Combining model-based RL with Learned Dynamics for Solving Continuos Control Problems

Ganga Nair B
M. Tech in Robotics and Autonomous Systems
Indian Institute of Science, Bengaluru
ganganairb@iisc.ac.in

Abstract

We explore the implementation of an Iterative Linear Quadratic Regulator (iLQR)-based reinforcement learning algorithm to solve benchmark control tasks from the OpenAI Gym suite. Specifically, we evaluate its performance in the MountainCarContinuous-v0 environment. Further, as iLQR traditionally requires access to accurate system dynamics, we address this limitation by introducing a neural network (NN) to learn the environment's dynamics from data. The learned model is then integrated into the iLQR framework to enable model-based policy optimization in an unknown environment. This hybrid approach combines the benefits of deterministic optimization offered by iLQR with the flexibility of data-driven modeling, demonstrating promising results in environments with initially unknown dynamics.

1 Introduction

Reinforcement Learning (RL) and optimal control share the common goal of optimizing decision-making through environment interactions. The Iterative Linear Quadratic Regulator (iLQR), an algorithm like many others, borrowed by RL from the world of optimal control, extends the classic LQR to nonlinear systems and to time varying dynamic equations. But it requires known dynamics. We address this limitation by integrating iLQR with neural network-learned dynamics, creating a model-based RL approach that maintains iLQR's optimization benefits while learning environment dynamics. Our method shows promising results on OpenAI Gym benchmarks, offering practical advantages for real-world applications where system models are unavailable.

1.1 Problem Statement

This project aims to investigate the application of the Iterative Linear Quadratic Regulator (iLQR) for solving classic continuous control tasks from the OpenAI Gym suite. We first implement the iLQR algorithm in the `MountainCarContinuous-v0` environment to evaluate its performance when the dynamics are known. To extend this approach to environments where the system model is unknown, we train a neural network to learn the environment's dynamics from data and use this learned model within the iLQR framework. We also apply this method to a custom Swing-Up Cart-Pole task and evaluate its effectiveness while identifying potential limitations and challenges associated with model learning and control performance.

2 Related Work

The iLQR algorithm was introduced by Li and Todorov [2], extending classical LQR to nonlinear systems via local linearization of dynamics and quadratization of the cost function around a nominal

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

trajectory. By iteratively solving local LQR subproblems, iLQR became a foundational method in trajectory optimization for control and robotics.

Recent work has focused on bridging model-based control with reinforcement learning (RL). Zong et al. [6] combined iLQR with an actor-critic agent trained in parallel, while Amos et al. [1] proposed a differentiable MPC framework embedded in neural networks, allowing end-to-end gradient-based learning. Similarly, Pan et al. [3] introduced Adaptive Constrained iLQR, integrating learned dynamics and safety constraints for autonomous driving.

Two key directions have emerged in integrating neural networks with iLQR: (1) differentiable control layers, as in [1, 4], and (2) neural approximations of components like gradients or trajectories, as in Neural iLQR by Yin et al. [5]. The latter embeds iLQR in a hybrid framework using shallow networks (2–3 layers) to model dynamics, alternating offline between iLQR updates and policy learning for stability.

In contrast, our work proposes an online method where iLQR optimization and neural dynamics learning occur jointly in a loop. This addresses the limitations of prior batch-based approaches by enabling continuous adaptation and tighter feedback between learning and control.

3 Background

3.1 Iterative Linear Quadratic Regulator

The iLQR framework used to find optimum action in RL is derived from the concepts of Linear Quadratic Regulator, Dynamic Programming, and further adapted to nonlinear systems. In this section, we will look at the problem setup and formulation for LQR to understand its implementation.

