Robustness of Control via Deep Reinforcement Learning

Sivaraman Rajaganapathy, Zeren Shui, Sijie He
Wednesday 23rd November, 2022

I. Motivation & Problem

The recent development of deep reinforcement learning [1] has given us a method of generating control algorithms for complex systems with high dimensional sensory inputs. This has generated interest in applying deep reinforcement learning to physical systems' control such as self driving cars, robot manipulation etc [2]. If any such technique is to be used for mission critical systems' control, we need some guarantees on the robustness of the control method. There are two main reasons for such a requirement:

- If the controller is trained on a model of the system, a robust controller will be immune to errors and uncertainties in the model. Conversely, one can use a simpler or less precise model to train the controller saving on the cost of model building.
- If the controller is trained on a system directly, a robust controller will work on variants of the system, thus avoiding retraining efforts.

This project aims to quantify and improve the robustness of control strategies derived using deep reinforcement learning.

The cart-pole system from OpenAI Gym [3] will be used as the testing system. Although the deep reinforcement technique is meant for high dimensional inputs (such as images), due to computational constraints, the actual states (angles, velocity etc) of the test system will be used. The problem formulation will be restricted to discrete action variables and to systems which have an analytical dynamic model. This is to aid in the validation and comparison with traditional model based control algorithms.

II. Background

Q-learning is a model-free reinforcement learning technique. It can be used to find an optimal action-selection policy for a given Markov Decision Process. In the recent decade, a DEEPRL system which combines Deep Neural Network with Reinforcement Learning has been proposed in [4]. In 2013, a new algorithm called Deep Q Network(DQN) was introduced by Google's DeepMind in [5]. The task for Neural Network in DQN is to learn which action to take based on the observations (screen shot) and previous experience. Essentially, the neural network was used to represent the Q-table efficiently. In this project, we apply DQN model as the reinforcement learning model for cartpole game.

A robust agent should be able to recognize and adapt to the differences between the current environment and its training environment while deployed. This means an extra module for adapting to disturbances or uncertainities should be included in the agent during training. Based on H_∞-Control theory, Morimoti et al, [6] proposed a policy learning method called Robust Reinforcement Learning (RRL), to increase the robustness of reinforcement learning. In RRL, they formed the learning process as a two player zero-sum game and trained two agents, a protagonist agent and an adversary agent, by solving a minimax problem. The adversary agent tries to minimize the reward that the protagonist agent gets. Each agent learns its policy while the other keeping their policy fixed. The learning process iterates alternatively until both agents converge. Recently, Pinto et al [7] incorporated the RRL paradigm to deep reinforcement learning and proposed Robust Adversary Reinforcement Learning.

In our project, we have proposed some ideas inspired by the previous work to improve the robustness of the DQN method. We are trying to find an efficient and economic way to familiarize the DQN agent to potential disturbances and model uncertainties it may face in the real world.

III. Experimental Setup

The environment we used for the project was the CartPole system from OpenAI [3]. The CartPole system will be reffered to as the 'plant' in this report. The system consists of balancing a pole connected with one joint on top of a moving cart. The only actions are to add a force of -1 (left) or +1 (right) to the cart with a fixed magnitude, pushing it left or right. In the CartPole system's environment, there are four states at any given observation sample, representing the angular position & the angular velocity of the pole and the position and velocity of the cart. Using these observations, the agent needs to decide on one of two possible actions: move the cart left or right, to keep the pole upright. There are six environment parameters, gravity (9.8), mass of the cart (1), mass of the pole (0.1), length of the pole (0.5), magnitude of force (10) and sampling time (0.02).

For training a reinforcement learning controller, a reward function based on the position of the cart and the angular position of the pole is created as follows:

$$R(x,\theta) = \begin{cases} 0.1, & \text{if } (x \leq 2.4m)\&(\theta \leq 12deg) \\ -1, & \text{otherwise} \end{cases} \tag{1}$$

For representing a the Q-Function in Deep Q learning, we used the neural network algorithm. We used two different architectures for the neural network, a shallow network with 1 hidden layer and hyperbolic tangent activation function, and a deep network with 2 hidden layers and relu [8] activation functions.

IV. Measuring Robustness

For this project, the key requirement for quantifying robustness has been to systematically and quantitatively compare the robustness of two or more controllers. Since the controllers generated here are non-linear system (Neural Networks) that are not amenable to first principles analysis, we measured robustness empirically. This was possible as we were working with a simulated model of the plant.

