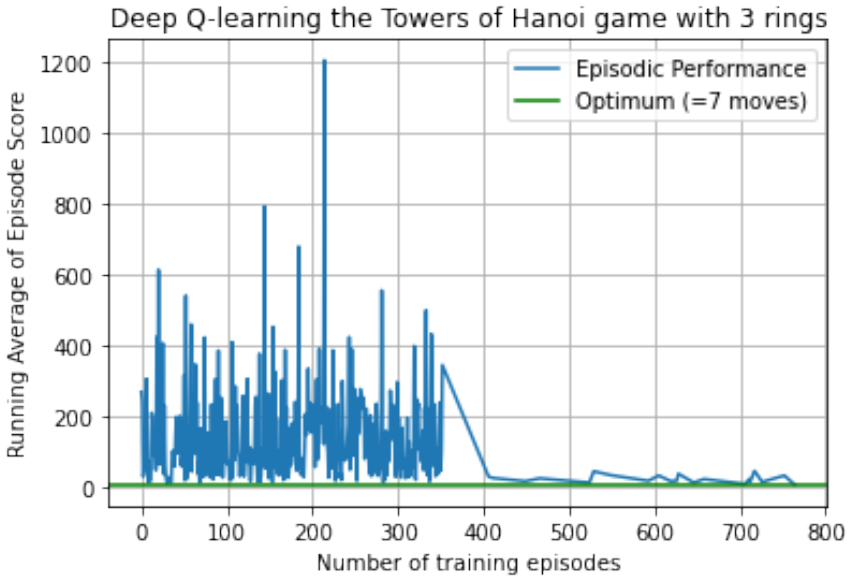


Figure 16: CL problem Deep Q-Learning results with 3 rings and 200 neurons

4.3 Deep Reinforcement Learning Tower of Hanoi: Analysis of Neurons on Effectiveness

An analysis of the effect the number of neurons has on problem solving was inconclusive. Because the models were not able to successfully achieve a five episode optimal streak, first successful run performance was measured for the two ring ToH problem. Three trials were conducted at neuron densities of 100, 200 and 500 per layer. Assume a null hypothesis that number of neurons do not impact episode length.

Table 4: Table of observations for runs until first optimal performance in two ring Tower of Hanoi game

Number of Neurons	**Number of Episodes**
100	**5**
100	**7**
100	**45**
200	**3**
200	**4**
200	**56**
500	**13**
500	**25**
500	**32**

To test for differences in the means, a One-way Analysis of Variance (ANOVA) test was conducted. A One-way ANOVA test is a statistical procedure to evaluate the means of three or more factor levels against the same response. Under the null hypothesis for the test, the means are equal for all factor levels. Under the alternative hypothesis, at least one factor level results in a different mean than the others. Following this test, there was insufficient data to discern the effect of neuron density on episode training time at the 95% confidence level, as shown in Table 5.

Table 5: Table of observations for runs until first optimal performance in two ring Tower of Hanoi game

Source	Sum of Squares	Mean Square Error
Model	**28.2**	**14.1**
Error	**3038.7**	**506.4**
Total	**3066.9**	**-**
P-value	**0.97**	**-**

A visual representation of these points, their means and their variation are shown in Figure 17. It should be noted that while the statistical test can positively identify a difference, it cannot positively identify a lack of difference. With additional sample points, it may be possible to identify a relationship between neurons and episode length that cannot be discerned with limited data.

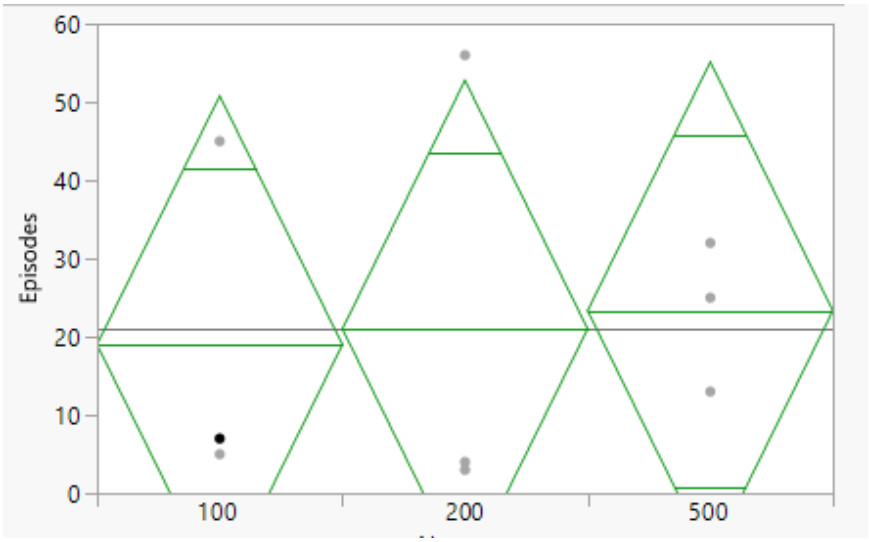


Figure 17: ANOVA of the effects of neuron density on episode length. No discernable difference was detected with a p-value of 0.97.

Following these results, further testing was conducted to determine if additional

neurons would increase the agents ability to converge on the four ring Tower of Hanoi solution. In this case the agent still failed to accomplish the task. This seems to represent a symptom of the curse of dimensionality. As the state-space grows the agent's information requirements increase faster.

4.4 Excursion: Deep Reinforcement Learning Knapsack

The agents did not perform well on this problem set. While agents were able to converge on the correct solution in simple two item problems, when more than two items were required to be evaluated the agent did not converge quickly or accurately. While this would seem to correlate with the Tower of Hanoi results, the root cause is likely different. This agent does not gain reward information for each item stored, but instead gains reward information once all possible items for that episode have been stored. Furthermore, because the agent cannot observe the value of the cargo before placing, it must decouple the final state reward with individual cargo placements. As a result, the model is slow to observe and iterate on options. Furthermore, as the agent cannot perceive the difference in the items ahead of time, what it has learned cannot be readily transferred to other games. As a result, the utility of this method is dubious as evolutionary algorithms have shown better results historically.

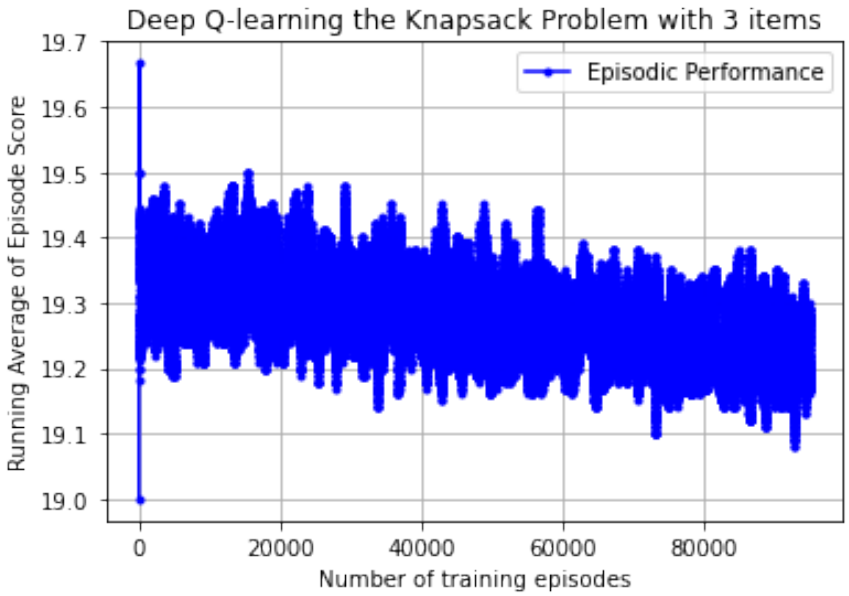


Figure 18: KP problem with Deep Q-Learning. Items had weight and value of (2,20), (1,11), (1,8) with 2 weight units available. Under the premise the agent only receives the reward at the end of operations, the agent failed to optimize the loadings in over 80,000 episodes

