Reinforcement Learning Control for Energy-Recycling Actuators

CS234 Project Report

Erez Krimsky [1]

Abstract

Low mechanical efficiency of untethered robotic systems greatly limits their potential applications. To overcome this issue we have proposed an actuator system that uses an array of tension springs coupled with mechanical clutches to allow for energy transfer in to and out of the system. As springs generate force conservatively, in the right environment this system can use significantly less energy than a conventional motor. Planning which clutches to activate and deactivate over some finite time horizon leads to a combinatorial optimization problem for which traditional control theory approaches are not well suited. Therefore, we are interested in RL based approaches to produce robust control strategies.

1. Introduction

1.1. Motivations

In many mobile robotic applications power consumption requirements in actuators greatly limits the feasibility of autonomous systems. This is true even when negligible positive mechanical work is done by the robot on its environment as is the case for walking robots. Recent state of the art biped and quadruped robots have operating times often less than 2 hours (spo) (che) and sometimes even as low as 20 minutes (min). It is clear that to achieve the dream of autonomous mobile robots we need to reduce energy consumption in actuators. Here we investigate RL based control strategies for the proposed "Energy-Recycling Actuator" (ERA) system.

1.2. Proposed System

The proposed system consists of an array of n elastic elements each connected to two mechanical switches (clutches) allowing each elastic element to be connected to the actuator output, thereby transmitting a force, or locked in place, thereby maintaining its elastic energy (see Fig 1). For negative net work tasks, such as a robot walking down stairs, all force generation could be done by the elastic elements. For a zero, or slightly positive, net work task a small motor would be added to the system to overcome inefficiencies introduced by elastic losses and friction. The resulting system could generate large forces while obviating the need for a large motor and high power consumption.

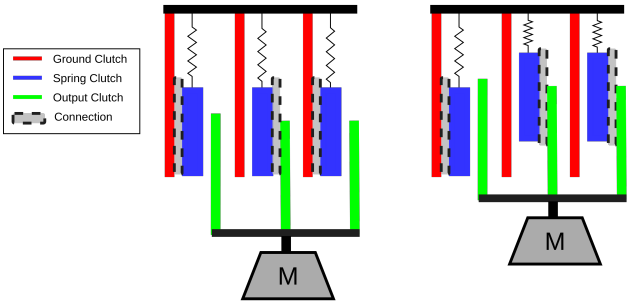

Figure 1: Example energy recycling actuator configurations. *Left:* 1 spring is attached to the output loaded with a mass and the system is in equilibrium *Right:* A second spring is attached to the output and the mass is raised.

We introduce the following variables to define the system dynamics

- $\mathbf{x} \in \mathbb{R}^n$ stretched spring displacements
- $\mathbf{u} \in \{0, 1\}^n$ binary control vector
- $\mathbf{K}$ diagonal matrix of spring stiffnesses
- $\mathbf{b}$ spring force offsets
- y actuator output position
- $\dot{y}$ actuator output velocity

To allow for higher elastic energy density the device uses elastomer springs instead of metal springs. Whereas standard metal springs follow a Hookean force-displacement

[1]Mechanical Engineering Department, Stanford University, Stanford, California, USA. Correspondence to: <ekrimsky@stanford.edu>.

relationship ($f = kx$), elastomer springs typically follow a non-linear Mooney-Rivlin force-displacement curve. As the springs in the ERA will only operate over a fixed displacement range we linearize the Mooney-Rivlin force-displacement curve over this range leading to an affine spring model given by $f = kx + b$ where b gives the spring force at some initial displacement where we define x to be zero.

The second order system dynamics can now be written as

$$m(y)\ddot{y} + \mathbf{u}^T(\mathbf{K}\mathbf{x} + \mathbf{b}) = f_{ext}(y, t) \quad (1)$$

$$\dot{\mathbf{x}} = \dot{y}\mathbf{u} \quad (2)$$

where $m(y)$ is configuration dependent mass (or inertia), $f_{ext}(y, t)$ is an external force that may depend on the position of the actuator and time. For now we restrict the study to exclude the additional motor and system inefficiencies. Although the system above is written for a linear actuator, mapping the dynamics to a corresponding rotary actuator where the springs apply torque about a joint, as is more likely in a robotic system, is trivial.

