March 2022

Analyzing Decision-making in Robot Soccer for Attacking Behaviors

Justin Rodney
University of South Florida

Analyzing Decision-making in Robot Soccer for Attacking Behaviors

by

Justin Rodney

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Alfredo Weitzenfeld, Ph.D.
Alessio Gaspar, Ph.D.
Zachariah Beasley, Ph.D.

Date of Approval:
March 10, 2022

Keywords: Reinforcement Learning, grSim, SSL, Deep Deterministic Policy Gradient

Dedication

Thank you, mom. In dedication to Anastasia (2003-2021).

Table of Contents

List of Tables ii

List of Figures iii

Abstract iv

Chapter 1: Introduction 1

Chapter 2: Related Works 4
2.1 Deep Deterministic Policy Gradient 4
2.1.1 Description 4
2.1.2 Stepping Through the Environment 4
2.1.3 Updating the Networks 5

Chapter 3: Approaches 6
3.1 Environment/Setup 6
3.2 RoboBulls: Attack Main 7
3.2.1 The Probability Field 8
3.2.2 Detecting a Clear Shot 10
3.3 Neural Network 10
3.3.1 Data Set 11
3.3.2 Model Training 13
3.4 DDPG 14
3.4.1 Go-to-Ball 16
3.4.2 Shoot-Towards-Goal 17
3.4.3 Go-to-Ball-and-Shoot Behavior 18

Chapter 4: Results 19
4.1 Neural Network Results 19
4.2 DDPG Results 22

Chapter 5: Conclusion and Future Work 24
5.1 Summary 26

References 28

Appendix A: Copyright Permissions 32

About the Author . . . . . End Page

List of Tables

Table 3.1 The number of samples collected using each method. 12

Table 3.2 The number of samples used in the training and validation sets. . . . 13

Table 3.3 Neural network architecture. 14

Table 3.4 Testing scenarios. 14

Table 4.1 Neural network confusion matrix. 19

Table 4.2 Logistic regression confusion matrix. 19

Table 4.3 Results of the corrected resampled paired t-test. 21

Table 4.4 The average time taken for each skill, comparing RoboBulls baseline behaviors to the reinforcement learning implemented behaviors. . 23

List of Figures

Figure 1.1 An example of the physical robots that may be used in SSL games. . 2

Figure 1.2 A brief summary of how the different programs communicate. 2

Figure 3.1 Summary of the flow between training programs and the RoboBulls software system. 6

Figure 3.2 Overview of Attack Main state transitions. 7

Figure 3.3 Visualization of static probabilities in the Probability Field. 8

Figure 3.4 Visualization of dynamic probabilities in the Probability Field. . . . 9

Figure 3.5 Visualization of checking clear shots by segments, using a small number of segments for clarity. 10

Figure 3.6 Visualization of distance tolerance for detecting a clear shot. 11

Abstract

In robotics soccer, decision-making is critical to the performance of a team's Software System. The University of South Florida's (USF) RoboBulls team implements behavior for the robots by using traditional methods such as analytical geometry to path plan and determine whether an action should be taken. In recent works, Machine Learning (ML) and Reinforcement Learning (RL) techniques have been used to calculate the probability of success for a pass or goal, and even train models for performing low-level skills such as traveling towards a ball and shooting it towards the goal[1, 2]. Open-source frameworks have been created for training Reinforcement Learning models with the purpose of expanding upon existing research and allowing for further applications of RL to robot soccer[3]. This thesis aims to use these frameworks to supplement the existing publicly available resources, as well as to investigate whether implementing trained Neural Network (NN) models can improve the performance or quality of the existing USF RoboBulls software system.

Chapter 1: Introduction

The RoboCup research federation aims to promote the research of robotics and Artificial Intelligence (AI) by presenting challenges and competitions for autonomous robots that are open to the public[4]. The RoboCup Small Sized League (SSL) competes in Division A and Division B with teams of 11 and 8 robots, respectively, and is concentrated on the problem of "multi-agent coordination and control"[5].

The RoboBulls team at the University of South Florida has a software system oriented towards competing in SSL-Division B[6]. The RoboBulls team utilizes both the SSL-vision system for testing physical robots, depicted in Figure 1.1, as well as grSim for simulating an environment. grSim[7] simulates an SSL field and robots in real-time, allowing for a team's software system to control the robots over network protocols. Additionally, grSim distributes field information in the same manner as the SSL-Vision software used for physical SSL games, allowing for a team to make use of their software in both simulated and physical environments with ease. The flow of information is summarized in Figure 1.2. The RoboBulls software system is primarily written in C++ and utilizes a Skills, Behaviors, and Strategies hierarchy for team logic and control, similar to the architecture used by Carnegie Mellon University (CMU)[2]. Low-level behaviors such as traveling towards or shooting a ball are referred to as Skills. Behaviors, which can otherwise be thought of as roles, use a combination of Skills to carry out the robot's motions, but also utilize higher level logic to account for the states of the robot's teammates and opponents. An example of an attacking Behavior would consider passing the ball to another robot if the attacking robot deemed itself unable to score. At the highest level, Strategies exist and essentially act as coaches, assigning Behaviors to different

robots depending on the state of the game and high level events such as the opposing team gaining control of the ball.

Figure 1.1: An example of the physical robots that may be used in SSL games.

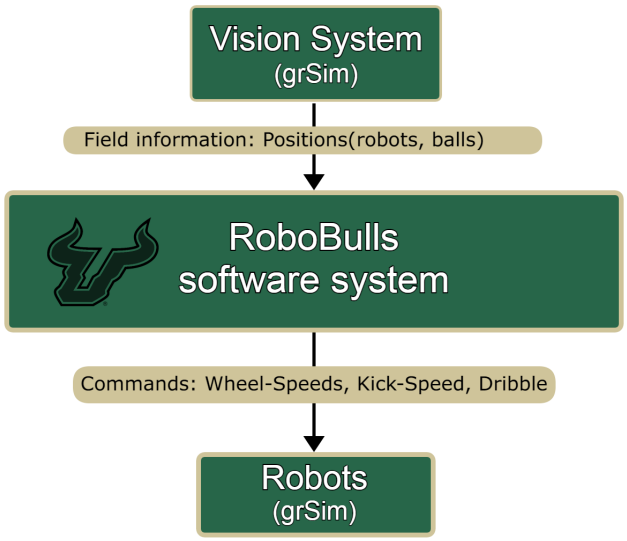

Figure 1.2: A brief summary of how the different programs communicate. In this work, grSim was used for the vision system and robots.

rSoccer[3] provides a framework for applying RL to SSL and Very Small Size leagues by leveraging the rSim simulator and OpenAIGym environments. OpenAIGym[8] is commonly used for training and evaluating RL models and provides a common interface for interacting with different environments. rSim builds upon grSim and allows the simulation to be progressed through API calls. Manual control of the environment and its progression is advantageous to training RL models because the time between frames is consistent and training

can be done faster than real-time without the manual labor of setting up the environment prior to each episode.

