A Dimension-Agnostic Reinforcement Learning Framework for Physics-Based Environments

Kaden Seto, Kane Pan, Ryan Qian

MAT292 — Ordinary Differential Equations

Instructors: Professor Vardan Papyan and Professor Murdock Aubry
TA: Delaram Sadatamin

December 19, 2025

Abstract

Deep reinforcement learning has proven to be a powerful approach in machine learning, where agents can learn to solve problems in environments through the application of neural networks. In particular, reinforcement learning has been applied to solve physics-based environments, which is prevalent in systems with non-linear dynamics. However, standard neural networks struggle to tackle physics problems because neural networks do not have any concept of the underlying math and physics of the system. Time dependencies in physics problems are not represented in the neural network, and in Deep RL, neural networks will learn trajectories of solutions based on probabilities of best actions versus the actual dynamics of the system. Neural Ordinary Differential Equations (NODEs) represent and learn the dynamics of the system by defining the fundamental differential equation as a neural network that learns the solution to an ODE, making it a powerful architecture for physics-informed machine learning. In this work, we propose **Continuum**, a deep RL framework and neural network architecture for physics-informed reinforcement learning. The architecture combines NODEs, Autoencoders, and model-free RL algorithms, where the latent space of the Autoencoder is represented by a time-dependent NODE that learns the continuous-time dynamics of the environment. In this architecture, we aim to build a neural network that has stronger physics alignment and interpretability, thus encouraging policies to make predictions based on structured latent representations of the learned system dynamics that promote stability and performance.

1 Introduction

Reinforcement learning (RL) is a branch of machine learning in which an AI agent learns through trial and error to make optimal decisions for achieving a set goal in an environment. RL has found application in optimizing decision-making in physical environments, such as robotics or modeling nonlinear behaviour in dynamics [1].

RL tasks often employ and combine techniques such as deep learning and neural networks in the form of Deep Reinforcement Learning (Deep RL) to learn solutions across environments. Although Deep RL allows powerful function approximators to learn effectively in complex environments [2], agents are given continuous observation and action spaces in physics-based environments where variables contain physical information that is easily modeled by ODEs. As such, Neural Ordinary Differential Equations (NODEs) [3], which model the inputs as a continuous-time dynamic system, may be a viable method for modeling dynamic physics-based environments. In our work, we propose a Deep RL framework and neural network architecture implementing Neural ODEs (NODEs), Autoencoders, and model-free algorithms that auto-configures to different physics-based environments, and we evaluate it against a standard PPO baseline framework across multiple Gymnasium and MuJoCo tasks. Moving forward we will refer to our proposed framework as **Continuum**.

2 Methodology

Physical control tasks like swinging joint systems or walking with a quadruped robot exist as testbeds for classical control and reinforcement learning (RL) algorithms. Traditional approaches often linearise the equations of motion and design linear–quadratic regulators (LQR), while modern RL methods can learn directly from observations. Many published RL implementations, however, are tightly coupled to a specific environment: they assume a fixed observation vector and either a discrete or continuous action space. We built a dimension-agnostic RL framework that automatically adapts to the size and type of the state and action spaces, and we benchmarked PPO against NODE-PPO variants (Euler and mainly RK4) on several physics-based environments.
Our project aims to build a modular framework that:

- Implements a single training and evaluation pipeline that adapts to the state and action space dimensions of any Gymnasium-compatible physics environment.
- Trains agents using PPO and Neural ODE (NODE) policies with Euler and RK4 integration [4].
- Evaluates across multiple environments. Testing will be done on classical environments such Acrobot, (discrete control) and MuJoCo environments such as HalfCheetah, Humanoid, etc. (continuous control).

Classical [5] and MuJoCo [6] environments are going to be the primary means of testing our framework. For context, an example environment from the classical set and MuJoCo set are: the AcroBot environment, which models two links forming a chain hanging in 2D, where the AI agent applies torque at the joint to try to swing the chain above a certain height; and the Ant environment, which represents a 3D quadruped robot where each leg has 2 joints, where the agent must apply torques at the eight hinges to move itself forward. Both tasks have low-dimensional observation spaces with unique action types (discrete vs. continuous), which makes them ideal to demonstrate the flexibility of our framework.

3 Mathematical Background & Model

3.1 Neural Ordinary Differential Equations (NODEs)

In classical deep learning, feedforward neural network architectures are designed with a finite sequence of discrete transformations/hidden layers. In each hidden layer, activation functions are applied which introduce non-linearity, allowing hidden layers to model complex data patterns [7]. Later on, recurrent neural network architectures were introduced that model sequences through time. The RNN updates hidden states h_t over time to form a sequence that represents the latent space transformation from the input to the output, thus allowing it to model dynamic systems via discrete updates:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{1}$$

However, in dynamically evolving processes such as classical control problems analyzed in this paper, the limitation of classic deep learning architectures is that it is limited by discrete layers which did not allow for smooth transitions between states [7]. Therefore, NODEs were introduced as a neural network architecture which is able to model continuous data transformations, replacing the standard discrete neural network layers with a continuous-time model that is able to adapt to changes in the system's dynamics [3].
A neural ODE is represented by:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta) = A(t)\mathbf{h}(t) + b(t) \tag{2}$$

where:

- $\mathbf{h}(t)$ is the hidden state of the system (or activations of a neural network) at a given time t
- f is a function, parameterized by the parameters of the neural network θ, representing the rate of change of the hidden state.
- t is the time/depth within the continuous model

Optimizing the NODE uses the same optimization structure as regular optimization loops, but for the ODE Network the **adjoint sensitivity** method is applied for computing the gradients. This method applies to all ODE solvers, making NODE frameworks easily adaptable and testable between different numerical methods. The loss function L is represented by:

$$L(z(t_1)) = L\Big(z(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\Big) = L\Big(\text{ODESolver}(f(\mathbf{z}(t_0), t_0, t_1, \theta))\Big) \tag{3}$$

Then, for finding the gradients, let $\mathbf{a}(t)$ represent the adjoint function (i.e. the gradient of the loss with respect to the hidden state) such that $\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{z}(t)}$. Using the adjoint, this allows back-propagation through the solver by providing the dynamics of the gradient signal (i.e. chain rule but applied to a continuous system):

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^\top \frac{\partial f(z(t), t, \theta)}{\partial z} \tag{4}$$

Then for the parameters θ, the gradient is represented as:

$$\frac{d\mathbf{L}}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^\top \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt \tag{5}$$

The gradients can then be updated via regular gradient descent, where:

$$\theta \leftarrow \theta - \alpha \frac{dL}{d\theta} \tag{6}$$

A comparison of a general neural network and ODE network in depth is shown in 1.

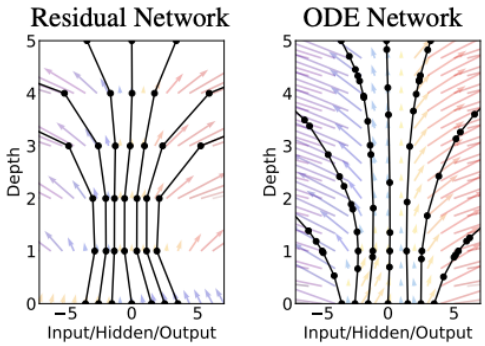


Figure 1: Depths of the hidden state compared between a residual feedforward network versus an ODE network. The ODE network represents the state as a vector field for continuous transformations, while the residual network uses discrete, piecewise-like, transformations [3]

For physics-based tasks and continuous control, which is present in Gym's classical control environments, the NODE is a fundamental architecture for **physics-informed neural networks**. NODEs also have several benefits that are beneficial in RL, where NODEs have better parameter efficiency since data is transformed through a continuous process, and they also have adaptive computation where ODE solvers can adapt evaluation to a required level of complexity which varies between different physics environments, supporting our framework to be more environment-agnostic as well.