1.3. Optimization Objective

Given a desired reference trajectory $y_d(t)$, or goal position y_g, the objective is to minimize the error in system tracking. If a motor is included to make up for losses the objective now becomes a weighted sum of the tracking error and the motor power consumption.

2. Background & Related Work

2.1. Non-RL Approaches

The binary control vector $\mathbf{u} \in \{0, 1\}^n$ can take on 2^n different values at each time step so planning for h steps yields 2^{nh} possible solutions making any brute force attempts infeasible. Before introducing an RL based approach for this problem we briefly explain some more conventional approaches and why they may fall short.

2.1.1. Classical Control

Designing controllers to minimize tracking error and/or control effort has been studied extensively in literature with early solutions going back to the 1960s (Kalman, 1960). The classic formulation considers a Linear Time Invariant (LTI) system with dynamics given by

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \quad (3)$$

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \quad (4)$$

After discretizing the dynamics, the optimization (minimization) objective is written

$$J = \sum_{i=1}^{N} (\mathbf{y}_i - \mathbf{y}_g)^T \mathbf{Q} (\mathbf{y}_i - \mathbf{y}_g) + \mathbf{u}_i^T \mathbf{R} \mathbf{u}_i \quad (5)$$

which penalizes tracking errors and control usage based on the scale of the entries in $\mathbf{Q}$ and $\mathbf{R}$. As the optimization objective is jointly convex in $\mathbf{x}$ (or $\mathbf{y}$) and $\mathbf{u}$ and the linearized dynamics contain only affine equality constraints, the problem can be solved to arbitrary precision with any commercial convex optimization solver.

These approaches fail for our system for 2 reasons. Firstly, the control space (binary vectors) is non-convex which turns the control problem into a boolean optimization problem. Lower bounds or feasible (yet suboptimal) solutions for binary control problems can often be found by solving for $\mathbf{u} \in [0, 1]^n$ and rounding the results. For the ERA however, the discretized dynamics constraints are non-affine and cannot be solved by conventional methods.

2.1.2. Combinatorial Optimization

Finite horizon binary control optimization has been shown to be NP-hard (Garey & Johnson, 1979) (and in many cases NP complete) although efficient solvers that take advantage of problem structure have been written for many cases. We can construct the control problem as a non-linear integer program (NLIP) and solve it with a Branch & Bound search method.

Branch & Bound methods are frequently used for discrete combinatorial problems and guarantee optimal solutions (Wolsey, 1998). When a problem has structure that can be easily exploited or can be efficiently converted to a relaxed form, Branch & Bound algorithms are especially effective. However, when this is not the case, these algorithms lead to an exhaustive or near-exhaustive search and are therefore infeasible. As the problem cannot be easily relaxed to a convex problem, efficient computation of useful lower cost bounds in the search tree would rely on solving the convex dual problem which will not provide tight bounds in many cases which leads to an infeasibly large search tree. Additionally these methods may be too slow for real time high bandwidth control.

2.2. RL Background

Recent work has shown the ability of RL algorithms to perform well in extremely high dimensional state spaces (see DQN (Mnih et al., 2015)), however proficiency in high dimensional action spaces is much more limited. Value-

based algorithms like DQN define their policy by

$$\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, a) \tag{6}$$

As such, algorithms that work well for ERA control for $n = 3$ (corresponding to 8 discrete actions) are often intractable for $n = 10$ (corresponding to $1,024$ discrete actions) as the $\arg\max$ operation needs to be performed over all possible actions. Extending value-based algorithms like DQN to continuous domains typically involves a discretization of the action space which is intractable for high dimensional action spaces. In contrast to value-based algorithms, policy gradient algorithms learn a a stochastic parametrized policy that does not depend on taking a maximum over all possible actions. As such policy gradients have become widely used for large action spaces. As the policy is initialized as a distribution over the action space, exploration occurs naturally. If the optimal policy is indeed deterministic, the distribution of the stochastic policy should converge to this over time.

