

Robust Reinforcement Learning Control of a Furuta Pendulum

Philip Ohager



LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6150
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2021 by Philip Olhager. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2021

Abstract

The use of Reinforcement Learning (RL) to design controllers for safety critical systems is an important research area. On the one hand, RL can function in and adapt to complex and changing environments without requiring a model of the system. On the other hand, in such systems robustness is of high importance, as well as ways to guarantee and certify a level of robustness. This project investigates state-of-the-art methods for training for and evaluating robustness in a neural network, when applied to RL control of a real-world system. The use of Projected Gradient Descent (PGD) as an adversary for robust Deep RL, as well as the Lipschitz constant as a measure of a neural network controller's robustness, are evaluated.

The study is conducted by training an agent to perform swing-up and balancing of a Furuta pendulum, and further training it with PGD of varying magnitudes as an adversary. The agents are evaluated by their robustness towards normally-distributed measurement noise as well as their estimated Lipschitz constant.

The results show that while training with PGD does result in better robustness for a classifier on the MNIST dataset, applying the technique to the Furuta pendulum in a Deep Reinforcement Learning setting is not so simple. One of 30 agents managed to outperform the baseline agent, indicating that while the technique may have some promise, further fine-tuning of the training process is necessary. Further, the Lipschitz constant did not correlate with robustness performance, indicating that it may not be an ideal measure of a neural network's robustness.

Acknowledgements

I would like to thank Richard Pates and Johan Grönqvist for supervising the project, and suggesting the topic. I would also like to thank Anders Nilsson and Anders Blomdell for their help in interfacing with a physical pendulum, experiments with which could unfortunately not be incorporated in the final report due to time constraints.

Contents

Acronyms	9
1. Introduction	10
1.1 Background	10
1.2 Problem Description	12
2. Theory	13
2.1 Furuta pendulum	13
2.2 Classical controller	15
2.3 Reinforcement Learning controller	17
2.4 Lipschitz constants	23
3. Method	27
3.1 MNIST	27
3.2 Furuta pendulum	28
4. Results	33
4.1 MNIST	33
4.2 Furuta pendulum	36
5. Discussion and Conclusion	39
5.1 MNIST	39
5.2 Furuta pendulum	40
5.3 Conclusions	42
Bibliography	43
A. MOSEK formulation of Lipschitz SDP	45
B. Table of Furuta results	47
C. Furuta pendulum plots	49

Acronyms

CPU Central Processing Unit

DDPG Deep Deterministic Gradient Descent

DQN Deep Q-Network

FGSM Fast Gradient Sign Method

GPU Graphics Processing Unit

LQR Linear Quadratic Regulator

MDP Markov Decision Process

MSE Mean-Square Error

NaN Not a Number

NN (Artificial) Neural Network

PGD Projected Gradient Descent

RAM Random Access Memory

ReLU Rectified Linear Unit

RL Reinforcement Learning

SARS State, Action, Reward, next State

SARTS State, Action, Reward, Terminal, next State

SDP Semi-Definite Program

VRAM Video Random Access Memory

1

Introduction



Figure 1.1: A Furuta pendulum.

1.1 Background

Machine Learning techniques, in particular Reinforcement Learning (RL), have been gaining interest within control applications due to their ability to function in complex and changing environments. Deep Reinforcement Learning is a subset of these techniques which uses neural networks instead of the more traditional lookup tables. Lookup tables store one action for each possible combination of observations, which requires a discrete observation space. Neural networks, on the other hand, allow the methods to handle continuous observations and in some cases continuous actions, which is often the

case in control applications. Model-free deep RL methods, like DQN (Deep Q-Network) [Mnih et al., 2013] and DDPG (Deep Deterministic Policy Gradient) [Lillicrap et al., 2015] are of particular interest since they allow for implementation on systems that are difficult to model or entirely unknown. Due to these models' adaptability, safety-critical systems is an interesting application. In order to function in such applications, the controller must have a certain level of robustness. In particular, the controller should have a certified and guaranteed level of robustness.

There have been recent efforts within the control community to analyze, improve and guarantee levels of robustness for neural network controllers. An important example is [Fazlyab et al., 2019] which presents a method to estimate the Lipschitz constant for a deep neural network. The Lipschitz constant is typically used as a robustness certification metric of sorts within control theory, and [Fazlyab et al., 2019] is a step towards applying this concept to neural networks. One popular method for improving robustness during training of neural networks is the concept of adversarial attacks, providing the network with some form of worst-case perturbations on the inputs during training. [Huang et al., 2017] applied the Fast Gradient Sign Descent (FGSM) method [Goodfellow et al., 2015] to a Deep RL setting. FGSM is a single gradient step with a fixed step size, making it a basic first-order adversary. [Pattanaik et al., 2017] added a variable step-size by sampling noise from a beta distribution. This introduces some stochasticity in the training process, which gave better results. [Oikarinen et al., 2020] augmented the loss function itself to account for worst-case perturbations, moving the robustness training from environment interaction into the network updates. [Madry et al., 2019] proposed the projected gradient descent as the "ultimate" first-order adversary to a neural network and applied it to image classification on the MNIST [LeCun and Cortes, 2010] and CIFAR-10 [Krizhevsky, 2009] datasets. This method proved effective at improving the robustness of the classifiers. It has yet to be tested for Reinforcement Learning schemes, but due to its effectiveness for supervised learning and its theoretical strength as a first-order adversary, it was chosen as the method to be used in this project.

The Furuta pendulum [Furuta et al., 1991], see Figure 1.1, is an extension of the swing-up cartpole experiment, which is a very commonly used benchmark within control literature. It is highly non-linear and offers a challenging environment for the controller to operate in. At the same time, there are methods to extract a highly functional controller using classical theoretical methods, which provides a good comparison for the rl controllers.

1.2 Problem Description

The primary goal of this project is to take the theoretical results of [Fazlyab et al., 2019] and [Madry et al., 2019], which demonstrated that PGD training reduces the Lipschitz constant and improves robustness of neural networks, and investigate how this transfers to an RL controller of a physical system. This will be the first attempt at using PGD for adversarial training on Reinforcement Learning. In doing this, we will investigate its effectiveness at improving the robustness and/or reducing the Lipschitz constant of the controller, as well as the relation between the controller’s Lipschitz constant and its robustness for a practical system. To do this, we will first demonstrate the method on the simpler case of image classification on the MNIST dataset. We will then train a DDPG agent on a simulated Furuta pendulum and augment it with PGD training with varying perturbation bounds. These augmented agents will be analyzed with respect to their robustness to measurement noise in simulation and on a physical process as well as their estimated Lipschitz constants, and compared to the classical controller and the baseline agent.

The questions we seek to answer are these:

- Does PGD training increase robustness to measurement noise in a DDPG controller of a Furuta pendulum?
- Does PGD training reduce the Lipschitz constant of a DDPG controller of a Furuta pendulum?
- Does the Lipschitz constant of a DDPG controller of a Furuta pendulum correlate with its robustness to measurement noise?

2

Theory

In this section, relevant theory concerning the Furuta pendulum, the classical controller, the RL controller and Lipschitz constants, is presented.

2.1 Furuta pendulum

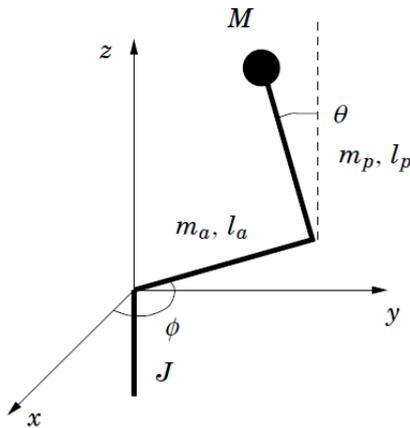


Figure 2.1: The Furuta pendulum. [Gäfvert, 1998]

The Furuta pendulum [Furuta et al., 1991] consists of a pendulum attached to the end of a rotating arm, see Figure 2.1. The pendulum has uniformly distributed mass m_p and length l_p , the arm has uniformly distributed mass m_a and length l_a , and the center pillar has moment of inertia J . At the end of the pendulum is a spherical mass M . The pendulum angle θ is defined such that $\theta = 0$ corresponds to the upright position. The derivation for the equations of motion is given in [Gäfvert, 1998].

Defining

$$\begin{cases} \alpha = J + (M + \frac{1}{3}m_a + m_p) l_a^2, \\ \beta = (M + \frac{1}{3}m_p) l_p^2, \\ \gamma = (M + \frac{1}{2}m_p) l_a l_p, \\ \delta = (M + \frac{1}{2}m_p) g l_p, \end{cases} \quad (2.1)$$

the equations of motion can be written as

$$\begin{cases} v = (\alpha + \beta \sin^2 \theta) \ddot{\phi} + \gamma \cos \theta \ddot{\theta} + 2\beta \cos \theta \sin \theta \dot{\phi} \dot{\theta} - \gamma \sin \theta \dot{\theta}^2, \\ 0 = \gamma \cos \theta \ddot{\phi} + \beta \ddot{\theta} - \beta \cos \theta \sin \theta \dot{\phi}^2 - \delta \sin \theta, \end{cases} \quad (2.2)$$

where v contains the (possibly rescaled) torques and forces applied to the arm joint. Rewriting this to the standard form of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ yields

$$\begin{aligned} \frac{d}{dt} \theta &= \dot{\theta}, \\ \frac{d}{dt} \dot{\theta} &= \frac{1}{\alpha\beta - \gamma + (\beta^2 + \gamma^2) \sin^2 \theta} \left(\beta(\alpha + \beta \sin^2 \theta) \cos \theta \sin \theta \dot{\phi}^2 \right. \\ &\quad \left. + 2\beta\gamma(1 - \sin^2 \theta) \sin \theta \dot{\phi} \dot{\theta} - \gamma^2 \cos \theta \sin \theta \dot{\theta}^2 + \delta(\alpha + \beta \sin^2(\theta)) \sin \theta - \gamma \cos \theta v \right), \\ \frac{d}{dt} \phi &= \dot{\phi}, \\ \frac{d}{dt} \dot{\phi} &= \frac{1}{\alpha\beta - \gamma + (\beta^2 + \gamma^2) \sin^2 \theta} \left(\beta\gamma(\sin^2 \theta - 1) \sin \theta \dot{\phi}^2 \right. \\ &\quad \left. - 2\beta^2 \cos \theta \sin \theta \dot{\phi} \dot{\theta} + \beta\gamma \sin \theta \dot{\theta} - \gamma\delta \cos \theta \sin \theta + \beta v \right). \end{aligned} \quad (2.3)$$

This is the state-space description used in the simulation of the pendulum. Linearizing this system at the equilibrium point $\mathbf{x}_0 = [\theta_0, \dot{\theta}_0, \phi, \dot{\phi}_0]^\top = [0, 0, 0, 0]^\top$ (i.e. upright stationary pendulum) yields the matrices

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{\alpha\delta}{\alpha\beta - \gamma^2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{\delta\gamma}{\alpha\beta - \gamma^2} & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ -\frac{\gamma}{\alpha\beta - \gamma^2} \\ 0 \\ \frac{\beta}{\alpha\beta - \gamma^2} \end{bmatrix}. \quad (2.4)$$

The eigenvalues of this A matrix are

$$\left\{ 0, 0, \pm \sqrt{\frac{\alpha\delta}{\alpha\beta - \gamma^2}} \right\}. \quad (2.5)$$

One can verify that in the limit case $J, m_a \rightarrow \infty$ and $m_p \rightarrow 0$ the open-loop poles for a simple pendulum are restored:

$$\sqrt{\frac{\alpha\delta}{\alpha\beta - \gamma^2}} \rightarrow \sqrt{\frac{g}{l_p}}. \quad (2.6)$$

There exists significant friction in the ϕ -joint. This friction is modelled in the following way:

$$u_F = \begin{cases} \tau_C \operatorname{sgn} \dot{\phi} & \text{if } \dot{\phi} \neq 0, \\ u & \text{if } \dot{\phi} = 0 \text{ and } |u| < \tau_S, \\ \tau_S \operatorname{sgn} u_0 & \text{otherwise,} \end{cases} \quad (2.7)$$

with $v = u - u_F$, u being the nominal control signal, and τ_C and τ_S being coefficients for Coulomb friction and stiction respectively.

