Koopman Constrained Policy Optimization

A Koopman operator theoretic method for differentiable optimal control in robotics

Matthew Retchin

Thesis Committee

Shuran Song, Ph.D.	*Columbia University*
Carl Vondrick, Ph.D.	*Columbia University*
Brandon Amos, Ph.D.	*Meta AI*

A document submitted in partial fulfillment of the requirements for the degree of
Master of Science
at
Columbia University School of Engineering and Applied Science

Abstract

Deep reinforcement learning has recently achieved state-of-the-art results for robotic control. Robots are now beginning to operate in unknown and highly nonlinear environments, expanding their usefulness for everyday tasks. In contrast, classical control theory is not suitable for these unknown, nonlinear environments. However, it retains an immense advantage over traditional deep reinforcement learning: guaranteed satisfaction of hard constraints, which is critically important for the performance and safety of robots.

This thesis introduces Koopman Constrained Policy Optimization (KCPO), combining implicitly differentiable model predictive control with a deep Koopman autoencoder. KCPO brings new optimality guarantees to robot learning in unknown and nonlinear dynamical systems.

The use of KCPO is demonstrated in Simple Pendulum and Cartpole with continuous state and action spaces and unknown environments. KCPO is shown to be able to train policies end-to-end with hard box constraints on controls. Compared to several baseline methods, KCPO exhibits superior generalization to constraints that were not part of its training.

Acknowledgments

Thank you to mom, dad, Sarah, Michael, and friends (especially Lexie).

Thank you to my Principal Investigator Shuran Song for giving me the freedom to pursue the topics that interested me while providing excellent mentorship. Thank you also to Carl Vondrick for being an exceptional professor as well as serving on my committee. I am so grateful to Brandon Amos for generously spending many hours discussing the theory and code underlying his differentiable MPC method and other concepts from his Ph.D. thesis. I also greatly appreciate Steve Brunton for his advice and support as I debugged my Koopman autoencoder implementation.

Open-source projects like PyTorch and others have made my work possible, and I am indebted to their maintainers for creating such ergonomic tools of thought supported by brilliant feats of engineering.

Thank you to the teachers I had growing up who showed me that learning is an adventure (especially Mr. Feinberg, Ms. Fitzpatrick, Mr. DeVore, and Mr. Phillips).

Thank you finally to the science fiction writers who have always inspired me.

Contents

1 Introduction

> The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.
> —Igor Stravinsky [Str39]

In this thesis, I have sought a method for generating box-constrained motion plans for robots, bounding controls between an upper and lower limit. Box constrained controls have a number of real-world applications, including the rotor acceleration in a quadrotor or the torque of the many actuators in a quadruped as it engages in locomotion. Serious damage to the robots or to surrounding humans and property could result if either of these robots were to exceed their physical limits.

One way to accomplish this goal is through model-based optimal control, the use of a mathematical model of a system's dynamics to influence a system's state trajectory and achieve an objective. There are many challenges to building and using models for the environments in which robots typically operate.

To narrow the scope of this thesis from tackling both control and perception at once, two assumptions have been made to help streamline focus solely on realistically challenging conditions of the environments for robotic control:

1. The environment is always fully observable and deterministic, eliminating perception as an issue.
2. Robots typically operate in unknown environments with nonlinear dynamics, making control difficult.

Due to the second assumption, the problem of real-world control presents difficulties due to nonlinearity and an unknown physical model. In order to achieve optimal control, it becomes necessary to do *system identification*, or data-driven estimation of a mathematical model characterizing a system's dynamics. The linear quadratic regulator (LQR) of classical control theory is only available for *linear* dynamical systems. Trajectory optimization for nonlinear dynamical systems is thus usually dealt with via local linearization, which for any given point in state space produces a linear approximation of the overall nonlinear system. Convergence to optimal trajectories with these trajectory optimization algorithms is usually quite sensitive to initialization due to nonconvexity.

Koopman operator theory is an alternative path via global linearization: every point in state space shares one linear model of the dynamics [Bru]. It is possible to employ functions that "lift" from the original state space, where the dynamics are nonlinear and difficult to both model and control, to a new space where a linear model becomes sufficient to characterize the

dynamics for the entire space. Crucially, once the dynamics have been linearized globally, the strong guarantees of optimality from classical control theory become available for use.

Koopman operator theory has recently been put to use in many domains in robotics, including rigid-body and soft-body robotics [WTL23]. This thesis focuses on one approach to apply Koopman operator theory, a neural network architecture called a Koopman autoencoder that learns the lifting functions and linear operator from data [LKB18]. While the first Koopman autoencoders focused on simply predicting dynamics, some later works have used these autoencoders for optimal control with great success [KM18] [YWK22] [HEK22].

Concurrently with the recent Koopman optimal control work, differentiable mathematical optimization methods have been introduced in recent years. Mathematical optimization is an inseparable part of the training of neural policies, but pioneering work from [Amo+18] introduced the use of differentiable convex optimization layers inside neural network architectures for a variety of applications, from quadratic programming to constrained trajectory optimization.

The principal contribution of this work has been to harness Koopman autoencoders' linearized dynamics model together with differentiable trajectory optimization for end-to-end trainable constrained policy learning.

1.1 Outline

In the Background, the mathematical framework needed to understand the motivation behind and machinery underlying the method is provided. The method is explained in detail in the Method section, and it is tested and analyzed rigorously with imitation learning experiments in the Experiments & Results section. The thesis concludes with some remarks on the insights gathered from the results as well as promising future directions.

1.2 Notation

Within this thesis, sets are indicated with blackboard bold (e.g. $\mathbb{R}$ is the set of real numbers), vectors are indicated with bold, lowercase letters (e.g. $\mathbf{k} \in \mathbb{R}^n$), matrices and tensors are indicated with bold, capital letters (e.g. $\mathbf{M} \in \mathbb{R}^{m \times m}$ is an $m \times m$-dimensional matrix). All other variables may be assumed to be scalars unless otherwise indicated in the text.

$\dot{x}$ and $\frac{dx}{dt}$ both refer to the first-order derivative of x with respect to time t. The former is Newton's notation, and the latter is Leibniz's.

2 Background

2.1 Dynamical Systems

Beginning with the work of Henri Poincaré on celestial mechanics and the three-body problem, mathematicians have studied the so-called *dynamical systems*. Dynamical systems are equations that describe the behavior of time-evolutionary processes [Lay]. They are exceedingly useful mathematical models and have seen applications in many fields; of particular interest to this thesis is their application to robotics, where the interaction of a robot with an environment can be viewed as dynamical system. Much of the presentation that follows is inspired by [Bru] and [Had22].

Dynamical systems take the general form:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t), t; \beta) \tag{2.1}$$

where $\mathbf{x}(t)$ is the vector representing the state of a system at timestep t; for instance, the position and velocity of an object.

f is a vector field (a vector-valued function) for how $\mathbf{x}$ evolves over time. For every point in state space, f provides the derivative of state change.

Dynamics may differ depending on time t, meaning dynamics functions can be *time-varying*. In contrast, *time-invariant* dynamical systems lack a dependency on time; time-invariant dynamics is a subset of time-varying dynamics.

$\mathbf{u}(t)$ represents actuation or control; it is an external intervention to perturb the naturally evolving state of the system. When modeling physics, actuation usually corresponds to force or torque. In this work, we study only systems that have actuation, but not all systems do. A system without actuation is *autonomous* (it does not depend on an independent variable), while a system with actuation is *non-autonomous*.

β represents any remaining parameters of the system that cannot be controlled and may not be modeled in the state space, but that nevertheless impact the dynamics of the system. For instance, the mass of an object may be considered a parameter of the system, even as it cannot be controlled.

For reasons of readability, β is omitted unless necessary, and we assume time-invariance, simplifying 2.1 to

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.2}$$

As we continue to introduce definitions and analyze properties of dynamical systems, we will for now omit actuation, further simplifying:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

2.1.1 Discrete-Time Dynamical Systems

A discrete-time version of the continuous dynamics described above consists of the following expression:

$$\mathbf{x}_{t+1} = F(\mathbf{x}_t) \tag{2.3}$$

Note that F is not a vector field, as seen in the continuous-time case. Instead, it is a so-called *map* from the state vector at one timestep to the next. Repeated application of this function advances the dynamics.

Discrete-time dynamics can be shown to be equivalent to continuous-time dynamics via the following integral expression corresponding to a *flow map*:

$$F_t(\mathbf{x}_{t_0}) = \mathbf{x}_{t_0} + \int_{t_0}^{t_0+k} f(\mathbf{x}_t)\, dt \tag{2.4}$$

For the goal of data-driven methods for control in unknown systems, modeling a system in discrete-time with a map is more appropriate than a vector field, as data are frequently presented in discretely sampled trajectories.

2.1.2 Linear Dynamical Systems

Within the family of dynamical systems, linear dynamical systems are a particularly useful set because much of classical control theory concerns the control of linear dynamical systems. A continuous-time, time-invariant linear dynamical system without actuation has the following form:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} \tag{2.5}$$

The dynamics of the above system are characterized by the eigenvalues and eigenvectors derived by the *spectral decomposition* (or *eigendecomposition*) of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{T}\mathbf{D}\mathbf{T}^{-1} \tag{2.6}$$

Where $\mathbf{T}$ and $\mathbf{D}$ contain the eigenvectors and eigenvalues λ_i of $\mathbf{A}$, respectively.

In the more general *Jordan decomposition*, $\mathbf{D}$ will be a diagonal block matrix with *Jordan blocks*, which is known as the *Jordan normal form* of $\mathbf{D}$ [HJ13]:

$$\mathbf{D} = \begin{bmatrix} \mathbf{J}_1 & & \\ & \ddots & \\ & & \mathbf{J}_N \end{bmatrix} \tag{2.7}$$

where $\mathbf{J}_k$ is the Jordan block corresponding to the kth eigenvalue:

$$\mathbf{J}_k = \begin{bmatrix} \mathbf{C}_k & \mathbf{I} & & \\ & \mathbf{C}_k & \ddots & \\ & & \ddots & \mathbf{I} \\ & & & \mathbf{C}_k \end{bmatrix} \tag{2.8}$$

The above block matrix contains the identity matrix $\mathbf{I}$ and $\mathbf{C}_k$, a real-valued matrix representing the kth complex-conjugate eigenvalue pair $a_k \pm ib_k$:

$$\mathbf{C}_k = \begin{bmatrix} a_k & b_k \\ -b_k & a_k \end{bmatrix} \tag{2.9}$$

In the context of a linear dynamical system, the solution for a linear dynamical system is as follows, using eigendecomposition 2.1.2 and integrating 2.5:

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}(0) = \mathbf{T}e^{\mathbf{D}t}\mathbf{T}^{-1}\mathbf{x}(0) \tag{2.10}$$

Euler's formula [Eul48], $e^{\pm ix} = \cos x \pm i \sin x$, gives us a new $\mathbf{C}_k$ for complex-conjugate eigenvalue $a_k \pm ib_k$:

$$e^{(a \pm ib)t} = e^{at \pm ibt} = e^{at}e^{\pm ibt} = e^{at}(\cos x \pm i \sin x) \tag{2.11}$$

$$\mathbf{C}_k = e^{at}\begin{bmatrix} \cos(bt) & \sin(bt) \\ -\sin(bt) & \cos(bt) \end{bmatrix} \tag{2.12}$$

The geometric intuition for this new $\mathbf{C}_k$, tailor-made for the Jordan normal form of a linear dynamical system, is that it is a rotation matrix scaled by an exponential decay (or growth) term, rotating clockwise around the eigenspace's complex plane. The degree to which the amplitude of rotation decays towards or grows away from a fixed point is determined by a. If $a > 0$, there is growth away from a fixed point. If $a < 0$, there is decay toward a fixed

point. Finally, if $a = 0$, there is neither growth nor decay. A *phase portrait* of a decaying linear dynamical system ($a < 0$) is in Figure 2.1. Due to this geometric intuition, it is often simpler to interpret the behavior of a linear system than that of a nonlinear system.

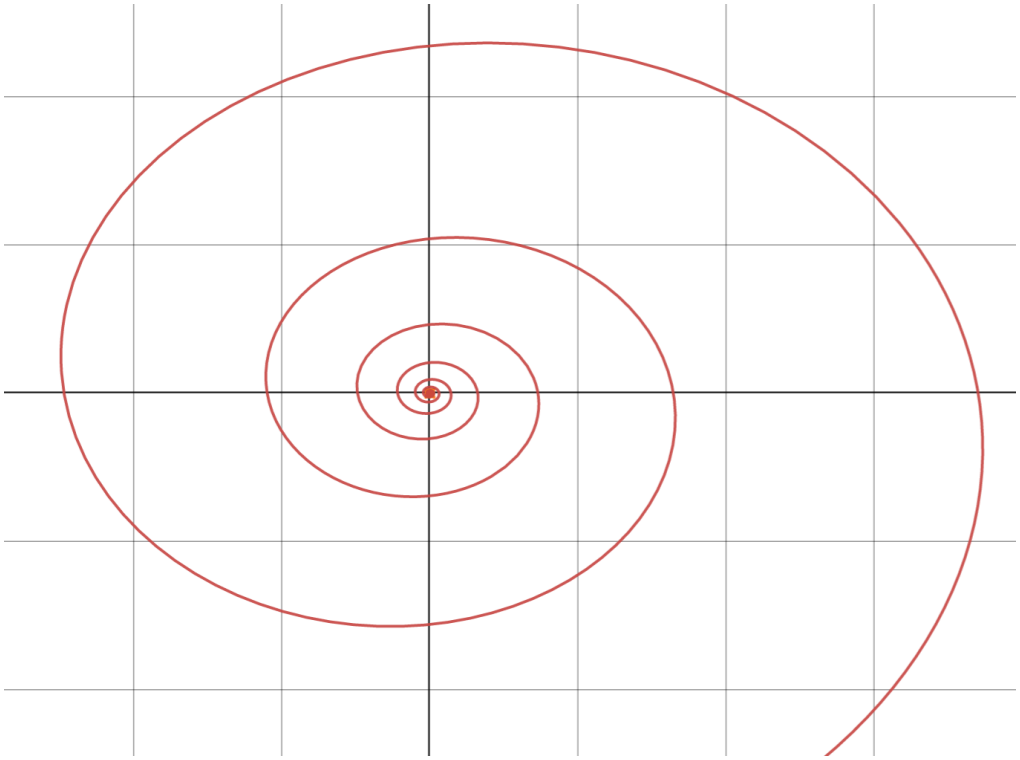

Figure 2.1: A phase portrait of a linear dynamical system that decays towards a fixed point at 0 due to $a < 0$ in Equation 2.12. Created using the Desmos online graphing tool [Kas].

Discretizing this continuous-time formulation with a discrete timestep Δt, we obtain:

$$\mathbf{x}_t = e^{\mathbf{A}\Delta t}\mathbf{x}_0 = \mathbf{T}e^{\mathbf{D}\Delta t}\mathbf{T}^{-1}\mathbf{x}_0 \tag{2.13}$$

Correspondingly, the discrete $\mathbf{C}_i$ is:

$$\mathbf{C}_i = e^{a\Delta t}\begin{bmatrix} \cos(b\Delta t) & \sin(b\Delta t) \\ -\sin(b\Delta t) & \cos(b\Delta t) \end{bmatrix} \tag{2.14}$$

Section 2.5 will introduce a solution to the central challenge of predicting and controlling a nonlinear dynamical system, such as a robot interacting with an environment. This is the Koopman autoencoder [LKB18], which can accurately predict these challenging dynamics. This prediction strategy is made possible by using the Jordan sub-block of Equation 2.14 to encode input states from a discrete nonlinear dynamical system into a new coordinate system, in the Jordan normal form of Equation 2.7. This dynamics encoding and Koopman autoencoder together comprise key components of the KCPO architecture.

2.2 Optimal Control

Mathematicians have long studied the properties of linear dynamical systems and developed classical control methods to obtain globally optimal actuation. Some of these methods are analytical, while others can guarantee convergence to globally optimal solutions through iterative numerical optimization [Teda]. KCPO's trajectory optimization exemplifies the iterative approach.

2.2.1 Model Predictive Control with Linear Quadratic Regulator

KCPO uses the basic Linear Quadratic Regulator (LQR) as a foundation for its trajectory optimization method. The discrete-time finite-horizon unconstrained Linear Quadratic Regulator (LQR) possesses an analytical solution to finding the optimal control trajectory $\tau^{\star}_{1:T} = \{\mathbf{x}, \mathbf{u}\}_{1:T}$ from a starting state $\mathbf{x}_0$ to a goal or reference state $\mathbf{x}_r$ with a finite horizon T.

Following [Amo+18], the problem formulation of LQR is as follows:

$$\begin{aligned} \tau^{\star}_{1:T} = \arg\min_{\tau_{1:T}} & \sum_{t=1}^{T} \frac{1}{2} \tau_t^{\top} \mathbf{C}_t \tau_t + \mathbf{c}_t^{\top} \tau_t \\ \text{s.t. } & \mathbf{x}_1 = \mathbf{x}_{\text{init}} \\ & \mathbf{x}_{t+1} = \mathbf{F}_t \tau_t + \mathbf{f}_t \end{aligned} \tag{2.15}$$

This problem formulation captures the desire to find a state-control trajectory that minimizes distance to a goal state over time using the minimum control possible while respecting linear time-varying dynamics constraints. The cost and dynamics can be potentially time-varying throughout the trajectory, which is notated by temporal subscript t in the objective function. Concretely, this problem formulation could be used to characterize a quadrotor flying towards a goal location from a starting location. KCPO extends this problem formulation by introducing additional box constraints in Section 2.6.1.

Importantly, the solution to this problem can be expected to yield a globally optimal state-control trajectory because the problem's formulation is an instance of the more general class of convex problems [Teda].

This problem formulation of Equation 2.15 is classically minimized by recursively solving the discrete algebraic Riccati equation (DARE) backward from the last timestep [Teda]. Metaphorically, this is a "tail-wags-dog" optimization: the optimization proceeds from the goal state at the last timestep in the horizon and work backward to the first timestep to determine what control trajectory would be necessary to reach that goal. Another term for this approach is *value iteration*, a kind of *dynamic programming* where the overall problem of computing the optimal control at the starting timestep depends recursively on sub-problems computing optimal controls proceeding into the future. Pseudo-code of the Forward and Backward Passes of Linear Quadratic Regulator is presented in Algorithms 1 and 2 respectively. The backward pass does value iteration to compute the *gain* matrix and bias $\mathbf{K}_t$ and $\mathbf{k}_t$, and the forward pass uses these gains to compute controls [Teda].