3.1.1 Euler Method

A standard initial value problem (IVP) for an ordinary differential equation (ODE) can be written as

$$\frac{dy}{dt} = f(t, y), \qquad y(t_0) = y_0. \tag{7}$$

The simplest one–step method for approximating its solution is the Euler method. We introduce a uniform time grid

$$t_n = t_0 + nh, \qquad n = 0, 1, \ldots, N, \tag{8}$$

where $h > 0$ is the step size, and denote the numerical approximation to $y(t_n)$ by y_n. Using a first–order Taylor expansion of the exact solution around t_n we get the equation for Euler's method,

$$y_{n+1} = y_n + h\, f(t_n, y_n). \tag{9}$$

In a Neural ODE, we treat the hidden state $\mathbf{h}(t)$ as the solution of a learned ODE

$$\frac{d\mathbf{h}(t)}{dt} = f_\theta(\mathbf{h}(t), t), \tag{10}$$

where f_θ is a neural network with parameters θ. To advance the hidden state from t_n to $t_{n+1} = t_n + h$ using forward Euler, we apply

$$\mathbf{h}_{n+1} = \mathbf{h}_n + h\, f_\theta(\mathbf{h}_n, t_n). \tag{11}$$

This equation has the same form as a residual block in a deep network, $\mathbf{h}_{n+1} = \mathbf{h}_n + g_\theta(\mathbf{h}_n)$, if we set $g_\theta = h\, f_\theta$. In other words, using Euler as the `ODESolver` makes our NODE behave like a standard residual network with a fixed step size in time.
Euler's Method has two main advantages:

- It requires a single evaluation of f_θ per step, making it computationally cheap and fast.
- Its simple structure makes gradient backpropagation using the adjoint method straightforward

However, having only the first order of accuracy means that Euler's Method may require many small steps to accurately track rapidly changing dynamics in physics-based environments (e.g. HalfCheetah or Ant). It has a relatively high local truncation error of order $O(h^2)$ and a global error of order $O(h)$ over a fixed time interval. This motivates us to test higher order solvers such as the fourth–order Runge–Kutta method to see whether a more accurate and stable solver of the latent state $\mathbf{h}(t)$ improves RL performance.

3.1.2 Fourth-Order Runge Kutta Method

The classical fourth–order Runge–Kutta (RK4) method is a higher–order one–step method for solving the same IVP in Euler's Method 7. Given a current approximation $y_n \approx y(t_n)$ and a step size h, RK4 computes four intermediate slopes

$$k_1 = f(t_n,\, y_n), \tag{12}$$
$$k_2 = f\big(t_n + \tfrac{h}{2},\, y_n + \tfrac{h}{2}k_1\big)\,, \tag{13}$$
$$k_3 = f\big(t_n + \tfrac{h}{2},\, y_n + \tfrac{h}{2}k_2\big)\,, \tag{14}$$
$$k_4 = f(t_n + h,\, y_n + hk_3)\,, \tag{15}$$

and updates the solution via the weighted average

$$y_{n+1} = y_n + \frac{h}{6}\,(k_1 + 2k_2 + 2k_3 + k_4). \tag{16}$$

The method has a local truncation error of order $O(h^5)$ and a global error of order $O(h^4)$, making it significantly more accurate than the first–order Euler method.
In our Neural ODE setting, using Equation 10 and replacing f with f_θ in the RK4 update gives

$$\mathbf{h}_{n+1} = \mathbf{h}_n + \frac{h}{6}\,\big(k_1 + 2k_2 + 2k_3 + k_4\big), \tag{17}$$

where each k_i is evaluated using the neural network f_θ. Choosing `rk4` as the `ODESolver` therefore trades four function evaluations per step for a much higher–order approximation of the latent dynamics.
This method is strong because many physics-based RL tasks exhibit fast, nonlinear changes in complex observation spaces. A more accurate and stable integration of the hidden state $\mathbf{h}(t)$ may allow the NODE to represent these dynamics with fewer integration steps, potentially improving learning stability and final RL performance compared to the cheaper, but less accurate, Euler's method.

3.2 On-Policy Algorithms and Proximal Policy Optimization (PPO)

In reinforcement learning, on-policy algorithms are algorithms that represent and directly update the parameters θ of a policy $\pi_\theta(a|s)$ by optimizing an objective function $J(\pi_\theta)$. In on-policy algorithms, the goal is to find a policy gradient $\nabla_\theta J(\pi_\theta)$ that increases the probabilities of actions that maximize higher return (i.e. cumulative reward that the agent earns from interacting with the environment), and decreases the probabilities of actions that lead to lower return [8]. A policy's parameters are updated via gradient ascent on the objective:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta), \theta = \theta_k \tag{18}$$

where the policy gradient is given as:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right] \tag{19}$$

which is the summation over all trajectories sampled from the current policy π_θ, where for each trajectory the log-probability of taking an action in a current state is $\log \pi_\theta(a_t|s_t)$ and $R(\tau)$ is the reward that quantifies how good taking trajectory τ was.

The main issue with adjusting the policy's parameters is that large updates can cause instabilities in policy performance. To solve this issue, PPO introduces a **surrogate objective** $L(s, a, \theta_k, \theta)$ that is able to take the largest possible policy update without causing performance instability and collapse. PPO updates policies with:

$$\theta_{k+1} = \theta_k + \mathop{\mathbb{E}}_{(s,a) \sim \pi_{\theta_k}} [L(s, a, \theta_k, \theta)] \tag{20}$$

For a state-action pair that provides a positive advantage (i.e. how good taking an action is in a state versus simply following the current policy in a current state), the surrogate objective is clipped by a minimum to ensure the update does not increase too far:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1 + \epsilon) \right) A^{\pi_{\theta_k}}(s, a) \tag{21}$$

For a state-action pair providing a negative advantage (or disadvantage), the surrogate objective is clipped by a maximum to ensure the update does not decrease too far:

$$L(s, a, \theta_k, \theta) = \max \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1 - \epsilon) \right) A^{\pi_{\theta_k}}(s, a) \tag{22}$$

Putting Equations (10) and (11) together, PPO can be described as one function using the clip function:

$$J_{\text{PPO}}(\theta) = L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{ clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right) \tag{23}$$

3.3 Formulation of Continuous-Time Proximal Policy Optimization with NODEs

For our framework, we introduce a **continuous-time NODE-PPO** method for optimization by combining the concepts of regular NODEs introduced in Section 3.2 and regular PPO in Section

3.3. For a state-action pair (s, a), the fundamental neural ODE in Equation (2) is applied for the state-action pair to solve:

$$\frac{d\mathbf{s}(t)}{dt} = \mathbf{f}(\mathbf{s}(t), \mathbf{a}(t)) \tag{24}$$

where $\mathbf{a}(t)$ represents the control input function, and $\mathbf{s}(t)$ represents the vector-valued state function for representing the continuous dynamics of the environment [9] which can be solved by:

$$\mathbf{s}(t) = \mathbf{s}_0 + \int_0^t \mathbf{f}\left(s(T), a(T)\right) dT \tag{25}$$

where $\{T \in \mathbb{R} | T > 0\}$ represents an arbitrary time variable.

For continuous-time control tasks in RL, the goal is to maximize a value function/discounted infinite horizon reward integral for the objective function:

$$J_{CT}(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\int_t^\infty e^{-\frac{T-t}{\eta}} r(\mathbf{s}(T), \mathbf{a}(T)) dT \right] \tag{26}$$

where $r(\mathbf{s}(T), \mathbf{a}(T))$ represents an instantaneous reward for a given state function and action function of a trajectory sampled from the policy, and $\eta \in (0, 1)$ represents the exponential decay discount factor [9].