2.2 Classical controller

Designing a classical controller as a baseline for the swing-up problem can be broken down into two steps: designing a balancing controller once the pendulum is upright and designing a controller to make the pendulum swing up. Designing a balancing controller can be done using various linear methods, as the pendulum can be accurately linearized for small angle deviations. The method of choice here is the Linear Quadratic Regulator (LQR). Designing a swing-up controller is slightly more involved, as the system cannot be linearized due to large angle deviations. The method used here is commonly referred to as Lyapunov based design or energy shaping control.

LQR

Given a discrete-time LTI (Linear, Time-Invariant) system

$$\mathbf{x}_{k+1} = \Phi \mathbf{x}_k + \Gamma \mathbf{u}_k, \quad (2.8)$$

the LQR method defines a quadratic cost function

$$J(\mathbf{x}, \mathbf{u}) = \sum_{k=0}^T (\mathbf{x}_k^\top \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R} \mathbf{u}_k + 2\mathbf{x}_k^\top \mathbf{N} \mathbf{u}_k), \quad (2.9)$$

where \mathbf{Q} , \mathbf{R} and \mathbf{N} are chosen weight matrices, and formulates finding the control law as an optimization problem:

$$\begin{aligned} \min_{\mathbf{u}} \quad & J(\mathbf{x}, \mathbf{u}) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \Phi \mathbf{x}_k + \Gamma \mathbf{u}_k, \quad k = 0, 1, \dots, T. \end{aligned} \quad (2.10)$$

The solution to this problem is given by $\mathbf{u}_k = -\mathbf{K}_k \mathbf{x}_k$ with \mathbf{K} given recursively by the (discrete-time) Riccati equation:

$$\begin{cases} \mathbf{K}_k = (\mathbf{\Gamma}^\top \mathbf{S}_{k+1} \mathbf{\Gamma} + \mathbf{R})^{-1} (\mathbf{\Phi}^\top \mathbf{S}_{k+1} \mathbf{\Phi} + \mathbf{N})^\top, \\ \mathbf{S}_k = \mathbf{\Phi}^\top \mathbf{S}_{k+1} \mathbf{\Phi} + \mathbf{Q} - (\mathbf{\Phi}^\top \mathbf{S}_{k+1} \mathbf{\Gamma} + \mathbf{N}) (\mathbf{\Gamma}^\top \mathbf{S}_{k+1} \mathbf{\Gamma} + \mathbf{R})^{-1} (\mathbf{\Phi}^\top \mathbf{S}_{k+1} \mathbf{\Gamma} + \mathbf{N})^\top. \end{cases} \quad (2.11)$$

As the time horizon approaches infinity, \mathbf{S}_k and \mathbf{K}_k approach the stationary \mathbf{S} and \mathbf{K} , and are given by the (discrete-time) algebraic Riccati equation:

$$\begin{cases} \mathbf{K} = (\mathbf{\Gamma}^\top \mathbf{S} \mathbf{\Gamma} + \mathbf{R})^{-1} (\mathbf{\Phi}^\top \mathbf{S} \mathbf{\Phi} + \mathbf{N})^\top, \\ \mathbf{S} = \mathbf{\Phi}^\top \mathbf{S} \mathbf{\Phi} + \mathbf{Q} - (\mathbf{\Phi}^\top \mathbf{S} \mathbf{\Gamma} + \mathbf{N}) (\mathbf{\Gamma}^\top \mathbf{S} \mathbf{\Gamma} + \mathbf{R})^{-1} (\mathbf{\Phi}^\top \mathbf{S} \mathbf{\Gamma} + \mathbf{N})^\top. \end{cases} \quad (2.12)$$

To design a controller this way, one simply chooses the matrices \mathbf{Q} , \mathbf{R} and \mathbf{N} , and solves the Riccati equation using some numerical method. The result is an optimal linear state-feedback controller \mathbf{K} , with the control signal given by $\mathbf{u} = -\mathbf{K}\mathbf{x}$.

Lyapunov based design

To design a controller using Lyapunov based design, first consider the Lyapunov stability theorem:

Theorem 2.1. *Given a system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ with an equilibrium point \mathbf{x}^* and a function $V(\mathbf{x})$ defined on Ω , if*

- i) $V(\mathbf{x}) = 0$ for $\mathbf{x} = \mathbf{x}^*$,*
- ii) $V(\mathbf{x}) > 0$ for $\mathbf{x} \neq \mathbf{x}^*$,*
- iii) $\dot{V}(\mathbf{x}) = \nabla V(\mathbf{x})^\top \mathbf{f}(\mathbf{x}) \leq 0$ for all $\mathbf{x} \in \Omega$,*

then \mathbf{x}^ is stable. If instead of (iii) we have*

- iv) $\dot{V}(\mathbf{x}) < 0$ for all $\mathbf{x} \in \Omega : \mathbf{x} \neq \mathbf{x}^*$,*

then \mathbf{x}^ is locally asymptotically stable. If instead of (iii) and (iv) we have*

- v) $\dot{V}(\mathbf{x}) < 0$ for all $\mathbf{x} \in \Omega : \mathbf{x} \neq \mathbf{x}^*$, $\Omega = \mathbb{R}^n$ and $V(\mathbf{x}) \rightarrow \infty$ as $\|\mathbf{x}\|_2 \rightarrow \infty$,*

then \mathbf{x}^ is globally asymptotically stable.*

Applying this theorem is typically done by choosing a candidate Lyapunov function and testing which parts of the theorem the function fulfil. If a function fulfils requirements *i)* and *ii)*, and its time derivative contains the control signal u , one can design this u to achieve $\dot{V} < 0, x \neq x^*$, guaranteeing asymptotic convergence to the desired equilibrium point. To do this for the Furuta

pendulum, approximate the model by a planar pendulum, simplifying the equations of motion to

$$\ddot{\theta} = \frac{m_p g l_p}{J_p} \sin \theta - u \frac{m_p l_p}{J_p} \cos \theta. \quad (2.13)$$

Defining $\omega_0 = \sqrt{\frac{m_p g l_p}{J_p}}$ simplifies this further to

$$\ddot{\theta} = \omega_0^2 \sin \theta - u \frac{\omega_0^2}{g} \cos \theta. \quad (2.14)$$

The total energy (kinetic energy + potential energy) of the simplified pendulum is

$$E = m_p g l_p (\cos \theta - 1) + \frac{J_p}{2} \dot{\theta}^2 = \cos \theta - 1 + \frac{1}{2\omega_0^2} \dot{\theta}^2. \quad (2.15)$$

Taking our candidate Lyapunov function to be $V(\mathbf{x}) = E(\mathbf{x})^2$ with $\mathbf{x} = [\theta, \dot{\theta}]^\top \in \Omega = [(-\pi, \pi), (-\infty, \infty)]$, $\mathbf{x}^* = [0, 0]^\top$ fulfils conditions (i) and (ii) of the Lyapunov stability theorem. Taking the time derivative of V gives

$$\begin{aligned} \dot{V}(\mathbf{x}) &= \nabla V(\mathbf{x})^\top \mathbf{f}(\mathbf{x}) \\ &= 2 \left(\cos \theta - 1 + \frac{1}{2\omega_0^2} \dot{\theta}^2 \right) \begin{bmatrix} -\sin \theta & \frac{\dot{\theta}}{\omega_0^2} \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ \omega_0^2 \left(\sin \theta - \frac{u}{g} \cos \theta \right) \end{bmatrix} \\ &= -\frac{2}{g} u \dot{\theta} \cos \theta \left(\cos \theta - 1 + \frac{\dot{\theta}^2}{2\omega_0^2} \right). \end{aligned} \quad (2.16)$$

We now wish to choose u such that the above quantity is nonpositive for all θ and $\dot{\theta}$. There are many ways to do this, but in this project the expression

$$u = K \frac{\dot{\theta} \cos \theta \left(\cos \theta - 1 + \frac{\dot{\theta}^2}{2\omega_0^2} \right)}{b + \left| \dot{\theta} \cos \theta \left(\cos \theta - 1 + \frac{\dot{\theta}^2}{2\omega_0^2} \right) \right|}, \quad (2.17)$$

where $K, b > 0$ are tunable parameters, was used.

2.3 Reinforcement Learning controller

Reinforcement Learning

A Reinforcement Learning setup consists of an agent and an environment. The agent takes actions based on observations provided by the environment, and the environment returns a reward based on a predetermined reward

function, as well as updated observations. The agent aims to take actions that maximize the rewards. An environment is defined as a Markov Decision Process (MDP), consisting of the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$ where \mathcal{S} is the set of possible states, \mathcal{A} is the set of possible actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ are the state transition probabilities, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function and $\gamma \in [0, 1]$ is the discount factor. An RL algorithm aims to learn a (possibly stochastic) policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ which describes the probabilities of taking each possible action given a state. This policy then governs the actions taken by the agent. The policy is learned by attempting to maximize the long-term discounted reward $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ where t is the current time step and r_i is the reward received at timestep i . This is typically done by defining the action-value function $q_\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ which gives the expected value of G_t given the current state S_t , and action A_t , and using the Bellman equation to compute this function recursively

$$q_\pi(s, a) = \mathbb{E}_\pi[r_t + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]. \quad (2.18)$$

In the most basic form, with discrete observation and action spaces, q can be defined as a lookup table. Constructing this function requires explicit knowledge about the transition probabilities of the environment, which is not always available. A model-free version of this is called Q-learning, which aims to learn a function Q approximating q by interacting with the environment and observing rewards. During interaction with the environment, state transitions are recorded in the form of the tuple (s_i, a_i, r_i, s_{i+1}) , where s_i is the observed state, a_i is the action taken by the actor and r_i is the reward obtained at time i . This format of recording transitions is called SARS. Q then uses the following update formula, based on the Bellman equation:

$$Q^{(k+1)}(s_i, a_i) = Q^{(k)}(s_i, a_i) + \alpha \left(r_i + \gamma \max_a \{ Q^{(k)}(s_{i+1}, a) \} - Q^{(k)}(s_i, a_i) \right), \quad (2.19)$$

where α is a learning rate.

Artificial Neural Networks

An artificial neural network (ANN or NN) is an attempt to mimic the inner workings of the human brain. It consists of a number of "neurons" in different layers connected by weighted "synapses". See Figure 2.2 for an illustration. Each neuron receives the weighted outputs from the previous layers as inputs and applies an "activation function" to the sum of the inputs.

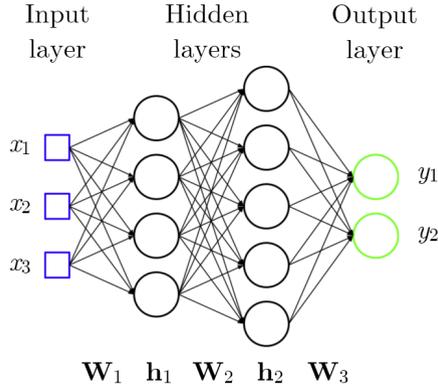


Figure 2.2: A conceptual illustration of a neural network.

