# SF3: Machine Learning
# Interim Report

Raghavendra Narayan Rao

June 13, 2025

**Abstract**

This report investigated the cart-pole system. In the first half, linear and non-linear models were built to predict the cartpole's behaviour. The non-linear model performed best. In the second half, a linear policy was trained to control the cartpole. The policy trained on actual dynamics performed better than that trained on our best model. Lastly, adding noise to the observations and the actual dynamics worsened the policy's ability to control an upright pole.

## 1 Introduction

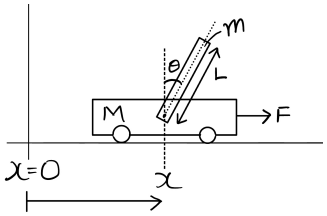


Figure 1: Cartpole (Pendulum on a Cart).

In this project, a cartpole system in Figure 1 is modelled and then used to control the system. A cartpole system in Figure 1 behaves according to the following equations:

$$\begin{aligned} 3\ddot{x}\cos\theta + 2L\ddot{\theta} &= 3g\sin\theta - 6\mu_\theta\dot{\theta}/mL, \\ (m+M)\ddot{x} + \tfrac{1}{2}mL\ddot{\theta}\cos\theta - \tfrac{1}{2}mL\dot{\theta}^2\sin\theta &= F - \mu_x\dot{x}. \end{aligned} \tag{1}$$

This project is split into two phases. In the first phase (Tasks 1 and 2), models that mimic the true carpole behaviour are built and analysed. The initial model built is a linear model. The cartpole system, at very small time steps, behaves linearly. However, when forecasting for long periods of time, the linear model fails at accurately predicting the next state of the cartpole. The model is improved by intRoducing non-linearities through basis functions. Finally, periodicity is also modelled by adding trignometric functions inside the non-linear model. The latest model predicts the true states of the cartpole to high accuracy even 100 steps into the future.

In the second phase (Tasks 3 and 4), a policy is learnt that controls the cartpole system by exerting an external force on the cart. The best performing policy is trained on the actual dynamics. For comparison, another policy is trained using the non-linear model learnt in the previous phase. Finally, noise is added to outputs observed and the cartpole system. The effect this has on the learn policy is studied.

# 2 Task 1

## 2.1 Train Linear Model

$Y = X(T) - X(0)$ (change in state) is defined as the output of our models, where $T$ is the time corresponding to a single 'PerformAction' function, set at 0.1s and represents 50 updates to the state with Euler integration. 500 datapoints were collected, $\{\mathbf{x}_n, \mathbf{y}_n\}_{n=1}^{500}$, where $\mathbf{x}_n$ are randomly initialised states and $\mathbf{y}_n$ are change in states due to a single call to `performAction`. By appending all $\mathbf{x}_n$ and $\mathbf{y}_n$ as rows to form matrices $\mathbf{X}$ and $\mathbf{Y}$, the linear regression problem can be simplified to solving for the least squares solution $\hat{\mathbf{C}}$ to the equation $\mathbf{Y} = \mathbf{XC}$.

The exact form $\hat{\mathbf{C}}$ is $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$. However, the average condition number for $\mathbf{X}^T\mathbf{X}$ is 22.38 (after repeating data collection 30 times). As a rule of thumb [1], $\mathbf{X}^T\mathbf{X}$ is not a well-conditioned matrix as its condition number is larger than 10 (much larger than 1). Therefore, `np.linalg.lstsq`, which does not take matrix inverses, is used instead to solve for $\hat{\mathbf{C}}$. The code for performing linear regression is provided in the appendix (Listing 1).

## 2.2 Forecast using Linear Model

In this task, a forecast of 100 steps into the future is carried out with initial points that result in a simple oscillation as well as complete revolutions. The update equation used for the states is $\mathbf{X}_{n+1} \leftarrow \mathbf{X}_n + \mathbf{X}_n\hat{\mathbf{C}}$. Note that an additional remap of the angle is carried out after every iteration to ensure the angle stays within $[-\pi, \pi]$. The angle diverges if remap is not carried out (for example in complete revolution). Mathematically,

$$\begin{aligned}\lim_{\theta_n \to 0} x_{n+1} &= \lim_{\theta_n \to 0} \hat{C}_{1,1}x_n + \hat{C}_{2,1}\dot{x}_n + \hat{C}_{3,1}\theta_n + \hat{C}_{4,1}\dot{\theta}_n \\ &= \infty \text{ (as } \hat{C}_{3,1} \neq 0)\end{aligned}$$

The linear model forecasts poorly (Figure 2). The linear model converges to a 0 angle, which is an unstable equilibrium, instead of oscillating. The linear model inherently is unable to generalise well due to the non-linearities, such as periodicity, in the true dynamics.