3.1.1 LQR Formulation

The LQR framework assumes linear time-varying system dynamics of the form:

$$\mathbf{x}_{t+1} = \mathbf{F}_t \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix} + \mathbf{f}_t \tag{1}$$

where $\mathbf{x}_t$ represents the state and $\mathbf{u}_t$ the control input at time t. The associated cost function is quadratic and defined as:

$$c(\mathbf{x}_t, \mathbf{u}_t) = \frac{1}{2} \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix}^T \mathbf{C}_t \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix} + \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix}^T \mathbf{c}_t \tag{2}$$

The objective of LQR is to minimize the cumulative cost:

$$\min_{\mathbf{u}_0, \ldots, \mathbf{u}_{T-1}} \sum_{t=0}^{T-1} c(\mathbf{x}_t, \mathbf{u}_t) \tag{3}$$

subject to the system dynamics specified above.

3.1.2 Dynamic Programming Approach

To accommodate a time-varying linear system, LQR employs dynamic programming, proceeding backward from the terminal time step and finding the optimum action at each step.

At each step, it defines a value function V and an action-value function Q:

$$Q(x_{T-1}, u_{T-1}) = c(x_{T-1}, u_{T-1}) + V(x_T) \tag{4}$$

$$Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T C_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T c_{T-1} + V(x_T)$$

Action-value function is optimised by substituting $x_T = Ax_{T-1} + Bu_{T-1}$ and minimizing Q with respect to u_{T-1} (i.e., solving $\nabla Q = 0$). The optimal action can be computed in the form:

$$u_{T-1} = \mathbf{K}_{T-1} x_{T-1} + \mathbf{k}_{T-1} \tag{5}$$

These gain matrices are then used to recursively update the value function for the previous time steps.

3.1.3 Backward Pass Optimization

In each step, the action-value Q can be written in the form

$$Q(x_t, u_t) = \underbrace{\frac{1}{2}\begin{bmatrix} x_t \\ u_t \end{bmatrix}^T \begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu} \end{bmatrix} \begin{bmatrix} x_t \\ u_t \end{bmatrix}}_{\text{Quadratic term}} + \underbrace{\begin{bmatrix} Q_x & Q_u \end{bmatrix} \begin{bmatrix} x_t \\ u_t \end{bmatrix}}_{\text{Linear term}} + \text{const}$$

where,

$$Q_x = A_t^\top V_x + C_x, \quad Q_u = B_t^\top V_x + R u_t \tag{6}$$

$$Q_{xx} = A_t^\top V_{xx} A_t + C_{xx}, \quad Q_{uu} = B_t^\top V_{xx} B_t + R \tag{7}$$

$$Q_{ux} = Q_{xu} = B_t^\top V_{xx} A_t + C_{ux} \tag{8}$$

After optimisation to minimize the action-value function, the control law is determined from:

$$k_t = -Q_{uu}^{-1} Q_u, \quad K_t = -Q_{uu}^{-1} Q_{ux} \tag{9}$$

The value function is updated recursively using:

$$V_x \leftarrow Q_x + K_t^\top Q_{uu} k_t + K_t^\top Q_u \tag{10}$$

$$V_{xx} \leftarrow Q_{xx} + K_t^\top Q_{uu} K_t + K_t^\top Q_{ux} + Q_{ux}^\top K_t \tag{11}$$

3.1.4 iLQR: Extension to Nonlinear Systems

While LQR assumes linear system dynamics, many real-world systems are nonlinear. The Iterative Linear Quadratic Regulator (iLQR) extends the LQR approach to such cases by iteratively applying LQR to locally linearized dynamics.

In iLQR, the nonlinear system is defined by:

$$x_{t+1} = f(x_t, u_t), \quad c_t = \ell(x_t, u_t) \tag{12}$$

The dynamics are approximated using a first-order Taylor expansion:

$$\delta x_{t+1} = A_t \delta x_t + B_t \delta u_t \tag{13}$$

where $A_t = \nabla_{x_t} f(\hat{x}_t, \hat{u}_t)$ and $B_t = \nabla_{u_t} f(\hat{x}_t, \hat{u}_t)$.