Algorithm 1 Linear Quadratic Regulator: Backward Pass, as presented in [Sal18]

$T \in \mathbb{Z}_+$ is the finite horizon length
$\mathbf{x} \in \mathbb{R}^n$ is a state vector
$\mathbf{u} \in \mathbb{R}^m$ is a control vector
$\mathbf{C} \in \mathbb{R}^{T \times n+m \times n+m}$ and $\mathbf{c} \in \mathbb{R}^{T \times n+m}$ are the objective terms. Each $\mathbf{C}_t$ must be positive semi-definite
$\mathbf{F} \in \mathbb{R}^{T \times n \times n+m}$ and $\mathbf{f} \in \mathbb{R}^{T \times n}$ are the dynamics constraint terms

for t = T to 1 **do**
  $\mathbf{Q}_t = \mathbf{C}_t + \mathbf{F}_t^\top \mathbf{V}_{t+1} \mathbf{F}_t$
  $\mathbf{q}_t = \mathbf{c}_t + \mathbf{F}_t^\top \mathbf{V}_{t+1} \mathbf{f}_t + \mathbf{F}_t^\top \mathbf{v}_{t+1}$
  $\mathbf{K}_t = -\mathbf{Q}_{\mathbf{u}_t,\mathbf{u}_t}^{-1} \mathbf{Q}_{\mathbf{u}_t,\mathbf{x}_t}$
  $\mathbf{k}_t = -\mathbf{Q}_{\mathbf{u}_t,\mathbf{u}_t}^{-1} \mathbf{q}_{\mathbf{u}_t}$
  $\mathbf{V}_t = \mathbf{Q}_{\mathbf{x}_t,\mathbf{x}_t} + \mathbf{Q}_{\mathbf{x}_t,\mathbf{u}_t} \mathbf{K}_t + \mathbf{K}_t^\top \mathbf{Q}_{\mathbf{u}_t,\mathbf{x}_t} + \mathbf{K}_t^\top \mathbf{Q}_{\mathbf{u}_t,\mathbf{u}_t} \mathbf{K}_t$
  $\mathbf{v}_t = \mathbf{q}_{\mathbf{x}_t} + \mathbf{Q}_{\mathbf{x}_t,\mathbf{u}_t} \mathbf{k}_t + \mathbf{K}_t^\top \mathbf{Q}_{\mathbf{u}_t} + \mathbf{K}_t^\top \mathbf{Q}_{\mathbf{u}_t,\mathbf{u}_t} \mathbf{k}_t$
end for
return $\mathbf{K}_{1:T}, \mathbf{k}_{1:T}$

and the forward pass applies forward the gain matrix and bias computed in the backward pass.

Algorithm 2 Linear Quadratic Regulator: Forward Pass, as presented in [Sal18]

$T \in \mathbb{Z}_+$ is the finite horizon length
$\mathbf{x} \in \mathbb{R}^n$ is a state vector
$\mathbf{x}_{\text{init}} \in \mathbb{R}$ is the initial state vector
$\mathbf{u} \in \mathbb{R}^m$ is a control vector
$\mathbf{C} \in \mathbb{R}^{T \times n+m \times n+m}$ and $\mathbf{c} \in \mathbb{R}^{T \times n+m}$ are objective terms. $\mathbb{C}$ must be positive semi-definite
$\mathbf{F} \in \mathbb{R}^{T \times n \times n+m}$ and $\mathbf{f} \in \mathbb{R}^{T \times n}$ are dynamics constraint terms

$\mathbf{x}_1 = \mathbf{x}_{\text{init}}$
for t = 1 to T **do**
  $\mathbf{u}_t^\star = \mathbf{K}_t \mathbf{x}_t + \mathbf{k}_t$
  $\mathbf{x}_{t+1} = \mathbf{F}_t \tau_t + \mathbf{f}_t$
end for
return $\mathbf{u}_{1:T}^\star$

Problematically, the time complexity for solving DARE scales linearly with the horizon T, which becomes intractable as the limit of the horizon approaches infinity.

In practice, *model predictive control* (MPC), also known as receding horizon control, provides a tractable solution for a finite horizon. MPC selects $\mathbf{u}_1^*$ at our current timestep using a finite horizon and re-solves DARE from scratch in the next timestep, this time advancing the limited horizon forward also by a timestep [Tedb].

2.2.2 Challenges for Control of Nonlinear Dynamical Systems

Linear dynamical systems have the very attractive property that globally optimal controls are available, but one challenge is that real-world dynamics are frequently nonlinear. Another

challenge is that dynamics are often unknown — usually, only time series data is available, not the equations that produced the data. These two major challenges can be addressed through the use of Koopman autoencoders.

2.3 Koopman Operator Theory

As in the Dynamical Systems section, we derive much of this introduction to Koopman Operator Theory from [Bru].

Due to the attractive properties of linear dynamical systems, much of the existing body of work for control of nonlinear dynamical systems focuses on locally linearizing around a point. While this approach is taken by widely used algorithms such as Iterative Linear Quadratic Regulator [TMT14] and the Sparse Nonlinear OPTimizer (SNOPT) [GMS05], it is unlikely to yield globally optimal control, because the approximation error of the linearized dynamics grows tremendously beyond the small region around the linearization point. In addition, the nonconvex problem formulations these algorithms are designed to solve typically require a warm start initialization, stymieing their practical use. Engineers using these algorithms must therefore rely on expert knowledge to initialize as close as possible to the optimal solution.

Koopman operator theory offers hope that it may be feasible to extend the benefits of existing linear control theory techniques to nonlinear dynamical systems. In 1931, Bernard Koopman showed that it is possible to diffeomorphically [1] lift an original state $\mathbf{x}$, which has nonlinear dynamics, into a new space with linear dynamics. The *Koopman operator* is a linear operator [2] that completely characterizes the original nonlinear dynamics. The implication is that in this newly lifted space, classical linear control techniques like LQR could be used.

For many decades after Koopman's discovery, an immense obstacle to the practical use of his operator for dynamics prediction or control applications was that its diffeomorphism is infinite-dimensional: a Hilbert space [3] of measurement functions $g(\mathbf{x})$, also called *observables*, which requires a suitably infinite-dimensional Koopman operator. This is an obstacle indeed, as LQR only applies to finite-dimensional matrices. Thus, many researchers have focused on how to approximate Koopman operators with finite-dimensional matrices and also approximate the Koopman operator's true infinite-dimensional Hilbert space with finite-dimensional approximations.

The following sections will review data-driven techniques that have been employed to approximate a subset of the Koopman operator's eigenvalues and eigenfunctions and a corresponding subspace of the full Hilbert space. This approximate operator would enable the use of globally optimal LQR technique for control of nonlinear physical systems.

[1] a *diffeomorphism* is a differentiable, invertible mapping from a smooth manifold to another smooth manifold — in the context of Koopman operator theory, the first smooth manifold is the original nonlinear dynamical system, and the second smooth manifold is a linear dynamical system.

[2] an *operator* is a general term for a function mapping from elements in one space to another; a matrix is an example of an operator.

[3] a *Hilbert space* is the infinite-dimensional generalization of the finite-dimensional vector space for which linear algebra was originally developed.

2.3.1 Dynamic Mode Decomposition

Dynamic Mode Decomposition (DMD) is one of the simplest approaches to the problem of approximating the infinite-dimensional Koopman operator with a finite-dimensional one. DMD takes snapshots of data and arranges them into time-lagged matrices $\mathbf{X} \in \mathbb{R}^{m \times n}$ and $\mathbf{X}' \in \mathbb{R}^{m \times n}$: one matrix starts with snapshot $\mathbf{x}_1$ and ends at snapshot $\mathbf{x}_n$, while the other matrix starts with snapshot $\mathbf{x}_2$ and ends with snapshot $\mathbf{x}_{n+1}$.

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \\ | & | & & | \end{bmatrix} \tag{2.16a}$$

$$\mathbf{X}' = \begin{bmatrix} | & | & & | \\ \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_{n+1} \\ | & | & & | \end{bmatrix} \tag{2.16b}$$

DMD then applies least squares linear regression to the problem $\mathbf{AX} = \mathbf{X}'$ to find a low-dimensional operator A that approximates the true infinite-dimensional Koopman operator that characterizes the dynamics.

However, this method has significant limitations. Using least squares regression is prone to overfitting, so DMD is not robust to noise and cannot generalize well beyond its training data.

2.3.2 Extended Dynamic Mode Decomposition

One major improvement upon the basic DMD algorithm is Extended Dynamic Mode Decomposition (eDMD), which extends DMD by introducing nonlinear measurements of the data prior to constructing the time-lagged matrices and performing least squares regression.

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \psi(\mathbf{x}_1) & \psi(\mathbf{x}_2) & \cdots & \psi(\mathbf{x}_n) \\ | & | & & | \end{bmatrix} \tag{2.17a}$$

$$\mathbf{X}' = \begin{bmatrix} | & | & & | \\ \psi(\mathbf{x}_2) & \psi(\mathbf{x}_3) & \cdots & \psi(\mathbf{x}_{n+1}) \\ | & | & & | \end{bmatrix} \tag{2.17b}$$

eDMD has turned out to be much more effective than DMD because it is able to model more complicated systems: when the limit of the number of snapshots approaches infinity, eDMD's $\mathbf{A}$ converges to the actual infinite-dimensional Koopman operator.

Yet this convergence property only holds true if the measurement functions are closed under the Koopman operator, a property called *Koopman invariance*. For some problems, expert knowledge that a particular set of observables will be effective and span a Koopman invariant subspace is available. Prior works have usually seen these observables take the form of monomials [Bru+16].

Since expert knowledge is not always sufficient or even available, recent work has shown that deep neural networks can effectively learn observables from scratch [Bru] [YWK22] [Had22] [Li+20]. Before exploring these works, we will review the theory of deep neural networks and their backpropagation training procedure.

2.4 Deep Neural Networks

Artificial neural networks (or just "neural networks" when context is clear) are algorithms inspired by the metaphor of biological neurons in the brain. However, the artificial neural network as a mathematical model is typically much simpler than a real biological neural network. A *multilayer perceptron* (MLP), a commonly used neural network architecture type, comprises weight matrices, bias vectors, and nonlinear activation functions. Sets of weights, biases, and activation functions are segregated together into *layers*. The activation functions could be any kind of nonlinearity, from sine to the hyperbolic tangent function. [GBC16a]

Using an MLP requires matrix-vector multiplication and addition, where an input vector is successively multiplied by weight matrices, added to bias vectors, and then passed through a nonlinear activation function element-wise. The output is known as an *activation vector*. This is

$$\mathbf{a}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \tag{2.18}$$

where σ is the nonlinear activation function and $\mathbf{a}^{(l-1)}$ is the activation vector from a previous layer.

Activation vectors are passed from layer to layer until reaching the end of the neural network, producing an output vector. Some output layers have activation functions, while others do not, depending on the application and intent of the designer of a particular neural network architecture.

A *feedforward* neural network connects its layers to each other in only one direction, while a *recurrent* neural network can feed back into itself. This distinction becomes important for later sections where the Koopman autoencoder and other recurrent neural networks are discussed. Recurrent neural networks are often used for tasks involving prediction [HS97].

[HSW89] proved a Universal Approximation Theorem showing that a multi-layer neural network with at least one hidden layer is a universal approximator of any function, from the XOR logic gate to image classification. Despite the result that any function may be approximated by a neural network with at least one hidden layer, the theorem carries no hint as to how to find the perfect parameters for that neural network. *Deep learning*, the training of neural networks with more than one hidden layer, empirically has enabled the discovery of quality parameters for a vast variety of applications. The motivation for training deep neural networks is their demonstrably superior learning ability compared to shallower neural networks [GBC16a].

2.4.1 Backpropagation and the Chain Rule

Although other algorithms are also used (such as neuroevolution [GM21]), MLPs are generally trained using *backpropagation*, an application of the chain rule from calculus with *gradient descent*, a first-order optimization method [GBC16b]. Let us assume that scalar-valued loss function $\ell(\mathbf{o})$ represents how optimally our neural network achieves the objective, where $\mathbf{o}$ is the neural network's output activation vector. The *forward pass* is the computation of $\ell(\mathbf{o})$ by propagating the input vector through the MLP's layers and passing the activation vector of one layer to the next in the manner of Equation 2.18. After the forward pass, the MLP can be trained by *backpropagating* the gradient of the loss function $\ell(\mathbf{a})$ with respect to the MLP parameters $\mathbf{W}$ and $\mathbf{b}$ to those same parameters. Recall the chain rule,

$$\frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{x}))}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \tag{2.19}$$

Computing $\nabla_{\mathbf{W}} \ell(\mathbf{o})$ and $\nabla_{\mathbf{b}} \ell(\mathbf{o})$, the gradients of the loss function with respect to the weights and biases respectively, takes place by applying the chain rule to Equation 2.18. The backpropagation algorithm combines the chain rule with memoization [4] as a performance optimization to eliminate redundant computations.

After computing $\nabla_{\mathbf{W}} \ell(\mathbf{o})$ and $\nabla_{\mathbf{b}} \ell(\mathbf{o})$, because a gradient is the direction and rate of steepest *ascent*, gradient *descent* applies the gradients to their respective parameters with a small step size or *learning rate* and move toward a local minimum of the loss function:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} \ell(\mathbf{o}) \tag{2.20}$$

where α is the step size and θ is a parameters vectors containing the weights and biases of the neural network.

2.4.2 Constrained Policy Optimization

This section situates KCPO within a constrained policy learning formalism.

Markov Decision Processes & Policy Optimization

Constrained policy learning, of which KCPO is an example, is a subfield of reinforcement learning (RL) and optimal control (OC), which both seek to model agents that act optimally in an environment. Mathematically, this idea of an optimal agent within an environment can be formulated as a Markov decision process (MDP), a tuple $(\mathcal{X}, \mathcal{U}, T_u, R)$ [Lev21].

$\mathcal{X}$ is the set of states the environment and agent can inhabit, and $\mathcal{U}$ is the set of controls an agent can exert on the environment. $T_u(\mathbf{x}, \mathbf{x}') = \Pr(\mathbf{x}_{t+1} = \mathbf{x}' \mid \mathbf{x}_t = \mathbf{x}, \mathbf{u}_t = \mathbf{u})$ is the transition function, which outputs the likelihood that the action $\mathbf{u}$ in the state $\mathbf{x}$ at timestep t will transition to the state $\mathbf{x}'$ at timestep $t+1$. $R(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})$ is a reward function

[4] *memoization* is a type of caching that stores the return values of functions (in this case, the activation vector $\mathbf{a}^{(l)}$ of each layer) so they do not need to be redundantly computed again and again [Cor+09]

determining the immediate reward provided to the agent at state $\mathbf{x}_{t+1}$ when the agent acts with control $\mathbf{u}_t$ and state $\mathbf{x}_t$.

MDPs are characterized by the *Markov property assumption* [SB20a]. This property assumes that the past may be decoupled from the future, such that the present contains all the information necessary to accurately model the future. The MDP framework enforces this assumption with a transition function that explicitly expresses $\mathbf{x}_{t+1}$'s sole dependency on $\mathbf{x}_t$ and $\mathbf{u}_t$. This simplification permits agents to make efficient decisions without holding every past event in memory.

Based on [Tedc] and [Ach+17]'s definitions, *policy optimization* is the process of optimizing (or learning) a parameter vector θ to parameterize a *policy* $\pi_\theta(\mathbf{u}_t \mid \mathbf{x}_t)$ to model the distribution over controls $\mathcal{U}$ that maximizes discounted expected reward given current state $\mathbf{x}_t$, where the discount factor is $\gamma \leq 1$:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_t \gamma^t R(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})\right] \tag{2.21}$$

$$\theta^* = \arg\max_\theta \; J(\theta) \tag{2.22}$$

where $J(\theta)$ is the objective, otherwise known as the *value function*, which measures the expected discounted reward if controls generated by policy π_θ are followed after starting at an initial state $\mathbf{x}_0$.

The *policy gradient* is defined as $\nabla J(\theta)$ and may be used to solve Equation 2.22.

Note that $\pi_\theta(\mathbf{u}_t \mid \mathbf{x}_t)$ could be a neural network trained with a combination of the policy gradient method and backpropagation.

There are two classes of policy optimization methods, *model-free* and *model-based*, although hybrid methods do exist. Model-based methods rely on a state-transition probability function $T_u(\mathbf{x}, \mathbf{x}')$, while model-free methods do not do so explicitly. The method discussed in this thesis, KCPO, is model-based, because it trains a Koopman operator as a model of the world [SB20b].

There is also a distinction between *online* and *offline* training. Online training optimizes the agent's policy by making the agent engage with the environment directly. For offline training, data are first generated from the environment and subsequently used to train the policy [Lev+20]. While nothing precludes KCPO from being trained online, this thesis reports only offline imitation learning experiments.

ReflexNet

Not all neural network-based policies are strictly of the form $\pi_\theta(\mathbf{x}_t \mid \mathbf{u}_t)$. [Kur+22] introduce a highly effective neural network architecture trained to imitate an expert, with access to a

classical controller and the true dynamics. Their architecture is a multilayer perceptron that takes in the initial state and outputs the entire trajectory at once:

$$\mathbf{u}_{1:T} = \pi_\theta(\mathbf{x}_0) \tag{2.23}$$

This is unlike a typical policy architecture that maps one-to-one from a single state to a single control:

$$\mathbf{u}_t = \pi_\theta(\mathbf{x}_t) \tag{2.24}$$

In other words, the input and output dimensions for the constrained version of the architecture are n_x and $n_u \cdot T$ respectively, where n_u is the dimension of a single control for one timestep, and T is the total number of timesteps in the trajectory.

The architecture is referred to as the ReflexNet because the paper describes their model as mimicking reflexive movements in animals, where reflexive movements can be thought of as full trajectories generated in response to a single stimulus.

Because [Kur+22] find that the ReflexNet architecture of 2.23 has better performance than the traditional policy network architecture of 2.24, we use it as a baseline in Section 4.