V. Conclusions

This research attempts to compare the performance of Deep Reinforcement Learning (DRL) with Reinforcement Learning (RL) methods as an avenue to assess potential obstacles to strong Artificial Intelligence (AI) development. In this assessment, RL agents were used as a proxy for weak AI while DRL were used as a proxy for strong AI. The agents were evaluated on their ability to accomplish two separate tasks and one excursion: the Tower of Hanoi (ToH), the Cargo Loading (CL), and the Knapsack (KP) as an excursion. Results indicate that the DRL agents were able to perform a multiple problems, but were unable to perform as well as specialized algorithms, although additional testing is required for confirmation

5.1 Implications for Generalized Artificial Intelligence

Lack of certainty regarding AI agents' ability to solve problems is a chief issue in non-supervised learning methods. While strides have been made in analyzing neural networks and creating explainable AI, they are naturally black boxes. As a result, AI can perform unexpectedly. Furthermore, simply increasing the power of AI is likely not to result in improved results without improvements in data sampling as shown in Deepmind's AlphaZero [42]. The same research has also reinforced that ensemble methods using tailored programs for pieces of a problem provide general results than singular models. In particular, trial and error methods are likely to be less reliable in higher dimensional spaces or as actions stretch into the infinite horizon.

Similarly, goal based learning methods seem unlikely to generate human-level artificial intelligence. This is because well-defined and measurable goals are inherently limiting, and a human-level artificial intelligence would need to reassess and evaluate its goals. It should be noted that attempts for humans to understand machine intelli-

gence may hinder its development as machine intelligence is divorced from biological and social functions that inform human intelligence [4].

5.2 Future Work

Many aspects of this research can be improved for future work. Neural network architecture was not optimized as a consequence of standardization requirements. Alterations to neuron density and number of layers will likely change results. The utility of convolutional layers is somewhat dubious due to the granularity of the data, however its possible impacts cannot be dismissed. Additional testing with different architecture may indicate better results for some or all models.

Testing on non-rectangular shapes for cargo loading would be of interest for assessment of model strength and practical use. Additionally, the original scope of the model would include using the parameters and loading arrays as observations for the agent. While the inclusion of this data resulted in longer training times than this research could permit, follow on research may be able to achieve better results. Code supporting this functionality can be modified in the appendix. The use of cognitive architecture to assess the learning and function of the model's neurons will likely be of note to future AI researchers.

Actor-critic models may also be adopted for better results. While these models were outside the scope of this research, these methods attempt to navigate some of the issues discussed in chapter IV, specifically issues regarding information quality. Similarly, an approach similar to those of Bayesian Neural Networks could be investigated (i.e. the agent would investigate actions it has the least confidence regarding as opposed to at random).

To further explore the range of problems the deep reinforcement model could solve, tests could be conducted using transfer learning. For instance, the weights of

an agent trained on the ToH could be transferred to an agent for CL or a similar problem. Analysis on the effectiveness of the new agent could be conducted.

5.3 Recommendations

As indicated from the results between the RL and DRL agents in the ToH game, its reasonable to assume a general form of artificial intelligence will not outperform specialized methods. Furthermore, AlphaZero's success in combining search and deep reinforcement learning methods seen to indicate a path for success [42]. Another factor inhibiting generalized AI development may be an inability to properly model an environment. As shown in Cohen's work in 1996, a sterile environment can result in an inability for living organisms to adapt to more generalized conditions, so it may be reasonable to assume a similar obstacle for artificial intelligence [5]. Long term operation of a robot with real world interactions and unspecified goals may prove more fruitful than simulations in a virtual environment. If the goal is solely problem solving, an Artificial Neural Network (ANN) classifier front-end with specialized problem-tailored back-ends may prove more beneficial.