Given inputs \mathbf{x} , weight matrices \mathbf{W}_k and activation functions $\varphi_k(\cdot)$, for an NN with n hidden layers the outputs of the 1st hidden layer is $\mathbf{h}_1 = \varphi_1(\mathbf{W}_1\mathbf{x})$, of the k th hidden layer $\mathbf{h}_k = \varphi_k(\mathbf{W}_k\mathbf{h}_{k-1})$ and of the entire network is $\mathbf{y} = \varphi_o(\mathbf{W}_{n+1}\mathbf{h}_n)$. Typically layers share activation functions, possibly except for the output layer. The three activation functions used in this project are ReLU (Rectified Linear Unit), tanh and the linear function $f(x) = x$. ReLU is defined as

$$\varphi(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \quad (2.20)$$

Training a neural network is done by updating the weight matrices using error backpropagation and a specified loss function. The loss function represents some quantity that the NN should minimize. A common example used in supervised learning is the mean-square error (MSE), given by

$$L = \frac{1}{N} \sum_{i=1}^N \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2, \quad (2.21)$$

where \mathbf{y} is the network's output and $\hat{\mathbf{y}}$ is the desired output. To perform the error backpropagation an optimization method, often called an optimizer, is used. In this project, Adam [Kingma and Ba, 2014] is used for baseline training and Optimistic Adam [Daskalakis et al., 2017] is used for adversarial training.

Deep Deterministic Policy Gradient (DDPG)

The full DDPG training scheme is presented in algorithm 1.

The DDPG is an actor-critic method using neural networks as the actor and the critic, specifically designed to handle continuous action spaces

[Lillicrap et al., 2015]. The method utilizes two sets of actor/critic networks, where one set is updated much slower and is called the target network. During training, the agent interacts with the environment through actions taken by the actor. At each time-step the transition is saved in a replay buffer \mathcal{R} of finite size in the form of SARTS tuples: $(s_i, a_i, r_i, t_i, s_{i+1})$, where s_i is the observed state, a_i is the action taken, r_i is the reward, and t_i is 1 if the exploration terminated and 0 if not, at time i . When updating the networks, these tuples, sometimes called experiences or trajectories, are randomly sampled in mini-batches.

The actor network is updated based on the assumption that the output of the value network is continuously differentiable with respect to the actor network's parameters. The expected return $J = \mathbb{E}[Q(s, \mu(s|\theta^\mu)|\theta^Q)]$ is differentiated with respect to θ^μ using the chain rule:

$$\begin{aligned}\nabla_{\theta^\mu} J &= \mathbb{E}[\nabla_{\theta^\mu} Q(s, \mu(s|\theta^\mu)|\theta^Q)] \\ &= \mathbb{E}[\nabla_{\mu} Q(s, \mu(s|\theta^\mu)|\theta^Q) \nabla_{\theta^\mu} \mu(s|\theta^\mu)].\end{aligned}\quad (2.22)$$

Thus the loss function used is

$$L_A^\mu = -\frac{1}{N} \sum_{i=1}^N Q(s_i, \mu(s_i|\theta^\mu)|\theta^Q), \quad (2.23)$$

where N is the number of samples in the mini-batch. The critic network $Q(s, a|\theta^Q)$ is, as in Q-learning, updated using an MSE loss function

$$L_{MSE}^Q = \frac{1}{N} \sum_{i=1}^N (y_i - Q(s_i, a_i|\theta^Q))^2, \quad (2.24)$$

where y_i is given by the Bellman equation:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})(1 - t_i), \quad (2.25)$$

with Q' and μ' being the target critic and target actor respectively. The parameters of the target networks are then updated as $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$, with $0 < \tau \ll 1$.

In order to make the training more efficient (and actually converge at all) a demonstration system was implemented using ideas from [Nair et al., 2017] and [Goecks et al., 2019]. A second replay buffer \mathcal{R}^d , called the demonstration buffer, was initialized with state transitions performed by an "expert", in this case the classical controller. A second loss function was added to the actor:

$$L_{BC}^\mu = \frac{1}{2N_D} \sum_{j=1}^{N_D} (a_j^d - \mu(s_j^d|\theta^\mu))^2 c_j, \quad (2.26)$$

where s_j^d and a_j^d are states and actions from transitions in \mathcal{R}^d , and

$$c_j = \begin{cases} 1 & \text{if } Q(s_j^d, a_j^d) > Q(s_j^d, \mu(s_j^d)), \\ 0 & \text{otherwise.} \end{cases} \quad (2.27)$$

c_j exists to acknowledge that the demonstrations may not be perfect, so if the agent determines that its own action is better then this portion of the loss function will be discarded for that sample. During updating, 25% of the samples are taken from \mathcal{R}^d and 75% from \mathcal{R} . In addition, the agent is pretrained using only expert demonstrations before it gets to explore the environment. In the pretraining step, $c_j = 0$ is used. Weight regularization terms were also added to the loss functions of both actor and critic. The full loss functions used were thus

$$\begin{aligned} \mathcal{L}^\mu &= L_A^\mu + L_{BC}^\mu + L_R^\mu = \\ &\quad - \frac{1}{N} \sum_{i=1}^N Q(s_i, \mu(s_i | \theta^\mu) | \theta^Q) + \frac{1}{2N_D} \sum_{j=1}^{N_D} (a_j^d - \mu(s_j^d | \theta^\mu))^2 c_j + \frac{1}{4} \|\theta^\mu\|_2^2, \\ \mathcal{L}^Q &= L_{MSE}^Q + L_R^Q = \frac{1}{N} \sum_{i=1}^N (y_i - Q(s_i, a_i | \theta^Q))^2 + \frac{1}{2} \|\theta^Q\|_2^2. \end{aligned} \quad (2.28)$$

Projected Gradient Descent

Given a function $f(\mathbf{x})$, the projected gradient descent (PGD) method seeks to minimize f using successive steps on the form

$$\mathbf{x}^{t+1} = \Pi_{\mathbf{x}+\mathcal{S}} (\mathbf{x}^t - \alpha \text{sgn}(\nabla f(\mathbf{x}^t))), \quad (2.29)$$

where $\Pi_{\mathbf{x}+\mathcal{S}}$ signifies projection onto a ball of some vector space centered on x . If f is the loss function of some neural network, PGD can be used as an effective adversary by maximizing f . As presented in [Madry et al., 2019], in the case of supervised learning, the loss function is of the form $f = L(\theta, x, m(x))$, with $m(x)$ being the output of the network m , and the PGD step is

$$\mathbf{x}^{t+1} = \Pi_{\mathbf{x}+\mathcal{S}} (\mathbf{x}^t + \alpha \text{sgn}(\nabla_{\mathbf{x}^t} L(\theta, \mathbf{x}^t, m(\mathbf{x}^t))). \quad (2.30)$$

Here, \mathcal{S} is an l^∞ -ball of some size $\varepsilon > 0$. For Reinforcement Learning, and specifically an actor-critic method, the objective is to perturb the state that the actor is given in such a way that it takes a suboptimal action. The function to be minimized is the critic's evaluation of the actor's action, $Q(s, \mu(s))$.

Algorithm 1: DDPG training scheme with demonstrations

Data: Pre-training steps L , number of episodes M , steps per episode T , batch size N , exploration noise level σ , Polyak averaging constant τ .

Randomly initialize critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$.

Initialize replay buffer \mathcal{R} and demonstration buffer \mathcal{R}^d with trajectories from the expert.

for *pre-training step* = 1 ... L **do**

Randomly sample a minibatch of N transitions

$(s_i^d, a_i^d, r_i^d, t_i^d, s_{i+1}^d)$ from \mathcal{R}^d .

Update critic network Q by minimizing the loss \mathcal{L}^Q .

Update actor network μ by minimizing the loss \mathcal{L}^μ .

Update target networks:

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

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

end

for *episode* = 1 ... M **do**

Reset environment.

Receive initial observation state s_1 .

for $k=1 \dots T$ **do**

Select action $a_k = \mu(s_k|\theta^\mu) + \mathcal{N}_k$ with $\mathcal{N}_k \sim N(0, \sigma^2)$.

Execute action a_k , observe reward r_k , termination status t_k and new state s_{k+1} .

Store transition $(s_k, a_k, r_k, t_k, s_{k+1})$ in \mathcal{R} .

Randomly sample a minibatch of $0.75N$ transitions

$(s_i, a_i, r_i, t_i, s_{i+1})$ from \mathcal{R} and $0.25N$ transitions

$(s_i^d, a_i^d, r_i^d, t_i^d, s_{i+1}^d)$ from \mathcal{R}^d .

Update critic network Q by minimizing the loss \mathcal{L}^Q .

Update actor network μ by minimizing the loss \mathcal{L}^μ .

Update target networks:

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

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

end

end

Given an initial state s^0 , finding an adversarial state to give the actor is done with the following PGD step:

$$s^{k+1} = \Pi_{s^0 + \mathcal{S}} (s^k - \alpha \operatorname{sgn}(\nabla_{s^k} Q'(s^0, \mu(s^k)))) . \quad (2.31)$$

Again, \mathcal{S} is an l^∞ -ball of some size $\varepsilon > 0$. Typically in similar implementations, such as [Pattanaik et al., 2017], the target critic Q' is used instead of the behavior critic Q , which is what is done in this project. It should be noted that in this project, the environment is a pendulum, and observations of the pendulum angle are given in sine and cosine; while the state of the environment is $(\theta, \dot{\theta}, \phi, \dot{\phi})$, the observations given to the agent are on the form $(\sin \theta, \cos \theta, \dot{\theta}, \dot{\phi})$. Since logically the adversary should perturb the angle itself, as opposed to the sine and cosine, the PGD algorithm is performed on the state rather than the observation. The full PGD algorithm for the pendulum is given in algorithm 2.

Algorithm 2: Projected Gradient Descent for the Furuta pendulum

Data: Initial state $x_0 = (\theta_0, \dot{\theta}_0, \phi_0, \dot{\phi}_0)$, actor network μ , target critic network Q' , ε , number of steps n .
Set initial observation base on initial state:
 $s_0 \leftarrow (\sin \theta_0, \cos \theta_0, \dot{\theta}_0, \dot{\phi}_0)$
 $x^1 \leftarrow x_0$
Set step length: $\alpha \leftarrow 2.5\varepsilon/n$
for $k = 1 \dots n$ **do**
 Set current observation based on current state:
 $s^k \leftarrow (\sin \theta^k, \cos \theta^k, \dot{\theta}^k, \dot{\phi}^k)$
 Take one (non-projected) step:
 $x^{k+1} \leftarrow x^k - \alpha \operatorname{sgn}(\nabla_{x^k} Q'(s_0, \mu(s^k | \theta^\mu) | \theta^{Q'}))$
 Project onto l^∞ -ball: $x^{k+1} \leftarrow \operatorname{clamp}(x^{k+1}, x_0 - \varepsilon \mathbf{1}, x_0 + \varepsilon \mathbf{1})$
end

The adversarial training was performed by perturbing the observation given to the actor when it takes actions during exploration. The training scheme using this method is described in algorithm 3.

2.4 Lipschitz constants

A Lipschitz constant L for a function f , defined on Ω , is defined by

$$\inf L > 0 \text{ s.t. } \|f(\mathbf{x}) - f(\mathbf{y})\|_2 \leq L \|\mathbf{x} - \mathbf{y}\|_2, \forall \mathbf{x}, \mathbf{y} \in \Omega. \quad (2.32)$$

In essence, L is a measure of how quickly the function value can vary; it is often called a smoothness constant. In robust control, the Lipschitz constant

Algorithm 3: Adversarial DDPG training using PGD

Data: Trained DDPG agent with networks μ, Q, μ', Q' , number of episodes M , steps per episode T , batch size N , Polyak averaging constant τ, ε , number of PGD steps n .

Initialize replay buffer \mathcal{R} and demonstration buffer \mathcal{R}^d with trajectories from the expert.

for $episode = 1 \dots M$ **do**

 Reset environment.

for $k=1 \dots T$ **do**

 Receive current state x_k .

 Perturb current state: $\tilde{x}_k \leftarrow PGD(x_k, \mu, Q', \varepsilon, n)$.

 Set $\tilde{s}_k = (\sin \tilde{\theta}_k, \cos \tilde{\theta}_k, \tilde{\theta}, \tilde{\phi}, \tilde{\phi})$.

 Select action $a_k = \mu(\tilde{s}_k | \theta^\mu) + \mathcal{N}_k$ with $\mathcal{N}_k \sim N(0, \sigma^2)$.