Similarly, the cost is approximated with a second-order expansion:

$$\delta c_t = c_t^\top \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T C_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} \tag{14}$$

The linearized system then resembles the LQR structure, with:

$$A_t \leftarrow \text{Jacobian of } f, \quad C_t \leftarrow \text{Hessian of } \ell \tag{15}$$

3.1.5 iLQR Algorithmic Procedure

The iLQR method proceeds as follows:

1. Initialize with a nominal state-control trajectory $(\hat{x}_t, \hat{u}_t)$.
2. Linearize the dynamics and quadratize the cost about this nominal trajectory.
3. Solve the LQR problem on these approximations to compute control law updates.
4. Apply the updated control to generate a new nominal trajectory.
5. Repeat the process until convergence is achieved.

This iterative refinement allows iLQR to handle complex nonlinear control problems while maintaining computational efficiency.

3.2 Neural Networks for Dynamics Approximation

In scenarios where the system dynamics are unknown or too complex to model analytically, **neural networks (NNs)** offer a powerful alternative. Neural networks are highly expressive function approximators capable of modeling nonlinear relationships between inputs and outputs based on observed data.

3.2.1 Neural Network Structure

A neural network maps an input vector $\mathbf{x} \in R^n$ to an output $\mathbf{y} \in R^m$ by passing it through multiple layers of interconnected units or *neurons*. Each neuron performs a linear transformation followed by a nonlinear activation:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right) \tag{16}$$

where $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector of the l-th layer, and $\sigma(\cdot)$ is a nonlinear activation function (e.g., ReLU, tanh, sigmoid). This nonlinear activation is what allows neural networks to model complex, nonlinear mappings, which are crucial for capturing realistic system behaviors.

3.2.2 Training Neural Networks

The training process of a neural network involves adjusting its weights θ to minimize a prediction loss over a dataset. For dynamics modeling, the network is trained on transition tuples $\{(x_t, u_t, x_{t+1})\}$ collected from rollouts:

- **Forward Propagation:** Computes predicted next state $\hat{f}_\theta(x_t, u_t)$ using the current weights.
- **Loss Function:** Measures prediction error, here via mean squared error (MSE).
- **Backpropagation:** Computes gradients $\nabla_\theta \mathcal{L}$ using the chain rule to update weights via gradient descent.
- **Optimization:** Uses stochastic gradient descent (SGD) or variants like Adam to minimize the loss.
- **Regularization:** Methods like dropout and weight decay help avoid overfitting and improve generalization.

4 Implementation and Results

We begin by evaluating our approach on the `MountainCarContinuous-v0` environment, a standard benchmark in reinforcement learning. The task involves driving a car up a steep hill by first building momentum through oscillatory motion, as the engine is not powerful enough to ascend directly. This environment is challenging due to delayed action effects and nonlinear dynamics.

An initial implementation using a Linear Quadratic Regulator (LQR) highlights its limitations: the controller applies force in a single direction toward the goal, failing to build the necessary momentum. This illustrates that linear controllers, based on linear dynamics and quadratic cost assumptions, are insufficient for solving such nonlinear tasks effectively.

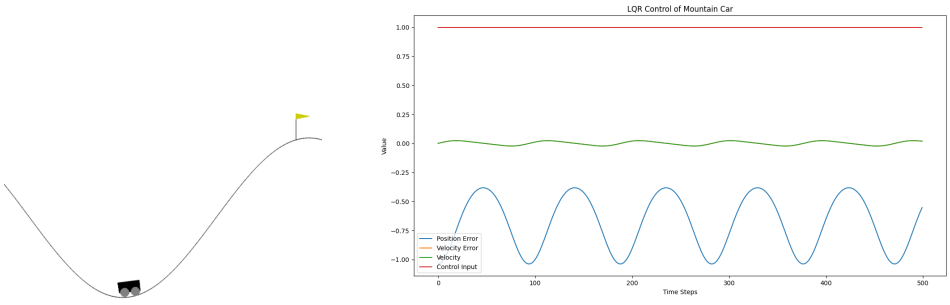

Figure 1: (Left) Mountain Car environment (Right) Result with simple LQR, inadequate to reduce position error to 0.