Appendix A. Code

```
!pip install pyyaml h5py  # Required to save models in HDF5 format
import numpy as np
import itertools
import pandas as pd
import matplotlib.pyplot as plt
import keras
import tensorflow as tf
from numpy.core.multiarray import concatenate
from collections import deque
import random
from google.colab import drive
import matplotlib.pyplot as plt
drive.mount('/content/drive')


class Storage:
    """Creates a storage space for cargo loading Deep RL problem or
    for Towers of Hanoi
    Args:
      length: Scalar
      width: Scalar
      height: Scalar


    Return:
        Object representing a box with dimensions according to arguments
    """
```

```
def __init__(self, length = 3, width = 3, height = 3,
packing_list = [], human_player= False):
    self.__human_player = human_player
    self.length = length
    self.width = width
    self.height = height
    self.occupancy = np.zeros(shape=(self.height, self.length, self.width))
    self.loadings = np.zeros(shape=(self.height, self.length, self.width))
    self.packing_list = packing_list
    self.stored_items = []
    #self.__cross_reference = [None]*len(self.packing_list)



def __check_size(self,cargo, xcoord, ycoord, zcoord):
    """ quick cursory check to see if the box can fit.
    """
    if zcoord == None:
        zcoord = 0
    #print("length", cargo.length, xcoord,
    self.length, "width", cargo.width, ycoord,
    self.width, "height", cargo.height, zcoord, self.height)

    if  (cargo.length+xcoord > self.length)
    or (cargo.width+ycoord > self.width) or
    (cargo.height+zcoord > self.height):
```

```
        return(False)
    else:
        return(True)

def __check_level_clear(self,cargo, xcoord,ycoord,
zcoord, SearchClear= True):
    """Checks a plane of the storage box to determine if it is empty
    if SearchClear=True
    and checks to make sure all points are
    filled if SearchClear = False

    """
    if zcoord == None:
       zcoord = 0
    free= True

    if SearchClear is True:
        for x in range(cargo.length):
            for y in range(cargo.width):
                #Check level clear comment 1
                #print(x, y, xcoord, ycoord, zcoord)
                if self.occupancy[zcoord][xcoord+x][ycoord+y]!=0:
                    free = False
                    return(free)
    elif SearchClear is False:
        for x in range(cargo.length):
```

```
            for y in range(cargo.width):
                if self.occupancy[zcoord][xcoord+x][ycoord+y] == 0:
                    free = False
                    return(free)
    return(free)


def __find_free_level(self,cargo,xcoord,ycoord, zcoord):

# find the first level of the block to stack
    """ Finds the first free level in the storage space based on
    the cargo's side and returns this level if there is
    a level with enough space and returns False
    if there is not enough space. At the same time we
    do not want floating boxes, so if there is not
    something below each point of the cargo, we also return False.

    Returns:
        Outputs an intenger representing the first free level
        if there is one available and False if otherwise
    """
    if zcoord == None:
       zcoord = 0

    level = zcoord
    free = False # we do not know if there is a free space yet,
```

```
change once a
floor is cleared successfully

while level < self.height and free is False: #repeat the
process until you either hit the top of the box or find a
clear floor
    if self.__check_level_clear(cargo,xcoord,ycoord,
    level, True) is True:
        if level >0:
        #if you find a clear floor and you are
        above the first floor, you need to check if
        something is underneath it so it does not float
            if self.__check_level_clear(cargo,xcoord,ycoord,
            level-1, False) is True:
                free = True
                return(level)
            else:
                if  self.__human_player == True:
                    print("Cargo", cargo, " does not stack properly")
                return(free)
        else:
            free = True
            if level == None:
              level = 0
            return(level)
    else:
```

```
            level += 1


def __check_collision(self, cargo, xcoord, ycoord, zcoord):
    """Function to ensure that the cargo does not clip
    into other packages

    Returns:
        Returns True if there is enough space for the object
        in the given coordinates and False otherwise.
    """
    if zcoord == None:
       zcoord = 0

    free = True
    for z in range(cargo.height):
        if self.__check_level_clear(cargo, xcoord,ycoord,
        zcoord+z, True) != True:
            free = False
            return(free)
    return (free)

def __check_weights(self, cargo, xcoord, ycoord, zcoord):
    """Function to make sure heavier cargo are not placed
    onto lighter cargo
```

```
    Returns:
        False if the new cargo is heavier than everything below it.
    """
    if zcoord == None:
       zcoord = 0

    cargo_weight = cargo.weight
    cargo_length = cargo.length
    cargo_width = cargo.width

    for x in range(cargo_length):
        for y in range(cargo_width):
            if self.loadings[zcoord-1][xcoord+x][ycoord+y]
            < cargo_weight:
                return(False)
    return(True)

def __check_cargo_fits(self,cargo,xcoord,ycoord):
    """Checks if the cargo fits

    Returns:
        False (Boolean) if cargo does not fit,
        Tuple (Boolean, Scalar) True and a scalar representing
        the first free level otherwise.
    """
```

```
    if (cargo.length+xcoord > self.length) or (cargo.width+ycoord
    > self.width) or (cargo.height > self.height):
        return(False)
    else:
        free_level = self.__find_free_level(cargo,xcoord,ycoord, 0)
        if free_level is False:
            return(False)
        elif self.__check_size(cargo, xcoord, ycoord,
        free_level) == False:
            return(False)
        elif self.__check_collision(cargo, xcoord, ycoord,
        free_level) is False:
            return(False)
        else:
            return((True, free_level))


def __add_cargo_occupancy(self, cargo_to_add_index,
xcoord,ycoord, zcoord):
    if zcoord == None:
       zcoord = 0
    cargo = self.packing_list[cargo_to_add_index]
    for x in range(cargo.length):
      for y in range(cargo.width):
          for z in range(cargo.height):
              self.occupancy[zcoord+z][xcoord+x][ycoord+y]=
              cargo_to_add_index+1
```

```
def __add_cargo_loadings(self, cargo_to_add_index,
xcoord,ycoord, zcoord=0):
    if zcoord == None:
       zcoord = 0

    cargo = self.packing_list[cargo_to_add_index]
    cargo_weight = self.packing_list[cargo_to_add_index].weight
    for x in range(cargo.length):
      for y in range(cargo.width):
          for z in range(cargo.height):
              self.loadings[zcoord+z][xcoord+x][ycoord+y] = cargo_weight



def store(self, cargo_to_store_index, xcoord, ycoord):
    cargo= self.packing_list[cargo_to_store_index]
    if cargo.stored is True:
        if self.__human_player == True:
            return("Cargo", cargo_to_store_index, " already stored")
        else:
            return(None)
    else:
        placement = self.__check_cargo_fits(cargo,xcoord,ycoord)
        if placement is False:
            if  self.__human_player == True:
                return("Cargo", cargo_to_store_index, " Too Large ")
```

```
        else:
            return(None)
    elif placement[1] == 0:
        self.__add_cargo_occupancy(cargo_to_store_index,
        xcoord, ycoord, placement[1])
        self.__add_cargo_loadings(cargo_to_store_index,
        xcoord, ycoord, placement[1])
        self.stored_items.append((cargo_to_store_index,
        xcoord, ycoord,placement[1]))
        cargo.load()


    else:
        if self.__check_weights(cargo, xcoord, ycoord,
        placement[1])is False:
            if  self.__human_player == True:
                return("Cargo", cargo_to_store_index, " Too Heavy")
            else:
                return(None)
        else:
            self.__add_cargo_occupancy(cargo_to_store_index,
            xcoord,ycoord, placement[1])
            self.__add_cargo_loadings(cargo_to_store_index,
            xcoord,ycoord, placement[1])
            self.stored_items.append((cargo_to_store_index,
            xcoord, ycoord,placement[1]))
            cargo.load()
```

```
def remove_cargo(self, cargo_to_remove_index):
    cargo = self.packing_list[cargo_to_remove_index]
    if cargo.stored is False:
        if self.__human_player == True:
            return("Cargo", cargo_to_remove_index, " not in storage")
        else:
            return(None)
    else:
        for i in range(len(self.stored_items)):
            if self.stored_items[i][0]==cargo_to_remove_index:
                row_record = i
                xcoord = self.stored_items[row_record][1]
                ycoord = self.stored_items[row_record][2]
                zcoord = self.stored_items[row_record][3]


    if zcoord == self.height-1:
        for x in range(cargo.length):
            for y in range(cargo.width):
                for z in range(cargo.height):
                    self.occupancy[zcoord+z][xcoord+x][ycoord+y] = 0
                    self.loadings[zcoord+z][xcoord+x][ycoord+y] = 0
        cargo.unload()
        del self.stored_items[row_record]


    else:
```

```
if (cargo.height+zcoord < self.height):
    if self.__check_level_clear(cargo, xcoord,ycoord,
    zcoord+1, SearchClear=True) is False:
        if self.__human_player == True:
            return("Cargo", cargo_to_remove_index, "
            is in the middle of a stack, clear
            cargo on top before removal")
        else:
            return(None)
    else:
        for x in range(cargo.length):
            for y in range(cargo.width):
                for z in range(cargo.height):
                    self.occupancy[zcoord+z][xcoord+x]
                    [ycoord+y] = 0
                    self.loadings[zcoord+z][xcoord+x]
                    [ycoord+y] = 0
        cargo.unload()
        del self.stored_items[row_record]
else:
    for x in range(cargo.length):
        for y in range(cargo.width):
            for z in range(cargo.height):
                self.occupancy[zcoord+z][xcoord+x]
                [ycoord+y] = 0
                self.loadings[zcoord+z][xcoord+x]
```

```
                            [ycoord+y] = 0
                cargo.unload()
                del self.stored_items[row_record]