 Execute action a_k , observe reward r_k , termination status t_k and new state s_{k+1} .

 Store transition $(\tilde{s}_k, a_k, r_k, t_k, s_{k+1})$ in \mathcal{R} .

 Randomly sample a minibatch of $0.75N$ transitions

$(\tilde{s}_i, a_i, r_i, t_i, s_{i+1})$ from \mathcal{R} and $0.25N$ transitions

$(s_i^d, a_i^d, r_i^d, t_i^d, s_{i+1}^d)$ from \mathcal{R}^d .

 Update critic network Q by minimizing the loss \mathcal{L}^Q .

 Update actor network μ by minimizing the loss \mathcal{L}^μ .

 Update target networks:

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

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

end

end

is a tool for estimating how robust a controller is, particularly against measurement noise. The idea is that a controller with a small Lipschitz constant will produce small variations in control output as a response to a varying measurement input, and thus not be affected as much by the noise. [Fazlyab et al., 2019] recently proposed a method to estimate the Lipschitz constant of a neural network. This allows us to in theory evaluate the robustness of a neural network controller. The estimation method is summarized here: Assuming all activation functions in a network are the same, a neural network model can be written compactly as

$$\mathbf{B}\mathbf{x} = \varphi(\mathbf{A}\mathbf{x} + \mathbf{b}) \text{ and } f(x) = \mathbf{C}\mathbf{x} + \mathbf{b}^l, \quad (2.33)$$

where $\mathbf{x} = [\mathbf{x}^{0\top} \mathbf{x}^{1\top} \dots \mathbf{x}^{l\top}]^\top$ is the concatenation of the input and the activation values of the network's layers and the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and \mathbf{b} are given by

$$\mathbf{A} = \begin{bmatrix} \mathbf{W}^0 & 0 & \dots & 0 & 0 \\ 0 & \mathbf{W}^1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & \mathbf{W}^{l-1} & 0 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 0 & \mathbf{I}_{n_1} & 0 & \dots & 0 \\ 0 & 0 & \mathbf{I}_{n_2} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \mathbf{I}_{n_l} \end{bmatrix}, \quad (2.34)$$

$$\mathbf{C} = \begin{bmatrix} 0 & \dots & 0 & \mathbf{W}^l \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{b}^{0\top} & \dots & \mathbf{b}^{l-1\top} \end{bmatrix}^\top.$$

In addition, define the set \mathcal{T}_n as

$$\mathcal{T}_n = \left\{ \mathbf{T} \in \mathbb{S}^n \mid \mathbf{T} = \sum_{t=1}^n \lambda_{ii} \mathbf{e}_i \mathbf{e}_i^\top + \sum_{1 \leq i < j \leq n} \lambda_{ij} (\mathbf{e}_i - \mathbf{e}_j)(\mathbf{e}_i - \mathbf{e}_j)^\top, \lambda_{ij} \geq 0 \right\}, \quad (2.35)$$

where \mathbb{S}^n is the set of n -by- n symmetric matrices, and \mathbf{e}_i are unit basis vectors. With this, the Lipschitz constant of the network can be estimated using the following theorem:

Theorem 2.2. *Consider an l -layer fully connected neural network. Let $n = \sum_{k=1}^l n_k$ be the total number of hidden neurons and suppose the activation functions are slope-restricted in the sector $[\alpha, \beta]$. Define \mathcal{T}_n as in (2.35). Define \mathbf{A} and \mathbf{B} as in (2.34). Consider the matrix inequality*

$$\mathbf{M}(\rho, \mathbf{T}) = \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}^\top \begin{bmatrix} -2\alpha\beta\mathbf{T} & (\alpha + \beta)\mathbf{T} \\ (\alpha + \beta)\mathbf{T} & -2\mathbf{T} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix} + \begin{bmatrix} -\rho\mathbf{I}_{n_0} & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & (\mathbf{W}^l)^\top \mathbf{W}^l \end{bmatrix} \preceq 0. \quad (2.36)$$

If (2.36) is satisfied for some $(\rho, \mathbf{T}) \in \mathbb{R}_+ \times \mathcal{T}_n$, then

$$\|f(\mathbf{x}) - f(\mathbf{y})\|_2 \leq \sqrt{\rho} \|\mathbf{x} - \mathbf{y}\|_2, \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^{n_0}. \quad (2.37)$$

In other words, $\sqrt{\rho}$ is a bound on the network's Lipschitz constant, and ρ is given by solving the SDP (2.36). It should be noted that the ReLU activation function is slope-restricted on $[0, 1]$ which here gives $\alpha = 0, \beta = 1$. The method can be adapted by making assumptions on the set \mathcal{T}_n . As each λ_{ij} is a decision variable, in the base case the variable complexity is $O(n^2)$. [Fazlyab et al., 2019] refers to this method as **LipSDP-Network**, and lists two alternatives.

LipSDP-Neuron ignores cross-coupling of neurons and sets $T = \text{diag}(\lambda_{11}, \dots, \lambda_{nn})$. This gives a variable complexity of $O(n)$.

LipSDP-Layer considers only one constrain per layer, setting $T = \text{blkdiag}(\lambda_1 I_{n_1}, \dots, \lambda_l T_{n_l})$. This gives a variable complexity of $O(l)$.

As solving SDPs typically has a computational complexity scaling with decision variables as $O(n^3)$ in the best case, and $O(n^6)$ in the worst case, keeping variable complexity low is desirable. In this project, the **LipSDP-Neuron** variant is used.

When solving this problem in Julia, it appeared that Julia's front-end interface for optimization, JuMP, was unable to exploit the sparsity in the problem, leading to untenable computation times and memory requirements. Instead, the problem was solved using Mosek's back-end Julia implementation, which required the problem to be formulated on a standard form. This reformulation is presented in Appendix A.

3

Method

In this section the methodology for training and evaluating the classifiers and agents is described.

All code used is available at <https://github.com/polhager/FurutaRL>.

Computational setup

All training and experiments in this project were conducted using a Ryzen 5 3600 CPU (6 cores, 12 threads), 16GB of RAM and a NVIDIA 2070 Super GPU with 8GB of VRAM. Training was performed on the GPU where possible. Julia was the programming language used, and MOSEK was used to solve the Lipschitz SDP.

3.1 MNIST

A neural network with the structure given in Table 3.1 was trained on a set of 60 000 images from the MNIST set. A set of 10 000 images was used as the testing set. The resolution of the images are 28x28, and so the inputs to the network are of size $28^2 = 784$. The output is a vector of length 10 containing the probability that the input image contains each digit (0-9). The network was trained for 10 epochs with a batch size of 16. The network structure was chosen because it is similar to the networks tested in [Fazlyab et al., 2019], and it achieved a high accuracy on the unperturbed test set (0.974).

To visualize the effectiveness of PGD as an adversary compared to white noise, the accuracy of the baseline model was evaluated on the test set perturbed by Gaussian zero-mean noise with standard deviations $\sigma \in [0.0, 0.1]$, as well as by PGD with $\varepsilon \in [0.0, 0.2]$. Since for Gaussian distributions, 95% of samples lie within $\pm 2\sigma$, using $\varepsilon = 2\sigma$ gives roughly the same interval of possible perturbed values, and thus somewhat comparable disturbance levels. It should be noted that Gaussian noise is not used at all during training and only during testing, and PGD is only used during training and not dur-

Table 3.1: Network structure for the classifier used in the MNIST experiment.

layer	neurons	activation function
input	784	-
1	100	ReLU
2	100	ReLU
3	10	Softmax

ing testing. This comparison is only here to get an idea of how potent PGD potentially is.

The baseline model was then subject to different levels of adversarial training. It was trained for another 10 epochs with the training set perturbed by a PGD adversary, with the different values of $\varepsilon \in \{0.01, 0.05, 0.1, 0.15\}$. All classifiers were evaluated by perturbing the test set with Gaussian zero-mean noise with standard deviations $\sigma \in [0.0, 2.0]$. The accuracy of each model for each σ was recorded and plotted against σ . In addition, the Lipschitz constant of each model was estimated.

3.2 Furuta pendulum

Environment

Most of the training and testing was done in a simulated environment. The environment implements the state-space system defined in (2.3) and solves it using the Runge-Kutta method RK3/8, with a sample time of 0.006s. This sample time was chosen since it is the effective sample time of the physical process. Parameter values can be found in Table 3.2. τ_C and τ_S were manually estimated, max input and dt were chosen, and other values were given by [Katz, 2019].

Table 3.2: Parameter values for the Furuta pendulum model.

quantity	value
J [kg·m ²]	0.000154
M [kg]	0.0
m_a [kg]	0.0
m_p [kg]	0.00544
l_a [m]	0.0430
l_p [m]	0.0646
τ_C [Nm]	0.0076
τ_S [Nm]	0.0080
max input [a.u.]	0.04
dt [s]	0.006

The reward function used for the environment was

$$R(\mathbf{x}, u) = -5\theta^2 - 0.05\dot{\theta}^2 - \phi^2 - 0.05\dot{\phi}^2 - 0.05u^2 - 10000(|\phi| > 2\pi). \quad (3.1)$$

The classical controller used was given by

$$u = \begin{cases} \begin{bmatrix} 0.8139 & 0.0541 & 0.0226 & 0.0258 \end{bmatrix} \mathbf{x} + u_F & \text{if } |\theta| < 0.5, |\dot{\theta}| < 5, \\ 0.04 \frac{\dot{\theta} \cos \theta \left(\cos \theta - 1 + \frac{\dot{\theta}^2}{2\omega_0^2} \right)}{10 + \left| \dot{\theta} \cos \theta \left(\cos \theta - 1 + \frac{\dot{\theta}^2}{2\omega_0^2} \right) \right|} & \text{otherwise,} \end{cases} \quad (3.2)$$

with u_F given by equation (2.7). The linear part is an LQR controller with weight matrices

$$\mathbf{Q} = \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}, \quad R = 100, \quad \mathbf{N} = \mathbf{0}, \quad (3.3)$$

and the nonlinear part is simply equation (2.17) with $K = 0.04$, $b = 10$.

Training

The baseline agent was trained on a simulated process using the demonstration implementation described in algorithm 1. The total loss functions used are restated here: for the actor

$$\mathcal{L}^\mu = L_A^\mu + L_{BC}^\mu + L_R^\mu, \quad (3.4)$$

with

$$\begin{cases} L_A^\mu &= -\frac{1}{N} \sum_{i=1}^N Q(s_i, \mu(s_i|\theta^\mu)|\theta^Q), \\ L_{BC}^\mu &= \frac{1}{2N_D} \sum_{j=1}^{N_D} (a_j^d - \mu(s_j^d|\theta^\mu))^2 c_j, \\ L_R^\mu &= \frac{1}{4} \|\theta^\mu\|_2^2, \end{cases} \quad (3.5)$$

and for the critic

$$\mathcal{L}^Q = L_{MSE}^Q + L_R^Q, \quad (3.6)$$

with

$$\begin{cases} L_{MSE}^Q &= \frac{1}{N} \sum_{i=1}^N \left(r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) (1 - t_i) - Q(s_i, a_i|\theta^Q) \right)^2, \\ L_R^Q &= \frac{1}{2} \|\theta^Q\|_2^2. \end{cases} \quad (3.7)$$

The network structures of the actor and critic are given in Tables 3.3 and 3.4 respectively. The network structures were chosen because they are similar to the DQN used in [Wilroth et al., 2020], which successfully performed swing-up of a Furuta pendulum. Training hyperparameters are given in Table 3.5. It should be noted that the exploration noise is the standard deviation of the Gaussian noise applied to the action, and the action is scaled to be limited to $[-1.0, 1.0]$.