The Deep Deterministic Policy Gradient (DDPG)[9] is an algorithm that has been used for RL in robotics with continuous action spaces, and has been applied by CMU for learning low level SSL skills[2]. Spinning Up[10] by OpenAI is an educational resource that provides many examples and open-source implementations for RL algorithms, serving as an excellent tool for creating a link between theoretical understanding of these algorithms and their practical implementation in code.

Chapter 2: Related Works

2.1 Deep Deterministic Policy Gradient

2.1.1 Description

DDPG[9] adapts elements of Deep-Q Networks to be applicable to continuous action spaces by utilizing concepts from the Deterministic Policy Gradient[11]. Deep-Q networks have been able to develop performance akin to humans at video games such as Atari utilizing only pixels as the input[12]. DDPG uses an actor-critic network that takes advantage of target networks, which are initialized with the same parameters as their local counterparts. These target networks serve an important role in updating the critic and additionally have delayed updates to their own weights in order to improve learning stability[9]. In this work, the updates to the target networks were accomplished through polyak averaging[10]. The network is trained off-policy with a replay buffer, meaning the network is updated with experiences or samples obtained at any point in the training period[9, 10]. Noise was added to the actions during training time and it is recommended to have the robot act entirely randomly at the beginning of training to encourage exploration while training.

2.1.2 Stepping Through the Environment

The algorithm consists of an agent interacting with an environment in a series of discrete time steps, at each step observing the environment and selecting an action. The action is the output of the actor network which learns a policy, $\mu(s)$, where s is an observation of the current state. After the agent executes the action, the agent gets a reward, r, and the observation s' which is the state at the following time step. During training time, the agent

then stores the transition or experience defined as a tuple of (s, a, r, s'). The Spinning Up implementation adds whether the robot has reached the goal state after taking the action[10], making the tuple (s, a, r, s', d).

2.1.3 Updating the Networks

When the networks are updated, batches of experiences are sampled from the replay buffer and the critic is updated with gradient descent using the mean squared Bellman error loss function in Equation (2.2).

$$y_t = r + \gamma(1 - d)Q_{\text{targ}}(s', \mu_{\text{targ}}(s')) \tag{2.1}$$

$$\text{loss} = \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q(s, a) - y_t))^2 \tag{2.2}$$

Q_{targ} and $\mu_{\text{targ}}(s')$ are the outputs of the target actor and critic networks and γ is the discount factor. The actor is updated by using Equation (2.3) as the loss function.

$$\text{loss} = -\frac{1}{|B|} \sum_{s \in B} (Q(s, \mu(s)) \tag{2.3}$$

This results in gradient ascent on the output of the critic. At a high level, this can be personified as the critic trying to learn how to properly predict all possible future rewards that can be obtained by taking an action in a given state, using the target networks for the Bellman equation. The actor attempts to learn the optimal policy by determining which action will yield the highest output from the critic, ideally converging to a policy that returns actions that will ultimately maximize all future rewards. The parameters of the target networks are then updated using polyak averaging in Equation (2.4), where ρ is a value between 0 and 1.

$$\theta_{\text{targ}} = \rho\theta_{\text{targ}} + (1 - \rho)\theta \tag{2.4}$$

Chapter 3: Approaches

3.1 Environment/Setup

The RoboBulls software system is currently running on Windows 10 and built using Qt 5.14.2 and MSVC2017 64 bit. Pytorch[13] modules were loaded by linking with LibTorch 1.10.1 CUDA 11.3 Binaries. TensorFlow[14] modules were loaded through libTensorFlow 2.7.0. For the ML approach for determining a successful action, Jupyter Notebook was used in an Anaconda environment on Windows 10 with TensorFlow and sklearn libraries.

In order to train the RL models, an Ubuntu 20.04 WSL2 environment with python version 3.7.11 was used, as most of the package requirements would not build on Windows. rSoccer was used for the simulator/training environments and the Spinning Up package was used for the testing and training algorithms and modules. The DDPG Pytorch implementation from Spinning Up was modified to export the Actor module as TorchScript for use in the RoboBulls system.

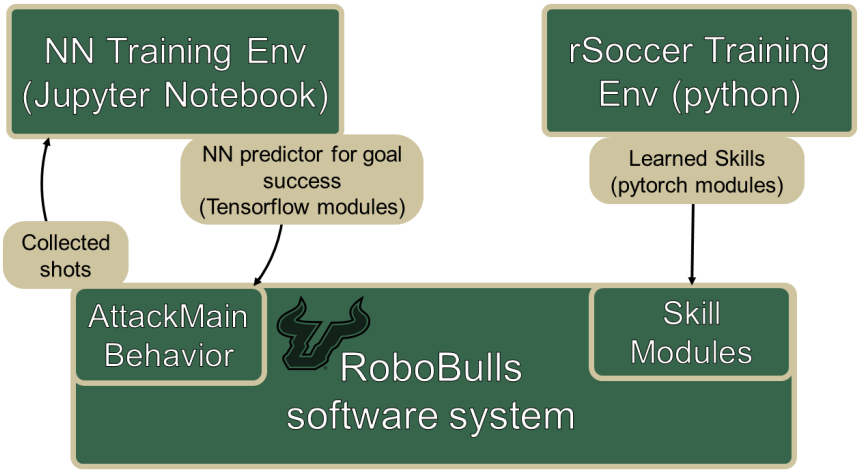

Figure 3.1: Summary of the flow between training programs and the RoboBulls software system.

3.2 RoboBulls: Attack Main

Existing RoboBulls Software and strategies were finalized and improved to set a baseline for the performance comparisons. This paper primarily focuses on studying the attacking behaviors, therefore, the RoboBulls Attack Main behavior serves as the baseline for comparisons. A robot that holds possession of the ball gets assigned the Attack Main Behavior and may reside in one of three states, which are depicted in Figure 3.2.