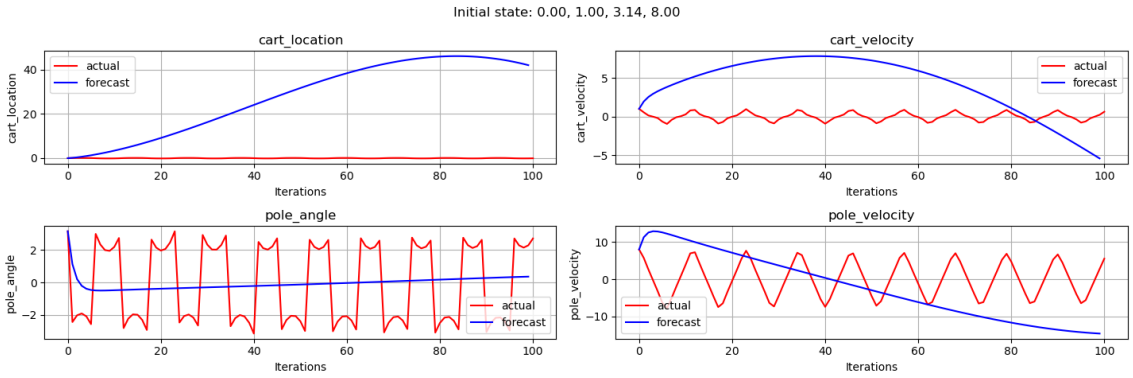


Figure 2: Forecast of cart and pole oscillation

# 3 Task 2: Non-linear models

The cartpole system is not a linear system, As evident from Task 1, the linear model fails at capturing various non-linear behaviours such as periodicity and remaps. In task 2, a non-linear model is built using gaussian

basis functions. The centres of the basis functions are chosen from a subset of the training data (randomly initialised states). The pitfall of the linear model was lack of periodicity. This can be instriduced with the sin of the distance to the center angles instead of the angles.

$$K(X, X') = e^{-\sum_j \frac{\left(x^{(j)} - x'^{(j)}\right)^2}{2\sigma_j^2}}$$

$$(\theta - \theta_c)^2 \rightarrow \sin^2 \frac{(\theta - \theta_c)}{2}$$

The kernel matrix formed from these radial basis functions is then multiplied by a trainable parameter matrix to obtain the model outputs, $K_{NM}\alpha_M = Y_N$. The parameter matrix cannot be found using pseudoinverses due to the condition number problem discussed in Task 1. Therefore, Tikhonov regularisation (Equation 2) will be used to obtain the model parameters.

$$\alpha_M^{(j)} = \left(K_{MN}K_{NM} + \lambda K_{MM}\right)^{-1} K_{MN}Y_N^{(j)} \tag{2}$$

## 3.1 Task 2.1: Build a Non-linear model

Following Equation 2, 4 models are trained where each model parameter vector $\alpha_M^{(j)}$ represents a state variable. Each state variable also has a corresponding length scale parameter ($\sigma_j$) used in the basis functions. The total number of hyperparameters amounts to 5 (four $\sigma$ and one $\lambda$). Based on a sparse grid scan, $\lambda = 10^{-4}$ and $\sigma = std(x) \approx [5.8, 5.7, 1.8, 8.6]$ produced the best loss.

Another setting that can be adjusted is the number if basis centres M. As M approaches the training set size N, the complexity of the model increases. This can potentially improve the model if it is currently underparametrised. however, it will also increase the computation time. Figure 3 shows that the model is underparametrised so increasing M reduces the loss (MSE). M above 1024 has marginal improvement so an M of 1024 and N of 4096 are chosen throughout this report.

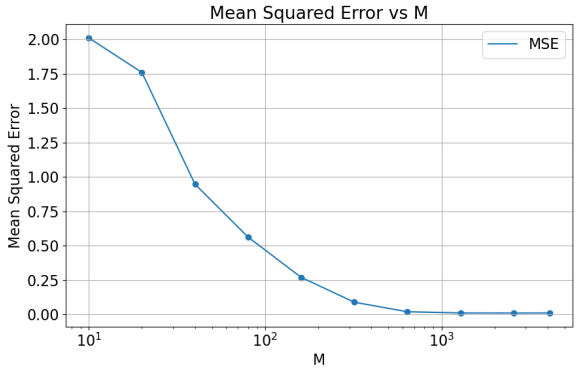


Figure 3: MSE vs M

Figure 4 shows that the model has fit position and angle well but struggles with the velocities. This is to be expected as the current hyperparameter settings are not optimised. Figure 5 shows that the model stops matching the actual dynamics after 4s (40 iterations, 4 cycles).

## 3.2 Task 2.2: Gradient based hyper-parameter tuning

A better way to tune hyperparameters is to perform gradient descent on the loss (MSE). The loss dropped from 0.025 to 0.0045, an 82% drop. This resulted in much better forecasting accuracy as observed in Figure