Any robustness measure, analytical or empirical must eventually give a picture of of how the performance of the controller would change with changes in the plant parameters. For obtaining such

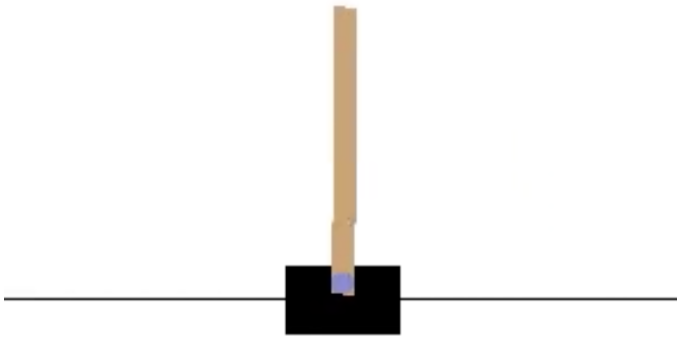


Figure 1: Cartpole Environment

a picture, we defined a parameter called the Difference Disturbance Ratio (DDR), which measures the change in performance of the controller from the nominal case as a parameter of the plant is varied. The exact equation of the DDR is given as:

$$DDR = \frac{\Delta P}{W * P_0} \tag{2}$$

where ΔP denotes the difference between the agent's performance on the nominal plant and the plant with change in parameters, W denotes the percentage change of a parameter and P_0 is the nominal performance. From the definition, an ideal robust controller would have a DDR of 0, whereas a fragile controller would have a DDR of large magnitude. The performance P is measured in terms of the survival time measured in seconds. The survival time is duration for which the controller is able to stabilize the cart-pole system when initialized randomly, within ± 0.05 radians from the unstable equilibrium position.

Although this definition of the DDR can only measure the robustness against one plant parameter, it can be extended for multi-parameter variations. For example, a linear combination of the DDRs of different parameters can be used. This would be reasonable in the case where the disturbance effects from the different parameters do not interact with each other.

V. Improving Robustness

In this project, we tried to develop modifications to the deep Q-Learning strategy [1], [5] that would result in robust control. Figure 2 shows a summary of the strategies used in this project to improve robustness. These ideas are described below:

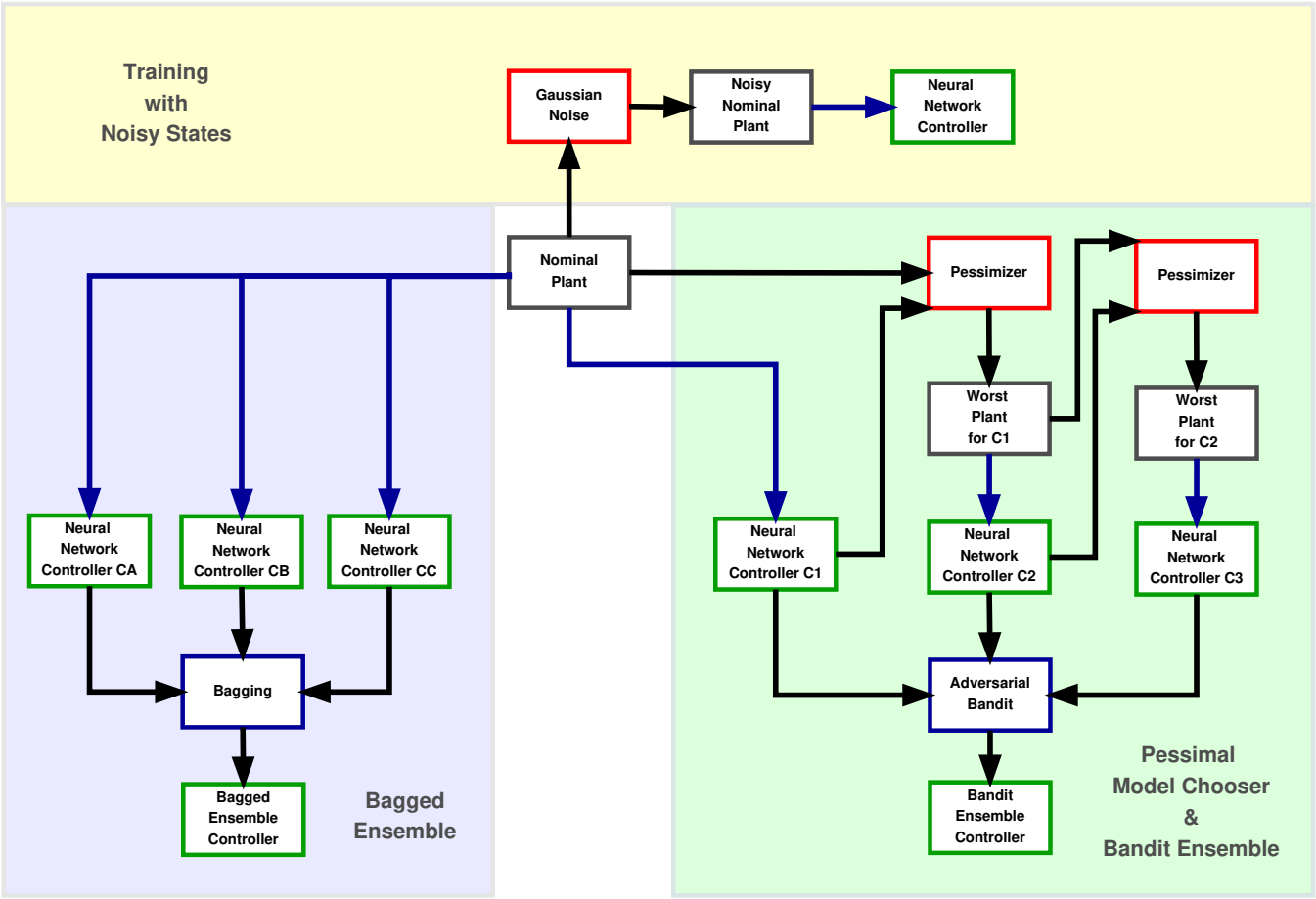


Figure 2: The key strategies used in this project for improving robustness. The main ideas were to train with noisy states, train multiple controllers for use in a bagged ensemble, and train controllers on worst case plants using a pessimizer.

A. *Extremum-model training*

A brute force method of generating a robust controller would have been to train on all possible variations of the plant model within the parameter bounds. A reasonable simplification would be do a brute force search in a discretized model parameter space. However, the computation effort for such a search would scale exponentially with the number of plant parameters. We could overcome this scale complexity if we had a way of selecting a subset of models that are the most difficult to control. This led to the idea of a 'Pessimal Model Chooser' described next.

B. *Pessimal Model Chooser & Ensemble Control*

The idea here was to train a nominal controller on the nominal plant, then find the 'worst' model variation for that controller. This is the task of the 'Pessimal Model Chooser', which is essentially an optimizer that minimizes the performance (survival time) of a controller over the constrained space of the plant parameters. We decided to restrict the plant parameter space to the pole length, mass of the

pole, mass of the cart, and the gravity.

Once we have the worst case model, we train a new controller that works best for that plant and repeat the process. This would lead to multiple controllers that work best on different variants of the plant. We then planned to use these controllers in an ensemble, which should then have the capability to withstand a wider range of plant parameters. The description of the solutions for the optimization and the ensemble problem follow:

1) **Hill-Climbing Search [9]**
Here, a form of Hill-Climbing Search in the parameter space is used to find the worst case parameter setting for a given controller.
For each parameter, the parameters were constrained within 10% to 200% of the nominal value. For each well trained controller, of which the expected survival time is 1000s, we picked up a parameter and compare the survival time of the controller with the parameter set to 120% and 80% of the previous value. The direction with the largest decrease in survival time is chosen as the direction for search. If neither directions result in a decrease of performance, the parameter is kept unchanged. Once the new direction is found, the parameter will continuously change in that direction with step sizes equal to 10% of the current parameter value. The algorithm terminates when the parameter reaches one of its bounds or the relative decrease in performance is greater than 10%. The hill-climbing search procedure repeats for each parameter, then the "worst" parameter setting is reached. A new controller will be trained based on the new parameter setting.

2) **Genetic Algorithm**
We looked for an alternative to the hill-climbing search method, primarily to improve the optimum found and to reduce the number of plant specific tuning parameters required. Many of the quasi-gradient search methods we employed used to get stuck in this problem. The reason was that most controllers had a significant parameter domain in which their performance remained unchanged. In those cases, the gradient of the performance with respect to small parameter changes were close to 0. The Genetic-Algorithm [10], a global optimization search method appeared to overcome this problem and generally resulted in finding good worst case models. The current version of the reinforcement learning algorithm however could not keep pace the genetic algorithm based Pessimiser, with no controllers that successfully trained on the worst case models generated.

3) **Bagging Controllers**
This idea is a variant of the concept of bootstrap aggregation (bagging) [11]. Multiple controllers are trained, each on the worst case models from the Pessimiser. All the controllers are then deployed simultaneously in an ensemble. Each controller is then treated as an independent expert. At each step, all the experts will generate their control action suggestions. The final decision is then a vote of all the experts. A direct implementation of such a technique did not produce the best results, as each controller works best for a different model variant. This also means that a simple vote is not the best choice - in an ensemble of 3 controllers, 2 could be wrong resulting in an incorrect ensemble control action.
To overcome this deficiency, we tried two methods. We trained multiple controllers on the same nominal plant, to form an ensemble, as was described in the original paper [11]. An improvement can be expected because multiple controllers trained on the plant are usually quite dissimilar,

the intended scenario for bagging to be used.
This however, still doesn't utilize the pessimal controllers trained on the plant variants. For a sensible combination of the pessimal controllers, we needed a weighted voting scheme. The weights in such a scheme would determine the importance of the action suggestions made by each controller. These weights should then be determined by 'expertise level' of a controller for the plant at hand. We attempted to that using adversarial bandits, described next.