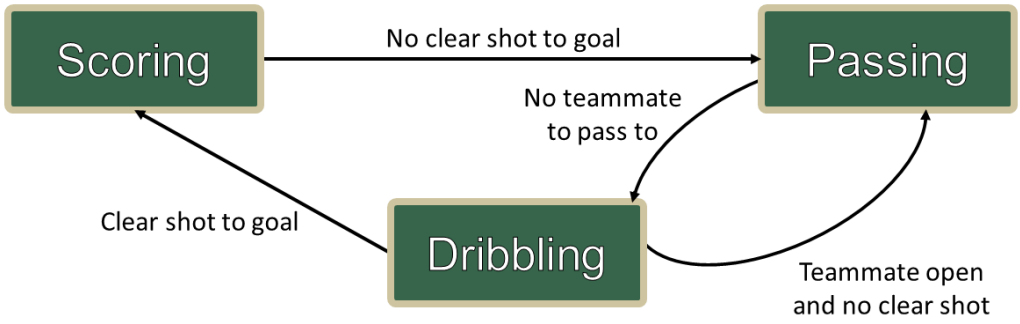

Figure 3.2: Overview of Attack Main state transitions.

- Scoring: In the Scoring state the robot will face a clear segment of the goal area and shoot the ball. While in this state, if the clear shot becomes obstructed, it will instead switch to the Passing state and attempt to pass to a teammate.

- Passing: The robot will pass to a teammate if the teammate is in a suitable receiving position as deemed by the Probability Field and there is no robot intersecting the straight line path between the passing and receiving robot. If there are no teammates that are available, the robot will switch to the Dribbling state.

- Dribbling: In the Dribbling state, the robot will travel to a location in the field with the highest pseudo-probability of scoring given by the Probability Field. The robot will switch to Scoring or Passing whenever there is an available teammate or clear shot to goal.

For purposes of this work, we will not consider the Passing state, but assume the Attack Main robot is in Scoring or Dribbling.

3.2.1 The Probability Field

When the attacking robot cannot feasibly shoot and score, it will start dribbling towards a point that puts the robot in a better position to score. The Attack Main robot determines the optimal scoring position using a method called the Probability Field (PF). The PF is a matrix of pseudo-probabilities that determines the coordinates at which a robot will have better opportunity of scoring. The PF is initialized with static probabilities using Equation (3.1). The static probabilities can be visualized in Figure 3.3.

$$\text{probability}_{x,y} = (1.0 - 0.000333 * \text{distance}_{x,y-\text{goal}}) * 2 + (1.0 - 0.674 * \theta_{\text{goal}}) \tag{3.1}$$

Points that fall within the opposing team's goalkeeper area, are more than 3 meters away from the goal, or have larger than an 85 degree angle to the goal are considered to have a pseudo-probability of 0.

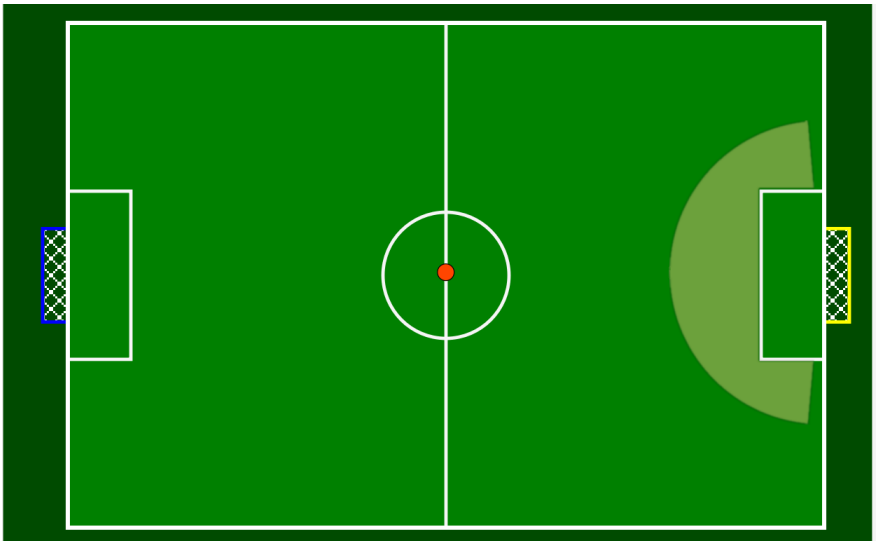

Figure 3.3: Visualization of static probabilities in the Probability Field. The area of the field that is highlighted has increased static probabilities.

Dynamic probabilities are calculated during each cycle of the game, as seen in Figure 3.4. Initially, opponent robots that have less than a ball's diameter and 0.01 meters between each other are clustered together. These clustered robots are perceived as obstacles by the robot shooting the ball to the goal. For each of these clusters, the gradient of the line from the top of the goal area that is tangent to the top-most robot is calculated, with the gradient of the line intersecting the bottom goal post and tangent to the bottom-most robot also being calculated. Any point that falls within these two lines is set to a dynamic probability of -10, disqualifying these points from being chosen as the target shooting position.

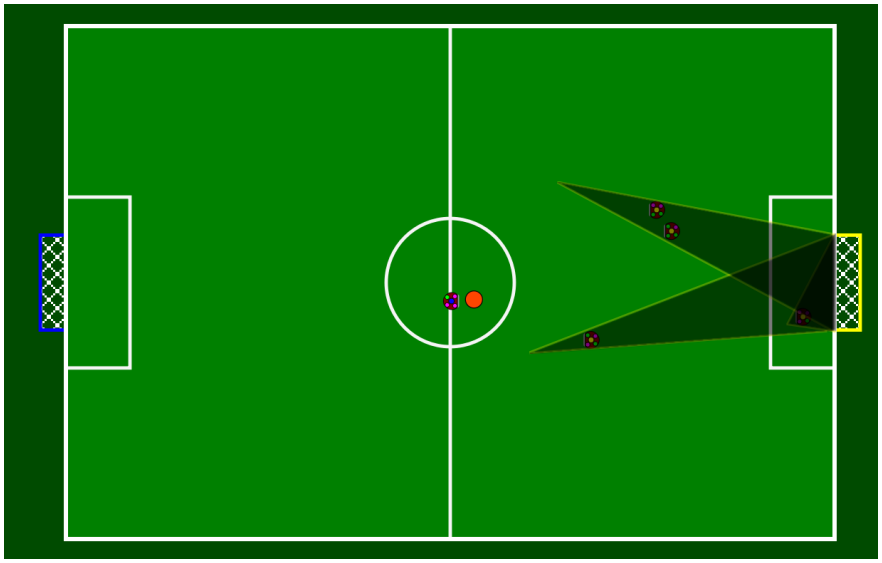

Figure 3.4: Visualization of dynamic probabilities in the Probability Field. The shaded area represents points on the field with a lower dynamic probability.