Table 3.3: Network structure for the actor used for the Furuta pendulum.

layer	neurons	activation function
input	5	-
1	24	ReLU
2	48	ReLU
3	72	ReLU
4	96	ReLU
5	128	ReLU
6	1	tanh

Table 3.4: Network structure for the critic used for the Furuta pendulum.

layer	neurons	activation function
input	6	-
1	24	ReLU
2	48	ReLU
3	72	ReLU
4	96	ReLU
5	128	ReLU
6	1	linear

Table 3.5: Training hyperparameters for the Furuta agents.

parameter	value
batch size	256
γ	0.99
τ	0.995
exploration noise	0.1
buffer size	100000
expert buffer size	50000

The baseline agent was trained in three steps. First it was pretrained using only expert demonstrations for 35000 steps. Secondly it was trained until it attained an average reward over five episodes of -3500 or greater. Each episode was terminated if $|\phi|$ exceeded 2π , the pendulum had $|\theta| \leq 0.05$, $|\dot{\theta}| \leq 0.5$, or after five seconds. This effectively served to teach the agent to swing up and catch the pendulum. Finally it was trained the same way as in step two, except the episodes only terminated if $|\phi| \geq 2\pi$ or after five seconds. This trained the agent to swing up, catch and balance the pendulum. Dividing the final two steps like this was found to be more effective.

The baseline agent was further trained using PGD according to algorithm 3. For each $\varepsilon \in \{0.025, 0.05, 0.075, 0.1, 0.125, 0.15\}$, five agents were trained until their average reward over five episodes exceeded -10000. Agents that failed to swing-up or balance an unperturbed pendulum, or exceeded 400 episodes, were discarded and retrained.

Testing and evaluation

The agents were tested using the following test: an agent performs one episode of swing-up and balancing on the environment. After half the episode, if the pendulum angle drops below $\pi/2$ in either direction, the test is failed. If not, the test is successful. During the episode, the states given to the agent are perturbed with Gaussian zero-mean noise with standard deviations $\sigma \in [0.0, 0.3]$. This test is carried out 30 times for each agent at each noise level, with the number of successful tests recorded. Results are then collected for all agents of the same level of PGD-training, and the fraction of successful test of each training level is plotted against σ . In addition, the Lipschitz constants of each actor network was estimated. The means and standard deviations of the constants of the five agents for each PGD-level were calculated.

4

Results

In this section results from the experiments are presented. First the effect of PGD training on the MNIST classifier is shown, followed by corresponding results for the RL agents on the Furuta pendulum. Results consist of a comparison of the strength of PGD and Gaussian noise, the effect of PGD training on robustness performance, as well as a comparison between robustness and Lipschitz constant.

Certain numerical problems were encountered in this project, affecting some of the results. For MNIST, running the PGD algorithm against the baseline model with $\varepsilon \geq 0.35$ or attempting PGD training with $\varepsilon \geq 0.16$, Julia reported a NaN error. This occurred when running in a Jupyter notebook, but not in the Atom IDE.

For the Furuta pendulum, running the PGD algorithm against the baseline agent with $\varepsilon \geq 0.34$ resulted in the algorithm returning NaN states at some point, which interestingly caused the pendulum's angle measurements to be set to either 0 or approximately 1.347. In addition, during PGD-training of an agent with $\varepsilon \geq 0.125$ a NaN would appear somewhere, causing every weight in the neural networks to be set to NaN, ruining this agent. This became more frequent with higher ε , and training an agent with $\varepsilon \geq 0.175$ was impossible for this reason.

4.1 MNIST

Figure 4.1 shows the accuracy of the baseline classifier on the test set perturbed by PGD with $\varepsilon \in [0.0, 0.2]$ and white noise with $\sigma \in [0.0, 0.1]$. The accuracy at $\varepsilon = 0.2$ was 0.0041, and the accuracy at $\sigma = 0.1$ was 0.97.

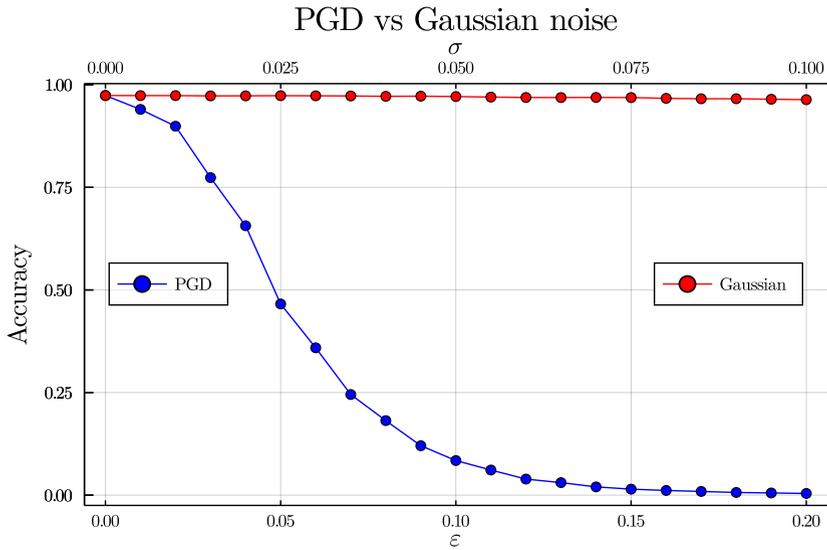


Figure 4.1: The accuracy of the baseline classifier when perturbed by PGD (blue) and white noise (red).

Figure 4.2 shows one sample of the MNIST data set together with the same sample perturbed by Gaussian zero-mean noise with standard deviation $\sigma = 0.1$, and by PGD against the baseline classifier with $\epsilon = 0.2$. The baseline model classified the unperturbed and the Gaussian noise samples as 5, and the PGD sample as 3.

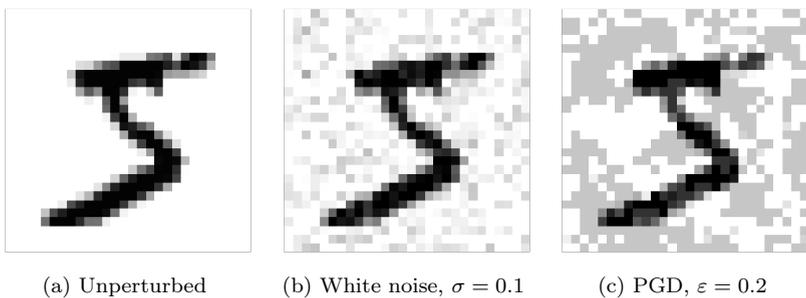


Figure 4.2: A sample from the MNIST dataset, with different perturbations. The baseline model classifies (a) and (b) as 5, and (c) as 3.

Figure 4.3 shows the accuracy of the baseline MNIST classifier as well as

the PGD-trained ones when tested on the test set perturbed by zero-mean Gaussian noise with varying standard deviations.

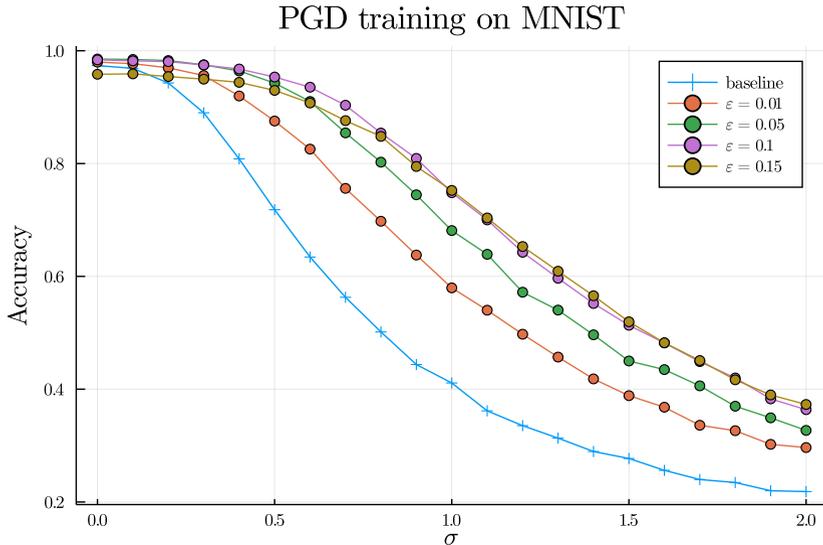


Figure 4.3: The accuracy of the MNIST models on the test set perturbed by white noise.

Table 4.1 shows the estimated Lipschitz constants for the MNIST classifier networks, as well as a performance metric, which is simply the mean accuracy of each model in Figure 4.3.

Table 4.1: Lipschitz constants of MNIST classifiers with different levels of PGD training.

PGD level	Lipschitz constant	performance
baseline	279.88	0.5050
0.01	233.10	0.6239
0.05	81.09	0.6865
0.10	138.91	0.7242
0.15	223.34	0.7166

4.2 Furuta pendulum

Figure 4.4 shows the success rate of the baseline Furuta agent when fed states perturbed by zero-mean Gaussian noise with $\sigma \in [0, 0.4]$ and PGD with $\varepsilon \in [0, 0.8]$. The success rate is the average over 30 experiments. For all PGD data points marked with crosses, the PGD algorithm returned NaN at some point. As previously mentioned, this caused the pendulum angle measurements to be set to 0 or 1.347, which would pass the test. For this reason, crossed out points that show successful tests (1) should be considered false positives, and those showing failed tests (0) had already failed before the NaN appeared and should be considered true negatives.

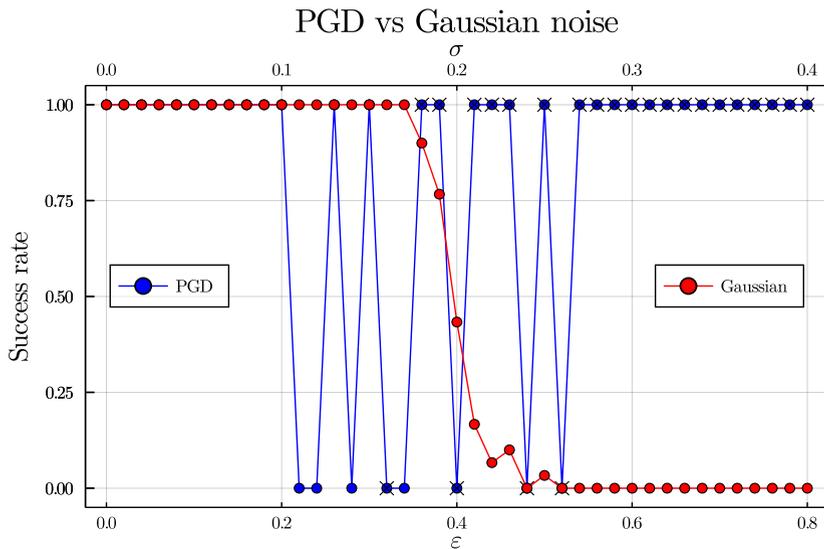


Figure 4.4: Success rate of the baseline Furuta agent when perturbed by PGD and white noise.

Figure 4.5 shows the average success rates of the five PGD-trained agents at each PGD level when performing the robustness test, plotted against the noise level. Also shown are the success rates for the classical controller and the baseline agent. Figure 4.6 shows the success rates of the best performing PGD-trained agents at each level, as well as the classical controller and the baseline agent.

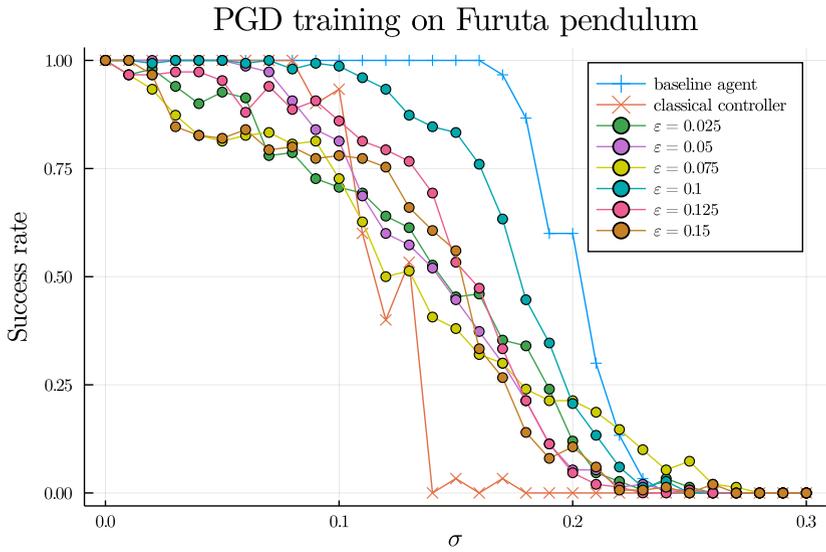


Figure 4.5: Average success rates of the PGD-trained agents, along with the baseline agent and the classical controller.