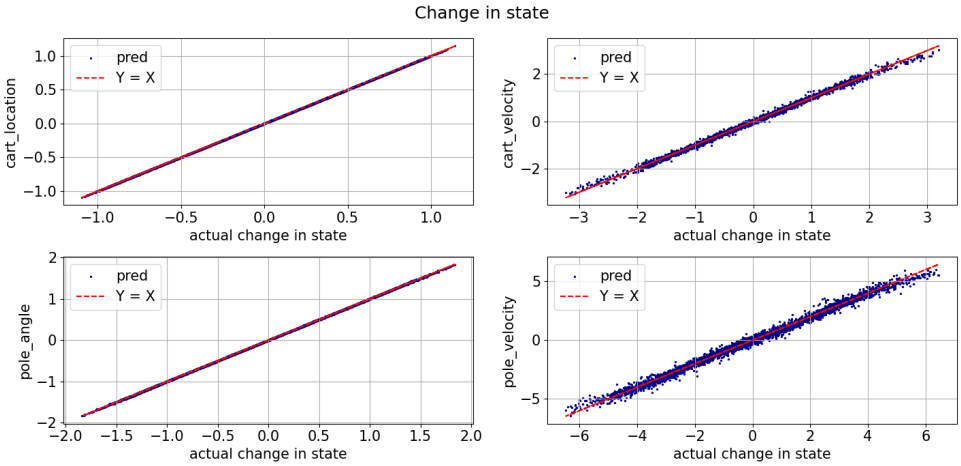


Figure 4: 2D Slice of target and fit

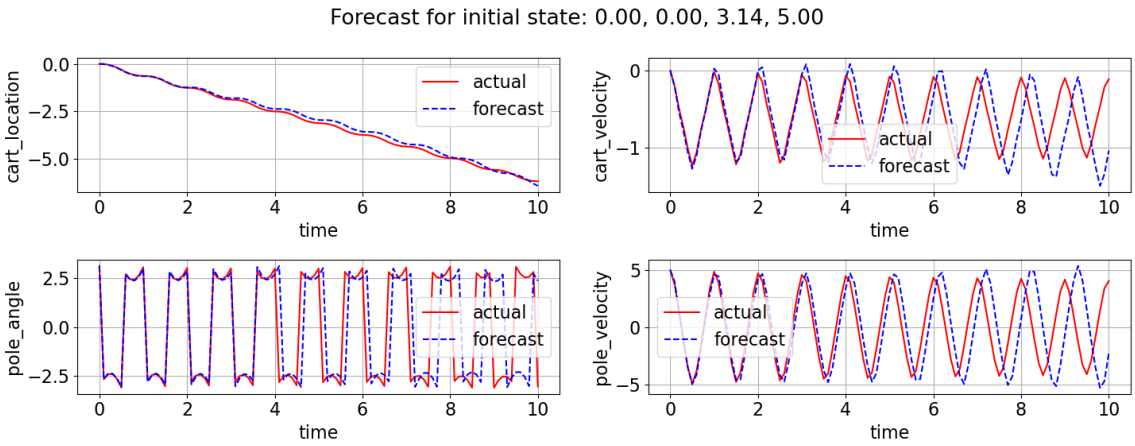


Figure 5: Model rollout

6. The model matches the actual behaviour up to 7s (70 iterations, 7 cycles). The optimal value of $\lambda$ found is $3.2 \times 10^{-5}$. This means that the kernel is not as unstable (high condition number) and hence a lower $\lambda$ can be afforded to inrease the accuracy of the model. Note the tradeoff that a lower $\lambda$ means that the linear equation solution is not as consistent but is more honest to the true solution. The $\sigma$ found are $[7.7, 7.6, 1.0, 6.6]$ which is close to the standard deviation. The angle's scale parameter is also close to 1, the half range of sine.

### 3.3 Task 2.3: Sine and Cosine

in Task 2.2, we only added a sine function to capture the periodic behaviour. Adding both sine and cosine can provide a better representation of periodic data. Therefore, the angle state has been replaced by sin and cos of the angle to create a 5 variable state. Remap is also no longer required as sine and cosine are already $2\pi$ periodic. This improved the model's prediction accuracy. Figure 7 shpws the improved model matching actual behaviour up to 9s (90 iterations, 9 cycles).
The linear model on the other hand, even with sine and cosine functions, are unable to forecast well, showing similar results to Task 1. Therefore, the rest of the report on control focusses on building a policy from the

