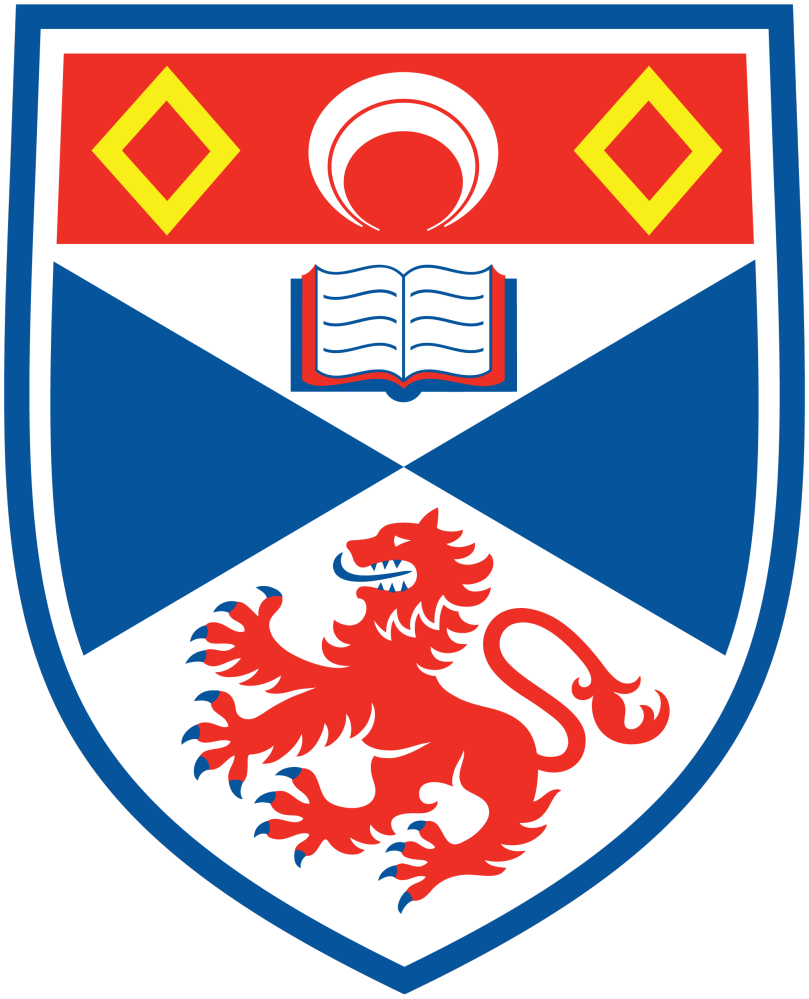
Reinforcement Learning with a NAO Robot

Andre M. Gras*

University of St Andrews

E-mail: amg25@st-andrews.ac.uk



Supervisor: Dr. Michael Weir

Abstract

This project explores learning in a humanoid robot, the NAO (V5). The essay explores simple Computer Vision, specifically, the Circular Hough Transform, using the robot to detect a red ball. Then, the experiments move on to using Computer Vision to complete a task (kicking a red ball), then utilizing the skills and techniques learned to develop a Reinforcement Learning agent with autonomous attraction to red and repulsion from other colors (namely blue and green). The robot can detect a red circle and go towards and kick a red ball that is detected. Furthermore, over many experiments, it was found that the robot can learn to move towards red autonomously with short-term rewards, but will take much more training to move towards red when the rewards become more long-term or episodic (versus stepwise). The robot manages to do these tasks purely in a simulated environment in Webots, and the behavior has not been ported to a real-life environment yet.

Contents

Chapter 1: Introduction

Motivation

The motivation for this project stems from various areas. The field of robotics is advancing, along with field of Artificial Intelligence (AI). Combining these two, we can start looking towards semi-autonomous and autonomous behaviour in computers and, specifically, robots. The NAO robots are the model for use in the RoboCup Challenge, which requires autonomous action pitting teams of robots against each other to play soccer. The RoboCup engages all areas of autonomous learning, from computer vision to robotic movement. This thesis specifically attempts to address the issue of the built-in trackers and provide more robust object and overall significance detection. Since the robot needs to be able to adapt and learn in a chaotic environment, it must detect signal from noise and form a plan of action for how to explore an environment and learn from it in a human-like way. Robust action and development in a chaotic environment provides myriad practical applications in the field of robotics and AI, pushing the AI continuum further ahead towards autonomous action.

Objectives

Primary Objectives

- Experiment briefly with the robot and framework to either port the remote accessibility to Python or determine if I continue using Java
- Develop a model for basic significance detection based on color or shape
- Devise and perform a simple experiment with reinforcement learning that punishes bad behavior and rewards good behavior using a significance detection model
- Devise and perform a simple experiment with adaptive learning that allows the robot to learn on the fly using a significance detection model

Secondary Objectives

- Create a more complex significance detection model based on various factors, not just color or shape
- Enhance the simple experiments and merge them to create a robot that uses both adaptive learning and reinforcement learning

- Combining visual significance detection with audio significance detection to create a more robust model that is capable of interacting with all environment
- Attempt to detect an item, or items, without using the built-in software (building a simple computer vision model)

NAO Robot

The NAO Robot was developed by Aldebaran Robotics in 2004, which was later acquired by Softbank Robotics in 2015. We are using the NAO V5, which is 574 mm tall, 275 mm wide, and 311 mm deep. It has two MT9M114 72.6DFOV cameras, four microphones, sonar rangefinder, two infrared emitters and receivers, nine tactile sensors, an inertial board, and eight pressure sensors.[1] It is used in education in the UK, has been tested for use in healthcare settings, and is used internationally for research purposes (much like this one).

Computer Vision

Computer vision (CV) is an interdisciplinary field that attempts to gain understanding from images and/or videos. Computer vision does so by making use of physics, statistics, learning, geometry, and more to extract information from an image (or series of images in a video?) and then use that information to gain a better understanding of the context. Some subdomains of computer vision are event detection, object detection, pose estimation, video tracking, image restoration, etc. This project will be focusing on object detection and video tracking along with learning. Computer vision specifically in robotics tends to lead towards autonomous movement through an environment. In this project, to have the robot learn and adapt its actions to its environment, it must be able to detect an object, but also the significance of that object ("is it the shape I am looking for?").

Techniques

There are many modern techniques in CV that make use of the color in an image. These techniques use the HSV color range and set a threshold for the color of the object they are searching for. Then, they can look at the HSV of every pixel and train neural networks with the inputs, having certain systems either learn through supervision ("this is the color you are looking for"), or learn by themselves ("I do not know what color, but I will attempt to recognize a pattern"). Often, CV uses Convolutional Neural Nets (CNNs) to broach this learning, as it is based on the visual cortex (specifically the connectivity of the neurons in

the net). Individual "cortical" neurons respond to stimuli only in a restricted region, which then overlaps with different receptive fields of different neurons, forming the full visual field. A major importance in CNNs (and why they are used in CV) is that they require less pre-processing than other image classification techniques, learning filters that would be hand-engineered in a different neural net. Learning on its own gives it independence from prior knowledge, marking an important step towards our autonomous action. At its basis, CNNs use an input and output layer as well as various hidden layers. The hidden layers are where "convolutional" comes from, as they convolve with multiplication or dot product. These hidden layers use common activation functions (usually ReLU), and the final convolutional layer also typically uses backpropagation to go back through the net and adjust the weights. The only problem with using a Deep CNN in this project's case is that it becomes computationally intensive, which in turn can be too slow to work real-time on a robot. I will attempt to do so using a remote module and NAOQi API to connect and feed the outputs back to the robot after training the model on a remote machine. While CNNs are powerful, they can use help from other algorithms and processes to aid in its effectiveness and speed.

Since the HSV values of every R, G, and B pixel of a medium resolution image can be A LOT of inputs, methods are used to reduce the search space and dimensionality of certain CV problems (usually classification). One such method is Principal Component Analysis (PCA). PCA works by extracting the features that have the largest variance. This stems from the fact that the dimension with the largest variance also contains the largest entropy, thereby encoding the most information. The smaller values will generally encode noise while the larger features correspond to principal components that define the data. Ultimately, PCA is used to find features that are statistically independent or not correlated to then reduce the dimensionality and overall size of the problem. This becomes particularly important in CV where the input space is so big and in our project, where time is of the essence. PCA has its own pitfalls and limitations, such as potentially losing out on information from less variance features that could have encoded particularly important data for whatever classification problems. In this case, methods such as Linear Discriminant Analysis (LDA) can be used, which attempts to find a projection vector that that characterizes or separates two classes.

Further processing methods include using Support Vector Machines (SVM) to segment the images, producing smaller images where there is only background (a false example) and images where just the object and some background is found, etc. Image segmentation helps with reducing the input space and allowing or faster video tracking without getting caught up by fast motion in real-time uses. SVMs are used in learning to assign new data to previously defined classes in a supervised example, or attempt to find patterns of classes

in an unsupervised example. In an unsupervised example, SVMs essentially cluster data by finding the hyperplane that represents the largest separation between the classes.

CNNs using PCA or LDA are a great way to detect patterns and objects, but there are various other methods we can do to help pre-process or process our images that may not have to do with specifically color. What happens if we need to detect an object that is the same color as its background? We can use grayscaled values, but then we must look or shape, and what if the shape or object you are looking for is partially occluded by the context?

The Hough Transform is another feature extraction technique that attempts to find imperfect instances of objects within certain classes of shapes by voting. The classical Hough Transform was designed to find edges or lines in an image, but I will be using a Circular Hough Transform (CHT), a slight modification of the original designed by Duda and Hart.[2] The idea of the CHT is to use an edge detector to find the edges of a shape and build the shape out from its edges. However, shapes can be noisy or occluded by other things in the image or context. The idea of the CHT is to group the edge points into candidates for an object by explicit voting over a set of image objects. To do this, a general HT algorithm uses a 2d array called an accumulator. The accumulator determines if there is enough evidence of a line (in the general form, in a CHT it would see if there was enough evidence of a curve) and calculate the parameter of the line, look for the bin in the accumulator that matches those parameters, then increment that bin. By finding the local maxima of these bins (the bins with the highest values), we will find the most likely lines. In the case of the CHT, this algorithm is simply extended to the geometry and parameters of a circle, so the pixels are then voted on for having a curve that could be the edge of a circle, and the votes are accumulated in the accumulator matrix. Knowing the approximate radius of the circles or ellipse in question helps a lot as then the radii can be used as like spokes on a wheel to pass through all the edges (found using Canny detection) and form the circle. If the radius is unknown, a similar algorithm is used, but then one iterates through all the possible radii values and votes according to various different radii. Then the same process is done to find the local maxima of all the possible circles, with the most probable circles being the ones voted on the most in the accumulator.

All these techniques will come together to help make the NAO visual system more robust, allowing me to progress to my ultimate experiment using CV and reinforcement learning. I will use OpenCV, an open-source library for Computer Vision to perform the CHT and change the colorspaces to grayscale as well as HSV for detecting the red values. OpenCV is an incredibly helpful library that performs Computer Vision algorithms (like the CHT), while the only thing I must adjust is the input and parameters depending on my problem space.[3]

Reinforcement Learning

Reinforcement Learning is a subset of Machine Learning based on B.F. Skinner's Operant Conditioning, a learning process which utilizes rewards and punishment to modify behavior.[4] Reinforcement learning differs from supervised learning, where a specific output is desired and the neural net tries to converge on that output. With reinforcement learning, the neural net tries to achieve a *goal*, maximizing a particular dimension or ability to perform a task. A great example is AlphaGo, a model that plays Go against itself and learns over time. It has now beaten several masters of Go. Reinforcement learning allows machines to learn much as humans do, which is slower than a supervised model, but can provide robust models that can effectively learn and use the tools at their disposal to learn how animals do. Learning in this manner is implemented through agents performing actions and updating their state in an environment where all the ins and outs are not known. To quote Sutton and Barto,

> "The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions." [5]

Utilizing reinforcement learning in this way is known as a Markov decision process, as shown in the figure below.