Since the loss is computed with the state function $s(T)$ and the policy π_θ, the objective gradient term with respect to the parameters $\frac{dJ}{d\theta}$ of NODE-PPO now become the sum of the policy gradient term and the adjoint $\mathbf{a}(t)$ term (from Equation (4)):

$$\frac{dJ}{d\theta} = \frac{\partial J}{\partial \pi_\theta} \frac{\partial \pi_\theta}{\partial \theta} + \int_{t_0}^{t_1} \mathbf{a}(t)^\top \frac{\partial f(\mathbf{s}(t), \mathbf{a}(t), t)}{\partial \theta} dt \tag{27}$$

Note that the gradient descent parameter update rule in Equation (6) is applied the same way for this new $\frac{dJ}{d\theta}$ term.

In Equation (15), $J_{CT}(\theta)$ represents the objective for vanilla policy gradient (VPG). Extending Equation (15) and the regular PPO expression in Equation (12), the surrogate objective for the continuous-time NODE-PPO method used in our framework, $J_{\text{NODE-PPO}}(\theta)$, becomes the clipped version of the regular objective:

$$\begin{aligned} J_{\text{NODE-PPO}}(\theta) &= \mathbb{E}_{\tau \sim \pi_\theta} \left[\int_t^\infty e^{-\frac{T-t}{\eta}} \min\Big(r(T, \theta)\, A(T),\ \text{clip}\big(r(T, \theta),\, 1-\epsilon,\, 1+\epsilon\big)\, A(T) \Big)\, dT \right], \\ A(T) &= A^{\pi_{\theta_k}}\big(\mathbf{s}(T), \mathbf{a}(T)\big), \qquad r(T, \theta) = \frac{\pi_\theta\big(\mathbf{a}(T) \mid \mathbf{s}(T)\big)}{\pi_{\theta_k}\big(\mathbf{a}(T) \mid \mathbf{s}(T)\big)}. \end{aligned} \tag{28}$$

where the instantaneous advantage of a state-action pair function at a given time T is $\mathbf{A}(T)$ and the probability ratio of taking an action in the new policy versus the old policy is $\mathbf{r}(T, \theta)$.

3.4 Representation Learning & Autoencoders (AEs)

In machine learning, autoencoders are a type of neural network architecture that learns a compressed/encoded representation of data and reconstructs the original data in the output [10]. The

goal of using an autoencoder is to use the network to effectively learn the latent space, which is the lower-dimensional representation of the data that extracts and preserves the most important features. Encoder layers of the autoencoder consist of normal network layers such as a fully-connected MLP layer that reduces the dimensionality of the data via matrix multiplication, and the decoder layers map the data back into a higher-dimensionality representation via network layers similar to the encoder. An encoder example is shown in 2.

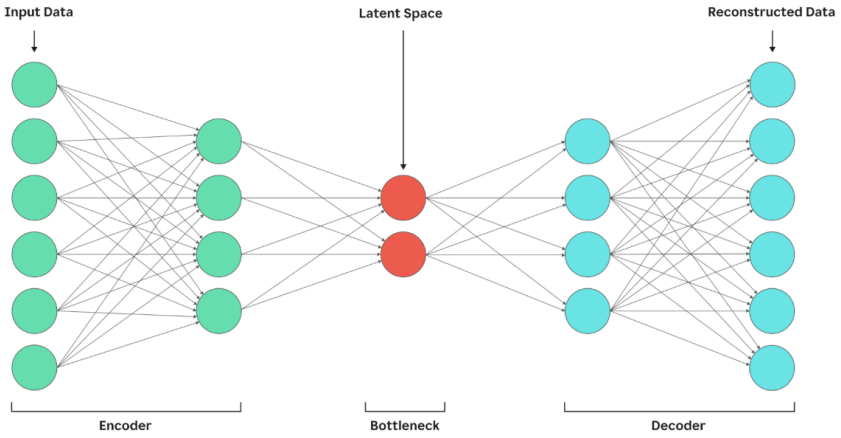


Figure 2: Autoencoder neural network architecture. Input data dimensionality is gradually reduced via the encoder to learn the latent space, and is then increased to the reconstructed data dimension [11]

The model architecture proposed in this paper is inspired by the autoencoder architecture, where the goal is to transform the raw observation space input into a low-dimensional latent representation vector that will best represent the dynamics of the environment. The observations will be passed into an encoder layer to reduce the dimensionality before it is fed into the Neural ODE, which will capture the most important observations into the environment's physics while pruning noisy or redundant observations. Thus, we aim to have better feature representation using this autoencoder architecture.

For the Neural ODE, this is particularly useful since the NODE aims to model continuous-time dynamics (from Figure 1). When the hidden state is of smaller dimension and has a relatively clean feature representation to work from, the NODE can learn the dynamics more easily and stable, with large discontinuities and unstable trajectories prevented. Additionally, since the dimensionality of the observations are reduced, this allows the policy to be more sample-efficient, and the Neural ODE can learn much faster with less latent variables.

3.5 Model Architecture

In our proposed architecture, we combine all the background on NODEs, PPO, and AEs. First, the observation space inputs are passed into an encoder layer to reduce the input dimension space. The latent space is then represented by the Neural ODE, which receives the reduced observation input and passed into the NODE for learning and representing dynamics. The output of the NODE is then increased in dimensionality shape via the decoder layer and passed into the actor and critic heads for the Actor-Critic network architecture used in PPO. The PPO algorithm (in Sections 3.3

and 3.4) is then implemented to optimize the model. The finalized proposed architecture combining NODEs, PPO, and AEs is viewable in 3.

