APPLYING DEEP CONVOLUTIONAL NEURAL NETWORKS TO THE DRAGON BOAT PARTITION PROBLEM

BRETT REGNIER
Bachelor of Science, University of Lethbridge, 2019

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Brett Regnier, 2021

APPLYING DEEP CONVOLUTIONAL NEURAL NETWORKS TO THE DRAGON BOAT PARTITION PROBLEM

BRETT REGNIER

Date of Defence: August 25, 2021

Dr. J. Zhang Thesis Supervisor	Associate Professor	Ph.D.
Dr. W. Osborn Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Y. Chali Thesis Examination Committee Member	Professor	Ph.D.
Dr. J. Sheriff Chair, Thesis Examination Committee Member	Assistant Professor	Ph.D.

Dedication

I dedicate this thesis to my beloved wife, Jenny Regnier, who challenged, encouraged, and loved me from start to finish, as well to my parents, siblings, and friends for supporting me and listening to my ramblings.

Abstract

We investigate approximating the Dragon Boat Partition problem, a practical real-world variant of the Partition problem. A team of dragon boat participants must be partitioned with an approximately balanced arrangement with a preferable weight difference of 0. We present two approaches that capture the participant characteristics. The first approach takes a heuristic route. The second approach applies Deep Convolutional Neural Networks to the problem, with two versions. In our 10,000 episodes per experiment, our heuristic implementation had an average episode runtime of 1.84ms, an average of 7.39 steps per episode, perfect left-right approximation rate of 98.53%, perfect front-back approximation rate of 89.16%, and a perfect combined approximation rate of 90.15%. Whereas our best deep learning model has an average episode runtime of 1.23ms, an average of 4.65 steps per episode, perfect left-right approximate rate of 98.00%, perfect front-back approximation rate of 95.13%, and a perfect combined approximation rate of 94.28%.

Acknowledgments

First and foremost, I would like to thank Dr. John Zhang, for the help he provided through this whole thesis. His consistent encouragement to try new ideas, guidance on which topics would be best to investigate, and lastly for being a friend.

I would like to extend my gratitude to my committee members, Dr. Wendy Osborn, and Dr. Yllias Chali, for the encouragement, improvements, and good times throughout both my Bachelor and Master degrees.

I would not be here today without my parents pushing me to be best I could be, and loving me every step of the way.

Many thanks to my brother Ryan and his wife Emily for calling to check up on me and chat about regular life.

To my sister Alyssia, and her family, thank you for taking time to support me.

Deep gratitude goes to my best friend Daylend de Grasse, your humor and friendship was highly valued and greatly appreciated.

Jenny, my loving wife, you are the reason I continued to push forward with this thesis. Thank you for being there every step of the way. I truly could not have finished this thesis without you.

Praise be to God and Jesus Christ my saviour for everything in this wondrous life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial neural networks, which shall be referred to as neural networks (NN), are inspired by biological neural circuits [56]. The most basic NN contains 2 layers, the input layer, and the output layer. The output layer is connected in a mesh-like network to the input layer through a set of weights. The weights scale the inputs, which determines the activation strength of the neuron based on the scaled input [1]. Neural networks and have gained substantial popularity in the last decades. This popularity growth is largely due to the broad applications, as well as the creation of deep neural networks. Deep neural networks are extremely versatile and can be applied generically to any problem, given the right architecture and sufficient training time. In other words, neural networks are generic approximators and can be applied in approximating a given function [1, 37]. The versatility of deep neural networks comes from having multiple neural layers, known as a hidden layer. Deep neural networks are multilayered neural networks, which have at least one hidden layer between the input layer and the output layer [37]. Deep neural networks have accomplished many incredible feats that used to be considered near impossible for machines to achieve without a considerably complex algorithm, such as surpassing humans at the game of Go [47, 48], facial recognition [30], and playing Atari video games [34, 35].

There are many forms of learning for neural networks. The most commonly used form is supervised learning, unsupervised learning, and reinforcement learning, each with their respective uses. Supervised learning utilizes human-labeled data to fit a model for classification problems [5]. Unsupervised learning expects a model to fit itself to the data without

labels [19]. Reinforcement learning uses neither labels nor clusters, but rather, a learning agent that interacts with an environment and fits the model around the interactions [1].

As mentioned earlier, neural networks are regarded as generic function approximators [37]. Therefore we could find a neural network that can approximate any function. For example, a simple function problem is the exclusive or (XOr) problem, where a deep neural network, particularly a 2-layer deep neural network, is taught to predict the logic of a XOr table [9]. In this thesis, we will discuss the use of deep neural networks for function approximation on optimization problems, particularly, a variant of the partition problem.

1.1 Machine Learning

The Mark 1 perceptron, created by Frank Rosenblatt in 1958, is known as the first successful neuro-computer. It is a custom-made machine from IBM, which was built primarily for image recognition [1, 17]. It is able to recognize some images; however, it struggled to identify many visual patterns, causing frustration and disinterest in neural network research. Research however resumed in the 1990s, where neural networks moved from hardware to algorithmic software approaches [17].

The advancement of hardware and the increase in data collection caused the revival of neural network research. The collected data can be used to discover hidden relationships in everyday lives, such as purchasing history, activity interest, and entertainment patterns. Using new hardware, in particular, graphical processing units (GPUs), the time to train neural networks drastically dropped because GPUs are inherently designed to be parallel processing units, so the training is split across processing cores in the GPUs [1].

1.1.1 Neural Networks

A neural network consists of a few fundamental pieces: layers, neurons, weights, and activation functions. All neural networks start with an input layer, where data is fed into the network and then propagated through the layers. Each layer consists of a group of neurons.

Each neuron in the layer, also known as a computational unit, is connected to other neurons by an adjustable parameter, known as weight, in the next layer. The weights from the previous neurons have a direct effect on the activation function inside of the connected neuron. The neural network output is based on the propagation of data through the neural layers until it reaches the final layer, known as the output layer [1].

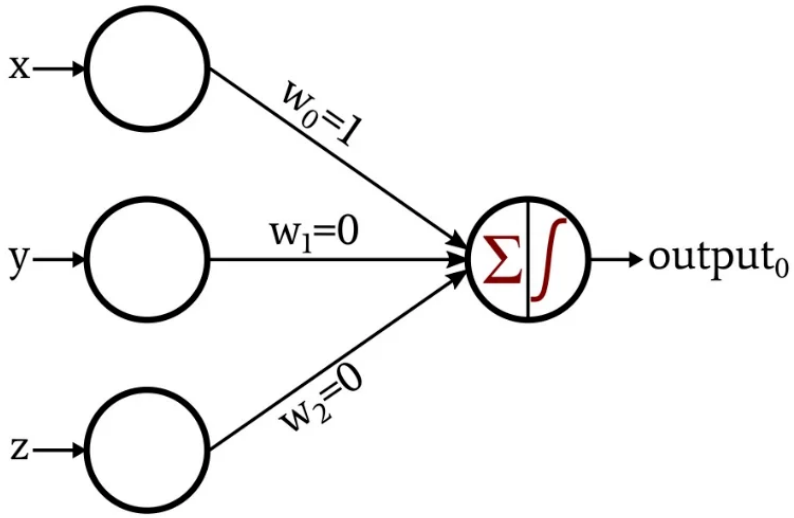

Figure 1.1: Perceptron architecture retrieved from All About Circuits.

The perceptron is the simplest neural network, which consists of an input layer, and output neuron as seen in Figure 1.1[1]. Perceptrons are only able to perform linearly separable problems and tend to struggle with classification when problems become non-linearly separable [1]. The solution to this issue with perceptrons was the use of intermittent layers, called multilayered perceptrons or more commonly deep neural networks. Deep neural networks are very similar to single-layer perceptrons. The difference is that there are one or more layers of neurons between the input and the output known as hidden layers giving the notion of being deep. Each subsequent layer feeds its output data into the next layer until the propagation reaches the output layer where the final prediction is calculated [1]. Neural Networks are good at distinguishing and organizing distributions on finite noisy data. Once the data distribution has been learned, NNs are very good at generalizing problems within

[1]https://www.allaboutcircuits.com/technical-articles/how-to-train-a-basic-perceptron-neural-network/.

the learned distribution; however, NNs sometimes struggle with problems and data that fall outside the distribution [7].

Aggarwal [1] describes convolutional neural networks (CNN), as being the historically most successful neural network type. Image recognition, object detection, and localization, and text processing are all examples that CNNs excel in. Coupled with big data collections, CNNs quickly became a powerful force in classification problems. CNNs share a lot of similarities to typical NNs. The primary difference is that neurons are organized into a 3-dimensional matrix for height, width, and depth in CNNs [39]. The CNN architecture fits well for image data, as images are organized into a 3-dimensional object of pixels, height, width, and RGB colour channels. A notable CNN known as LeNet, proposed by LeCun et al. [31], for being a significant influence in advancing deep learning. LeCun et al. [31] shows that CNNs could replace human-made feature extractors.

1.1.2 Learning Algorithms

Supervised learning utilizes human labelled data, where data has been categorized into a 2-tuple, (X,t), where X is a set of features and t is the label for the features in X. Using the data-label pair, the output predictions of the NN can be updated by measuring the prediction (output) against the ground truth (label) [5]. Unsupervised learning, on the contrary, has no human labelled data; rather, the NN is expected to discover clusters of similar structure in the data [19]. Reinforcement learning is in between supervised learning and unsupervised learning. Instead of being provided complete data, a learning agent, which contains a deep NN, interacts with an environment and makes a decision based on the current state of the environment and the future expected reward given the decision [1].

1.2 Optimization

Optimization has a broad range of problems, with the same goal in mind to find the optimal solution [40]. Predregal et al. [40] states the two important steps to optimality: the

first being the cost must be minimized, second being constraints must be explicitly enforced. one such optimization problem is the Partition Problem. Another optimization problem is the dragon boat partition problem. After defining the dragon boat partition problem we will show how it is a variant of the partition problem.

1.2.1 The Partition Problem

Splitting a group of items into two or more numerically equal unique groups is a prominent problem. Some examples include creating a team of balanced athletes, multiprocessor scheduling, state asset partitioning, VLSI circuit size minimization, and minimizing the delay for public-key cryptography [22, 33]. These examples are all practical applications of the partition problem. The partition problem is defined as: for a list of n positive integers, find a partition such that the set is evenly distributed [22]. Formally, given a set $S \subset \mathbb{R}^+$, find a partition $S_1 \subset S$, and $S_2 \subset S$ where $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$, such that the difference, known as discrepancy,

$$E(A) = \sum_{s \in S_1} s - \sum_{s \in S_2} s \tag{1.1}$$

is minimized [33]. A perfect partition is defined as $E = 0$ for when $\sum_{s \in S} s$ is even and $E = 1$ for when $\sum_{s \in S} s$ is odd. Although a simple definition, the partition problem is deceptively hard. The problem on a small set, such as equally splitting the list of {1,2,3,4,5} into two groups, is trivial. However, splitting a group that has one hundred elements, starts to become difficult to achieve efficiently, especially when using the brute force method of checking every $n!$ possible combination [22]. Therefore, the use of fast approximate algorithms is needed, such as the greedy heuristic [22, 33], or the Karmarker and Karp differencing method [33].

1.2.2 Dragon Boat Optimization Problem

The partition problem has a broad scope, such that it can be applied to real-world tasks, like balancing a dragon boat. The original dragon boat optimization problem is defined as

follows. A dragon boat is a long slender boat with 22 seating arrangements, as shown in Figure 1.2. A boat is filled with a team of 22 members with 20 paddlers sitting side by side down the boat, a steersman at the back directing the boat, and a drummer at the front facing the rest of the team [23]. Before teams get into the boat, the first step is to find an approximate optimal partitioning of the team members in their subsequent positions. Having a balanced dragon boat ensures that the boat is parallel with the water allowing maximum surface area and water flow around the boat, increasing boating efficiency and speed [49]. For example, if the back is heavier it would cause the front-end to lift and the back-end to sink deeper into the water, effectively slowing the boat. To find an approximate optimal partitioning multiple constraints must be considered. There are certain positions in the boat that each rower prefers to be in, such as the second seat on the left. Accommodating preferences can only be taken so far, as the balance of the boat will affect the boat's performance. After team members are placed in their preferred position, the next step is positioning participants by their dominant hand. Following is the partitioning of the left and right sides, such that the weight difference between the left-hand side and right-hand side is minimized. Then the front and back are partitioned where the difference between the front-half and back-half must be close to 30lbs, with the front-half being heavier. Lastly, the heaviest members of the team should be positioned in the center of the boat and the weights are gradient outwards towards the front and back [49].

Therefore, we can say the dragon boat partition problem is a variant of the partition problem if we only require that the left-hand side and right-hand side have equal weight in a dragon boat partition problem instance, then it is the original partition problem. So we reduce the partition problem to the dragon boat partition problem through restriction and this reduction can take polynomial time. Since the partition problem is NP-hard, so is the dragon boat partition problem [18]. We believe that, from the best of our knowledge, this is the first time the dragon boat partition problem is handled seriously.

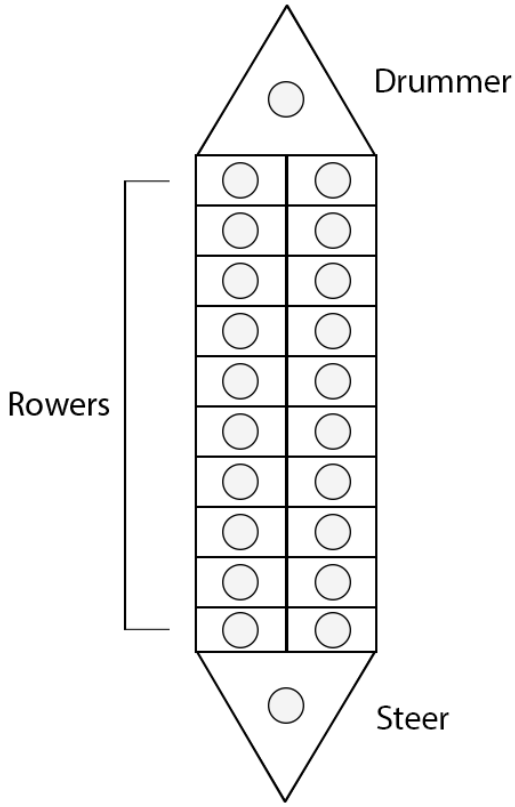

Figure 1.2: Top down view of a Dragon boat.

1.3 Application of Convolutional Neural Networks and Deep Learning to Optimization

The most common uses for convolutional neural networks fall in the realm of vision, such as identifying handwritten characters [31], image classification [28], facial recognition [30], and playing Atari games [34, 35]. However, there are other applications that use convolutional neural networks, such as AlphaGo's Go board positions [47, 48].

Image pattern recognition is the primary use for CNNs since typically convolutional neural networks inputs are organized in matrix form [39]. However, Silver et al. [47, 48] use convolutional neural networks for AlphaGo, and variants, in a different manner. Rather than using a convolutional neural network for image pattern recognition, Silver et al. feed in Go board positions as a 19x19 image to encode a representation of the positions.

1.4 Problem Statement

Our primary focus is to implement a practical solution to explore the dragon boat partition problem using the convolutional neural network model. We plan to digest positional data and output meaningful sorting actions to arrange boat team members. We take two different approaches - one using a heuristic, and the other using deep learning CNNs.

For inputs, we design and create a dragon boat partition problem environment. As stated above, the dragon boat partition problem has many constraints. We will need to address the hand dominance, left-right weight, and front-back weight constraints. However, we will be ignoring seating preferences as finding a close approximation through sorting is the intended task. As well, the final constraint will not be addressed using CNNs as we can use the quick-sort algorithm on the front-left and front-right groups and the reversing the quick-sort algorithm, where instead of sorting in ascending order the algorithm would instead output in descending order, on the back-left and back-right groups to approximate this constraint [45].

A few assumptions were made for our problem so that we could run our experiments in a reasonable amount of time. The first assumption is that the boat will only need to balance eight participants, as the training time and network size growth are directly related to the number of participants. The aim is to show the proof of concept in a small case as it can always be scaled upward given enough time and hardware. Secondly, we assume that given the list of participants there is an appropriate number of rowers who can only row left or right. This means that there is at most an equal number of left or right-handed rowers as there are seats on the left-side or right-side for that handed type. Thirdly, it is assumed that in our generated data a perfect discrepancy approximate partition exists for the left-right and front-back goals, with the relaxation values for left-right and front-back of zero, one, two, and five. Lastly, it is assumed that the drummer and the steersman are always the same weight per generated instance and do not change.

1.5 Contributions

This thesis proposes two novel approaches for combinatorial optimization algorithms, particularly the dragon boat optimization problem. First, we present a heuristic agent that can approximate the optimal dragon boat instance. Secondly, we convert a combinatorial optimization problem with multiple constraints and data with a channeled context, into an image to be predicted using a convolutional neural network. To create two models - the Supervised Reinforcement Learning Model, and the Pure Reinforcement Learning Model, also known as the SRL-model and PRL-model respectively. Both of these models can approximately predict expert moves in partitioning a dragon boat.

The heuristic approach has two different designs, heuristic sorter, and heuristic agent. The heuristic sorter is designed as a typical algorithm, with direct access to the data it will manipulate. This method is considerably more effective and efficient than our heuristic agent. We compare the explicitly given rules to the heuristic agent against the CNNs learned rules. The heuristic agent has a slight advantage, as it is provided constraints and relaxations at the start. In contrast, the CNN must implicitly learn the constraints and relaxations.

Generally, if a CNN can approximate constraints based on positional and dimensional information in a converted dragon boat environment input as an image, we believe that a CNN can be used to approximate other combinatorial optimization problems, potentially better than humans or algorithms. We believe that our CNN models will be an effective approximator with a high success rate, rivaling the heuristic approaches. We want to show that by using a CNN we can reduce the steps required to find an approximate. We believe this can show how we could potentially use a CNN to approximate other balancing algorithms, such as shipping freighters, or aeroplane weight balancing.

1.6 Outline

The remainder of this thesis is as follows. In Chapter 2, we will discuss optimization problems and approximation algorithms concerning the partition problem, and discuss a

variety of topics related to convolutional neural networks and deep learning.

Chapter 3 presents our heuristic approach. We define all of the assumptions we have for the dragon boat environment. Furthermore, after detailing our assumptions, we move into the formalization of the dragon boat partition problem, and implementation of our dragon boat environment, along with our data generation for the first phase of our implementation in Chapter 4. Following that, we explain our heuristic algorithm. We designed our heuristic algorithm into two versions, one being a classical algorithm and the other emulating a deep learning agent. Finally, our experiments, framework, and results are shown and will be used as a benchmark to test against our implementation in Chapter 4. We will also discuss the differences in the designs and their effectiveness.

In Chapter 4, we present our deep learning approach. There is a discussion on how we apply deep learning to the dragon boat partition problem. We will discuss how our problem fits in a convolutional neural network, followed by the architecture we designed. The training strategies are presented in detail, and we explore the training processes and hyperparameters in our approach. We split our deep learning approach into two models. The first is pre-trained using supervised learning and later is tuned with reinforcement learning. The second version is trained only using reinforcement learning. We compare the results of our two models against our approach proposed in Chapter 3. We close the chapter with a summary of our results, the effectiveness, and the limitations.

To conclude this thesis, in Chapter 5, we present a synopsis of our approaches and results in Chapters 3 and 4. We finish with a discussion of the limitations to our approaches and our potential future work along the same direction.

Chapter 2

Background

The purpose of this chapter is to familiarize the reader with the technologies, training methods, and evaluations in optimization problem and approximation algorithms, and deep learning. We begin by explaining optimization problems and approximation algorithms in a high-level manner, moving forward into convolutional neural networks, applications that have utilized convolutional neural networks for optimization problems, deep learning, supervised learning, unsupervised learning, and finally reinforcement learning. Our goal of each section is to familiarize the reader with the general knowledge needed to understand our proposed approaches.

We organize this chapter in the following manner. We begin by briefly discussing optimization problems and approximation algorithms below. Next, we have a detailed discussion of the architecture and layers of convolutional neural networks in Section 2.2. Then, in Section 2.3 we briefly introduce deep learning and neural network optimizers. We cover the two types of deep learning used in this thesis in Sections 2.4 and 2.5. Furthermore, we cover the applications of convolutional neural networks in relation to optimization problems Section 2.6. We close this chapter with a summary in Section 2.7.

2.1 Optimization Problems and Approximation Algorithms

Our goal with the dragon boat problem is to show the possibility of having an NP problem approximated using convolutional neural networks, by training a CNN on input converted into an image. Optimization problems come in two classes, P or NP [10]. Class

P consists of all solvable problems in polynomial time. These problems all have a deterministic solution [10]. The other class is NP which consists of problems whose solutions can be verified in polynomial time[10].