    def move_cargo(self, cargo_to_move_index, xcoord, ycoord):
        cargo = self.packing_list[cargo_to_move_index]
        if cargo.stored is False:
            if  self.__human_player == True:
                return("Cargo", cargo_to_move_index, " not in storage")
            else:
                return(None)
        else:
            for i in range(len(self.stored_items)):
                if self.stored_items[i][0]==cargo_to_move_index:
                    row_record = i
                    x_loc = self.stored_items[row_record][1]
                    y_loc = self.stored_items[row_record][2]
                    z_loc = self.stored_items[row_record][3]

                    break
        self.remove_cargo(cargo_to_move_index)
        self.store(cargo_to_move_index, xcoord,ycoord)

        if cargo.stored is False: self.store(cargo_to_move_index, x_loc,y_loc)

class Cargo:
```

```
def __init__(self,name = None, length = 2, width = 2,
height = 1, weight = 1, value= 0):
    self.stored = False
    self.length = length
    self.width = width
    self.height = height
    self._name = name
    self.weight = weight
    self.value = value

@property
def name(self):
    return self._name
@name.setter
def name(self, name):
    self._name = name

def rotate_xy(self):
    if self.stored == False:
      tmp = self.length
      self.length= self.width
      self.width = tmp
      tmp = None

def rotate_xz(self):
    if self.stored == False:
```

```
            tmp = self.length
            self.length = self.height
            self.height = tmp
            tmp = None

    def rotate_yz(self):
        if self.stored == False:
            tmp = self.width
            self.width = self.height
            self.height = tmp
            tmp = None

    def load(self):
        self.stored = True
    def unload(self):
        self.stored = False

class Environment:
  def __record_initial_cargo_orientations(self):
    initial_cargo_orientations = []
    Storage = self.Storage_Area
    for i in range(len(Storage.packing_list)):
      initial_cargo_orientations.append((Storage.packing_list[i].length,
      Storage.packing_list[i].width, Storage.packing_list[i].height,
      Storage.packing_list[i].stored))
```

```
  return(initial_cargo_orientations)


def initialize_items(self):
  packing_list = self.Storage_Area.packing_list
  if self.ProblemType ==2:
    items_to_initialize = len(packing_list)
    initialized_items = 0


    while (initialized_items < items_to_initialize):
      max_weight = 0
      max_weight_index = 0
      for i in range(len(packing_list)):
        if packing_list[i].stored == False and
        packing_list[i].weight>=max_weight:
          max_weight =packing_list[i].weight
          max_weight_index = i
      self.Storage_Area.store(max_weight_index,0,0)
      initialized_items+=1


def update_environment(self):
  packing_status = []
  cargo_x = []
  cargo_y = []
  cargo_z = []
  packing_weight = []
  packing_value =[]
```

```
x = self.Storage_Area.length
y = self.Storage_Area.width
z = self.Storage_Area.height


for i in range(len(self.Storage_Area.packing_list)):
  packing_status.append(self.Storage_Area.packing_list[i].stored)
  cargo_x.append(self.Storage_Area.packing_list[i].length)
  cargo_y.append(self.Storage_Area.packing_list[i].width)
  cargo_z.append(self.Storage_Area.packing_list[i].height)
  packing_weight.append(self.Storage_Area.packing_list[i].weight)
  packing_value.append(self.Storage_Area.packing_list[i].value)

cargo_x = np.array(cargo_x, dtype = "object")
cargo_y = np.array(cargo_y, dtype = "object")
cargo_z = np.array(cargo_z, dtype = "object")
cargo_weight = np.array(packing_weight, dtype = "object")
cargo_value = np.array(packing_value, dtype = "object")
cargo_status = np.array(packing_status, dtype = "object")
storage_occ = np.array(self.Storage_Area.occupancy, dtype = "object")
storage_load = np.array(self.Storage_Area.loadings, dtype = "object")

self.PARAM_STATUS = np.stack([cargo_x,cargo_y,cargo_z,
cargo_weight, cargo_value, cargo_status])
self.PARAM_SHAPE = self.PARAM_STATUS.shape
```

```
    self.ENVIRONMENT_STATUS = (self.PARAM_STATUS, storage_occ, storage_load)
    self.ENVIRONMENT_SHAPE = (self.PARAM_SHAPE, storage_occ.shape,
    storage_load.shape)

def __init__(self, Storage, ProblemType=1):
    self.Storage_Area = Storage
    self.ProblemType = ProblemType
    # 1 is for CargoLoading, 2 is for ToH, 3 is for Knapsack
    self.__initial_cargo_orientations =
    self.__record_initial_cargo_orientations()
    self.__select_item_len = len(self.Storage_Area.packing_list)
    #noting the number ofitems is useful for various functions

    self.game_over = False # once all cargo are loaded force game over.

    self.move_penalty = 0
    # we want to penalize excess moves that do nothing,
    #but not immediately in the training process
    #self.completion_reward = 1000 # large reward for
    #winning the game in the case of ToH and CargoLoading

    self.moves = 0 # initial moves
    self.score = 0 # the current score
    self.penalties_received = 0
    self.punishment = -100000
```

```
if self.ProblemType==2:
  self.initialize_items()
self.update_environment()

self.__store_actions = []
self.__remove_actions = []
self.__rotate_xy = []
self.__rotate_xz = []
self.__rotate_yz = []

#we want to reecord all of the possible actions
for the action space for future reference by the agent
for i in range(len(self.Storage_Area.packing_list)):
  self.__remove_actions.append(i)
  self.__rotate_xy.append(i)
  self.__rotate_xz.append(i)
  self.__rotate_yz.append(i)
  for x in range(self.Storage_Area.length):
    for y in range(self.Storage_Area.width):
        self.__store_actions.append((i,x,y))
self.__move_actions = self.__store_actions.copy()

if self.ProblemType==1: #CargoLoading Problem
  self.ACTION_SPACE = self.__store_actions
  +self.__remove_actions
```

```
    #self.__rotate_xy+self.__rotate_xz+self.__rotate_yz
    self.ACTION_LEN = len(self.ACTION_SPACE)
    #len(self.__store_actions)
  elif self.ProblemType==2: #Tower of Hanoi
    self.ACTION_SPACE = self.__move_actions
    self.ACTION_LEN = len(self.ACTION_SPACE)
  else: #Knapsack Problem
    self.ACTION_SPACE = self.__store_actions
    self.ACTION_LEN = len(self.ACTION_SPACE)


def restart(self):
  self.score = 1
  self.moves = 0
  self.penalties_received = 0
  self.game_over = False

  for i in range(len(self.Storage_Area.packing_list)):
      self.Storage_Area.packing_list[i].length
      = self.__initial_cargo_orientations[i][0]
      self.Storage_Area.packing_list[i].width
      = self.__initial_cargo_orientations[i][1]
      self.Storage_Area.packing_list[i].height
      = self.__initial_cargo_orientations[i][2]
      self.Storage_Area.packing_list[i].stored
      = self.__initial_cargo_orientations[i][3]
```

```
  for i in range(self.Storage_Area.length):
      for j in range(self.Storage_Area.width):
          for k in range(self.Storage_Area.height):
              self.Storage_Area.occupancy[k][i][j] = 0
              self.Storage_Area.loadings[k][i][j] = 0

  self.Storage_Area.stored_items = []
  self.initialize_items()#call initializing function somewhere in here
  self.update_environment()

def step(self, action_index, Exploit=True):
  Exploit = Exploit
  #because in training there is a nonzero chance that
  #the agent attempts to make an impossible move,
  #we keep this variable to categorize the
  #nature of the move, exploitation or exploration
  value_tracker = 0
  initial_state = np.copy(self.ENVIRONMENT_STATUS[1])
  self.moves +=1

  ### Logic checking for games  moves###
  if self.ProblemType==1:
    if action_index<len(self.__store_actions):
      self.Storage_Area.store(self.ACTION_SPACE[action_index][0],
      self.ACTION_SPACE[action_index][1],
```

```
    self.ACTION_SPACE[action_index][2])
  else:
    self.Storage_Area.remove_cargo(self.ACTION_SPACE[action_index])
elif self.ProblemType==2:
   self.Storage_Area.move_cargo(self.ACTION_SPACE[action_index][0],
   self.ACTION_SPACE[action_index][1],
   self.ACTION_SPACE[action_index][2])
elif self.ProblemType==3:
   self.Storage_Area.store(self.ACTION_SPACE[action_index][0],
   self.ACTION_SPACE[action_index][1],
   self.ACTION_SPACE[action_index][2])
else:
    return("Undefined problem type")

### Logic to penalize or cancel out moves that do not
do anything for training purposes###
self.update_environment()
if np.array_equal(initial_state, self.ENVIRONMENT_STATUS[1]):
  if Exploit == True:
   self.penalties_received+=1
   self.game_over = True
  else:
    self.moves-=1
    return(False)
value_tracker = self.moves*self.move_penalty +
self.penalties_received*self.punishment
```

```
### Coding to check for win conditions for the various games
if self.ProblemType==1: #cargo loading
  total_stored = 0
  for i in range(self.__select_item_len):
      if self.Storage_Area.packing_list[i].stored == True:
        total_stored += 1
  if total_stored == self.__select_item_len:
      value_tracker += self.completion_reward
      self.game_over = True
  self.score = value_tracker
  return(self.ENVIRONMENT_STATUS, self.score, self.game_over)

elif self.ProblemType==2: #Tower of Hanoi
  xgoal = self.Storage_Area.occupancy.shape[1]-1
  ygoal = self.Storage_Area.occupancy.shape[2]-1
  correct_location = 0
  for i in range(self.__select_item_len):
      if (self.Storage_Area.stored_items[i][1] == xgoal) and
      (self.Storage_Area.stored_items[i][2] == ygoal):
        correct_location += 1
  if correct_location == self.__select_item_len:
    value_tracker+=self.completion_reward
    self.game_over = True
  self.score = value_tracker
  return(self.ENVIRONMENT_STATUS, self.score, self.game_over)
```

```
    elif self.ProblemType==3:
        value_tracker = 0
        self.game_over = True
        free_space = 0
        for i in self.Storage_Area.occupancy:
          if i == 0:
            free_space+=1
        for i in range(len(self.Storage_Area.packing_list)):
          if self.Storage_Area.packing_list[i].stored ==False and
          self.Storage_Area.packing_list[i].height<= free_space:
              self.game_over = False
        if self.game_over == True:
          for i in range(len(self.Storage_Area.packing_list)):
            if self.Storage_Area.packing_list[i].stored == True:
              value_tracker+= self.Storage_Area.packing_list[i].value
        self.score = value_tracker
        return(self.ENVIRONMENT_STATUS, self.score, self.game_over)
```