2.2.1. DDPG

Building off the previous success of (stochastic) policy gradient algorithms the Deterministic Policy Gradient (DPG) (Silver et al., 2014) learns a deterministic policy using the deterministic policy gradient which is equivalent to the *expected value* of the gradient of the action-value function. DPG has been shown to be much more sample efficient than stochastic policy gradient algorithms in many cases. As a completely deterministic policy naturally prevents exploration to overcome this DPG selects actions according to a stochastic behavior policy but uses an off-policy actor-critic algorithm to estimate the action-value function and update the policy using the deterministic policy gradient (Silver et al., 2014).

The Deep DPG (DDPG) algorithm incorporates insights from DQN (Mnih et al., 2015) to stabilize and speed up training. DDPG incorporates a replay buffer to minimize correlations between samples and trains the network using a separate target Q network to improve stability (Lillicrap et al., 2015).

2.2.2. Wolpertinger Architecture

In the standard actor-critic DDPG approach (Lillicrap et al., 2015), the actor network generates a single action based on the state which is then evaluated by the critic network which avoids taking a costly $\arg\max$ over all possible actions, which would grow linearly with the size of the problem space. The Wolpertinger architecture effectively takes a middle of the road approach between DDPG and linear complexity of DQN. Instead of looking at all possible actions, k nearest neighbor actions are found using the "proto-action" generated by the actor network (Dulac-Arnold et al., 2015). All k neighbor actions are then evaluated by the critic network which takes a maximum over all k actions. Setting $k = 1$ amounts to a plain DDPG evaluation. Setting $k = |\mathcal{A}|$ more closely resembles a DQN approach and often takes infeasibly long to train. When k is kept relatively small taking the $\arg\max$ over k actions is trivial. As the maximum needs to be determined every time an action is generated a high k is prohibitive for real time control as well as training. Given its ability to learn in large discrete action spaces, the Wolpertinger architecture was used to train RL agents in this work.

3. Approach

3.1. OpenAI Gym Environment

To allow for testing on a variety of RL implementations the MDP described below has been encoded in an OpenAI gym environment. The environment simulates a 0.35 m 1-DOF arm with a 5 kg mass on its end under gravity. The elastic elements in the actuator provide counter torques which can move the arm up. Every time the environment is reset the goal position and initial start angle of the arm are randomly generated. This is done to give experience over a large portion of the state space.

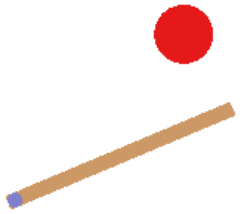

Figure 2: Rendering from custom gym environment using the "classic control" rendering environment. The red ball shows the goal location for the end of the beige arm.

We treat the environment as episodic and impose early termination conditions so extensive time is not wasted on overly poor trajectories. We define terminal conditions for when the arm swings outside a predefined window and for when spring lengths are outside a specified range. This corresponds to configurations that would physically break the

actuator or exceed the holding force of one of the internal clutches. If the arm reaches the goal state with sufficiently low velocity (as defined by the environment) the episode ends early with a large positive reward. In the physical system this would correspond to locking the arm in place once it reaches the desired position which is only feasible if the kinetic energy of the system is sufficiently low.

3.2. MDP

The MDP for the ERA is given below.

3.2.1. REWARDS

The rewards for the MDP are defined as

$$R(s', a, s) = \begin{cases} -200, & \text{invalid configuration} \\ +100, & \text{goal reached} \\ -(y_d(s') - y(s'))^2, & \text{otherwise} \end{cases} \tag{7}$$

The reward on non-terminal steps parallels the quadratic state penalty commonly used in control theory. For a system with a motor we would add an additional term that penalizes motor power consumption in a similar manner. The design of rewards on the possible early termination states $(-200, +100)$ will be discussed later.