It is extremely difficult, maybe even impossible, depending on if P=NP is proved, to get optimal solutions to NP problems in polynomial time [10, 53]. Therefore, instead, we focus on trying to find an approximate solution, thus allowing us to access that domain of solutions without having to spend exhaustive time searching [53].

As described above there are many NP problems. We will focus on the partition problem, particularly the dragon boat partition problem. While it is a simple problem by definition, it is still an NP-complete problem. As such, there is no polynomial algorithm to solve this problem deterministically using the current computing technology. However, an exhaustive algorithm that will find the solution is the brute force method, which involves, searching every possible combination [22]. This method guarantees an answer to whether or not the set S of n elements can be perfectly partitioned. As well, the brute force method can identify the partition with the lowest discrepancy. However, the brute force method is extremely slow with a growth rate of $O(2^n)$ [22]. So it is more practical to use a fast approximation algorithm. The simplest of those algorithms is the greedy heuristic which is mildly successful, but only with a set of small numbers, particularly positive integers less than or equal to 10 [22]. The greedy heuristic consists of sorting all of the numbers by their magnitude, step through each item in the sorted set, pick the next item with the largest magnitude, and add it to the subset with the lower sum to keep the discrepancy minimized [22, 33]. The greedy heuristic scales at a rate of $O(n-1)$ for real-valued items. Furthermore, the time complexity of it is $O(nlogn)$. The differencing method by Karmarker and Karp [22], also known as the KK heuristic, takes a different approach to achieve a polynomial-time approximation. Instead of viewing the next largest value, the differencing method seeks to shrink and replace the magnitude two numbers at a time into more manageable blocks. Eventually, by selecting two numbers and taking their absolute difference, there will be one

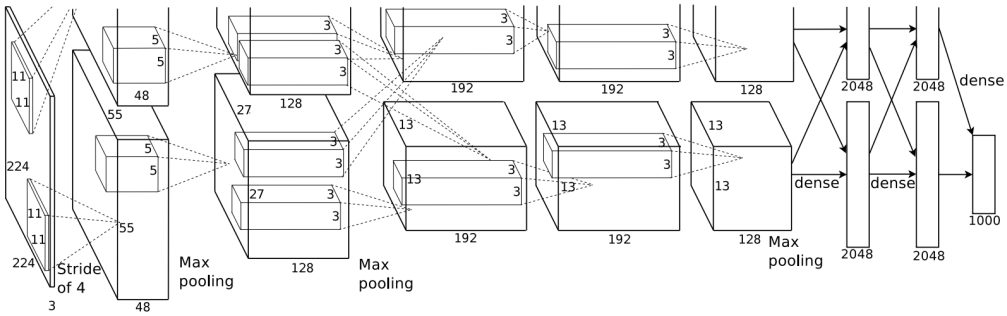

Figure 2.1: CNN architecture used for classifying image net by Krizhevsky et al. [28].

number remaining, which is the discrepancy. This search can be undone by reversing each step, starting and stopping at any point to try again. [33].

The KK heuristic itself approximates a near-optimal solution. Hopefully, using deep neural networks we can achieve better optimality to the problem than traditional algorithms and humans accomplish. We conjecture that this is possible, as it has been shown by teams akin to Silver et al. [47, 48], Mnih et al. [34, 35], Krizhevsky et al. [28], and Lawrence et al. [30].

2.2 Convolutional Neural Networks

In our experiments, we use CNNs due to their versatility and structure. In this section, we will explore the architecture of CNNs, why they are popular, and their most common layers. CNNs hold a great deal of potential and have major contributions to several advancements in research and applications. Such contributions include playing Atari games with human-level or better control [34, 35], becoming the world champion at the game of Go [47, 48], image recognition [28], recognition of handwritten characters [31], facial recognition [30], and even breast and gastric cancer classification [51, 26]. As such, CNNs are regarded as the most successful model and the most popular neural network architecture.

2.2.1 Architecture

The most important feature of a CNN is its ability able to effectively extract features of a multidimensional array, without having to hand-design a feature extractor [31]. CNNs are well suited to problems involving images, due to their 3-dimensional structure of neurons. The neurons are organized to receive an input in the spatial dimensions, height and width, and depth as shown in Figure 2.1. In contrast, typical neural networks only handle single-dimensional input and thus an image of shape *width* x *height* x *depth* would be rolled out into 1 x $(width * height * depth)$ input, requiring a far larger number of weights [39, 3]. For example, assume we are using a typical fully connected neural network (multi-layered perceptron), where each neuron in each layer is attached to each neuron in the next layer. If we try to input an image with the width of 64, height of 64, and depth of 3 for RGB values, then we will need $64 * 64 * 3$ weights per neuron in the next layer. Therefore, we will have $64 * 64 * 3 * n$ weights, where n is the number of neurons in the next layer. For clarification, if we had a layer with the same number of pixels, in greyscale, to neurons, $64 * 64 * 1$ pixels values and $64 * 64$ neurons, then there would be $16,777,216$ weights to adjust between these layers. To deal with this inefficiency CNNs look at a small local area, known as a receptive field, within the input, and feed them through a group of neurons in a filter or a kernel. The only difference between a kernel and filter is that a filter is a set of stacked kernels for inputs with a depth greater than 1. We will refer to both, filter and kernel, as kernel. The number of weights for each filter is determined by the width of the kernel k_w, height of the kernel k_h, and number of kernels k plus the 1 bias. So if a single 3x3 kernel connected to a layer with $64 * 64$ neurons, the result would be $((3 * 3) + 1) * 1 * 4096 = 40,960$ weights. Moreover, CNNs can also share weights in each kernel, and thus only the depth of the image will affect the number of neurons, further reducing the number of weights in a CNN. Since CNNs greatly reduce the number of weights, they have a distinct advantage in training speed, while still having the ability to extract complex features [39, 3].

Whilst CNNs use the spatial dimension in their convolution operation, it is required for

the problem input to be spatially independent in an accessible form. The position in which the object is detected in the input should not matter in order to identify it [3]. Typically deep CNNs have multiple layers of convolutions with non-linear activation functions and pooling layers between convolutional layers. This allows for each layer to extract more distinct and rich features from the input data [39, 3].

2.2.2 Layers

The CNN architecture has tremendous versatility. There are several different types of layering utilized in CNNs. We will only be covering the most common layers. The main operation used in CNNs takes place in the convolutional layer, where the mathematical linear operation convolution is applied. After convolution the input is passed into the next layer, which could be a non-linear activation function that transforms the inputs into a certain range in order to remove or compress, or into a max-pooling layer, which simply has an *n x m* receptive field and takes the largest value from the field and passes it forward. This set of layers can be stacked, with optional activation functions and max-pooling [39, 3].

2.2.3 Convolution

The convolution layer is the primary operation of CNNs. The goal of this layer is to extract features and is made up of learnable neuron weights in each kernel. Kernels also have three hyperparameters, depth, stride, and zero-padding [39]. Hyperparameters consists of the variables determining the structure of an NN and the variables determining how the CNN is trained, such as learning rate α. The depth is directly responsible for detecting the features, as it is the number of kernels stacked together allows for more visual features to be extracted. Stride is the distance the receptive field slides upon the input. Overlapping occurs when the stride distance is less than the dimension of the kernel, resulting in similar features being convolved. Lastly, zero-padding pads the input, with zeros, around its border, which allows for better edge detection and helps control the size of the destination matrix [39, 3].

CNNs have the ability to stack convolutional layers. Each subsequent layer will extract different visual features than the last, and create a heat map of the most important visual features in a given dataset. Activations occur when a kernel 'sees' a visual feature that it is optimized for. Convolutional layers use kernels to extract relational data from the given input, such as shapes, curves, and edges in images. It does this by using the sliding window concept, sliding across the matrix based on the set stride, and kernel spatial dimensions [3]. CNNs have a special operation known as the convolutional operation, where the receptive field is multiplied and summed with the kernels for the entire input [39, 3, 1]. For the convolutional operation, let the input be denoted by I, the kernel be denoted as K, where K_h is the height of the kernel, K_w is the width of the kernel, I_h is the height of the image, I_w is the width of the image, and I_c is the number of channels in the image. The formula for a convolutional operation is defined as [32]:

$$conv(I,K)_{x,y} = \sum_{i=1}^{I_h} \sum_{j=1}^{I_w} \sum_{k=1}^{I_c} K_{i,j,k} * I_{x+i-1,y+j-1,k} \tag{2.1}$$

Thus, we can see that a local extraction of the receptive field from the input is element-wise multiplied with the kernel at each resulting matrix element, for each filter-channel pair [32]. However, in order to perform this equation, we first need to know what the spatial dimensions of the destination matrix are. Mebsout [32] defines the formula as follows:

$$dim(conv(I,K)) = \begin{cases} \left(\left\lfloor \frac{I_h+2p-K_h}{s} + 1 \right\rfloor, \left\lfloor \frac{I_W+2p-K_w}{s} + 1 \right\rfloor \right) & \text{if } s > 0 \\ \left(I_h + 2p - K_h, I_w + 2p - K_w \right) & \text{Otherwise} \end{cases} \tag{2.2}$$

where p is the zero-padding parameter, and s is the stride parameter. The destination matrix will only be in the spatial dimension as the convolution operation will press the entire tensor, a n-dimensional vector, into a 2D tensor, which will be stacked upon all other destination matrices from the other kernels. As was shown in Formula 2.2, the stride and padding have a significant effect on the dimensionality and must be considered for keeping the original

dimensionality equal [3, 32, 39].

2.2.4 Activation Functions

We define activations as occurring when a kernel sees a feature whereas activation functions are used to determine whether a neuron will activate/fire or not depending on whether the weighted sum of its input and bias reaches a certain threshold. They come in two varieties, linear or non-linear, which depends on the function it represents and the target domain. We will discuss only non-linear activation functions. Non-linear activation functions are particularly useful for problems that are not linearly separable, as they introduce non-linearity into the NN [38]. Some of the most often used activation functions are Sigmoid, Hyperbolic Tangent (Tanh), Softmax, and Rectified Linear Unit (ReLU) [3, 38].

The Sigmoid activation function is the most commonly used activation function. It adds non-linearity by squashing the input between 0 and 1. It is used often in the output in many deep learning NN. Unfortunately, Sigmoid has seen less favour in recent years, as it has a fundamental problem known as the vanishing gradient problem. The vanishing gradient problem is where the gradient value in back-propagation begins to disappear the deeper the calculation travels [3]. Moreover, Sigmoid is notorious for its slow convergence [38]. The Sigmoid function is defined by:

$$f(x) = \left(\frac{1}{1+e^{-x}}\right) \tag{2.3}$$

Lastly, the Sigmoid function can be also be used in the final layer. However, it is only particularly useful in binary classification models [38]. Despite the shortcomings of the Sigmoid function, we found that our CNN seemed to only learn when it was used in the hidden layers during reinforcement learning.

The Tanh function was proposed to address some issues with Sigmoid. One such issue Tanh addresses is being zero-centered by squashing the inputs between -1 to 1, instead of 0 to 1. Further, Tanh has reduced training time in deep NNs and therefore has a better overall

performance in general. The Tanh function is defined by

$$f(x) = \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right) \tag{2.4}$$

Even though Tanh addresses and improves upon some problems Sigmoid has, Tanh is not able to fix the vanishing gradient problem experienced in Sigmoid [38].

Softmax, is a slightly different activation function than Sigmoid, Tanh, and ReLU, as it is most often used in the final output layer. Ramachandran et al. [38] describe the Softmax function as a technique to calculate probability distributions of a given tensor. Thus, the range for Softmax is between 0 and 1, since all the probabilities of each element in the input must add up to 1. Softmax is defined by the following equation:

$$f(z)_i = \frac{exp(z_i)}{\sum_j exp(z_j)}[21] \tag{2.5}$$

where z is the input vector. Softmax is used in most multi-classification models, as it provides a probability for each output value and can be easily measured when sampling and analyzing [38].

ReLU has recently taken the limelight in deep learning. The surge in popularity for ReLU is due to its faster learning rate, simpler function and gradient, and its preservative representation of linear models. These advantages make optimization with linear models much easier. [3, 38].

$$f(x) = max(0,x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.6}$$

As shown in Equation 2.6 ReLU squashes numbers between 0 and the activation, introducing sparsity into the network. ReLU has its advantages (faster computation and sparsity introduction), it has a large downside, known as dying ReLUs [38, 14]. This occurs because it rectifies the output, forces a minimum of 0, and allows only the positive signals to pass. Since only positive signals pass the gradients for these values, then the model will only fit

around the passed signals, eventually updating to the point where some neurons will result in dying ReLUs. Dying ReLUs are unfavorable, as their output is always zero, providing no contribution to network learning [38, 14]. There are some fixes proposed, including Leaky ReLU, Parametric ReLU, Randomized Leaky ReLU [38], and the Scaled Exponential Linear Unit SELU [14]. Each of these fixes allows some negative values to flow through to backpropagation in order to prevent dying ReLU.

In conclusion, each activation function has its upsides and downsides, as well as its own specific uses. The activation function used for a NN fully depends on the type of input and output data and the problem under attack.

2.2.5 Pooling

The purpose of the pooling layer, which is one of the two extra layers used between convolutional layers, is to reduce the number of weights by further reducing the dimensionality and complexity of the model it is known as down-sampling [3, 39]. The pooling layer will have a similar setup as the convolutional layer kernel, where it has a receptive field view of the input, however, instead of convolving, it takes the values respective to the pool's operation, and slides the window over based on the stride amount. For example, a max-pooling layer, with a receptive field of 2x2, the most common pool and size used, would take the largest value in the field, pass it to the resulting tensor, and the rest is discarded [3, 39]. Unlike the convolutional layer, the pooling layer preserves the input depth. As pooling is a destructive operation, typically only 2x2 with a stride of 2, or 3x3 with a stride of 2 for overlapping pooling is used [39].

2.2.6 Fully Connected Layer

The last layer in a CNN is typically a fully connected (FC) layer. The FC layer looks analogous to a multi-layered perceptron. Each neuron in the FC is fully connected to each neuron in the previous layer [3, 39]. Before the output of a CNN can be fed into the FC layer, the CNN output has to be flattened into a single-dimensional vector. The FC layer

has the most number of neurons, and therefore the most number of weights in a CNN since each neuron will be directly connected, causing it to be the slowest part to train [3].

2.3 Machine Learning

Machine learning has many subcategories of learning such as: supervised learning, unsupervised learning, and reinforcement learning, each with its unique uses, advantages, and disadvantages. Supervised learning is particularly good at predicting human-labeled data [5]. Unsupervised learning is designed to extract patterns from data without human nor environmental feedback [19]. Reinforcement learning falls in-between supervised and unsupervised learning [29], where an agent interacts with an environment around it and is rewarded based on the interactions [1].

Neural networks, particularly multi-layered feed-forward networks, often learn using a process known as backpropagation. There are two phases to a neural network the forward phase and the backward phase. The forward phase begins with the input into the NN and propagates through each subsequent layer, with the final layer outputting a prediction [1]. The output prediction can be measured against some ground truth y, which can be a label in supervised learning, or a reward value in reinforcement learning, to receive a loss value or error [1, 5]. Next, the algorithm calculates the derivative of the loss value with respect to output and feeds it backwards through the network in the backward phase. After inputs have been fully digested in a feed-forward network and an output is given, the backpropagation phase beings. It feeds the derivative backwards to learn the gradient of the loss using the chain rule by updating the NN weights $\overline{W}$, the set of weight connections between neurons, by multiplying the loss with a hyperparameter known as the learning rate α. The weights determine which, and how strong, neurons are activated [1].

There is a large variety of loss functions and gradient descent optimizers, and choosing which one depends on the task you are trying to accomplish. Loss functions will be discussed after establishing different types of learning and covering gradient descent opti-

mizers below.

2.3.1 Gradient-based Optimizers

For neural networks to properly learn, they must be optimized in the opposite direction of loss gradients with respect to the network's parameters [43]. There is a multitude of different algorithms that achieve a form of optimizing. A very popular optimizer is Gradient Descent, GD, which minimizes the objective function's parameters [43]. The rate in which the GD algorithm minimizes is determined by the step size α, which determines how large of a step towards local minima [43]. There are three types of GD - batch gradient descent, stochastic gradient descent, and mini-batch gradient descent. Batch gradient descent, the original gradient descent, updates the parameters using the gradients for an entire dataset. Conversely, stochastic gradient descent, SGD, executes descent per training sample. Mini-batch gradient descent combines both GD and SGD, to update parameters for every mini-batch in the training set [43].

There are a variety of GD algorithms at our disposal, such as Adaptive Gradient, Adagrad [15, 43], RMSProp [43], and a combination of Adagrad and RMSProp, Adaptive Moment Estimation, and Adam [27, 43]. Adagrad, as the name implies, adapts the learning rate to be greater for more sparsely seen data, and to be lower for more frequently seen data [43]. RMSProp seeks to rectify the intense diminishing learning rates in Adagrad [43]. Lastly, Adam [27] seeks to merge the improvements from Adagrad and RMSProp, the ability to handle sparse features and online non-stationary settings respectively.

2.4 Supervised Learning vs. Unsupervised Learning

Supervised learning relies on human-labeled datasets. Having human-labeled data can be translated easily into classification models. There are 4 classification categories: binary classification, multi-class classification, multi-label classification, and imbalanced classification [12]. In binary classification, there are only two class labels to be chosen from [12].

For example, in a group of images where there are only two labels, cat or dog. Multi-class classification is when there are two or more class labels to be chosen from [12], such as classifying the type of animal in an image with multiple labels to choose from. Multi-label classification is when there are multiple classes in a single dataset, where each item in the dataset can be classified with one or more labels, such as an image with multiple fruits in it [12]. Lastly, imbalanced classification is when there are a majority of data in a single class while the other class is sparse. This classification is akin to binary classification but requires special techniques [12].

As mentioned above, there are many classification categories in supervised learning. Each has a commonly-used loss function [13]. For binary classification, binary cross-entropy loss is typically used, and for multi-class classification a similar variant is used, known as categorical cross-entropy loss [13]. There are indeed more complex and impressive loss functions, but their use depends on the problem at hand [13].

While supervised learning has very useful advantages, it cannot achieve what humans do not know already. Since Supervised learning relies on input data with a target label to learn with, if the data is not already categorized supervised learning cannot be used [36]. This is when unsupervised learning comes in handy. Bengio et al. [7] describe unsupervised learning to be similar to supervised learning in which input data lacks a ground truth to compare its prediction with. Rather, it is tasked to find the differences and similarities in the input data [7].

2.4.1 Challenges

Supervised learning is the most researched deep learning problem [29]. There are distinct advantages and disadvantages. One such advantage of supervised learning is quick at fitting a neural network; however, at a disadvantage, supervised learning suffers from overfitting and underfitting [7]. Overfitting can seem good on the surface as the NN appears to be understanding the training data with excellence, but when the NN is fed test data discon-

nected from the training dataset it fails to predict well [7]. Conversely, underfitting is the lack of variety in the training data, resulting in poor generalization and therefore failing to predict well [7].

2.5 Reinforcement Learning

Reinforcement learning is achieved differently from supervised and unsupervised learning. Rather than providing a labelled training dataset, provided by an external expert and measuring the difference of the predicted output and the ground truth or discovering relationships implicitly in a dataset, reinforcement learning takes its inspiration from how biological brains learn, where an agent independently interacts with an environment around it and learns how the environment reacts to the actions taken by the agent [50]. The environment provides the agent with a state X_t as input, then the agent outputs a vector of predictions A and selects the best action $a \in A$, given its policy π. When the action is fed into the environment the state updates and provides the agent with a reward r and the next state X_{t+1} [7].

Reinforcement learning has the largest potential, as it has shown to be able to outperform experts [7], as seen with Google DeepMind's AlphaGo [47] and AlphaGo Zero [48], as well with Atari games [34, 35]. On the other hand, AlphaGo, from DeepMind, uses a mix of both supervised learning and reinforcement learning, where it was pre-trained on expert human games and then later trained using reinforcement learning through self-play [47]. DeepMind's AlphaGo Zero instead was trained purely with reinforcement learning self-play, with absolutely no human knowledge [48]. Interestingly, AlphaGo Zero completely outperformed AlphaGo [48]. Although having high potential comes at a high cost though, reinforcement learning takes significantly longer to train. As well, deciding a good reward function is a difficult task [7]. As the NN learns a different behaviour from previous actions the data distribution will change as well, since the actions taken in the future will be different from the actions in the past, thus resulting in some further difficulty to learn an

environment [34].