The sum of the static and dynamic probabilities are used to determine the point with the highest probability, which is then selected as the target point for the attacking robot to travel towards. It is important to note that during the traversal towards the target point, if there is a clear shot to goal, the robot will break from the dribbling behavior and attempt to score.

3.2.2 Detecting a Clear Shot

In order to determine if or where there is a clear shot to goal, the robot breaks the goal area into several discrete segments, as seen in Figure 3.5. At each segment the straight line from the segment to the ball is used to check whether any robot intersects or comes within a certain distance threshold to the line which is visualized in Figure 3.6. If there is no obstacle, the segment is indicated as a clear shot. Consecutive segments that are indicated as clear shots are then grouped together, and the mid-point of the largest consecutive segment is deemed the shooting point. This is similar to the methodology used by Tigers Mannheim[15] to determine a score chance; however, the RoboBulls implementation does not actually determine any chance that a shot will be successful, but instead only selects the largest segment for determining the target. Tigers Mannheim uses the distance and angle to the goal-point in determining the chance a shot will be successful.

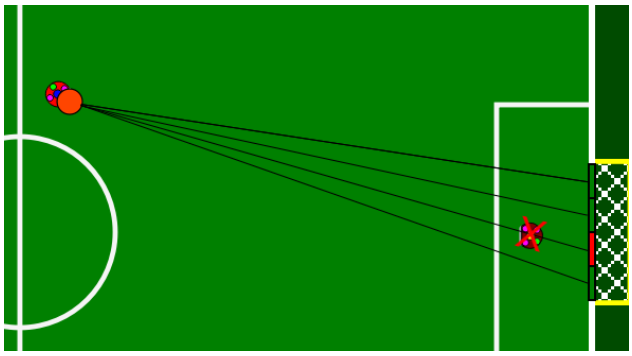

Figure 3.5: Visualization of checking clear shots by segments, using a small number of segments for clarity.

A main criticism of the current RoboBulls implementation is that the robot will shoot if there is a clear shot to goal without taking into consideration the distance to this point. The following section seeks to rectify this issue with the use of NNs.

3.3 Neural Network

The utilization of NNs to calculate the probabilities that a shot, pass or goal, will be successful has been previously researched by KIKS at National Institute of Technology,

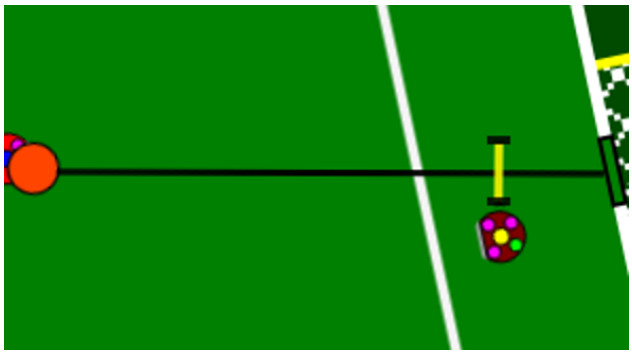

Figure 3.6: Visualization of distance tolerance for detecting a clear shot.

Toyota College in Japan[1]. The data set from their study is not public, and the data set of more than 100 samples was only described as being obtained from grSim, so it may be unfair to make direct comparisons when analyzing the results.

3.3.1 Data Set

The data set was collected from the RoboBulls software system with grSim being used as the environment. Data was collected during the cycle that a robot first kicks the ball, with the success of the shot, S, being stored when the trial ends. A goal was considered a failure if the opposing robot intercepted or collided with the ball. A goal was considered successful if the ball entered the goal area without colliding with the enemy robot. Any shot by the attacking robot that was directed outside of the goal area was discarded. R represents the location and velocity of a robot and is defined in Equation (3.2) where P_x and P_y are the coordinates of the robot in meters and V_x and V_y are the velocity of the robot in meters per second. Equation (3.3) represents the data collected, as all features relevant to goal prediction were attempted to be captured.

$$R = (P_x, P_y, V_x, V_y) \tag{3.2}$$

$$\text{data} = (R_{\text{robot}}, \theta_{\text{robot}}, R_{\text{opp}}, P_{x_{\text{goal}}}, P_{y_{\text{goal}}}, S) \tag{3.3}$$

Table 3.1: The number of samples collected using each method.

	# Success	# Failure
Attack Behavior (Method 0)	62	107
Stationary Shot (Method 1)	103	270
Total	165	377

R was collected for both the attacking and opponent robot, θ_{robot} is the angle the robot is facing, P_x and P_y are the coordinates for the point the robot is shooting to, and S is either `True` or `False`.

The data displayed in Table 3.1 was collected using a single shooting robot and a single goalie robot. The RoboBulls goalie behavior defends extremely well against a single attacker, therefore, the goalie was slowed to half its normal speed in order to reduce the time taken to gather the data set, as well as obtain a more balanced ratio of successful and failed shots. The data was collected using two main methods:

1. Attack Behavior: The robot was started with the ball on the left side of the field, its home side, and acted using the Attack Main behavior.

2. Stationary Shot: The robot was started at fixed locations within the center, lower-right, upper-right, and mid-right sections of the field. The robot was restricted to this quadrant and, in most cases, remained stationary, rotating only to shoot.

Gathering data using these two methods sought to reduce bias potentially introduced by taking samples exclusively from positions the attacking behavior would navigate to. In order to further diversify the robots actions, modifications were made to the clear shot distance tolerance so a diverse set of goal points were sampled. For Method 1, a variable restriction was placed on the x-coordinate at which the robot was allowed to start shooting, in an attempt to further diversify the collected shots.

Table 3.2: The number of samples used in the training and validation sets.

	# Success	# Failure
Training Set	111	252
Validation Set	54	125

The validation set was created with sklearn to create a train/test split with stratification on the successes. This partitioned 2/3 of the data for training and the remaining 1/3 for validation.

3.3.2 Model Training

The features selected for input into the NN are shown in Equation (3.4).

$$\text{input} = (\theta_{\text{robot}-\text{opp}}, d_{\text{goal}}, d_{\text{opp}}, P'_{x_{\text{opp}}}, P'_{y_{\text{opp}}}) \tag{3.4}$$

where $\theta_{robot-\text{opp}}$ is the angle between where the robot is facing and the angle to the opponent robot, d_{goal} is the distance to the point to which the ball is being shot, d_{opp} is the distance to the opponent robot, and $P'_{x_{\text{opp}}}$ and $P'_{y_{\text{opp}}}$ are the coordinates of the opposing robot in reference to the attacking robot.