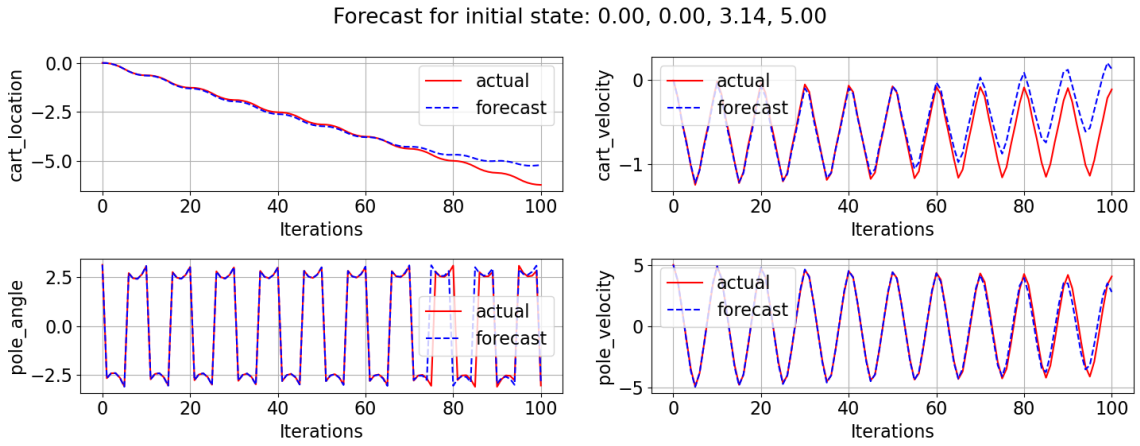


Figure 6: Model rollout

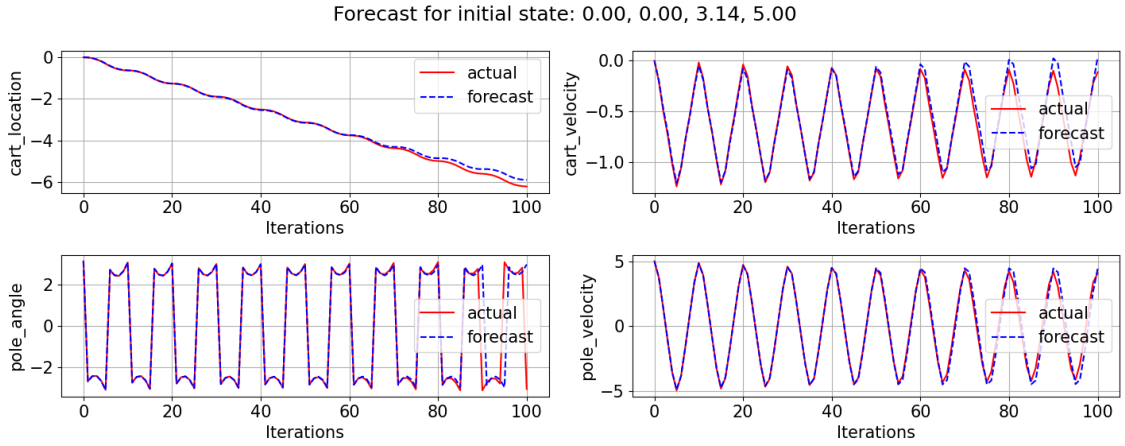


Figure 7: Model rollout

successful non-linear model. The code for training the non-linear model is provided in the appendix (Listing 2)

# 4 Task 3: Control

In this section, a policy is trained to control the cartpole. In typical reinforcement paradigms, a reward function guides the optimisation of the policy. In this report, we instead perform direct policy optimisation. In the cartpole environment, we will define actions as force applied on the cart and the policy returns the best action (appropriate force) required to reach a specified target state.

A linear policy, $Force = P^T X$, will first be trained on the actual cartpole dynamics. This policy represents the best that can be done. Another policy will be trained on our best model of the system (model predicticve control). Both policies are compared to determine the effectiveness of our models.

### 4.1 Task 3.2: Training policy using actual dynamics

To train the policy, a loss function is required. The loss function is defined a sequence of states derived from the policy's actions (25 steps). If $\mathbf{x}^t$ is the target state, then the loss function is defined as the sum over all steps.

$$L(\{\mathbf{x}\}_{n=1}^{N}) = \sum_{n=1}^{N} 1 - e^{\sum_{i=1}^{4} \frac{\left(x_{n,i} - x_i^t\right)^2}{\sigma_i}} \tag{3}$$

Tuning the length scaling parameters $\sigma_i$ is akin to assigning importance to each parameter. If we prioritise the angle to be 0, then the length scale parameter corresponding to that is given a lower value.

Before optimising, it is useful to keep all policy parameters at 0 and sweep one of the parameters to verify the validity of the defined loss function. The sweep also recommends regions where optimal policy parameters can be found. Such a sweep has been plotted in Figure 8. The suggested optimal policy parameters for position, velocity and angular velocities are all negative, indicating a negative feedback strategy adopted by the policy. The interesting finding is that the optimal policy parameter suggested for angle is highly positive. This can be explained by the force pulling on the cartpole in the same direction as the cartpole's tilt, resulting in the cartpole angle stabalising later on.

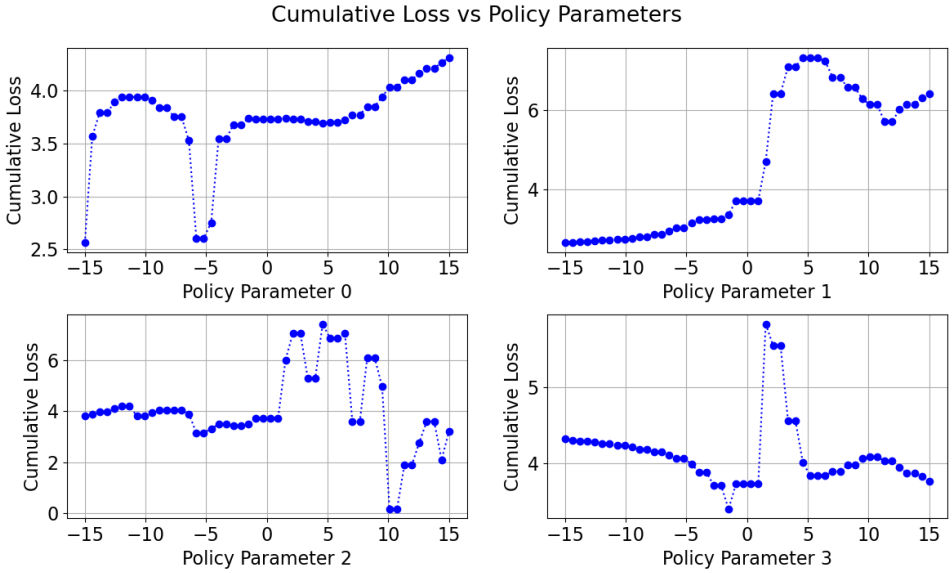


Figure 8: 1-D Scan of Loss against Policy parameters

The policy is now trained on the actual cartpole dynamics. The target state is the upright stationary state, $[0, 0, 0, 0]$. Figure 9 shows the cartpole initially at an almost upright position and stabilised. The scaling parameters that achieved this are also all equal to $[10, 10, 10, 10]$ for the same reasons discussed in Task 3.2. The maximum force during training was restriced to 8 as high forces can result in the cartpole diverging from the unstable equilibrium.