Reinforcement learning comes in many algorithms and each with its own uses. The two most common types of policy learning are known as off-policy and on-policy. Off-policy learning is when a model is trained using a policy that is not used to select the action that creates the used state-action pair experiences. Training a model using on-policy learning is when the policy is evaluated using the current policy that is used to select the action for the state-action pair experiences [50].

Reinforcement learning expects an agent and an environment. The agent performs actions and in turn the environment provides observations and rewards [11]. To implement these functions concisely, frameworks and toolkits have been developed. One such toolkit is the gym toolkit, which focuses on episodes that define the agent actions, and environment observations and rewards in a series. Thus, the gym toolkit ensures a concise and practical framework for developing, implementing, and testing environments and models [11].

2.5.1 Main Components

There are a lot of moving parts involved with reinforcement learning, some of the most important parts are policy, reward signal, value function, and sometimes a model. The agent's policy determines how the agent acts at a certain point, given an environment observation. The environment's reward signal, reward for short, is a scalar value provided to the agent for performing an action. The sign and magnitude is dependant on a given state-action pair. For example, the reward for a good state-action pair is positive, while a bad action-state could be negative. The learning agent's main objective is to maximize this signal during an environment episode [50]. Similar to the reward, value functions determine if the current policy is good in the future [46]. Lastly, the model in reinforcement, not to be confused with the resultant trained model, is used for determining the behaviour of an environment and predicting the future state given a state-action pair [50]. Thus, the loss for a reinforcement learning agent is dependant on the reward signal, and the algorithm used

for learning. One such algorithm used is called Deep Q-learning.

In the case of Deep Q-learning, the Q value function is approximated using a NN [46]. In our case, since we are using a variant of Deep Q-learning, our training will be model-free. Wherein the optimal policy will be estimated without predicting the future states in the environment [46]. We discuss Q-learning and its variants below.

2.5.2 Challenges

While there are lots of upsides to reinforcement learning, it has its downfalls as well. Since agents want to maximize the reward received from an environment, then they run into a problem - which decision is the best to make given the environment to maximize reward. An agent has to decide on a trade-off exploit, or explore the environment. In other words, to receive the maximum reward the agent must exploit actions, which it has discovered, to provide a good outcome; however, it cannot find good actions to exploit without exploring. Therefore, the agent must balance out how much exploitation and exploration to perform [50].

2.5.3 Q-learning

Q-learning is a learning algorithm that is used to train an agent how to predict an optimal action in an environment. The Q-learning algorithm has a look-up table $Q(s,a)$ for all known state-action pairs with their associated Q values [55]. Thus, given enough exploration, the optimal Q^* values can be exploited. The Q-learning function to adjust the Q-table is defined as:

$$Q_n(s,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n[r_n + \gamma V_{n-1}(y_n)] & \text{if } x = x_n \text{ and } a = a_n \\ Q_{n-1}(x,a), & \text{otherwise,} \end{cases} \tag{2.7}$$

where

$$V_{n-1}(y) \equiv \max_b \{Q_{n-1}(y,b)\} \tag{2.8}$$

is the best action it can perform on state y. Where X_n is the current state, a_n is the action performed, y_n is the next state, r_n is the received reward, Q_{n-1} is the Q values before the update, and lastly, α_n is the current learning rate. Its assumed that in the second equation that all of the q-values for each state-action pair already exists [55].

2.5.4 Deep Q-learning

Deep Q-learning (DQL), is a reinforcement learning algorithm created by Mnih et al. at Google Deepmind [35]. The goal of DQL was to create an algorithm that can learn a variety of problems using deep NN. They formulated the algorithm to provide a discounted feedback with a focus on maximizing cumulative future rewards through interactions with an environment by replaying the experiences. Additionally, DQL addresses instability and divergence when approximating the Q-function in reinforcement learning, in this case, the Q-function is approximated using a network, known as the Q-network. Mnih et al. [35] proposed randomly sampling experiences from an experience replay memory, in order to reduce sequence correlation and an iterative target update process to address the problem of data distribution problems from updating the Q-values. Updates to the Q-values result in changes in data distributions. Therefore, by creating a copy of the Q-network, the target Q-network can be updated iteratively and can remain stable for the duration of the expected environment episode of C steps, thus resulting in reduced correlations to the target. Keeping the target network stable while the Q-networks' Q-values continue to update is vitally important. Since the target outputs from the Q-network will frequently change with the updates to the Q-network's weights, which can cause divergence to the policy. Instead retrieving targets from the target Q-network that has older weights causes a delay to the target outputs, which makes policy divergence less likely to occur and improves learning stability. The Q-learning function is approximated by the Q-network, and is defined as follows:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi]. \tag{2.9}$$

Equation 2.9 calculates the maximum sum of rewards r_t, received from the environment given the current policy $\pi = P(a|s)$ when the policy acts a upon the state s. The reward at every time step t in the environment episode, which is the accrued experiences before episode termination, is discounted by a factor of γ [35].

2.5.5 Experience Replay

Experience replay is a collection $D_t = \{e_1, ...e_t\}$ of sequenced 5-tuple experiences $e_t = (s_t, a_t, r_t, s_{t+1})$, where s_t is the current observed state, a_t is the action provided to the environment, r_t is the reward for the state-action pair, and s_{t+1} is the next observed state, from an agent's interactions with the environment. Each experience is stored in a dataset $D_t = \{e_1, ..., e_t\}$, collected over the perceived episodes. Randomly sampling experience replay is an integral part of DQL, since it helps remove observation sequence correlation by allowing the policy to be updated from a variety of experiences, and thus flattens the data distribution from the collected experiences [35]. Deep learning algorithms require a loss, or error, to evaluate the performance of the state-action pair. To provide the algorithm with a loss value Mnih et al. [35] propose randomly drawing a mini-batch set of experiences e_t from an experience replay D_t, with samples in the form of $(s, a, r, s') \sim U(D)$, the loss is then calculated using the following equation:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s_{t+1})\ U(D)}\left[\left(r + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_i^-)) - Q(s, a; \theta_i)\right)^2\right] \tag{2.10}$$

again a discounted factor γ is used to determine the importance of future steps or immediate steps. As the next state s_{t+1} with the predicted next action a' from the target Q-network weights denoted θ_i^- and θ_i is the parameters of the Q-network, both at i [35].

2.5.6 Double Q-learning

DQL tends to overestimate the Q-value, which may have a detrimental impact on the NN's action policy and overall ability [52]. Hasselt et al. [52] propose an update to the

DQL target defined as:

$$Y_t^Q = Rt + 1 + \gamma\, Q(S_{t+1}, \underset{a}{argmax} Q(S_{t+1}, a; \theta_t); \theta_t) \tag{2.11}$$

and revises the target NN update to account for the new error proposed in Double Deep Q-learning as follows:

$$Y_t^{DoubleQ} = Rt + 1 + \gamma\, Q(S_{t+1}, \underset{a}{argmax} Q(S_{t+1}, a; \theta_t); \theta_t^{'}) \tag{2.12}$$

the action selection still uses the current NN weights for its policy. Instead the value of the policy is being evaluated by the target NN's weights, to reduce the overestimations occurring in DQL [52].

2.5.7 Dueling Network

One other improvement to DQL is Dueling Deep Q-learning (DuelingDQL). DuelingDQL addresses the occurrence of environments having multiple states with similar Q-values, which causes difficulty in selecting approximate optimal actions [54]. Wang et al. [54] created an architecture dubbed the *dueling architecture*, which has two estimators for the state-value function $V(s)$, and the state-dependent action advantage function $A(s,a)$. Wang et al. [54] achieve this architecture by splitting the output head of the network into two streams, with one being the advantage stream, outputting the state-action advantage values, and the other outputting a value scalar. An advantage of DuelingDQL is that the network is able to identify states in which the action does not impact the environment [54], such as running into a wall repeatedly in a video game. The Q function with dueling is defined as:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\theta,\alpha) \right) \tag{2.13}$$

where $\alpha_t \in \mathcal{A} = \{1, ..., |\mathcal{A}|\}$ is the set of environment actions. In this equation, θ is the parameters for the body of the network. In the original paper their network body was a convolutional neural network [54]; however it could denote generically any network body. α and β represent the split head into the advantage stream and value stream, respectively [54]. At the time of writing Wang et al. [54] found their dueling architecture outperformed the state-of-the-art RL on the Atari 2600 games.

2.6 Application of Convolutional Neural Networks to Optimization Problems

One of the primary objectives of using machine learning is to generically approximate hard problems. As specified above, there is a broad domain of problems that CNNs have been fitted towards and the list is continuing to expand. Some of such problems have been conceptualized with CNNs such as Atari video games from raw pixels [34, 35], board games [47, 48], image classification [31], facial recognition [30], and more. We will be focusing on AlphaGo and AlphaGo Zero primarily as we took inspiration from Silver et al. [47, 48] in using a CNN to approximate a non-image problem.

2.6.1 AlphaGo and AlphaGo Zero

As mentioned before we take some inspirations from AlphaGo and AlphaGo Zero particularly, in the concept of fitting a board representation into a CNN [47, 48]. Silver et al. [47, 48] used the Go board positions of 19x19 to feed into the CNN, with 48 feature maps as channels, therefore resulting in a 19x19x48 image. The channels are used to describe the state of the game extensively, such as the stone colour, turns since the stone was placed, the number of empty adjacent tiles, the number of stones that could be captured, how many personal stones could be captured, the successful captures or escapes, the legal moves, and who played that current state [47].

AlphaGo impressed the world. It is the first computer program to defeat a professional

player at Go [47]. Silver et al. [48] took the program a step further, creating AlphaGo Zero, a variation to AlphaGo, where supervised pre-training was not used, rather they only trained the model using just reinforcement learning. Therefore, it implicitly learned the Go rules by playing against itself. This iteration of AlphaGo ended up defeating the original for a score of 100 - 0 [48].

AlphaGo

AlphaGo is an impressive feat to artificial intelligence, as it broke boundaries that were thought to still lie a decade further in the future [47]. Silver et al. [47] noted that optimal value tree search was not a practical method, as Go has a very large search space. In fact, the search space can be approximated by b^d, where the breadth, b, has an approximate value of 250, the breadth of Go is the number of moves per game state, whilst the depth, d, the possible game states, has approximately 150 possible combinations per move. Thus making the search space for Go 250^{150}. Therefore, Silver et al. [47] called upon deep learning to approximate the massive search tree.

The search space of Go can be pruned by reducing the breadth and depth. To reduce the breadth of the search tree, actions can be sampled using a probability distribution from a certain action policy, $P(a|s)$, where a is the action and s is the current state of the Go board. The depth can be reduced by removing the sub-tree of each state and replacing it with an optimal approximation value defined by $V(s) \approx V^*(s)$. $V^*(s)$ is the optimal value function calculated from an AlphaGo game with perfect information, where this value function determines the outcome of a game [47].

Effective Go programs employ and integrate Monte Carlo tree search (MCTS), combined with Monte Carlo rollouts, to estimate the value of the next state of a certain action. The deeper the rollout is calculated, the more accurate the estimated action value is, and so the longer the computation time given the closer value evaluation converges to the optimal value. Thus given enough computation time the program converges on optimal play. Paired

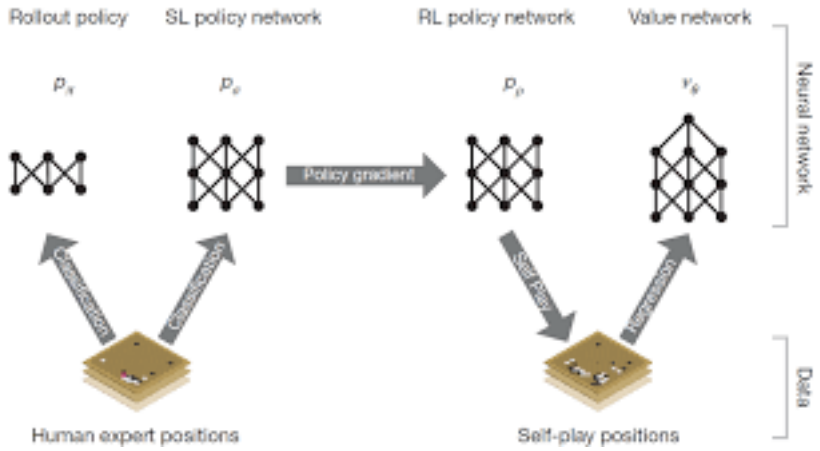

Figure 2.2: AlphaGo training pipeline and neural network architecture [47].

with human predicting policy models the best Go programs can go toe-to-toe with amateur players. Silver et al. [47] proposed that replacing the Monte Carlo rollouts using a value neural network, a NN with a single output that estimates the value of the given state input, drastically reduces the required computing time by 15,000-fold. Furthermore, instead of simply using probability distributions Silver et al. [47] use a NN to reduce the breadth of the search tree, by sampling a learned action probability distribution.

Silver et al. [47] use a series of machine learning techniques, some of which we also employ for our project, starting with supervised learning and fine-tuning using reinforcement learning [47], as shown in Figure 2.2. During supervised learning Silver et al. [47] use expert human moves as the ground truth given the board positions as input for both training and testing of their policy NN; intending to maximize the likelihood usage of humans moves during gameplay at any given state. By using a 13-layer CNN, which alternates between convolutional and ReLU layers outputting a probability distribution of legal actions using a Softmax layer, had an improved accuracy of raw board positioning to 55.7% and test-set accuracy of 57.0% from the state-of-art accuracy of 44.4% [47].

The supervised learning pre-trained model is then further tuned using reinforcement learning, in which the focus is shifted on training both the policy NN and the value NN. While the supervised learning training pipeline used human expert moves, the reinforce-

ment learning pipeline uses self-play to collect the state-action pairs, optimizing the weights of the CNN every game time step. To address the overfitting problem, the policy model is pitted against a random prior iteration of itself per match. The team for AlphaGo uses a sparse reward function, where all moves performed are evaluated with a reward of 0, except at the two terminal states, either victory, with a reward of +1, or defeat, with a reward of -1 [47].

AlphaGo Zero

AlphaGo had many subsequent versions, AlphaGo Lee, AlphaGo Fan, AlphaGo Master, and the most interesting one, AlphaGo Zero. Unlike past variants, AlphaGo Zero is not pre-trained using example human expert game knowledge, hence the term "Zero" in the name. Supervised learning attempts to train a NN to replicate human decisions, whilst reinforcement learning seeks to have the NN discover an expert policy on its own. Reinforcement learning is a good alternative as supervised learning datasets can be costly, unreliable, or non-existent. As well, supervised learning has a ceiling that a NN cannot fully surpass, reinforcement learning is able to far surpass humans [48].

Silver et al. [48] found that AlphaGo Zero had significantly improved abilities, defeating all previous iterations of AlphaGo, including their most powerful version, at the time, AlphaGo Master. In a match of 100 games, AlphaGo Zero defeated AlphaGo Master 89% of the games. Scoring an Elo rating, which is a relative skill level for predicting game outcomes, of 5,185, and AlphaGo Zero defeats the reigning champ AlphaGo Master with an Elo rating of 4,858 [48].

AlphaGo Zero had significant revisions to reach these feats. First, as mentioned above, there is no supervised learning used with AlphaGoZero, only self-play reinforcement learning. Additionally, the network input features have been extremely simplified, where only the board positions of the black and white stones are used. Silver et al. [48] solely relied on a policy network. Lastly, they simplified their tree search algorithm to only use the policy

network and completely removed Monte Carlo rollouts [48].

2.6.2 Atari Games

Video games require extensive knowledge of the inputs and actions taken per game, which can be difficult to obtain. Atari 2600 had many popular game titles, and many programs designed to play them [34]. Mnih et al. [34] created a deep learning model that was superior for 6 games. The model used was a generic CNN model to play Atari games, where the model was taught to play 7 unique Atari games. The model managed to surpass previous approaches on Beam Rider, Breakout, Enduro, Pong, Q*bert, and Seaquest, it also managed to surpass human experts at playing Breakout, Enduro, and Pong. In fact, this model was the first CNN model to learn control policies from only visual input using reinforcement learning. Mnih et al. [34] designed the model to take the current state of the game as raw pixels as input. The model was trained purely with reinforcement learning, specifically with DQL, and experience replay memory that would randomly sample data to avoid variance. The model interacts directly with an Atari 2600 emulator, by passing in actions, given the current policy, and then evaluated and rewarded on the next state from the action. The reward is fully based on the score of the current game, which has plenty of variance per game for game score evaluations [34].

The raw pixels underwent some changes to speed up the training of the model, as well as fit into the GPU implementation, which required a square matrix input [34]. Minh et al. [34] transformed the video game image into greyscale, downsampled, and then cropped the image into an 84x84 image. This input was fed through a 6 layered network, with an output for each legal action. By directly tying the game's score with the reward given, the model naturally wants to maximize the score of the game in order to maximize the reward received. Mnih et al. [34] introduced an important and novel technique to reinforcement learning, Q-learning, with stochastic mini-batch updates and experience replay.

2.6.3 OpenAI

We have only scratched the surface of applications of convolutional neural networks and machine learning. There are plenty of incredible projects and research from many groups around the world. Impressive models have come from a research team called OpenAI, a team that works on a wide array of machine learning projects, some of which use convolutional neural networks. Some of their projects are: DALL-E, which creates images from text [42], a pair of neural networks for solving a Rubik's Cube with a single robot hand [2], and to learn human-like dexterity [4]. OpenAI Five, a complex model, which is able to play the video game Dota 2 at professional levels, is one of the most interesting projects they have created [8].

2.7 Summary

We have covered a variety of topics related to our problem with respect to optimization problems, approximation algorithms, convolutional neural networks, deep learning, supervised learning, reinforcement learning, and applications of convolutional neural networks. In the following chapters, we plan to discuss our findings. In Chapter 3 we propose a heuristic approximation algorithm for the dragon boat partition problem and in Chapter 4 where we use convolutional neural networks, trained with both supervised learning and reinforcement learning, to approximate the same problem.

Chapter 3

Heuristic Generation and Optimizing of the Dragon Boat Partition Problem

In this chapter, we will explore the Dragon Boat Partition problem. We will discuss how we have designed our dragon boat environment, how the data is collected for supervised learning, and the approaches we took to create a heuristic approach. We designed two versions - the heuristic sorter and heuristic agent. In the heuristic sorter, we have treated the algorithm simply as a typical algorithm with full access to the dragon boat environment. In the heuristic agent, we designed an algorithm that behaves similar to a reinforcement learning agent, having to directly interact with an environment.

This chapter is organized as follows. In Section 3.1 we will contextualize the dragon boat problem, accompanied by our goals. Section 3.2 will detail the assumptions we have made in our experiments and the results on our findings. Next in Section 3.3, we discuss the dragon boat partition problem environment and its operations. We then discuss, in Section 3.4, in detail, our heuristic approach and how we have utilized it and its effects. Section 3.5 will provide details on our experiments and comparisons. Finally, we present our summary, concerns, and future work in Section 3.6.

3.1 Introduction

Computationally, optimization problems are difficult to approximate and as problem size increases, the evaluation of the problem could become more complex. Thus, given limited available research in the dragon boat partition problem, we formulate our own al-

gorithm to approximate the optimal solution to it. Albeit, this is not an algorithm with a rigorous proof, as our main goal is to apply deep learning to the problem and thus is simply a practical application.

The algorithm we designed is loosely inspired by the rules and preferences by Stickels [49]. However, other online sources have different preferences, such as having exactly the same weight, or even heavier in the back of the boat [16], in contrast to heavier in the front [49]. For consistency, we used the rules and references by Stickels [49] as our original source of information.

The goal of this chapter is to build a domain that our algorithm can interact with. Towards this, we establish our assumptions, characteristics, constraints, and environment. We formally describe the environment's rules, states, and actions. Then, we discuss our method of generating and gathering data, as it is hard to find an existing partition dataset for dragon boats.

3.2 Assumptions

We have made some significant assumptions in our work since we were been constrained by time to explore all possibilities. Some assumptions made in our heuristic approach we also use with our deep learning approach, to keep consistency across the experiments and expectations.

In our approaches, three fundamental assumptions were made. The first assumption is with the dragon boat seating arrangements, where the number of participants allowed on the boat and their positions was constrained. The second assumption involves certain characteristics and attributes of the participants in the dragon boat. Lastly, we assume that our constraints can be relaxed for better optimal approximations.

3.2.1 Dragon Boat Team Characteristics

Normally dragon boat teams have twenty-two participants in total, consisting of twenty rowers, one steer, and one drummer [23]. We do not take into consideration bench participants in our approaches. Therefore we assume that the participants provided is the complete list of participants and will be the only data manipulated.