Note that training code was modified from Keras documentation [**?**].

```
class DQNAgent:
  def __init__(self,Environment):
    self.env = Environment

    #Main Model
```

```
    self.model = self.create_model()

    #Target Model
    self.target_model = self.create_model()
    self.target_model.set_weights(self.model.get_weights())

    #Details for plotting performance
    self.running_record =[]

def create_model(self):
    # Create Inputs for Parameters
    env = self.env

    # Create Inputs for Occupancy and Weight Spatial Array
    OCC_INPUT = keras.layers.Input(shape =
    env.ENVIRONMENT_SHAPE[1],name = "OCC_INPUT") # the occupational array
    OCC_NORM_1 = keras.layers.BatchNormalization(name = "OCC_NORM_1")
    (OCC_INPUT)
    OCC_L1 = keras.layers.Dense(200,activation = "relu",name = "OCC_L1")
    (OCC_NORM_1)
    OCC_NORM_2 = keras.layers.BatchNormalization(name = "OCC_NORM_2")
    (OCC_L1)
    OCC_L2 = keras.layers.Dense(200,activation = "relu",name = "OCC_L2")
    (OCC_NORM_2)
    OCC_NORM_3 = keras.layers.BatchNormalization(name = "OCC_NORM_3")
    (OCC_L2)
```

```
  OCC_L3 = keras.layers.Dense(200,activation = "relu",name = "OCC_L3")
  (OCC_NORM_3)
  OCC_NORM_4 = keras.layers.BatchNormalization(name = "OCC_NORM_4")
  (OCC_L3)
  OCC_L4 = keras.layers.Dense(200,activation = "relu",name = "OCC_L4")
  (OCC_NORM_4)
  OCC_NORM_5 = keras.layers.BatchNormalization(name = "OCC_NORM_5")
  (OCC_L4)
  OCC_L5 = keras.layers.Dense(200,activation = "relu",name = "OCC_L5")
  (OCC_NORM_5)
  OCC_NORM_5 = keras.layers.BatchNormalization(name = "OCC_NORM_5")
  (OCC_L5)
  OCC_L6 = keras.layers.Flatten(name = "OCC_FLATTEN")(OCC_L5)
  OCC_L7 = keras.layers.Dense(env.ACTION_LEN, activation = "softmax")
  (OCC_L6)# additional layer added after eliminating the loading section
  OCC_MODEL = keras.Model(inputs = OCC_INPUT, outputs = OCC_L7)
  return(OCC_MODEL)

def plot_results(self,block=True):
  episodes = []
  episode_reward = []
  episode_moves = []
  N= len(self.env.Storage_Area.packing_list)

  for i in range(len(self.running_record)):
    if self.running_record[i][3] == True:
```

```
    episodes.append(self.running_record[i][0])
    episode_reward.append(self.running_record[i][1])
    episode_moves.append(self.running_record[i][2])


fig = plt.figure()
plt.loglog(episodes, episode_moves,label='Episodic Performance')
#plt.semilogx(episodes, upper_reward_std, 'b', alpha=0.5)
#plt.semilogx(episodes, lower_reward_std, 'b', alpha=0.5)
#plt.fill_between(episodes, lower_reward_std,
upper_reward_std, facecolor='blue', alpha=0.5)

if self.env.ProblemType ==1:
  optimum_moves = N
  plt.axhline(y=optimum_moves, color='g', label='Optimum
  (=%s Score)' % optimum_moves)
  plt.xlabel('Number of training episodes')
  plt.ylabel('Number of moves in successful episodes')
  plt.grid('on', which='both')
  plt.title('Deep Q-learning Cargo Loading with %s items' % N)
  handles, labels = plt.gca().get_legend_handles_labels()
  plt.legend(handles, labels)
  plt.show(block=block)
elif self.env.ProblemType ==2:
  optimum_moves = 2**N - 1
  plt.axhline(y=optimum_moves, color='g', label='Optimum
```

```
    (=%s moves)' % optimum_moves)
    plt.xlabel('Number of training episodes')
    plt.ylabel('Running Average of Episode Score')
    plt.grid('on', which='both')
    plt.title('Deep Q-learning the Towers of Hanoi game with %s rings' % N)
    handles, labels = plt.gca().get_legend_handles_labels()
    plt.legend(handles, labels)
    plt.show(block=block)
  elif self.env.ProblemType ==3:
    plt.xlabel('Number of training episodes')
    plt.ylabel('Running Average of Episode Score')
    plt.grid('on', which='both')
    plt.title('Deep Q-learning the Knapsack Problem with %s items' % N)
    handles, labels = plt.gca().get_legend_handles_labels()
    plt.legend(handles, labels)
    plt.show(block=block)
  else:
    return("Problem Type not defined")

  return(plt)



def train(self, streak_crit=5, plot=False, seed= None,
model_name=None, gamma=0.99, learning_rate=0.00025, batch_size=32,
max_steps_per_episode=10_000, epsilon_random_frames=50_000,
epsilon_greedy_frames=1_000_000,max_memory_length=100_000,
```

```
update_after_actions=4, update_target_network=100_000,
solved_criterion=950):
  N= len(self.env.Storage_Area.packing_list)
  streak = 0

  if self.env.ProblemType==1:
    move_crit = N
  elif self.env.ProblemType==2:
    move_crit = 2**N-1
  else:
    move_crit = N

  optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
  # original lr=0.00025 clipnorm=1.0
  plot = plot
  # Configuration paramaters for the whole setup
  gamma = gamma # Discount factor for past rewards
  epsilon = 1.0  # Epsilon greedy parameter
  epsilon_min = 0.01  # Minimum epsilon greedy parameter (originally 0.1)
  epsilon_max = 1.0  # Maximum epsilon greedy parameter
  epsilon_interval = (
      epsilon_max - epsilon_min
  )  # Rate at which to reduce chance of random action being taken
  batch_size = batch_size  # Size of batch taken from replay buffer
  (originally 32)
  max_steps_per_episode = max_steps_per_episode #originally 10_000
```

```
if seed != None:
  random.seed(seed)
# Experience replay buffers
action_history = []
occ_history = []
occ_next_history = []
action_next_history = []
rewards_history = []
done_history = []
episode_reward_history = []
episode_move_count_history = []

#Data for plotting performancegg
running_reward = 0
episode_count = 0
frame_count = 0
# Number of frames to take random action and observe output
epsilon_random_frames = epsilon_random_frames
# Number of frames for exploration (originally 1_000_000)
epsilon_greedy_frames = epsilon_greedy_frames
# Maximum replay length
# Note: The Deepmind paper suggests 1_000_000 however this
#causes memory issues (originally 100_000)
max_memory_length = max_memory_length
# Train the model after 4 actions
```

```
update_after_actions = update_after_actions
# How often to update the target network
update_target_network = update_target_network
# Using huber loss for stability
loss_function = keras.losses.Huber()

while True:  # Run until solved
    self.env.restart()
    #convert current states into tensor usable one. note that
    #because our states
    are 3 different inputs, the data needs to be split
    occ = self.env.ENVIRONMENT_STATUS[1].astype('float')
    occ = tf.convert_to_tensor(occ)
    occ_state = tf.expand_dims(occ, 0)
    episode_reward = 0

    for timestep in range(1, max_steps_per_episode):
        frame_count += 1
       # Use epsilon-greedy for exploration
        if frame_count < epsilon_random_frames or epsilon
        > np.random.rand(1)[0]:
            # Take random action
            action = (np.random.choice(self.env.ACTION_LEN), False)
        else:
            # Predict action Q-values from environment state,
            #if the state is to small to run converts the stack
```

```
    # to a readable format
    action_probs = self.model(occ_state , training=False)
    # Take best action
    action = (tf.argmax(action_probs[0]).numpy(), True)

# Decay probability of taking random action
epsilon -= epsilon_interval / epsilon_greedy_frames
epsilon = max(epsilon, epsilon_min)

#Attempt to eliminate the agent from trying to
#go to an impossible state
# now if the agent chooses an impossible act it will
#be penalized, however
# if the agent instead is forced into a random state
#it will reroll the
#actions until a valid one is chosen
step = self.env.step(action[0],action[1])
if step == False:
  while step == False:
    action = (np.random.choice(self.env.ACTION_LEN), False)
    step = self.env.step(action[0],action[1])
action = action[0]

# Apply the sampled action in our environment
state_next_raw = step[0]
reward = step[1]
```

```
game_over = step[2]
occ_next = state_next_raw[1].astype('float')
occ_next = tf.convert_to_tensor(occ_next)
occ_next = tf.expand_dims(occ_next,0)
episode_reward += reward

# Save actions and states in replay buffer
action_history.append(action)
occ_history.append(occ_state)

action_next_history.append(action)
occ_next_history.append(occ_next)

done_history.append(game_over)
rewards_history.append(reward)
occ_state = occ_next

# Updates  target netwrok after a target number of actions once
the history reaches a certain size
if frame_count % update_after_actions == 0 and
len(done_history) > batch_size:

    # Get indices of samples for replay buffers
    indices = np.random.choice(range(len(done_history)),
    size=batch_size)
```

```
# Using list comprehension to sample from replay buffer
occ_sample = [occ_history[i] for i in indices]
occ_sample = tf.stack(occ_sample)
occ_sample = tf.reshape(occ_sample, (batch_size,
occ_state.shape[1],
occ_state.shape[2], occ_state.shape[3]))
occ_next_sample = [occ_next_history[i] for i in indices]
occ_next_sample = tf.stack(occ_next_sample)
occ_next_sample = tf.reshape(occ_next_sample,(batch_size,
occ_next.shape[1], occ_next.shape[2], occ_next.shape[3]))

rewards_sample = [rewards_history[i] for i in indices]
action_sample = [action_history[i] for i in indices]
done_sample = tf.convert_to_tensor(
    [float(done_history[i]) for i in indices]
)

# Build the updated Q-values for the sampled future states
# Use the target model for stability
future_rewards = self.target_model.predict(occ_next_sample)
# Q value = reward + discount factor * expected future reward
updated_q_values = rewards_sample + gamma *
tf.reduce_max(future_rewards, axis=1)

# If final frame set the last value to -1
updated_q_values = updated_q_values * (1 - done_sample)
```

```
        - done_sample

        # Create a mask so we only calculate loss on the
        #updated Q-values
        masks = tf.one_hot(action_sample, self.env.ACTION_LEN)

        with tf.GradientTape() as tape:
            # Train the model on the states and
            #updated Q-values
            q_values = self.model(occ_sample)

            # Apply the masks to the Q-values to get the Q-value
            for action taken
            q_action = tf.reduce_sum(tf.multiply(q_values,
            masks), axis=1)
            # Calculate loss between new Q-value and old Q-value
            loss = loss_function(updated_q_values, q_action)

        # Backpropagation
        grads = tape.gradient(loss,
        self.model.trainable_variables)
        optimizer.apply_gradients(zip(grads,
        self.model.trainable_variables))

    if frame_count % update_target_network == 0:
        # update the the target network with new weights
```

```
            self.target_model.set_weights(self.model.get_weights())
            self.target_model.save_weights((model_name+
            "_target_model.h5"),
            overwrite=True, save_format="h5", options=None)
            template = "running reward: {:.2f} at episode {},
            frame count {}"
            print(template.format(running_reward, episode_count,
            frame_count))



        # Limit the state and reward history
        if len(rewards_history) > max_memory_length:
            del rewards_history[:1]
            del occ_history[:1]
            del occ_next_history[:1]
            del action_history[:1]
            del done_history[:1]

        if game_over:
            break

    # Update running reward to check condition for solving
    episode_reward_history.append(episode_reward)
    if len(episode_reward_history) > 100:
        del episode_reward_history[:1]
    running_reward = np.mean(episode_reward_history)
```

```
        success = True if reward >= 1000  else False
        self.running_record.append((episode_count, episode_reward,
        timestep, success))
        episode_count += 1

        if episode_reward >= solved_criterion and move_crit>= timestep:
          streak+=1
        else: streak=0

        if streak >= streak_crit:
        # Condition to consider the task solved
            print("Solved at episode {}!".format(episode_count))
            if plot==True:
                  self.plot_results()
            break
```