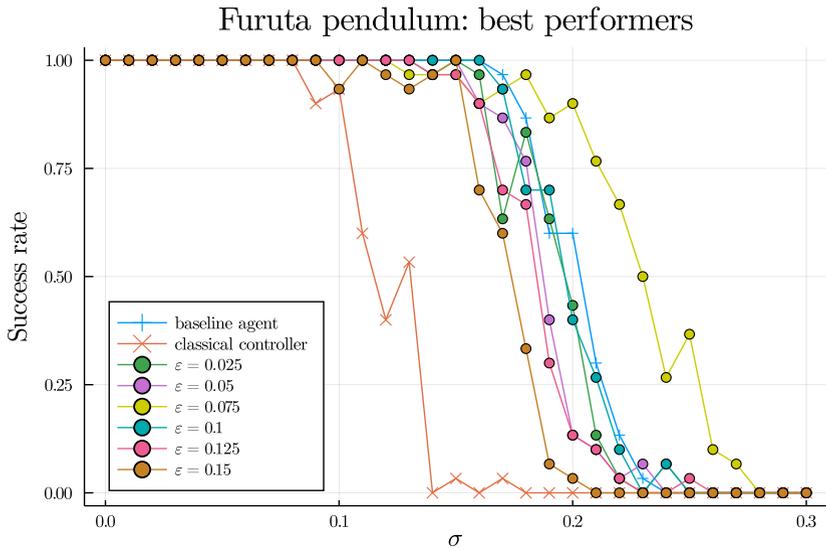


Figure 4.6: Success rates of the best performing PGD-trained agents, along with the baseline agent and the classical controller.

Figure 4.7 shows the average success rates of each agent plotted against their Lipschitz constants. The average success rate here means the average value of curves like the ones plotted in Figures 4.5 and 4.6. A table containing all values along with means and standard deviations per PGD level can be found in Appendix B.

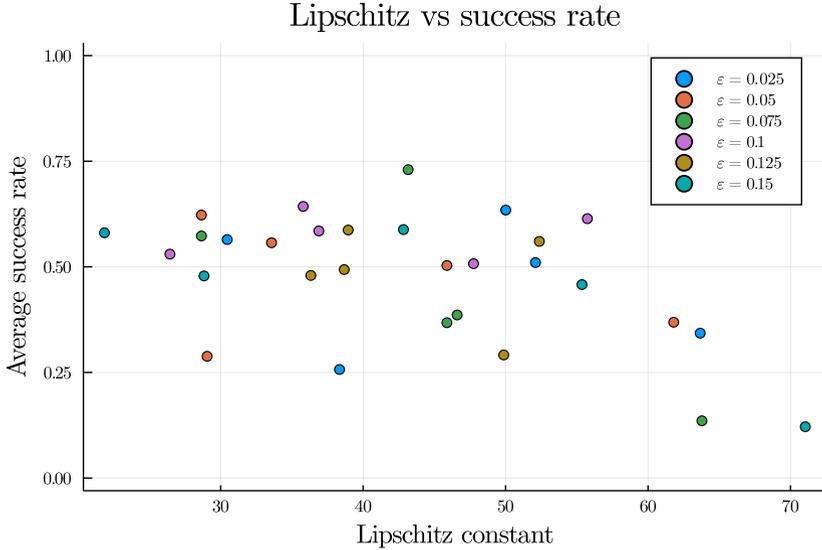


Figure 4.7: The relation between the Lipschitz constants and success rates of every PGD-trained Furuta agent.

Appendix C contains plots from simulations of the Furuta pendulum, with various controllers and adversaries.

5

Discussion and Conclusion

5.1 MNIST

Figures 4.1 and 4.2 clearly demonstrate the potency of PGD as an adversary when applied to the MNIST dataset. The accuracy of the model on adversarial samples deteriorates very quickly when exposed to PGD, and barely at all the subject to similar levels of white noise, as seen in Figure 4.1. Figure 4.2 shows a typical adversarial example: the PGD disturbed sample is clearly still a 5 to a human eye, but the model classifies it as a 3. We also see that the PGD adversary pushes most of the pixels to their extreme permitted values; most of the background pixels are grey, which corresponds to the PGD algorithm changing their values from 0 to 0.2, which is the upper limit given $\varepsilon = 0.2$. This makes sense, and indicates that the loss function has a maximum where most pixels are perturbed by more than 0.2. The white noise example, as expected, has more randomly distributed perturbations. This tells us that while $\varepsilon = 2\sigma$ guarantees that 95% of the random disturbances fall within the same interval as the PGD disturbances, the PGD adversary likely has a higher average disturbance, which means that $\varepsilon = 2\sigma$ might not be an entirely fair comparison of disturbance levels. Constructing a more fair comparison would however be difficult and ultimately not very useful, since the conclusions will be based on using PGD for robustness training and Gaussian noise for evaluation, which means there is no real need for a comparison between the two disturbance models.

The PGD training does improve model robustness to white noise, as shown in Figure 4.3. The baseline model performed the worst, and models with higher PGD levels performed better, except for $\varepsilon = 0.15$, which performed similarly (or slightly worse, as shown in Table 4.1) to $\varepsilon = 0.1$. This indicates that there is an upper limit to the adversarial training, beyond which returns diminish or even deteriorate.

What is interesting is the fact that higher model robustness does not seem to correlate with lower Lipschitz constants, as claimed in [Fazlyab et al., 2019]. Table 4.1 shows that while the baseline model does have the worst

performance and the highest Lipschitz constant, the 0.05 model has the lowest Lipschitz constant but not the best performance, and the 0.10 model has the best performance but a higher Lipschitz constant. Also, 0.10 and 0.15 show similar performances but greatly differing constants. This indicates that this Lipschitz constant might not be an ideal representation of robustness for neural networks. One reason for this is that the input values are limited; for the MNIST dataset, all pixel values are between 0 and 1. On the other hand, the Lipschitz estimation considers any possible input values which can technically be fed to the network, but do not appear in any data even actually given to the network. A better metric might be given by a "local" estimation of the Lipschitz constant which considers a restricted input space. It is interesting to note that [Fazlyab et al., 2019] shows decreasing Lipschitz constants for the baseline model, $\varepsilon = 0.01$ and $\varepsilon = 0.05$ in a similar test, which is corroborated by our results. The trend does not continue however.

5.2 Furuta pendulum

Attempting to compare the strengths of PGD and Gaussian noise as adversaries in the same way as for MNIST resulted in Figure 4.4, which is not as simple to interpret. Firstly, the numerical issues with NaNs appearing made much of the PGD data unreliable. As previously discussed, this NaN phenomenon caused the pendulum angle to be set to either 0 or 1.347, which would evaluate as passing the test. This is why most of the crossed out points show passed tests. The ones that show failed tests simply mean that the agent failed the test before the NaN appeared. Secondly, the PGD results were binary, showing either 30 successful tests or 30 failed ones. This is explained by the fact that the PGD algorithm, the DDPG agent and the Furuta pendulum environment are all deterministic. Without artificially introducing randomness, this means that for a certain agent, a certain ε and a certain initial pendulum angle, the result will always be identical.

Using $\varepsilon = 2\sigma$ as roughly equivalent noise levels, PGD impacts the success rate of the agent at a lower threshold than Gaussian noise, lending some credence to the idea that it is a potent adversary. Drawing any conclusions beyond this is difficult however.

The effectiveness of the PGD training did not meet expectations. Figure 4.5 shows that the average effectiveness of the five agents at each PGD level was worse than the baseline agent at every level. This can be attributed to a number of things, but likely the chief culprit is the inherent instability of Reinforcement Learning. Even training the baseline agent, the agent's performance would wildly peak and trough during training, often losing and regaining the ability to balance the pendulum multiple times. For this reason, choosing when to halt training was not trivial. The baseline agent was

stopped when it reached an average reward over five episodes of -3500 or higher. Most agents during PGD training, at all ε , would simply never reach this value. The PGD adversary works by fooling the agent into reducing its returned reward, so it is natural that this would cause agents to reach lower best rewards. Simply training the agents for a certain number of episodes was also not feasible, since most of the time this would halt the agent at a point where it could not perform balancing even for an unperturbed pendulum, so the agent would have to be discarded. Since training took around 45 minutes per 100 episodes, this was simply too time inefficient, and also did not guarantee better agents. Finding the best stopping criterion could be a subject of further research, and would likely heavily contribute towards a more effective training process.

Another factor that potentially contributes is the actual effectiveness of the PGD algorithm as an adversary. As previously shown, evaluating the algorithm’s effectiveness proved quite difficult. It also depends heavily on the accuracy of the agent’s critic network, which itself changes during training. It is possible, though difficult to demonstrate, that since the actor, critic and adversary all depend on each other during training, they derail and start behaving strangely. This further plays into the instability of RL training.

Looking at Figure 4.6, we see that the best performing agents come close to the baseline agent, with one even outperforming it. It is difficult to say whether this is due to the PGD training or simply luck given the instability of the training process. This result does however indicate that there may be some promise here, and further testing, hyperparameter tuning and the like may lead to better reproducible results. This could be an area of further research.

Ignoring the difference between the PGD agents and the baseline, the PGD agents do not seem to improve with higher ε . Aside from $\varepsilon = 0.1$ performing slightly better on average, and one agent at $\varepsilon = 0.075$ significantly outperforming every other agent, all PGD levels are very close with no apparent order. This further supports the notion that PGD training did not have a noticeable effect on robustness. Again, however, we cannot discard the possibility that it could with further investigation and fine-tuning.

Figure 4.7 reveals no apparent correlation between lower Lipschitz constants and higher performance levels, contrary to expectation. There could possibly be a correlation between lower Lipschitz constants and worse performance, however that is neither very clear nor what we are interested in. This further reinforces the notion that the Lipschitz constant is not an ideal measure of neural network robustness, for the same reasons as previously discussed. Since observations contain sine and cosine functions which are limited to $[-1, 1]$, as well as angle velocities which have a practical upper bound, the input space is restricted, similar to the MNIST case. Again, a better measure might be one that only considers this limited input space. Designing such a

measure is an interesting topic for future research.

5.3 Conclusions

PGD training did positively influence the robustness of the MNIST classifier, although neither PGD training nor robustness seem to correlate with the Lipschitz constants, contrary to the claims in [Fazlyab et al., 2019]. For the Furuta pendulum, there is no evidence that PGD training positively affected the robustness or the Lipschitz constants of the RL controllers, although we cannot discard the possibility that it could, given a more finely tuned training process. There is also no evidence of a correlation between robustness and the Lipschitz constant. In summary, expectations are mostly met for the simple MNIST case, but more investigation is needed for the more complicated RL case.

Possible future work includes designing a more local robustness measure, fine-tuning the adversarial training process and exploring the impact of the agents' network structures. One could also investigate other types of robustness, as this project only looked at robustness towards white measurement noise.