Since hard problems take increasingly more time with respect to their size, we assume that our problem would be constrained to ten participants, eight participants are rowers, one is a steer, and one is a drummer. However, we also assume that only the rowers can change and affect the boat after generation, while the steer and drummer stay constant. This assumption is also used in Chapter 4 since we are comparing our heuristic approach against our deep learning approach. It is necessary to constrain the dataset size, in part due to being limited by the hardware available to us, as it will heavily affect our training time. We also assume that there are no fixed positions, in which participants in that position cannot be moved.

3.2.2 Participant Characteristics and Attributes

We make a rough estimate of human-like characteristics and attributes of a participant. Therefore, we assume that each participant weighs between 110lbs to 270lbs, with each weight rounded to an integer number. We also assume that most of the participants are able to row on either side, since having many participants that are only able to row left or right makes the problem too easy to approximate. Additionally, we assume that if the rower is strictly left-handed, they are only able to row on their handed side and being on the opposite side is infeasible and thus not allowed, and vice-versa for rowing on the right. Furthermore, we assume that participant height does not affect the environment. Lastly, we assume that no participants have a preferred position to be positioned in.

3.2.3 Constraints and Relaxation

One of the rules we ignore for balancing a dragon boat team, is the rule in which the heaviest participants should be positioned in the middle of the boat, while the lighter the participant the further from the middle they are positioned relative to their collective group. We call this the weight gradient rule. We fully relax the perfect weight discrepancy rule by allowing a small margin of error from the optimal weight.

3.2.4 Weight Values

Consider a 2D structure, analogous to a 2D plane, where we balance our participants on the positive and negative values of our axes x, and y. We defined each grouping to an axis direction, where the left group is placed on the positive x-axis, and conversely, the right group is placed on the negative x-axis. The front group is placed upon the positive y-axis, while the back group is placed upon the negative y-axis. As such, when a dragon boat has a positive left-right weight this means that the dragon boat is heavier on the left-hand side and vice-versa for a negative left-right weight. This same logic applies to the front-back weight being positive and negative. For example, let there be two rowers in the left group, where the total weight of the left group is 220lbs. This would place our current x position at +220. Next, consider a rower that weighs 200lbs and is placed in the right group, then our current x position would be +20 since the right group is placed on the negative x-axis.

3.3 Dragon Boat Environment

We implemented our dragon boat partition problem as an environment, which we call the dragon boat environment, using a toolkit from OpenAI called Gym [11]. The goal of the Gym toolkit is to standardize the development of environments across developers so that sharing environments is simple and can be easily plugged into a learning agent.[2] As such, in order to follow the standard in Gym, we designed and implemented the three required

[2]The OpenAI Gym documentation retrieved from: https://gym.openai.com/docs/.

functions, namely, ***step***, ***reset***, and ***render***. These functions are mostly used as an endpoint API to the agent. An API is an intermediary interface that allows separate applications to communicate [3]. This enables generic communication to environments so that any agent can use our environment. The heuristic agent utilizes these functions to emulate a reinforcement learning agent. We will further discuss these functions in Section 4.2.1.

3.3.1 Defining The Dragon Boat Features

As discussed previously in Section 1.2.2 we are focusing on optimizing the dragon boat partition problem. The objective of the dragon boat partition problem is to partition the participants in the boat where each participant must be placed based on each rower's dominant hand, the summed participant weights of the left-hand and right-hand side must be equal or as close to equal as possible, and the summed participant weight of the front-half must be no more than 30 lbs heavier than the summed participant weight of the back half of the boat. We refer to *features* as a set of rowing participants in a dragon boat environment.

We define left-handedness as a participant's ability to row effectively with their left hand and refer to them as *left-rowers*, and similarly for right-handedness (*right-rowers*). A participant may also have the ability to row both left-handed and right-handed and we refer to them as *either-rowers*.

Left-handed rowing ability is a Boolean value denoted $p_l \in \{0,1\}$ and similarly $p_r \in \{0,1\}$ denotes the right-handed rowing ability. As such, if a participant has a $p_l = 1$ and $p_r = 1$, then they are a *either-rower*. Finally, a participant's weight is a positive integer number in pounds (lbs) denoted by $p_w \in \mathbb{Z}^+$,

The dragon boat partition problem can now be formally defined. Assume there is a set P with all possible participants, where participant $p \in P$ is a 3-tuple (p_w, p_l, p_r) and given a list of participants $T \subseteq \{p \mid p \in P\}$, let $L \subset T$ where $|L| = |T|/2$ and $\forall p \in L, p_l = 1$ denote the left group, and let $R \subset T$ where $|R| = |T|/2$ and $\forall p \in R, p_r = 1$ denote the right group. Similarly, let $F \subset T$, where $|F| = |T|/2$, denote the front group and $B \subset T$,

[3] API definition retrieved from: https://www.mulesoft.com/resources/api/what-is-an-api.

where $|B| = |T|/2$, denote the back group.Let $L_w = \sum p_w \ \forall p \in L$ be the summed participant weight for the left-group, $R_w = \sum p_w \ \forall p \in R$ be the summed participant weight for the right-group, $F_w = \sum p_w \ \forall p \in F$ be the summed participant weight for the front-group, and $B_w = \sum p_w \ \forall p \in B$ be the summed participant weight for the back-group. Let V_{lr}, V_{fb} represent the left-right and front-back optimal discrepancy relaxation values respectively. $W_{lr} - |L_w - R_w|$ is the left-right discrepancy. $W_{fb} = |F_w - B_w|$ is the front-back discrepancy. W^*_{lr} is the optimal weight discrepancy for the left and right. Lastly, W^*_{fb} is the optimal weight discrepancy for the front and back. Find a partition where $|L| = |R|$, $|F| = |B|$, $(L \cap R) = \emptyset$, $(F \cap B) = \emptyset$, $L \cap (F \cup B) = L$, $R \cap (F \cup B) = R$, $(F \cap L) \cup (F \cap R) = F$, $(B \cap L) + (B \cap R) = B$ in which W_{lr}, is close to W^*_{lr}, such that $W_{lr} \geq W^*_{lr} - V_{lr}$ and $W_{lr} \leq W^*_{lr} + V_{lr}$, and W_{fb} is close to W^*_{fb} such that $F_w > B_w$, $W_{fb} \geq W^*_{fb} - V_{fb}$, and $W_{fb} \leq W^*_{fb} + V_{fb}$, where $W^*_{lr} = 0$ and $W^*_{fb} = 30$.

In simpler terms, a dragon boat has four groups, the left-side, right-side, front-half, and back-half. Each group has a cardinality of half of the number of participants in the dragon boat. The left-side can only have rowers that can row on the left or either side, while the right-side can only have rowers that can row on the right or either side. The left-right discrepancy is defined as the difference between the sum of the left-side's participant weights and the sum of the right-side's participants' weights. Similarly, the front-back discrepancy is defined as the difference between the sum of the front-half's participant weights and the sum of the back-half's participants' weights. The rowers on the left-side are disconnected from the right-side group but can exist in either the front-half or back-half, depending on the seat they are assigned to. This logic applies to the right-side as well. The same concept applies to the front-half and back-half, as any participant assigned to the front-half cannot exist in the back-half, but can exist in either the left-side or the right-side. Furthermore, for the dragon boat to be balanced, the left-right discrepancy must be greater than or equal to the left-right optimal weight, typically 0lbs, within the range of the left-right relaxation. The same rules mostly apply to the front and back, where the front-back

discrepancy must be as close to the front-back optimal weight, typically 30lbs heavier in the front, within the range of the front-back relaxation.

We can summarize each of these conditions into a list of rules, with an addition of the Weight Gradient Rule:

1. Handed Rule: There cannot be any left-rowers on the right-side, nor any right-rowers on the left-side.

2. Left-Right Weight Rule: The left-right discrepancy must be close to the left-right optimal weight within the range of the left-right relaxation. Formally, $W_{lr} \geq W^*_{lr} - V_{lr}$ and $W_{lr} \leq W^*_{lr} + V_{lr}$.

3. Front-Back Weight Rule: The front-back discrepancy be as close to the front-back optimal weight within the range of the front-back relaxation, where the front-half is heavier than the back-half. $W_{fb} \geq W^*_{fb} - V_{fb}$, $W_{fb} \leq W_{fb} + V_{fb}$ and $F_w > B_w$.

4. Weight Gradient Rule: The rowing participants should be graded by heaviest to lightest from the center of the board outward.

The current environment episode terminates when the first three rules are fulfilled or when the agent stops. While we do not enforce the last rule in our environment, we include it to be holistic.

3.3.2 Input Features

Our environment observation space is organized into a 3-dimensional array. The first dimension denotes the channels, which are the properties of the participant, the second dimension is the rows, and the third dimension is the columns. We format the last three channels as 1-hot encoded. The channels are further shown in Table 3.1 and Figure 3.1.

Table 3.1: Input features of our environment.

Feature	Data type	Description
Weight	Integer	The weight of the participant
Left	Boolean	The ability to row left handed
Right	Boolean	The ability to row right handed
Either	Boolean	The ability to row with Left $\wedge$ Right hand

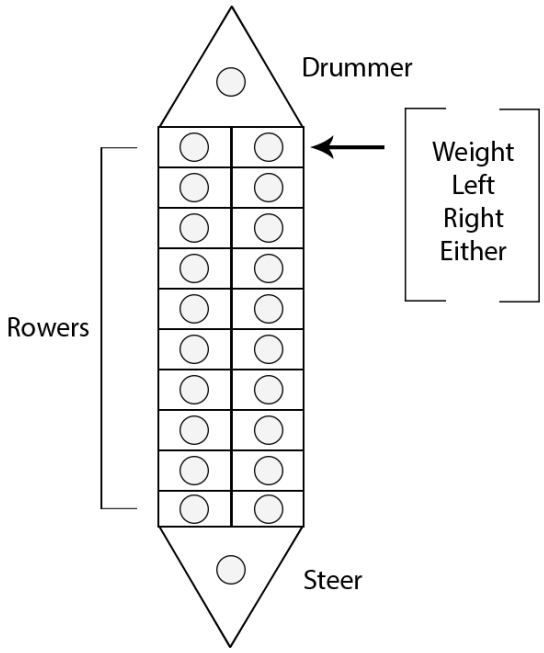

Figure 3.1: Dragon boat seating features.

3.3.3 Actions

The action space A in the dragon boat environment is split into two discrete spaces, the left-right action space, A_{lr} and the front-back action space A_{fb}. Thus, our action space $A = A_{lr} + A_{fb}$. An action space is the set of actions that can be performed in an environment. Each action is mapped to a positional swap. A positional swap corresponds to two positions in the dragon boat state, and when a positional swap is performed the two rowers in those positions are swapped. The number of actions in the action space is calculated based on the number of rowers n in the environment, where n is an even number. The left-right action space has an action for swapping each rower on the left side, with each rower on the right

side. Therefore, the left-right action space is calculated by the following equation:

$$A_{lr} = \left(\frac{n}{2}\right)^2 \tag{3.1}$$

We ignore swapping rowers on the same side as this would not affect the left-right balancing. For example, if we have 8 rowers, the number of left-right actions would be 16.

The front-back action space has an action for swapping each front-left rower with each back-left rower, and for each front-right rower with each back-right rower. Thus the equation to calculate the number of front-back actions in the front-back action space is:

$$A_{fb} = 2\left(\frac{n}{4}\right)^2 \tag{3.2}$$

In this action space we do not allow swapping between front-left and back-right, or with back-left and front-right, since this would break our left-right rule when balanced. With 8 rowers, there will be 8 front-back actions. Therefore, in our example, there is a total of 24 actions.

It is important to note that when two rowers are swapped, their weight difference has a double impact on the weight discrepancy they are affecting. Consider the following example: assume there is one rower on the left-side and one rower on the right-side, let $W_{lr} = -30$, $L_w = 100$ and $R_w = 130$. We know that the right-side is heavier than the left-side. Thus, the difference between the only two rowers is 30lb. Therefore, when we swap the two rowers $W_{lr} = 30$, which is 2x the amount of the difference. This occurs because when swapping the rower on the left-side with the rower on right-side, the left-side gains 30lb, while the right-side loses 30lb with a total change of 60lb.

3.3.4 Dragon Boat Generation

Given our definition of the dragon boat environment, we are able to demonstrate our approach of generating data in which we are to test on. As mentioned earlier we generate

the data to be used. To handle biased generation, we tried to keep the generation as broad, but as reasonable as possible. The algorithm of our generation goes as follows:

```
LET EXTRA_WEIGHT = 1
LET RANDOM_WEIGHT_MIN = 1
LET num_left_rowers = CALL random WITH 0, n/2
LET num_right_rowers = CALL random WITH 0, n/2
LET num_rowers_on_left = 0
LET num_rowers_on_right = 0
FOR number of participants to generate
    // The number of left_rowers and right_rowers is randomly
    // generated for variance
    IF num_left_rowers > 0
        LET left_rower = CALL Generate WITH left-rower
        DECREMENT num_left_rowers
    ELSE
        LET left_rower = CALL Generate WITH either-rower
    END IF

    PUSH left_rower ONTO L
    INCREMENT num_rowers_on_left

    // The rower is in the front if they number of rowers on the
    // left is less than half of the number of seats on the boat.
    IF num_rowers_on_left <= n/2
        PUSH left_rower ONTO F
    ELSE
        PUSH left_rower ONTO B
    END IF

    IF num_right_rowers > 0
        LET right_rower = CALL Generate WITH right rower
        DECREMENT num_right_rowers
    ELSE
        LET right_rower = CALL Generate WITH either rower
    END IF

    PUSH right_rower ONTO R
    INCREMENT num_rowers_on_right

    IF num_rowers_on_right <= n/2
        PUSH right_rower ONTO F
    ELSE
        PUSH right_rower ONTO B
    END IF
END FOR

WHILE W_lr != W*_lr
    LET diff = CALL abs WITH W_lr - W*_lr
    LET rand_weight = CALL random WITH RANDOM_WEIGHT_MIN, diff
    IF W_lr > W*_lr
        // left-side is heavier, so add weight to the right-group
        LET rand_rower = GET random rower on right-side
        ADD rand_weight to rand_rower
```

```
    ELSE
        // right-side is heavier, so add weight to the left-group
        LET rand_rower = GET random rower on left-side
        ADD rand_weight to rand_rower
    END IF
END WHILE

WHILE W_{fb} != W*_{fb}
    LET abs_goal_diff = CALL abs WITH W*_{fb} - W_{fb}
    // half the weight to be added to maintain W_{lr} = W*_{lr}
    LET weight_to_add = (CALL random with RAND_WEIGHT_MIN,
    abs_goal_diff) / 2

    IF W_{fb} > W*_{fb}
        // front-half is heavier, so add weight
        LET rand_rower = GET random rower in back group
        ADD weight_to_add to rand_rower
    ELSE
        // back-half is heavier, so add weight
        LET rand_rower = GET random rower in front group
        ADD weight_to_add to rand_rower
    END IF
    ADD weight_to_add to rower in the same row, opposite column to
    rand_rower

    IF W_{fb} == W*_{fb} + 1
        // front-half is heavier add weight to the back-half
        ADD EXTRA_WEIGHT to steersman
    ELSE
        ADD EXTRA_WEIGHT to drummer
    END IF
END WHILE

IF make_odd_configuration
   LET rand_rower = GET random rower
   ADD EXTRA_WEIGHT to rand_rower
END IF
```

Algorithm 3.1: Dragon Boat Participant Generation.

In other words, to generate a dragon boat environment we first need to determine the number of rowers in the dragon boat environment. We always assume the drummer and steer are ignored, and only consider the number of left-handed, right-handed, and either-handed rowers only. Two rowers are randomly generated at the same time, as a way to keep the left-right weight steady. To ensure that there are the required amount of left-handed rowers and right-handed rowers we generate left-handed and right-handed rowers first, and then only generate either-handed rowers to fill in the rest of the positions available.

Consequently, any rowers appended to the left and right groups will be added to the front and back groups with respect to their position in the boat.

After the participants of the entire boat have been generated, we try to balance out the left-right weight, by repeatedly adding small to moderate amounts of weight to a randomly selected participant, on the side that is causing imbalance, until the left-right discrepancy is equal to the optimal weight. By this point, we have completed two of the three rules required of the dragon boat environment, namely that only left-handed rowers can be on the left-side and right-handed rowers on the right-side, and the left-right discrepancy is equal to the optimal weight.

Next, we move into balancing the front-back weight. Since the front-half and back-half groups are coupled to the left-side and the right-side groups; changes made to the left-side and right-side groups will heavily affect the front-half and the back-half groups and vise versa. Any changes to balance the front-back weight will affect the left-right weight balances as well. We proceed with a similar fashion with the left-right balance, where we add small to moderate amounts of weight to a randomly selected participant on the side that is out of equilibrium until the front-back balance reaches the goal. This process has an extra stipulation that any changes made to the front-back weight must be equally distributed in the left-side and right-side weights.

As for rule 4, as we mentioned above, is a simple sort problem from heaviest to lightest. Since this rule does not affect the overall balance of the boat we just ignore it. Generally, rule 4 is only important when we want to holistically approximate the dragon boat partition problem.

We use our dragon boat generation algorithm for the creation of our dataset as well as for our testing our algorithm. Our tests consist of 10,000 randomly generated dragon boats and are provided to each of our approaches.

3.3.5 Dragon Boat Episodes

Now that we have defined the dragon boat environment, we can discuss what constitutes a gym episode. An episode, in our problem, is defined as each state from when the dragon boat environment has been reset until it reaches a terminal state. The current state progresses by an agent performing actions until an action that results in a terminal state occurs. We define resetting a dragon boat environment as the rowers being randomly generated, then randomly shuffled. A terminal state is when the environment enters a state where the 3 required rules are fulfilled and the environment stops. In the dragon boat environment a terminal state occurs when the $W^*_{lr} = 0$ within the range of V_{lr}, and $W^*_{fb} = 30$ within the range of V_{fb}.

3.4 Two Heuristic-Based Approaches

Both of our heuristic approaches share the same balancing logic but do have a couple of differences. The heuristic sorter takes in the entire input matrix and outputs a modified matrix, while the heuristic agent will emulate how our deep learning agent behaves, taking the environment observation state and outputting an action to the environment. As well, our heuristic sorter fulfills rule 4, however, our heuristic agent does not.

Since our focus is on applying deep learning we designed our heuristic agent to be able to collect data for pre-training. The design is not perfect, but achieves a heuristic approximate, with decent perfect partition accuracy for the left-right and front-back discrepancies. Recall from Section 1.2.1, a perfect partition is defined where the discrepancy is E=0 or E=1.

3.4.1 Algorithm Design

The design for our algorithm follows a repeating logic, in which if it fails to find a suitable action given the current state, it will begin searching with a wider range of discrepancy by relaxing the search size z. The search size is how far from the optimal weight

difference the algorithm will accept. This will repeat until a suitable action is found up to a certain maximum search size z_m. Upon reaching this z_m, our program will revert to the best-known previous iteration and stop. Our heuristic has four distinct stages: the cleaning phase, the left-right partition phase, the front-back partition phase, and the middle partition. Our heuristic agent purposefully ignores the final stage, which we discuss below.

```
/* Heuristic Algorithm */
Step 1: Clean Phase
    WHILE misplaced-participants breaking the Handedness rule:
        IF left-handed participant on right-side
            GET left-handed participant on the right-side AS p1
            GET misplaced right-handed participant or either-handed
     participant AS p2
            CALL Swap WITH p1 and p2
        END IF

        IF right-handed participant on left-side
            GET right-handed participant on the left-side AS p1
            GET misplaced left-handed participant or either-handed
    participant AS p2
            CALL Swap WITH p1 and p2
        END IF
    END WHILE

Step 2: Left-Right Partition Phase
    WHILE W_lr >= W*_lr - V_lr AND W_lr <= W*_lr + V_lr AND search_size !=
    search_size_max
        FOR EACH either-handed participant IN non-offending side
            GET next lightest participant IN the non-offending side
     AS P^x
            FOR EACH either-handed participant IN offending side
                GET next heaviest participant IN offending side AS
    P^y
                // find the best discrepancy change
                // for the current input
                LET difference = P^y_w - P^x_w
                IF difference is closer to W*_lr than
    current_difference
                    current_difference = difference

                    // stop if a current swap will result in a
                    // perfect discrepancy
                    IF current_difference == left-right goal weight
                        break
                    END IF
                END IF
            END WHILE
        END WHILE

        IF P^y and P^x swap has occurred in SwapMemory
            // If the same two participants have been swapped
```

```
40              // repeatedly in the left-right, then increase the
    search
41              INCREMENT search_size
42          ELSE
43              CALL Swap WITH P^y and P^x
44
45              // store the state-action pair in memory
46              CALL SaveSwapMemory WITH P^y and P^x
47          END IF
48      END WHILE
49
50      IF z == z_m AND W_lr != W*_lr
51          // No perfect discrepancy can be found.
52          // Recall to last best left-right state
53          BEGIN Reversion Phase
54
55          // After reverting to best known phase begin the next phase
56          BEGIN Front-Back Partition Phase
57      END IF
58      IF W_fb == W*_fb
59          BEGIN Front-Back Partition Phase
60      END IF
61
62  Step 3: Front-Back Partition Phase
63      WHILE W_fb >= W*_fb - V_fb AND W_fb <= W*_fb + V_fb AND search_size !=
    search_size_max
64          FOR EACH participant IN non-offending side
65              GET next lightest participant on the non-offending side
     AS P^x
66              FOR EACH participant IN offending side
67                  GET next heaviest participant on offending side AS
    P^y
68                  LET difference = P^y_w - P^x_w
69                  IF difference is close to W*{fb} than
    current_difference
70                      current_difference = difference
71
72                      // stop front-back partition phase
73                      // if the current
74                      // swap will result in a perfect discrepancy
75                      IF current_discrepancy == front-back goal
    weight
76                          break
77                      END IF
78                  END IF
79              END WHILE
80          END WHILE
81
82          IF P^y and P^x swap has occurred in swap_memory
83              // If the same two participants have been swapped
84              // repeatedly in the front-back, then increase the
    search
85              INCREMENT search_size
86          ELSE
```

```
            CALL Swap WITH P^y and P^x

            // store the state-action pair in memory
            CALL SaveSwapMemory WITH P^y and P^x
        END IF
    END WHILE

    IF z == z_m AND W_fb != W*_fb
        // No perfect discrepancy can be found.
        // Recall to last best front-back state
        BEGIN Reversion Phase

        // After reverting to best known phase begin the next phase
        BEGIN Weight Gradient Partition Phase OR STOP
    END IF
    IF W_fb == W*fb
        BEGIN Weight Gradient Partition Phase OR STOP
    END IF

Step 4: Weight Gradient Partition Phase (optional)
    CALL QuickSort with front-left group
    CALL QuickSort with front-right group
    CALL Reverse QuickSort with back-left group
    CALL Reverse QuickSort with back-right group

 Step 5: Reversion Phase (if needed)
    WHILE swap_memory contains elements
        POP most recent tuple (P^x, P^y)
        CALL Swap with P^x, P^y
```

Algorithm 3.2: Dragon Boat Heuristic Partitioning Algorithm.