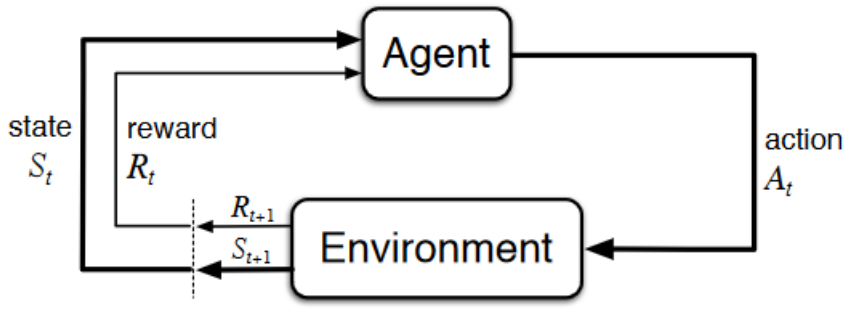


Figure 1: Agent-environment interaction in a Markov decision process[5]

Here it is easier to see the agent taking in its current state and reward and producing an action, which the environment then takes as a function and outputs the next state along with a reward. Using this finite Markov Decision Process and the underlying math, we can compute state-transition probabilities, expected rewards from state-action pairs, or expected

rewards for state-action-next-state triples. Now that I have explained a bit about how reinforcement learning works holistically, one can see how it transitions to being used in robots for generalized learning tasks.

Techniques

There are two algorithms I looked at to perform RL: Q-Learning and Deterministic Policy Gradients.

The idea of both of these is to learn in a model-free environment by learning a policy that tells an agent what actions to take under a specific environment and state.

Q-learning attempts to find an optimal policy that provides the greatest total reward over all concurrent steps (starting from the current state).[6] Q-learning works by calculating the "Quality" (Q) of any state-action combination. Q is initialized (generally to 0, but can be to any arbitrary number) and updated every time-step t the agent performs an action, gets a reward, or enters a new state. This iterative value updating creates a Q-table of the best possible state-action pairs. The reward is also discounted by a factor γ at each time-step the agent does not reach its goal, incentivizing the agent to complete the task as soon as possible. The discount factor can be modified depending on how much you want the agent to explore or stay on task (exploit). The discounted reward (state-value) equation is known as the Bellman equation. The Bellman equation is incredibly important in RL and is found ubiquitously in the field. To understand why the Bellman equation is important, I will go over some of the aforementioned basis as well as explain why achieving Bellman level of control is important in RL. So, you are given a discounted reward based on a state s, and the equation tells the expected reward continuing from a state s following a policy π. This expectation can be found by summing over all possible states and actions. By distributing the expectation between these two, we can get to the Bellman equations. This equation is important because they let us express the value of a state as a value of another state. So, knowing the value of s at time t+1 lets us calculate the value of s at time t. Bellman control opens up the door to iterative approaches to calculating value for each state, since if we know the value of the next state, we know the value of the current one. The general equation for updating the Q is pictured and labeled in Figure 2, using the Bellman state-value and action-value equations..

As you can see from this formula, the future maximum action value is calculated at the same time the current Q is. This simultaneous calculation can slow down learning as it sometimes overestimates action values. Double Q-learning is an algorithm designed to help fix that and is used in DeepMind. Double DQN (Double Deep Q-Learning) is an off-policy algorithm, meaning it selects the next action using a different policy than the value

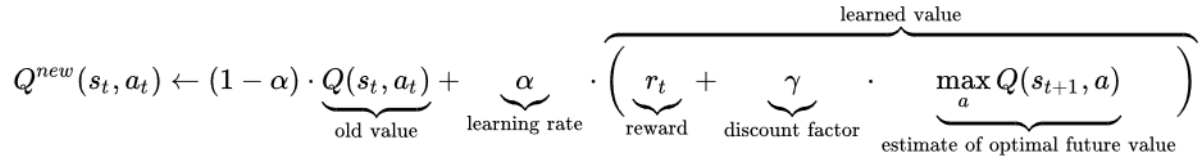


Figure 2: Formula for Updating Q table

evaluation. Here one can see where the "Double" comes in, as it requires two separate value functions trained symmetrically using different policies. Both these Q-values are compared, and this way, the estimated value of the future is chosen by a separate policy, reducing issues of overestimation.[7]

Q-learning is a relatively simple to understand algorithm that works in discrete environments. However, what happens when we want to learn in a continuous environment? With a stochastic process like DQN, this process becomes difficult. This quandary is what the Deep Deterministic Policy Gradient (DDPG) algorithm looks to solve.[8] Through research, I realized that it would be much more interesting to learn from raw pixels than a hard-coded, discretized environment. A deterministic policy gradient can be estimated much more efficiently than a stochastic policy gradient, as it is the expected gradient of the action-value function. There is one main difference in how these policy gradient algorithms work:

> In the stochastic case, the policy gradient integrates over both state and action spaces, whereas in the deterministic case it only integrates over the state space. As a result, computing the stochastic policy gradient may require more samples,especially if the action space has many dimensions[8]

A stochastic policy is still usually necessary to be able to fully explore the state and action spaces. Therefore, DDPG (Deep Deterministic Policy Gradient) uses an off-policy algorithm to ensure exploration. The idea is for a stochastic behaviour policy to choose actions, while learning about a deterministic target policy (thereby utilizing the efficiency of the deterministic policy gradient). To accomplish this task, they use an off-policy actor-critic algorithm that estimates the action-value function, then updates the policy in the direction of the action-value gradient. To understand this algorithm better, we must delve into what "actor-critic" means.

Actor-critic is an architecture based on the policy gradient theorem. The actor adjusts the parameters of the stochastic policy using gradient ascent. Instead of an unknown true action-value function, a function approximator is used. The critic then estimates the action-value function using a policy evaluation algorithm, such as Monte Carlo evaluation or temporal-difference learning. The DDPG takes it a step further and estimates the policy gradient off-

policy, like in Double DQN. Normally, policy improvement is done through greedy methods (greedy maximization of the action-value function); however, these methods are problematic in continuous action spaces as they require global maximization at each step. This global optimization is too slow for practical use with large, unconstrained function approximators and nontrivial action spaces. In my experiment, the action space is quite trivial, but the environment still takes continuous input, meaning the state space and action-state space become rather large. Ergo, the actor-critic model is used, where a parameterized actor function specifies the current policy by deterministically mapping states to a specific action. The critic is learned using the Bellman Equation (in the algorithm I utilize), just like in Q-Learning. The actor is updated by applying the chain rule to the expected return from the start distribution J with respect to the actor parameters. Silver et al. proved this as the *policy gradient*, the gradient of policy performance, as shown in Figure 3:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t} \right]\end{aligned}$$

Figure 3: Policy Gradient formula[8][9]

I will utilize Keras' RL libraries to implement the DDPG algorithm, which follows the process in Figure 4.

This algorithm uses a replay buffer to sample experience and update neural net parameters. These minibatches from the replay buffer are sampled to update the policy and value networks. The actor (policy) and critic (value) are updated as in Q-learning using the Bellman equation. The difference in DDPG is that the next state Q values are calculated with the target value and policy networks. Then, the Mean Squared Loss is minimized between the updated and original Q value. A copy of the target network parameters is used to "soft update" the parameters by slowly tracking those of the learned network. We can use these old experiences from an outdated policy because the Bellman Equation does not care which transition tuples are used, as the optimal Q-function will satisfy the Bellman Equation for *all* transitions. That is why the replay buffer helps and does not hinder the learning. One difference seen in discretized environments (Q-learning) versus continuous environments is the idea of "exploration." Exploration in Q-learning involves probabilistically choosing a random action (usually by greedy-epsilon). Exploration in continuous environments is done by adding noise to the action itself. In DDPG, they use the Ornstein-Uhlenbeck Process.[10] The process generates noise that is correlated to the previous noise, preventing "freezing" or cancelling out the overall dynamics. The main difference between DDPG and DQN, is that DQN only works for discrete actions, and DDPG only for continuous action spaces.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 4: Pseudocode for DDPG Algorithm[9]

This main change boils down to how the max over actions (max Q*(s, a)) is calculated. In a discrete action space, the max can be computed for each discrete action and compared. However, for a continuous action space, this computation becomes much less trivial as one cannot exhaustively evaluate an infinite space (it becomes painfully expensive). This calculation would have to be run every single time an action is taken. DDPG solves this by presuming that a continuous action space is differentiable with respect to the action space. This presumption allows for a gradient-based learning rule for a policy $\mu(s)$. So, instead of running the expensive subroutine and calculate max Q(s,a), we can approximate it using Q(s, $\mu(s)$).

Challenge

The combination of Computer Vision and Reinforcement Learning provides a unique challenge of having real-time analysis of the environment through cameras while attempting to also learn from the analysis and deeper "sight" gained by the CV models. Specifically, having a robot be able to engage with its environment and identify objects poses a challenge both of having complex sight and identification processes, but also of being able to use the information to produce more rewards in a reinforcement learning model. Robots have the ability to move around on their own and explore the environment, which creates another difference to regular "brain in a jar" learning models, where a stationary computer is fed information for a learning model, be it supervised or unsupervised. Having a robot be able to move their brain around and actually change their environment to stimulate further rewards is incredibly potent and novel. The robot must look not to converge on specific outputs, but to be adaptive and diverge in its thought process, allowing for randomness and chaos to not bring it down, but rather enhance its learning. The robot can achieve this chaos or noise through the Orsntein-Uhlenbeck process in the DDPG, discussed earlier. However, the robot should still attempt to converge on states and actions that provide similar benefit. The main challenge in this learning is to achieve not only short-term learning but legitimate long-term learning, where colors are approached due to intensity at each step, but the episodic reward is negative if the color approached is not red.

Chapter 2: Background

This project deals mainly with computer vision, machine learning, and robotics (specifically with the NAO Robot). Therefore, the context survey looks at the papers that students have done before from which I will step off from as well as other research in the field of computer

vision with robots, NAO or otherwise.

Previous Work

There are three dissertations I use for this project that I am building on or taking inspiration from: Lamb and Weir (2016), Li and Weir (2017), and Finlay and Weir (2019).

Lamb and Weir devised a semi-autonomous genetic algorithm that managed to simulate juggling in 333 and 531 patterns, while adapting on-the-go.[11] I will pursue a different form of semi-autonomous adaptive learning in my third primary objective.

Li and Weir looked into scripted and unscripted actions for the NAO, researching basic programming and obstacle navigation.[12] I will use this paper to help in programming the NAO.

Similarly, I will mainly use the NAO manual Finlay used to adapt NAOqi in Java if I use Java.[13] From this thesis, I hope to gain the ability to use more computational power, such as Tensorflow models.