4) **Adversarial bandit**
When several controllers have been trained with different parameter settings, the following problem is how to combine them together. An on-line learning algorithm, multi-armed adversarial bandit algorithm is introduced. A multi-armed bandit [12] algorithm can solve a sequential allocation problem defined by a set of actions. At each time step, a unit resource is allocated to an action and some observable payoff is obtained. Then the goal is to maximize the total payoff obtained in a sequence of allocations. In this scenario, the action at each time step is provided by the chosen controller and the angular deviation of the pole from the unstable equilibrium point (vertical direction) is regarded as the loss (negative of the reward). Then the problem is converted to keep choosing one "optimal" controller among all the possible controllers, such that we can maximize the total payoff, i.e. achieve the longest survival time. A controller will be randomly chosen based on an updated distribution given by the adversarial bandit algorithm. The pseudo code for this is shown in Algorithm1.

Algorithm 1: Exp3 (Exponential weights for Exploration and Exploitation)

Input : Parameters: $\eta_t \in \mathbb{R}^+$, $\gamma \in [0, 1]$.
Initialize probability distribution p_1 uniformly over K controllers.
for *each round* $t = 1, 2, \ldots, n$ **do**
  Choose one controller based on the probability distribution $I_t \sim p_t$
  Receive rewards $g_{I_t,t}$
  For each controller $i = 1, \ldots, K$ compute estimated gain $\widetilde{g}_{i,t} = \frac{g_{i,t}}{p_{i,t}} \mathbb{I}(I_t = i)$.
  Cumulative rewards for controller i is $\widetilde{G}_{i,t} = \widetilde{G}_{i,t-1} + \widetilde{g}_{i,t}$
  Update probability distribution over each controller:

$$p_{i,t+1} = (1 - \gamma) \frac{\exp(\eta_t \widetilde{G}_{i,t})}{\sum_{k=1}^{K} \exp(\eta_t \widetilde{G}_{k,t})} + \frac{\gamma}{K}$$

end

C. Training with noisy states

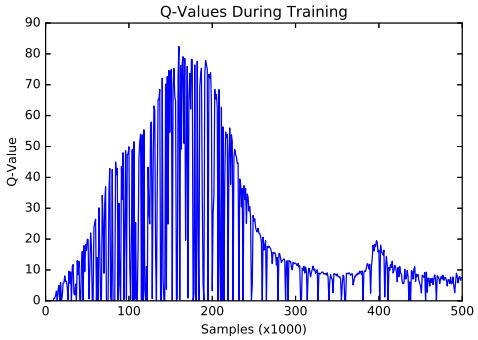
The idea here was to recreate the effect of parameter variations on the states, without actually changing the parameters. We wanted to see if this effect can be recreated by adding Gaussian noise of some arbitrary magnitude to each of the states. In all the methods described above, one needs a model and an estimate of the bound on the individual model parameter variations. This method, if successful could be used in the case where a model of the plant is not available. Such a model independent method could potentially be used without the explicit need to find the 'worst case' model. Our initial attempts

along this direction were not extremely successful. Controller training was unsuccessful for all but a few cases where the noise was added to a subset of the samples and was of small magnitude.

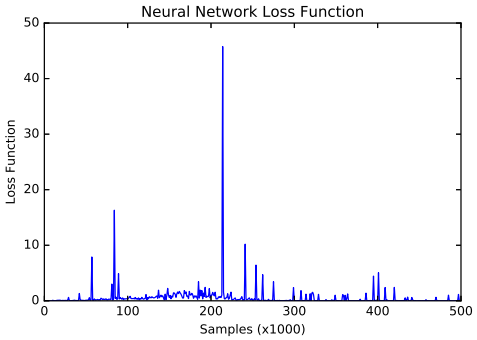
VI. Results

A. *Deep Q Training*

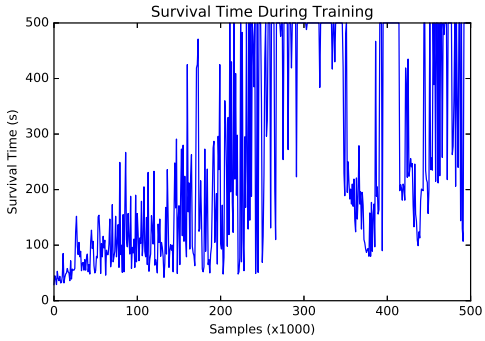
We have successfully implemented a Deep Q-learning controller on the cart pole system. The training process of this algorithm is shown in figure 3. This algorithm was heavily based on [13]. The figure shows the plots of the Q values, the neural network loss function, and the survival times on the cart pole system. We observe that over a period of about 500000 samples of the states, the Q function converges to the expected maximum discounted reward for indefinite survival (= 10). The loss function remains close to 0, indicating the neural complexity is sufficient to capture the Q function effectively. The mean survival times increase gradually to the maximum possible (artificial) cut-off of 500s. This algorithm usually produces successful results on most of the plant variants.



(a) Q values over the training iterations on a nominal plant. The Q values increase initially and settle to the expected maximum value of 10.