3.2.2. ACTIONS

Although the control vector $\mathbf{u}$ is in $\{0, 1\}^n$ we can model the corresponding control input as a single integer with n bits.

3.2.3. STATES

Minimally, the state of the system should include $\mathbf{x}$, y, and $\dot{y}$. Both $\mathbf{x}$ and y are constrained to be within a range of valid values based on the physical dimensions of the actuator. When inefficiencies are added to the model we have elastic losses every time there is a change in $\mathbf{u}$. To allow for this without violating the Markov assumption we include the previous control input in the state. We also augment the state with a desired goal state or trajectory instead of considering this to be a feature of the environment itself. This allows the desired control to be purely a function of the MDP state and allows for the environment to easily generate random goal positions whenever it is initialized.

3.2.4. TRANSITIONS

For now, we assume a deterministic transition model. The transition model is defined by discretizing the continuous dynamics outlined above and applying euler integration.

3.2.5. DISCOUNT FACTOR

We use a discount factor $\gamma = 0.99$ as is common in RL literature. Empirically it has been shown that using < 1 helps with convergence in many cases.

3.3. Wolpertinger Architecture Implementation

The base code for the Wolpertinger implementation can be found here (jim). In the general implementation the actor network generates a single "proto-action" and an approximate k nearest neighbors search is performed to generate k actions to be evaluated by the critic network. For an m dimensional continuous control tasks the the control space is mapped to an m dimensional unit hypercube. The approximate nearest neighbor (ANN) search is then done within the hypercube by calling a fast ANN subroutine (using FLANN in the original implementation) with $\log |\mathcal{A}|$ time complexity.

Representing n dimensional binary vectors as 1-D discrete numbers allows us to generate k nearest neighbors in constant time. For example, if the actor network generated the action 9, the corresponding neighbors with $k = 3$ would be $8, 9, 10$. Whether or not the computed neighbors are similar in terms of resulting system response depends on the force offsets of each spring and the order that these are stored in $\mathbf{b}$. The value b_i corresponds to the force output from the ith spring at some initial position. To best take advantage of the Wolpertinger architecture we arrange the entries of $\mathbf{b}$ in ascending order (since higher initial forces typically also means a higher spring stiffness the diagonal of $\mathbf{K}$ ends up similarly sorted). This leads to the torque output vs. control input show in Fig. 3.

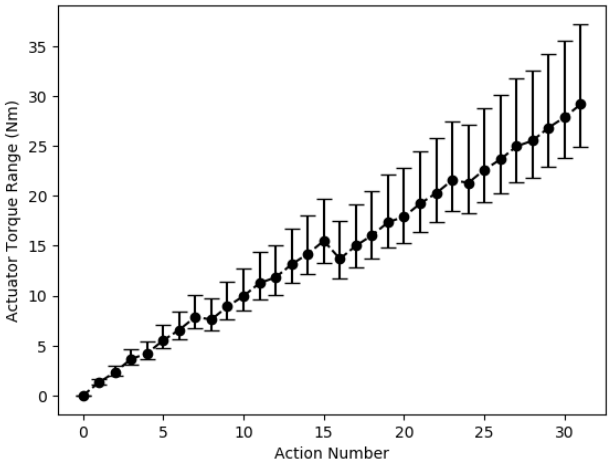

Figure 3: Range of state-dependent torque outputs for the ERA with $n = 5$ for different control inputs.

A naive implementation could alternatively use the Hamming distance (bitwise distance) between the bit representations of different actions. When each spring has the same parameters this may be a good approach. When there is a large difference between spring parameters in the device this approach can have poor results. This can be seen in in Fig. 3 considering control actions 15, 16, and 31. The Hamming distance between 15 and 16 is 4 whereas the Hamming distance between 31 and 15 is only 1 although actions 15 and 16 are more similar in torque output than 15 and 31.