Related Work

Reinforcement Learning

First and foremost, my project deals with learning, specifically reinforcement learning as that is usually the method for learning in robotics. There are various methods and techniques for performing reinforcement learning in robots, and I found some generalized models as well as more specific. Most researchers agree that model-based approaches, such as decision trees and behavior rules are key to developing solid algorithms that work on low-quantity real-world samples.[14][15] There is also work done on using Hidden Markov Models to "teach" robots.[16] One project shows reinforcement learning in the NAO with their own "Decision Based Rule" method to perform gesture imitation.; however, this method uses Microsoft Kinect computer vision features, and I would attempt to recreate this on my own.[14] Another project shows a more generalized reinforcement learning with the NAO, using decision trees to learn a model quickly with a targeted exploration policy, showing improvement in kicking a ball in a penalty kick scenario, which I can look to incorporate for my second experiment.[15] I also looked at different forms of behavior-based models in robotic learning, specifically through emotions as well.[17]

Since I also need to learn as quickly as possible to perform real-time actions, I looked at shaping, a method for speeding up reinforcement learning. Autonomous shaping can help the robot learn tasks that are similar but distinct much more quickly, which could help reduce

labor of manually shaping as I progress through my experiments.[18]

Computer Vision

Much research has been done in object detection and significance detection in computer vision, specifically with robotics. While I looked at computer vision papers and algorithms, I also narrowed my search towards robotics (hopefully with the NAO), as they must work with less computational power or with less time (real-time).

Some main computer vision methods for significance detection and object location are using Deep Convolutional Neural Networks to segment images into smaller pieces to reduce the search space and validate objects.[19][20] These networks also make use of Principal Component Analysis and feature extraction with Support Vector Machines to segment the images and detect objects.[21][22]

Much of the important work done on significance detection with the NAO is done for the RoboCup 2050 project, so many projects will be referencing those conference proceedings.

There are multiple ongoing projects to detect balls with the NAO robots. Many of them use Deep Convolution Neural Nets (CNN) to perform semantic segmentation and object detection.[23] Some of them use feature extraction as well as Support Vector Machines (SVM) specifically for image segmentation.[14] I took a look at some feature extraction in chaotic environments and information-theoretic learning as well to understand better how to create a model that would perform the feature extraction portion of my object detection.[24][25] An interesting project performed by Albani et. al is similar to my first primary objective in that they perform object detection using a NAO robot, but they train the robot using the publicly available library for NAO robots to detect between robot or not.[23] However, they tested different methods such as contrast-based gradients and adapting exposure due to robot interest, which is usable for the first primary objective. They also found that deep learning technqiues are computationally expensive and need to use the publicly available dataset. For my purposes, I cannot train on this dataset as it is from a specific soccer arena made for the RoboCup 2050. Furthermore, I attempt to do away with the computational expense by utilizing Finlay's thesis project connecting the NAO to a more powerful computer with a GPU to run Tensorflow.

Keeping the object detection robust in lighting and hue changes is the problem for the second primary objective, and I found some projects attempting to maintain a robust object detection platform. Most use Deep CNNs focusing usually on HSV (Hue, Saturation, Value) and contrast-based gradients. Hartl et al. attempted to solve the static color classification problem I am performing in my second primary objective, and managed to improve accuracy under changing illumination, which will be a focus of my second experiment.[26] This paper

goes into depth on how to attempt segmenting and pre-processing I will need to do to detect the ball in my first and second experiments, while maintaining robustness in a chaotic environment. I also looked at other experiments in computer vision to reduce noise and improve robustness as I will be demoing the project in a different presentation room.[27][28] Specifically, I looked at robotic fruit detection algorithms (such as Whittaker's)[29] that could pick out fruits by shape, even if the fruit was partially hidden or occluded. These algorithms are important and serve as a basis for my first experiment in detecting not only the color of the ball but the shape and size to establish "significance" from noise. Detecting the shape is based on a circular adaptation to the Hough Transform devised by Duda and Hart designed to detect lines and curves in images.[2] Delving deeper into the CHT (Circular Hough Transform), there is work done by Sengupta and Lee that utilizes this CHT alongside texture classification using SVMs and image segmentation to detect as many circular green citrus fruits in a green canopy as possible and remove false positives before continuing to process using a voting system.[30] Adapting these algorithms is important because they demonstrate robustness to lighting conditions as well as color-invariant object detection, which forms a part of all of my experiments. Sengupta and Lee also used feature extraction, specifically to make the recognition scale-invariant (invariant to rotation, scaling, transformation) by utilizing Lowe's algorithms for scale-invariant feature transformation (SIFT).[31][32] Using SIFT in the vision pipeline would decrease my false positives and is helpful when seeing out of multiple cameras oriented at different angles as in the NAO. The trick is to be able to combine these algorithms in real-time functionality. Moreover, I looked at an algorithm modified from the CamShift algorithm to track an object as I will have to do so in the second experiment. The juxtaposition of the fruit identification algorithms and the CamShift tracking to maintain view of the ball under changing conditions will be used in the second experiment.[33]

A further approach on object detection with the NAO was done by Lüders using contrast-based object localization, which will form a basis for a method I will attempt on my first and second experiments.[34] Lüders' paper also ran into the issue that a deep CNN with SVM is very computationally expensive and could not get it to run fast enough to be useful. He also trained using the public dataset, which I will most likely not be doing. However, global contrast-based region detection proves to be a good way to detect objects and could be used in my first and second experiments to compare results.[35]

All of these projects still use the provided public RoboCup Challenge datasets, which I may or may not be able to use. Furthermore, all projects with the NAO run into the issue of computational power and time. Some projects try to fix this by batch processing asynchronously across NAO robots, but I cannot do this as it will be an embodied learning concept in one individual NAO robot.[36][37] There is another project that looks at the multi-

robot domain, but it also goes into detail on behaviors and conditions and how they can make a model more robust in chaotic environments, therefore it is a bit more helpful than the batch object detection.[38] Due to the issue of computational power, I looked at some papers for higher speed and less computationally intensive computer vision solutions.[39][40]

Adaptive Learning

My final experiment has to do with adaptive learning, so I wanted to see what projects besides the juggling project done by Lamb there were in the NAO field, as well as in robotics in general. Ergo, I found some methods for producing genetic algorithms and for optimization in adaptive learning.[41] There has also been work specifically in reinforcement learning with NAO but in movement, which is helpful at least to see methods for adaptive learning in stochastic environments.[42]

While not with a NAO robot, a small paper about Embodied Learning shows an adaptive learning approach to embodied learning, which is essentially the theme of my ultimate experiment.[43] It is a simple version and does not go into too much depth, but still helps in designing an adaptive learning methodology for the last experiment.

Chapter 3: Red Ball Detection

Design

The NAO comes with a built in module called ALRedBallTracker that attempts to find red pixels and then make sure they are circle-shaped; however, the tracking system is poor and does not correctly identify between a red ball and a red chair/object in general. So, this experiment is an attempt to make the tracking system more robust and be able to spot between the red balls and chairs. This demo will pertain a comparison between the official ALRedBallTracker module and the new red ball tracker, which should track a red ball more effectively.

Implementation

To test the system and learn how to build the software, I first attempted to build a module that would detect the red ball through an image rather than video, then adapt the image recognition into video tracking by having it perform the operation on continuous frames and keep the ball in the center of the robot's "view" (camera).

I used the getandsaveanimage.py example on the Aldebaran website to start off and added OpenCV. I created HSV versions of the images using OpenCV and used a separate open source program, ColorPicker.py, to find the threshold HSV values for the red balls.

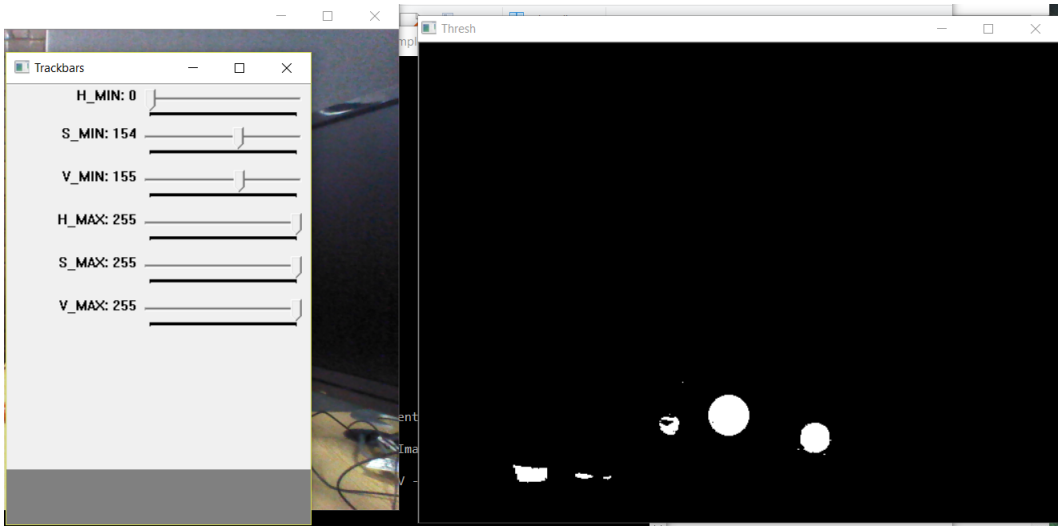


Figure 5: ColorPicker.py sliders to judge HSV thresholds

From there, I created two masks to find just the red pixels (one for the lower range of HSV red and one for the upper), then performed morphology (erosion and dilation) on the red-pixel mask to remove blots and fill in any holes in the ball.

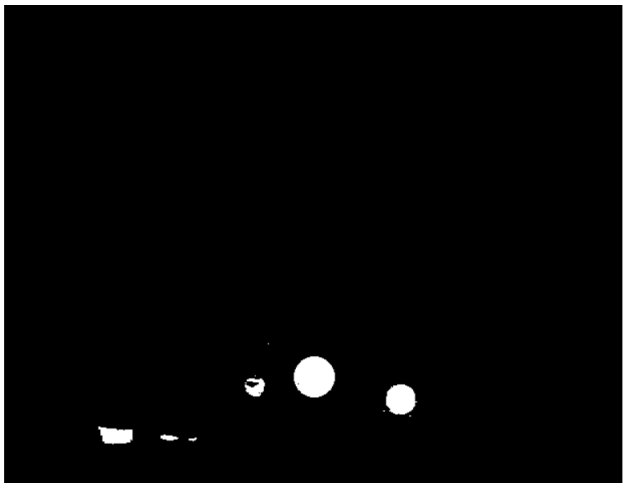


Figure 6: Image with HSV mask applied (flipped by ColorPicker)

The masks are then joined together, and the image is blurred using a Gaussian blur to remove some noise.

Finally, The CHT is performed on the image to find circles and the circles are drawn on an image and printed out. It took me some time to get the CHT to finally identify the

ball as the pre-processing and parameters of the function needed to be tuned to accurately detect the ball.

Figure 7: First successful detection

We can see from Figure 7 that we still do not manage to find the two balls on the sides in this example. Sometimes, I could find them using different HSV thresholds and CHT parameters, but it also meant that there was more noise and false positives from the chair. I suspect this generalization may be why the built-in red ball tracker is so poor.

I then attempted another method of detecting circles than CHT, which is findContours from OpenCV. findContours works in a similar way to the CHT, but is a more generalized function and finds the minimum enclosing circle after finding the center of a set of contours.

I also wrote images to file of each process as I initially tested to benchmark (and show above), but remove the writes in the video version as they reduce performance.