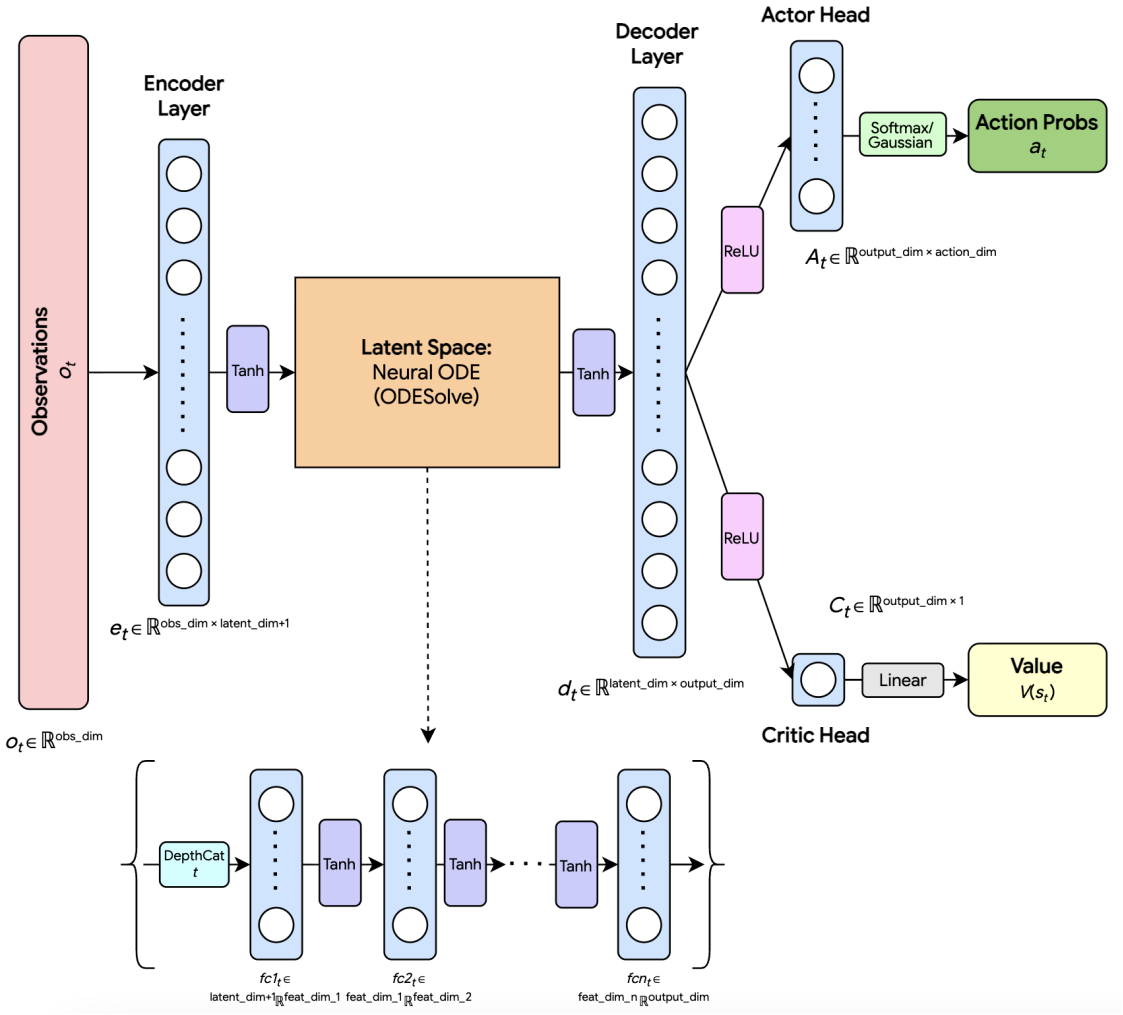


Figure 3: Schematic of the proposed model architecture. Note that the Neural ODE network provided is an MLP, but it can be any type of neural network architecture (e.g. an LSTM).

4 Results

The training infrastructure successfully handled both lower-dimensional classic control (Acrobot-v1), non-MuJoCo dynamics (LunarLander), standard locomotive MuJoCo environments (HalfCheetah-v5, Ant-v5), and higher-dimensional locomotion tasks (Humanoid/HumanoidStandup), without having to rewrite the environment beyond the initial configuration. Thus, **Continuum** functions as a general training framework rather than a single-environment demo. We evaluated the training pipeline for Acrobot-v1, LunarLander-v3, HalfCheetah-v5, Ant-v5, Humanoid-v5, and HumanoidStandup-v5. For each environment, we used three models to train and observe performance: (1) standard PPO (baseline), (2) PPO + Neural ODE with Euler integration (NODE-Euler), and (3) PPO + Neural ODE with RK4 integration (NODE-RK4). We used a TensorBoard to track the results for mean episodic reward, average episode length, and training time. We also

tested NODE-Euler on a subset of environments (specifically Acrobot-v1 and Ant-v5) and found that its compute cost was comparable to, and in some cases longer than, NODE-RK4, while its performance was substantially worse than both NODE-RK4 and standard PPO (as shown in Figure 5 in Appendix A). To save time, we therefore focus on comparisons between standard PPO and NODE-PPO (RK4) as the candidate models.
We will use two metrics for the performance of the models:

- **Mean episodic reward (final return)** is the primary success metric for all tasks.
- **Training time overhead (Δ_{time})** quantifies the computational cost of augmenting PPO with a Neural ODE block and using higher-order integration, which is essential for judging whether any performance gains are practical.

Average episode length (final ep_len) provides additional information on the behaviour of the models, and helps us predict behaviour if the models were to be trained for more steps. Broadly, episode lengths reveal whether reward changes are coming from actually improving control or if its coming from early terminations (e.g. frequent falls or crashes).
A full summary of the final performance of every model for each environment is provided in Table 1. Because each environment uses a different reward function and scale, absolute return values are not directly comparable across environments, so we additionally report a **normalized return improvement** for NODE-PPO (RK4) relative to PPO within each environment:

$$\Delta_{\text{norm}} = \frac{R_{\text{RK4}} - R_{\text{PPO}}}{|R_{\text{PPO}}|},$$

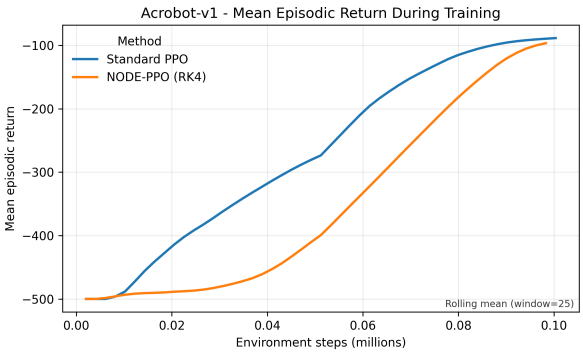
where R_{PPO} and R_{RK4} are the final mean episodic returns at the fixed training budget (positive values indicate improvement over PPO). We also report a **normalized time increase** based on elapsed training time:

$$\Delta_{\text{time}} = \frac{T_{\text{RK4}} - T_{\text{PPO}}}{T_{\text{PPO}}},$$

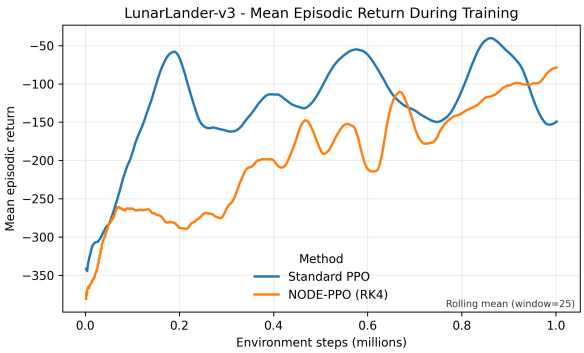
where T_{PPO} and T_{RK4} are the elapsed times to reach the final step count (lower is better). Per-environment learning curves (mean reward) are shown in Figure 4.

Table 1: Final performance and compute cost at a fixed training budget for each environment and model. "Elapsed time" is the time to reach the final step count, and "Δ_{time}" is normalized time overhead relative to PPO (lower is better). Acrobot-v1, Humanoid-v5, and HumanoidStandup-v5 were trained on a MacBook (M3 Pro), while the other runs were trained on a Windows PC (Ryzen 9950X3D, RTX 5080). For HalfCheetah-v5 and HumanoidStandup-v5, there are no terminating conditions, so the terminating time limit was set to 1000 steps. In general, higher `final ep_len` means better model performance, with the exception of Acrobot-v1, where lower is better. Higher `Final return` indicate better policies, while lower `Elapsed time` indicates lower compute cost. Δ_{time} reports normalized compute cost increase and Δ_{norm} reports normalized return improvement of NODE-PPO (RK4) relative to PPO.

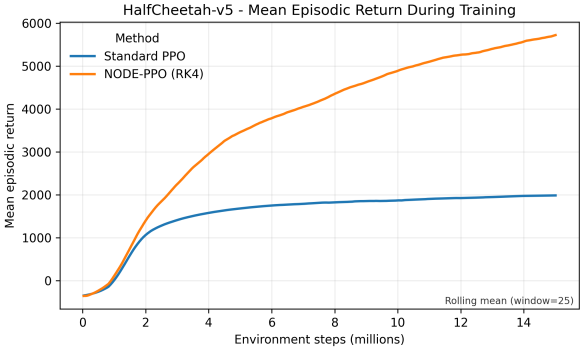
Environment	**Model**	**Steps**	**Final return**	**Final ep_len**	**Elapsed time**	$\boldsymbol{\Delta}_{\text{time}}$	$\boldsymbol{\Delta}_{\text{norm}}$
Acrobot-v1	PPO	100,352	-83.05	84.05	1.858 hr	—	—
Acrobot-v1	NODE-PPO (RK4)	100,352	-86.78	87.78	2.411 hr	0.2976	-0.0449
LunarLander-v3	PPO	1,001,472	-141.4014	533.08	5.166 min	—	—
LunarLander-v3	NODE-PPO (RK4)	1,001,472	-75.4475	303.07	1.652 hr	18.1870	0.4663
HalfCheetah-v5	PPO	15,007,744	1990.3373	1000	18.42 min	—	—
HalfCheetah-v5	NODE-PPO (RK4)	15,007,744	5812.5371	1000	3.563 hr	10.6059	1.9203
Ant-v5	PPO	30,015,488	1182.2028	483.52	50.78 min	—	—
Ant-v5	NODE-PPO (RK4)	30,015,488	1050.5352	617.24	7.009 hr	7.2816	-0.1114
Humanoid-v5	PPO	15,001,600	530.1929	104.06	2.34 hr	—	—
Humanoid-v5	NODE-PPO (RK4)	15,001,600	3468.7671	617.37	13.44 hr	4.7436	5.5420
HumanoidStandup-v5	PPO	15,007,744	276386.8750	1000	41.46 min	—	—
HumanoidStandup-v5	NODE-PPO (RK4)	15,007,744	156949.6094	1000	6.21 hr	7.9870	-0.4322