Constrained Markov Decision Process & Constrained Policy Optimization

Constrained Policy Optimization (CPO), as originally published by [Ach+17], refers to a specific neural network-based constrained policy learning method. But in this thesis, we invoke a more general *constrained policy optimization*, referring to any approach of optimizing a policy's parameters such that the policy respects constraints. KCPO performs constrained policy optimization, optimizing a policy that respects hard box constraints (e.g., torque limits of a quadrotor) during both training and inference.

A *constrained* Markov decision process (CMDP) augments the MDP tuple with constraint functions $C_i(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}) \leq d_i$ and a feasible set, $\mathcal{C}$. This set contains all the state-control transitions that are feasible (do not violate the constraints). Auxiliary objective functions $J_{C_i}(\theta)$ measure the expected discounted return of policy π_θ with respect to constraint C_i: $J_{C_i}(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_t \gamma^t C_i(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})\right]$. While the objective $J(\theta)$ is to be maximized, these auxiliary objective functions are to be minimized during constrained policy optimization.

Given a discrete-time environment with a continuous action and state space and a deterministic transition function, the constrained policy optimization problem can be formulated as

$$\begin{aligned} \theta^* &= \arg\max_\theta \; J(\theta) \\ &\text{s.t. } J_{C_i}(\theta) \leq d_i \end{aligned} \tag{2.25}$$

The specific Constrained Policy Optimization outlined in [Ach+17] optimizes a neural policy $\pi_\theta(\mathbf{u} \mid \mathbf{x})$ parameterizing a conditional probability distribution over actions $\mathcal{U}$ conditioned on states $\mathcal{X}$. The optimization method is model-free and combines the policy gradient with the trust region method. The trust region method follows the intuition that an update to a policy's parameters should keep the new policy within the feasible neighborhood (or "trust region") surrounding the existing policy in parameter space. If the updated policy is feasible, then the update is accepted, and the trust region's radius expands; if the updated policy is infeasible, then the update is rejected, and the trust region's radius shrinks. Thus, the training process itself is constrained.

There are two constraints that CPO minimizes with its trust region method during training. The first constraint minimizes the average *Kullback-Leibler divergence* $\bar{D}_{\mathrm{KL}}(\pi_{k+1} \mid\mid \pi_k) = \mathbb{E}[D_{\mathrm{KL}}(\pi_{\theta_{k+1}}(\mathbf{u} \mid \mathbf{x}) \mid\mid \pi_{\theta_k}(\mathbf{u} \mid \mathbf{x})]$ between the old policy π_k and the new policy π_{k+1}, restricting within a tolerable amount ϵ how much information [5] is lost if the new policy's probability distribution replaces the old. I.e., this first constraint restricts how dissimilar the new policy's distribution can be from the old policy's distribution. The second constraint is that the new policy π_k's violation of constraint functions $C_i(\mathbf{x}_t, \mathbf{u}_t) \leq d_i$, where d_i is a separate violation tolerance assigned to each constraint function C_i.

The output of CPO is a policy that approximately solves Equation 2.25 because, like Koopman Differentiable Predictive Control, constraints cannot be guaranteed to always be satisfied by the policy's controls. A CPO-trained policy does not ever solve a constraint optimization problem directly — at best, it is statistically unlikely to violate constraints.

Although the CMDP has a stochastic transition function in its most general form, the environment's dynamics are assumed to be only deterministic within this thesis. Stochastic extensions to KCPO are addressed briefly in 5.

There have been many more methods besides CPO proposed for neural net-based constrained policy optimization in recent years, all of which to the best of our knowledge rely on similar methods of policies that are statistically likely to satisfy constraints after training. This family of methods is known as *probabilistic constraint satisfaction*, rather than hard constraint satisfaction. An extensive survey of these methods may be found in [Gu+22]. Of these methods, one significant inspiration has been [Amo+18], which forms the foundational inspiration of this thesis. An in-depth overview of the work of [Amo+18] can be found in Section 2.6.

2.5 Koopman Autoencoders

Koopman autoencoders like those used in KCPO combine Koopman operator theory with the traditional autoencoder neural network architecture to learn observables and the Koopman operator simultaneously from data. Below, the usual autoencoder architecture is summarized before moving on to the specific Koopman variant that inspired the one used for KCPO.

[5] *Information* is a measure of the degree to which an event reduces uncertainty about the state of a random variable [Sha48]. In the case of CPO, the random variable is the policy's output control.

An autoencoder is a deep neural network with encoder φ and decoder φ^{-1} modules that mirror each other: the former learns to encode its input $\mathbf{x}$ into a representation $\mathbf{z}$, while the latter seeks to reconstruct the original input by decoding the representation. The architecture can be trained using backpropagation, as most neural networks are, with a reconstruction loss function $\|\mathbf{x} - \varphi^{-1}(\varphi(\mathbf{x}))\|$, which measures the difference between the originally observed data and the reconstruction from the decoder. Due to its special structure, the architecture can learn hidden variables that are representative of its input. In other words, from a probabilistic perspective, it is designed to infer the latent variables $\mathbf{z} = \varphi(\mathbf{x})$ that condition the observed variables (the input $\mathbf{x}$) [GBC16c].

An *undercomplete* autoencoder has a bottleneck between the encoder and decoder, with fewer dimensions in this latent space than in the observed space, contrasting with the *overcomplete* autoencoder, having greater dimensions in the latent space than in the observed. A bottleneck is useful for distilling the original observation into the most essential information needed for reconstruction [GBC16c]. This compression is very useful for applications such as computer vision. But in the context of Koopman operator approximation, the high dimensionality of an overcomplete autoencoder turns out to be much more germane to modeling the infinite-dimensional Hilbert space that a Koopman operator resides in.

Thus, the standard Koopman autoencoder is overcomplete, parameterizing latent variables $\mathbf{L}$, a matrix between the encoder and decoder that also serves as the finite-dimensional approximation of the true infinite-dimensional Koopman operator [LKB18]. The encoder and decoder from this perspective are learning the diffeomorphism from the original state space (which has nonlinear dynamics), to a latent state space (with linear dynamics). Because the main goal of the Koopman autoencoder is prediction rather than reconstruction, a prediction loss $\|\varphi(\mathbf{x}_{t+1}) - \mathbf{L}\varphi(\mathbf{x}_t)\|$ is preferred to train the autoencoder. The Koopman autoencoder is typically used to predict an entire trajectory, not just one step ahead, by iteratively applying $\mathbf{L}$. Others have therefore used a k-step prediction loss function during training: $\|\varphi(\mathbf{x}_{t+k}) - \mathbf{L}^k\varphi(\mathbf{x}_t)\|$. This design has been useful for predicting many dynamical systems, from natural phenomena like fluid flows to the rigid-body dynamics of robotics [Aze+21] [MWK19].

[LKB18] describes a variant of the basic Koopman autoencoder design, utilizing an auxiliary neural network to contextually generate a conditional Koopman operator $\mathbf{L}_{x_t}$, where the operator is conditioned on the initial state $\mathbf{x}_t$ from which prediction begins, $\mathbf{x}_{t+1:t+k}$ (see Figure 2.3). This variant allows the latent space to use significantly fewer dimensions, making this autoencoder undercomplete with fewer parameters. For example, [LKB18] successfully use a 2-dimensional latent space to model the Simple Pendulum (see Figure 2.2). In contrast, [YWK22] use a basic Koopman autoencoder with a 200-dimensional latent space.

In this thesis, [LKB18]'s variant is used because it requires fewer parameters and theoretically has more accurate prediction than the basic version. This is important because the control performance that concerns KCPO is tied directly to prediction accuracy. The basic Koopman autoencoder harms prediction accuracy by using a global operator $\mathbf{L}$ everywhere in state space, because any global operator ought to have infinite dimensions for a true diffeomorphism. Koopman operator theory insists that a finite operator is likely limited. By contrast,

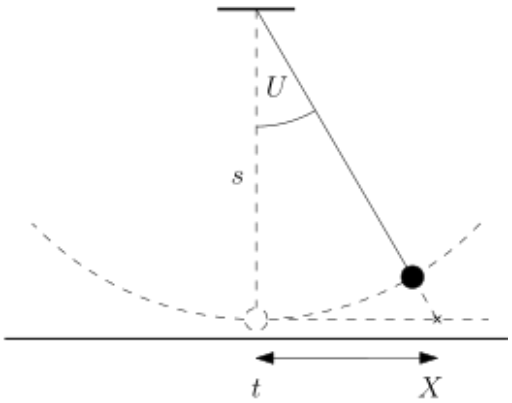

Figure 2.2: Pendulum Environment

instead of a finitely sized operator with a discrete set of eigenvalues, Lusch et al. point out that their auxiliary network can learn the entire continuous spectrum of **L**'s eigenvalues.

This auxiliary network parameterizes the Jordan normal form of the Koopman matrix using Equation 2.14 discussed in the Section 2.1.2.

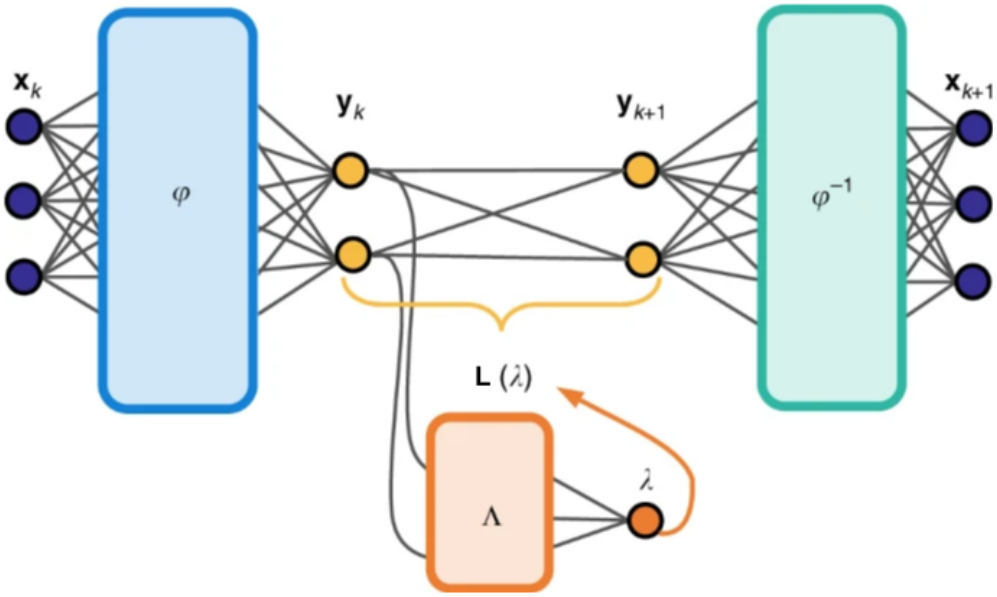

Figure 2.3: Koopman autoencoder with continuous spectrum auxiliary network for autonomous systems, used with permission from [LKB18]

2.5.1 Koopman Autoencoders for Control

While the Koopman autoencoder architecture began with dynamics prediction, its use for control has exploded in recent years.

[YWK22] were the first to use a Koopman autoencoder for optimal control in an end-to-end trainable architecture, and their work directly inspired KCPO. Their method used a Koopman autoencoder with unconstrained, analytically solved LQR embedded within a *Koopman policy*. They treated the Koopman policy component as a black box neural component and trained it within a reinforcement learning framework. Because this thesis additionally introduces constraints on top of optimizing an objective, KCPO relies on a different approach to achieve end-to-end differentiability. However, the spirit of the two methods is similar.

Unlike [YWK22], typical approaches to using Koopman autoencoders for optimal control often do not make possible end-to-end training. Instead, dynamics models are often first pre-trained to minimize prediction error. These dynamics models are then used within MPC under the assumption that low prediction error results in a low control error. Examples of this second approach to Koopman optimal control include [Li+20], [HEK22], [Wei+22], [Kin+22], and [SM22].

[Li+20]'s Compositional Koopman Operator (CKO) is able to model and control the dynamics of rigid bodies. Uniquely, the compositional aspect of the operator is that it is a block matrix with several copies of the operator, one for each body in the rigid body in the system with the assumption that all the bodies will have identical physics.

One obstacle that the authors of CKO faced was their choice of a "metric" cost function for MPC, meaning that their cost function was the identity matrix I_n. Because CKO is not trained end-to-end and does not learn the cost function, there can be a mismatch between the cost function used for minimizing control error and the learned latent Koopman space that is optimized for minimizing prediction error. Thus, CKO is trained with a so-called *metric loss*, which penalizes the Koopman autoencoder such that it encodes latent state embedding vectors that are the same Euclidean distance from each other as their original corresponding state vectors are from each other.

$$|x_\alpha - x_\beta| = |z_\alpha - z_\beta| \tag{2.26}$$

Equation 2.26 shows how CKO's metric loss ensures that x states are the same distance from each other as the z latent states (encoded versions of x).

Taking note of the difficulty that CKO faces with its cost function, we designed the KCPO method to learn a cost function that is always valid, or positive semi-definite, by construction. This will be covered in more detail in the Method section.

Deep Stochastic Koopman Operator (DeSKO) takes advantage of Koopman operator theory's linearization to analyze how to provide robust control in a stochastic environment, which is an exciting development in the literature due to Koopman operator theory's original use-case for predicting deterministic dynamics [HEK22]. Unlike the typical autoencoder design that has a nonlinear decoder, DeSKO uses a linear decoder in order to make their stochastic control analysis possible.

[SM22] similarly stretch the definition of the Koopman operator, concatenating the original state vector x with the encoder's latent vector z: $[x, z]$. After several steps of prediction, there

is no nonlinear decoder because x can be taken from that concatenated vector directly, a linear decoder where the decoder is simply the identity matrix.

Because Koopman operator theory suggests that a nonlinear decoder is required for perfect reconstruction from latent Koopman space, it can be expected that these approaches severely damage the prediction error with an infinite horizon prediction. However, for the purposes of control with MPC, a perfect model may not be required because the model needs only to predict a finite and short number of steps ahead.

Besides these works, the use of a linear decoder also appears in a paper on EDMD-based optimal control [KM18]. The benefit of using a linear decoding is that the architecture does not require a cost function in the latent Koopman space and can penalize directly in state space, instead of learning a special cost function as in KCPO. Although KCPO's approach does come at a cost of extra implementation complexity, this trade-off was chosen in order for KCPO to optimize its control objective without compromising on Koopman operator accuracy at longer horizons.

[Wei+22]'s Koopman Q-Learning is one prior work that uses Koopman operator theory to aid reinforcement learning. Contrasting with [YWK22], which is also a reinforcement learning work using Koopman operator theory, Koopman Q-Learning is model-free and not trained end-to-end with backpropagation. Unlike the model-based methods presented in this thesis and [YWK22], Koopman Q-Learning relies on the Koopman autoencoder to discover useful symmetries for data augmentation during offline training.

[Kin+22] introduces Koopman-Based Differentiable Predictive Control, a Koopman-based extension of a previous work by the authors called Neural Differentiable Predictive Control [Drg+22]. Koopman-Based Differentiable Predictive Control, like the work presented in this thesis, is focused on constrained model-based policy optimization with a differentiable linear Koopman operator as its model, but it enforces constraints softly using a variation on the penalty method: the outputs of the constraint functions are heavily penalized in the loss function as the policy's neural network is trained via policy gradient. In other words, this method's constraints are not hard, meaning that the policy is always permitted to violate the constraints at any time. Statistically speaking, after training, the policy is highly likely to satisfy the constraints during inference, but there is no guarantee. KCPO has no such limitation, which is an asset in safety-critical online training. Indeed, the authors of Koopman-Based Differentiable Predictive Control themselves state their method should be trained offline. Like many of the other methods mentioned, and unlike [YWK22], the weights of the autoencoder are pre-trained for prediction and frozen throughout training on the control task.

None of the above actuated Koopman autoencoder architectures use [LKB18]'s auxiliary network to parameterize their Koopman operator eigenspectrum. Instead, they each use a common operator for the entire state space. In practice, this requires very high dimensionality in the latent space, resulting in the overcomplete autoencoder design. To the best of our knowledge, KCPO's autoencoder is the first Koopman autoencoder design to extend [LKB18]'s auxiliary network design from dynamics prediction to control tasks.

2.6 Implicit Differentiation of Model Predictive Control

The focus of this thesis is on MPC with box-constrained control constraints for safe torques. KCPO relies on a method known as Box-DDP to enforce these box constraints.

2.6.1 Control-Limited Differential Dynamic Programming (Box-DDP)

Box-DDP solves a trajectory optimization problem with nonconvex cost function $c(\tau_t)$, nonconvex dynamics $f(\tau_t)$, and box constraints $\underline{\mathbf{u}}$ and $\overline{\mathbf{u}}$:

$$\begin{aligned} \tau_{1:T}^{\star} = \arg\min_{\tau_{1:T}} & \sum_{t=1}^{T} c(\tau_t) \\ \text{s.t. } & \mathbf{x}_1 = \mathbf{x}_{\text{init}} \\ & \mathbf{x}_{t+1} = f(\tau_t) \\ & \underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}} \end{aligned} \tag{2.27}$$

[Amo+18], whose work KCPO builds upon, uses an implementation of MPC with box-constrained controls. This is achieved via a modified version of the Control-Limited Differential Dynamic Programming heuristic (also known as Box-DDP) [TMT14].

Box-DDP and its general family of trajectory optimization methods are known as the so-called *indirect* trajectory optimization methods. They contrast with the *direct* trajectory optimization methods, which optimize and constrain the entire trajectory holistically by considering both states and controls as decision variables to optimize. Instead, indirect trajectory optimization methods only consider controls as decision variables to optimize, indirectly enforcing dynamics constraints on the controls by simulating forward rollouts of the controls with the environment. These methods are attractive because they ensure trajectories corresponding to decision variables are always feasible (i.e. they respect dynamics constraints) throughout optimization [Jac].

Differential Dynamic Programming (DDP), which is itself a modification on Iterative Linear Quadratic Programming (iLQR), are indirect methods. Like LQR, they are value iteration methods that break down the entire trajectory optimization problem into individual timesteps that each depend on each other. If the last timestep is fully optimized with respect to the objective, then the penultimate timestep can use that result itself to be optimized. This piecemeal, tail-wags-dog optimization process proceeds all the way to the first timestep.