Figure 10 shows the cartpole initially at a downward position and successfully stabilised. The scaling parameters that achieved this are $[10, 10, 4, 10]$. Reducing loss due to angle has to be prioritised to ensure the incorrect policy is not learnt. The incorrect policy learnt when scaling parameters are all nearly equal is to remain in the downward stationary position as the other 3 parameters are minimised to 0. This means that deviation in angle can be compensated for by the reduced loss in other states, which is not desirable. The

maximum force during training was increased to 10 as the initial force required to swing the pole is high. The code for enabling JAX support to cartpole is provided in the appendix (Listing 3)

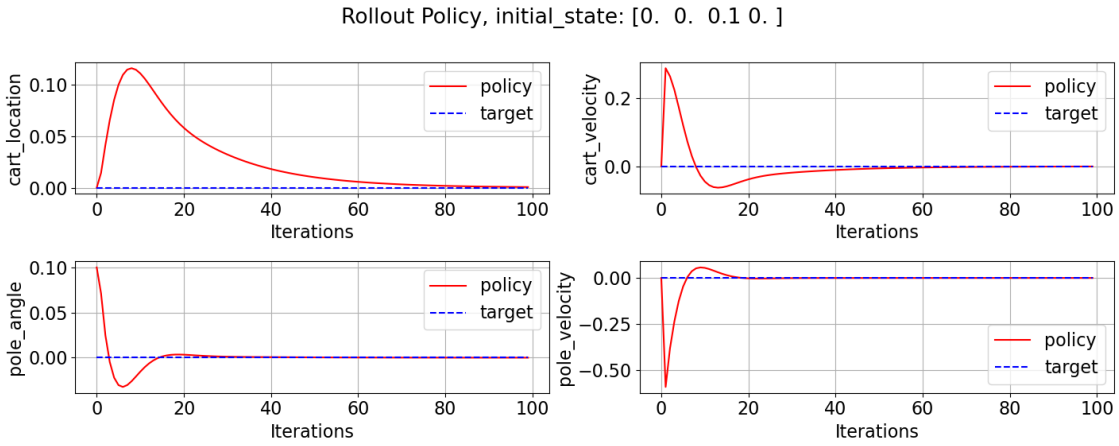


Figure 9: Rollout under linear policy trained on actual dynamics (cartpole starts upright)

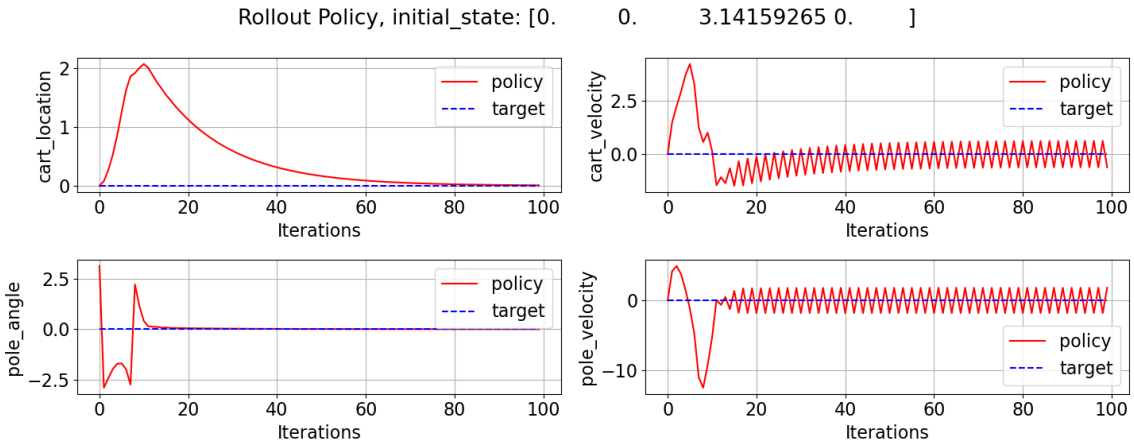


Figure 10: Rollout under linear policy trained on actual dynamics (cartpole starts downward)

## 4.2 Task 3.1 and 3.3: Model Predictive Control

By adding the force as a 5th state variable, the models in Task 2 are updated to predict with forces being applied. The non-linear model's predictions are plotted in Figure 11. For low forces below 5, the model forecast matches real dynamics up to 60 iterations (6s). At a force of 10, the model starts diverging at 30 iterations (30s). Therefore, a maximum force of 10 is set.