(b) Neural Network Loss Function over the training iterations. A low loss indicates good training on the replay memory.



(c) Survival Times over the training iterations. Survival time is capped at 500s.

Figure 3: The training progress of the DQN method modified to work on stabilizing the cart-pole system.

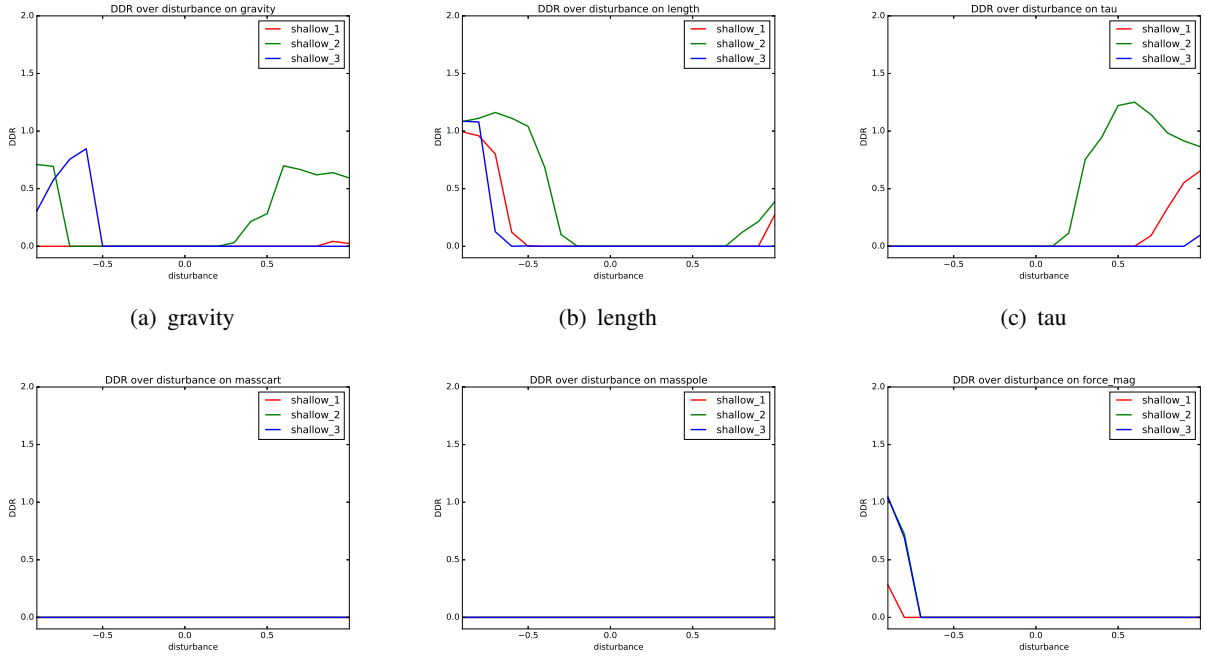
B. *Baseline - Shallow vs Deep*

We computed the DDR (measure of robustness - higher DDRs correspond to non-robust controllers) for four kinds of models, simple shallow (2 layer Neural Networks) models, simple deep models (4 Layer Neural Networks), deep models trained on noise states and bagging simple shallow models, over plant parameters ranging from 10% to 200% of the nominal. In all the figures, a DDR of 0 implies that the controller is perfectly stable, and therefore has an infinite survival time (in our experiments limited to 1000s, which is much larger than the time constant of the plant). The DDRs were computed for six parameters of the plant, varying one at a time, while keeping the others fixed at their nominal values.

As a baseline, we first computed the DDRs for simple controllers trained using DQN. The DDR results for shallow and deep neural networks are shown in figures 4 & 5 respectively.

From the figures we can see that, that the variations in masscart, masspole and magnitude of force do not have a significant impact on the performance of the baseline controllers.