However, iLQR and DDP broaden LQR's usefulness, since LQR can only optimize trajectory with respect to a quadratic cost function and can further only optimize trajectories respecting dynamics constraints provided that the dynamics constraints are linear. With iLQR and DDP, trajectories can be optimized with respect to arbitrary nonconvex cost functions and nonconvex dynamics constraints.

If the dynamics function is once-differentiable and nonlinear, and the cost function is twice-differentiable and not quadratic, iLQR *linearizes* the nonlinear dynamics function using first-order Taylor expansions and *quadratizes* the cost function using a second-order Taylor expansion. As the Taylor series for any function is an infinite sum that exactly equals that function, iLQR and DDP use this fact to approximate the function by truncating its corresponding infinite Taylor series to a finite sum of a few terms.

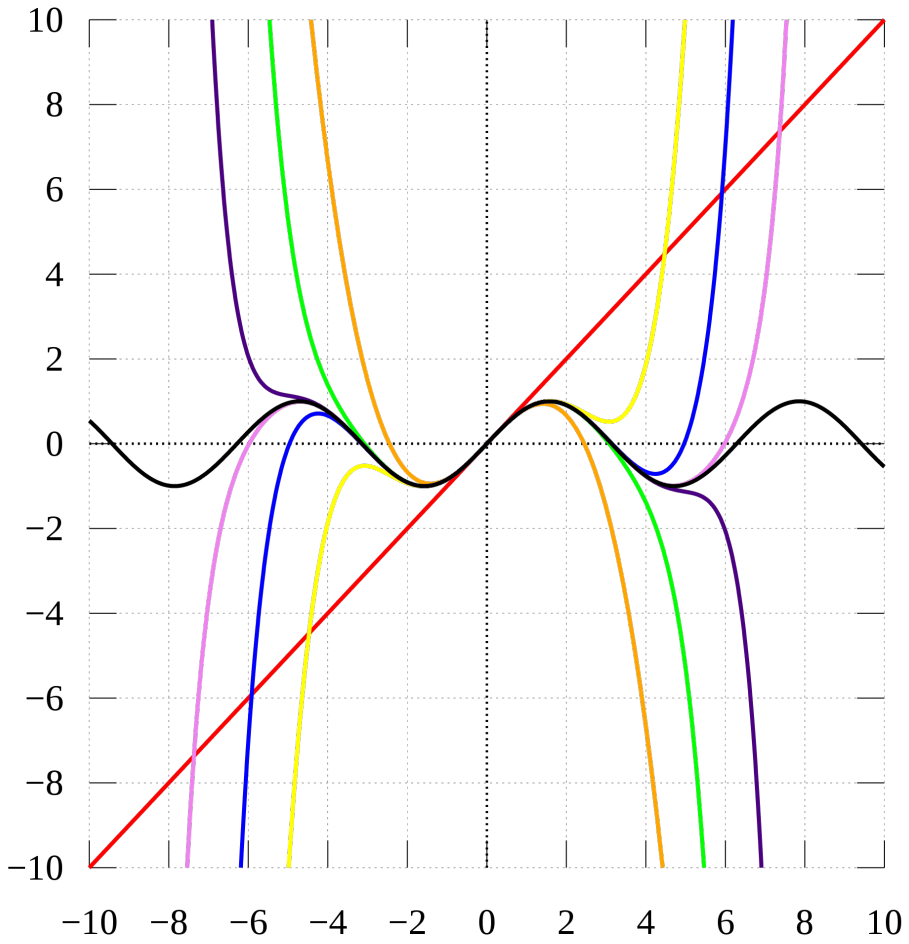

Figure 2.4: Taylor series expansion. Licensed under Creative Commons: [Ika]

The nonlinear dynamics function can be approximated with a linear function using the first-order Taylor expansion for the dynamics centered around the initialization $\hat{\tau}_{1:T}$ [Sal18].

$$f(\tau_t) \approx f(\hat{\tau}_t) + \nabla_{\tau_t} f(\hat{\tau}_t)(\tau_t - \hat{\tau}_t) \tag{2.28}$$

$$c(\tau_t) \approx c(\hat{\tau}_t) + \nabla_{\tau_t} c(\hat{\tau}_t)(\tau_t - \hat{\tau}_t) + \frac{1}{2}(\tau_t - \hat{\tau}_t)\nabla^2_{\tau_t} c(\hat{\tau}_t)(\tau_t - \hat{\tau}_t) \tag{2.29}$$

Based on the truncated Taylor expansions of Equations 2.28 and 2.29 (see Figure 2.4), the original nonconvex problem can be reformulated as an LQR problem formulation with a quadratic objective and linear dynamics:

$$\begin{aligned}
\tau_{1:T}^{\star} = \arg\min_{\tau_{1:T}} & \sum_{t=1}^{T} \frac{1}{2}\tau_t^\top \mathbf{C}_t \tau_t + \mathbf{c}_t^\top \tau_t \\
\text{s.t. } & \mathbf{x}_1 = \mathbf{x}_{\text{init}} \\
& \mathbf{x}_{t+1} = \mathbf{F}_t \tau_t + \mathbf{f}_t \\
& \mathbf{F}_t = \nabla_{\tau_t} f(\hat{\tau}_t) \\
& \mathbf{f}_t = f(\tau_t) - \mathbf{F}_t \tau_t \\
& \mathbf{C}_t = \nabla_{\tau_t}^2 c(\hat{\tau}_t) \\
& \mathbf{c}_t = \nabla_{\tau_t} c(\hat{\tau}_t) - \mathbf{C}^\top \tau_t
\end{aligned} \tag{2.30}$$

With this new formulation, the iLQR algorithm is described in Algorithm 3.

Algorithm 3 Iterative Linear Quadratic Regulator, as presented in [Sal18]

$T \in \mathbb{Z}_+$ is the finite horizon length
$\mathbf{x} \in \mathbb{R}^n$ is a state vector
$\mathbf{x}_{\text{init}} \in \mathbb{R}$ is the initial state vector
$\mathbf{u} \in \mathbb{R}^m$ is a control vector
$\mathbf{C} \in \mathbb{R}^{T \times n+m \times n+m}$ and $\mathbf{c} \in \mathbb{R}^{T \times n+m}$ are objective terms. $\mathbf{C}$ must be positive semi-definite
$\mathbf{F} \in \mathbb{R}^{T \times n \times n+m}$ and $\mathbf{f} \in \mathbb{R}^{T \times n}$ are dynamics constraint terms

1: Generate random control sequence $\hat{\mathbf{u}}_{1:T}$
2: Create rollout with true dynamics and $\hat{\mathbf{u}}_{1:T}$ to get $\hat{\mathbf{x}}_{1:T}$, giving a $\hat{\tau}_{1:T}$ initialization
3: **while** unconverged **do**
4: $\quad \delta\mathbf{x}_t = \mathbf{x}_t - \hat{\mathbf{x}}_t$
5: $\quad \delta\mathbf{u}_t = \mathbf{u}_t - \hat{\mathbf{u}}_t$
6: $\quad \mathbf{K}_t, \mathbf{k}_t \leftarrow$ LQR backward pass with $\mathbf{F}, \mathbf{f}, \mathbf{C}, \mathbf{c}$, and $\delta\mathbf{x}$ and $\delta\mathbf{u}_t$
7: $\quad$ Run LQR forward pass with true nonlinear dynamics and $\mathbf{u}_t = \hat{\mathbf{u}}_t + \mathbf{K}_t(\delta\mathbf{x}_t) + \mathbf{k}_t$
8: $\quad$ Update $\hat{\tau}_t$ using states and controls from forward pass
9: **end while**
10: **return** $\mathbf{u}_{1:T}^{\star}$

After quadratizing the cost function and linearizing the dynamics constraints with a state-control trajectory initialization, iLQR applies LQR iteratively to polish the trajectory. This iteration is necessary because unlike LQR, iLQR does not provide an exact analytical expression for the optimal solution to the trajectory optimization problem. In practice, the forward pass update step may overshoot the true solution $\mathbf{u}_{1:T}^{*}$ due to cost and dynamics approximation error. Therefore, the forward pass is usually amended with a line search, but this detail is not pertinent to explain the underlying mechanism of the algorithm.

DDP is a variant of iLQR that assumes a twice-differentiable dynamics function and uses a second-order Taylor expansion for dynamics linearization, but these methods are otherwise quite similar and have roughly the same rate of convergence to optimal trajectories [Tedc].

Because the experiments reported in this thesis involve navigation to goal states, the cost function is already quadratic in distance from current state to goal state. Thus, quadratization of the cost function for the use-case in this thesis is unnecessary. Following [Amo+18], the objective function $c(\tau_t)$ for weighted distance from current τ_t to goal τ_g, or $c(\tau_t) = \|w \circ (\tau_t - \tau_g)\|_2^2$ with w being the weighting, becomes in quadratic form

$$c(\tau_t) = \frac{1}{2}\tau_t^\top \text{diag}(w)\tau_t - (\sqrt{w} \circ \tau_g)^\top \tau_t \tag{2.31}$$

where $\text{diag}(w)$ is a diagonal matrix with w filling its entries. In other words, $\mathbf{C}_t = \text{diag}(w)$ and $\mathbf{c}_t = -(\sqrt{w} \circ \tau_g)$. Box-DDP either expects a quadratic objective or will quadratize the objective itself, motivating the use of a quadratic form of the objective to preempt the Box-DDP solver and save computation.

In addition, dynamics are already linear after Koopmanization, rendering linearization unnecessary. The reason for using Box-DDP despite having a quadratic cost function and linear dynamics via a Koopman operator is because of the primary goal of KCPO: constraints.

Box-DDP uses a quadratic program solver to introduce box constraints into the DDP method. In the backward pass of DDP's Riccati recursion, Box-DDP employs a heuristic where each backward step in the recursion is constrained one at a time. Recalling that Box-DDP is an indirect method, the downside of this approach is that it does not apply box constraints to the entire trajectory holistically. A direct method would, by contrast, apply box constraints everywhere simultaneously; hypothetically, box constraints imposed at the beginning of the trajectory ought to impact the optimization of later timesteps in the trajectory. Thus, Box-DDP is unlikely to result in an optimal trajectory when applied just once.

However, with repeated iterations, Box-DDP will converge to a trajectory that is optimal with respect to the objective and box constraints. Again, being an indirect method, Box-DDP's trajectories already implicitly satisfy the dynamics constraints.

Note that [Amo+18]'s implementation of Box-DDP actually uses first-order linearization instead of the second-order linearization called for by the original Box-DDP. However, in keeping with [Amo+18]'s terminology, "Box-DDP" will be used to refer to the first-order iLQR version within this text.

Readers are encouraged to return to the backward pass pseudocode for Iterative Linear Quadratic Regulator algorithm. The significant change in a box-constrained iterative linear quadratic regulator from the original unconstrained version is the introduction of box constraints (lower bounds $\underline{\mathbf{u}}$ and upper bounds $\overline{\mathbf{u}}$) in the backward pass at Line 4. These constraints are enforced only one timestep at a time as the loop steps backwards, instead of holistically as in a direct trajectory optimization method. A numerical quadratic program solver is typically used to enforce these constraints instead of using the analytical formulas of the unconstrained version.

Although linearization and quadratization are not needed for KCPO, linearization is still required to generate a dataset from a Box-DDP-based expert for imitation learning experiments, where the expert has access to the true nonlinear dynamics functions. This topic will be revisited in Section 4.

2.6.2 Implicit Differentiation of Model Predictive Control

In practice, most constrained MPC solvers are not explicitly differentiable, making it challenging to use them to train policies with hard constraints. Following [Amo+18] and [KD], let us treat a constrained MPC solver as a function $\mathbf{z}^\star(\mathbf{x})$, with $\mathbf{x}$ being the input state and $\mathbf{z}^\star$ being the optimal outputs (both the optimal primal variables and optimal dual variables). In practice, $\mathbf{z}^\star(\mathbf{x})$ does not easily admit an explicit definition in terms of its independent variable $\mathbf{x}$. A consequence of lacking such a definition is that the Jacobian $\frac{\partial \mathbf{z}^\star(\mathbf{x})}{\partial \mathbf{x}}$ cannot be computed using the typical automatic differentiation pipeline. Unlike a function such as $\sin(\alpha)$, for which the derivative can be easily computed as $\cos(\alpha)$, $\mathbf{z}^\star(\mathbf{x})$ is a black box. One could compute $\frac{\partial z^\star(x)}{\partial x}$ by programming an MPC solver with differentiable operations and unrolling all the operations into a differentiable computational graph. But computing that Jacobian via unrolling is prohibitively slow in practice because it often requires hundreds of iterations for an MPC solver to reach a fixed point solution.

Fortunately, the Implicit Function Theorem (IFT) provides an alternative differentiation method [KP02]. The constrained MPC solver may be considered to be an *implicit* function $g(\mathbf{x}, \mathbf{z}^\star)$, which parameterizes a new function in terms of both the independent and dependent variable. Intuitively, our choice of $g(\mathbf{x}, \mathbf{z}^\star)$ is an optimality function where suboptimal inputs u will result in $g(\mathbf{x}, \mathbf{u}) > 0$, but the root $g(\mathbf{x}, \mathbf{z}^\star) = 0$ exists for the fixed point solution to the MPC optimization.

With this implicit formulation of the constrained MPC solver function, $\frac{\partial g(\mathbf{x}, \mathbf{z}^\star)}{\partial \mathbf{x}}$ can be *solved for*, under the condition that $g(\mathbf{x}, \mathbf{z}^\star) = 0$. First, let us define the optimality function, which is at a root at MPC's optimal fixed point $\mathbf{z}^\star$, satisfying the aforementioned condition.

$$g(\mathbf{x}, \mathbf{z}^\star(\mathbf{x})) = 0 \tag{2.32}$$

In the next step of the theorem, both sides may be differentiated w.r.t. x.

$$\frac{\partial g(\mathbf{x}, \mathbf{z}^\star(\mathbf{x}))}{\partial \mathbf{x}} = 0 \tag{2.33}$$

The chain rule expands the partial derivative of 2.33 into two new partial derivatives, each w.r.t. $\mathbf{x}$. Because $g(\mathbf{x}, \mathbf{z}^\star)$ is a multivariate function, both terms must be summed together.

$$\frac{\partial g(\mathbf{x}, \mathbf{z}^\star)}{\partial \mathbf{x}} + \frac{\partial g(\mathbf{x}, \mathbf{z}^\star)}{\partial \mathbf{z}^\star} \frac{\partial \mathbf{z}^\star(\mathbf{x})}{\partial \mathbf{x}} = 0 \tag{2.34}$$

Because this differentiation takes place at a root, a linear system of equations can be set up, and $\frac{\partial \mathbf{z}^\star(\mathbf{x})}{\partial \mathbf{x}}$ can be solved.

$$\frac{\partial \mathbf{z}^\star(\mathbf{x})}{\partial \mathbf{x}} = -\left(\frac{\partial g(\mathbf{x}, \mathbf{z}^\star)}{\partial \mathbf{z}^\star}\right)^{-1} \frac{\partial g(\mathbf{x}, \mathbf{z}^\star)}{\partial \mathbf{x}} \tag{2.35}$$

The IFT has provided an analytical path to compute $\frac{\partial \mathbf{z}^\star(\mathbf{x})}{\partial \mathbf{x}}$ entirely in terms of $g(\mathbf{x}, \mathbf{z}^*)$ instead of $\mathbf{z}^\star(\mathbf{x})$. With IFT, the MPC solver's black box becomes transparent and available for use within a differentiable constrained policy training pipeline.

3 Koopman Constrained Policy Optimization

3.1 Forward Pass: Combining Koopman Autoencoders with Differentiable MPC

Koopman Constrained Policy Optimization combines the Koopman Autoencoder of [LKB18] with the implicitly differentiable MPC of [Amo+18] to optimize a policy that provably respects hard box constraints while maximizing control performance, solving Equation 2.27.

This combination of [LKB18]'s Koopman autoencoder with [Amo+18]'s implicitly differentiable MPC overcomes a great prior drawback of the original differentiable MPC: "Sometimes the controller does not run for long enough to reach a fixed point [...], or a fixed point doesn't exist, which often happens when using neural networks to approximate the dynamics." The requirement for fixed point solutions to trajectory optimization arises inherently and unavoidably in [Amo+18]'s use the IFT to differentiate through MPC.

Unfortunately, if using neural networks for the dynamics of MPC, [Amo+18]'s algorithm was very unlikely to reach these required fixed points due to the strong nonconvexity introduced by the nonlinear neural network-based dynamics. If neural networks cannot be used in MPC, then that limits the scope of applicability for differentiable MPC to tasks such as system identification, where the physics equations are a white box save for a few parameters.

The Koopman autoencoder addresses this issue by producing entirely linear dynamics without any Taylor expansion. With linear dynamics, constrained MPC optimization provably converges to a fixed point, assuming the feasible set is nonempty. We enable end-to-end training in the *forward pass* of the KCPO algorithm, shown below, by a composition of the Koopman autoencoder and implicitly differentiable MPC.

The positive semi-definiteness of the cost function is a critical assumption for the feasibility of the trajectory optimization problem, otherwise a solution will likely not exist. To ensure that the cost matrix $\mathbf{C}$ for the KCPO algorithm is positive semi-definite, the auxiliary neural network that outputs the parameters for the cost function (AuxiliaryCostNN in 3.1) generates the lower triangle of $\mathbf{C}$'s Cholesky decomposition: $\mathbf{C} = \mathbf{L}\mathbf{L}^\top$. In other words, the cost function $\mathbf{C}$ generated by AuxiliaryCostNN is positive semi-definite by construction [GF96].