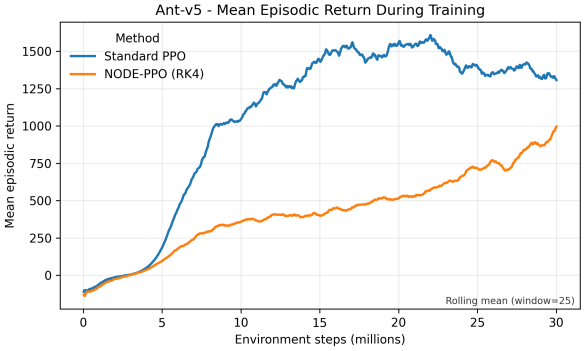
(a) Acrobot-v1



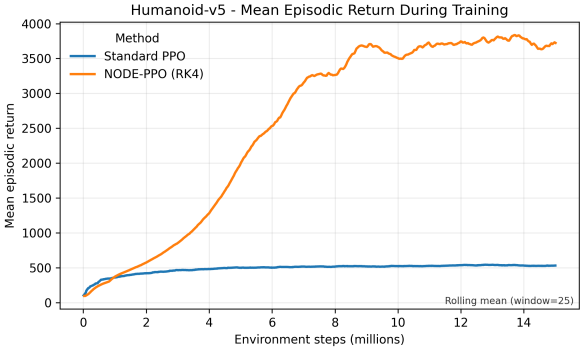
(b) LunarLander-v3



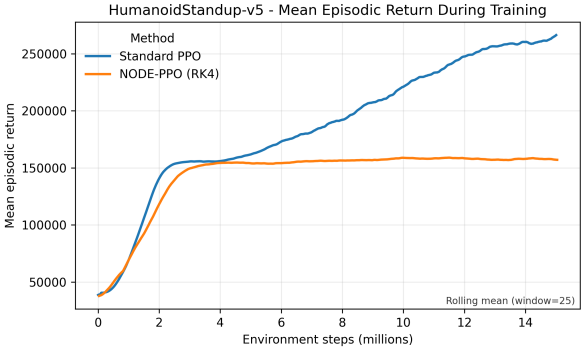
(c) HalfCheetah-v5



(d) Ant-v5



(e) Humanoid-v5



(f) HumanoidStandup-v5

Figure 4: Learning curves for mean episodic return versus environment steps (millions). Curves are computed from TensorBoard exports and plotted with a rolling mean (window = 25) for plot smoothing and readability. Visualization of some of the NODE trained models are hosted at our website

Over the six environments, NODE-PPO (RK4), which we named **Continuum** and will be referred as such from now on, displayed mixed results when compared with the standard PPO baseline model (Table 1, Figure 4). In HalfCheetah-v5 and Humanoid-v5, **Continuum** ended with significantly higher final return than PPO, growing past the plateaus that PPO struggled to pass. For these environments, **Continuum** acheived large normalized return gains over PPO ($\Delta_{\text{norm}} = 1.92$ and 5.54, respectively). **Continuum** also outperformed PPO in LunarLander-v3 after 1 million steps ($\Delta_{\text{norm}} = 0.47$); PPO showed fast early growth in its reward, but then had high instability (large oscillations) and a gradual decline in performance, while **Continuum** remained comparatively stable (smaller oscillations) throughout training and continued to improve and eventually

surpassing PPO. In contrast, PPO outperformed **Continuum** on Ant-v5, HumanoidStandup-v5, and Acrobot-v1, where **Continuum** either converged slower (Acrobot-v1, Ant-v5) or plateaued at a lower final return (HumanoidStandup-v5). In all environments, **Continuum** had a significantly higher training-time cost than PPO, having normalized time overheads ranging from $\Delta_{\text{time}} = 0.2976$ to $\Delta_{\text{time}} = 18.187$. Overall, these results indicate that the RK4-integrated model can yield large performance gains over standard PPO in some continuous-control settings, but these gains are not consistent across all tasks and come with a high compute-cost tradeoff.

5 Discussion

5.1 Analysis

While Table 1 indicates that **Continuum** only produced better results in 3/6 environments, there are reasonable explanations for this. By examining the return plots (Figure 4), we can see that the return for our framework is still increasing with steps, and has not completely converged aside from the Humanoid-v5, HumanoidStandup-v5 and Acrobot-v1 environments. We theorize that in HumanoidStandup that the framework got stuck at a local minima, and thus will be disregarding it in further discussion. Acrobot has a maximum return of 0, as it terminates when the goal has been achieved, thus the graph indicates that both PPO and **Continuum**, have solved the environment. Out of the rest of the environments, the only environment such that **Continuum** has finished converging is Humanoid, where the $\mathbf{\Delta}_{\text{norm}}$ is the greatest at 5.5. Examining the rest of the environments, (Ant-v5, LunarLander-v3, HalfCheetah-v5) we can see that the trend in return is still increasing; and as shown in Figure 8 in Appendix A, the episode lengths have increased to be greater than that of PPO. Thus we expect, given more steps in training, that **Continuum**, could outperform the standard PPO baseline. We can also observe in Acrobot, LunarLander, and Ant, that the return curve is more stable and smoothly learning than standard PPO, implying that the **Continuum** is learning the continous-time dynamics well.

5.2 Limitations and Suggestions

As the autoencoder works to transform the raw observation space into a lower dimension prior to being fed to the NODE, it inherently increases the training speed and streamlines inference. However, it does not necessarily decouple the prior states into physically meaningful components (e.g. position vs. velocity), and thus does not guarantee that the learned latent variables correspond to interpretable physical quanties. The inherent issue with this is that in extremely dynamic and complex environments, the autoencoder can lose explicit constraints (such as conservation of momentum), leading to poor inference. Thus, while the dimension-agnostic nature of the framework may be advantageous for software flexibility, the physics component is implicit rather than explicit, making human interpretability more difficult. Furthermore, given the already slow training times of the NODE in comparison to PPO, we can not recommend removing the autoencoder in future cases either. We also suggest in future iterations to explore the latent space to increase interpretability and explore what the model learns at each layer.

Even with the autoencoder, a significant limitation of **Continuum**, is still that the computational cost is great. Unlike a fixed-depth feedforward neural network, that will have a fixed computational depth of a simple activation function, **Continuum** requires an ODE solver (Euler, Runge-Kutta, etc.) to repeatedly solve hidden states in each forward pass as seen in Equation 3. When consid-

ering higher complexity dynamic systems, the ODE solver may require smaller step sizes, or even more function evaluations to maintain numerical stability in the solved function, resulting in even longer training times compared to the baseline PPO, causing further limitations on **Continuum**'s scalability without substantial hardware.

On the note of numerical stability, there are also relevant limitations with the numerical solvers used. In many physics based environments (such as those tested in MuJoCo, e.g. Ant, HalfCheetah) contact forces and collisions must be addressed. Mathematically this can cause discontinuities or sharp jumps in the differential equations that govern the given environment. Thus, non-stiff solvers like the Euler method and RK4 may struggle to accurately model these complex systems, as if the step size is too large, the solver may overshoot, creating instability or even exploding gradients. While we aim for the framework to be dimension agnostic, **Continuum** as it is currently, assumes the underlying dynamics are smooth enough to be modeled by a continuous ODE. More complicated systems may require more specialized solvers. In future iterations or frameworks, we suggest testing other solvers alongside stiff solvers, such as Matlab's ode15s [12].