The above is a top-down view of our algorithm. Next, we will define each of our phases in more detail, starting from the Cleaning Phase, and moving forward to the Left-Right Partition Phase, Front-Back Partition Phase, Weight Gradient Partition Phase, and the Reversion Phase.

A few key terms must be defined first as they are used in our algorithm for completeness.

- Misplaced participant: a participant who is a left-handed rower is on the right-side or a right-handed rower on the left-side.
- Offending side: occurs when a side from the four groups is breaking one of the rules from Section 3.3.1.
- Non-offending side: is the side that is not breaking any rules at the current process in

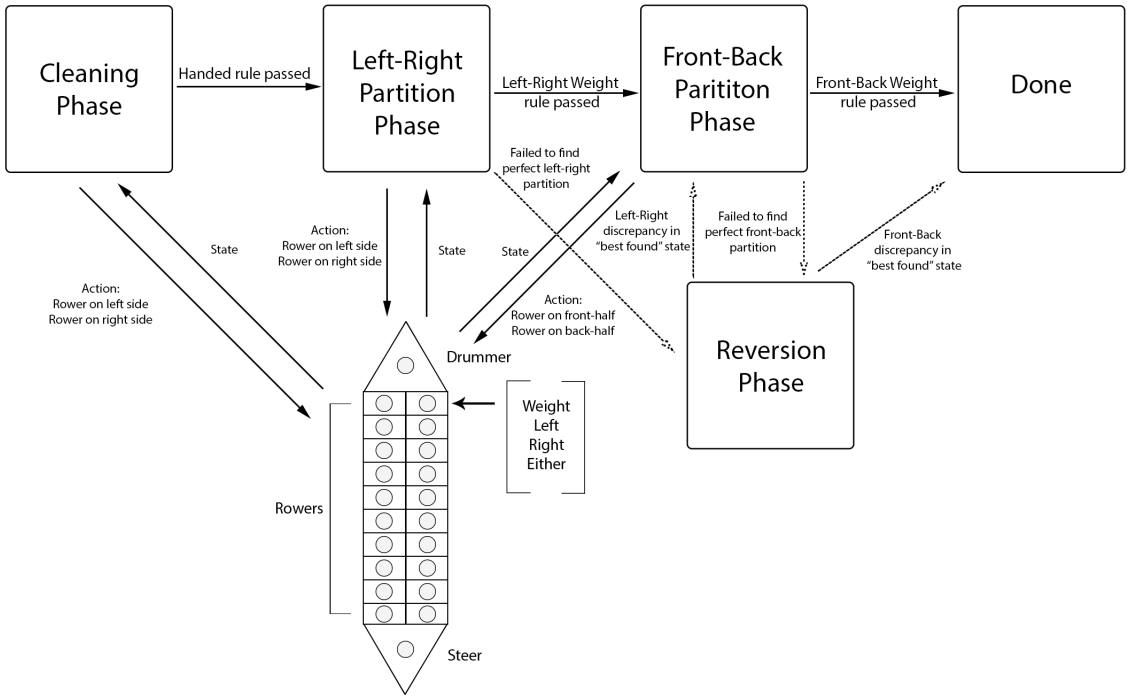

Figure 3.2: Top-down view of the heuristic agent acting on the dragon boat environment.

the algorithm.

- Swap Memory: stores each swap's state, action, and weights. When it is called upon to revert, it acts like a reverse stack, where the most recent memories are the first to be removed.

Cleaning Phase

The Cleaning Phase has only one objective, to satisfy rule 1: The Handed Rule. This phase finds any misplaced participant on the offending side and swaps them with a rower on the non-offending side. It is important to note that although it is called the non-offending side, it can be breaking a rule, however, the algorithm has not identified it yet. Importantly, this phase attempts to speed up the process by finding a misplaced participant on the offending side, and swap with a misplaced participant who is also violating the rule on the non-offending side, i.e. a left-handed participant on the right-side would be swapped with a right-handed participant on the left-side. If there is no such participant, the offending

participant will simply be swapped with an either-handed participant. This phase continues until there are no more misplaced participants.

This phase assumes there are at most an equal number of left-handed participants to the number of positions on the left-side, and an equal number of right-handed rowers to the number of positions on the right-side. Thus, it is assumed that there should be $n/2$ positions on each side, and there are at most $n/2$ left-handed participants, and $n/2$ right-handed participants. Therefore, if there were $(n/2)+1$ left-handed participants this Phase would break, as there would be no positions where each left-handed participant could be on the left-hand side.

Left-Right Partition Phase

After the Cleaning Phase, The Left-Right Partition Phase has a significantly simpler task. We can now ignore the left-handed and right-handed rowers, as they are on their proper respective sides. During this phase, the only eligible participants for swapping positions are either-handed rowers. If there are no either-handed rowers to swap, then this phase will move to the next phase. Otherwise, our algorithm searches for an eligible swap that will minimize the left-right discrepancy. The algorithm achieves this by first deciding which is the offending-side. Next, it performs, in the worst case, a $O(n^2)$ search to find any two rowers whose swapping minimizes the left-right discrepancy. First, the algorithm will retrieve the lightest rower on the non-offending-side, and then the algorithm retrieves the heaviest rower on the offending-side and finds the difference in their weights. If the current difference results in a perfect discrepancy it is selected. Otherwise, the searching process repeats until each rower has been checked. If there is no swap that results in a perfect discrepancy then the best minimized discrepancy seen is selected. Each swap and current state is stored in the swap memory, and if it has occurred before, then the search size is incremented and the search for the minimized discrepancy is performed again which results in a larger acceptable difference range. This repeats until the perfect or near-perfect left-

right partition is found before proceeding to the front-back partition phase. Otherwise, the algorithm continues into the reversion phase, and restores the last best known configuration, and then continues to the next phase.

Each swap performed is stored in the swap memory in the case when no acceptable solution can be found. At this moment the past must be recalled and reapplied. Such cases occur where the algorithm is not able to find a perfect partition, i.e. $E = 0$ or $E = 1$. The reversion phase is covered further in the Reversion Phase section.

Front-Back Partition Phase

The Front-Back Partition Phase follows directly after the Left-Right Partition Phase and is very similar. The method of selecting which two rowers to swap is the same, except the groups used are the front-half and back-half groups. The only real difference in these two phases is that in the Front-Back Partition Phase participants must remain on the left-side or right-side they were assigned in the previous phase. If there is a perfect selection to swap between the front-left and the back-right is ignored, as this will heavily affect the left-right balance rule. Therefore, only participants positioned in the front-left and back-left can be swapped, and the same logic applies to the front-right and back-right.

This phase also saves each swap into memory and can move into the Revision Phase in the case where no acceptable solution is found. The phase ends after reverting or if a perfect partition is achieved. Finally, when this phase ends the algorithm moves into the Weight Gradient Phase, the final stage.

Weight Gradient Partition Phase

The Weight Gradient Partition Phase is by far the simplest of the Partition phases, as this phase simply runs the Quicksort algorithm [45] on each of the four groups. In particular, the front-left, and front-right are sorted from lightest to heaviest participants, while the back-left and back-right are sorted in the reverse order.

Reversion Phase

The reversion phase uses a simple memory, where the states and actions of all swaps performed to reach this state are saved. When the Reversion Phase occurs it recalls each previous swap, in reverse order, until the best configuration for the given phase is found.

3.4.2 Heuristic Sorter Variant

We wrote two variants of our heuristic approach. Our first variant *Heuristic Sorter* is a classical algorithm, where it receives an input and then performs the tasks required of it. This variant employs each phase, and it terminates when it has determined it has fulfilled all of the required rules. This variant does not interact directly with an environment except to receive the input. The results are directly outputted from this algorithm.

3.4.3 Heuristic Agent Variant

The other variant of our heuristic approach is the *Heuristic Agent*. The logic it follows is shown in Figure 3.2. This variant emulates a reinforcement learning agent. Where it is provided with limited knowledge on the environment, and it must perform an action given the current state and receive the next state. It will continue to act on the environment until a terminal state is reached, where it will terminate. We designed this variant to have a more akin method to our reinforcement learning agents allowing us to use the same environment we developed for both methods, and to measure the performances and differences. As well, having a variant that functioned like a reinforcement agent made it simple to collect and assemble a generated dragon boat partition dataset. Lastly, this variant does not employ the Weight Gradient Partition Phase, as we found it to be a trivial problem of using the Quicksort algorithm.

3.5 Experiments

First, in our experiments, we only show results from the heuristic agent, as the difference between the two agents is marginal, and have no growth time difference as they run on the

same algorithm, but emulation of a deep learning agent requires extra constant time logic. Also, we exclude the Weight Gradient Partition phase, given our previously stated reasons.

Our approach involves randomly generating 10,000 perfect partitioned dragon boat environments, then shuffling the positions of each participant. This is followed by finding an approximation of the optimal solution for each generated environment per parameter combination. We have 3 different versions of our environment - the left-right boat where only the cleaning and the left-right phase are considered, the front-back boat involving only the front-back phase, and lastly, the full-boat environment where both phases are implemented and expected to be approximated. Our approach has 3 significant variables - left-right relaxation V_{lr}, front-back relaxation V_{fb}, and the maximum search size z_m. Each significantly affects the ability to find an approximation. Note that, increasing the relaxation values will result in non-perfect discrepancies being accepted. Moreover, in the full-boat V_{lr} and V_{fb} will be increased at the same time.

In all of our experiments, we refer to accuracy as the percent of randomly generated dragon boat partition problem configurations in which a discrepancy is equal to the goal weight or falls within the relaxation. In Table 3.2 on row $V_{lr} = 0$ we can see that decreasing z_m results in a significant impact on finding perfect discrepancies and in turn affects the accuracy. However, increasing z_m eventually hits an upper limit and lacks improvement, as we observe in columns $z_m = 150$ and $z_m = 200$. This is because z_m will eventually be equal or greater than the largest weight difference, which will allow no more margin of error between the lightest rower and heaviest rower. Thus a maximum search size greater than or equal to the largest weight difference cannot push our algorithm into a state that it has not tried before. We observe that by increasing V_{lr} and V_{fb} we approximate more often within the optimal weight limit with relaxation. Continuing to increase the relaxation values eventually leads to almost any action, resulting in a "close enough" approximation, which could require no action at all. Thus, we only consider small relaxation values of 0, 1, 2, and 5.

Table 3.2: Percentage of perfect discrepancy approximate W^*_{lr}.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{lr} = 0$	86.65%	91.48%	98.31%	98.71%	98.63%	98.79%
$V_{lr} = 1$	89.48%	93.33%	98.73%	99.09%	99.08%	99.00%
$V_{lr} = 2$	89.30%	93.04%	98.55%	98.93%	99.00%	98.89%
$V_{lr} = 5$	92.56%	95.66%	99.29%	99.47%	99.45%	99.65%

We note that by increasing the search value z we in turn will naturally increase the amount of time required to search for an approximation. This is an expected outcome. As shown by Table 3.3 we observe that even with a larger z, the average elapsed time per episode is quite minimal and appears to stagnate at $z = 150$. On the other hand, by increasing v_{lr}, we will naturally lower the time to find an approximate, since there is more margin of error allowed.

Table 3.3: Average elapsed time in milliseconds to approximate W^*_{lr}.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{lr} = 0$	0.85ms	0.86ms	0.90ms	0.92ms	0.93ms	0.93ms
$V_{lr} = 1$	0.84ms	0.85ms	0.88ms	0.90ms	0.91ms	0.91ms
$V_{lr} = 2$	0.84ms	0.86ms	0.91ms	0.94ms	0.91ms	0.92ms
$V_{lr} = 5$	0.84ms	0.85ms	0.87ms	0.87ms	0.88ms	0.88ms

While we can see that the executing times increase steadily until $z = 100$, where it plateaus, we can see it takes more time. The differences in time for $z_m = 100$, $z_m = 150$, and $z_m = 150$ can be attributed to fluctuation in CPU busy cycles. However, the more important information, as shown in Table 3.4, that we need to discuss is the average steps taken to complete an episode.

Table 3.4: Average steps (sp) taken to approximate a left-right phase dragon boat episode.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{lr} = 0$	2.76sp	2.89sp	3.14sp	3.17sp	3.21sp	3.14sp
$V_{lr} = 1$	2.60sp	2.70sp	2.89sp	2.91sp	2.97sp	2.92sp
$V_{lr} = 2$	2.62sp	2.70sp	2.92sp	2.93sp	2.90sp	2.94sp
$V_{lr} = 5$	2.41sp	2.43sp	2.55sp	2.58sp	2.59sp	2.54sp

As shown, the average number of steps required to approximate a boat configuration

is heavily affected by the maximum search size. This reflects in perfect and imperfect discrepancies. We also need to look at best case and worst case scenarios. We only observe the search size that offers the best accuracy, and we look at the least amount of relaxation. We found that in the worst case over all of our scenarios for the left-right partitioning phase was 50 steps and our best case scenario was 1 step.

Next, we consider the front-back phase separately. We observe that this phase struggles to have a strong accuracy with minimal steps. This is due to the phase being completely constrained by the left-right phase's output. While many different combinations could potentially equal a perfect left-right discrepancy, which could affect whether the front-back could be perfectly partitioned. We observe a significant drop in accuracy, and a significant increase in elapsed time and steps, as shown in Tables 3.5-3.7.

Table 3.5: Percentage of perfect discrepancy approximate W^*_{fb}.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{fb} = 0$	55.85%	62.80%	81.36%	88.78%	89.16%	88.92%
$V_{fb} = 1$	59.99%	65.80%	83.73%	90.40%	90.41%	90.54%
$V_{fb} = 2$	63.63%	69.09%	86.25%	91.84%	92.47%	92.07%
$V_{fb} = 5$	72.06%	77.50%	90.43%	95.34%	94.96%	95.00%

As the front-back phase is expected to have a goal weight of $W^*_{fb} = 30$, we observe a decrease in accuracy with approximating a solution when the expected perfect discrepancy is not 0. We are fully convinced that the degraded performance is solely due to the expected goal weight being $W^*_{fb} \neq 0$, which requires an increasingly larger search space to find suitable actions.

Table 3.6: Average elapsed time in milliseconds to approximate W^*_{fb}.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{fb} = 0$	0.28ms	0.34ms	0.59ms	0.72ms	0.77ms	0.83ms
$V_{fb} = 1$	0.28ms	0.33ms	0.54ms	0.66ms	0.71ms	0.75ms
$V_{fb} = 2$	0.27ms	0.32ms	0.50ms	0.62ms	0.65ms	0.69ms
$V_{fb} = 5$	0.27ms	0.30ms	0.42ms	0.47ms	0.52ms	0.54ms

Table 3.7: Average steps (sp) taken to complete a front-back phase dragon boat episode.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{fb} = 0$	3.00sp	3.52sp	5.18sp	5.42sp	5.45sp	5.50sp
$V_{fb} = 1$	2.83sp	3.36sp	4.64sp	4.92sp	5.01sp	5.03sp
$V_{fb} = 2$	2.70sp	3.11sp	4.30sp	4.55sp	4.45sp	4.55sp
$V_{fb} = 5$	2.37sp	2.61sp	3.39sp	3.42sp	3.55sp	3.56sp

With a more constrained search task, where there are fewer options for swapping, performed during the front-back phase, there will be a significant impact on the episodic runtime. Additionally, there is a large increase in average steps per episode. Lastly, we found that in the worst case, the front-back phase took 64 steps, and in the best case, it took only 1 step. Therefore, we draw the conclusion that, despite having a lower number of actions available in the front-back phase, and not having to consider the handedness rule, W^*_{fb} has an impressive impact.

Thus we can suspect that when we test on the left-right and the front-back consecutively, there is a significant drop in perfect partition accuracy, and a significant increase in time and steps, just as it happened in the front-back phase experiments alone. Therefore, in a dragon boat episode with both left-right and front-back phases, our algorithm struggles to find a suitable significant outcome.

Next, we see if our findings match our intuition involving the combination of both phases, which we will refer to as the full-boat. We first observe the data involving the accuracy for the full-boat in Table 3.8. As is shown, the full-boat's accuracy is heavily affected by the ability to find a perfect partition during the front-back phase as seen in the previous tables. This is followed by investigating the average steps taken. We can see that the number of steps is heavily affected by the front-back phase as well. Lastly, we observe that the average steps will naturally be influenced to have a higher average from both.

While there is a small increase in the accuracy in the full-boat environment experiments over the front-back environment, we can assume this is caused by random variance, since we randomly generated our boat environments for every episode.

Table 3.8: Percentage of perfect discrepancy approximate $W^*_{lr,fb}$.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{lr,fb} = 0$	59.53%	67.28%	84.55%	89.95%	90.15%	89.29%
$V_{lr,fb} = 1$	63.19%	69.15%	86.10%	90.99%	90.13%	90.63%
$V_{lr,fb} = 2$	65.78%	72.69%	87.81%	92.05%	92.14%	92.66%
$V_{lr,fb} = 5$	73.28%	78.93%	91.24%	94.44%	94.66%	94.62%

Table 3.9: Average elapsed time in milliseconds to approximate $W^*_{lr,fb}$.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{lr,fb} = 0$	1.38ms	1.43ms	1.66ms	1.80ms	1.84ms	1.92ms
$V_{lr,fb} = 1$	1.37ms	1.42ms	1.62ms	1.74ms	1.83ms	1.86ms
$V_{lr,fb} = 2$	1.37ms	1.42ms	1.60ms	1.72ms	1.76ms	1.80ms
$V_{lr,fb} = 5$	1.37ms	1.41ms	1.53ms	1.61ms	1.63ms	1.65ms

Table 3.10: Average steps (sp) taken to complete a front-back phase dragon boat episode.

	$z_m = 0$	$z_m = 10$	$z_m = 50$	$z_m = 100$	$z_m = 150$	$z_m = 200$
$V_{lr,fb} = 0$	5.25sp	5.76sp	7.14sp	7.42sp	7.39sp	7.51sp
$V_{lr,fb} = 1$	4.99sp	5.47sp	6.64sp	6.95sp	7.12sp	7.10sp
$V_{lr,fb} = 2$	4.81sp	5.22sp	6.22sp	6.59sp	6.53sp	6.57sp
$V_{lr,fb} = 5$	4.37sp	4.66sp	5.41sp	5.63sp	5.56sp	5.60sp

The full-boat has a higher overall average number of steps taken, as it will have the potential of both worst case scenarios of the left-right and front-back. The full-boat resulted in the worst case of 68 steps. Consequently, with both phases included, we can assume we have the potential for the best case scenario to happen, where a single move during the left-right partition results in both the completion of both phases, thus the least amount of steps being 1. We will show this by an example. Let there be an episode where $W_{lr} = 20$, and $W_{fb} = 10$, then we know that the left-side is heavier than the right-side by 20lb and the front-half is heavier than the back by 10lb. Therefore, if we find two participants, one in the back-left, and the other in the front-right, with a weight difference of $10lbs$, we can see that swapping them would result in $W_{lr} = 0$ and $W_{fb} = 30$, requiring a single step to fulfill both requirements.