Bibliography

- Daskalakis, C., A. Ilyas, V. Syrgkanis, and H. Zeng (2017). “Training gans with optimism”. *CoRR* **abs/1711.00141**.
- Fazlyab, M., A. Robey, H. Hassani, M. Morari, and G. J. Pappas (2019). “Efficient and accurate estimation of lipschitz constants for deep neural networks”. *CoRR* **abs/1906.04893**.
- Furuta, K., M. Yamakita, S. Kobayashi, and M. Nishimura (1991). “A new inverted pendulum apparatus for education”. In: KHEIR, N. et al. (Eds.). *Advances in Control Education 1991*. Pergamon, Amsterdam, pp. 133–138.
- Gäfvert, M. (1998). *Modelling the Furuta Pendulum*. Tech. rep. Technical Reports TFRT-7574. Lund University, Department of Automatic Control.
- Goecks, V. G., G. M. Gremillion, V. J. Lawhern, J. Valasek, and N. R. Waytowich (2019). “Integrating behavior cloning and reinforcement learning for improved performance in sparse reward environments”. *CoRR* **abs/1910.04281**.
- Goodfellow, I. J., J. Shlens, and C. Szegedy (2015). “Explaining and harnessing adversarial examples”.
- Huang, S. H., N. Papernot, I. J. Goodfellow, Y. Duan, and P. Abbeel (2017). “Adversarial attacks on neural network policies”. *CoRR* **abs/1702.02284**.
- Katz, B. (2019). *Furuta pendulum*. Unpublished.
- Kingma, D. P. and J. Ba (2014). “Adam: a method for stochastic optimization”. *CoRR* **abs/1412.6980**.
- Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Tech. rep.
- LeCun, Y. and C. Cortes (2010). “MNIST handwritten digit database”.
- Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2015). “Continuous control with deep reinforcement learning”.

Bibliography

- Madry, A., A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu (2019). “Towards deep learning models resistant to adversarial attacks”.
- Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller (2013). “Playing atari with deep reinforcement learning”. *CoRR* **abs/1312.5602**.
- MOSEK (2021). *Mosek modeling cookbook*. URL: <https://docs.mosek.com/MOSEKModelingCookbook-letter.pdf>.
- Nair, A., B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel (2017). “Overcoming exploration in reinforcement learning with demonstrations”. *CoRR* **abs/1709.10089**.
- Oikarinen, T. P., T.-W. Weng, and L. Daniel (2020). “Robust deep reinforcement learning through adversarial loss”. *CoRR* **abs/2008.01976**.
- Pattanaik, A., Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary (2017). “Robust deep reinforcement learning with adversarial attacks”. *CoRR* **abs/1712.03632**.
- Wilroth, J., J. Hansson, and L. Hassbring (2020). *Regulating the furuta pendulum*. Course project report at LTH.

A

MOSEK formulation of Lipschitz SDP

The Julia implementation of MOSEK takes SDPs on the form

$$\begin{aligned} \min \quad & \langle \bar{\mathbf{C}}_1, \mathbf{Z}_1 \rangle + \dots + \langle \bar{\mathbf{C}}_m, \mathbf{Z}_m \rangle + \mathbf{c}^\top \mathbf{z} \\ \text{s.t.} \quad & \begin{bmatrix} \langle \bar{\mathbf{a}}_1, \mathbf{Z} \rangle \\ \vdots \\ \langle \bar{\mathbf{a}}_n, \mathbf{Z} \rangle \end{bmatrix} + \mathbf{A}\mathbf{z} = \mathbf{b}, \\ & \mathbf{Z} \succeq \mathbf{0}. \end{aligned} \tag{A.1}$$

We will aim to rewrite (2.36) on this form. With $\alpha = 0, \beta = 1$ we get

$$\begin{aligned} \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}^\top \begin{bmatrix} -2\alpha\beta\mathbf{T} & (\alpha + \beta)\mathbf{T} \\ (\alpha + \beta)\mathbf{T} & -2\mathbf{T} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix} &= \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{T} \\ \mathbf{T} & -2\mathbf{T} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix} \\ &= \mathbf{B}^\top \mathbf{T} \mathbf{A} + \mathbf{A}^\top \mathbf{T} \mathbf{B} - 2\mathbf{B}^\top \mathbf{T} \mathbf{B}. \end{aligned} \tag{A.2}$$

Assuming $\mathbf{T} = \text{diag}\{\lambda_i\}_{i=1, \dots, n}$, we can decompose this matrix into

$$\mathbf{B}^\top \mathbf{T} \mathbf{A} + \mathbf{A}^\top \mathbf{T} \mathbf{B} - 2\mathbf{B}^\top \mathbf{T} \mathbf{B} = \sum_{i=1}^n \mathbf{V}_i \lambda_i, \tag{A.3}$$

where each \mathbf{V}_i is determined by setting $\lambda_i = 1$ and $\lambda_j = 0, i \neq j$ in \mathbf{T} . Additionally define

$$\mathbf{V}_0 = \begin{bmatrix} -\mathbf{I}_{n_0} & 0 & \dots \\ 0 & 0 & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}. \tag{A.4}$$

Now define $\mathbf{x} = [\rho, \lambda_1, \dots, \lambda_n]^\top$, and

$$\mathbf{W} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & (\mathbf{W}^l)^\top \mathbf{W}^l \end{bmatrix}. \quad (\text{A.5})$$

$\mathbf{M}(\rho, \mathbf{T})$ can now be written as

$$\mathbf{M}(\rho, \mathbf{T}) = \mathbf{W} + \sum_{i=1}^n \mathbf{V}_i x_i. \quad (\text{A.6})$$

The constraint $\mathbf{M}(\rho, \mathbf{T}) \preceq 0 \iff -\mathbf{M}(\rho, \mathbf{T}) \succeq 0$ can now be expressed as

$$\mathbf{A}_0 + \sum_{i=1}^{n+1} \mathbf{A}_i x_i \succeq 0, \quad (\text{A.7})$$

where $\mathbf{A}_0 = -\mathbf{W}$, $\mathbf{A}_i = -\mathbf{V}_{i-1}$, $i > 0$. This lets us write the optimization problem as

$$\begin{aligned} \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}_0 + \sum_{i=1}^{n+1} \mathbf{A}_i x_i = \mathbf{X}, \\ & \mathbf{X} \succeq 0. \end{aligned} \quad (\text{A.8})$$

The dual to this problem is [MOSEK, 2021]

$$\begin{aligned} \max \quad & -\langle \mathbf{A}_0, \mathbf{Z} \rangle \\ \text{s.t.} \quad & \langle \mathbf{A}_i, \mathbf{Z} \rangle = c_i, \quad i = 1, \dots, n+1, \\ & \mathbf{Z} \succeq \mathbf{0}, \end{aligned} \quad (\text{A.9})$$

which is on the desired form. The value of ρ , which is what is interesting, is simply given by the first element of $-\langle \mathbf{A}_0, \mathbf{Z} \rangle$, since the primal and dual have the same objective value at optimality. Reformulating (A.9) with our original variable names gives

$$\begin{aligned} \max \quad & \langle \mathbf{W}, \mathbf{Z} \rangle \\ \text{s.t.} \quad & -\langle \mathbf{V}_{i-1}, \mathbf{Z} \rangle = c_i, \quad i = 1, \dots, n+1, \\ & \mathbf{Z} \succeq \mathbf{0}. \end{aligned} \quad (\text{A.10})$$

B

Table of Furuta results

This Figure contains the Lipschitz constants and average success rates of every PGD-trained Furuta agent. Means and standard deviations for each PGD level are also given. The best (lowest) Lipschitz constant and best (highest) performance at each level is highlighted.

Table B.1: Lipschitz constants and performance scores for PGD-trained Furuta agents.

	Lipschitz	mean	std	performance	mean	std
baseline	36.10			0.6527		
$\varepsilon = 0.025$						
1	52.10			0.5101		
2	50.01			0.6344		
3	30.46	46.91	12.86	0.5645	0.4619	0.1571
4	38.34			0.2570		
5	63.66			0.3430		
$\varepsilon = 0.050$						
1	29.05			0.2882		
2	33.57			0.5570		
3	61.80	39.79	14.14	0.3688	0.4680	0.1371
4	45.88			0.5032		
5	28.65			0.6226		
$\varepsilon = 0.075$						
1	46.60			0.3860		
2	43.16			0.7301		
3	63.78	54.59	9.15	0.1355	0.4385	0.2251
4	45.88			0.3677		

Appendix B. Table of Furuta results

$\varepsilon = 0.100$	5	28.65			0.5731		
	1	36.89			0.5849		
	2	35.79			0.6430		
	3	55.74	40.52	11.38	0.6140	0.5759	0.0566
	4	47.75			0.5075		
	5	26.44			0.5301		
$\varepsilon = 0.125$	1	52.37			0.5602		
	2	38.95			0.5871		
	3	49.87	43.24	7.32	0.2914	0.4824	0.1158
	4	38.67			0.4935		
	5	36.33			0.4796		
	$\varepsilon = 0.150$	1	42.83			0.5882	
2		21.84			0.5806		
3		71.04	43.98	19.89	0.1215	0.4430	0.1927
4		55.36			0.4580		
5		28.83			0.4785		

C

Furuta pendulum plots

The following figures show sample simulations of the Furuta pendulum controlled by the classical controller, baseline agent, $\varepsilon = 0.075$ agent 2 (the best performing agent) and $\varepsilon = 0.15$ agent 3 (the worst performing agent), with various adversaries.

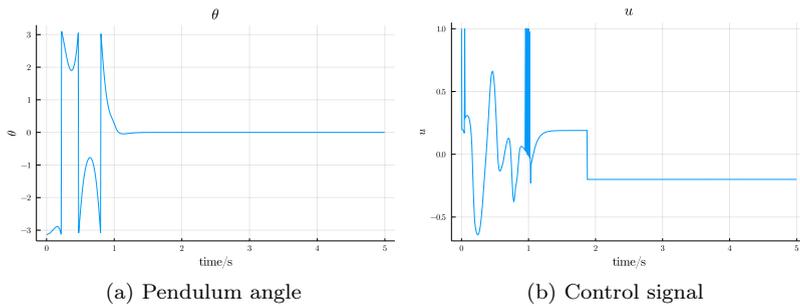


Figure C.1: The classical controller with unperturbed states.

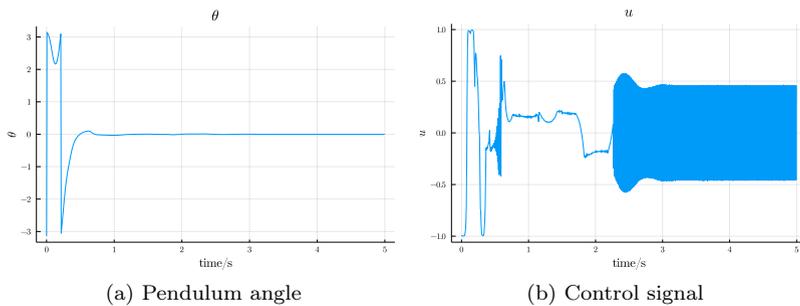
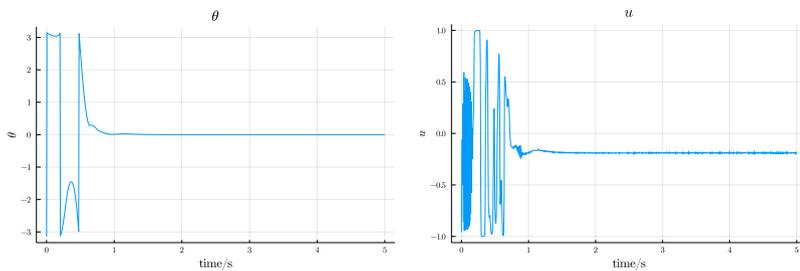


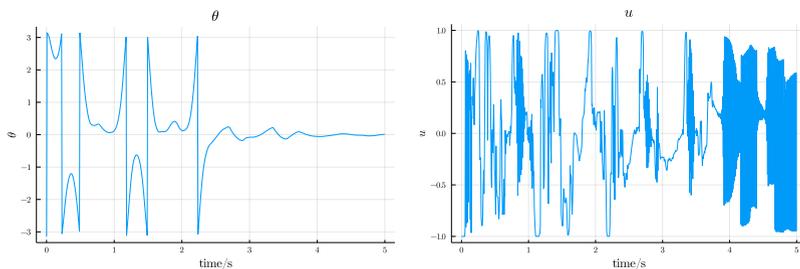
Figure C.2: The baseline agent with unperturbed states.