Note that this code was modified from Kurt Peek's original code [36].

```
def plot_results(agent_choice, block=True):
    agent = agent_choice
    episodes = []
    episode_reward = []
    episode_moves =[]
    N= len(agent.env.Storage_Area.packing_list)

    for i in range(len(agent.running_record)):
      if agent.running_record[i][3] == True:
```

```
    episodes.append(agent.running_record[i][0])
    episode_reward.append(agent.running_record[i][1])
    episode_moves.append(agent.running_record[i][2])


fig = plt.figure()
plt.plot(episodes, episode_moves,label='Episodic Performance')
#plt.semilogx(episodes, upper_reward_std, 'b', alpha=0.5)
#plt.semilogx(episodes, lower_reward_std, 'b', alpha=0.5)
#plt.fill_between(episodes, lower_reward_std, upper_reward_std,
facecolor='blue', alpha=0.5)

if agent.env.ProblemType ==1:
  optimum_moves = N
  plt.axhline(y=optimum_moves, color='g', label='Optimum (=%s Score)' %
  optimum_moves)
  plt.xlabel('Number of training episodes')
  plt.ylabel('Number of moves in successful episodes')
  plt.grid('on', which='both')
  plt.title('Deep Q-learning Cargo Loading with %s items' % N)
  handles, labels = plt.gca().get_legend_handles_labels()
  plt.legend(handles, labels)
  plt.show(block=block)
elif agent.env.ProblemType ==2:
  optimum_moves = 2**N - 1
  plt.axhline(y=optimum_moves, color='g', label='Optimum
```

```
      (=%s moves)' % optimum_moves)
      plt.xlabel('Number of training episodes')
      plt.ylabel('Running Average of Episode Score')
      plt.grid('on', which='both')
      plt.title('Deep Q-learning the Towers of Hanoi game
      with %s rings' % N)
      handles, labels = plt.gca().get_legend_handles_labels()
      plt.legend(handles, labels)
      plt.show(block=block)
    elif agent.env.ProblemType ==3:
      plt.xlabel('Number of training episodes')
      plt.ylabel('Running Average of Episode Score')
      plt.grid('on', which='both')
      plt.title('Deep Q-learning the Knapsack Problem
      with %s items' % N)
      handles, labels = plt.gca().get_legend_handles_labels()
      plt.legend(handles, labels)
      plt.show(block=block)
    else:
      return("Problem Type not defined")


    return(plt)
```