An important distinction between the left-right, front-back, and full-boat environments,

in particular, is the front-back phase in the full-boat does not start from a randomized perfect discrepancy. Rather, the phase begins only after the left-right phase has completed, or has been reverted. Conversely, the front-back boat environment only performs random front-back swapping actions for shuffling, leaving the left-right discrepancy perfect. Therefore, we can conclude that the particular order of the left-right phase partitioning has important significance on the front-back phase.

3.6 Summary

A novel approach to the dragon boat partition problem has been given. The performance of our approach has been measured, and different combinations of our parameters have been assessed. However, our approach still raises some questions for our future work.

One concern of our approach is the disconnection between the left-right partition phase and the front-back partition phase. We clearly have a decrease in close approximation during the front-back partition phase, which is directly affected by the left-right partition phase. This is partially due to the front-back partition phase being limited to swapping on only the left-hand side or the right-hand side and no crossover. We could allow this to occur. However, unfortunately, allowing crossover has a high potential to break the Handedness rule and Left-Right Weight rule. Then in turn the algorithm would need to return to the prior left-right partition phase, which in turn has a high potential of breaking the Front-Back Weight rule resulting in having to readdress the front-back phase, and so on. In Chapter 5 we discuss potential future work to handle this problem.

To summarize, we presented an approach that is able to approximate close to a perfect discrepancy, without relaxation, where $z_m = 150$, the accuracy for left-right balancing is 98.79%, front-back balancing is 89.16%, and full-boat balancing is 90.15% in 10,000 experiments each. Our unique approach is able to find a perfect discrepancy for all experiments in each environment with the left-right having 3.21 steps, the front-back having 5.45 steps, and the full-boat having 7.39 steps on average. Additionally, in the worst case

number of steps for each environment took 50, 64, 68 steps in the left-right, front-back, and full-boat environments respectively. While we could reducing the search size would decrease the number of steps, there is a significant drop in perfect partitions found.

Chapter 4

Applying Deep Learning to the Dragon Boat Partition Problem

In this chapter, we continue to explore techniques to approximate the Dragon Boat Partition problem. We make use of deep learning by converting our Dragon Boat environment state into a set of features characterized by the participants in their seating positions. To further extend our findings, this chapter presents a deep learning approximation algorithm. We describe the steps we have performed to improve the approximation achieved by deep learning.

This chapter contains a brief introduction in Section 4.1, followed by a description of how we combine the dragon boat partition problem with deep learning in Section 4.2. We describe the Deep Convolutional Neural Network we use in our models in Section 4.3. In Section 4.4 we discuss in detail our results and their analysis, with differences regarding our results from Chapter 3. Lastly, in Section 4.5, we close the chapter by summarizing our findings and potential future work.

4.1 Introduction

In this chapter, we dive into our approach to the Dragon Boat Partition problem using deep learning in a variety of methods. The first method is using supervised learning to predict expert swaps. In this case, the expert swaps were performed by our approach in Chapter 3 to generate a large dataset. After the model is pre-trained using supervised learning, we feed the model into a reinforcement learning agent to be further tuned by attempting to pre-

dict correct answers on the fly. We call this version the Supervised Reinforcement Learning Model, SRL-model. The second version skips the first method of using supervised learning and instead directly attempts to use reinforcement learning from start to finish, we call this version the Pure Reinforcement Learning Model, PRL-model.

Silver et al. [47] used previous known expert moves in AlphaGo to train a deep convolutional neural network to play the game Go through supervised learning. Afterward, they tuned their pre-trained model using reinforcement learning self-play. We follow a similar approach in our methodology, where a deep neural network is trained using collected good actions. The dataset obtained was collected using our heuristic agent version discussed in Chapter 3. The training methodology created for AlphaGo, as well as the conversion techniques used by Silver et al. [47, 48], were both direct influences on the techniques we derive and explore. In order to use a convolutional neural network Silver et al. [47, 48] converts Go board positions to a 2D image. In our research, we perform a similar approach by converting our boat state into an image that contains participant spatial and numerical data, allowing us to feed a 2D image into our CNN.

We aim to familiarize the reader with how the Boat Partition Problem interacts with deep learning. Our approach is applying deep CNNs to optimize the Dragon Boat Partition Problem in multiple avenues such as supervised learning and reinforcement learning. In addition, we discuss and present our findings in a detailed manner. Through experimentation, we aim to achieve competitive dragon boat partitioning results against our heuristic approaches through both supervised learning and reinforcement learning.

4.2 Fitting the Dragon Boat Partition Problem with Deep Convolutional Neural Networks

In our experiments, we used only 8 rowers (n) and we followed the same rules stated in Section 3.3.1. We stayed with 8 rowers since the dragon boat partition problem does not become more difficult with more participants but the training time for each even participant

distribution, 12, 16, and 20, takes exponentially longer to train. Also, we had limited computer hardware at our disposal. So increasing past 8 rowers is infeasible for us. We decided showing our approach with a small example would suffice.

Ideally, our goal was to have a generic model that would transition from the left-right partition phase to the front-back partition phase. However, we found separating the training into two distinct networks, and having each of them responsible for different phases is better. After training the two networks we append them in a consecutive manner where the one network policy handles the left-right partition phase and the front-back partition phase. We found this approach yielded positive results, whereas a generic model is unable to fit to the problem. We believe the generic model is unable to fit because of the close coupling of the rules, where one change on the left-right can affect 3 rules. Perhaps, given enough time a generic model could be fit, however, we did not have such a luxury. Thus, in our models, we have the left-right policy CNN to approximate the Handed rule and the Left-Right Weight rule and another front-back policy CNN to approximate the Front-Back Weight rule.

One additional challenge we encountered is creating an input that included the drummer and steer. This proved to be difficult, due to coding limitations in some of the code libraries we used. Thus during the training and testing of our model, they are excluded but will be included in future work.

4.2.1 Acting in a Dragon Boat Environment

The dragon boat environment implements the standardized OpenAI Gym environment framework [11]. There are 3 main functions that each environment must implement.

- reset - This function is responsible for bringing the environment back to an initial state and then returning that initial state. Our implementation involved generating a dragon boat followed by shuffling the rowers by randomly sampling actions from our Action Space A.

- step - The step function is where most of the state changing takes place. This function is responsible for receiving an action and moving the environment into the next state, which determines if the state is good or bad, and provides a reward, depending on whether the next state is a terminal state as well as any additional info.
- render - Displays the current state in a human-readable manner.

We created two reward functions to fit the situation in our dragon boat environment. The first reward function involves shaping the reward based on how closely the current state matches a perfect partition and a terminal state receives a reward of 10. This proved very difficult to estimate the weights, positions, and the handedness of each participant in a meaningful manner. Our second reward function is a significantly more productive, albeit far simpler, sparse reward function. A sparse reward function in our case is where each state is given a reward of 0, except a reward of 10 on approximately optimal partition terminal states. As such, our reinforcement agent will converge towards finding an approximately optimal partition as only terminal states receive a positive reward.

4.2.2 Input Shaping

We have formalized the Dragon Boat problem in Section 3.3.1. Now we can discuss how we converted our dragon boat vector to an image. From a top-down view, as shown in Figure 3.1, it is clear that a dragon boat can be represented as an image, where there is a width of 2 for the columns of positions, height is equal to $n/2$, which is the number of rows in the dragon boat environment. Just width and height do not fully describe an image. Thus, each participant has 4 channels, that describes their properties. This effectively converts our dragon boat configuration into an image. For example, with 8 rowers, we have 4 rows and 2 columns, with 4 channels, therefore our image dimensions would be 4x2x4. See Table 3.1 for the definition of each channel. Using an image enables us to use convolutional neural networks, which excel at fitting to spatial and numerical features.

4.3 Deep Convolutional Neural Network Models

We used two different CNN architectures, to conduct the two training strategies we employed. The first model is a Double Deep Q Learning [52] Convolutional Neural Network which we pre-trained using supervised learning. The second model is a Dueling Double Deep Q Learning Convolutional Neural Network, which is trained purely with reinforcement learning. Using DuelingDQL results in extra layers at the end of the network, where the head is split into an advantage subnetwork and a value subnetwork, see Section 2.5.7. We will discuss the significant features of our model throughout Sections 4.3.1-4.3.5. Next we discuss our dataset generation in Section 4.3.6. We finish this section by discussing training our CNNs on a GPU in Section 4.3.7

4.3.1 Network Architecture

The neural network architecture for our CNN is summarized in Figures 4.1 and 4.2. The SRL-model contains 4 layers: one (1) convolutional layer, and three (3) fully connected layers. By contrast, our pure reinforcement learning model contained 5 layers: one (1) convolutional, and four (4) fully connected layers.

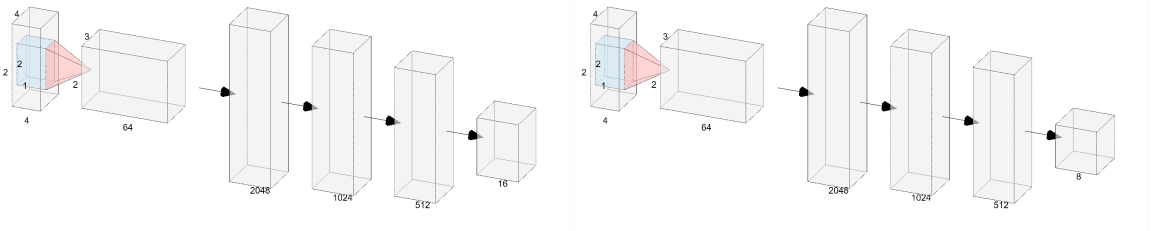

Figure 4.1: CNN architectures used for the Left-Right phase and Front-Back phase respectively during pre-training in supervised learning.

Sigmoid Nonlinearity

The Sigmoid nonlinearity function is the most important piece for our tuning for reinforcement models. No other activation functions appeared to improve the final performance. Therefore, our reinforcement CNN architecture exclusively features the Sigmoid

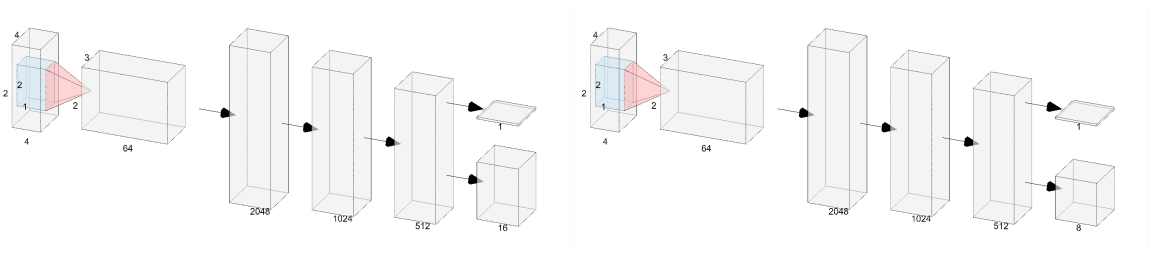

Figure 4.2: CNN architectures used for the Left-Right phase and Front-Back phase respectively during reinforcement learning tuning.

nonlinear function defined by $f(x) = \left(\frac{1}{1+e^{-x}}\right)$. Sigmoid has its problems, as discussed in Section 2.2.4. Despite its shortcomings, Sigmoid is the only activation function we found that worked with our problem. To our advantage, we did not encounter the vanishing gradient problem to a large degree, as our network ended up being a shallow network, with few layers. Unfortunately, the training time on our network does suffer from the use of Sigmoid, but we found the trade-off is worth it. Originally, we tried the popular ReLU activation function. However, our CNN is not able to learn the problem using ReLU and its variants during reinforcement learning. Similarly, we attempted to use the Tanh activation function, which also failed.

During pre-training in supervised learning we found that the network managed to fit to the data for sigmoid, Tanh, and ReLU, As shown in Figures 4.3 and 4.4. We are not certain why supervised learning succeeds with those functions, but reinforcement learning fails to learn using ReLU and Tanh. Therefore, we trained our reinforcement models using only the Sigmoid activation function.

Pooling

Unlike most CNNs, we do not include a pooling layer. We excluded pooling solely due to our input vector since it is small spatially and each position is vital. Pooling would destroy the spatial data significantly. As such, we maintain the spatial context as much as possible.

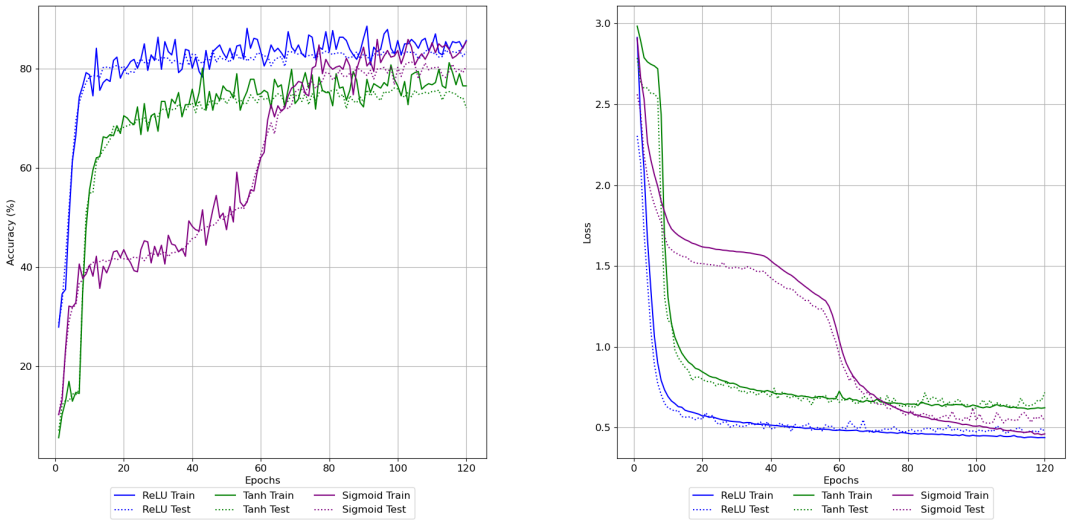

Figure 4.3: Supervised left-right policy training activation function differences. Left: training and testing Accuracy. Right: training and testing loss.

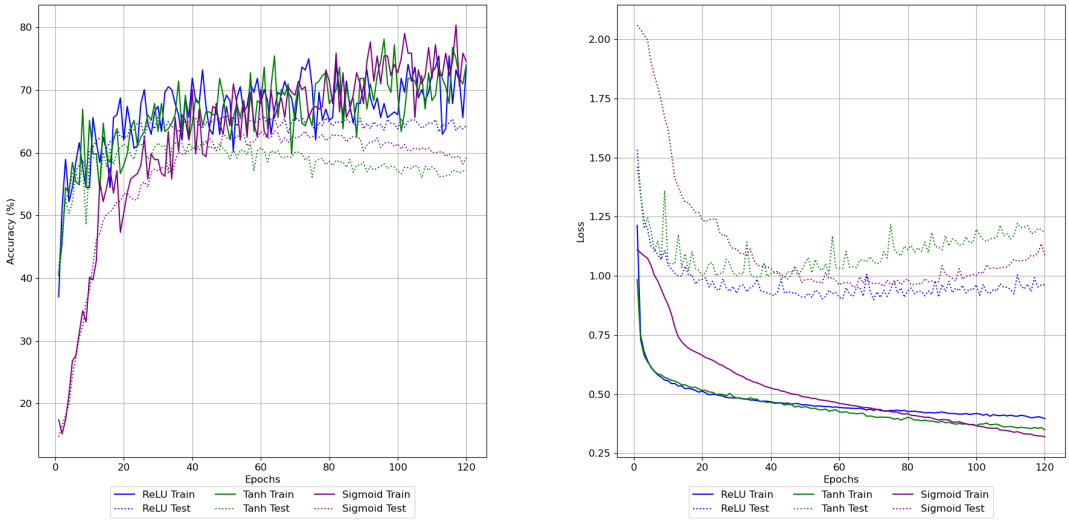

Figure 4.4: Supervised front-back policy training activation function differences. Left: training and testing Accuracy. Right: training and testing loss.

In-depth Architecture

Now that we have introduced the main components and overall architecture of our CNN, we can describe the architecture in-depth. As shown by Figures 4.1 and 4.2, our SRL-model CNNs has 4 layers with weights and the PRL-model CNNs has 5 layers with weights. The first layer on both models is a convolutional layer, and the rest are fully connected layers. The total output of both models is 24 actions: 16 in the left-right action space, and 8 in the front-back action space. Each output is mapped to an action in the dragon boat environment action space, and each action is mapped to a swap.

As we have one convolutional layer, this means the neurons of the second layer, are fully connected to each of the previous convolutional layer's neurons. The rest of the fully connected layers neurons are connected to the neurons in the previous layers. As discussed, we also apply the Sigmoid activation function to all of our convolutional and fully connected layers, except the output layer.

Since our experiments use only 8 rowers, then the convolutional layer receives an input 'image' of $4x2x4$, with 64 kernels of size $2x1$, with a stride of $1x1$ and no padding. The fully connected layers decrease in size progressively, starting from 1024 neurons on the first fully connected layer, 512 on the second fully-connected layer, and 256 on the final layer before outputting. We use a single stride and no padding because we want the positional and numerical data to be the main focus. We found that increasing the stride and padding led to a lack of understanding of the input image and difficulties in converging. We discuss our other hyperparameters further in Section 4.3.5.

The selection of a kernel size of $2x1$ is largely due to how a dragon boat with 8 rowers is split into four sub-groups. These groups consist of front-left, front-right, back-left, and back-right. Thus, by using a filter of $2x1$, we are able to capture the groups in the kernels. Ideally, this provides the CNN with the ability to see individual contextual groups compiled together.

4.3.2 Inputs and Outputs

As described above, the inputs for our models consist of converted dragon boat positional images to fit in a convolutional network. Each image is in the form of $4x2x4$, where the first value is channels, followed by rows and columns. Each input for the SRL-models and PRL-models data has been generated using our generation function.

Since our outputs are treated as actions, we treat each action as a class label in the supervised training. Recall from earlier that our total number of actions is 24 with 8 rowers. Therefore, since our actions are treated as a class we can use the categorical cross-entropy loss objective function as discussed in Section 2.4. This objective function allows us to quickly train our model using supervised learning from the collected dataset. Which enabled us to find an architecture faster, since using pure reinforcement learning took multiple days.

Our PRL-model has an improved methodology approximating q-values. The CNN in the PRL-model ends with a split head, an advantage layer, and a value layer. In this CNN the q-values are calculated by combining the outputs of both heads. The advantage network is a traditional output as described above. However, the value network outputs a scalar, which is the value of the current state. The value scalar is applied across the advantage of each action, and then the mean of the advantages is subtracted as seen on Equation 2.13 [54]. We used this architecture since our environment has many similar states, such as two participants having equal weights, but different positions. Results were obtained quicker and were more stable, in part due to that reinforcement learning has a higher potential [7] with an improved architecture [54].

4.3.3 Training Strategies

We used two training strategies as mentioned in Section 4.1. The first strategy we explored is using supervised learning to first pre-train our model using a dataset collected by our heuristic agent. Our model which is trained using supervised training is then fur-

ther tuned using reinforcement training. Particularly, the SRL-model is tuned with Double Deep Q Learning [52]. It uses a uniform experience replay (UER) to save and recall memories [35]. The second training strategy is pure reinforcement learning, where a completely untrained model is fitted using reinforcement learning via environmental interaction. The PRL-model employs Dueling Double Deep Q Learning [54, 52]. We also used a UER to save and recall memories [35]. All of our memories are of the form $m = (s_t, a, r, s_{t+1}, d)$, where s_t is the current state, a is the action performed in the current state, r is the reward received for the state-action pair, s_{t+1} is the next state, and d is the Boolean value done, stating whether the environment episode is completed or not.

The pre-training methodology helped as a guide to an effective CNN architecture and hyperparameters for learning in the SRL-model and PRL-model. Originally we attempted to find a pure reinforcement learning approach without the supervised pre-training version but were unable to find an architecture that would learn the semantics of our dragon boat environment. After investigating with failures, we decided to attempt the approach used by AlphaGo [47]. Since supervised learning trains significantly faster than reinforcement learning, we were able to construct a suitable architecture in a fraction of time.