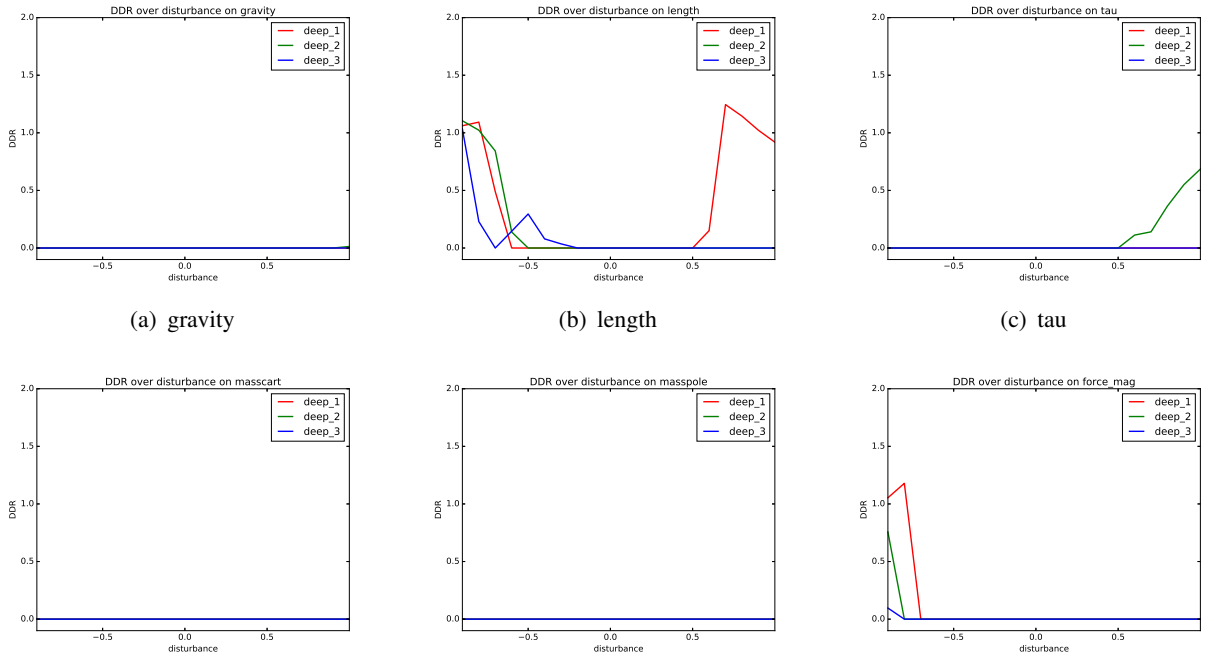
In figure 4 and figure 5 we can see that the performance of deep neural networks is better than the shallow neural networks on gravity and sampling time τ. While for the other two parameters, the shallow neural networks have a slightly better performance than thee deep neural networks for the magnitude of force and length. However, the deep neural networks appear to have better robustness overall.



(a) gravity (b) length (c) tau

(d) masscart (e) masspole (f) forcemag

Figure 4: DDR results for variations on 6 parameters for simple shallow neural network controllers.



(a) gravity (b) length (c) tau

(d) masscart (e) masspole (f) forcemag

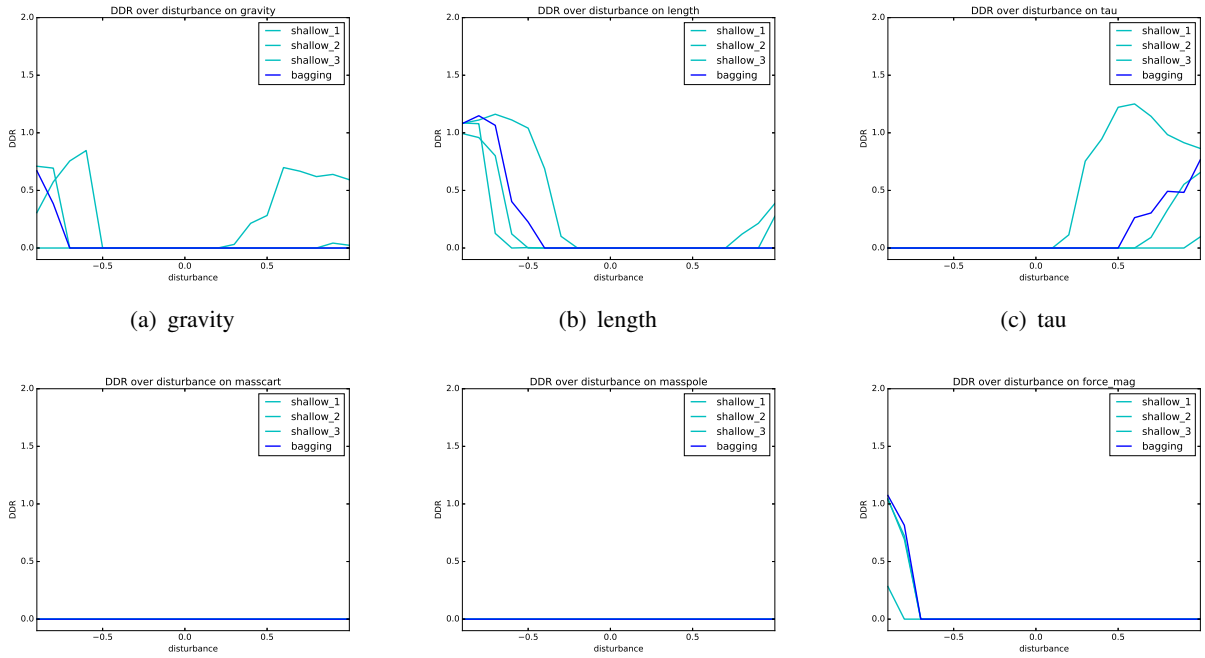
Figure 5: DDR results for variations on 6 parameters for simple deep neural network controllers.