4.1 iLQR with Known Dynamics

We begin our study by implementing the iterative Linear Quadratic Regulator (iLQR) using the known true dynamics of the system. This method iteratively optimizes a sequence of control inputs by alternating between backward and forward passes.

At each iteration, the algorithm:

1. Performs a **backward pass** to compute time-varying linear feedback gains using linearized dynamics.

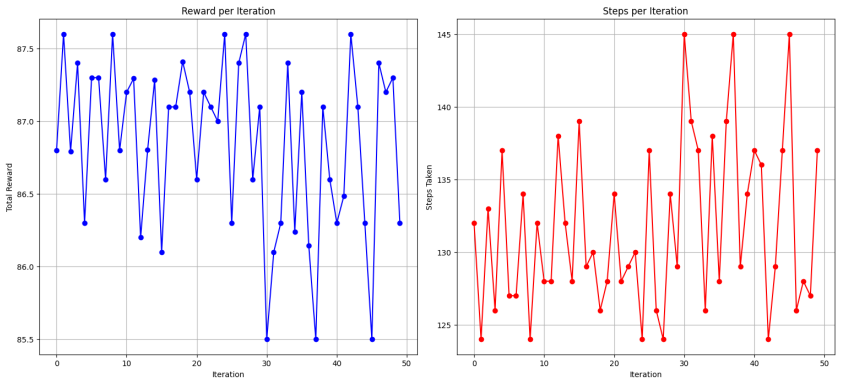

Figure 3: Rewards and steps taken over 50 iterations. Shows that the algorithm is consistent in reaching the target in 100-150 steps.

2. Uses a **forward pass** to simulate the nonlinear dynamics with new control sequence, producing an updated trajectory.
3. Repeats the above steps until convergence.
4. Applies the first action from the optimized control sequence to the system to move to the next step.

4.1.1 Implementation Details for MountainCar

The known system dynamics of the MountainCar environment are directly used in the forward pass, while the backward pass employs locally linearized dynamics to compute control updates via matrices A and B.

The iLQR implementation uses a planning horizon of $T = 40$, with 5 LQR iterations performed at each step. The terminal cost matrix is set as $Q_T = \begin{bmatrix} 10 & 0 \\ 0 & 0 \end{bmatrix}$, and the control cost matrix is $R = [0.1]$. The backward pass uses the linearized dynamics to compute time-varying feedback gains, and the forward pass simulates the actual nonlinear dynamics to generate updated trajectories.

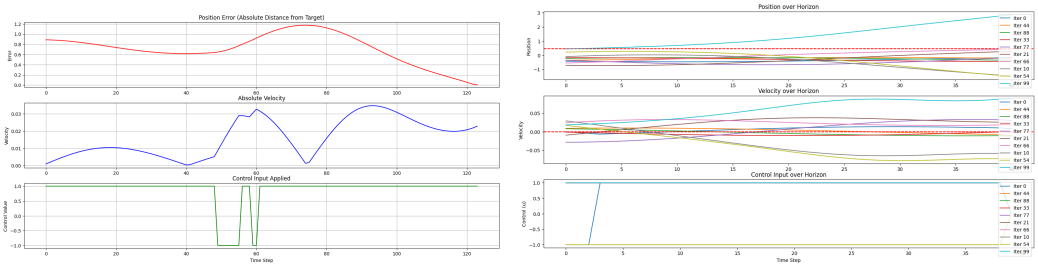

Figure 2: iLQR on MountainCar with known dynamics. **(Left)** Actual rollout of the system showing position error, velocity, and control input over time. **(Right)** iLQR-predicted state trajectories (position and velocity) and the corresponding optimal control inputs over a 10-step horizon, which were used to generate the rollout shown on the left.