4.3.4 Training Process

The training process for our SRL-model is summarized in Figure 4.5. The dataset used to train our pre-train our SRL-model is discussed in Section 4.3.6. The pre-training is split into the left-right process and the front-back process. In the left-right process, we train a CNN on the left-right dragon boat environment dataset to approximate the Handed Rule and the Left-Right Weight Rule. This is equivalent to the Cleaning phase and the Left-Right Partition phase of our heuristic approach. Next, the front-back processes trains a different CNN using the front-back dragon boat environment dataset. This CNN is responsible for approximating the Front-Back Weight rule, which is equivalent to the last Front-Back Partition phase of the heuristic approach in Section 3.4.1. After the supervised training is

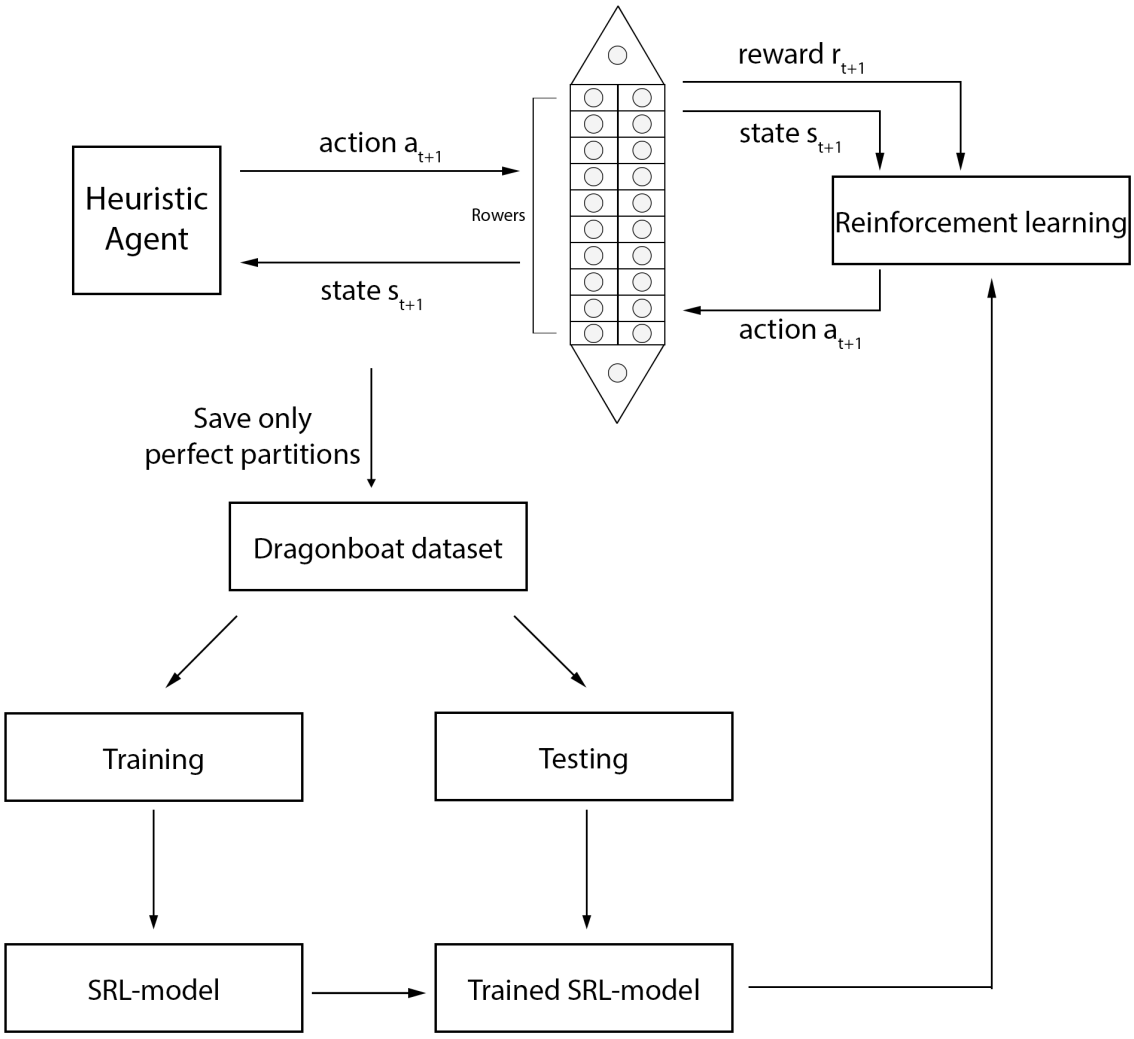

Figure 4.5: Training process for the SRL-model.

finished, we feed the trained models into reinforcement learning agents to be tuned on their respective left-right and front-back dragon boat environments. The left-right model is tuned for 6 million episodes, and the front-back model is tuned for 3 million episodes.

For the PRL-model we take a similar approach. A notable difference is that we do not pre-train our model using supervised learning before tuning with reinforcement learning. From the start, we train a blank CNN on two processes. The first process teaches a CNN the Handed Rule and the Left-Right Weight Rule. The second process teaches a different CNN the Front-Back Weight Rule. This pure reinforcement training continues for 6 million and 3 million episodes, respectively, for consistency. Both models later combine their CNNs to

function as one cohesive process during our testing phase with a full-boat.

4.3.5 Hyperparameters

Both of our models, SRL-model and PRL-model, have very similar hyperparameters, which will be described below. First, the number of layers and neurons is one of the starting points for our neural networks. It acts as the foundation of the model, as the number of layers and neurons affects the networks' ability to learn a context [41]. For the SRL-model's left-right CNN it has 4 layers: one (1) convolutional layer, with 64 kernels, and three (3) fully connected layers with 2048, 1024, 512 neurons respectively, with an output of 16. The front-back CNN has the same network structure, except it has eight (8) outputs. The left-right CNN has 3,420,240 learnable weights and the front-back CNN has 3,416,136 learnable weights in the SRL-model. The PRL-model has the same network structure, except the final layer is split into two outputs. The left-right CNN has an advantage output of 16, and a value output of one (1). The front-back CNN has an advantage output of eight (8) outputs and one (1) value output. Therefore, there are more parameters in this model with the left-right and front-back CNN having 3,420,753 and 3,416,649 respectively.

To initialize our networks we used uniform distribution, which affects the early training and allows for an early starting point for the network [41]. Activation functions have an important role in introducing nonlinearity to NNs, allowing them to learn nonlinear boundaries for prediction [41]. We trained using Sigmoid, Tanh, and ReLU activation functions for supervised pre-training. However, for reinforcement learning, we exclusively use Sigmoid as we found the other activation functions failed to converge.

The learning rate determines the network weight updates, a high learning rate can learn quickly, but may never converge, while small learning rates will converge, but at a slower pace [41]. In our case, we found that for supervised learning, we could set our learning rate between 0.0004 and 0.001, with 0.001 being the best, while with reinforcement learning, our models were only stable using a learning rate of 0.00001. For optimizing the CNNs

weights we used Adam as our optimizer. We chose Adam as it performs well in many different situations and with sparse reward functions [38].

The batch size determines the number of samples to provide a network for weight updates [41], and we found that a batch size of 512 yielded the best results. Specifically for supervised learning, the number of epochs is the number of iterations a neural network trains on the whole training data. For the models using Tanh and ReLU, we found that 50 epochs provided the most superior results, while Sigmoid models required 120 epochs. Target network update frequency C, discussed in Section 2.5.4, is another extremely important hyperparameter, as it reduces the correlation of the target data by keeping the target network temporarily fixed [35]. In our case, we found that $C = 100$ yielded the quickest and best results.

4.3.6 Data Generation

Large datasets are widely available in the era of big data. Unfortunately, as we were researching a niche problem, there were no large datasets available. Therefore, we generated our own dataset. Using the method below we are able to generate scalable datasets for both the left-right balancing model and the front-back balancing model. Recall from Section 3.3.3 where the action space is defined with 8 participants to have 24 total actions. The action space is split into two action spaces, the left-right action space and the front-back action space. The left-right dragon boat environment dataset consists of 16 classes, one for each action in the left-right action space, and the front-back dragon boat environment dataset consists of 8 classes, one for each action in the front-back action space, as discussed in Section 3.3.3.

The first step in generating our dataset is described in Section 3.3.4. To create a dataset we used our heuristic agent approach discussed in Chapter 3. To decide which state-action pairs were appropriate, the randomly generated dragon boat configurations are passed to the heuristic agent until the environment returns a terminal state or until the heuristic agent

cannot find an acceptable configuration. Appropriate state-action pairs were any pairs that led to a perfect discrepancy, i.e., E=0, or E=1. Therefore, each state-action pair is stored based on its action label. Each state-action pair is collected until 25,000 items for the respective class are accumulated. After a class has reached that threshold, any extra data points accrued are discarded. Respective classes in our dataset are mapped to an action from either the left-right action space or the front-back action space. We collected 25,000 data points for each class. Therefore, in a dragon boat environment with 8 rowers, we had 16 left-right actions and 8 front-back actions, a total of 24 actions. Since each action is mapped to a class we have 24 classes. Thus, we have 600,000 data points total in our dataset. Lastly, we use 70% of our dataset for training and 30% for testing. In our case, we performed no validation, only training and testing. We didn't use a validation set for optimizing architecture but only testing to evaluate the current performance.

4.3.7 Training on a Graphics Card

For our experiments and training, we used an Nvidia RTX 2070 and Nvidia RTX 2070 Super graphics cards. We have the cards in two different machines, allowing us to train two networks at the same time. Generally, the cards are equivalent, except in terms of speed. The Nvidia RTX 2070 and Nvidia RTX 2070 Super have 8GB of onboard memory; fortunately, our network did not reach the limits of the graphics cards. Additionally, we were able to fit the entire supervised training dataset onto the card. The Nvidia RTX 2070 contains 2304 CUDA cores, with a base clock speed of 1410Mhz[4]. The Nvidia RTX 2070 Super contains 2560 CUDA cores, with a base clock speed of 1605Mhz [5].

Using a graphics card reduced the training time significantly, allowing forward and backward propagation calculations to be parallelized. The CPU is then responsible for the remaining smaller tasks, such as experience retrieval, environment simulation, and data collection.

[4]https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070/.
[5]https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070-super/.

4.4 Results and Analysis

Next, we discuss our deep learning model's training and testing phases and the limitations of our models. Afterward, we compare each of our versions. Lastly, we compare the best deep learning model to our heuristic approach from Chapter 3. We have 3 types of models: the supervised learning model, the SRL-model, and the PRL-model, which we will discuss respectively. Each of these models contains two policy CNNs, one for the Handed and the Left-Right Weight rule, and the other for the Front-Back Weight Rule. As before, we will not consider the Weight Gradient rule. During our training our model's CNNs are trained separately, one solely trained using a left-right boat environment, and the other front-back boat environment. Recall from Section 3.5 about our other boat environment versions. After the models have been sufficiently trained we combine them into one and test on 10,000 episodes of the full-boat environment.

The first model we will discuss is the supervised learning model. The CNNs in this model are trained solely on supervised learning from the collected heuristic actions made by our approach in Chapter 3.

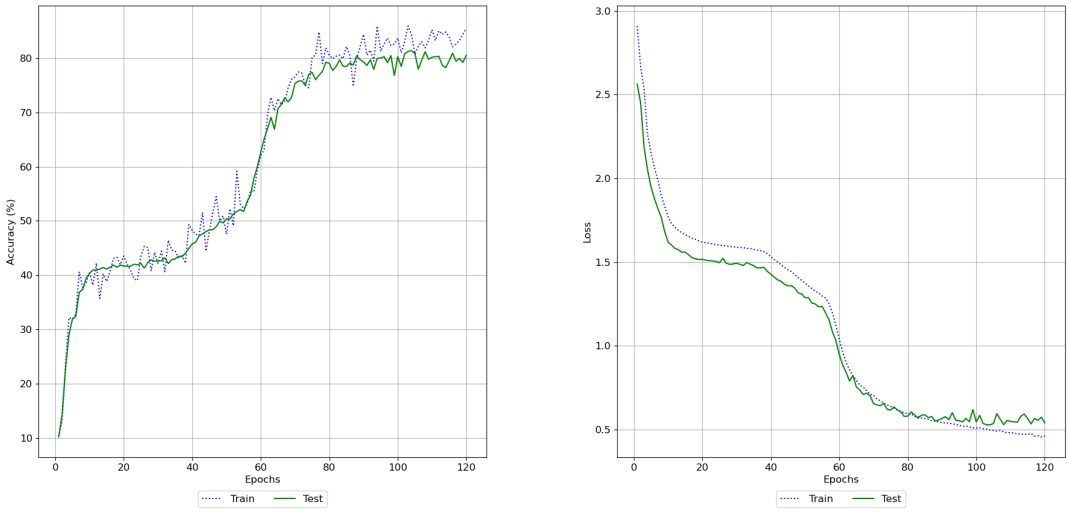

Figure 4.6: Supervised model left-right policy training and testing graphs. Left: training and testing accuracy. Right: training and testing loss.

We can see from Figure 4.6, that the left-right CNN in the supervised model learns the dataset quickly and has a high training and testing accuracy, both of which stay consistently close. As we approach 120 epochs, we can see testing accuracy and loss plateau. While the model's training accuracy and loss continue to improve, the testing accuracy and loss are stagnant, as expected for supervised learning. We suspect this is due to the fact that each data point is quite similar, and each action has related consequences to other actions.

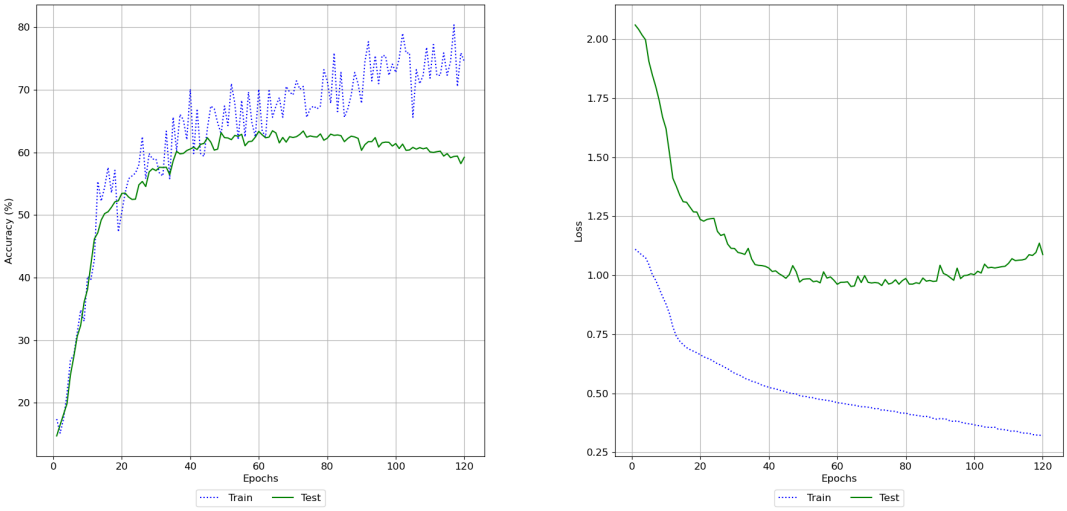

Figure 4.7: Supervised model front-back policy training and testing graphs. Left: training and testing accuracy. Right: training and testing loss.

We performed the same training and tests for the front-back CNN, as shown in Figure 4.7. However, we had a different result in this training instance. While we did allow training to continue further than necessary resulting in the testing accuracy and loss to deteriorate, we can see the front-back phase is significantly quicker to learn, but the testing never had a second spike. We can see that after 60 epochs, the model overfits. Therefore, we can draw a clear comparison to our heuristic model having a lower accuracy during the front-back phase.

While we do see some impressive training occurring from our supervised model, it appears to be doing slightly worse than our heuristic approach. Although our intuition assumed the supervised model would perform better, since it is training on perfect partition

Table 4.1: Supervised model average test results over 10,000 episodes.

	Time(ms)	Steps	W^*_{lr} Accuracy	W^*_{fb} Accuracy	$W^*_{lr,fb}$ Accuracy
$V_{fb} = 0$	1.48ms	8.89	82.02%	60.43%	55.30%
$V_{fb} = 1$	1.41ms	7.98	86.34%	66.42%	61.10%
$V_{fb} = 2$	1.39ms	7.94	85.92%	67.21%	61.62%
$V_{fb} = 5$	1.30ms	6.51	92.30%	76.03%	71.77%

training data only, we can see there is still some missing context in the input. We believe the supervised learning model is failing to be comparable to the heuristic algorithm because the heuristic algorithm is provided with some explicit rules such as W^*_{lr}, V_{lr}, W^*_{fb}, and V_{fb}. In addition, the heuristic model calculates the W_{lr}, and W_{fb} while our CNN-based model must implicitly discover these rules.

As such we can compare our testing results as shown in Table 4.1, against our heuristic approach, by feeding randomly generated test input into the model and using the maximum output value as a prediction as an action. We notice some noticeable decreases in approximating W^*_{lr}, W^*_{fb}, and $W^*_{lr,fb}$, as the accuracy for each is lower than the heuristic approach, where the heuristic approaches' W^*lr accuracy is 98.63%, W^*_{fb} accuracy is 89.16%, and $W^*_{lr,fb}$ accuracy is 90.15%. There is a measure of error that comes from both the number of steps and the time taken. We can drastically reduce the number of steps and time taken by our model by simply lowering the maximum allowed steps per episode. We see an improvement in our model's performance if we use a larger relaxation, but it still is not quite on par with the heuristic approach.

Next, we will make observations and discuss our training results, as well as the testing results of the PRL-model. We trained our left-right policy for over 6,000,000 episodes and the front-back policy over 3,000,000 episodes, with tests every 10,000 episodes. We observe in Figure 4.8 and Figure 4.9 that the loss has an extraordinary influx, we conjecture this is due to the pre-training supervised learning and reinforcement learning loss calculation differences. Furthermore, the accuracy and steps taken steadily improve, and appear to

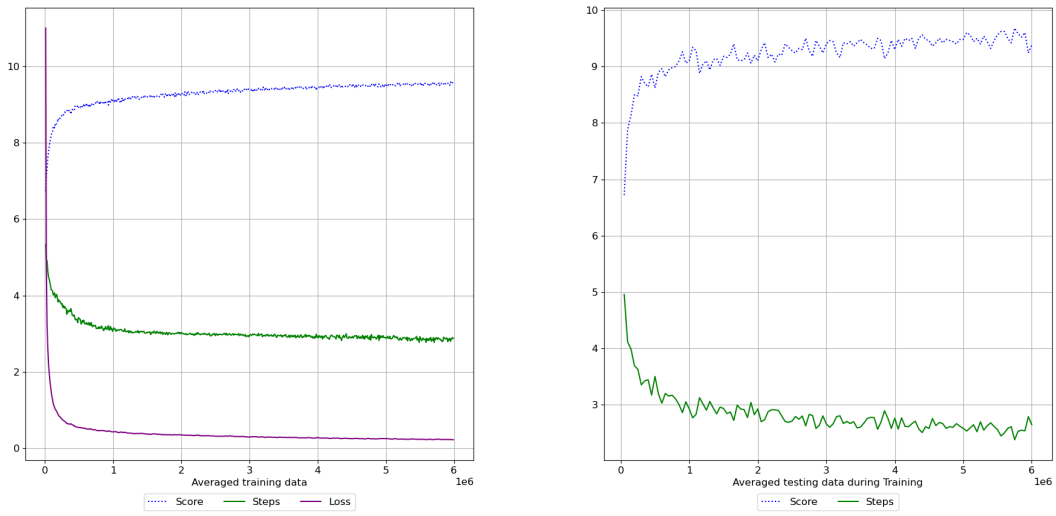

Figure 4.8: SRL-model left-right policy training and testing graphs. Left: Training score, steps and loss. Right: testing score and loss.

Table 4.2: SRL model average testing results over 10,000 episodes.

	Time(ms)	Steps	W^*_{lr} Accuracy	W^*_{fb} Accuracy	$W^*_{lr,fb}$ Accuracy
$V_{fb} = 0$	1.26ms	5.62	94.21%	89.53%	87.17%
$V_{fb} = 1$	1.23ms	5.14	96.29%	91.64%	89.93%
$V_{fb} = 2$	1.22ms	5.09	96.57%	91.55%	89.88%
$V_{fb} = 5$	1.20ms	4.65	97.65%	93.99%	92.68%

level out around 1 million episodes reaching a maximum.