Code from Kurt Peek's Tower of Hanoi project were used with some modifications for other problem sets [36]

```
''' Reinforcement learning of the Towers of Hanoi game.
Reference: Watkins and Dayan, "Q-Learning",
```

```
Machine Learning, 8, 279-292 (1992).'''

import numpy as np
import itertools
import pandas as pd
import matplotlib.pyplot as plt

def import_reward_matrix(sheet_name):
# N is the number of discs
    R =pd.read_excel("CL State_Space.xlsx", sheet_name = sheet_name,
    index_col=0, header = 0)
    R = R.values
    for i in range(R.shape[0]):
        for j in range(R.shape[1]):
            if R[i][j] == -1000:
                R[i][j]= -1000000

class TowersOfHanoi:
    def __init__(self, state):
        self.state = state
        # "State" is a tuple of length N,
        #where N is the number of discs, and
        #the elements are peg indices in [0,1,2]
        self.discs = len(self.state)

    def discs_on_peg(self, peg):
```

```
        return [disc for disc in range(self.discs)
        #if self.state[disc] == peg]

    def move_allowed(self, move):
        discs_from = self.discs_on_peg(move[0])
        discs_to = self.discs_on_peg(move[1])
        if discs_from:
            return (min(discs_to) > min(discs_from)) if discs_to else True
        else:
            return False

    def get_moved_state(self, move):
        if self.move_allowed(move):
            disc_to_move = min(self.discs_on_peg(move[0]))
        moved_state = list(self.state)
        moved_state[disc_to_move] = move[1]
        return tuple(moved_state)

# Generates the reward matrix for the Towers of Hanoi
#game as a Pandas DataFrame
def generate_reward_matrix(N):      # N is the number of discs
    states = list(itertools.product(list(range(3)), repeat=N))
    moves = list(itertools.permutations(list(range(3)), 2))
    R = pd.DataFrame(index=states, columns=states, data=-np.inf)
    for state in states:
        tower = TowersOfHanoi(state=state)
```

```
        for move in moves:
            if tower.move_allowed(move):
                next_state = tower.get_moved_state(move)
                R[state][next_state] = 0
    final_state = tuple([2]*N)
    # Define final state as all discs being on the last peg
    R[final_state] += 100
    # Add a reward for all moves leading to the final state
    return R.values

def learn_Q(R, gamma=0.8, alpha=1.0, N_episodes=1000):
    Q = np.zeros(R.shape)
    states=list(range(R.shape[0]))
    for n in range(N_episodes):
        Q_previous = Q
        state = np.random.choice(states)
        # Randomly select initial state
        next_states = np.where(R[state,:] >= 0)[0]
        # Generate a list of possible next states
        next_state = np.random.choice(next_states)
        # Randomly select next state from the
        #list of possible next states
        V = np.max(Q[next_state,:])
        # Maximum Q-value of the states accessible from the next state
        Q[state, next_state] = (1-alpha)*Q[state, next_state] +
        alpha*(R[state, next_state] + gamma*V)      # Update Q-values
```

```
    if np.max(Q) > 0:
        Q /= np.max(Q)      # Normalize Q to its maximum value
    return Q


def get_policy(Q, R):
    Q_allowed = pd.DataFrame(Q)[pd.DataFrame(R) >= 0].values
    policy = []
    for i in range(Q_allowed.shape[0]):
        row = Q_allowed[i,:]
        sorted_vals = np.sort(row)
        sorted_vals = sorted_vals[~np.isnan(sorted_vals)][::-1]
        sorted_args = row.argsort()[np.where(~np.isnan(sorted_vals))][::-1]

        max_vals = [val for val in sorted_vals if val==sorted_vals[0]]
        max_args = [sorted_args[i] for i,val in enumerate(sorted_vals)
        if val==sorted_vals[0]]
        policy.append(max_args)
    return policy


def play(policy):
    start_state = 0
    end_state = len(policy)-1
    state = start_state
    moves = 0
    while state != end_state:
        state = np.random.choice(policy[state])
```

```
        moves += 1
    return moves

def play_average(policy, play_times=100):
    moves = np.zeros(play_times)
    for n in range(play_times):
        moves[n] = play(policy)
    return np.mean(moves), np.std(moves)

def Q_performance(R, episodes, play_times=100):
    means = np.zeros(len(episodes))
    stds = np.zeros(len(episodes))
    for n, N_episodes in enumerate(episodes):
        Q = learn_Q(R, N_episodes = N_episodes)
        policy = get_policy(Q,R)
        means[n], stds[n] = play_average(policy, play_times)
    return means, stds

def Q_performance_average(R, episodes, learn_times = 100, play_times=100):
    means_times = np.zeros((learn_times, len(episodes)))
    stds_times = np.zeros((learn_times, len(episodes)))
    for n in range(learn_times):
        means_times[n,:], stds_times[n,:] = Q_performance(R,
        episodes, play_times=play_times)
    means_averaged = np.mean(means_times, axis = 0)
    stds_averaged = np.mean(stds_times, axis = 0)
```

```
    return means_averaged, stds_averaged


def plot_results(episodes, means_averaged, stds_averaged, N, block=False):
    fig = plt.figure()
    plt.semilogx(episodes, means_averaged,'b.-',
    label='Average performance')
    plt.semilogx(episodes, means_averaged + stds_averaged,
    'b', alpha=0.5)
    plt.semilogx(episodes, means_averaged - stds_averaged,
    'b', alpha=0.5)
    plt.fill_between(episodes, means_averaged-stds_averaged,
    means_averaged+stds_averaged, facecolor='blue', alpha=0.5)
    optimum_moves = 2**N - 1
    plt.axhline(y=optimum_moves, color='g', label='Optimum (=%s moves)'
    % optimum_moves)
    plt.xlabel('Number of training episodes')
    plt.ylabel('Number of moves')
    plt.grid('on', which='both')
    plt.title('Q-learning the Towers of Hanoi game with %s discs' % N)
    handles, labels = plt.gca().get_legend_handles_labels()
    plt.legend(handles, labels)
    plt.show(block=block)



N = 2
# Number of discs in the Towers of Hanoi game
```

```
R = generate_reward_matrix(N)
episodes = [0, 1, 10, 30, 60, 100, 300, 600, 1000, 3000]
means_averaged, stds_averaged = Q_performance_average(R, episodes,
learn_times=100, play_times=100)
plot_results(episodes, means_averaged, stds_averaged, N)
ToH2 = (means_averaged, stds_averaged)

N = 3
# Number of discs in the Towers of Hanoi game
R = generate_reward_matrix(N)
means_averaged, stds_averaged = Q_performance_average(R, episodes,
learn_times=10, play_times=10)
plot_results(episodes, means_averaged, stds_averaged, N)
ToH3 = (means_averaged, stds_averaged)



N = 4
# Number of discs in the Towers of Hanoi game
R = generate_reward_matrix(N)
episodes = [1, 10, 100, 200, 300, 1000, 2000, 3000,
6000, 10000, 30000, 60000]
means_averaged, stds_averaged = Q_performance_average(R, episodes,
learn_times=10, play_times=10)
plot_results(episodes, means_averaged, stds_averaged, N, block=True)
ToH4 = (means_averaged, stds_averaged)
```

```
N= 5
R = generate_reward_matrix(N)
means_averaged, stds_averaged = Q_performance_average(R, episodes,
learn_times=10, play_times=10)
plot_results(episodes, means_averaged, stds_averaged, N, block=True)
ToH5 = (means_averaged, stds_averaged)

N= 6
R = generate_reward_matrix(N)
means_averaged, stds_averaged = Q_performance_average(R, episodes,
learn_times=10, play_times=10)
plot_results(episodes, means_averaged, stds_averaged, N, block=True)
ToH6 = (means_averaged, stds_averaged)
```