Algorithm 4 Koopman Constrained Policy Optimization

$\mathbf{x} \in \mathbb{R}^n$ is a state vector
$\mathbf{u} \in \mathbb{R}^m$ is a control vector
$\underline{\mathbf{u}}, \overline{\mathbf{u}} \in \mathbb{R}^m$ are the lower and upper bound box constraints on controls
$\mathbf{C} \in \mathbb{R}^{T \times n+m \times n+m}$ and $\mathbf{c} \in \mathbb{R}^{T \times n+m}$ are quadratic cost terms. $\mathbf{C}$ must be positive semi-definite
$\mathbf{F} \in \mathbb{R}^{T \times n \times n+m}$ and $\mathbf{f} \in \mathbb{R}^{T \times n}$ are affine dynamics cost terms

1: **function** KCPO($\mathbf{x}_{1:T}, \underline{\mathbf{u}}, \overline{\mathbf{u}}; \theta$)
2: $\quad \mathbf{C}, \mathbf{c} \leftarrow \textsc{AuxiliaryCostNN}(\mathbf{x}_1; \theta)$
3: $\quad \mathbf{F}, \mathbf{f} \leftarrow \textsc{AuxiliaryDynamicsNN}(\mathbf{x}_1; \theta)$
4: $\quad \mathbf{z}_{1:T} \leftarrow \textsc{Encoder}(\mathbf{x}_{1:T}; \theta)$
5: $\quad \mathbf{u}^\star_{1:T} \leftarrow \textsc{DifferentiableMPC}_{T,\underline{\mathbf{u}},\overline{\mathbf{u}}}(\mathbf{z}_1, \mathbf{C}, \mathbf{c}, \mathbf{F}, \mathbf{f})$ ▷ Implicitly differentiable
6: $\quad \hat{\mathbf{x}}_{1:T} \leftarrow \textsc{Decoder}(\mathbf{z}_{1:T}; \theta)$
7: $\quad$ **return** $\mathbf{u}^\star_{1:T}, \hat{\mathbf{x}}_{1:T}$
8: **end function**

3.2 Backpropagation

Training KCPO requires backpropagating through a task-specific policy loss and an autoencoding reconstruction loss. Taking cues from the prior work by [YWK22] on training unconstrained Koopman policies, who note that prediction loss surprisingly damages the overall task accuracy, the training procedure of KCPO does not have a prediction loss. This is unlike prior works that usually use a prediction loss, minimizing the difference between the Koopman Autoencoder's predictions of future observations and actual observations in the future.

$$\ell = \ell_{\text{policy}}(\mathbf{u}^\star_{1:T}) + \ell_{\text{reconstr}}(\hat{\mathbf{x}}_{1:T}) \tag{3.1}$$

The gradient of the reconstruction loss, $\nabla_\theta \ell_{\text{reconstr}}$ is backpropagated through KCPO($\mathbf{x}_{\text{init}}; \theta$) in the typical manner with explicit differentiation.

However, $\nabla_\theta \ell_{\text{policy}}$ is backpropagated through KCPO($\mathbf{x}_{\text{init}}; \theta$) in two parts. First, lines $2-4$ (using the neural networks) are differentiated explicitly, while line 5 is differentiated implicitly, because the differentiable MPC solver relies on the IFT.

Using the chain rule, the PyTorch library seamlessly connects the derivative of the MPC layer w.r.t. the loss to the derivative of the previous layers w.r.t. their inputs and parameters, even as those layers are being differentiated using different methods (implicit and explicit, respectively) [Pas+19].

3.3 Stable Training

[Amo+18] acknowledge two stability problems in training for certain tasks.

First, as previously mentioned, there are occasions during training when MPC optimization fails to reach a fixed point for some or all samples in a batch. In that case, Amos et al. pro-

grammed their training procedure to detach from the ruined, unconverged samples to prevent spoiling of the training with a runtime exception.

Second, for a task involving simultaneous system identification and cost function learning when imitating an expert policy, training could not converge. The authors found that cycling between training the cost function parameters and training their system parameters was the only path to solve the task. Even the method of detaching unconverged samples could not help.

Koopman Constrained Policy Optimization addresses both of these instability problems by eliminating the nonconvexity of the dynamics, again thanks to Koopman operator theory: linearity means that MPC always converges. When MPC always converges to a fixed point, arbitrarily large and complex sets of parameters — such as those of a neural network with its many weight matrices — may be trained with batch gradient descent.

4 Experiments & Results

Given the motivation behind the development of KCPO, it is important to be able to evaluate whether the method can train with hard box constraints enforced for the entire episode. Training without hard constraints enforced from the start could be disastrously unsafe for policies trained online in risky control environments like nuclear power plants.

Also critical is the model's ability to generalize to new constraints once trained. Because training a neural network for complicated control tasks can be an expensive endeavor both computationally and monetarily, an appealing feature of KCPO would be training on one set of constraints and running with a different set as requirements change. In other words, box constraints, not environments with alternate dynamics constraints, are the focus of generalization in these experiments.

For these reasons, the experimental setup of [Amo+18] has been slightly modified. The training environments used are the classic Simple Pendulum and Cartpole (see Figures 2.2 and 4.1). The former environment is fully actuated, while the latter environment is underactuated and thus more difficult to solve.

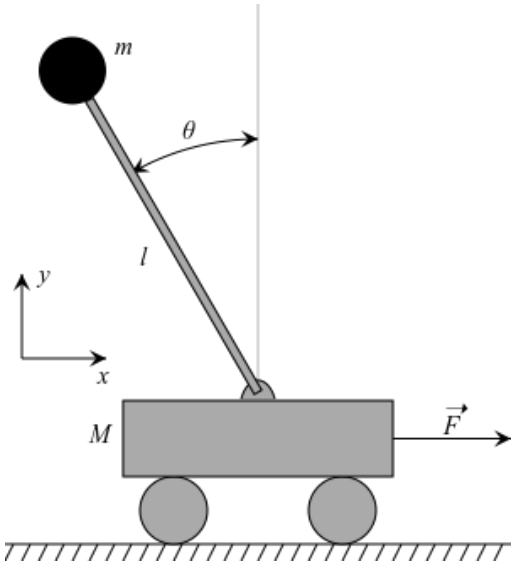

Figure 4.1: Cartpole Environment

4.1 Baselines

The KCPO's efficacy is evaluated by comparing against two baseline methods.

4.1.1 Recurrent Neural Network

The first baseline is inspired by the baseline of [Amo+18], a recurrent neural network (RNN) based on the Long Short-Term Memory (LSTM) cell [HS97] (see Figure 4.2). The trajectory's initial state is given to the RNN and recurrently fed back its output until a full trajectory has been generated. The LSTM cell was first introduced by [HS97] to address a stability issue that occurs during training of recurrent neural networks known as the *exploding (or vanishing) gradient problem*. The LSTM cell has special "forgetting gates" (orange and lime green in Figure 4.2) that determine which parts of the recurrent memory to remember or forget, fixing the exploding gradient problem.

Where this baseline version differs with the original baseline is that it has an activation function in the form of the hyperbolic tangent (frequently abbreviated to *tanh*) that ensures box constraints are satisfied. This activation function is described in more detail in Section 4.1.3 and can be viewed in Figure 4.3.

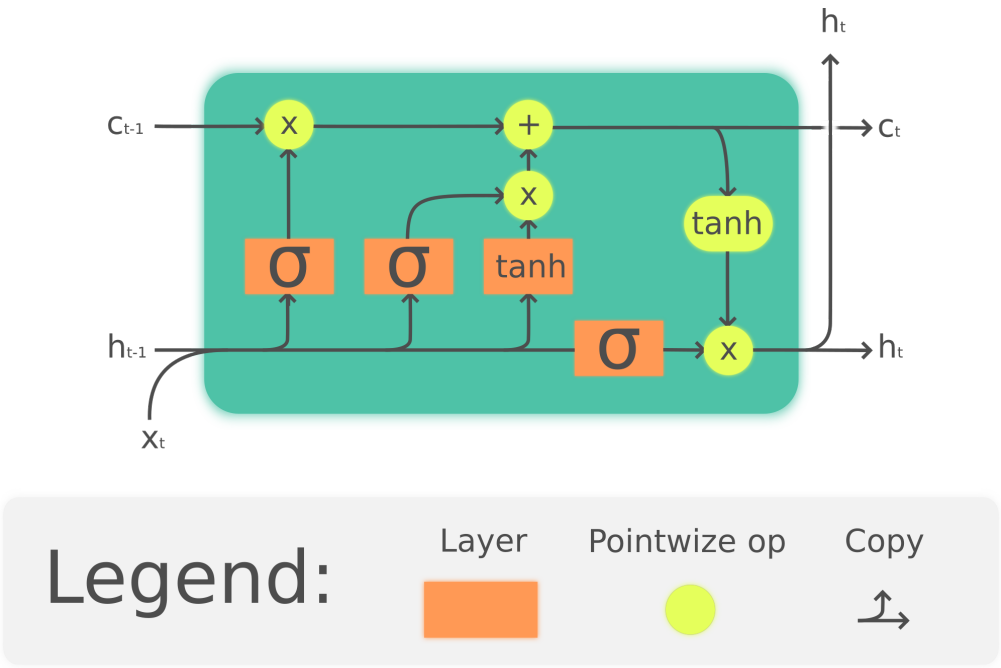

Figure 4.2: LSTM-Based RNN. Licensed under Creative Commons: [Che]

4.1.2 ReflexNet

The second baseline is inspired by [Kur+22]'s ReflexNet (see Section 2.4.2), but it is modified to enforce hard constraints.

Like the RNN baseline, the constrained ReflexNet architecture is capped at its end with the squashing activation function, again to impose box constraints.

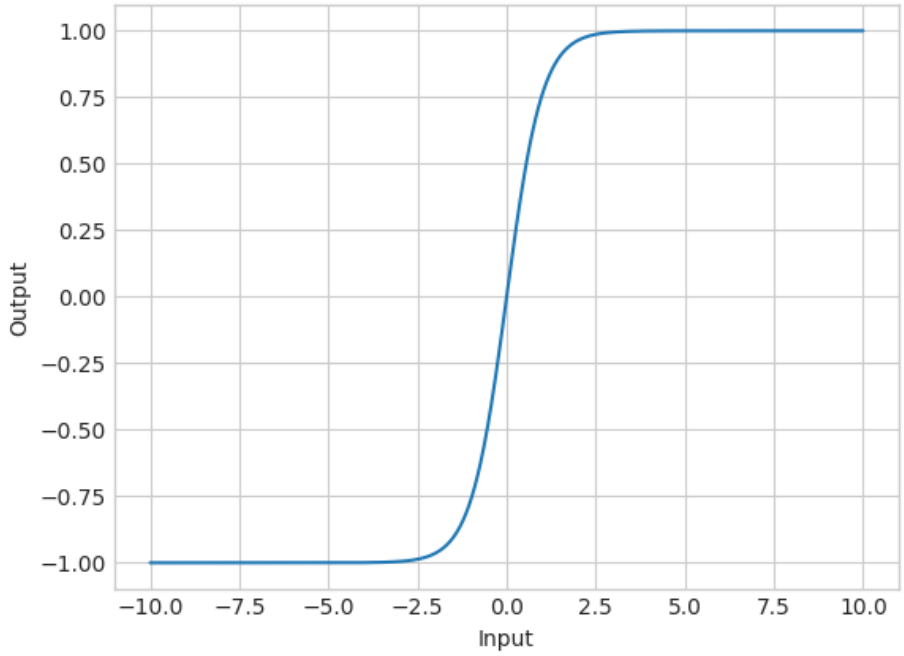

Figure 4.3: Hyperbolic Tangent Squashing Function

4.1.3 Squashing Activation Function

The hyperbolic tangent activation has been selected for use in the baselines because its range is $(-1, 1)$, asymptotically approaching 1 as the limit of the input approaches infinity, as can be seen in Figure 4.3. Hence, it can be metaphorically thought of as "squashing" its input. This works well for box constraints $-1 \leq \tanh(u) \leq 1$, but the following formula enables arbitrary lower and upper bounds (with u being the output of the neural network):

$$a = \frac{\overline{u} - \underline{u}}{2} \tag{4.1}$$

$$b = \overline{u} - a \tag{4.2}$$

$$u^{\star} = a \cdot \tanh(u) + b \tag{4.3}$$

Equation 4.3 scales and translates the output of $\tanh(u)$ so that it fits into any desired bounds. This equation is crucial for successfully bounding the trajectory outputs in both baseline neural networks.

4.2 Tasks

KCPO and the baseline methods are trained via imitation learning, attempting to copy an expert policy that uses MPC with full access to the true dynamics equations and system parameters. More specifically, the task samples a random initial state and requires the agent to navigate to an unstable fixed point.

Following [Amo+18], for Simple Pendulum, the initial state is randomly generated via

$$\theta \sim U(-\frac{\pi}{2}, \frac{\pi}{2}) \tag{4.4}$$
$$\dot{\theta} \sim U(-1, 1) \tag{4.5}$$
$$x_{\text{init}} = [\cos\theta, \sin\theta, \dot{\theta}] \tag{4.6}$$

and the fixed point goal is $[1, 0, 0]$.

For Cartpole, the initial state is randomly generated via

$$x \sim U(-0.5, 0.5) \tag{4.7}$$
$$\dot{x} \sim U(-0.5, 0.5) \tag{4.8}$$
$$\theta \sim U(-\pi, \pi) \tag{4.9}$$
$$\dot{\theta} \sim U(-1, 1) \tag{4.10}$$
$$x_{\text{init}} = [x, \dot{x}, \cos\theta, \sin\theta, \dot{\theta}] \tag{4.11}$$

and the fixed point goal is $[0, 0, 1, 0, 0]$.

Expert control trajectories were generated and split into training, testing, and validation sets with a distribution of 1000, 100, and 100 samples respectively. Given the initial state in the trajectory and the box constraints, each method must enforce the box constraints throughout training while aiming to reduce the imitation error and match the expert's control trajectory.

Training took 250 epochs, meaning that the full training set was shown to the model 250 times.

After training finishes, each method is evaluated with imitation error on the held-out test set. All methods are trained on the same dataset for ten random trials each with different reproducible pseudorandom seeds initializing neural network weights and shuffling the dataset.

Architectural and other kinds of hyperparameters are tuned using the held-out validation set. The RNN and ReflexNet experiments are trained with the Adam optimizer on the CPU with a learning rate of 1×10^{-3}, while KCPO is trained with a learning rate of 1×10^{-2} [KB14]. These settings were tuned to achieve best validation loss for all methods.

After training with the torque limits $(-2, 2)$, one experiment tests a held-out test set with the same torque limits of $(-2, 2)$, while another experiment tests whether policies can handle a distribution shift with new torque limits of $(-1, 1)$.

Unless otherwise stated, the loss function measured in experiments is the mean squared error (MSE) between the expert model predictive controller's controls and the policy's controls: $\frac{1}{n}\sum_{i=1}^{n}(\mathbf{u}_i - \hat{\mathbf{u}}_i)^2$ for a batch of n trajectories, where $\hat{\mathbf{u}}$ is the policy's controls. For KCPO, which has an autoencoder component, the loss function for measuring reconstruction error is MSE between the original input state and the reconstructed output state: $\frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \hat{\mathbf{x}}_i)^2$ for a batch of n states, where $\hat{\mathbf{x}}$ is the autoencoder's reconstruction of the state $\mathbf{x}$.

4.3 Results

4.3.1 Experiment A: Generalization to held-out test set

Figures 4.4 and 4.5 display testing imitation loss (with a zoomed-in view of the last twenty epochs) for Simple Pendulum and Cartpole, respectively, distributed over ten trials. The X-axis is time measured in epochs, and the Y-axis is loss measured in MSE. For these trials, the test set used bounds of $(-2, 2)$, which all the methods were trained with. Though the error bars overlap, the clear trend is that KCPO is competitive with the baselines and tends to have lower imitation loss.

4.3.2 Experiment B: Generalization to unseen box constraints

For this experiment, the policies are all provided with a new set of box constraints: $(-1, 1)$. Figures 4.6 and 4.7 show testing and validation imitation loss for Simple Pendulum and Cartpole, respectively. The X-axis is time measured in epochs, and the Y-axis is loss measured in MSE. For the baselines, the hyperbolic tangent squashing functions are rescaled, but the neural network weights remain frozen from the training phase. Meanwhile, for KCPO, the new box constraints are provided directly to MPC, and KCPO's neural network weights are also unchanged.

The results show that the baselines are less effective than KCPO at overcoming generalizing after a distribution change from the box constraints of (-2, 2) to (-1, 1). This is perhaps because KCPO's architecture has an MPC layer mirroring that of the expert, while the model-free neural networks lack a similar inductive bias. For the Simple Pendulum, KCPO has the lowest imitation loss. For Cartpole, error bars overlap; however, like with the $(-2, 2)$ constraints, the trend is that KCPO is competitive with the baselines and tends to have lower imitation loss.

4.3.3 Training and Validation

The validation and training loss curves for the $(-2, 2)$ constraints are included for completeness in Figures 4.8, 4.9, 4.10, and 4.11. For these figures, as in the other loss curve figures, the X-axis is time measured in epochs, and the Y-axis is loss measured in MSE.

4.3.4 Speed

While the matrix multiplication of neural nets is fast, the bottleneck of KCPO is MPC, which is slow even for inference. The baselines have the clear advantage in training and inference time. Due to this bottleneck, an MPC horizon of 10 was used during experiments instead of a longer horizon during both training and inference. Future work could improve this bottleneck and enable longer horizons to be tractably used.

The speed of training and inference for one batch was timed for both the Simple Pendulum and Cartpole environments using each model with horizon $T = \{10, 20\}$ for ten different batches. The results of the Cartpole timing experiment are shown in Figure 4.12. In this figure, the X-axis is separated into categories for training and inference forward passes of each method (KCPO, RNN, and ReflexNet), and the Y-axis is the logarithm of time in seconds per forward pass of a single batch of data. A black line is drawn where the time reaches on second to demonstrate when a given controller's speed becomes far too slow for real-time control.

For inference, it takes an entire second of wall clock time to do inference on a KCPO network trained for control on a horizon of 20. This result makes it currently impractical to use KCPO for real-time control, but with an improvement in speed in future work, the superior control performance of KCPO compared to the baseline would become an asset.

4.3.5 KCPO's Autoencoder Reconstruction Loss

The autoencoder reconstruction loss curves, while important for KCPO's learning performance, cannot be compared against the non-autoencoding baselines, but they are included for completeness in Figure 4.13.