4. Experimental Results

The DDPG algorithm with the Wolpertinger architecture was tested for $n = 3,\ 5,\ 8$ and 10 using $k = 1\%, 10\%, 50\%, 100\%$ of $|\mathcal{A}|$ as well as a $k = 1$. Setting the number of neighbors to 1 is equivalent to bypassing the additions of the Wolpertinger architecture entirely and just training with DDPG. Using $k = |\mathcal{A}|$ leads to very long training times and is only feasible for moderately sized action spaces. Training curves are shown in Fig 4.

The performance of trained agents is also tested on a separate set of 100 randomly generated starting configurations. These agents can be compared against 1-step greedy agents which simulate all 2^n possible actions for 1 time step and choose whichever leads to the highest immediate reward. The training results and performance of the trained and greedy agents is given in Table 1. All training performance metrics are defined as the average total reward for 64 consecutive training episodes.

For the cases of $n = 3$ and $n = 5$ the training quickly converged within the first $2,000$ episodes without significant divergence afterwards. The trained agents significantly outperform greedy agents for these cases. In both cases, the maximum averaged score reached and the final trained score increase with an increase in k which is to be expected as long as training does not diverge.

For the $n = 3$ case rewards start to rise significantly faster using $k = 4$ vs $k = 1$ however the added computational cost of $k = 8$ seems to eek out only negligible improvements. For the $n = 5$ case we see that increasing k from 1 to 3 which is only 10% of the total action space causes the training to be significantly more stable and achieve high rewards with much fewer episodes.

Training was carried out for many more episodes for $n = 8$ and $n = 10$ with the (futile) hope of consistent convergence which was not achieved in most cases. For the $n = 8$ case for $k = 1,\ 128$ and 256 there are periods where high rewards are achieved after which the training diverges. Applying an early stopping criteria to the training once it reaches some threshold performance may be one way to avoid this issue in a future implementation. Interestingly, the $n = 10$ case appears to converge for $k = 1$ but it converges to a low value. This will be discussed further in the next section.

Table 1: Summary of training results and testing results for trained and 1-step greedy agents

n	k	Final Avg	Max Avg	Test Avg	
				Trained	1-Step Greedy
3	1	98.49	98.97	98.06	50.498
	4	98.63	99.29	99.15	
	8	99.09	99.39	99.17	
5	1	98.07	98.58	86.70	53.34
	3	98.94	99.17	98.98	
	16	99.17	99.24	99.20	
	32	99.14	99.32	99.16	
8	1	-77.64	98.32	-98.09	56.33
	25	-203.15	37.02	-203.16	
	128	-11.40	99.23	-26.81	
	256	-112.86	99.31	-142.45	
10	1	-16.52	14.74	-14.61	56.39
	102	-78.97	-11.03	-16.41	
	512	-102.36	71.09	-91.20	
	1024	-152.67	-11.56	-196.72	

5. Discussion

If the optimal solution was reached for each training case, we would expect that the total expected reward increases with n as increasing the number of springs effectively increases the number of viable control actions that can be taken. This trend holds true when looking at the performance of 1-step greedy agents. However for trained agents with $n = 3$ and $n = 5$ we do not see a significant difference perhaps because we are only trying to reach some goal position and not trying to track a complicated trajectory at every time. A true input tracking problem would likely require the finer control capabilities of more elastic elements. This will be investigated in future work.