Bibliography

1. Richard D. Clarke et al. U.S. Special Operations Command Commander Talks Artificial Intelligence and Machine Learning With Hudson Institute Scholars. December 2021.

2. DoD data strategy. *United States, Congress, Office of Prepublication and Security Review*, 2020.

3. Evgeny Bryndin. Development of artificial intelligence for industrial and social robotization. *International Journal of Intelligent Information Systems*, 10(4):50–59, 2021.

4. Adriana Braga and Robert K. Logan. The Emperor of Strong AI Has No Clothes: Limits to Artificial Intelligence. *Information*, 8(4), 2017.

5. Joel E. Cohen and David Tilman. Biosphere 2 and Biodiversity–The Lessons So Far. *Science*, 274(5290):1150–1151, 1996.

6. Robert Larry Nance. An Advanced Tabu Search Approach to Solving the Mixed Payload Airlift Load Planning Problem, 2009.

7. Alan Mathison Turing. Intelligent Machinery, 1948.

8. Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in Medicine*, 23(1):89–109, 2001.

9. Alexander L. Fradkov. Early history of machine learning. *IFAC-PapersOnLine*, 53(2):1385–1390, 2020. 21st IFAC World Congress.

10. Auréliens Géron. Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems. 2020.

11. Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

12. S L Pinjare and E Kumar. Implementation of Artificial Neural Network Architecture for Image Compression Using CSD Multiplier. 08 2013.

13. *An Introduction to Neural Networks*. Taylor & Francis, 2018.

14. A. G. Ivakhnenko and V. G Lapa. *Cybernetics and Forecasting Techniques*. American Elsevier Publishing Co., 1967.

15. Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

16. R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

17. Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

18. Ronald Howard. Dynamic Programming and Markov Processes. 1960.

19. Edward L. Thorndike. *Animal intelligence; experimental studies*. New York, The Macmillan Company, 1911. https://www.biodiversitylibrary.org/bibliography/55072.

20. A Harry Klopf. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere Pub, 1982.

21. Haseeb Asif. What are the types of Reinforcement learning algorithms?

22. C.J.C.H Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

23. Vincent Mueller. Applying a Deep Q Network for OpenAI's Car Racing Game. 2021.

24. Allen Newell and Herbet Simon. Report on a general problem-solving program. 1958.

25. Stuart Jonathan Russell and Peter Norvig. Artificial intelligence: a modern approach. 2010.

26. J.M Bishop. Artificial intelligence is stupid and causal reasoning will not fix it. *Frontiers in Psychology*, pages 1—-18, 2021.

27. Chris Eliasmith. How to build a brain: a neural architecture for biological cognition. 2015.

28. Antonio Lieto, Mehul Bhatt, Alessandro Oltramari, and David Vernon. The role of cognitive architectures in general artificial intelligence. *Cognitive Systems Research*, 48:1–3, 2018. Cognitive Architectures for Artificial Minds.

29. Tim van Gelder. The dynamical hypothesis in cognitive science. *Behavioral and Brain Sciences*, 21(5):615–628, 1998.

30. Michael S. C. Thomas and James L. McClelland. Connectionist models of cognition. 2008.

31. John R. Anderson et al. Act-r: A theory of higher level cognition and its relation to visual attention. *Human-Computer Interaction*, pages 439—-462, 1997.

32. Vincent Müller and N. Bostrom. Future progress in artificial intelligence: A survey of expert opinion. In *Fundamental Issues of Artificial Intelligence*, volume 376. Springer, Cham, 01 2016.

33. Bin Liu. "weak ai" is likely to never become "strong ai", so what is its greatest value for us? 03 2021.

34. Oriol Vinyals et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

35. Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. 2017.

36. Kurt Peek. Q-learning-hanoi. 2016.

37. Watkins and Dayan. *Q-learning*. 2016.

38. Jaap. Tower of Hanoi.

39. Chris Versloot. Using Huber loss with tensorflow 2 and keras, Mar 2021.

40. Francois Chollet et al. Keras: Deep Q-Learning for Atari Breakout, 2015.

41. Tom Charnock, Laurence Perreault-Levasseur, and François Lanusse. *Bayesian Neural Networks*, chapter Chapter 18, pages 663–713.

42. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

Acronyms

AFRL Air Force Research Laboratory. 3

AGI Artificial General Intelligence. 21, 24

AI Artificial Intelligence. 1, 2, 5, 10, 21, 24, 36, 52, 53

ANN Artificial Neural Network. 8, 10, 12, 14, 15, 16, 18, 26, 54

CL Cargo Loading. 2, 3, 27, 30, 35, 37, 41, 42, 47, 52, 54

DNN Deep Neural Network. 14

DoD Department of Defense. 1

DRL Deep Reinforcement Learning. 1, 2, 5, 7, 8, 10, 16, 18, 26, 30, 36, 38, 40, 41, 42, 44, 45, 46, 52, 54

GPU Graphics Processing Units. 14, 26

HLAI Human-level Artificial Intelligence. 21

KP Knapsack. 2, 27, 51, 52

LeakyReLU Leaky Rectified Linear Unit. 15

ML Machine Learning. 5, 10, 11, 18

MLP Multilayer Perceptron. 14

NN Neural Network. 5

RL Reinforcement Learning. 1, 2, 5, 8, 10, 12, 18, 26, 27, 30, 41, 42, 43, 45, 46, 52, 54

SGD Stochastic Gradient Descent. 15

SPA Semantic Pointer Architecture. 23

SVM Support Vector Machine. 20, 21

ToH Tower of Hanoi. 2, 27, 30, 37, 42, 44, 45, 46, 47, 52, 54

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24–03–2020	Master's Thesis	Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE

ANALYSIS OF GENERALIZED ARTIFICIAL INTELLIGENCE POTENTIAL THROUGH REINFORCEMENT AND DEEP REINFORCEMENT LEARNING APPROACHES

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Turner, Jonathan, Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENS-MS-22-M-171

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory (AFRL)
Trevor Bihl, DAF, DR-III, PhD
2242 Avionics Circle
WPAFB OH 45433
trevor.bihl.2@afresearchlab.com

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Artificial Intelligence is the next competitive domain; the first nation to develop human level artificial intelligence will have an impact similar to the development of the atomic bomb. To maintain the security of the United States and her people, the Department of Defense has funded research into the development of artificial intelligence and its applications. This research uses reinforcement learning and deep reinforcement learning methods as proxies for current and future artificial intelligence agents and to assess potential issues in development. Agent performance were compared across two games and one excursion: Cargo Loading, Tower of Hanoi, and Knapsack Problem, respectively. Deep reinforcement learning agents were observed to handle a wider range of problems, but behave inferior to specialized reinforcement learning algorithms.

15. SUBJECT TERMS

artificial neural network (ANN), artificial intelligence (AI), deep learning, reinforcement learning (RL), deep reinforcement learning (DRL), Q-Learning, Deep Q-Learning, machine learning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Lance Champagne, AFIT/ENS
U	U	U	UU	117	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x9999; Lance.Champagne@afit.edu