4.3.6 Comparison of KCPO with Related Prior Works

Table 4.1 contrasts KCPO with related prior works by feature. KCPO combines many of the advantages of neural network-based policies with the hard constraint satisfaction guarantees of classical control methods using numerical optimization. KCPO is a model-based policy learning method with hard constraints that trains end-to-end with backpropagation and has guaranteed stable training, unlike [Amo+18] that KCPO built upon, because it uses a linear dynamics model. Because the RNN and modified [Kur+22] baselines are model-free, they lack the MPC inductive bias of KCPO, perhaps explaining why KCPO is a superior controller. [YWK22], which inspired this work in being the first end-to-end Koopman model-based policy training method, lacks the ability to impose hard constraints because it relies on unconstrained LQR.

The Koopman Differentiable Predictive Control architecture of [Kin+22] improves upon [YWK22] by having constraints, but they are probabilistic instead of hard, and thus the architecture is safe to use only with offline training. Another issue with Koopman Differentiable Predictive Control is that it is not trained end-to-end, requiring that the Koopman dynamics model be pre-trained. Finally, the original Constrained Policy Optimization introduced in [Ach+17] was a watershed work in constrained reinforcement learning, but it also adopts the approach of pursuing probabilistic (not hard) constraints like [Kin+22]. This probabilis-

tic constraint paradigm restricts the usage of architectures like [Kin+22] and [Ach+17] to be safely trained only offline.

Table 4.1: Method Comparison

Feature	*This Work*	Baselines	[Amo+18]	[YWK22]	[Ach+17]	[Kin+22]
Model-Based/Model-Free	Model-Based	Model-Free	Model-Based	Model-Based	Model-Free	Model-Based
Linear Dynamics Model?	✓	✗	✗	✓	✗	✓
End-to-End Training?	✓	✓	✓	✓	✓	✗
Hard Constraints During Training and Testing?	✓	✓	✓	✗	✗	✗
Stable Training?	✓	✓	✗	✓	✓	✓

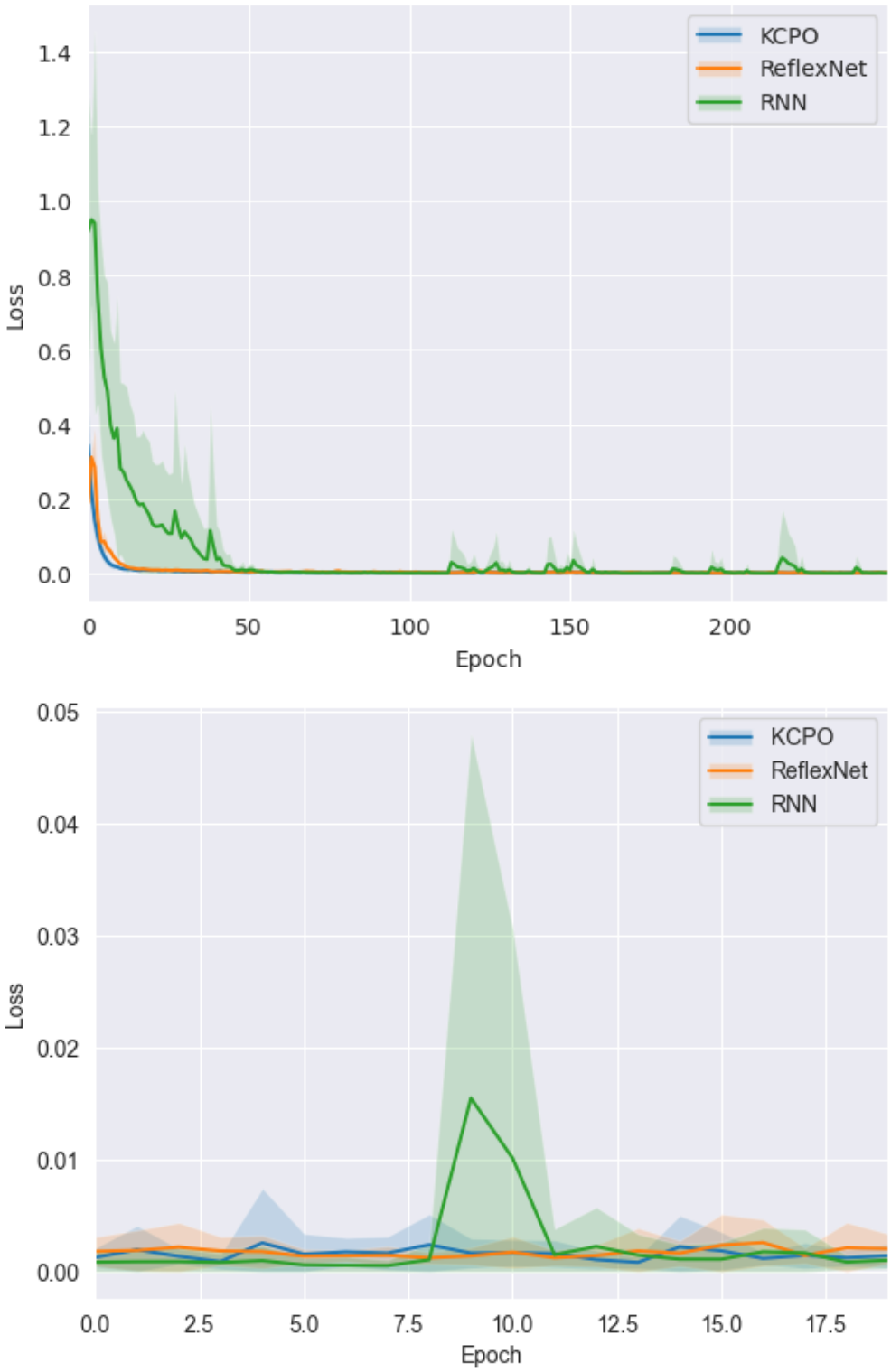

Figure 4.4: Simple Pendulum - Test Imitation Loss (-2, 2). Full and Last 20 Epochs. Lower loss is better.

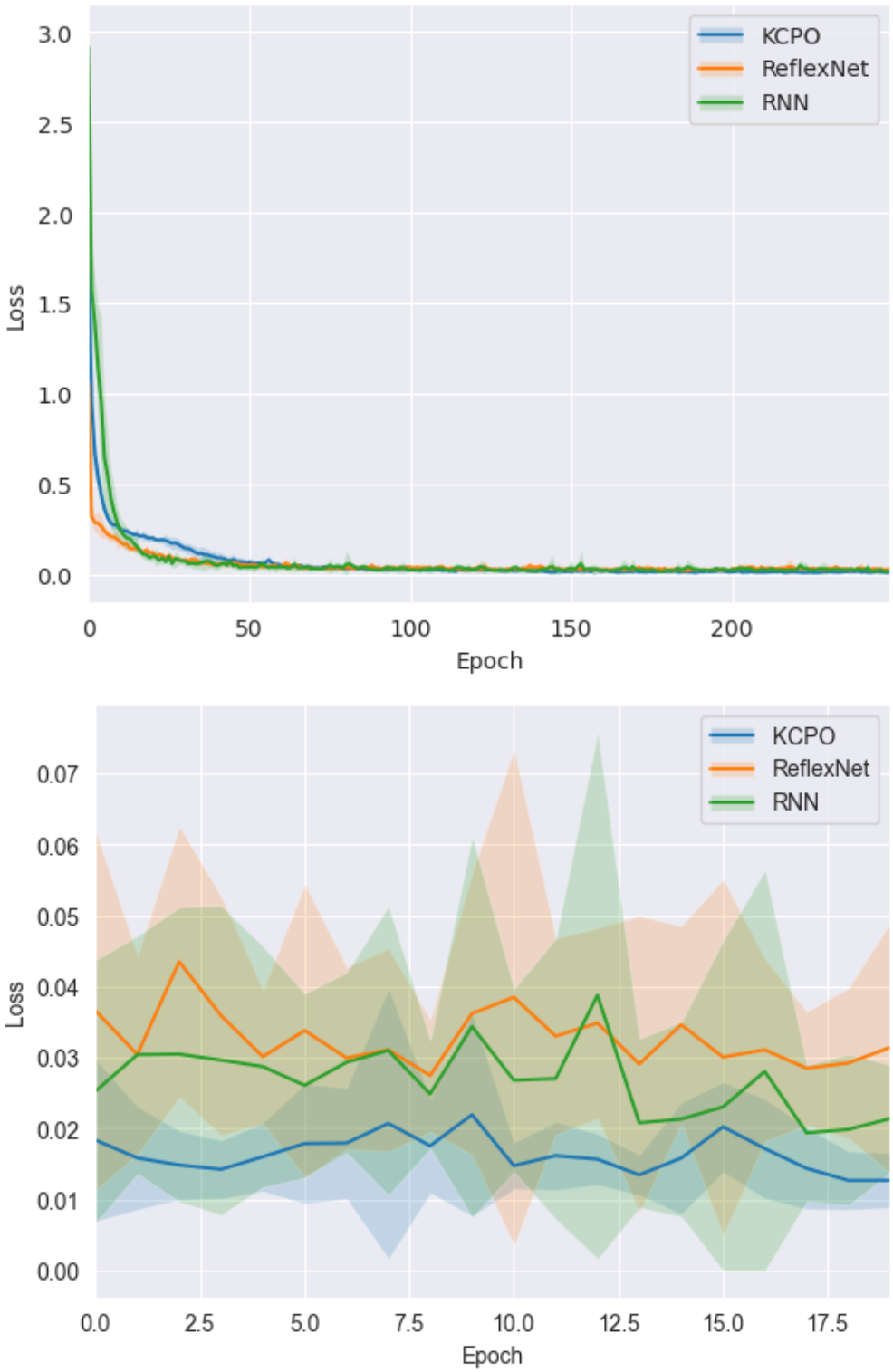

Figure 4.5: Cartpole - Test Imitation Loss (-2, 2). Full and Last 20 Epochs. Lower loss is better.

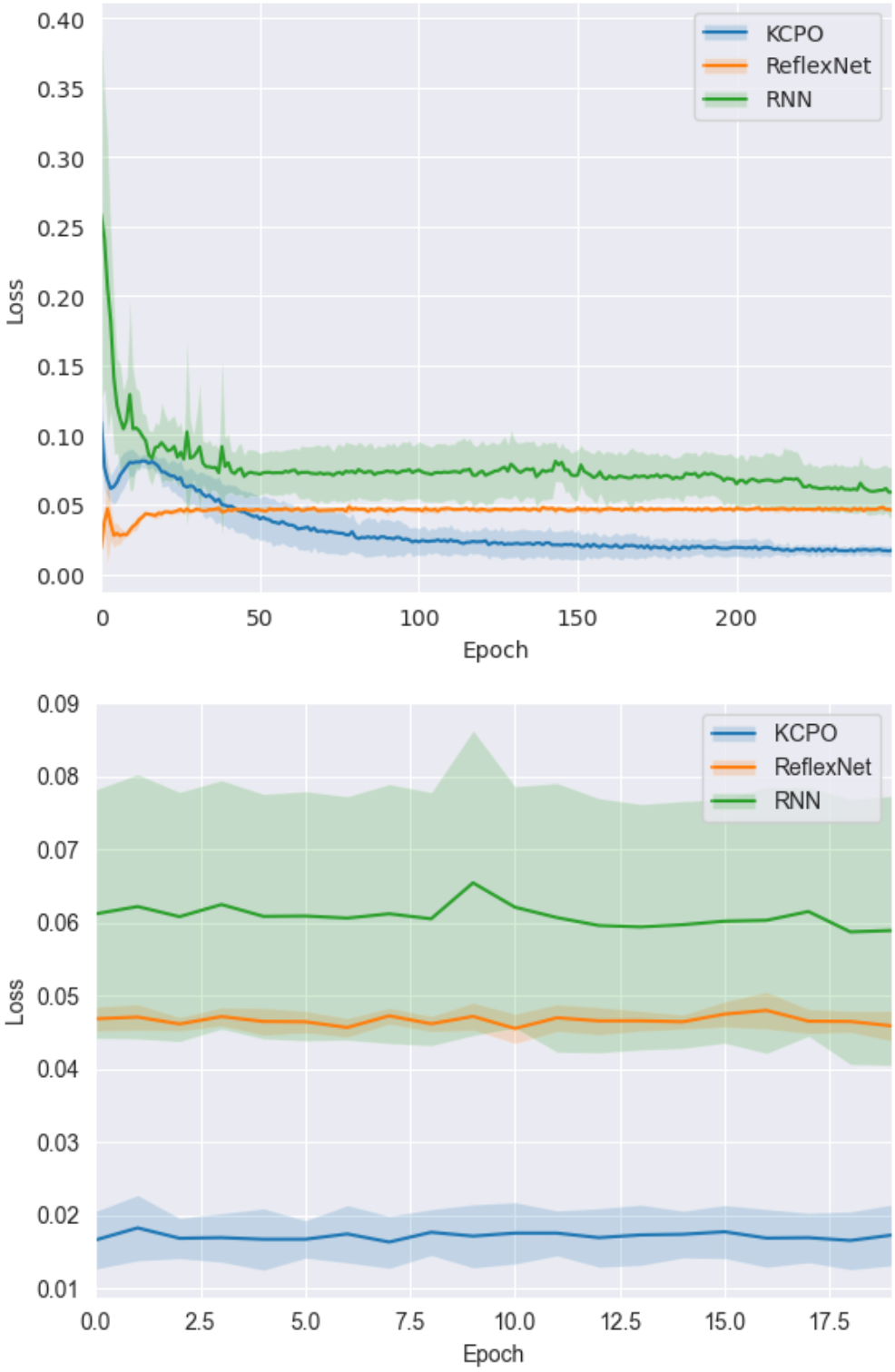

Figure 4.6: Simple Pendulum - Test Imitation Loss (-1, 1). Full and Last 20 Epochs. Lower loss is better.

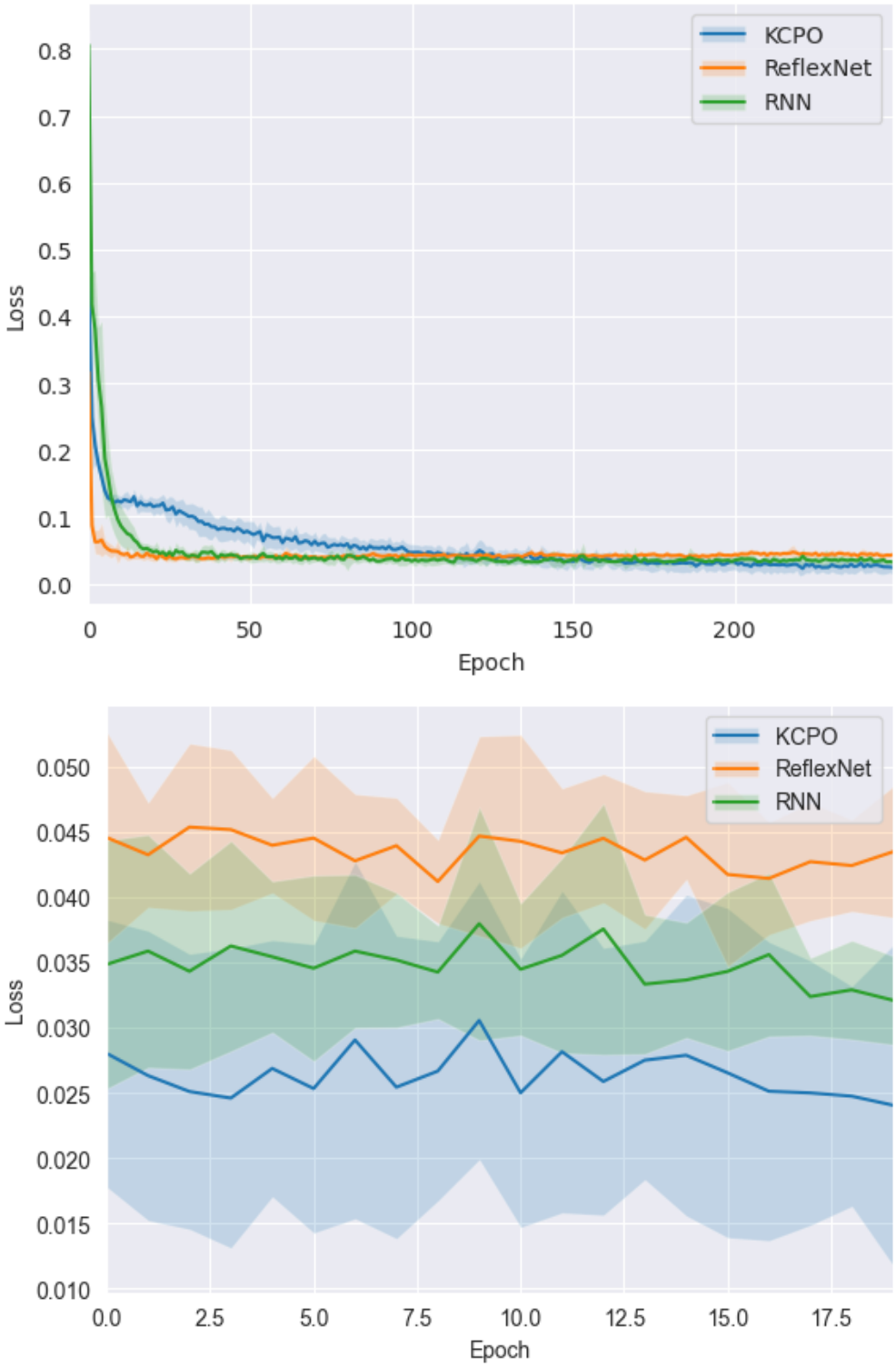

Figure 4.7: Cartpole - Test Imitation Loss (-1, 1). Full and Last 20 Epochs. Lower loss is better.