As in all reinforcement learning domains the design of the reward function impacts whether or not learning will be feasible. A variety of reward functions were tested before settling on the one described in section 3 above. To avoid the issue of learning with sparse rewards we add a negative quadratic position error reward for every time step. As rewards are negative on all non-terminal time steps one way to minimize penalties would be to drive the actuator into an invalid configuration as soon possible to end the episode and stop accumulating negative rewards. For this reason we add a negative penalty on ending the episode in an invalid configuration. With this reward structure the optimal move for the agent would be to reach the goal state

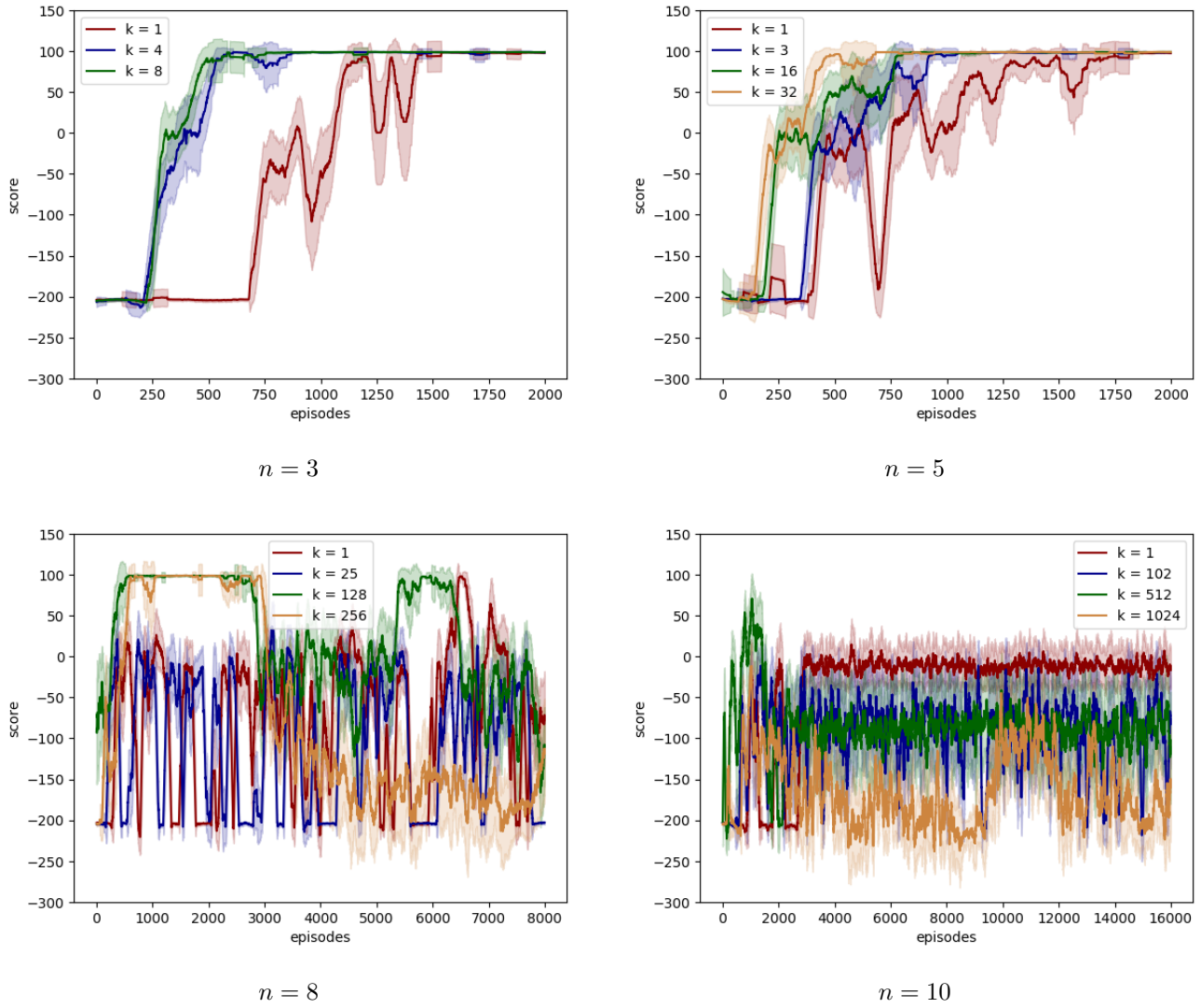

Figure 4: Training curves for $n = 3, 5, 8$ and 10 with various k values. Note the different number of episodes each agent was trained on. Values are given as averages over 64 consecutive episodes.

and end the episode as early as possible without violating any constraints. However, without an extra incentive to reach the goal state the agents would sometimes take a conservative approach and avoid negative configurations and run out the episode time limit without achieving the goal. For this reason an extra positive reward of $+100$ is added for reaching the goal state.