4.2 iLQR with Neural Network Dynamics

In practical scenarios, the true system dynamics are often unknown or difficult to model analytically. To address this, we implement iLQR using a learned dynamics model based on a feedforward neural network. The neural network acts as a surrogate model for the environment, enabling model-based planning without access to ground-truth dynamics.

Neural Network Dynamics Model

We use a Multi-Layer Perceptron (MLP) implemented via `sklearn`'s `MLPRegressor`. The architecture is as follows:

- **Input:** Concatenated state-action vector (x, u)
- **Network:** Two hidden layers, each with 64 ReLU-activated neurons (Input $\rightarrow$ [64] $\rightarrow$ [64] $\rightarrow$ Output)
- **Output:** Predicted next state x_{t+1}

The model is trained online using mini-batch gradient descent with the Adam optimizer. An initial random exploration phase of 200 steps is used to populate the training buffer and encourage broad state-space coverage. During this phase, actions are chosen heuristically: i.e., apply $+1$ if position > 0, and -1 otherwise.

iLQR Implementation with Learned Dynamics

Once the neural network is initialized, we use it to generate forward rollouts within the iLQR loop. Since the dynamics model is learned and autograd is not available in the model we chose, we approximate the necessary gradients for the backward pass using finite differences.

The planning horizon is set to 50 steps, and the neural network is retrained every 50 environment steps using the latest collected data. This keeps the model up-to-date with the agent's recent experiences.

System Architecture

Figure 4 shows an overview of the iLQR planning framework using a neural network dynamics model.

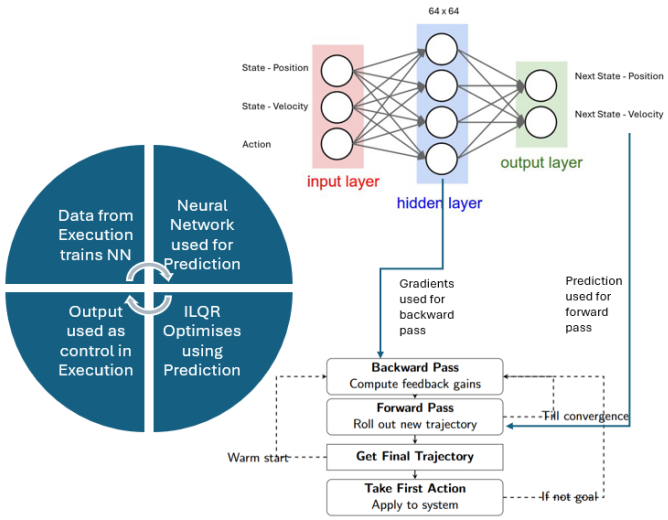

Figure 4: iLQR with neural network dynamics: system uses learned model to simulate forward trajectories, estimate gradients using finite differences, and compute optimal control via iLQR.

4.2.1 Results

We evaluate the performance of iLQR using neural network dynamics in two phases: first with partial integration, where the neural network predicts next states, but Jacobians are computed using known dynamics, and then with full integration, where both forward rollouts and Jacobians are estimated from the neural network.

We tested an intermediate setup (Figure 5) where the iLQR loop uses next-state predictions from the neural network but still relies on the true dynamics for computing gradients during the backward pass. This hybrid setting provides a smoother transition from model-based to learned dynamics.

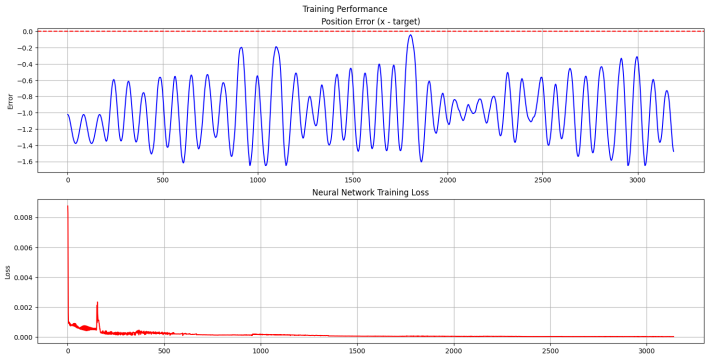

Figure 7: Example of a run that took significantly longer to reach the target. Unlike the previous implementation with known dynamics, this implementation has some uncertainty.