Next, a linear policy is trained on this model with the same target state as Task 3.2. Figure 9 shows the cartpole initially at an almost upright position and stabilised. The scaling parameters that achieved this are all equal to $[10, 10, 10, 10]$ for the same reasons discussed in Task 3.2. The maximum force during training was restriced to 8 as high forces can result in the cartpole diverging from the unstable equilibrium.

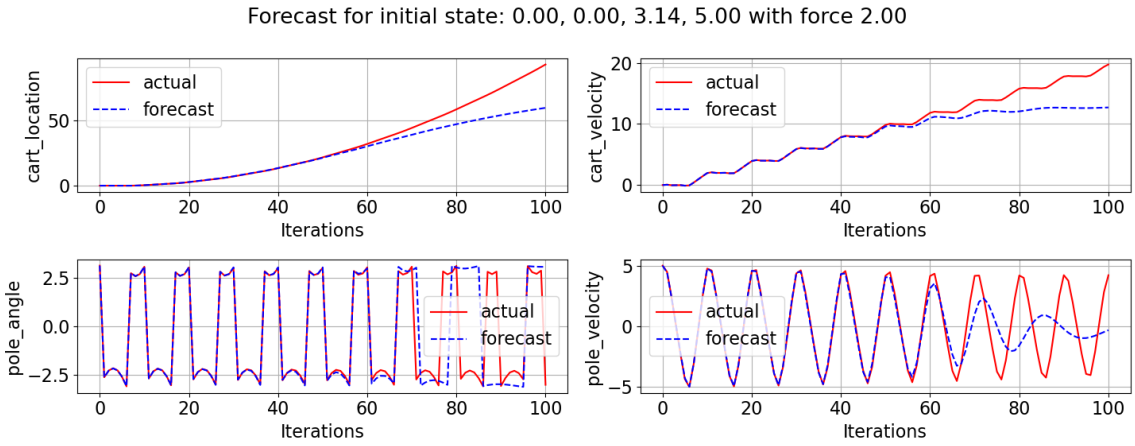


Figure 11: Non-linear model forecast

Figure 10 shows the cartpole initially at a downward position. The target state could not be reached as the position still has a residual of 1 The scaling parameters that achieved this are $[10, 8, 4, 10]$. The lower length scale parameter for velocity is to ensure the cart location stays low. Other configurations result in the cartpole moving off. The maximum force during training was increased to 10 as the initial force required to swing the pole is high. The time horizon had to be reduced to 25 steps as at a force of 10, the non-linear model being used diverges at about 30 steps.

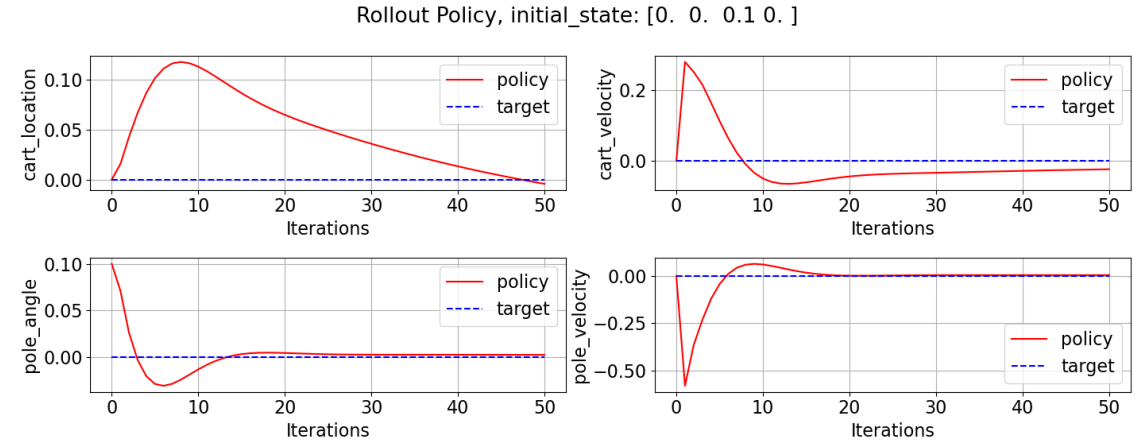


Figure 12: Rollout under linear policy trained on non-liner model (cartpole starts upright)

# 5 Task 4: Sensitivity and stability

If we are developing a policy for real physics, noise should be added into the dynamics. This task studies the effect of adding noise to the real dynamics as well as the non-linear model from Task 3.