Results

The static detection of a ball turned out to be much easier than expected, as OpenCV has tackled and made this issue obsolete. Mostly, it involved tuning of parameters for the formulas as well as understanding the methods used intimately to be able to tune the parameters to necessary. The most time was spent attempting to find the correct threshold for the red HSV mask, and I am still not sure HSV is the best color space to use as converting to RGB from the NAO's native color space (YUV422) already takes processing power and

then converting to HSV takes a bit more. Overall, the changes in computing time are minimal and detection takes less time than the acquisition of the image. I was pleased by this experiment as it turned out to be easier to detect certain shapes wihtout learning than I previously anticipated. For the future, I would want to teach the robot to detect multiple different objects, allowing for actual learning to occur in this step of the process.

Chapter 4: Red Ball Detection and Kicking

Design

In this experiment, the idea was to track a red ball versus another colored ball and be able to track the red ball in video form. The robot scans an area and attempts to find a red ball and go towards it. I found research performed by students at Boston University (David Lavy and Travis Marshall) that tracked a red ball and compared their approach to mine. I used their code for moving the robot, as it is the same lines as I would have used. Unfortunately, I did not find their repository until I was already attempting to fix problems with my robot code for this experiment and found their solutions, resulting in some duplication of efforts.

Implementation

In this section, I will discuss the process the robot takes to find and approach the ball along with some comments on the differences in approaches and pitfalls for future students that may want to recreate this experiment with different detection methods or algorithms.

First, the robot takes in an angle in radians and takes 5 snapshots of the environment by rotating its head between the max Angle and the negative max Angle. Once it takes these pictures, it performs the ball detection algorithm from Experiment 1. This action is the only difference between my experiment and theirs. I utilize the same function I found and used for Experiment 1, the Circular Hough Transform. They used a more generalized version OpenCV's FindContours, which captures the moments from an image or polygon. From there, they performed a Minimum Enclosing Circle around the contours of the red objects to find circular patterns, then found the center of those, saving them in the format [y,x]. The fact that it was saved as [y,x] actually took some time to understand as some calculations were not working right off the bat and it is a little counter-intuitive so if you are attempting to use your own detection formula to recreate this, make sure the centers are passed back as [y,x] rather than [x,y]

Their detection formula actually seems to outperform the CHT in accuracy (95% per their report), yet the CHT can be tuned further and returns multiple circles. They also had

Figure 8: Example of Minimum Enclosing Circle on a found contour[44]

to do a good amount of error correction to improve their accuracy and better distinguish ball contours from non-ball contours, then do error correction on the MEC (Minimum Enclosing Circle). For the purposes of Experiment 2, findContours works better; however, for transitioning to Experiment 3, where I need to be able to detect multiple different objects of the same color, the CHT is an easier choice to implement without having to do some serious error correction. They also utilize error correction on their Minimum Enclosing Circle function to make sure they get the right ball.

In the CHT, all the "error correction" is done by the parameters in the CHT, making it much easier to tweak and get up and running quickly. With the CHT, I can also apply a much wider HSV mask than red and still only get the red ball, as I am already bounding by circular shapes as opposed to a minimum enclosing circle around an object. The strictness of the CHT does create some problems as very circular shapes sometimes are not quite circular enough for the algorithm, leading to greater changes in the mask to allow for more pixels in the circle. This reason is also why I apply a moderately aggressive Gaussian blur to try to smooth out edges more as that is where pixels go missing due to changing luminosity.

If the robot finds no balls, it says it could not find a ball and turns around to scan elsewhere. If the robot finds a ball in one of the 5 pictures, it will say "I think I found the ball" and "I need to get closer". If the robot finds the ball in at least two pictures, it says "I found the ball," followed by centering its head on the angle of the median image with a ball detected (so if the ball was found in the second, third, and fourth images the head would center to 0, the angle the third image was taken at).

Then, the robot takes 3 snapshots at a more shallow angle to ensure it has found the ball after moving a bit closer. These pictures also allow for centering the robot's movement

on the ball's center. This calculation is done by taking the change in x position of the ball over the first two pictures it detects a ball and using the angles the pictures were taken at to center the robot on the ball's position.

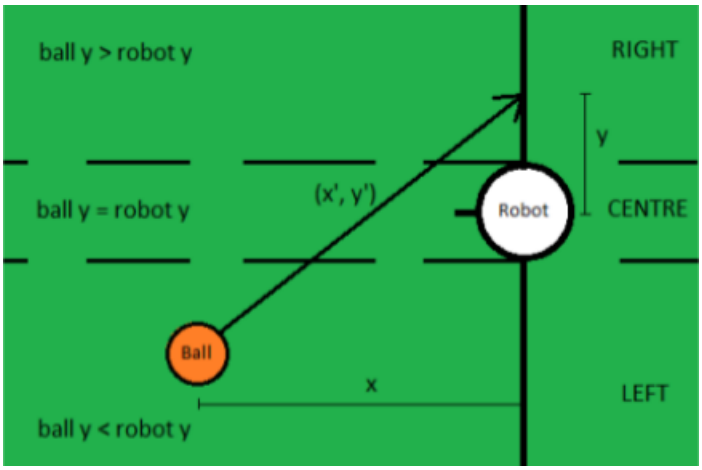


Figure 9: Visualization of Robot Centering[44]

For my program, I had to change which picture to look at for centering as the CHT would generally not detect the ball in one picture as it was obfuscated, blurry, or too luminous to where "red" appeared white in the image and would not transfer to the HSV mask. Lavy and Marshall use the first detected ball picture and correct angles from there, but I prefer to use the second detected picture as I found it provided a better median angle to turn to and kept the robot more on point with how my detection worked. Since they use findContours, if they are missing an arc segment of the ball due to light, it is made up by the MEC, actually allowing for a slightly more robust detection algorithm. Again, this is at the cost of time and energy due to manual error correction.

The robot then moves forward, checking as it goes that it is keeping the ball in the center. Once the ball is out of view of the upper camera, the bottom camera comes into use and pictures are taken more frequently to ascertain whether the ball is in kickable range. When the robot gets within 15 cm of the ball, it starts moving very precisely to match up with the ball. Sometimes, the ball rolls a bit which causes the robot to accidentally nudge it forward, making it miss the ball.

The robot then attempts to kick the ball, which appears incredibly shaky and one should keep their hands around the robot to make sure it does not fall and hit its head, especially since the surface one uses may change how the walking up works. The kick is generalized from previous NAO documentation on kicking and can definitely be improved. That is not an aspect that is important to my project, so I did not spend time doing so.

I found that the students used a tile flooring and I used a carpet flooring and there was

no issue with reaching the ball, but that is because there is correction once it is in the lower camera loop.

Results

The robot's task was to detect, approach, and kick a red ball. It performed exceptionally, except the robot sometimes took very long to take some snapshots and very quick for others. For the future, one could also implement some waiting if not in real-time to allow for the camera to steady, as shaky pictures lead to errors in detection. The idea is that detection should be in real-time, so shakiness is ok in this experiment as a form of "robustness" control. I was surprised that the findContours method worked better than a Circular Hough Transform, but mostly because I was not expecting the CHT to be so strict on circular shapes versus slightly elliptical ones. The research I found on the CHT explicitly lauded its ability to work with obfuscations and deviations, which I did not experience in this experiment. I still believe there are clear use cases for each and am happy to have learned about both. For findContours, a lot more is needed to get it detecting ball contours specifically and then finding the center of the ball contours. The CHT also provides you with a radius value, which is extra information as well as being able to identify multiple different circles in an image. CHT also takes much less time to get setup and running/detecting balls. findContours requires more work in the intermediate and general math setup along with error correction, which I did not employ.

Chapter 5: Autonomous Navigation & Learning

For the final experiment, the idea is to use RL to have the robot navigate to a "final" point. Since RL agents take time to train, I decided to use Webots to simulate the NAO robot. The main problem with Webots is that the new software does not support the NAO inherently. Therefore, I had to use a third-party library called naoqisim, developed by Olivier Michel.[45] The library took longer than anticipated to make work as my Windows environment was a bit of a mess and the "make" action would not work, even after downloading MSYS2 to make it easier. In the end, the simulator requires changing the environment variable of the Webots Home path running a Makefile to compile the library from the root directory of naoqisim. It is highly suggested to use MSYS2 or a conda env to make this library. The simulation allows you to connect to the robot with a default IP of 127.0.0.1, allowing me to use the same exact codes I used for Experiment 1 and 2, just changing the Robot IP (also allowing the same code to be tested on the physical robot later).

Design & Implementation V1

The naoqisim SDK gives a Nao controller in an empty, lit room. I added four walls around it. The wall to the robot's left is completely red and the wall in front is red to the left side of it. The rest of the walls are white. We can see in Figure 10 the top and bottom cameras and what the robot sees while performing its actions. The cameras are set to a 80x60 resolution for faster learning, but can be made even smaller as well as larger with a better model.

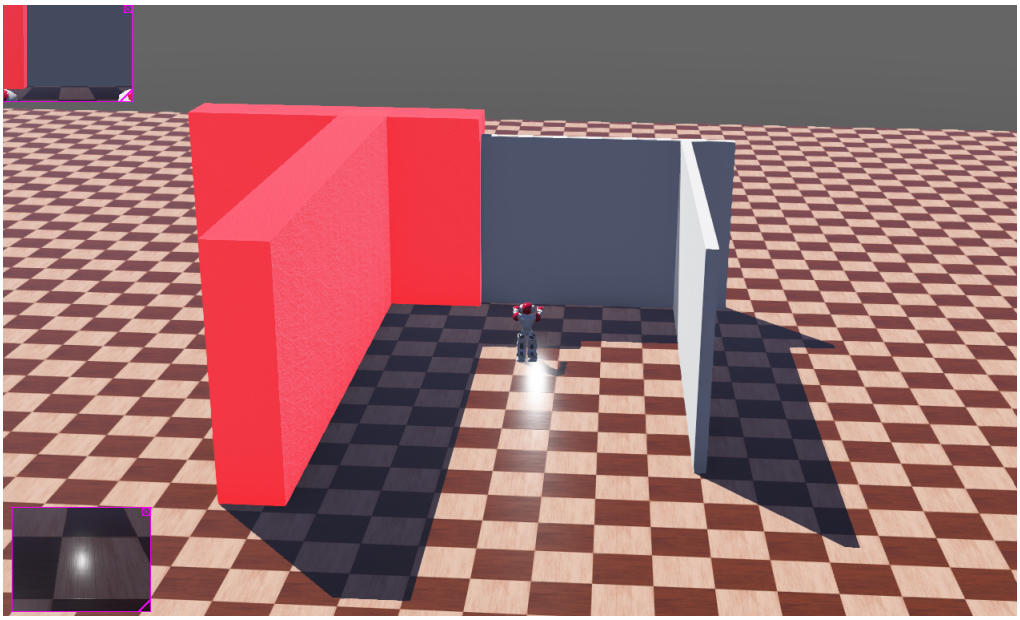


Figure 10: Simulation in Webots 2019 using naoqisim

The idea behind this setup is to have red in the direction the robot needs to go. The RL process finishes when the robot reaches a certain total red pixel value while staying under a certain blue and green pixel threshold. I calculated these using my Experiment 1 code to take pictures and analyze the pixels. I found that summing up all the red, blue, and green pixels respectively allow me to have an incredibly simple model for "redness," tested along with the new simluator world. I used RGB over HSV to begin as I am not incredibly familiar with Webots and wanted to make sure the luminosity of the light would not negatively impact the experiment. I found that the red pixels from the start Each step gives a reward = -1 + redPixelDifference/x - bluePixelDifference/x - greenPixelDifference/x, where x is an arbitrary factor (usually 4800, the amount of pixels processed). Discounting each step encourages exploration and doing tasks rather than inaction. The idea is that the more red pixels, the closer to completing the task, while also penalizing adding more blue or green (or rewarding reduction of blue/green). This method has a slight double effect as adding more red by nature reduces the blue and green. This effect is to penalize turning

right towards more blue or green and away from red. The discrete numbers can be changed and arranged over time until the model learns quickly and well.