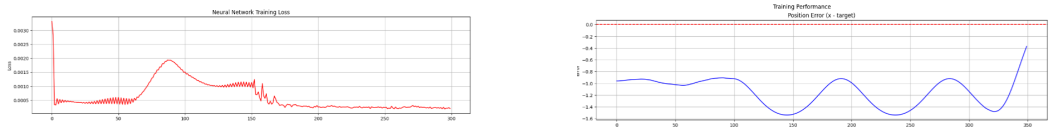

Figure 5: (Left) Training loss of the neural network over time. (Right) Position error during rollout using iLQR with partial NN dynamics.

iLQR with Fully Learned Dynamics

In the fully integrated setup, the iLQR loop relies entirely on the neural network: both for predicting state transitions and for estimating the Jacobians using finite differences. While this increases realism, it also introduces instability due to approximation errors in the learned model.

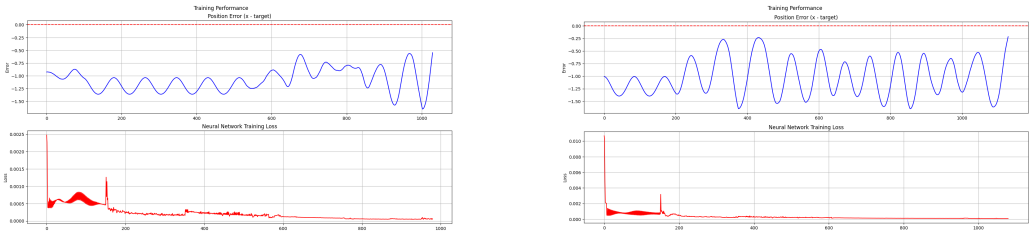

Figure 6: Training loss over time for full NN integration and position error during iLQR rollout using fully learned dynamics for 2 separate runs, which reached the target.

4.3 iLQR with Neural Dynamics on Swing-Up Pendulum

The Swing-Up Pendulum is a significantly more challenging control task compared to MountainCar. The system must not only balance the pendulum in the unstable upright position but also first swing it up from the downward configuration. This requires generating sufficient momentum and precisely timing energy injection, which makes the dynamics highly nonlinear and sensitive to model errors. Additionally, balancing at the upright position demands low-variance predictions and fine control, exposing the limitations of approximate models even more.

To meet these challenges, several improvements were introduced to the original MountainCar pipeline:

- **1. Experience Replay Buffer:** A buffer of 100K transitions was used to store past experiences. Batches of 64 transitions were sampled randomly for training, which helped break temporal correlations and stabilize learning.

- **2. PyTorch-Based Dynamics Model:** The neural network model was implemented using PyTorch, enabling GPU acceleration, scalable training, and—most importantly—automatic differentiation for precise Jacobian estimation in the iLQR backward pass.
- **3. Residual Network Architecture:** Instead of directly predicting the next state, the neural network was trained to predict the change in state Δx. This residual formulation improved convergence speed and produced smoother dynamics.
- **4. Enhanced Network Architecture:** The model used two hidden layers of 256 neurons each with ReLU activations. The network was trained using the Adam optimizer to minimize mean squared error (MSE) between predicted and true state transitions.
- **5. Structured Training Protocol:** Training followed an episodic loop: (1) collect rollout data using iLQR, (2) train the neural model for 50 epochs on the replay buffer, (3) replan with updated dynamics using iLQR, and repeat.
- **6. Adaptive Line Search:** To ensure stable convergence during control optimization, an adaptive line search was implemented. The control update rule was:

$$u_t^{\text{new}} = u_t + \alpha k_t + K_t(x_t - x_t^{\text{ref}})$$

with backtracking on α reducing by 0.5 to guarantee cost reduction at each iteration.