(a) Pendulum angle

(b) Control signal

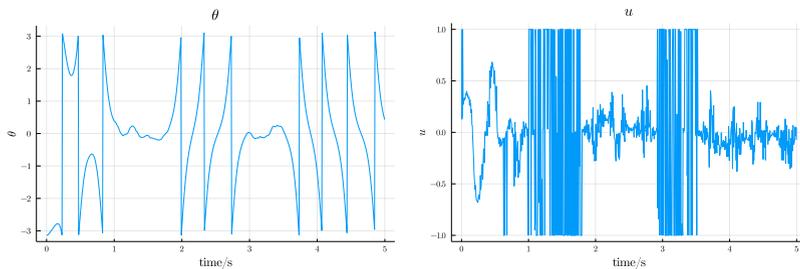
Figure C.3: The $\varepsilon = 0.075$ agent 2 with unperturbed states.



(a) Pendulum angle

(b) Control signal

Figure C.4: The $\varepsilon = 0.15$ agent 3 with unperturbed states.



(a) Pendulum angle

(b) Control signal

Figure C.5: The classical controller with states perturbed by Gaussian noise, $\sigma = 0.2$.

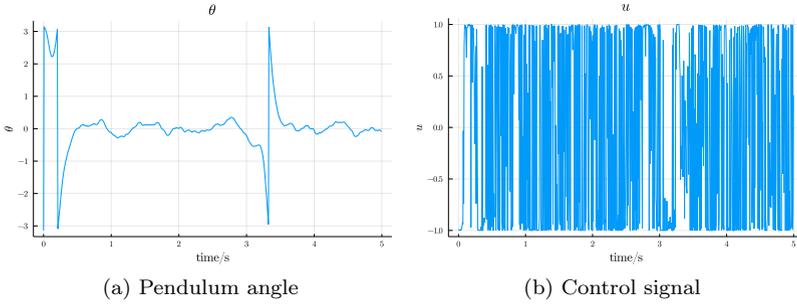


Figure C.6: The baseline agent with states perturbed by Gaussian noise, $\sigma = 0.2$.

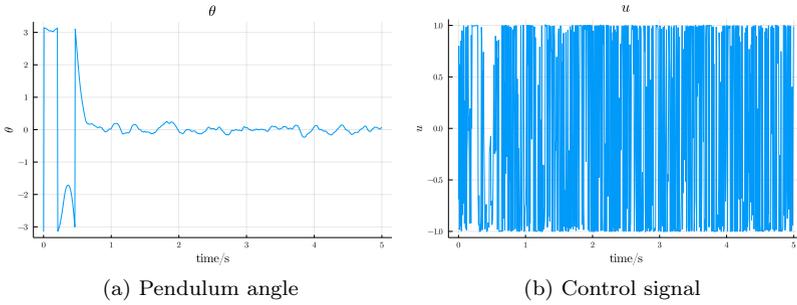


Figure C.7: The $\varepsilon = 0.075$ agent 2 with states perturbed by Gaussian noise, $\sigma = 0.2$.

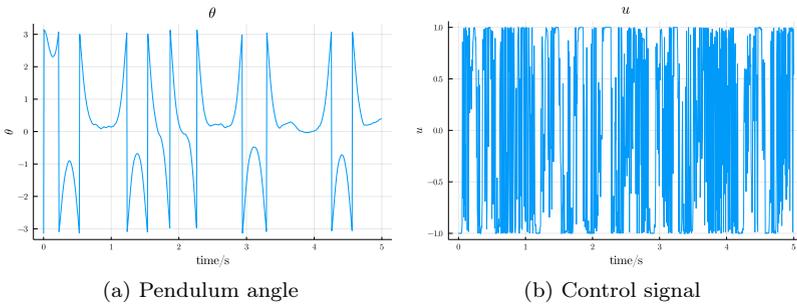


Figure C.8: The $\varepsilon = 0.15$ agent 3 with states perturbed by Gaussian noise, $\sigma = 0.2$.

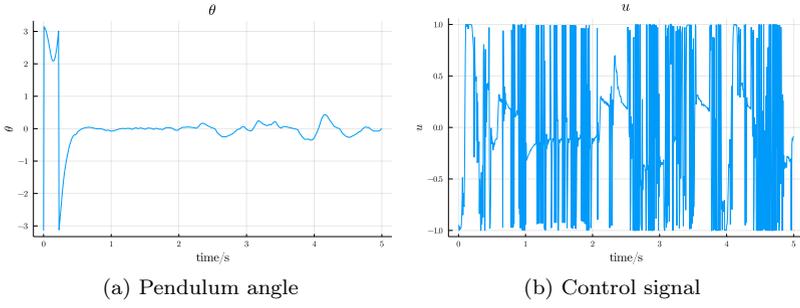


Figure C.9: The baseline agent with states perturbed by PGD, $\varepsilon = 0.2$.

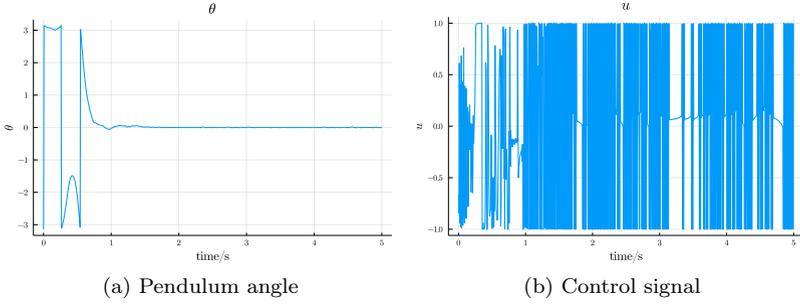


Figure C.10: The $\varepsilon = 0.075$ agent 2 with states perturbed by PGD, $\varepsilon = 0.2$.

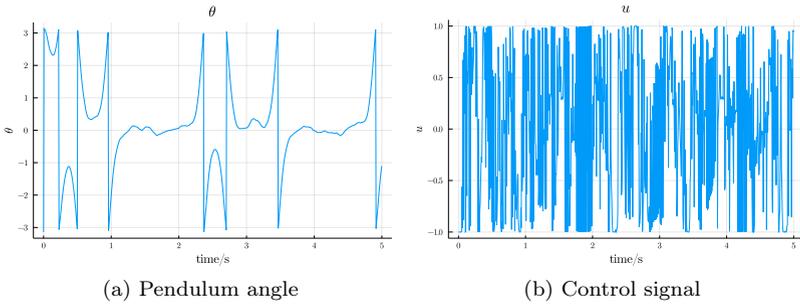


Figure C.11: The $\varepsilon = 0.15$ agent 3 with states perturbed by PGD, $\varepsilon = 0.2$.

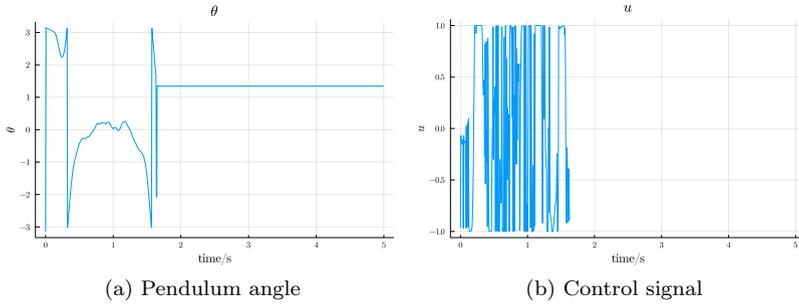


Figure C.12: The baseline agent with states perturbed by PGD, $\varepsilon = 0.36$. After approx. 1.7 seconds a NaN appears.

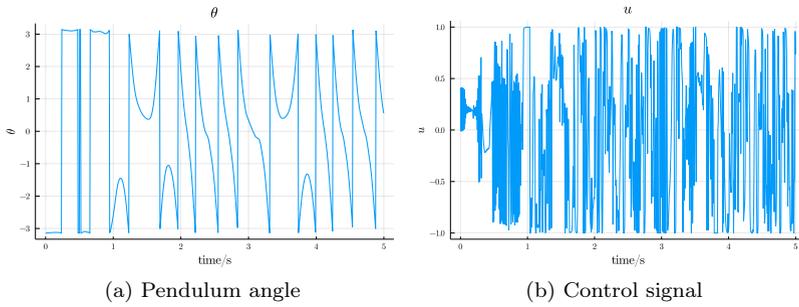


Figure C.13: The $\varepsilon = 0.075$ agent 2 with states perturbed by PGD, $\varepsilon = 0.36$.

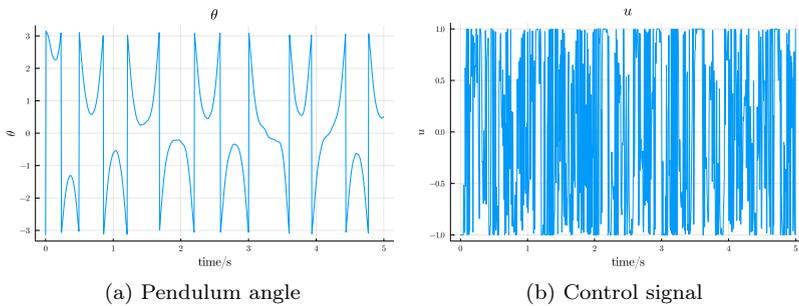


Figure C.14: The $\varepsilon = 0.15$ agent 3 with states perturbed by PGD, $\varepsilon = 0.36$.

Lund University Department of Automatic Control Box 118 SE-221 00 Lund Sweden		<i>Document name</i> MASTER'S THESIS	
		<i>Date of issue</i> October 2021	
		<i>Document Number</i> TFRT-6150	
<i>Author(s)</i> Philip Olhager		<i>Supervisor</i> Johan Grönqvist, Dept. of Automatic Control, Lund University, Sweden Richard Pates, Dept. of Automatic Control, Lund University, Sweden Anders Rantzer, Dept. of Automatic Control, Lund University, Sweden (examiner)	
<i>Title and subtitle</i> Robust Reinforcement Learning Control of a Furuta Pendulum			
<i>Abstract</i> <p>The use of Reinforcement Learning (RL) to design controllers for safety critical systems is an important research area. On the one hand, RL can function in and adapt to complex and changing environments without requiring a model of the system. On the other hand, in such systems robustness is of high importance, as well as ways to guarantee and certify a level of robustness. This project investigates state-of-the-art methods for training for and evaluating robustness in a neural network, when applied to RL control of a real-world system. The use of Projected Gradient Descent (PGD) as an adversary for robust Deep RL, as well as the Lipschitz constant as a measure of a neural network controller's robustness, are evaluated.</p> <p>The study is conducted by training an agent to perform swing-up and balancing of a Furuta pendulum, and further training it with PGD of varying magnitudes as an adversary. The agents are evaluated by their robustness towards normally-distributed measurement noise as well as their estimated Lipschitz constant.</p> <p>The results show that while training with PGD does result in better robustness for a classifier on the MNIST dataset, applying the technique to the Furuta pendulum in a Deep Reinforcement Learning setting is not so simple. One of 30 agents managed to outperform the baseline agent, indicating that while the technique may have some promise, further fine-tuning of the training process is necessary. Further, the Lipschitz constant did not correlate with robustness performance, indicating that it may not be an ideal measure of a neural network's robustness.</p>			
<i>Keywords</i>			
<i>Classification system and/or index terms (if any)</i>			
<i>Supplementary bibliographical information</i>			
<i>ISSN and key title</i> 0280-5316			<i>ISBN</i>
<i>Language</i> English	<i>Number of pages</i> 1-53	<i>Recipient's notes</i>	
<i>Security classification</i>			