The architecture of the NN in Table 3.3 was chosen to be similar to KIKS[1], as a Deep NN with a relatively small number of units in the hidden layers should be appropriate for the small number of features and samples that are currently available. This architecture also seemed to perform well on training and validation sets without over-fitting to the training data. The Adam optimizer was used with a learning rate of 0.05 and Binary Crossentropy as the loss function. The batch size was 16 and the Model would be trained for 64 epochs with a TensorFlow callback to reduce the learning rate on a plateau for loss on the validation set. The training was then continued with early stopping on the validation loss for a maximum number of 512 epochs, restoring the weights that achieved the lowest loss on the validation set.

Table 3.3: Neural network architecture.

Layer	# Units	# Activation
Batch Normalization	-	-
Dense	6	ReLU
Batch Normalization	-	-
Dense	3	Swish
Output	1	Sigmoid

Table 3.4: Testing scenarios.

Description	Tolerance	# Trials
Attack Behavior	Small	33
	Medium	33
	Large	35
Still Shots		
Center	Small	≈ 25
Lower-Right	Small	≈ 25
Mid-Right	Small	≈ 25
Upper-Right	Small	≈ 25
Total		201

Table 3.4 shows 201 samples of testing data that was held aside for evaluating the accuracy of the model. After testing the model, it was then deployed into the Clear Shot detection and the behavior was modified. In addition to detecting clear segments described in Section 3.2.2, each clear segment was analyzed by the trained NN model to determine whether or not a shot to the segment would be successful.

3.4 DDPG

As CMU proposed in their research[2], RL was utilized to train Skills existing at the lowest level of the logic hierarchy for the RoboBulls software system. This adds portability to these learned Skills and can be leveraged in a variety of behaviors. DDPG was used in order to learn both the Go-to-Ball and Shoot-Towards-Goal Skills.

To allow for training and testing within the RoboBulls software system using grSim as the simulated environment, a DDPG training algorithm, actor network, and critic network was implemented in LibTorch/Pytorch C++. The objective was to use RoboBulls' clasically

implemented policies to burn in the replay buffer so that demonstrations could be used for RL[2, 16]. Due to the training time required to learn a good policy, implementations for this approach were ultimately unsuccessful. For example, a network learning the Go-to-Ball skill started demonstrating convergence to an appropriate policy at around $250,000$ cycles. With each cycle taking around 0.03 seconds, this would take approximately 2 hours. The time it takes to manually reset the grSim environment roughly doubled this time, taking approximately 4 hours to even determine if the the network is learning anything. This also yielded a poor environment for tweaking the NN's hyper-parameters, observation space, and reward functions. Another noteworthy drawback of this environment is that grSim operates in real-time. If a process, such as an update to the NN, took more time than a single cycle, inconsistencies could be created in the time between observations.

In CMU's research[2] they utilized their own CMDragons simulator, that offers a step mode for the simulation so that it "will only advance when a new command is sent." Unfortunately, the CMDragons simulator currently is not publicly available. However, the rSoccer environment seeks to provide an open-source simulator that is "optimized for reinforcement learning experiments"[3]. This simulator operates in a similar fashion and does not advance the simulation unless a step function is called on the environments API.

rSoccer was used for the OpenAI learning environments and Spinning Up python implementations of the DDPG algorithm were used and modified to train the models[10]. These trained models had to be exported as TorchScript in order to be used in the RoboBulls C++ environment. It is important to note that the observations that are input into the actor network must follow the same conventions.

The following environments that will be discussed both operate with 0.03 seconds between each cycle, in order to maintain consistency with the RoboBulls software system which operates at 33 cycles per second. The RL models were trained and tested inside the rSoccer environment. A final benchmark for the trained actor modules was executed inside the RoboBulls software system using grSim as the simulated environment.

3.4.1 Go-to-Ball

For the Go-to-Ball Skill, the input state, s, was defined as shown in Equation (3.5), where θ_{ball} is the difference in the robot's orientation and the angle to the ball in radians. The translational velocity for the robot is in meters per second and is the velocity in relationship to the robot's orientation. The angular velocity is in radians per second. As CMU noted[2], keeping the state variables defined in relationship to the robot allow the learned policy to be more generalized. This also allows the model to avoid learning spurious observations about the robot's location in the field. The action space is defined in Equation (3.6).

$$s = (sin(\theta_{\text{ball}}), cos(\theta_{\text{ball}}), d_{\text{ball}}, V_{x_{\text{robot}}}, V_{y_{\text{robot}}}, V_{\theta_{\text{robot}}}) \tag{3.5}$$

$$a = (V_x, V_y, V_\theta) \tag{3.6}$$

The velocities (V) are in relation to the robot and in meters per second and radians per second, respectively.

The reward function proposed in CMU's research [2] was used with a modification made to r_{contact} shown in Equation (3.7). The unmodified equations utilized in the reward are defined in Equations (3.8), (3.9), and (3.10).

$$r_{\text{contact}} = \begin{cases} 100 & d_{\text{ball}} \leq 0.2 \\ 0 & d_{\text{ball}} > 0.2 \end{cases} \tag{3.7}$$

$$r_{\text{distance}} = \frac{5}{\sqrt{2\pi}} exp(\frac{-d_{r-b}^2}{2}) - 2 \tag{3.8}$$

$$r_{\text{orientation}} = \frac{1}{\sqrt{2\pi}} exp(-2\frac{\theta_{r-b}}{\pi^2}) \tag{3.9}$$

$$r_{\text{total}} = r_{\text{contact}} + r_{\text{distance}} + r_{\text{orientation}} \tag{3.10}$$

The episode length would be 330 cycles, or 9.9 seconds, and would be successfully completed if the robot comes within 0.2 meters of the ball.

3.4.2 Shoot-Towards-Goal

The state input from CMU's research[2] was used and is defined in Equation (3.11).

$$s = (P_{x_{\text{ball}}}, P_{y_{\text{ball}}}, V_{x_{\text{ball}}}, V_{y_{\text{ball}}}, V_{\theta_{\text{robot}}}, d_{r-g}, sin(\theta_{\text{top}}), cos(\theta_{\text{top}}), sin(\theta_{\text{bottom}}), cos(\theta_{\text{bottom}})) \tag{3.11}$$

$P_{x_{\text{ball}}}$ and $P_{y_{\text{ball}}}$ are the position of the ball in relation to the robot in meters. $V_{x_{\text{ball}}}$, $V_{y_{\text{ball}}}$ and $V_{\theta_{\text{robot}}}$ are velocities defined in Section 3.4.1. θ_{top} and θ_{bottom} are the differences between the robot's orientation and the angles to the top and bottom of the goalpost in radians. d_{r-g} is the distance from the robot to the center of the goalpost. The action space is defined in Equation (3.12).

$$a = (V_\theta, K) \tag{3.12}$$

K corresponds to the kicking action. This is simplified from the CMU action space as the robot will dribble the ball until it has kicked and always kicks at max kicking velocity.