Building and Using a Custom Gym Environment

To make this RL problem work, I wanted to use Keras' RL library to run a DDPG for my own custom Gym environment. I based my Gym environment (which I had to install as a package into gym as well) on a simple Car Racing game, that learns from raw pixel values of 80x80. Adapting this environment took the most work, as figuring out how to do the whole RL problem was in the environment. To make a custom Gym environment, you have to set up a specific package structure and have an environment file with 4 methods: reset, render, step, init. The init method intializes the environment, reset is the first step taken, then step takes over and iterates. Render is to visualize the learning process, but I do not implement anything with it as Webots will be the visualization. With these 4 methods written, you can use the Abstract Gym environment class to make a custom environment with its own rewards. The car racing environment took a while to adapt as it was much more complex and produced reward through the Car and FrictionDetection classes. I also looked at the basic pendulum swinging and a tic-tac-toe problem for more basic needs, but they were a bit too basic. The DDPG code I took from Keras' pendulum example and was using the model they used as a black box until I could get the environment up and running and then go tuning parameters, depth levels, amount of nodes, etc.

In the init, I set all variables to 0 and create the action and observation spaces, making use of the Box space provided by Gym. the action space is a 2D array containing the bounds of the turn and move radius, which I change many times depending on the experiment. The observation space is a 3D Box array (bounded 0-255) of 80x60x3 for the resolution and one for each channel of RGB. In reset, I take the first picture, save it, calculate the red, green, and blue pixel totals, and wake the robot up. The step method is where most the implementation goes and where "the action happens," so to say. The first block of code in the step is an if, for if there is an action coming into the step. If so, the action (the output of the DDPG agent) would be scaled to fit the bounds of the action-space array and the environment's x and angle variables would be set. Then, the action is taken by the robot (I call a separate move function using the variables x and ang). After the action is complete, I take another picture and go into another if action is not None statement to calculate the step reward. I do this in two separate ways in different experiments: rewarding an increase in red values/penalizing an increase in blue or green, or rewarding pure total pixel intensity (RGB values for every pixel added up). These different reward structures depended on different Webots environments and can be commented in and out depending on the environment. Before the step reward

is calculated, however, the total reward is discounted by 1 (representing the discount factor, γ). Then, the step reward is calculated. The next if statement checks if the episode has completed (if the goal has been reached) and is changed depending on what I wanted for each experiment. In the difference reward structure, the threshold was a certain intensity of red. For the NOMAD reward structure, it was a certain intensity of RGB total. Inside this "done" check, the done flag is set and the total reward is updated with either a positive or negative amount depending on the outcome of the episode. Figure 11 shows an example step reward calculation and "done" check. The goal here is merely achieving a certain total intensity, whereupon the total reward is changed if the color approached was not red.

```
209     if action is not None:
210       #discount
211       self.reward -= 1.0
212       step_reward = self.reward - self.prev_reward + float(x/240000)
213       self.prev_reward = self.reward
214
215       if x > 500000:
216         if self.red > self.blue and self.red > self.green:
217           self.reward += 100.0
218         else:
219           self.reward -= 100.0
220           step_reward = -10
221
222         done = True
```

Figure 11: Example of Step Reward and Done Check

Potential Pitfalls

I ran into some serious setbacks when I finished writing the environment, as I could not run it along with the robot code. The NAO SDK for our version of the robots only works in 32-bit Python. Keras' backend uses Tensorflow, which only works with Python 3.5+. Thankfully, Microsoft has another backend one can easily replace in the .keras settings called CNTK. The problem with CNTK is that it can only work with 64-bit Python in Windows. At the end of the day, I had to re-build the complete environment on a separate Linux machine, as it was clear my Windows machine could not handle it. Thankfully, there is a 64-bit Python NAO SDK for Linux, allowing me to re-spin up the process in Linux (which is also better for Gym, as it is only experimental in Windows). However, this approach did not work as the boost version necessary for Ubuntu 16.04+ to work (boost 1.58) and the boost necessary for the NAO SDK (boost 1.55) are incompatible on the same machine and cannot be installed simultaneously. Ultimately, the only backend that I could make function

on 32 Bit Python in Windows was Theano, and I ended up with a fairly simple model that learned to approach a red wall. It is highly recommended to create a new 32-bit Python conda environment with the commands:

```
set CONDA_FORCE_32BIT=1
conda create -n py27_32 python=2.7
```

. The model can run on an amount of steps chosen by the user, after which the weights are saved into a h5py file, the sequential memory is saved using pickle, and the training and testing is logged in log files. The file then loads these back up when running or can be commented out to test new models without previous training.

To install the custom gym environment you must run the command:

```
pip install -e .
```

. from src/Experiment3/gymNaoEnv/

Results & Analysis

The first time I ran the training, I had to remove the redDifference from rewards as the final destination and the starting point shared similar redPixel values, which I did not know why. The blue and green pixels reduced from starting to ending point, so I still used the simple difference reward function, merely removing the red difference. While this initially worked, the robot would start turning too far and rewarding itself for spinning in circles at its max spin angle, effectively rewarding itself for decreasing blue/green, then penalizing for increasing blue/green, then looping over and over. I fixed this problem by adding to the IBSStrength in the color of the red Wall objects in Webots, which meant the final destination redPixel value summed up to ¿1,000,000 and started out at around 200,000, allowing me to add red difference back into the reward function.

The second time through, the robot managed to reach towards red wall and continued to walk into it, falling over and causing me to have to restart the training with less steps. This behavior was still erroneous as after getting close enough to the wall to achieve ¿1 million redPixelValue should have completed the episode and granted an extra reward of 1,000 and rested the robot. After training more the robot several times and adjusted the reward function slightly by increasing the reward for red by a factor of 1.5, the robot starting making more moves towards red closer to itself rather than further away. Another major problem was that the difference between two bad states (no red in them) could potentially provide a positive result if there is "less" blue or "green," so I wanted to try solely focusing on red as well. Solely focusing on the red causes other problems of having the robot spin

around forever, as the light will cause red difference between two states without red, but having the blue there should still penalize the robot.

Another issue I have not been able to figure out is why the robot always seems to turn to the negative angles. It rarely ever turns left, but almost always right, and I am not sure why.

Furthermore, my supervisor aided me in realizing that learning would be more meaningful in the long-term if the robot was rewarded by approaching any color, as long as it was the closest, then determining a total reward based on the magnitude of red, meaning the learning is happening after an episode of the whole task, versus after each step with the difference in redPixels.

Design & Implementation V2

My next experiment did not go in this direction yet, but rather changed the whole wall of red to just a circle of red on a wall of white, to better understand if it would pinpoint red or was behaving erroneously. At first, the robot just spun around in circles, as some of the walls were cast in shadow. I changed the color of the walls to cyan (0 for red, 255 for green&blue). However, the robot would still spin around in circles as the reward function subtracts the blueDiff, so if there is a reduction in blue/green pixels, the agent is rewarded. This caused a problem as the lighting in the simulation casted shadows on the walls, as it is directional. Even without casting shadows, the luminosity reaching the walls was different, causing certain turns between the same colored walls to produce a reward. Increasing the inner luminosity (IBS Strength) on Webots of the Wall objects that were cast in shadow from the lighting allowed for step rewards within a +-1 margin of error, letting the robot escape the spinning loop. I also made the floor completely black (setting luminosity to -100), as I did not want the color differences changing from light changes to the floor as well. Moreover, I changed the coefficients of the reward function to reward 2*redDifference and only 0.5*blue/greenDifference. This change also helped escape the spin cycle as it increased the impact of just red and reduced the impact of either penalizing more blue/green or rewarding the reduction. This Webots simulation world is shown in Figure 12.

Results & Analysis

This version of the experiment managed to go towards the red square and complete in about 65 steps from the first training, but the red square is right in front of it, and the first training the robot spins around to explore more, but once it starts moving towards the red square, spins around less and less. I still get the same problem that it continues to spin around

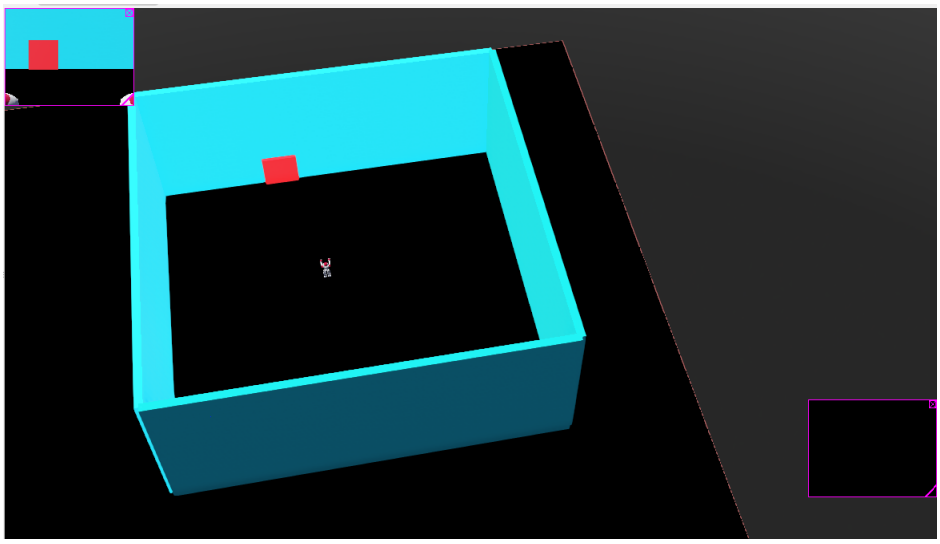


Figure 12: Updated Simulation World, Webots 2019

clock-wise rather than correct back to its left when it moves away from the red. I will examine if this behavior persists over multiple training sessions. In testing the agent, it will correct back towards the left when getting close to the red square. In further training, it would correct towards the left earlier with more and more training.

Trying out the training and testing without memory and weights led to more spin-arounds (see videos section for a training and testing without weights or memory). This shows that the training was actively improving the amount of time needed to complete the task. This version of the experiment still shows there are major problems with using the simple color difference reward function and there should be something more powerful and useful I can use to attract the robot to red and repel it from blue. For example, I also attempted to make the walls pure blue vs. cyan, but that ended up making the agent worse, as the green difference seemed rather arbitrary. I also get the issue where the agent tends to choose to turn while walking forward, as there is a lot of slippage on the simulation floor and going straight requires some turning. I believe this may affect the turning and also affects the rewards as normally going towards the red square would be seen as good, but there is a bad step reward associated with going forward and to the right enough to where there is actually a reduction in red and increase in blue/green because, while closer to the square, the robot ended up turning further away.

I think this is another major issue with the simplified difference function, as there is even a negative step reward associated with getting the red square centered, as the distortion from an angled view allows for "more" red pixels, meaning the robot prefers to come at the red square from an angle. Going straight at the red square obviously does increase the red

value sum, but the robot always tends to turn while moving straight (again, most likely due to the slippage and having learned to move straight that way). Sometimes, the slippage differs between training sessions, meaning the robot may choose different actions or states to correspond to the current state that do not produce optimal rewards.