6 Conclusion

Continuum advances deep reinforcement learning in stochastic physics-based environments by integrating model-free algorithms such as PPO, Neural ODEs, and Autoencoders to effectively learn continuous-time feature dynamics for effective and stable learning, creating a framework that excels at physics-informed reinforcement learning. Across Gymnasium and MuJoCo benchmarks, we find that the model architecture proposed by **Continuum** (using PPO and the RK4 solver) either performs comparable to or outperforms standard PPO and deep learning architectures in model performance and stability in many environments. In environments that especially involve continuous dynamics in mechanics, such as HalfCheetah-v5 or Humanoid-v5, **Continuum** outperforms standard PPO by approximately 300% and 600%, respectively. The novel architecture introduced in this project has been shown to enable better learning performance and stability, and opens opportunities for further research and advancements in related fields such as model-free reinforcement learning and imitation learning. Future efforts will focus on exploring opportunities to apply this architecture in to these related domains, as well as investigating model interpretability in our current architecture to gain further insights on the latent representations of physics and dynamics in the network arhictecture. Such insights may inspire the design of more effective neural network architectures in future work and research.

References

[1] C. Tang, B. Abbatematteo, J. Hu, R. Chandra, R. Martín-Martín, and P. Stone, *Deep reinforcement learning for robotics: A survey of real-world successes*, arXiv.org, 2024. [Online]. Available: https://arxiv.org/abs/2408.03539.

[2] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, pp. 219–354, 2018. DOI: 10.1561/2200000071.

[3] R. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural ordinary differential equations*, 2018.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Openai, *Proximal policy optimization algorithms*, 2017. Accessed: Oct. 19, 2025. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf.

[5] *Classic control documentation*, gymnasium.farama.org. [Online]. Available: https://gymnasium.farama.org/environments/classic_control/.

[6] *Mujoco documentation*, Farama.org, 2021. [Online]. Available: https://gymnasium.farama.org/environments/mujoco/.

[7] G. Xiao, *Neural ordinary differential equations (neural odes): Rethinking architecture*, Medium, May 2024. Accessed: Nov. 7, 2025. [Online]. Available: https://medium.com/%40justygwen/neural-ordinary-differential-equations-neural-odes-rethinking-architecture-272a72100ebc.

[8] *Part 3: Intro to policy optimization — spinning up documentation*, Openai.com, 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html.

[9] Ç. Yıldız, M. Heinonen, and H. Lähdesmäki, *Continuous-time model-based reinforcement learning*. Accessed: Nov. 7, 2025. [Online]. Available: https://proceedings.mlr.press/v139/yildiz21a/yildiz21a.pdf.

[10] P. Kashyap, *A comprehensive guide to autoencoders - piyush kashyap - medium*, Medium, Dec. 2024. Accessed: Nov. 30, 2025. [Online]. Available: https://medium.com/@piyushkashyap045/a-comprehensive-guide-to-autoencoders-8b18b58c2ea6.

[11] *Autoencoders in deep learning: How they work and why they matter*, What Is an Autoencoder in Deep Learning? Oct. 2024. [Online]. Available: https://www.grammarly.com/blog/ai/what-is-autoencoder/.

[12] *Solve stiff odes*, Mathworks.com, 2025. Accessed: Dec. 7, 2025. [Online]. Available: https://www.mathworks.com/help/matlab/math/solve-stiff-odes.html.

7 Appendix A: Extra Figures and Plots

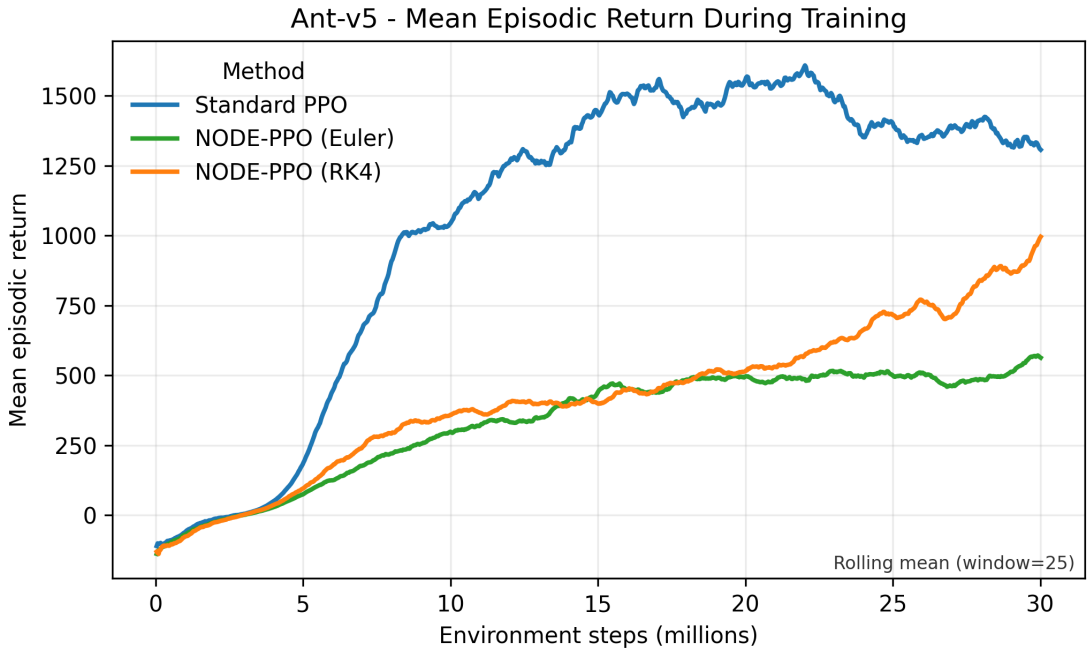


Figure 5: Ant-v5 mean episodic return during training (rolling mean, window = 25) comparing standard PPO, NODE-PPO (Euler), and NODE-PPO (RK4). Euler lags behind both PPO and RK4 in final return and required roughly ∼1 hour longer wall-clock time than RK4 to run.