C. *Ensemble Controllers*

The DDRs computed for the bagged controller composed of 3 shallow neural network controllers trained on the nominal plant are shown in figure 6. The DDRs for the bagged controller with deep constituents is shown in figure 7. We observe that for all of the parameters, the bagged controllers have DDRs lesser than the worst of their constituents in general. It is improbable for this ensemble controller to get a result better than or equal to its best constituent. This is because a simple voting scheme is used here and therefore unless a majority of the controllers have good performance, the ensemble will have poorer performance than the best. These results, while promising, need further work.

D. *Pessimizer Results*

The implementation of ensemble controllers built using the bandit algorithm and pessimal controllers is under progress. We tentatively have the results showing the effectiveness of the pessimizer in table 1. The table compares the survival times of different controllers on the plants they were trained with the survival times on the 'worst-case' plant found by the pessimizer. The survival times reduce significantly, indicating that the pessimizer is successful in finding the worst case plants efficiently.



(a) gravity (b) length (c) tau

(d) masscart (e) masspole (f) forcemag

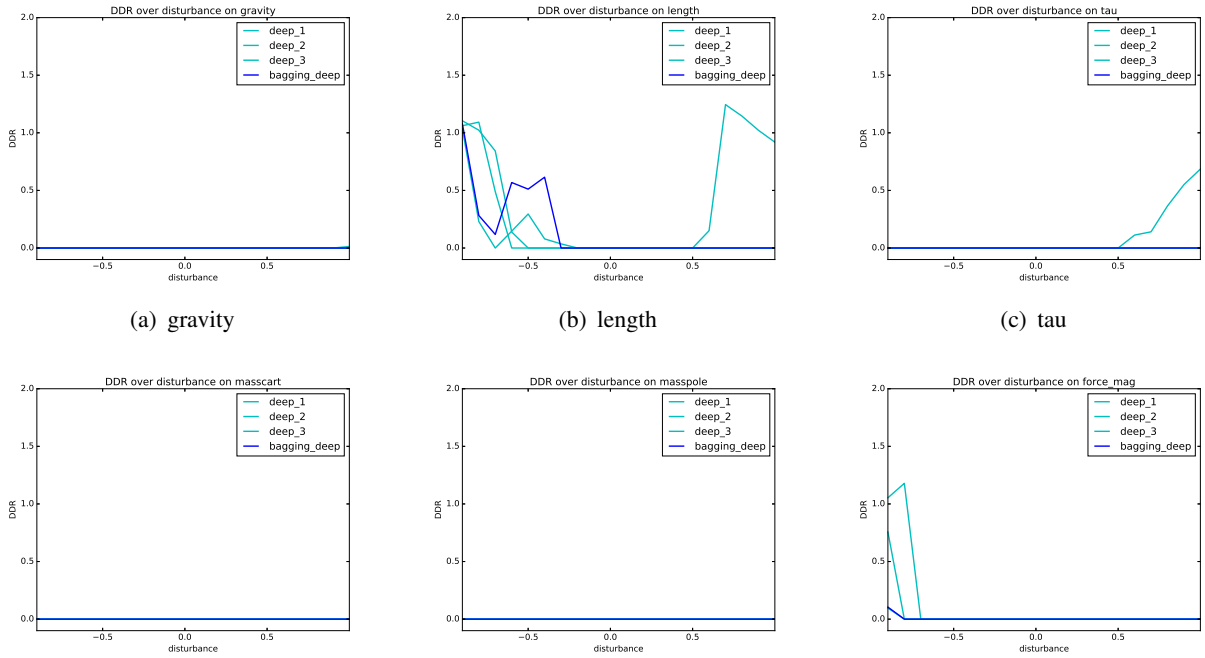
Figure 6: DDR results for variations on 6 parameters for bagged shallow neural network controllers.

Parameter	Model 1	Model 2	Model 3	Model 4
Trained parameters	500	500	500	500
Pessimal parameters	228.5	381.0	467.0	480

Table I: Comparison of survival times on the 'trained' plants and the pessimal plants. Here 'Model N' was the worst case model found by the pessimizer for 'Controller N-1' on 'Model N-1'.