I also tried a couple training sessions where I subtracted 10.0 from the step reward if the total red amount was under a certain threshold (meaning it was on the cyan walls), so even if it was turning between cyan walls and reducing blue/green, the step reward would still show a negative for not finding any red whatsoever. I also added a 15.0 increase if there was a certain amount of red in the view, as the agent could receive a negative reward for shifting the red a certain way, and I wanted to make sure the agent was always rewarded for putting red in view. This method added a separate logical error, where as the robot got closer to the walls, the red pixel threshold was still being surpassed, purely from the intensity of color from the wall (more cyan wall vs. black floor).

I would have loved to have a blank floor that I could change colors to test on, as this would have let me keep the floor and walls consistent, but all of the floor assets in Webots were patterned or completely white without a color appearance changer.

Design & Implementation V3

Another thing I noticed with the spinning was that the DDPG agent was always choosing the lowest bound for the actions necessary until testing. I realized that it may be going through its policy before going onto the next attempts and therefore wanted to run some training where the actions were severely factored down and the number of steps greatly increased to see if the agent would train over time to be much more accurate and precise and it would explain the "always-spinning" aspect of the past versions. I changed the bounds of turning from [-π/6, π/6] to [-π/72, π/72], while forward movement I capped at 0.05m as opposed to 1.0m. (I also experimented with several different turn radii and movement caps: π/36, π/18, π/12, and 0.1, 0.25, 0.3).

This change allowed me to see how the agent would perform over a long time, and I saw that over enough steps, the angle would change to the maximum bound and the forward action would also go from the minimum to maximum and back while training. This method of training also allowed me to be more hands-off and allow the simulation to go through multiple sets of 1,000 steps training then testing an episode (with max 1,000 steps), updating the weights after each training and logging the results. Since the actions taken are smaller, they take less time in the sim, meaning each step is only 2 seconds versus 30-45 seconds for the larger turns/moves. I had thought the spinning was due to a logical error, but in fact it

was the algorithm working as intended. I was not allowing it enough time and space to learn properly and wanted to produce results more quickly. While I did manage to produce pretty good results with the faster version (less steps, higher turn radius and forward movement), I believe moving forward it will be more powerful to train the agent over many steps, and I also want to see how the agent performs after training as such versus during the previous tests. The main goal of this method/test is to reduce/remove the spinning and have the robot immediately go towards red if it is in screen, rather than turn away from it, make a full revolution, then realize there is red and go towards it. I also believe any future work using these methods should begin with smaller action space ranges and only get larger if there are time constraints. However, it should be noted this method takes a long time to train and should be done with time. A main worry I have with this method is that I will have to change the simulation environment, as one block of red may be too small of a reward as it is punished almost every single step it takes. Out of 1,000 steps, approximately 250 are positively rewarding. I do not know enough to say whether this metric is negative or positive, but only time and further research will tell.

Results & Analysis

Ultimately, I ran over 10,000 steps using the very small turns and movements forward. While the agent learned to turn in the correct direction over time, it still began maximizing the continuous action space (giving the maximum turn radius and maximum movement forward) every single time, even over the course of 10,000 steps of training. I tried starting the robot in different places in the environment and resetting weights and memory even, but the actions remained stagnant throughout, solely changing the first 125 or so steps without memory or weights as it would try out the minimum angle then seems to get stuck on the maximum. I am still not sure if the number of steps is too low per episode (I tested to a max of 2,000), but I cannot test much further as the robot will begin to walk into the walls and fall over after too many steps, ruining any further training. I also tested what would happen if I made the minimum turning 0 radians and the maximum $\pi/6$ (effectively meaning the robot can only turn in one direction). The first 125 or so steps went incredibly quickly as the robot did not move at all, forward or turning. Then the robot turned in place (the maximum) for 200-300 steps, then turned and moved maximum for another 300-500 steps. The agent is rewarded every time it sees the red square on its spins, yet I am not sure the rewarding of 20-40 is enough to counteract the -10 steps every single turn for almost a whole rotation (except the steps where the red square is in view). Overall, I thought making the steps smaller and quicker would provide a more accurate model. While I was able to train on more steps quicker, and the agent managed to turn the correct direction at first, I still did

not see the behavior I was looking for: a direct approach to red without spinning in circles. If I had more time to train the agent, I would see if over 100,000 steps made a difference and if the training was merely too slow to train the behavior I was looking for in my time-frame.

Design & Implementation V3.5

Upon further testing, I realized using a tanh activation for the output layer would result in a more desired action value than merely clipping the infinite continuous action space into the bounds I had, resulting in the minimum and maximum bounds being used for every action. The linear activation was causing my clipped actions to be "all or nothing," resulting in the aforementioned undesired behavior. With the tanh activation, the outputs are in the range [-1, 1], making them easily transformed into my action space ranges. This change led to the actions being more transparent and visible changing of weights actually affecting the robot more pertinently. Before, every small change in the weights was still truncated, but now it is visible that the weights change slightly every action, and the robot begins to make more meaningful moves. —(see video)–

Results & Analysis

In the 200 step tanh output training, the robot interestingly only turned right to get itself situated at the "correct" angle of incidence, turning left only when it neared the square to slightly position itself better. I thought the robot would learn to separate turning and moving, but after only 200 steps, the robot had only learned to stop and make a full turn rather than turn while walking forward. Further training may tell if the robot learns to immediately turn left to begin walking towards the square. The square could potentially not be a good enough attractor to the agent, as the reward function does not include "intensity" of color (to an extent), so the agent does not immediately go towards the square. The robot instead chooses to explore different states, actions, and rewards, (albeit all the same negative rewards as it makes a full rotation). One can also see the angle of rotation to the right getting smaller as the robot continues to see nothing and be penalized, yet sometimes it gets larger as if to overcome the turn and get back to its initial state. The exploration is also affected by the random process, and I experimented with various θ for the Ornstein-Uhlenbeck noise (0.15, 0.25, 1.0) with some slight difference, but not much.

Design & Implementation V4

I then wanted to see what would happen with this trained model if I moved the square. So, the goal state would be changed but the weights and what was "learned" would remain.

First, I moved the square to the left of the robot's view, retaining a sliver of red, much like the first simulations with the two walls of red (see Figure 13).

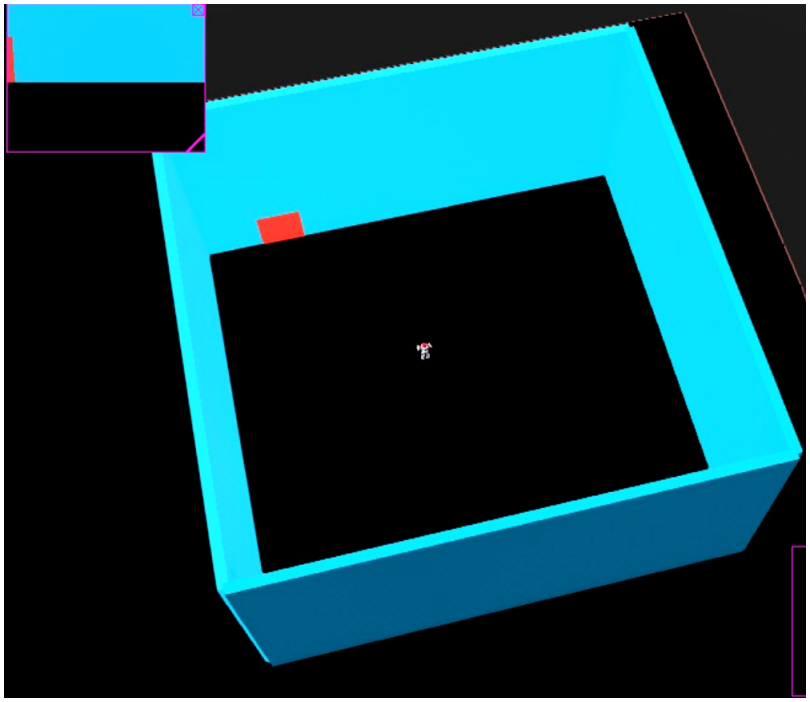


Figure 13: Simulation with Red Square to Left

The robot performed similarly to the previous experiment, but reached the goal more slowly, not converging after 200 steps of training and then 200 steps of testing. It took another 145 steps of training on top of where the testing left off to reach the the square (see video).

I also moved the square over to the right wall (Figure 14), wanting to see if the robot would latch onto it more readily if it naturally turned onto it, as the policy seems to start from the minimum and update from there (meaning the robot will likely always start by turning right before it is trained). I tested this both without memory or trained weights, as the trained weights would make the robot walk away from the square to "get into position" to shuffle over to the old red square spot (video).

Results & Analysis

The robot still seems to make the same kinds of decisions, turning around in circles and moving forward while turning every time the square is in view, only turning left at the end to adjust itself. At the end, one can see the broken threshold interaction discussed earlier, where the intensity of the pixels at close range produces enough "red" to give a positive step reward where a negative should be. I adjusted the threshold to see if this would help,

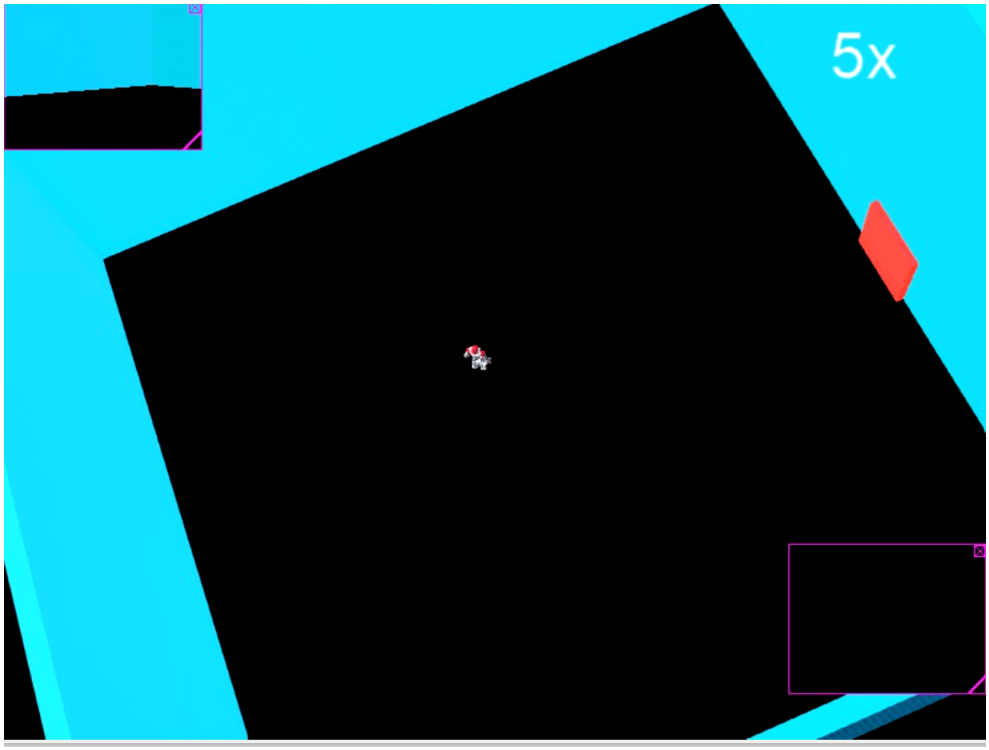


Figure 14: Simulation with Red Square to Right

but it also serves to cause negative thresholds when the red is in view sometimes from far away (lower intensity). To further combat this, I made a more piece-wise series of if,else statements granting different points depending on the intensity rather than all or nothing subtraction versus addition of points. After the threshold did not work still, I changed the factor to a difference between the red intensity and the threshold, divided by the number of pixels, hopefully giving a more accurate result. This method still does not deal with the fact that the "red" intensity of the cyan walls increase when they are close, but I am testing whether the red difference will be able to counteract this in the late stages of training. The "penalty" states (all cyan wall) are more variable in their negative reward using this method, but there are more clear delineations between positive and negative actions. This experiment still had an error near the end, as the robot kept repeating the same turns left and right over and over, getting a negative reward for moving away from the red but then getting a greater positive reward for turning back to the red square (video timestamp). This way, the robot is rewarding itself by "gaming" the system in a way (really, the reward function is just simple and poorly executed where the agent is rewarded for the difference between previous state's red and current red).