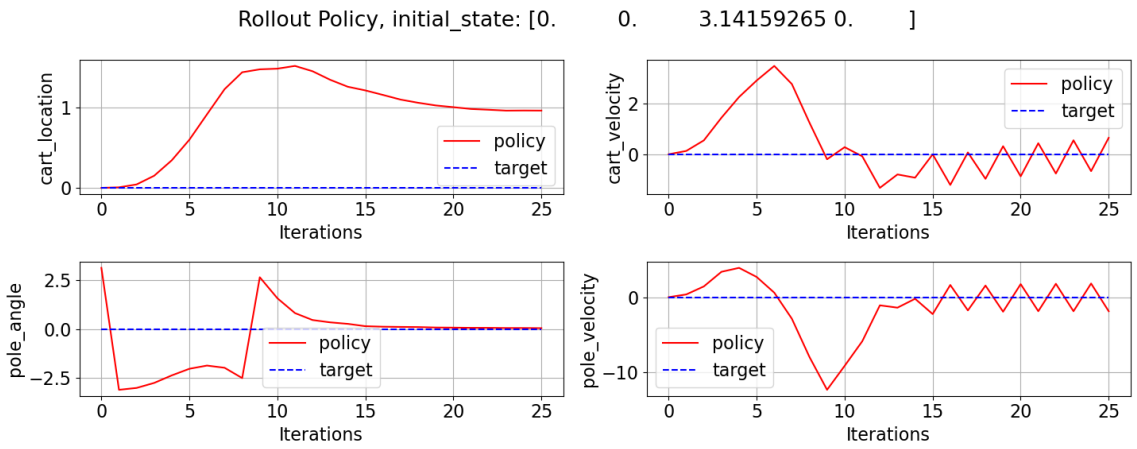


Figure 13: Rollout under linear policy trained on non-liner model (cartpole starts downward)

## 5.1 Task 4.1: Noise in Observed Dynamics

The new observed output (vector) is defined as $\mathbf{y} = \mathbf{x}(T) - \mathbf{x}(0) + \eta$ where $T$ is the time corresponding to a single 'PerformAction' and $\mathbf{x}(t)$ are states. The added noise is assumed to be gaussian with mean 0; $\eta \sim \mathcal{N}(\mathbf{0}, {\sigma_\eta}^2)$. The standard deviation of the noise is assumed to be proportional to that of $\mathbf{y}$ so that noise is distributed fairly to all output dimensions. The scale parameter $k$ where $\sigma_\eta = k\sigma_y$ was picked such that it is the largest for a reasonable policy to still be trained.

The non-linear model is trained on noisy observed data and its outputs are compared with real dynamics in Figure 14. Due to our scaling method, notice that all 4 state variables have a very similar noisy distribution width about the $y = x$ line. To test the prediction accuracy, rollouts were plotted in Figure 15. The model no longer matches the true response after 20 iterations (2s) and diverged after 40 iterations (4s). Therefore, a reasonable time frame of 25s will be used for policy training and prediction.

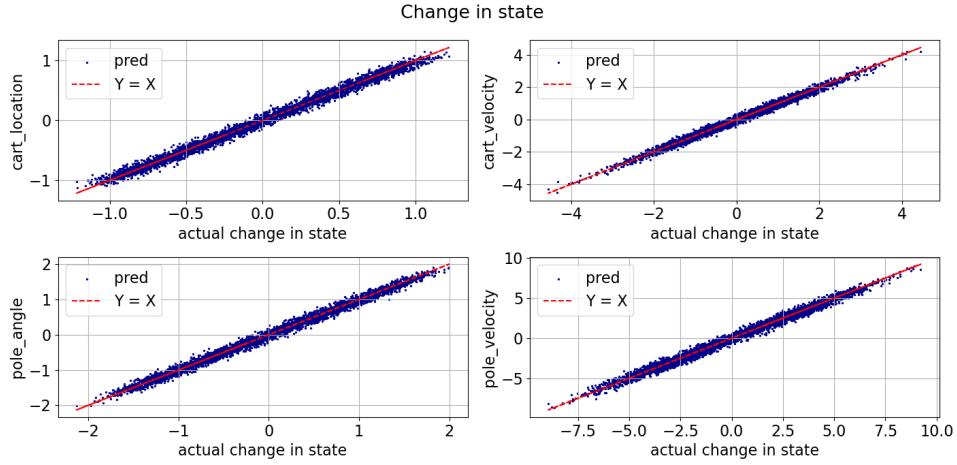


Figure 14: Non-linear model, Predicted vs Actual output (with observed noise)

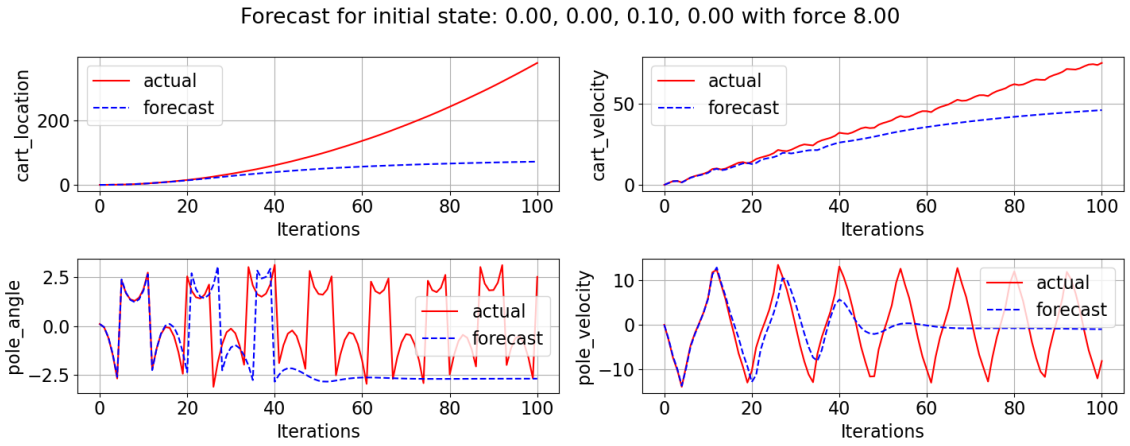


Figure 15: Non-linear model Rollout (with observed noise))

Using the same strategy as Task 3.3, a linear policy was trained on the above non-linear model's rollout. The goal is to achieve the target upright position, $\mathbf{x} = [0, 0, 0, 0]$. Two scenarios are plotted, when the pole starts upright (16) and when the pole starts from downward position (17). For the first scenario, the policy has managed to keep the cartpole oscillating about the target upright position. Note that the y-axis values are very low. However, it has failed to zero the angular velocity. A residue angular velocity remains, which is required for constant oscillation. The noise-free model (Figure 12) was however able to reach the target state.