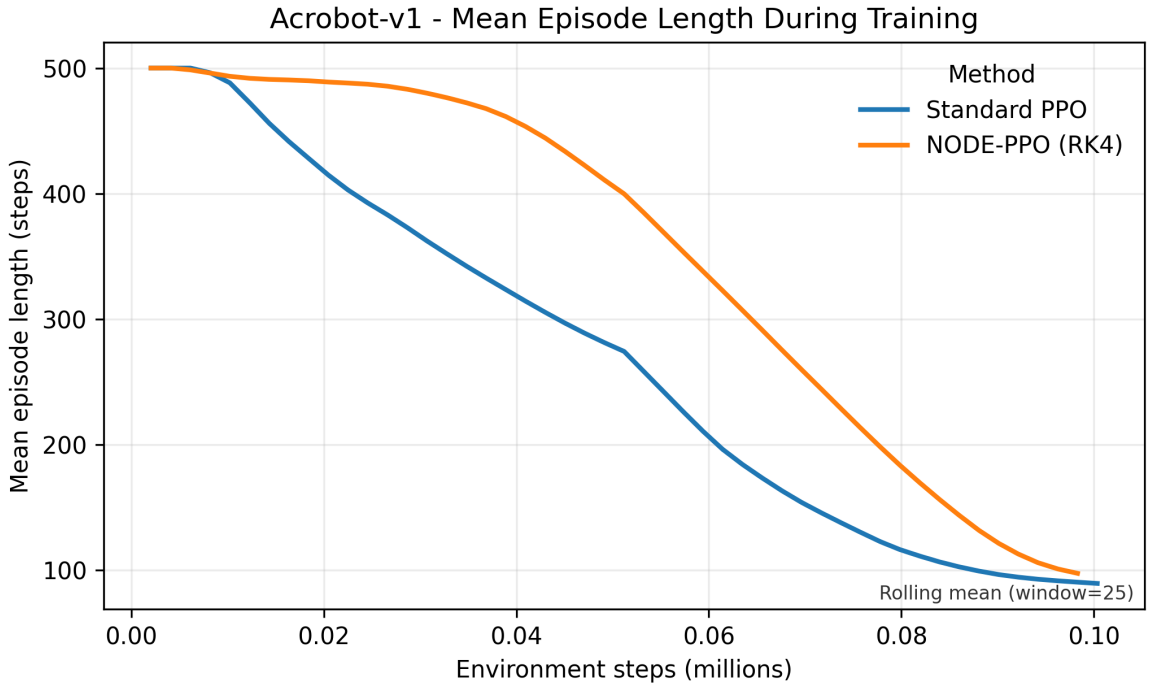


Figure 6: Acrobot-v1 mean episode length during training. Episode length decreases for both methods as policies learn to reach the terminal condition in fewer steps. Standard PPO reduces episode length earlier, while NODE-PPO (RK4) follows a similar downward trend but lags slightly and converges to the "solved" range of episode lengths.

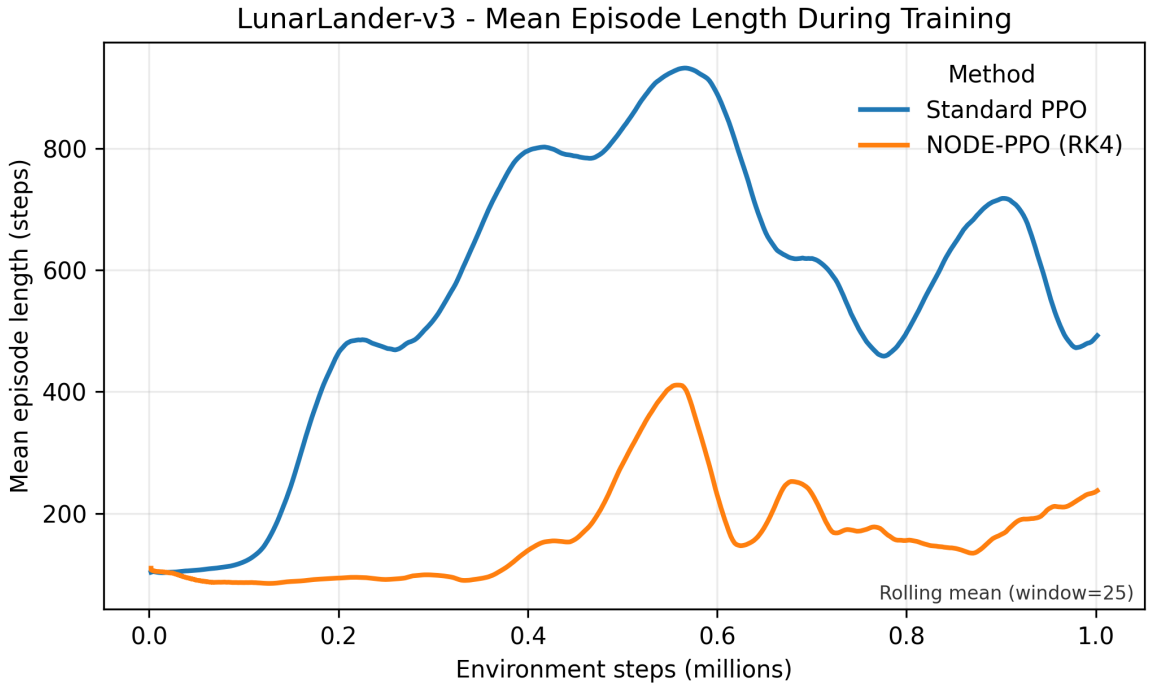


Figure 7: LunarLander-v3 mean episode length during training. Standard PPO exhibits large oscillations in episode length that could indicate inefficient task completion. NODE-PPO (RK4) maintains shorter episodes for much of training with a late upward trend, aligning with its more stable improvement in return despite not maximizing episode length.

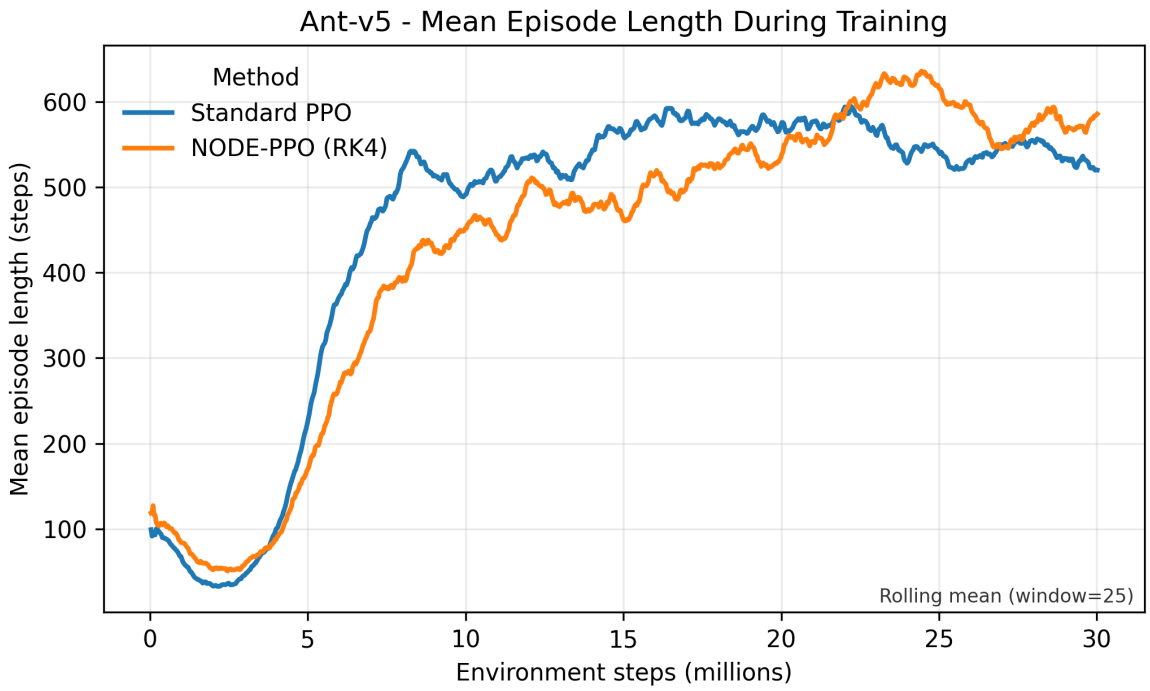


Figure 8: Ant-v5 mean episode length during training. After an early exploration phase with short episodes, both methods increase episode length as they learn to avoid early termination. Standard PPO reaches long episodes earlier, while NODE-PPO (RK4) increases more gradually and overtakes PPO after 22 million steps, showing that, if trained for longer, RK4 could surpass PPO in its reward curve (Figure 4d).