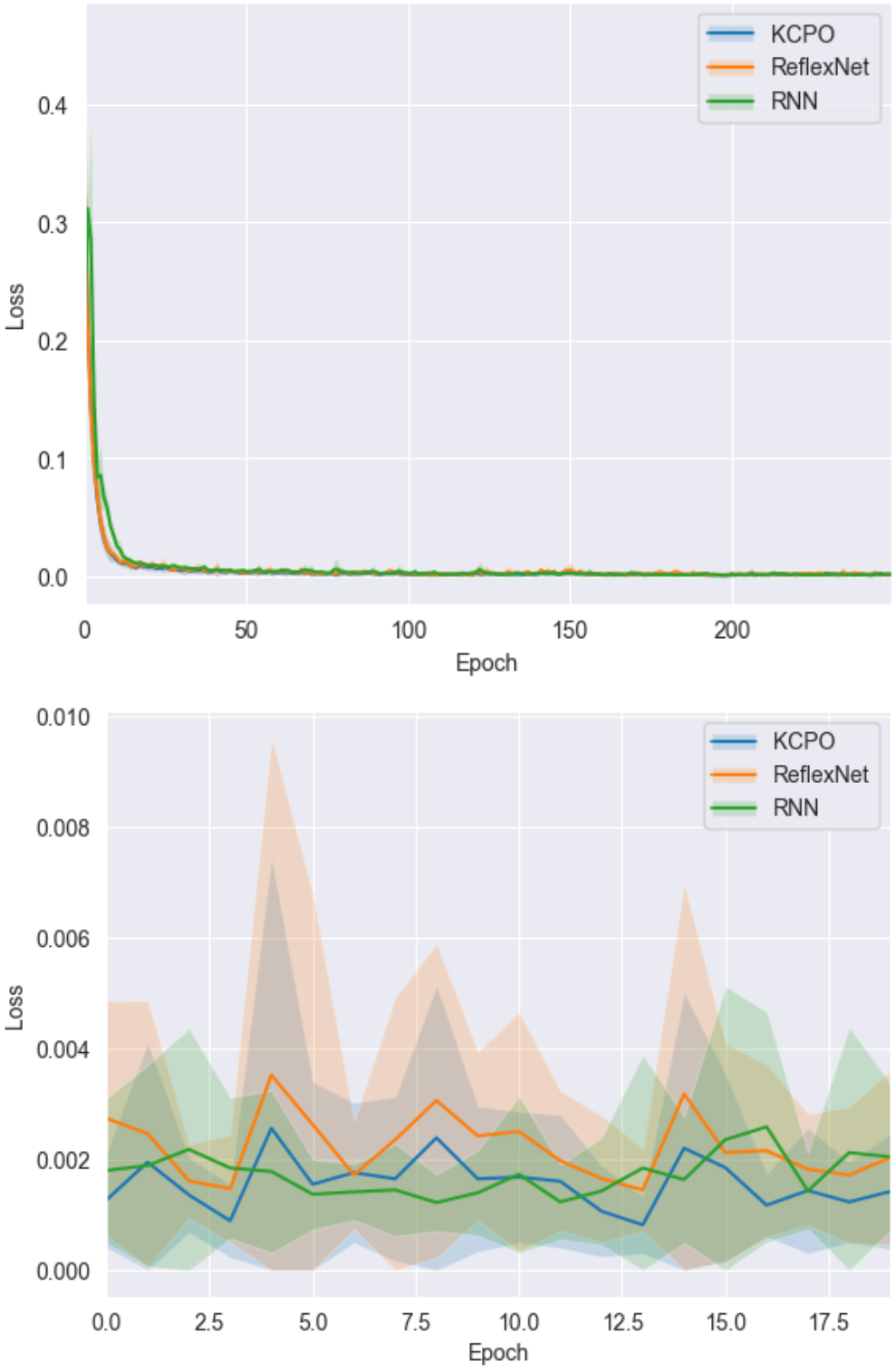

Figure 4.8: Pendulum - Training Imitation Loss (-2, 2). Full and Last 20 Epochs. Lower loss is better.

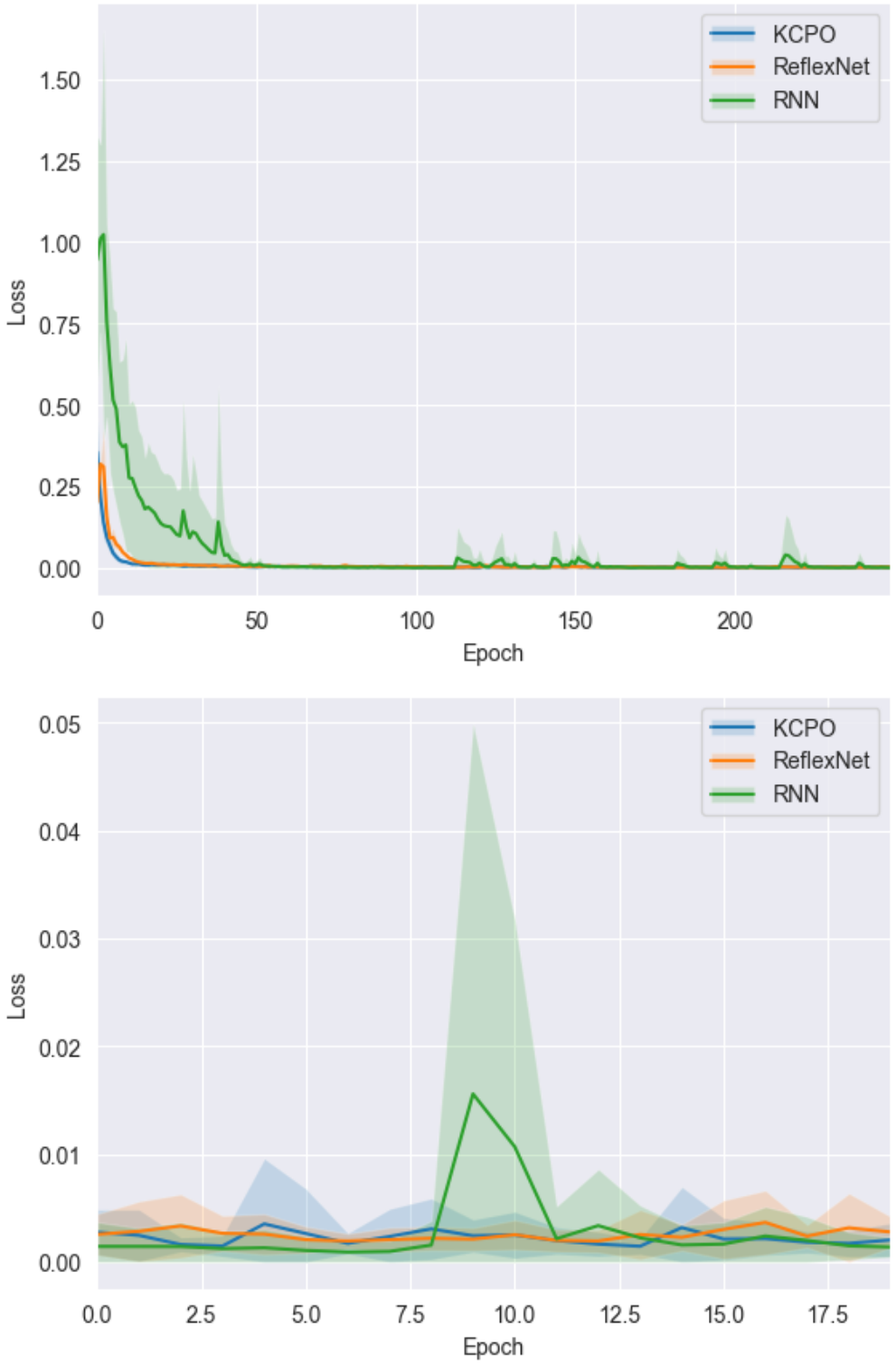

Figure 4.9: Pendulum - Validation Imitation Loss (-2, 2). Full and Last 20 Epochs. Lower loss is better.

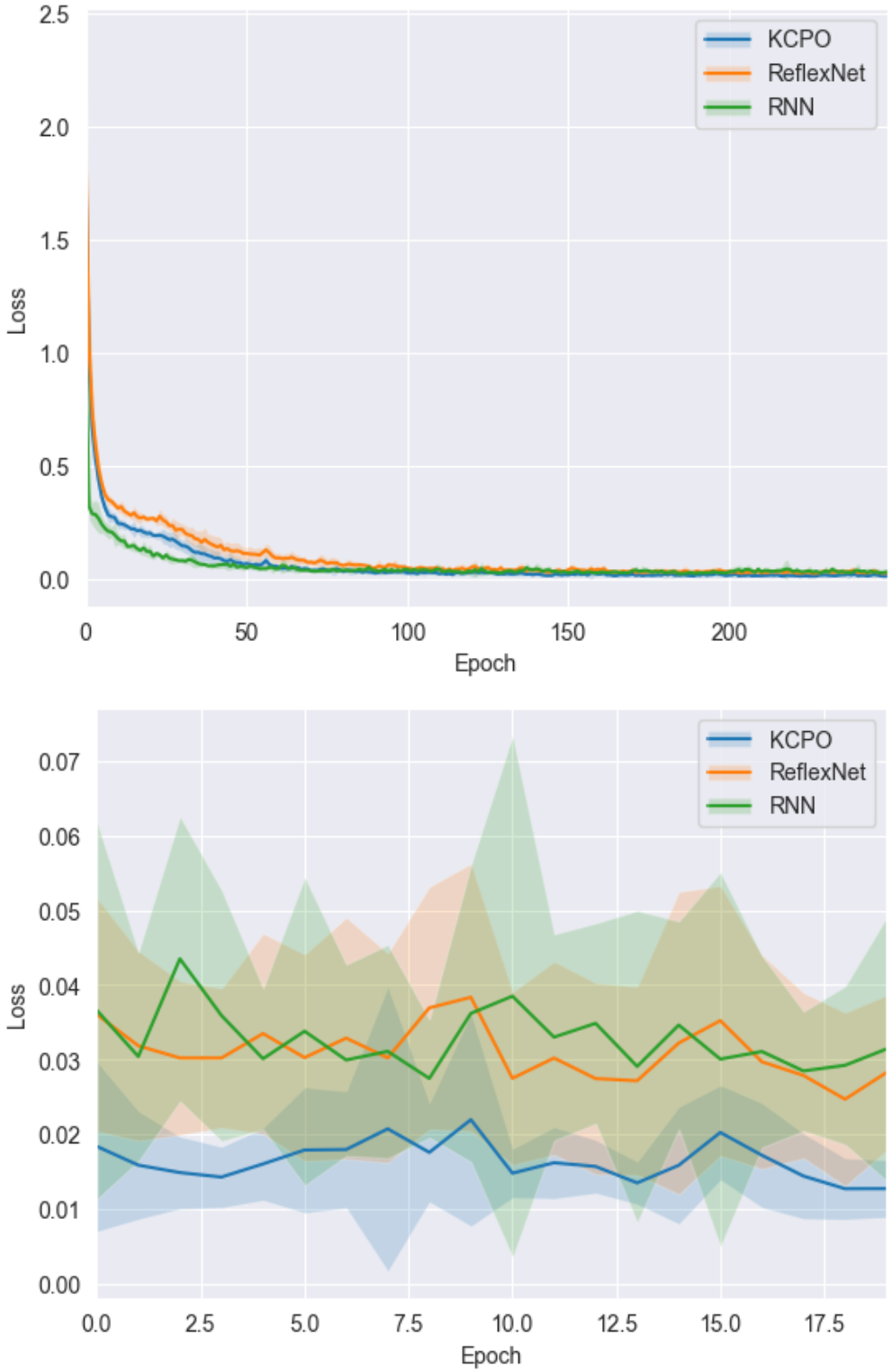

Figure 4.10: Cartpole - Training Imitation Loss (-2, 2). Full and Last 20 Epochs. Lower loss is better.

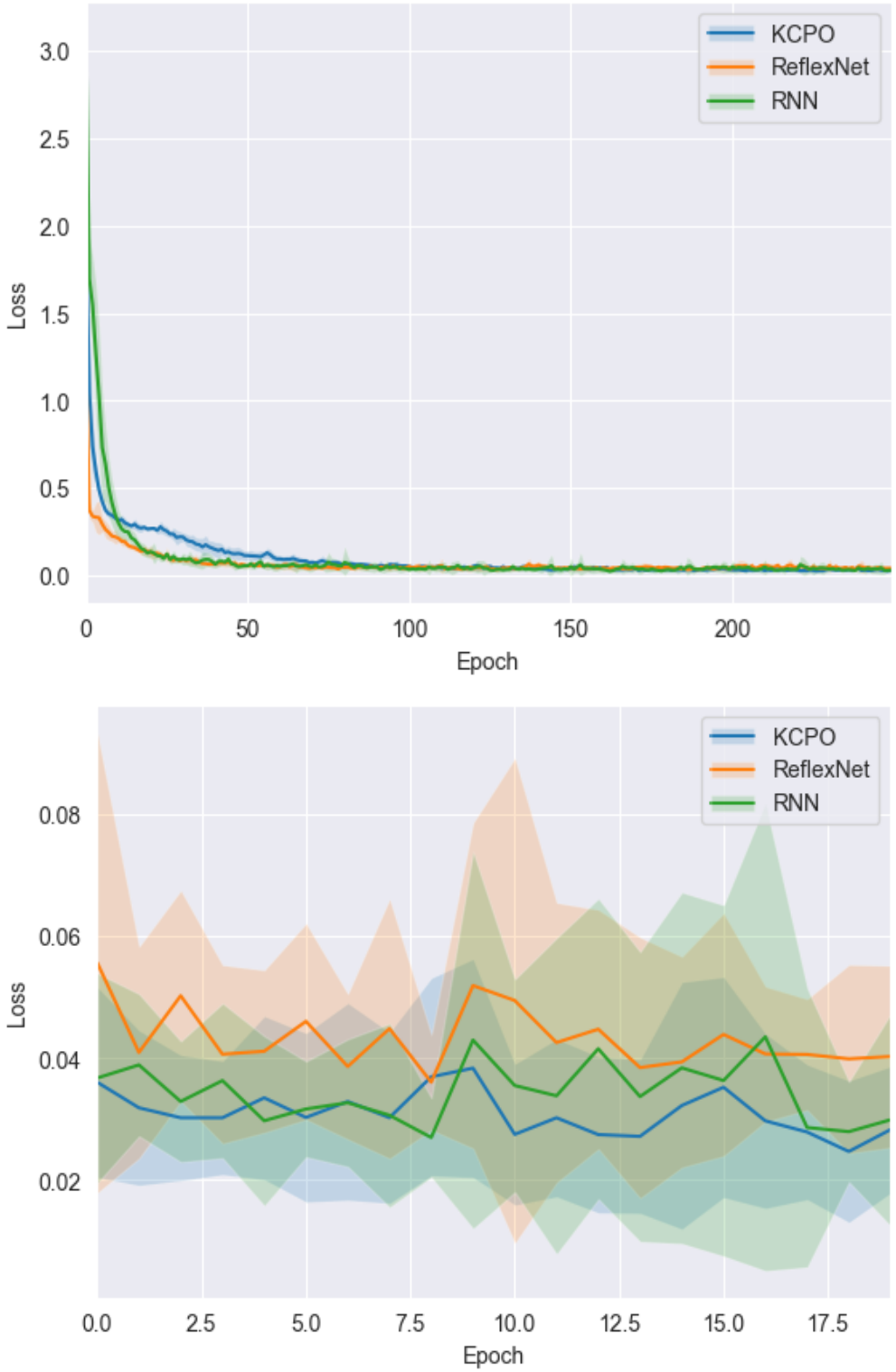

Figure 4.11: Cartpole - Validation Imitation Loss (-2, 2). Full and Last 20 Epochs. Lower loss is better.

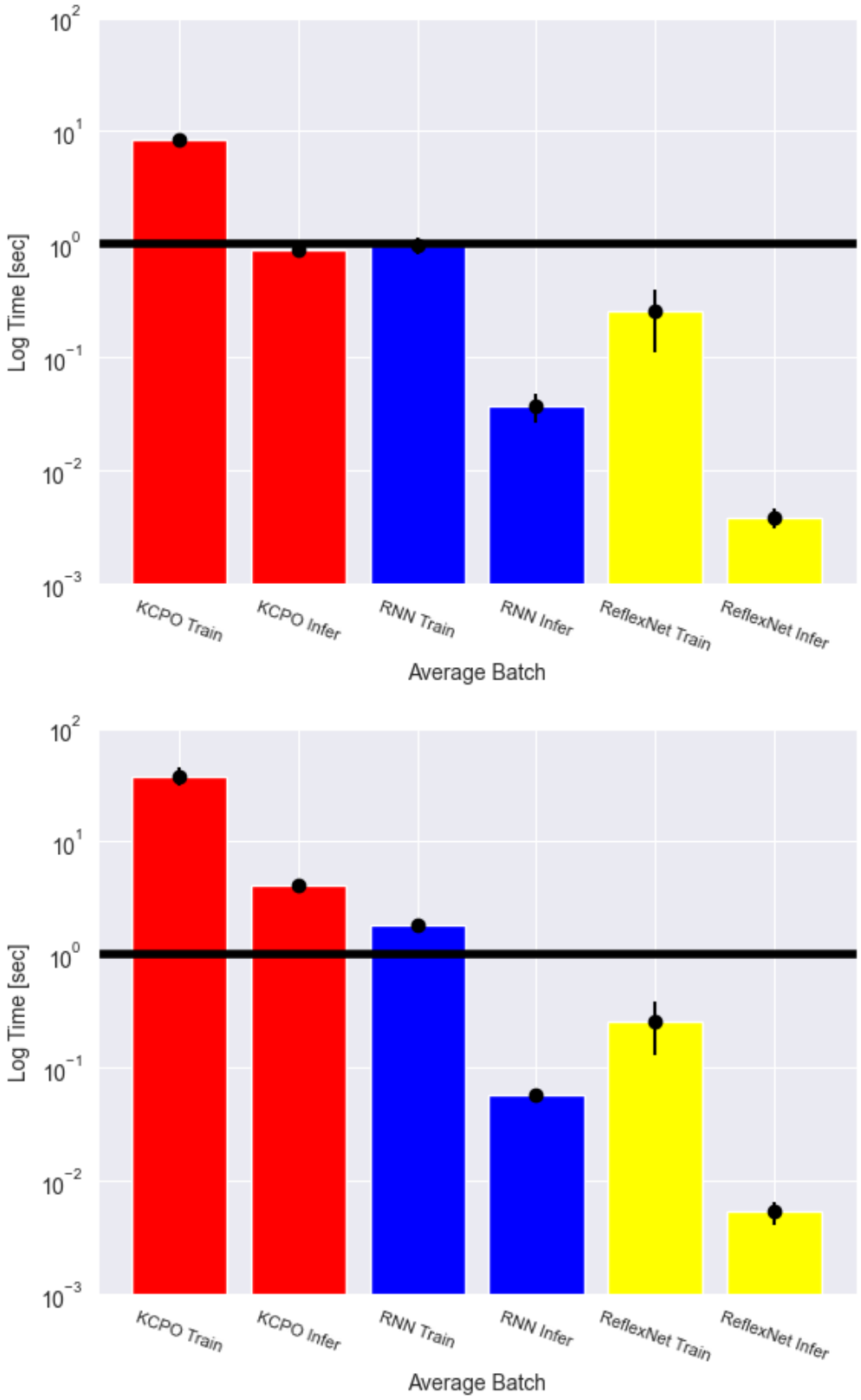

Figure 4.12: Time for batch inference in Cartpole. Top: **Horizon = 10**. Bottom: **Horizon = 20**. Black line drawn at one second. Faster (lower) is better.