Using the same reward function from CMU's research, defined in Equation (3.13), often resulted in the robot learning to never kick the ball, resulting in the addition of r_{kick}, shown in Equation (3.15). α is the difference of the angles between the robot and the top and bottom goalpost, representing the angle between goalposts in relation to the robot. β represents the difference between the robot's current orientation and either the left and right goalpost, choosing whichever goalpost results in the largest value. When $\alpha \geq \beta$, the robot is facing between the two goalposts. In order to prevent the robot from holding onto the ball, Equation (3.14) was included in addition to CMU's original reward function. The robot is given a constant penalty every cycle that the robot is not facing towards the goal, or is

facing the goal and has not kicked yet. Additionally, there is a sparse reward on the cycle that the robot has kicked and is facing the goal.

$$r_{\text{face-goal}} = \begin{cases} 0.05(\alpha - \beta)|V^B| & \alpha \geq \beta \\ (\alpha - \beta)|V^B| & \alpha < \beta \end{cases} \tag{3.13}$$

$$r_{kick} = \begin{cases} 10 & \alpha \geq \beta \wedge \text{Just Kicked} \\ \text{-0.25} & \alpha < \beta \vee \neg \text{ Kicked} \end{cases} \tag{3.14}$$

$$r_{\text{total}} = r_{\text{face-goal}} + r_{\text{kick}} \tag{3.15}$$

The episode length would be 100 cycles, or 3 seconds, and would be successfully completed if the shot landed in the goal area. The episode would be terminated if the robot shot the ball outside the field.

3.4.3 Go-to-Ball-and-Shoot Behavior

The Go-to-Ball and Shoot-Towards-Goal Skills were combined to make a Go-to-Ball-and-Shoot Behavior. If the robot is not in possession of the ball, the robot uses the Go-to-Ball Skill to travel towards the ball. When it comes within 0.2 meters of the ball, it travels a human implemented path to pick up the ball, in similar fashion to CMU's implementation[2]. The robot travels 0.5 meters per second towards the ball until it acquires the ball on the dribbler, at which point it switches to the learned Shoot-Towards-Goal Skill.

Chapter 4: Results

4.1 Neural Network Results

The trained NN achieved an accuracy of 85.07% on testing data, which is competitive with the accuracy of 84% achieved by KIKS[1]. However, it should be noted that a direct comparison between the two accuracies should not be made, as their model was trained and tested on their own collected data set. For comparison, a naive use of a Linear Logistic Regressor (LLR) fitted to the training data achieved an accuracy of 86.07% on the testing data. The confusion matrices for the NN and LLR can be seen in Tables 4.1 and 4.2. Achieving a better accuracy by the LLR lead to an investigation on whether there were any advantages to using a NN over LLR.

Table 4.1: Neural network confusion matrix.

		Predicted	
		Failure	Success
Actual	Failure	102	20
	Success	10	69

Table 4.2: Logistic regression confusion matrix.

		Predicted	
		Failure	Success
Actual	Failure	107	15
	Success	13	66

Stratified 10-10 fold cross validation, also commonly referred to as repeated stratified 10-fold cross validation[17], was used alongside the corrected resampled paired t-test[18] for comparing the accuracy achieved by the proposed NN architecture and a naively implemented LLR. In 10-fold cross validation, the training data is split into 10 unique non-overlapping

folds. The performance of the models are evaluated by creating a training split of 9 of the folds, and using the single remaining fold as the test split. In 10 fold cross validation, each of the folds will be used for testing, with the remaining 9 used as the training set. This process can be repeated 10 times, resulting in 10-10 fold cross validation, with a total of 100 models built and evaluated. Both the NN and LLR were trained and tested using the same splits, allowing the use of the corrected resampled paired t-test. The corrected resampled paired t-test was calculated using the variance of a subset of a population for calculating the variance of the paired differences, s^2, due to potential bias in the data set. The equations are shown in Equations (4.1) and (4.2).

$$x_{ij} = a_{ij} - b_{ij} \tag{4.1}$$

$$t = \frac{\frac{1}{k \cdot r} \sum_{i=1}^{k} \sum_{j=1}^{r} x_{ij}}{\sqrt{(\frac{1}{k \cdot r} + \frac{n_2}{n_1}) s^2}} \tag{4.2}$$

x_{ij} is the difference in accuracy of the models on the same iteration and split, where a represents the accuracy of the NN, and b represents the accuracy of the LLR. n_1 and n_2 are the sizes of the training and testing splits, respectively. As there were 542 samples, all 10 folds could not contain an equal number of samples, with 8 folds containing 54 samples and 2 folds containing 55. This resulted in different sizes for the training and testing splits, therefore, the average of the training and testing sizes, 487.8 and 54.2 were used for n_1 and n_2. The NN was trained in the same manner as in Section 3.3.2, where the NN set aside a third of the allocated training split for early stopping on the loss of unseen data. Comparing the NN trained in this manner against the naive LLR resulted in a t-value of 1.9666 and p-value of 0.0520, which does not suggest that there is a statistically significant difference in the performance of the classifiers. However, one concern with this evaluation is that the NN is starving itself of training data by setting aside a third of the already reduced training split. Lowering the percentage of the training data used by the NN for early stopping from 0.33% to 0.25% and repeating the test resulted in a t-value of 2.0552 and p-value of 0.0425,

Table 4.3: Results of the corrected resampled paired t-test. $\bar{a}$ is the mean accuracy of the NN, $\bar{b}$ is the mean accuracy of the LLR, $\bar{x}$ is the mean of the paired difference. The percentage of the NN's allocated training data used for early stopping is shown.

% Early Stopping	$\bar{a}$	$\bar{b}$	$\bar{x}$	s^2	t	p
33%	87.4024	84.4512	2.9512	18.5950	1.9666	0.0520
25%	87.5478	84.5923	2.9556	17.0759	2.0552	0.0425

which does suggest a statistically significant difference between the performance of the NN and LLR. The results of both tests are summarized in Table 4.3. It is important to note that this data set has potential bias in the collection process detailed in Section 3.3.1. Further analysis and collection of new data should be conducted in order to determine whether there is an advantage of using an NN over LLR. With a limited size of data and conflicting results, it would be useful to cross-reference a comparison of KIKS'[1] NN performance in comparison to LLR or other ML models; however, they only listed results of different NN architectures on their testing set of 50 samples and did not perform cross validation.