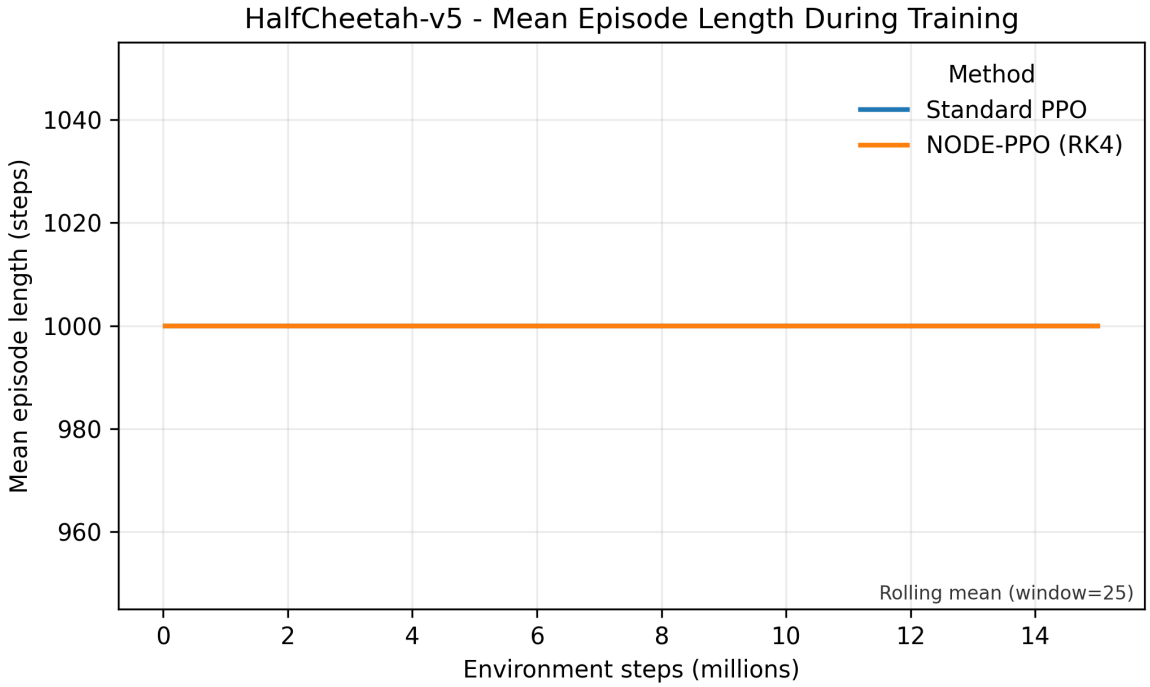


Figure 9: HalfCheetah-v5 mean episode length during training (rolling mean, window=25). Episode length are constant at the environment time limit (1000 steps) for both methods

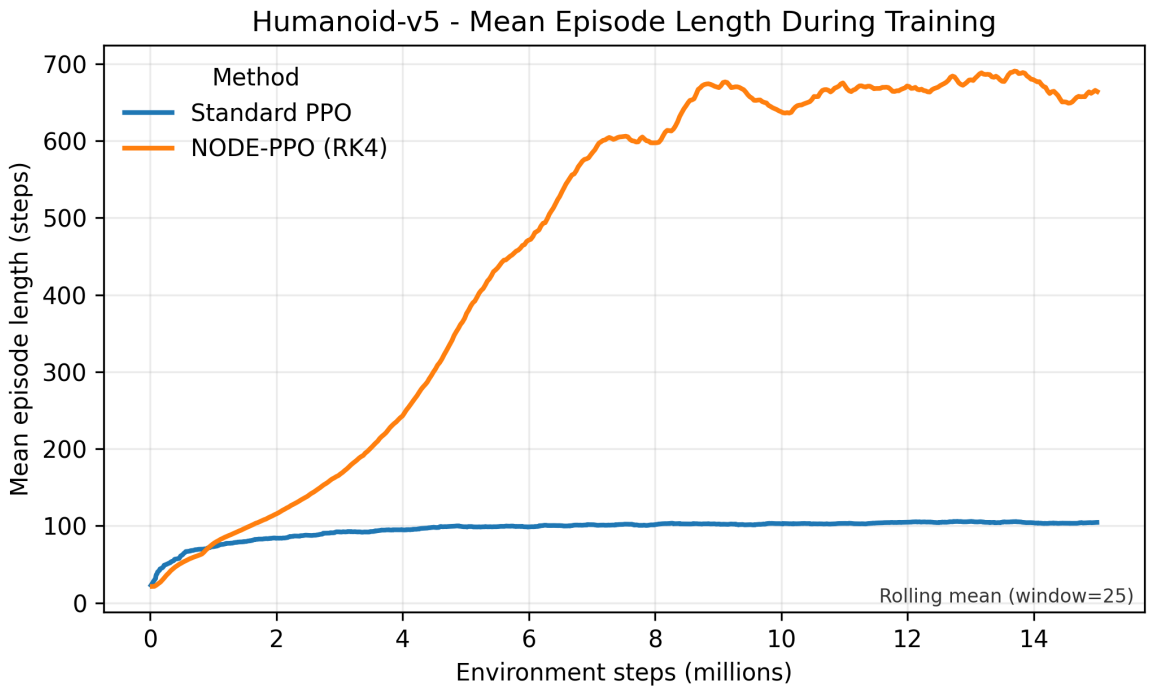


Figure 10: Humanoid-v5 mean episode length during training (rolling mean, window=25). NODE-PPO (RK4) shows a strong upward trend in episode length, reaching several hundred steps and indicating improved balance and reduced unhealthy termination. Standard PPO plateaus at much shorter episodes, consistent with frequent early termination; this matches the large return gap in favor of NODE-PPO (RK4).

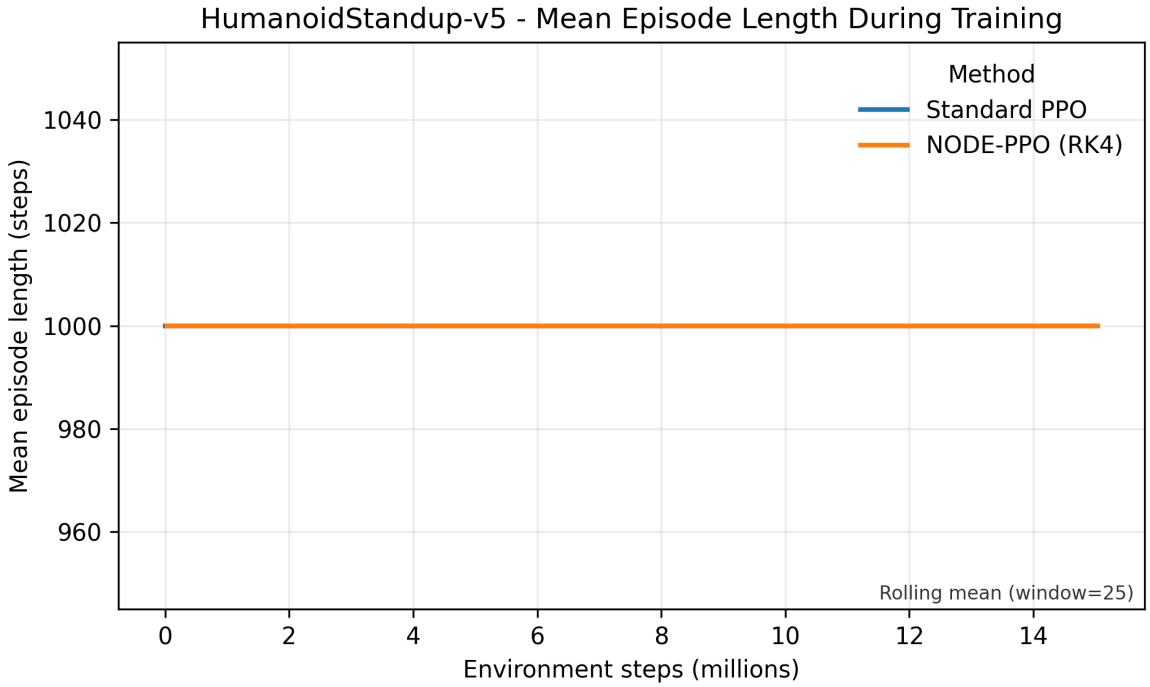


Figure 11: HumanoidStandup-v5 mean episode length during training (rolling mean, window=25). Episode length is stuck at the time limit (1000 steps)