Although this reward structure can lead to trained agents as it does consistently for $n = 3$ and $n = 5$ it leads itself to local optima that agents can get stuck in during training. One of these local optima is when the agent is performing poorly and running into infeasible configurations. This corresponds to relatively consistent periods of average rewards very close to -200. If the policy will lead to an invalid configuration anyway, it is optimal to do this as fast possible to minimize the accumulated negative tracking error rewards. Once the agent has learned to avoid infeasible configurations it still may fail to consistently reach the goal. This appears to be happening for the $n = 10$ case with $k = 1$ where the average rewards are near zero at the end of training indicating that the agent can consistently avoid bad configurations but only occasionally reach the goal. For the $n = 10$ case with $k = 102$ and 512 as the rewards are well above -200 it appears to be consistently avoiding infeasible configurations but (almost) never getting the positive reward from the goal. It may be beneficial to remove the early stopping conditions and large negative rewards for infeasible conditions. This would inherently mean simulating configurations which are

not physically possible but this would cease to be an issue once the policy converges to consistently finding the goal. This will be investigated in a future implementation.

6. Conclusions & Future Work

The results are very promising for the low and intermediate dimensional cases with $n = 3$ and $n = 5$. Although an algorithm like DQN could likely achieve similar performance for $n = 3$ it may not succeed for $n = 5$ and would certainly have taken significantly longer to train. Although training ultimately diverged for $n = 8$ given that high rewards were achieved at some point during training it seems feasible that we could learn a good policy in this case. For $n = 10$, corresponding to $1,024$ discrete actions, high rewards are not achieved at any point during training. Overcoming these issues for the higher dimensional action spaces could entail making modifications to the structure of the reward function. Adding a learning rate scheduler and batch normalization could also potentially improve training.

Future work will include incorporating friction and elastic losses into the environment as well as adding a motor for injecting energy to overcome these losses. Control optimization will then include optimizing motor torques as well clutch engagements.

7. Appendix

The github containing project code can be found at the following link: https://github.com/ekrimsk/CS234_Final_Project

References

https://spectrum.ieee.org/automaton/robotics/robotics-hardware/mit-cheetah-robot-running. Accessed: 2019-3-17.

https://github.com/jimkon/Deep-Reinforcement-Learning-in-Large-Discrete-Action-Spaces/tree/discrete-and-continuous. Accessed: 2019-3-17.

https://www.vectornav.com/docs/default-source/case-studies/ghost-robotics-case-study-(12-0011).pdf. Accessed: 2019-3-17.

https://www.bostondynamics.com/spot-mini. Accessed: 2019-3-17.

Dulac-Arnold, G., Evans, R., Sunehag, P., and Coppin, B. Reinforcement learning in large discrete action spaces. *CoRR*, abs/1512.07679, 2015. URL http://arxiv.org/abs/1512.07679.

Garey, M. and Johnson, D. "strong" np-completeness results: Motivation, examples, and implications. *JACM*, 3, 1979.

Kalman, R. Contributions to the theory of optimal control. *Bol. soc. mat. mexicana*, 5.2:102–119, 1960.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518: 529 EP –, Feb 2015. URL https://doi.org/10.1038/nature14236.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. Deterministic policy gradient algorithms. *JMLR*, 32, 2014.

Wolsey, L. *Integer Programming*. John Wiley and Sons, 1998.