Implementing the NN into the Attack Main behavior, defined as AttackNN, as proposed in Section 3.2.2, resulted in 80 out of 100 successful goal shots. This is a significant improvement when compared to the baseline Attack Main behavior which achieved 28 out of 100 successful shots. The enhancement of adopting goal prediction from the NN into AttackNN gave a improvement of 186% in successfully scoring compared to the baseline. One concern may be that the behavior implemented using the predictor may not score as successfully against other goalies, leaving this paper's proposed methodology unusable in SSL games. If the predictor has overfit to the data it was trained on, it may result in either a behavior that is too conservative in the shots that it takes, or inversely, too lenient. In order to determine if using the predictor can still increase the performance of the attacking behavior when facing an unseen opponent, additional trials were performed against the RoboBulls goalie behavior operating at full-speed. Out of 50 shots taken, AttackNN scored 35 times, with the baseline behavior scoring 19 times. AttackNN achieved an increase of 84% in the number of successful goals in relation to the baseline behavior, even when faced against the full-speed

goalie. This demonstrated an improvement in performance from using a predictor, despite not being trained on data collected with this opponent. Ideally, testing this behavior against another SSL team's goalie would help in verifying that this predictor generalizes well against other opponents, as there still may be patterns in the goalie's behavior that play a role in the prediction made by the model that are present in both the full-speed and half-speed goalies.

4.2 DDPG Results

In order to test the different behaviors, as well as a human participant controlling a robot via a video-game controller, two trials were performed. In both trials, the robot started at the coordinates (-4, 0), in meters, facing East with an orientation of 0 degrees. In the first trial, results seen in Table 4.4a, the ball was placed in the center of the field. For the second trial, results seen in Table 4.4b, the ball was placed at coordinates (0, 1.5).

The behavior learned with DDPG outperformed the human and the RoboBulls behavior during both trials in terms of average total time taken. The RoboBulls behavior was the most accurate in terms of shooting the ball into the goalpost, as it did not miss a single goal across both trials. The DDPG learned behavior was the only behavior which missed a goal on the first trial, but outperformed the human participant on the second trial by missing only 4 of its shots rather than 11. As expected, the human had the greatest variance in time taken across all Skills and trials. When comparing the total time taken, the RoboBulls behavior had slightly less variance on both trials. Observing the variance of time taken at the Skill level, neither the RoboBulls behavior nor the DDPG learned behavior consistently performed better. However, the DDPG Skills had variances in time taken more comparable to the RoboBulls behavior than the human participant.

Table 4.4: The average time taken for each skill, comparing RoboBulls baseline behaviors to the reinforcement learning implemented behaviors.

(a) Trial 1

Behavior	Go-to-Ball		Shoot		Total		# Goals
	μ	s^2	μ	s^2	μ	s^2	
DDPG	2.9217	0.0008	0.2604	0.0027	3.1821	0.0048	23
RoboBulls	3.4998	0.0040	0.2665	0.0006	3.7662	0.0044	25
Human	3.483	10.9620	0.4032	0.1709	3.8862	12.8650	25

(b) Trial 2

Behavior	Go-to-Ball		Shoot		Total		# Goals
	μ	s^2	μ	s^2	μ	s^2	
DDPG	3.1262	0.0196	0.4031	0.0012	3.5294	0.0196	21
RoboBulls	4.9846	0.0141	0.4713	0.0057	5.4560	0.0193	25
Human	3.1691	0.3725	1.1236	0.3481	4.2927	1.1300	14

Chapter 5: Conclusion and Future Work

In order to further improve the data collection process used to train the NN goal predictor, efforts should be made to collect samples with less bias. Suggestions for future work on the RoboBulls system would be to give the Attack Main behavior a randomized distance tolerance for detecting a clear shot as well as a randomized field coordinate it must reach before the robot behavior is allowed to shoot. Additionally, the goalie could be given a randomized speed modifier within the range of half to full-speed. Ideally, samples should be collected across games against multiple teams in order to have a variety of samples. After gathering data using these proposed methods, it would be suggested to revisit feature selection and compare the NN to LLR to determine whether there are statistically significant performance differences when evaluating their accuracies using data with less bias.

Future work into applying RL to SSL robot soccer should consider the use of DDPG from Demonstrations (DDPGfD)[16] and Distributed Distributional DDPG (D4PG)[19], as well as the many other variations of the original DDPG algorithm. Both DDPGfD and D4PG share the use of n-step returns and prioritized replay buffer, and have performed well even in problems with sparse rewards. The authors of D4PG[19] argue that the use of n-step returns contributes a larger amount of performance increase rather than prioritized replay buffer. CMU[2] did state they burned in the replay buffer which was utilized in DDPGfD, but do not clarify whether they adopted any of the other modifications. While the DDPG learned behaviors perform faster than the RoboBulls baseline Skills, in their current state they would likely be unusable in SSL games due to the lack of path planning and less accurate scoring. In order to learn Skills usable in actual games, future works will need to include the positions of other robots in order to perform obstacle avoidance or score against a goalie. The reward

function for the Shoot-Towards-Goal Skill will likely need to use a sparse reward function that receives a reward upon scoring and utilize a n-step return algorithm for learning, as the current reward function does not take into account any opposing robot and may result in the policy prioritizing shooting towards certain orientations towards goal. The failure of using CMU's reward function for the Shoot-Towards-Ball Skill was likely due to a local maxima in the reward function, seen in Equation (3.13). When the robot is facing away from the goal area, the robot is penalized by a factor of the velocity of the ball, therefore, the maximum reward may result from taking an action that does not move to face the goal, or even move at all, as motion will give the ball a non-zero velocity and result in penalty. Additionally, the reward function will return the greatest reward or penalty just after the robot has kicked the ball, as the ball velocity will be greatest at that point. This may result in the policy converging too quickly to a policy that does not shoot the ball. After collecting results, training using CMU's reward function was revisited, but only 2 of 5 trained actors would learn policies that would shoot the ball, with the other 3 learning to hold onto the ball. Using demonstrations in CMU's learning process appears to mitigate these potential issues with the reward function. The work of Matheron et al.[20] observed the failures of using DDPG for problems with sparse rewards, finding that if the reward is not found early enough in the learning process, the algorithm may ignore the reward in samples found later. It is possible that the models which failed in training would have more experiences shooting away from the goal, and the critic has learned to estimate that the largest reward will result from holding onto the ball.