Either way, there is negligible difference between the square ending points, as the robot still learns and performs the same behavior. The only difference would be the time it would

take and the weights at certain state would say to turn to get in position for different ending areas.

Not to mention, this reward function would not even remotely work in a real-world scenario as it even hinges on RGB and lighting as well as solid colors for everything. Scaling it to HSV would not take much time, but did not fit into the scope of this project as the focus was more on improving learning. The reward function also was too variable, resulting in steps that should have had the same negative reward seeming "better" steps than the other, when in reality the robot was spinning and small changes in pixel values changes the step reward slightly. It is debatable whether this reward system could actually help, as it distinguishes (ever so slightly) between different states that otherwise look the same, potentially giving the agent a sense of "direction" or relative location.

Design & Implementation V5

Therefore, I would like to attempt a version of the experiment more similar to the NOMAD where the agent is first given short-term rewards solely based on magnitude of color, so the robot would seek out the closest color, be it red, blue, or green. Then, once reaching the color, the total reward would be added or subtracted to based on whether the color is red/green/blue/etc. This way, the robot first attempts to "explore" by approaching a color and learns long-term that even though blue is a color that can be approached and rewarded in short-term, the overall reward will be much better (positive) if the color approached is red. This method may allow for better long-term learning and a more robust approach to "learning" as a human or exploratory agent would, by testing all the options and experiencing the rewards or pains for itself. The environment I set up for this was a simple change from the past experiments, as seen in Figure 15. I changed the movement maximum to 0.5m to give the robot some more choice for forward movement. The turn radius was kept at $\pi/12$. All the blocks of color are around the same pixel intensity when close and the initial training was done over 500 steps (see video).

Results & Analysis

One major thing I noticed as I researched was that the learning tended to focus more on the turning as the changed action, rather than the movement. The turning action is the first in the action array, so this may have something to do with it. After the initial training of 500 steps, I decided to try training another 500 without any memory or loading weights, but changing the forward movement action to the 0th index of the action space array. We can see in the video of the first 500 steps of training in the NOMAD version that the robot turns

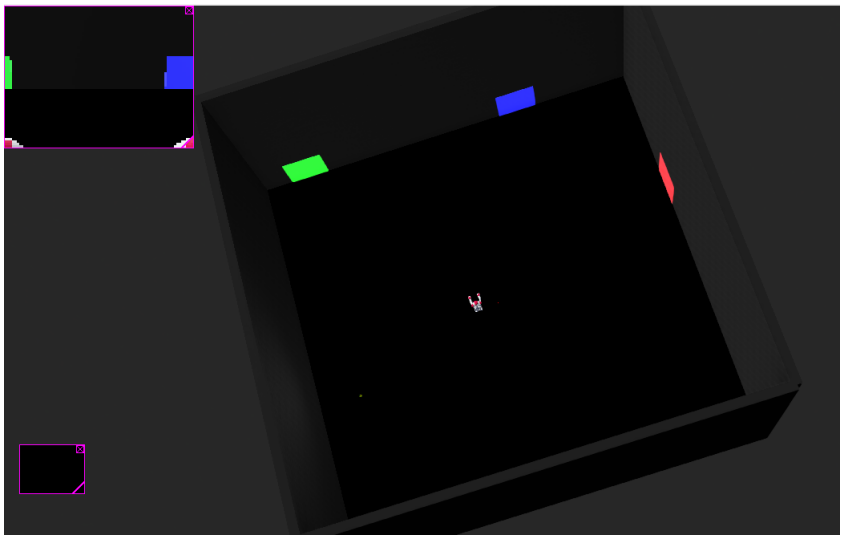


Figure 15: Simulation Environment with Different Colors

back and forth when close enough to the square to be producing a positive reward (albeit ¡ 1). I want to test the action space switch as I want to see if the robot will default to forward movement more often as exploration. This movement difference can also be due to the fact that I clip all negative output for movement to 0, and everything positive is multiplied by the maximum movement rate (0.5 in this case). I found that the agent still turned much more often than moving forward, but that the turning angles were much more varied, and the robot would often turn back and forth at the beginning of training. There is not as much of a change as I thought. The robot still does not walk forward as often, but the higher degree of precision on turning is promising, even going so far as to almost not turning sometimes, which has not happened in the tens of thousands of training steps on any experiment. A slight problem with this experiment is that I placed the blue and red too close together, so the robot is rewarded for getting them both in the screen together as it increases the intensity. The reward should not be nearly as great as walking close to them, but it takes a while for the robot to get to those rewards, as it is still not prioritizing moving forward (there are a particularly painful set of 250 steps or so where the robot turned minuscule amounts, then turned in a larger radius the other direction, then turned back slowly in small amounts, then went back and repeated; see video starting 13:30). As most of these conclusions go, more training could show a difference and I would continue to run these as much as possible to truly understand the depths and limits of the work. I would also be interested in testing this action space switch on all the previous experiments to see how it affects the learning. After the switch, I decided to change the way I scaled the tanh output to the forward movement, seeing if hopefully I could make it work a bit better. Therefore, I set -1 to 0 and 0 to 0.5

and 1 to 1, essentially squashing all of the negative tanh values into a positive movement. Then, I tested that for 500 steps, along with a slight numerical change in reward for pixel intensity and discount. This change meant the robot almost always moved forward, but the initial movements were all very small (often under .05m). Due to float overflows, sometimes negative movements were given to the NAO. Those negative technically make the robot back up, but the movements are small and done with turns, so it is usually masked as part of a stationary turn. If the backward movement becomes a problem, I can fix it by clipping all negative action values after the scaling to 0, (even though that is what the math should already be doing). I am also considering changing to allow backward movement in general, making the bounds of action space for moving [-0.25, 0.25]. After the initial changed attempt crashed into the wall after 150 steps (although it was very near the blue square), I tested this change with backward movement as part of the action space. With backward movement, the robot takes much longer as it automatically starts by exploring the minimum and updating from there, meaning the robot continually walks backwards while turning, reversing the behavior some previous experiments where it would walk forward and turn. This backward motion could mean the robot could take many thousands of steps to start learning how to walk forward, a resource I did not have in this resource. Therefore, I returned the experiment to only forward movement. (See video). I kept the action space change even though I am not sure it did much of anything at all as I think it is more clear that action[0] should be movement forward and action[1] should be the turning angle.

This environment and reward structure causes the robot to train more slowly, but the fundamental nature of its training is more astute and long-term, providing "better" learning. The reward function is still not complex whatsoever and can be improved greatly. As of now, the reward is a function of the "intensity" (RGB values) of the pixels, rewarding for pure intensity of color, then rewarding a large sum upon reaching red and penalizing a significant amount if the color reached is blue or green. This method (along with the previous methods) still has not managed to get past the problem of the black walls still containing pixel intensity. I have tried to subtract the base pixel value of the black wall, but the actual colors get subtracted out if the robot is too far away, causing incorrect behavior. This reward structure also maintains the same issue as the difference one, where "negative" states/actions all appear as different negative weights, and while some negative states/actions are better than others, the actual factors describing that difference are incorrect and arbitrarily based on the magnitude of pixels in that specific spot (also affected by the overhead light in the Webots simulation). To fix this, I realized the red, green, and blue pixel values for the walls were the same everywhere (except for the reflection of the overhead light), meaning I could do a simple equality check and set the reward to 0.0 when the robot was turning

against walls, meaning all the in-between states (states where no color is seen) were awarding equal reward (0.0). This smoothed out training and allowed there to be a marked difference when color showed while not penalizing for turning around and exploring. I will have to keep an eye on it to see if there is a difference in how the robot behaves in these states. Earlier training showed the robot favor the blue block first, but it would learn the overall reward was negative when reaching the blue block then test a different block. After doing that once, the rest of the time, it kept trending towards the blue block, even though it was generating less reward. I believe I need to train on completely new weights and memory, but the training takes a long time as it first goes to the blue, then must learn long-term to go towards red. I believe this reward function can truly achieve a long-term comprehension, albeit incredibly slowly over much training. The robot never attempted to go towards the green block, whether due to positioning, pseudo-random selection, only turning a certain direction, or other unforeseen factors. The robot also seemed to have some heavy slippage factors and overshooting of the bounds, so I reduced the maximum movement from 0.5m to 0.4m, hopefully allowing better movement.

Chapter 6: Conclusion

All in all, the three experiments were successful to different degrees.

Experiment 1 involved taking a picture and identifying a red ball using HSV colorspace and a mask along with the Circular Hough Transform. This experiment made use of OpenCV and was quick, easy-to-setup, and used throughout the project as a baseline testing.

Experiment 2 provided more struggles, namely with understanding the robot's motion and reverse engineering Lavy and Marshall's work to leverage my own identification method (CHT). In the end, I found my method was quicker to get running, but less powerful than the deep error correction Lavy and Marshall implemented. There are clear pros and cons of using a Minimum-Enclosing Circle with error correction versus a CHT, namely MEC+EC is more powerful (more robust to changes in shape or distortions due to angle, as it is not looking for only a circle initially, but creating an MEC around the largest red areas; however, it takes longer to get working and requires much more math.

Experiment 3 was where the real learning began, facing several successes and setbacks on every version of the experiment I tried. With the initial environment, the robot would always turn towards the red wall and get there, although it was unclear how powerful this was as it was essentially a 50/50 between right or left side. After changing the surrounding walls and attempting the "red difference" reward functions, the robot would always go towards the red square. Even when the square was moved, the only difference was the time it

took training to get to the square. While these experiments showed a cursory short-term learning, they did not capture the pure long-term value of exploring a color and wanting to explore a color, but then "tasting" it and seeing it as negative or positive. While the final experiments still tended towards the blue over and over, I believe the crux of this issue is time spent training. As the rewards are now long-term and granted per episode, the robot must train over a large amount of episodes to gain long-term understanding. This fits into what I wanted to do initially, but still achieved an objective of autonomously approaching red through Reinforcement Learning (through the earlier reward functions and short-term experiments). A major difficulty with training this agent is that it cannot be done (as of now) automatically, such as leaving it running over night. Each episode takes about 30 minutes - 1 hour (or more), depending on the max number of steps per episode. If it could be run all day without supervision, it would be great and allow for more powerful results. Achieving this would be in the next steps, requiring correction and fall detection, as the robot would sometimes fall over due to slippage/effort even when in the middle of the environment with no walls to hit. The nature of RL is to require a large amount of training to converge, yet I was unable to get an environment where the agent could learn indefinitely and continue to improve, as was shown in the keras-rl examples online (pendulum, car racing, mountain car, etc.).

The three together give a general basis for getting started with the NAO and

Future Steps

There is a good amount of work I would want to continue to solve many of the issues of this project. As I have said, I believe the learning would improve if it could be done indefinitely, so maybe rewarding corrective actions when close to a wall or making an infinite space with blocks in it so the robot would not fall (along with getting the NAO Fall Manager somehow working on Webots). A clear future step in this project would be to simply run more and more experiments, with varying environments and rewards.