As expected the SRL-model has much less to learn. The loss, score, and steps sharply improve over the first million episodes and then stagnates for the left-right policy. The front-back phase policy greatly improves over the first half-million episodes and then begins to stagnate. As we can see in Table 4.2, the SRL-model is a strong competitor against our heuristic approach. The perfect discrepancy approximation accuracy is lower during the cleaning and left-right phase by 4.50%. The SRL-model performs almost identically to our heuristic approach during the front-back phase, with a positive difference of 0.37%, which we conjecture this is due to random variance. Lastly, the heuristic approach remains the higher contender for the full-boat perfect discrepancy approximation, being 2.98% higher.

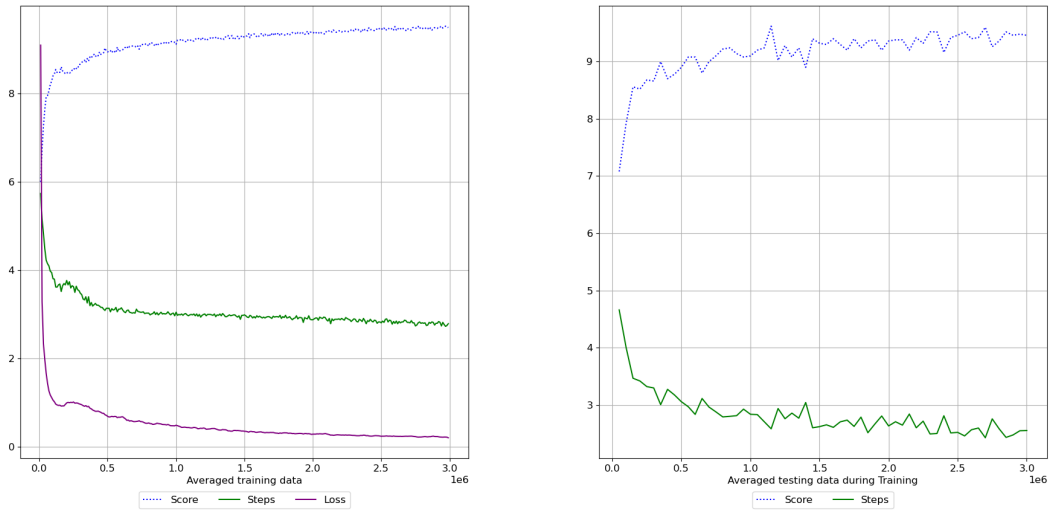

Figure 4.9: SRL-model front-back policy training and testing graphs. Left: Training score, steps and loss. Right: testing score and loss.

Onto our best deep learning model, we found that our PRL-model outperforms our other approaches in the cleaning and left-right phase, front-back phase, and the combined full-boat. We believe this is both due to the impressive capabilities of reinforcement learning, as well as the DDDQL [54, 52] in our agent. DDDQL is superior at differentiating similar states to DDQL [54]. We can observe the steady increase in performance for score, and decrease in loss and steps over time for both the left-right policy and the front-back policy as shown in Figure 4.10 and Figure 4.11 respectively.

As shown in Figure 4.10 that the PRL-model's left-right policy learns significantly for roughly 2 million episodes and then plateaus, while the PRL-model's front-back policy plateaus around 1 million episodes. Both phases in PRL-model take twice as many episodes to the SRL-model counterpart before plateauing.

As we can see in Table 4.3, our PRL-model has an impressive perfect discrepancy accuracy of 98.00% for the cleaning and left-right phase, 95.13% for the front-back phase, and 94.28% combined together in the full-boat. We observe that the cleaning and left-right phase is on par with our heuristic model, which again is within a small random variance. The PRL-model has a negative difference of 0.79% during the cleaning and left-right phase,

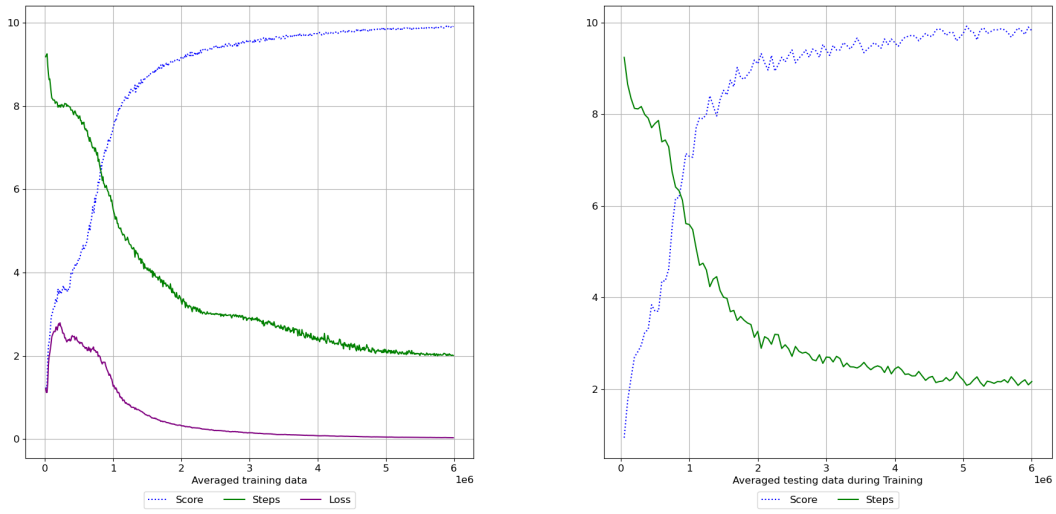

Figure 4.10: PRL-model left-right policy training and testing graphs. Left: Training score, steps and loss. Right: testing score and loss.

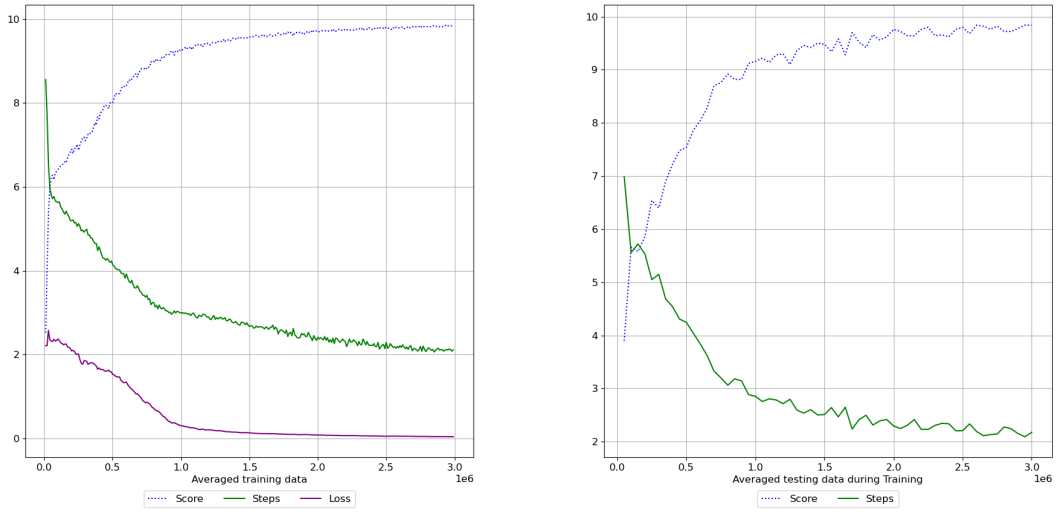

Figure 4.11: PRL-model front-back policy training and testing graphs. Left: Training score, steps and loss. Right: testing score and loss.

Table 4.3: Average testing results over 10,000 episodes.

	Time (ms)	Steps	W^*_{lr} Accuracy	W^*_{fb} Accuracy	$W^*_{lr,fb}$ Accuracy
$V_{fb} = 0$	1.23ms	4.65	98.00%	95.13%	94.28%
$V_{fb} = 1$	1.21ms	4.27	99.30%	96.37%	96.16%
$V_{fb} = 2$	1.22ms	4.31	99.17%	96.10%	95.75%
$V_{fb} = 5$	1.21ms	4.15	99.32%	96.48%	96.16%

Table 4.4: Comparison of heuristic agent with $z_m = 150$, supervised model, SRL-model, and PRL-model, each with $V = 0$ over 10,000 episodes.

	Heuristic	Supervised	SRL-model	PRL-model
Time (ms)	1.84ms	1.46ms	1.26ms	**1.23ms**
Steps	7.39	8.89	5.62	**4.65**
W^*_{lr} Accuracy	98.63%	82.02%	94.21%	**98.00%**
W^*_{fb} Accuracy	89.16%	60.43%	89.53%	**95.13%**
$W^*_{lr,fb}$ Accuracy	90.15%	55.30%	87.17%	**94.28%**

an upward leap of 5.97% during the front-back phase, and an impressive improvement of 4.13% on the full-boat. We hold the assertion that the front-back phase is heavily influenced by the left-right phase, and is a more difficult problem to approximate. We compare all of our models and our heuristic approaches in Table 4.4. One can see the impressive potential for reinforcement learning from our findings.

4.4.1 General Remarks

With the experiment results given, there are plenty of remarks towards our methodology, scalability, and effectiveness. While our training methodology has its limitations, we can begin discussing the method by which we generate our environments.

We see that we have made some strong assumptions that simply do not exist in real life, such as the drummer and steer's weights are always constant. Excluding them causes difficulty involving a practical setting, since they affect W_{fb}. We were unable to find a way to include the drummer and steer in the input, as using just the rowing participants as input we can easily manipulate it to be a 2D image. However, including the drummer and steer

complicates our input.

Even though we aimed to show a proof of concept towards the ability of deep learning to understand a partition problem, our scalability is problematic. A limitation of our approach is the lack of reasonable scaling to larger input sizes and including the drummer and steer. As the size of the input increases, the problem space increases exponentially, depending on the properties of the rowers (this is partly due to the inherent disadvantages of CNNs). This inhibits the scalability of our models as the time required to train the models also increases significantly. Using reinforcement learning and 8 participants, our models required 5 days to train the left-right policy, and 2.5 days for the front-back policy.

The parameter combinations, although they are a set of parameters that do in fact fit our model there is still room for improvement. While we did extensive testing to find a model that fits the best for our problem, finding a better combination of parameters could significantly improve the accuracy and training time.

4.5 Summary

A deep learning approach to the dragon boat partition problem has been presented. The performance of this approach has been evaluated against our heuristic approach and multiple avenues and parameters have been investigated. This approach has a significant advantage over the prior approach, as it can learn the nuances of a proper dragon boat configuration. Further improvements on this method with a full dataset using human expert choices could lead to a more sophisticated model capable of approximating the optimal partition in a fraction of the time after training.

We are happy to show our results from the PRL-model, which are able to outperform our heuristic approach by learning good dragon boat partitions through a sparse reward system. Our training took a significant amount of time, but with the model trained our results are easily reproducible and can be easily compared to others. We were unable to find an alternative solution using deep learning, thus we are led to believe this is the first of

its kind.

We have now seen the effectiveness of the use for deep learning on the dragon boat partition problem. Through our findings, it is obvious to us that creating a more sophisticated deep learning model with more constraints to handle this optimization problem is possible. This is something that will be addressed in our future work.

Chapter 5

Conclusion

The partition problem is broad and has many variants. One such variant is the dragon boat partition problem which we defined in Chapter 3. A dragon boat is a 2 columned boat with 10 seats per column. It also has a seat in the front for the drummer and one in the rear for the steersman. The dragon boat partition problem is an optimization problem with multiple constraints and rules. The dragon boat partition problem expects its participants to be on their proper side of either left or right, depending on their rowing ability. As a perfect partition is defined as having a summed difference of 0lb between the left-side and right-side, and a 30lb difference in the front-half and the back-half. Finding a perfect partition is a hard problem. We believe that a successful machine learning algorithm could be helpful in multiple different incarnations of the dragon boat problem, such as aeroplanes, freight ships, etc.

In Chapter 1, we describe the Partition problem and explain in detail the variant of the Partition problem known as the dragon boat partition problem. The goals and approaches of our thesis are also described. Chapter 2 presents a background to understand our approaches for the dragon boat optimization problem. Each section describes the concepts applied in our approaches, including Optimization problems and approximation algorithms, convolutional neural networks, introduction to deep learning, supervised learning, unsupervised learning, reinforcement learning, and application of convolutional neural networks. In addition, some algorithms used in our deep learning approach are also discussed, including Adam, Uniform Experience Replay, Gym environments, Deep Q-learning, Double Deep

Q-learning, and Dueling Deep Q-learning.

Afterward, we propose two approaches to approximating the dragon boat partition problem, one which utilizes heuristics, and the other deep learning. In Chapter 3 we present our heuristic approach. To our knowledge, there appears to be no other approaches that use a similar differencing heuristic to approximate the dragon boat problem as proposed in Chapter 3. Our differencing heuristic is not entirely unique (see the Karmarker and Karp differencing heuristic [33]). The parameters in our approach can be tuned for changes in speed and accuracy. These parameters include widening the search size z_m, which is a slower approximation to a perfect partition. Conversely, the more relaxation V provided, the faster approximating is achieved, but at the cost of being further from the perfect partition. We justify this approach with our experiments on randomly generated data and discuss the results we receive.

The next approach is discussed in Chapter 4. This approach attempts to apply deep CNNs to the dragon boat partition problem. We aim to improve both the accuracy of approximating perfect partitions as well as reduce the amount of time required to either succeed or fail. We intend to employ deep learning to approximate the dragon boat partition problem. The intuition in our approach involves generating a dragon boat, or knowing a partition is good, which is relatively simple. However, heuristically approximating is difficult. Thus, replacing a heuristic approach with a deep learning model to learn the nuances could be more effective. To our knowledge, applying deep learning to the dragon boat partition problem has not been attempted before. We examine our approach by comparing the results against our heuristic approach proposed in Chapter 3. While our supervised model and SRL-model were not quite up to par, our PRL-model was able to match the cleaning and left-right phase and surpasses both the front-back phase and the combination of the two.

5.1 Limitations of Our Approaches

The contributions in this thesis have been discussed. However, our approaches have limitations that need to be discussed. We have three different groups of limitations: environment limitations for depth and reward functions, heuristic agent's inflexibility of different combinations, and the deep learning models' unscalability.

The first limitation in our environment affects both of our approaches. The proposed environments lack definitions for deeper enumerable participant properties, in which the excluded attributes may heavily affect the dragon boat balance, such as height, and strongest position. Additionally, while we did have the steer and drummer in the environment they were left as a constant weight difference of 0, which does not influence the front-back weight. The second limitation affects only the reinforcement learning models, where the reward function is not complex enough. We chose to use a sparse reward function, as having a complex reward system for the dragon boat environment proved difficult and cumbersome with an extensive amount of edge cases. As a result, without a complex reward shaping function the training time will be negatively affected.

A second limitation comes from our heuristic agent's inability to generate multiple outcomes for a dragon boat configuration. Although finding a perfect partition is the task, having multiple approximately optimal outputs would be the preferred option in real-world applications since many times the cost to obtain a perfect partition is forbidden.

The last limitation is an important shortcoming of our deep learning approach. While it is able to have impressively better results than the heuristic approach, it lacks scalability. This approach is unable to receive an input that is greater than 8 rowers. As well, it is not able to handle the drummer and steer in the input. We could upscale the architecture to handle up to 20 participants, but a large problem domain results in a massive increase in training time. Furthermore, having a larger input size would require a larger architecture, which, given our hardware in Section 4.3.7, may become too large to finish our experiments.

5.2 Future Work

As mentioned above, there are a few improvements that could be made to our environment and our approaches. For our dragon boat environment, we would need to add another metric to determine where the best positions would be when measured additionally with height and strongest positions for each participant. To include the steer and drummer we would need to investigate including additional features that do not fit in a 2D image for a CNN. We could achieve this by having a separate network for additional features, then concatenate the output of the rower CNN with the additional feature network's output into an FC network body.

Additionally, we mentioned our heuristic agent lacks the ability to find multiple close approximates. To address this shortcoming we plan to implement a hash map to save potential close approximates. Furthermore, as mentioned in Chapter 3 we run into a problem where there is a significant decrease in close approximates during the front-back partition phase. A simple method to alleviate some of the problems caused by this is by shuffling the front-left and back-left, and the front-right and back-right each time the front-back partition phase fails, up to a maximum h number of times. In our experiments in Chapter 3, we discussed our dataset being generated by our heuristic approach. Even though our PRL-model had better results, the SRL-model is essential in discovering an architecture that worked in both instances. Thus, if we had real-world data, we could create a practical and realistic solution through supervised learning. In addition, we would be able to further improve the generation of our dataset for more diverse and precise results.

To address the substantial training time exhibited by our reinforcement learning models we have a few plans. We will further investigate into newer techniques for Deep Q-learning, such as CategoricalDQL [6], and RainbowDQL [24]. Furthermore, we would research improving our experience replay with various methods such as Prioritized Experience Replay [44], and Ape-X for distributed experience replay [25]. Lastly, we would like to investigate applying Genetic algorithms [20] to the dragon boat partition problem. Genetic algorithms

take two models and mix their weights together to produce an improved child model.

An important future goal involves our deep learning model's scalability. Due to the nature of our design, scaling up and down is impossible with our deep learning models as they have been statically trained on eight participants. Even though CNNs can receive varying sizes of input, assuming the channels stay the same, the number of outputs is statically decided as well. This creates a scaling issue on the number of outputs, as the number of actions is calculated based on the number of rowers. Therefore, we plan to do further research in dynamically sized input and output. These tasks will be the focus of our future studies.

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning A Textbook*. Springer International Publishing, 2018.

[2] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.

[4] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[5] Martin Anthony and Peter L Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 2009.

[6] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017.

[7] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[8] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[9] Avrim L Blum and Ronald L Rivest. Training a 3-node neural network is np-complete. *Neural Networks*, 5(1):117–127, 1992.

[10] Daniel Pierre Bovet, Pierluigi Crescenzi, and D Bovet. *Introduction to the Theory of Complexity*. Prentice Hall London, 1994.

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

[12] Jason Brownlee. 4 types of classification tasks in machine learning, Aug 2020. https://machinelearningmastery.com/types-of-classification-in-machine-learning/.

[13] Jason Brownlee. How to choose loss functions when training deep learning neural networks, Aug 2020. https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

[14] Timo Böhm. An introduction to selus and why you should start using them as your activation functions, Aug 2018. https://towardsdatascience.com/gentle-introduction-to-selus-b19943068cd9.

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011.

[16] Mark Fogliani. The best way to balance your dragon boat, Dec 2015. https://dragonanalytics.com.au/the-best-way-to-balance-your-dragon-boat/.

[17] Keith D. Foote. A brief history of machine learning, Mar 2019. https://www.dataversity.net/a-brief-history-of-machine-learning/.

[18] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[19] Zoubin Ghahramani. Unsupervised learning. In *Summer School on Machine Learning*, pages 72–112. Springer, 2003.

[20] David E Goldberg and John Henry Holland. Genetic algorithms and machine learning. 1988.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[22] Brian Hayes. Computing science: The easiest hard problem. *American Scientist*, 90(2):113–117, 2002.

[23] Sinai Health. What is dragon boat racing? https://www.mountsinai.on.ca/staff/sinai-lightning/what-is-dragon-boat-racing.

[24] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[25] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[26] Kai-Lung Hua, Che-Hao Hsu, Shintami Chusnul Hidayati, Wen-Huang Cheng, and Yu-Jen Chen. Computer-aided classification of lung nodules on computed tomography images via deep learning technique. *OncoTargets and therapy*, 8, 2015.

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[29] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[30] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[31] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[32] Ismail Mebsout. Convolutional neural networks' mathematics, Oct 2020. https://towardsdatascience.com/convolutional-neural-networks-mathematics-1beb3e6447c0.

[33] Stephan Mertens. The easiest hard problem: Number partitioning. *Computational Complexity and Statistical Physics*, 125(2):125–139, 2006.

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

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[36] Vladimir Nasteski. An overview of the supervised machine learning methods. *Horizons. b*, 4:51–62, 2017.

[37] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, 2015.

[38] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[39] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[40] Pablo Pedregal. *Introduction to optimization*, volume 46. Springer Science & Business Media, 2006.

[41] Pranoy Radhakrishnan. What are hyperparameters? and how to tune the hyperparameters in a deep neural network?, Oct 2017. https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a.

[42] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

[43] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[44] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[45] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

[46] Chathurangi Shyalika. A beginners guide to q-learning, Nov 2019. https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c.

[47] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

[49] Kristin Stickels. How to balance a dragon boat: Tips for your most successful race boat layout, Jul 2015. http://paddlechica.com/how-to-balance-a-dragon-boat-tips-for-your-most-successful-race-boat-layout.

[50] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[51] Fung Fung Ting, Yen Jun Tan, and Kok Swee Sim. Convolutional neural network improvement for breast cancer classification. *Expert Systems with Applications*, 120:103–115, 2019.

[52] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[53] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[54] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

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

[56] Bayya Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.