One of the primary limiting factors in this work is that the RoboBulls system is written in C++, with many relevant libraries to RL only being accessible from python. OpenAIGym had a deprecated HTML API, which would have allowed for interacting with gym environments from C++, however, it seems the demand for this has waned. There may be ways to interact with rSoccer from C++ such as embedding python into the existing C++ application, or by bypassing the user-friendly python API and accessing the underlying simulator,

rSim, via its C++ source code. These options will require thorough planning and research, however, if the RoboBulls system is intended to be used in future ML works, porting to python would be recommended at this time. The system needs revision to fix consistency issues between independent modules with differing functionalities sharing dependencies to variables that are used system-wide, as well as the unfinished GUI and behaviors. Since these revisions will require a thorough inspection of modules across the system, it may be worthwhile to port the system at the same time, allowing access to many more ML and RL packages and implementations, such as the many variations of DDPG. Bridging the software gap between the existing RoboBulls system and the RL training framework, would allow for human-coded behaviors to be utilized within the environments. More complex training environments could be created, such as a shooting behavior that receives a sparse reward when it scores against the RoboBulls goalie behavior. The inability to interface with rSoccer from the RoboBulls system severely limits learning from scenarios more practical for learning behaviors and skills for competitive games. Additionally, being able to interface with rSoccer from the RoboBulls system would allow for burning in the replay buffer, which has been shown to be advantageous for problems with sparse rewards[16, 2].

5.1 Summary

Implementing a NN into the RoboBulls Attack Main behavior resulted in an increase of 186% in goal success against the half-speed goalie, which the training data was collected against, and an increase of 84% in goal success against the full-speed goalie which was not utilized in the data collection process. Comparing the NN against a LLR in 10-10 fold cross validation with the corrected resampled paired t-test resulted in a p-value of 0.0520. However, the NN was trained using the same hyper-parameters as the deployed model, which held aside 33% of the allocated training data from each run for early stopping, potentially starving itself of data from the already reduced training split. Reducing the percentage of the training split used for early stopping to 25% resulted in a p-value of 0.0425. DDPG was

utilized for training basic SSL Skills that could outperform the RoboBulls behavior in time taken, but ultimately sacrificed accuracy in shooting the ball towards the goal. DDPGfD[16] and D4PG[19] would be suggested in future adaptations of this work due to their use of n-step returns and prioritized replay buffer in order to adapt to Skills that may use sparse rewards.

References

[1] Yusei Naito, Shin Ohno, Yuta Imaeda, Akihito Odanaka, Yasutaka Tsuruta, Ryoma Mitsuoka, Taisuke Tane, Masato Watanabe, and Toko Sugiura. Kiks extended team description for robocup 2020, 2020.

[2] Devin Schwab, Yifeng Zhu, and Manuela Veloso. Learning skills for small size league robocup. In Dirk Holz, Katie Genter, Maarouf Saad, and Oskar von Stryk, editors, *RoboCup 2018: Robot World Cup XXII*, pages 83–95, Cham, 2019. Springer International Publishing.

[3] Felipe B. Martins, Mateus G. Machado, Hansenclever F. Bassani, Pedro H. M. Braga, and Edna S. Barros. rsoccer: A framework for studying reinforcement learning in small and very small size robot soccer, 2021.

[4] RoboCup Federation. Objective. https://www.robocup.org/objective. Accessed Feb. 16, 2022 [Online].

[5] Robocup small size league. https://ssl.robocup.org/. Accessed Feb. 16, 2022 [Online].

[6] USF Robobulls. Home. http://usfrobobulls.org/. Accessed Feb. 16, 2022 [Online].

[7] Valiallah Monajjemi, Ali Koochakzadeh, and Saeed Shiry Ghidary. grsim – robocup small size robot soccer simulator. In Thomas Röfer, N. Michael Mayer, Jesus Savage, and Uluç Saranlı, editors, *RoboCup 2011: Robot Soccer World Cup XV*, pages 450–460, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[9] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[10] Joshua Achiam. Spinning up in deep reinforcement learning, 2018.

[11] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[14] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[15] Mark Geiger, Chris Carstensen, Andre Ryll, Nicolai Ommer, Dominik Engelhardt, and Felix Bayer. Tigers mannheim (team interacting and game evolving robots) extended team description for robocup 2017, 2017.

[16] Matej Vecerík, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin A. Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *CoRR*, abs/1707.08817, 2017.

[17] R.R. Bouckaert and E. Frank. Evaluating the replicability of significance tests for comparing learning algorithms. In *Advances in Knowledge Discovery and Data Mining*, volume 3056, pages 3–12, 01 2004.

[18] Claude Nadeau and Y. Bengio. Inference for the generalization error. *Machine Learning*, 52:239–281, 01 2003.

[19] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[20] Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud. The problem with ddpg: understanding failures in deterministic environments with sparse rewards. *ArXiv*, abs/1911.11679, 2019.

Appendix A: Copyright Permissions

The permission below is for the use of the USF logo in Figures 1.2 and 3.1.

Re: Use of USF logo in Thesis

Demontre Smith <demontre@usf.edu>
Wed 3/9/2022 12:03 PM
To: Justin Rodney <justinrodney@usf.edu>
Hi Justin,

I hope all is well. Please use this email as a one-time use agreement to use the university logo in your thesis.

Thanks

Tré Smith
Director of Marketing
University Communications and Marketing
University of South Florida
Tampa campus
813-974-4724
--

From: Justin Rodney <justinrodney@usf.edu>
Date: Wednesday, March 9, 2022 at 10:42 AM
To: Demontre Smith <demontre@usf.edu>
Subject: Re: Use of USF logo in Thesis

Hello Tré Smith,

I wanted to confirm that I have permission to use the USF logo, located at the link you sent me, within figures used in my electronic thesis.

Thank you,
Justin Rodney

About the Author

Justin Rodney is a graduate student at the University of South Florida where he is studying Computer Science towards a master's degree. From an early age, he was drawn to keyboards; whether it was in front of a monitor where he was introduced to photo editing software and PC gaming by his father, or sitting at a piano bench for a lesson to which his mother took him. Being classically trained in the art of music since the age of four, and spending a significant portion of his adolescence with a laptop in hand, would provide the groundwork for his future pursuit of higher education in fields involving both music and computers. Justin still carries his passion for both the arts and the tech world with him to this day, composing, producing and engineering music in his free time, while simultaneously working on machine learning algorithms at the same desk.