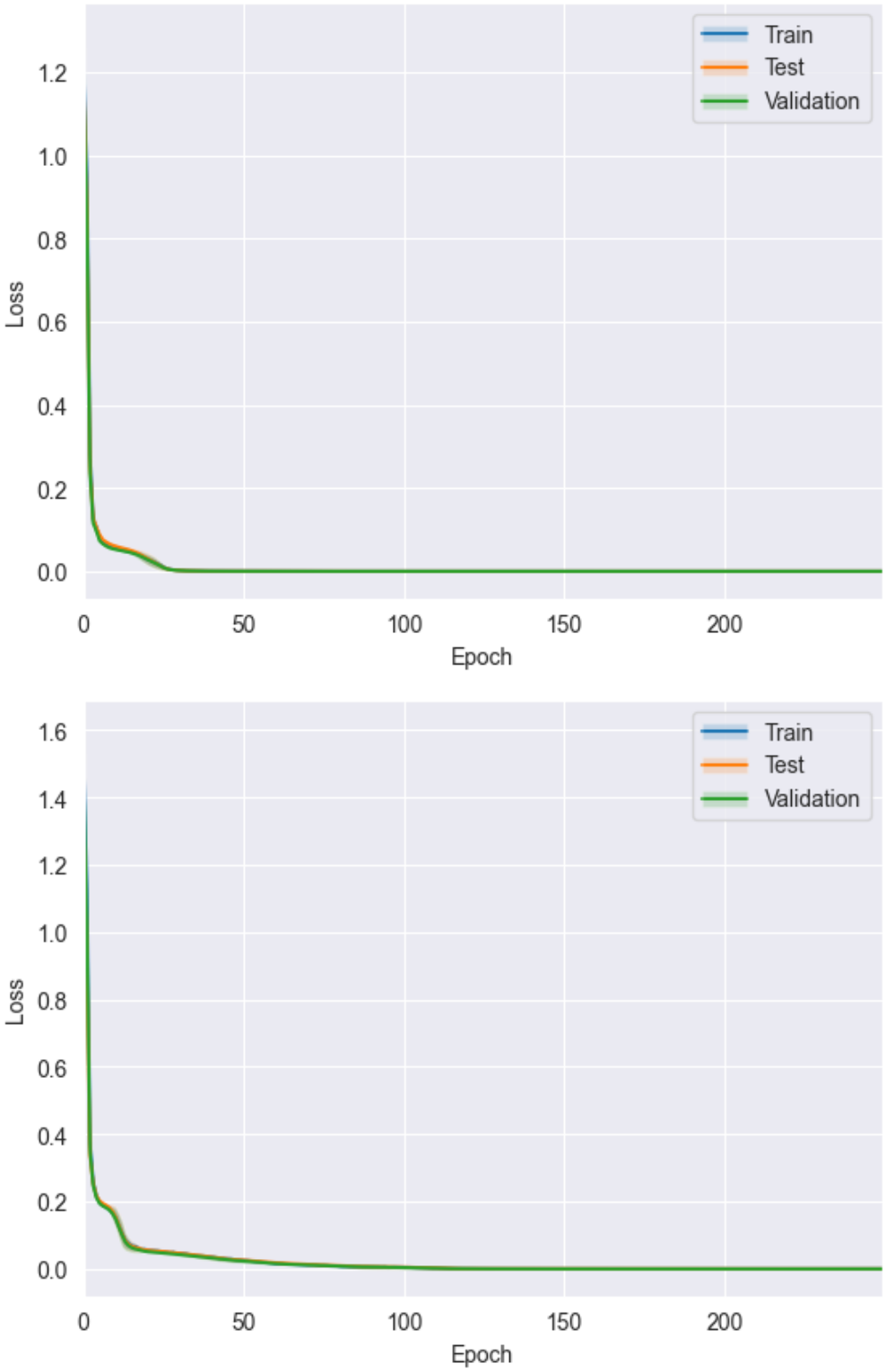

Figure 4.13: Top: Pendulum KCPO Reconstruction Loss. Bottom: Cartpole KCPO Reconstruction Loss. Lower loss is better.

5 Conclusion

In this thesis, a new method was presented for constrained policy optimization in unknown, highly nonlinear systems: Koopman Constrained Policy Optimization (KCPO). As noted in the Background, the paradigm of end-to-end differentiable Koopman policies originated with [YWK22], and this work's primary contribution has been to extend the power of the Koopman policy to tasks requiring hard box constraints on controls. This becomes useful for reasons of safety, when it would be self-destructive to the robot hardware or dangerous for humans if a robot's torque exceeded the real-life limitations of its hardware. Being able to impose hard box constraints on torque throughout training and inference phases ensures the safe operation of the robot.

The original Koopman policy used analytical Riccati solving to obtain optimal controls with their dynamics model, but this will not work when imposing box constraints. A numerical trajectory optimization method like Box-DDP is more appropriate. However, these optimization methods are not easily or efficiently differentiable. In this work, we proposed to retain end-to-end differentiability in a box-constrained Koopman policy by replacing Riccati solving with the implicitly differentiable constrained MPC layer of [Amo+18].

The results shown in this thesis demonstrate that KCPO can train with a guarantee to never violate constraints, and it can successfully change constraints at inference time without any retraining. KCPO beats baseline methods in the Simple Pendulum environment, while it is competitive in the Cartpole environment.

5.1 Future Extensions

The Koopman operator is an infinite-dimensional linear operator that exactly characterizes the evolution of an autonomous dynamical system, i.e. a system that is not actuated. However, in practice, prior literature has succeeded in applying Koopman autoencoders to optimal control settings, the control of non-autonomous systems. My own results extend this trend.

There are also potential limitations to the use of discrete time dynamics modeling that most recurrent neural networks use, including my own. Recent work on continuous-time neural networks has shown great promise for superior learning in time series applications, so an exciting direction could be to explore the power of these new recurrent architectures within the neural network-based optimal control paradigm my thesis has explored [Has+22].

One major limitation of KCPO is that it can only apply box constraints to controls. There are two approaches whereby KCPO could be extended to introduce state constraints.

By concatenating the state variables with the latent embedding vector z, following the example of [SM22], state constraints could be easily imposed on the trajectory optimization. However, one may expect for the dynamics model's accuracy to suffer with this design, so there is a trade-off to consider. Another approach to achieving state constraints with the Koopman framework can be found in [KM18] and [HEK22], who use a linear decoder. The trade-off with this approach is that a linear decoder cannot match the reconstruction error of the true nonlinear decoding, also damaging the dynamics model's accuracy.

As mentioned in the Background, DeSKO is a Koopman MPC method that, like a variational autoencoder, deterministically maps to the parameters for a random distribution over latent variables instead of mapping to latent variables directly. A future extension to KCPO could borrow aspects of the DeSKO design to achieve stochastic control robust to random perturbations in the dynamics. However, because LQR is already optimal for stochastic dynamics with a Gaussian distribution, KCPO is actually an optimal controller for that particular application of stochastic control [Tedd].

Finally, the timing experiments showed that KCPO is much slower than the baselines. Future work remains to improve KCPO's training and inference speed to enable real-time control.

5.1.1 From Box Constraints to Arbitrary Nonlinear Constraints

Future work could extend KCPO's scope beyond box constraints for controls by replacing the Box-DDP trajectory optimization algorithm within the differentiable MPC layer with the Augmented Lagrangian TRajectory Optimizer (ALTRO), which like Box-DDP is also based on iLQR. ALTRO retains the main advantage of iLQR, namely strict dynamic feasibility through optimization, but it expands on iLQR with the augmented Lagrangian method to support arbitrary nonlinear constraints [HJM19].

As long as those nonlinear constraints and the objective function remain convex, the optimizer will provably converge with enough iterations, just as Box-DDP does. Provided that ALTRO converges to a fixed point, the Implicit Function Theorem could be used in the same manner to differentiate through the ALTRO solver for backpropagation. Due to ALTRO's heritage from iLQR, it requires linearization of its dynamics, making KCPO a complementary fit.

Acronyms

ALTRO	Augmented Lagrangian TRajectory Optimizer
Box-DDP	Box-Dynamic Differential Programming
CMDP	Constrained Markov Decision Process
CPO	Constrained Policy Optimization
DARE	Discrete Algebraic Riccati Equation
DDP	Dynamic Differential Programming
DeSKO	Deep Stochastic Koopman Operator
DMD	Dynamic Mode Decomposition
eDMD	Extended Dynamic Mode Decomposition
IFT	Implicit Function Theorem
iLQR	Iterative Linear Quadratic Regulator
JNF	Jordan Normal Form
KCPO	Koopman Constrained Policy Optimization
LQR	Linear Quadratic Regulator
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
MLP	Multilayer Perceptron
MPC	Model Predictive Control
RNN	Recurrent Neural Network
SNOPT	Sparse Nonlinear OPTimizer
UAT	Universal Approximation Theorem

Bibliography

[Ach+17] J. Achiam, D. Held, A. Tamar, and P. Abbeel. "Constrained Policy Optimization". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017.

[Amo+18] B. Amos, I. Jimenez, J. Sacks, B. Boots, and J. Z. Kolter. "Differentiable MPC for End-to-end Planning and Control". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/ba6d843eb4251a4526ce65d1807a9309-Paper.pdf.

[Aze+21] O. Azencot, N. B. Erichson, V. Lin, and M. W. Mahoney. "Forecasting sequential data using consistent Koopman autoencoders". In: *Advances in Neural Information Processing Systems*. Vol. 34. 2021.

[Bru] S. L. Brunton. *Notes on Koopman Operator Theory*. URL: https://fluids.ac.uk/files/meetings/KoopmanNotes.1575558616.pdf.

[Bru+16] S. L. Brunton, B. W. Brunton, J. L. Proctor, and J. N. Kutz. "Koopman Invariant Subspaces and Finite Linear Representations of Nonlinear Dynamical Systems for Control". *PloS one* 11:2, 2016, e0150171. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0150171. URL: https://europepmc.org/articles/PMC4769143.

[Che] G. Chevalier. *The LSTM cell.png*. URL: https://en.wikipedia.org/wiki/File:Sintay_SVG.svg.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. Chap. 15.

[Drg+22] J. Drgona, K. Kis, A. R. Tuor, D. L. Vrabie, and M. Klauco. "Differentiable Predictive Control: Deep Learning Alternative to Explicit Model Predictive Control for Unknown Nonlinear Systems". *Journal of Process Control* 116, 2022, pp. 1–15.

[Eul48] L. Euler. *Introductio in analysin infinitorum*. Bernuset, Delamolliere, 1748.

[GBC16a] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 6.

[GBC16b] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 8.

[GBC16c] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 14.

[GF96] G. H. Golub and V. L. C. F. *Matrix computations*. Johns Hopkins University Press, 1996, pp. 147–147.

[GM21] E. Galván and P. Mooney. "Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges". *IEEE Transactions on Artificial Intelligence* 2:6, 2021, pp. 476–493. DOI: 10.1109/TAI.2021.3067574.

[GMS05] P. E. Gill, W. Murray, and M. A. Saunders. "SNOPT: An SQP algorithm for large-scale constrained optimization". *SIAM review* 47:1, 2005.

[Gu+22] S. Gu, L. Yang, Y. Du, G. Chen, F. Walter, J. Wang, Y. Yang, and A. Knoll. "A Review of Safe Reinforcement Learning: Methods, Theory and Applications". *arXiv preprint arXiv:2205.10330*, 2022.

[Had22] M. Hader. "Deep learning for Koopman operator approximations for control". Master's thesis. Johannes Kepler University Linz, 2022, pp. 4–7.

[Has+22] R. Hasani, M. Lechner, A. Amini, L. Liebenwein, A. Ray, M. Tschaikowski, G. Teschl, and D. Rus. "Closed-form continuous-time neural networks". *Nature Machine Intelligence* 4:11, 2022, pp. 992–1003. DOI: 10.1038/s42256-022-00556-7.

[HEK22] M. Han, J. Euler-Rolle, and R. K. Katzschmann. "DeSKO: Stability-Assured Robust Control with a Deep Stochastic Koopman Operator". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=hniLRD_XCA.

[HJ13] R. A. Horn and C. R. Johnson. *Matrix Analysis*. 2nd ed. Cambridge University Press, 2013.

[HJM19] T. A. Howell, B. E. Jackson, and Z. Manchester. "ALTRO: A fast solver for constrained trajectory optimization". *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019. DOI: 10.1109/iros40897.2019.8967788.

[HS97] S. Hochreiter and J. Schmidhuber. "Long short-term memory". *Neural Computation* 9:8, 1997, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". *Neural Networks* 2:5, 1989, pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

[Ika] IkamusumeFan. *Sintay_SVG.png*. URL: https://en.wikipedia.org/wiki/File:Sintay_SVG.svg.

[Jac] B. Jackson. *AL-iLQR Tutorial*. URL: https://bjack205.github.io/papers/AL_iLQR_Tutorial.pdf.

[Kas] L. Kassabian. *Phase Portraits of 2D Differential Systems*. URL: https://www.desmos.com/calculator/wcdzf0ksuq.

[KB14] D. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". *ArXiv* abs/1412.6980, 2014.

[KD] Z. Kolter and D. Duvenaud. *Chapter 5: Differentiable Optimization*. URL: http://implicit-layers-tutorial.org/differentiable_optimization/.

[Kin+22] E. King, J. Drgona, A. R. Tuor, S. G. Abhyankar, C. Bakker, A. Bhattacharya, and D. L. Vrabie. "Koopman-based Differentiable Predictive Control for the Dynamics-Aware Economic Dispatch Problem". In: *American Control Conference (ACC 2022)*. IEEE. 2022. DOI: 10.23919/ACC53348.2022.9867379.

[KM18] M. Korda and I. Mezić. "Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control". *Automatica* 93, 2018, pp. 149–160. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2018.03.046.

[KP02] S. G. Krantz and H. R. Parks. *The implicit function theorem: History, theory, and applications*. Birkhäuser, 2002.

[Kur+22] V. Kurtz, H. Li, P. M. Wensing, and H. Lin. "Mini Cheetah, the falling cat: A case study in machine learning and trajectory optimization for robot acrobatics". *International Conference on Robotics and Automation (ICRA)*, 2022. DOI: 10.1109/icra46639.2022.9812120.

[Lay] G. C. Layek. *An Introduction to Dynamical Systems and Chaos*. Springer, p. 1.

[Lev+20] S. Levine, A. Kumar, G. Tucker, and J. Fu. *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems*. 2020. DOI: 10.48550/ARXIV.2005.01643. URL: https://arxiv.org/abs/2005.01643.

[Lev21] S. Levine. *Deep Reinforcement Learning: Introduction to Reinforcement Learning*. 2021.

[Li+20] Y. Li, H. He, J. Wu, D. Katabi, and A. Torralba. "Learning Compositional Koopman Operators for Model-Based Control". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=H1ldzA4tPr.

[LKB18] B. Lusch, J. N. Kutz, and S. L. Brunton. "Deep Learning for Universal Linear Embeddings of Nonlinear Dynamics". *Nature Communications* 9:1, 2018. DOI: 10.1038/s41467-018-07210-0.

[MWK19] J. Morton, F. D. Witherden, and M. J. Kochenderfer. "Deep Variational Koopman Models: Inferring Koopman Observations for Uncertainty-Aware Dynamics Modeling and Control". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. 2019, pp. 3174–3180.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc. 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Sal18] R. Salakhutdinov. *Deep Reinforcement Learning and Control: Optimal control, trajectory optimization*. 2018.

[SB20a] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. 2nd ed. The MIT Press, 2020. Chap. 3.

[SB20b] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. 2nd ed. The MIT Press, 2020. Chap. 8.

[Sha48] C. E. Shannon. *A Mathematical Theory of Communication*. The Bell System Technical Journal, 1948.

[SM22] H. Shi and M. Q.-H. Meng. "Deep Koopman operator with control for Nonlinear Systems". *IEEE Robotics and Automation Letters* 7:3, 2022, pp. 7700–7707. DOI: 10.1109/lra.2022.3184036.

[Str39] I. Stravinsky. *Poetics of music in the form of six lessons*. 1939.

[Teda] R. Tedrake. *Underactuated Robotics*. Chap. 8: Linear Quadratic Regulators.

[Tedb] R. Tedrake. *Underactuated Robotics*. Chap. 10: Trajectory Optimization.

[Tedc] R. Tedrake. *Underactuated Robotics*. Chap. 11: Policy Search.

[Tedd] R. Tedrake. *Underactuated Robotics*. Chap. 14: Robust and Stochastic Control.

[TMT14] Y. Tassa, N. Mansard, and E. Todorov. "Control-limited differential dynamic programming". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 1168–1175.

[Wei+22] M. Weissenbacher, S. Sinha, A. Garg, and K. Yoshinobu. "Koopman Q-learning: Offline Reinforcement Learning via Symmetries of Dynamics". In: *Proceedings of the 39th International Conference on Machine Learning*. Ed. by K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato. Vol. 162. Proceedings of Machine Learning Research. PMLR, 2022, pp. 23645–23667. URL: https://proceedings.mlr.press/v162/weissenbacher22a.html.

[WTL23] Z. Wang, J. Tan, and C. K. Liu. *Koopman Operators for Modeling and Control of Soft Robotics*. 2023. arXiv: 2301.09708 [cs.SY].

[YWK22] H. Yin, M. C. Welle, and D. Kragic. "Embedding Koopman Optimal Control in robot policy learning". *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022. DOI: 10.1109/iros47612.2022.9981540.