The results obtained were as per the figures below.

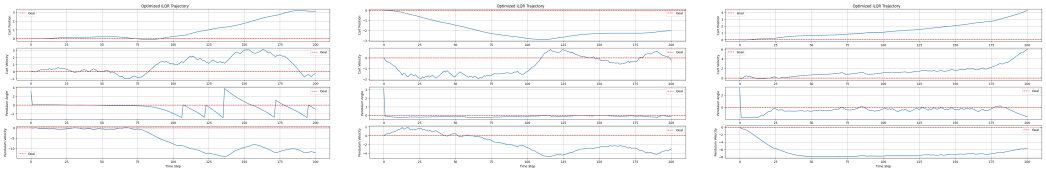

Figure 8: State variables predicted by iLQR in 3 different optimised paths, showing iLQR internally gives swing up and balance behaviour

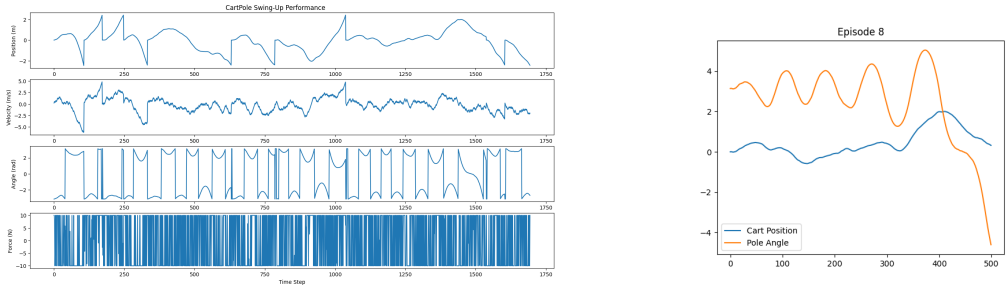

Figure 9: Simulation on environment gives very short, erratic episodes. (Right) Swinging up behaviour observed.

4.4 Conclusion and Future Work

Summary and Contributions

This work successfully demonstrates the application of **iterative Linear Quadratic Regulation (iLQR)** for solving nonlinear control tasks using both known and learned system dynamics. The proposed method was applied to two benchmark environments—**Mountain Car** and the more challenging **Swing-Up Pendulum** (CartPole)—within the Gymnasium framework.

A key contribution of this project is the integration of **online neural network training with control**, enabling simultaneous learning and planning in unknown dynamics. The neural network model was trained on-the-fly from environment rollouts and used to guide the iLQR optimization in real time.

iLQR combined with online-learned neural dynamics provides a powerful model-based control strategy even in challenging, nonlinear environments.

Future Work

While the current framework shows promising results, several directions remain open for further exploration:

- **Explicit System Identification:** Extend the learning to infer physical parameters such as mass, damping, or gravity, to make the model more interpretable and generalizable.
- **Complex Environment Transfer:** Apply the iLQR-with-NN method to higher-dimensional or real-world robotic systems to validate scalability.
- **Improved Sample Efficiency:** Explore model pretraining, hybrid model-based/model-free approaches, or uncertainty-aware learning to reduce the amount of interaction data required for effective control.

References

[1] Brandon Amos and J. Zico Kolter. Differentiable mpc for end-to-end planning and control. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[2] Weiwei Li and Emanuel Todorov. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, pages 222–229, 2004.

[3] Yunzhu Pan et al. A model predictive control approach for online driving style adaptation with learned vehicle dynamics. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[4] Tianyu Wang et al. Differentiable control for end-to-end planning and control. *Proceedings of the IEEE*, 2021.

[5] Hongkai Yin et al. Neural ilqr: Learning trajectory optimizers through backpropagation. *arXiv preprint arXiv:2103.00946*, 2021.

[6] Bo Zong et al. Model-based reinforcement learning with uncertainty regularization for optimizing long-term user engagement. *NeurIPS*, 2021.