E. Training with Noisy States

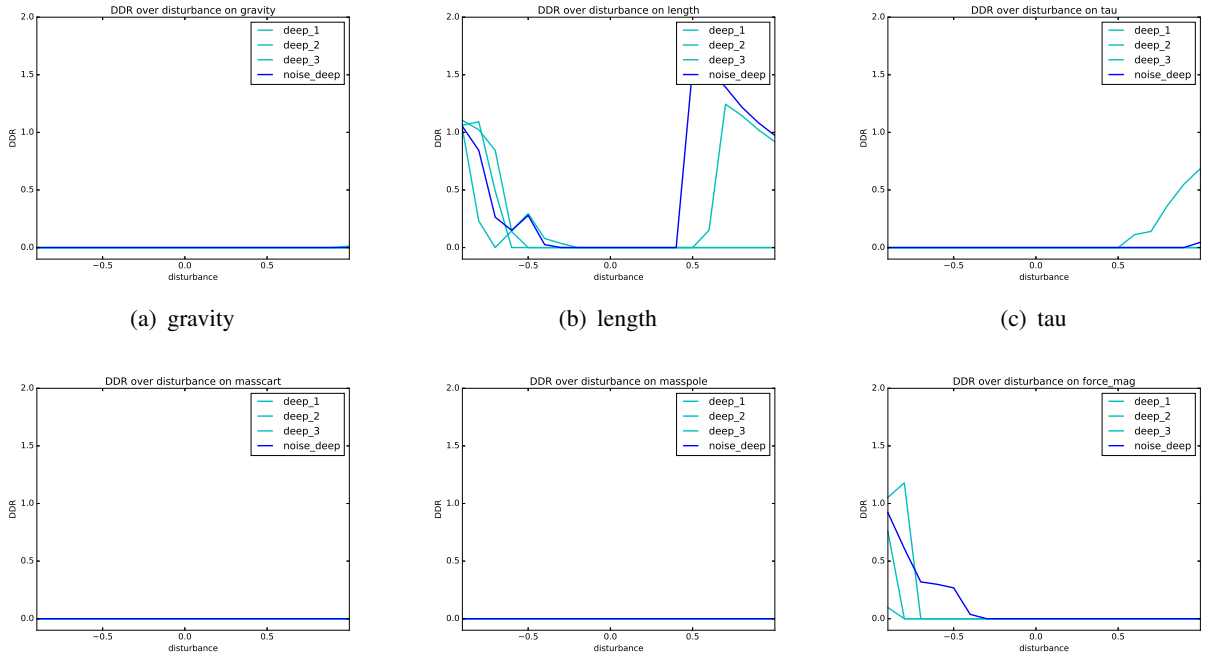
Training with noisy states wasn't a very successful approach, with the deep Q network not being able to train stable controllers. A controller which was trained with 2% noise is shown in figure 8. It does not appear to do any better than controllers trained without noise. Although this was not a successful method to train robust controllers, we believe that this seems to indicate that deep Q learning is susceptible to sensor noise.



(a) gravity (b) length (c) tau

(d) masscart (e) masspole (f) forcemag

Figure 7: DDR results for variations on 6 parameters for bagged deep neural network controllers.



(a) gravity (b) length (c) tau

(d) masscart (e) masspole (f) forcemag

Figure 8: DDR results for variations on 6 parameters for a deep controller which trained with noisy states (dark blue), compared with deep controllers trained without noise (cyan).

VII. Conclusions & Future Work

The experiments in this project seem to indicate the following:

- **Deeper networks are more robust** : The results indicate that deeper neural networks appear to produce controllers that are more robust under identical training conditions.
- **Bagged controllers have improved robustness** : Even though the voting scheme in an ensemble does not have ranking of its constituents, the bagged controllers seemed to be better than the worst of their constituents. This is a promising result motivating further work on the bandit based ensembles.
- **The Pessimizer finds the local worst case plant parameters efficiently** : The pessimizer is able to find the local worst case plant parameters. Further it has been possible to train new controllers on these worst case plants.
- **Training is sensitive to sensor noise** : Training becomes harder with noisy states.

We wish to pursue this project along the following directions:

- **Bandit based ensembles**: Ensembles using the bandit algorithm and pessimal controllers still needs to be explored.
- **Explore policy learning strategy**: Our current Q-learning strategy suffers long training times. We wish to explore alternate training methods such as policy learning to accelerate the training process. This would enable better statistical measurements of robustness, and accelerate development of robust control learners.
- **Explore online pessimal techniques**: The current proposed method of choosing pessimal plants and training controllers in discrete steps appears inefficient. We would like to explore online strategies to do the same. This would potentially produce a single robust controller and obviate the need to ensemble multiple controllers.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[3] "Open ai gym," https://gym.openai.com/.

[4] B. Schölkopf, "Artificial intelligence: Learning to see and act," *Nature*, vol. 518, no. 7540, pp. 486–487, 2015.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[6] R. M. Kretchmara, P. M. Young, C. W. Anderson, D. C. Hittle, M. L. Anderson, and C. Delnero, "Robust reinforcement learning control," in *American Control Conference, 2001. Proceedings of the 2001*, vol. 2. IEEE, 2001, pp. 902–907.

[7] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," *arXiv preprint arXiv:1703.02702*, 2017.

[8] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[9] "Open ai gym," https://en.wikipedia.org/wiki/Hill_climbing.

[10] "Numpy's genetic algorithm," https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.optimize.differential_evolution.html.

[11] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[12] S. Bubeck, N. Cesa-Bianchi *et al.*, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Foundations and Trends® in Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.

[13] "Dqn on flappy bird," https://yanpanlau.github.io/2016/07/10/FlappyBird-Keras.html.