Furthermore, decoupling the movement actions and having there be a separate "turn" and "move" action with their own continuous action spaces would potentially fix issues having to do with slippage and spinning, as there is not as much slippage if the robot walks straight without trying to turn. I could not complete this within the time, but it would definitely fall into the immediate next steps if I (or someone else) were to continue the project. While changing the actions from turn being the 0th index to move being the 0th index did not change much, it would be interesting to see if there was a way to have two discrete actions (move and turn) with continuous spaces. Using the DQN algorithm would also be interesting

to see if modeled on discrete actions (action[0] = move forward 0.3 m, action[1] = turn right 30 degrees, action[2] = turn left 30 degrees). It would be a good way to compare these methods and see how they work over the long-term in these different environments.

I believe this agent can become much more powerful as it showed a semblance of learning on few training steps with the larger actions, but that may have been solely due to how the environment was set up. Ergo, another further step would be to fashion more environments and experiments and run them in parallel on different computers to see which ways the robot could learn more quickly and where hangups in learning occur. I managed to do this to an extent, but would require more. I would like to attempt an environment where there are multiple red squares, but I thought it would be too clear for the robot if there were red squares on every wall and feed too much into the "perma-spinning" nature of the robot, obfuscating the true results of the experiments since the robot is being "rewarded" by the environment for faulty actions.

Ideally, multiple different reward functions would be run on multiple different environments. Now that the framework and environment is set up, it is much easier for a future student to activate the environment and immediately get to changing the reward functions, agent training and testing parameters, and the simulation environment. Moreover, the work scales to the physical robots and can be run on them, but the training should probably happen in a simulation as the NAO battery life is only around 2 hours. Future researchers can also change the agent used in keras easily if they want to explore a different agent or have a discrete action space and want to try DQN. As keras will be updated with new cutting-edge agents, this project scales through time. The major future work comes from porting the whole environment to Python 3, which depends on the NAO hardware being updated to a pertinent OS that can manage Python 3. A lot of the energy spent on this project went into figuring out how to make all the libraries I needed to work function with each other in outdated systems/libraries, and I would hope it would become much easier with Python 3 (and necessary, as pip and most packages will be deprecated for Python 2.7). The onus lies on Aldebaran to provide a Python 3 SDK, as even their latest NAO OS only has Python 2.7 SDKs. A future student could use C++ as well and merge the C++ and Python more easily than Java and Python, but it depends on the researcher's experience. Cython is also an excellent choice for working with Python and C++, but I did not explore this option during this project as I did not have time and wanted the work to be more agile and pythonic (more script-y). Cython is an optimising static compiler for Python and Cython (based on Pyrex), which allows you to call to and from C or C++ code natively in Python and achieves C performance by introducing static typing using Python syntax. If I had more time or had set up the design better earlier and learned about Cython, I would have probably used it.

Aldebaran encourages using both C++ and Python code to write for the robot, so it is recommended for a future researcher that is familiar with Python to use Cython and be able to juxtapose the two and potentially use the old C++ SDK in conjunction with Python 3.

I would also like to try changing the colorspace to HSV and attempting to fix the reward functions in this way. I chose RGB because I needed to get learning happening as quickly as possible and did not want to mess with masking or changes in Luminosity or Value that I did not understand. This change would potentially make this learning work more readily in a physical environment, with changing lighting. The lighting, even in the simulated environment, caused problems and changes in behavior, so I believe it would not be a small task (but not huge either) to convert everything to HSV and build a lighting-agnostic learning agent.

The primary objectives of this project seemed to shift over time as the focus became more on RL seeking red and being repulsed by blue/green and less around specifically significance detection (although one could say that knowing whether red is significant or not is a form of significance detection). Having said that, most of the primary objectives were completed, with adaptive learning being taken off the table later to focus on one aspect of learning (RL).

Overall, this project could have been much more successful had I been able to get training up earlier. That being said, I believe significant headway has been made, and all the setup I did to get this working can be expanded and changed very easily (depending on how Aldebaran chooses to keep SDK updated as well, or whether they will be making the switch to a Python 3 SDK in January 2020). Future researchers, whether at University of St Andrews or not, can use my work and fiddle with numbers or environments as well as the Webots simulation worlds to hopefully make important findings and more autonomous robots in the future.

References

(1) NAO - Technical Overview.

(2) Duda, R. O.; Hart, P. E. *Commun. ACM* **1972**, *15*, 11–15.

(3) Bradski, G. *Dr. Dobb's Journal of Software Tools* **2000**,

(4) Skinner, B. F. *The behavior of organisms : and experimental analysis / by B. F. Skinner*; Appleton-Century-Crofts New York, 1938; pp xv, 457 p. :.

(5) Sutton, R. S.; Barto, A. G. *Introduction to Reinforcement Learning*, 1st ed.; MIT Press: Cambridge, MA, USA, 1998.

(6) Watkins, C. J. C. H.; Dayan, P. Q-learning. Machine Learning. 1992; pp 279–292.

(7) Hasselt, H. v.; Guez, A.; Silver, D. Deep Reinforcement Learning with Double Q-Learning. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. 2016; pp 2094–2100.

(8) Silver, D.; Lever, G.; Heess, N.; Degris, T.; Wierstra, D.; Riedmiller, M. Deterministic Policy Gradient Algorithms. Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32. 2014; pp I–387–I–395.

(9) Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N. M. O.; Erez, T.; Tassa, Y.; Silver, D.; Wierstra, D. *CoRR* **2015**, *abs/1509.02971*.

(10) Uhlenbeck, G. E.; Ornstein, L. S. *Phys. Rev.* **1930**, *36*, 823–841.

(11) Lamb, C.; Weir, D. M. Evolving State-Based Behaviour through Adaptive Learning in Recurrent Neural Network. Ph.D. thesis, University of St Andrews, 2016.

(12) Li, Y.; Weir, D. M. NAO Robot Programming. Ph.D. thesis, University of St Andrews, 2017.

(13) Finlay, S. S. Programming the NAO Robot to Learn. M.Sc. thesis, University of St Andrews, 2019.

(14) Wang, D.; Lu, H.; Yang, M. *IEEE Transactions on Image Processing* **2013**, *22*, 314–325.

(15) Hester, T.; Quinlan, M.; Stone, P. Generalized model learning for Reinforcement Learning on a humanoid robot. 2010 IEEE International Conference on Robotics and Automation. 2010; pp 2369–2374.

(16) Butterfield, J.; Osentoski, S.; Jay, G.; Jenkins, O. C. Learning from demonstration using a multi-valued function regressor for time-series data. 2010 10th IEEE-RAS International Conference on Humanoid Robots. 2010; pp 328–333.

(17) Gadanho, S. C.; Hallam, J. *Adaptive Behavior* **2001**, *9*, 42–64.

(18) Konidaris, G.; Barto, A. Autonomous Shaping: Knowledge Transfer in Reinforcement Learning. Proceedings of the 23rd International Conference on Machine Learning. New York, NY, USA, 2006; pp 489–496.

(19) Erhan, D.; Szegedy, C.; Toshev, A.; Anguelov, D. Scalable Object Detection Using Deep Neural Networks. Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition. Washington, DC, USA, 2014; pp 2155–2162.

(20) Jimnez, A.; Ceres, R.; Pons, J. *Trans. ASAE* **2000**, *43*.

(21) Girshick, R.; Donahue, J.; Darrell, T.; Malik, J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition. Washington, DC, USA, 2014; pp 580–587.

(22) Dalal, N.; Triggs, B. Histograms of Oriented Gradients for Human Detection. International Conference on Computer Vision & Pattern Recognition (CVPR '05). San Diego, United States, 2005; pp 886–893.

(23) Albani, D.; Youssef, A.; Suriani, V.; Nardi, D.; Bloisi, D. A Deep Learning Approach for Object Recognition with NAO Soccer Robots. 2017; pp 392–403.

(24) Hild, K. E.; Erdogmus, D.; Torkkola, K.; Principe, J. C. *IEEE Trans. Pattern Anal. Mach. Intell.* **2006**, *28*, 1385–1392.

(25) Principe, J. C.; Xu, D.; Iii, J. W. F. Information-Theoretic Learning. 1999.

(26) Härtl, A.; Visser, U.; Röfer, T. Robust and Efficient Object Recognition for a Humanoid Soccer Robot. RoboCup 2013: Robot World Cup XVII. Berlin, Heidelberg, 2014; pp 396–407.

(27) Jung, B.; Sukhatme, G. **0002**, 980–987.

(28) Li, C.; Wang, X. Visual localization and object tracking for the NAO robot in dynamic environment. 2016 IEEE International Conference on Information and Automation (ICIA). 2016; pp 1044–1049.

(29) Whittaker, D.; E. Miles, G.; Mitchell, O.; D. Gaultney, L. *Transactions of the ASAE* **1987**, *30*, 0591–0596.

(30) Sengupta, S.; Lee, W. S. *Biosystems Engineering* **2014**, *117*, 5161.

(31) Lowe, D. G. Object Recognition from Local Scale-Invariant Features. Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2. Washington, DC, USA, 1999; pp 1150–.

(32) Lowe, D. G. *Int. J. Comput. Vision* **2004**, *60*, 91–110.

(33) Allen, J. G.; Xu, R. Y. D.; Jin, J. S. Object Tracking Using CamShift Algorithm and Multiple Quantized Feature Spaces. Proceedings of the Pan-Sydney Area Workshop on Visual Information Processing. Darlinghurst, Australia, Australia, 2004; pp 3–7.

(34) Lueders, N. O. Object Localization on the Nao Robotic System Using a Deep Convolutional Neural Network and an Image Contrast Based Approach. M.Sc. thesis, Technische Universitt Hamburg-Harburg, 2016.

(35) Cheng, M.-M.; Zhang, G.-X.; Mitra, N. J.; Huang, X.; Hu, S.-M. Global Contrast Based Salient Region Detection. Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition. Washington, DC, USA, 2011; pp 409–416.

(36) Riedmiller, M.; Gabel, T.; Hafner, R.; Lange, S. *Auton. Robots* **2009**, *27*, 55–73.

(37) Chang, C.; Wang, S.; Wang, C. Vision-based cooperative simultaneous localization and tracking. 2011 IEEE International Conference on Robotics and Automation. 2011; pp 5191–5197.

(38) Matarić, M. J. In *Robot Colonies*; Arkin, R. C., Bekey, G. A., Eds.; Springer US: Boston, MA, 1997; pp 73–83.

(39) Rosten, E.; Drummond, T. Machine Learning for High-Speed Corner Detection. Computer Vision – ECCV 2006. Berlin, Heidelberg, 2006; pp 430–443.

(40) Laporte, C.; Brooks, R.; Arbel, T. A fast discriminant approach to active object recognition and pose estimation. 2004; pp 91 – 94 Vol.3.

(41) Kingma, D. P.; Ba, J. *CoRR* **2015**, *abs/1412.6980*.

(42) Jackson, A.; Sukthankar, G. R. Learning Continuous State/Action Models for Humanoid Robots. FLAIRS Conference. 2016.

(43) Cheema, M. S.; Kahl, B.; Gnadt, T.; Prassler, E. An Active Learning Approach for Embodied Concept Learning. 2010.

(44) Lavy, D.; Marshall, T. Autonomous Navigation with NAO. 2015; `https://github.com/davidlavy88/autonomousNAO`.

(45) Michel, O. Naoqisim. 2018; `https://github.com/omichel/naoqisim`.