In the second scenario, The policy has found the required actions to keep the cartpole in an approximately stationary upright position (minor oscillations in velocities). However, this resulted in a deviation in cart position. The length scaling factors in the loss function of the policy used were 10, 8, 4 and 10 for $x$, $\dot{x}$, $\theta$ and $\dot{\theta}$ respectively. Therefore, deviations in position contributed less to the policy loss and hence was not optimised in compensation of the other variables. This is almost identical to the noise-free model (Figure 13). .

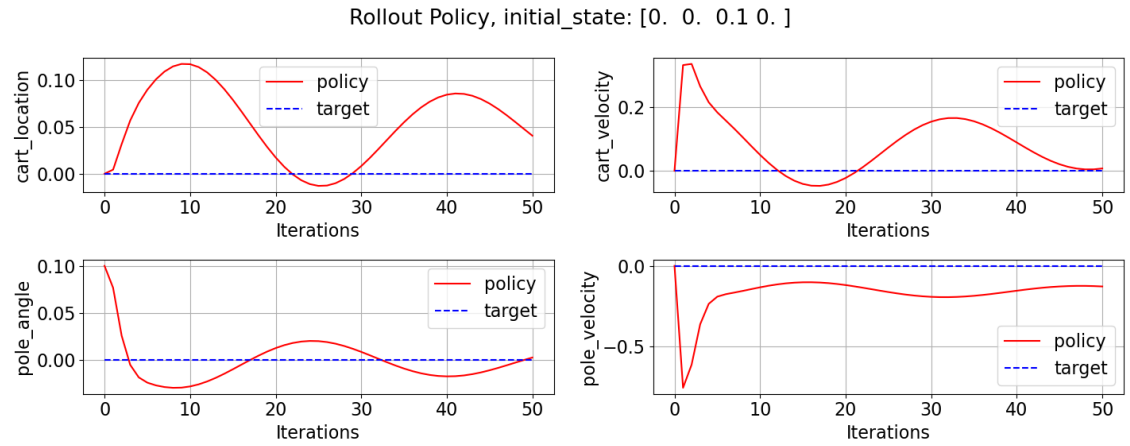


Figure 16: Linear policy trained on non-linear model (Pole initially upright)

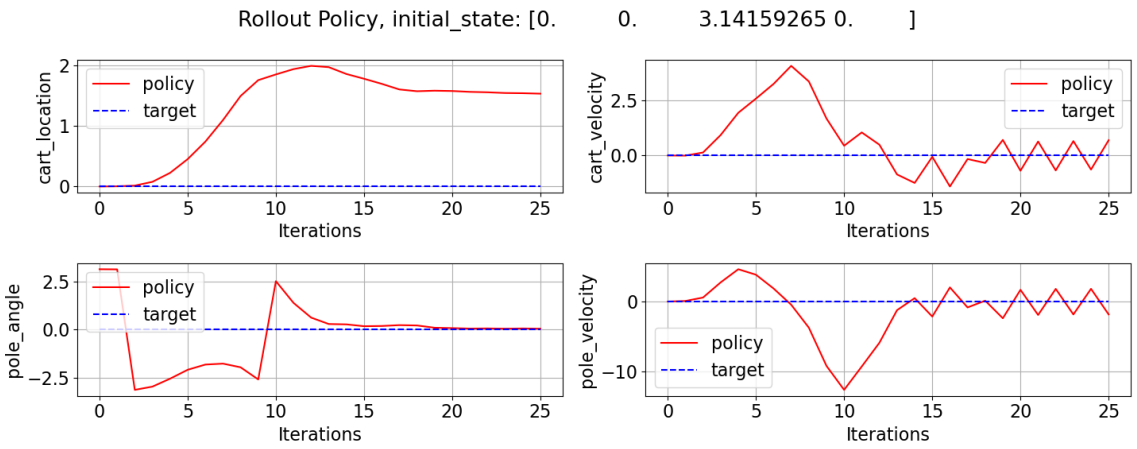


Figure 17: Linear policy trained on non-linear model (Pole initially downward)

## 5.2 Task 4.2: Noise in Real Dynamics

To simulate noise in real dynamics, a gaussian random variable was added directly to the state in 'PerformAction'. However, the gaussian variable used here is $\eta \sim \mathcal{N}(\mathbf{0}, 0.05^2\mathbf{I})$. All state variables being added the same noise means that the velocities are at a higher signal to noise ratio (observed in 18). In Euler integration, positions and angles are integrated from velocities so they are expected to have higher SNR than velocities.

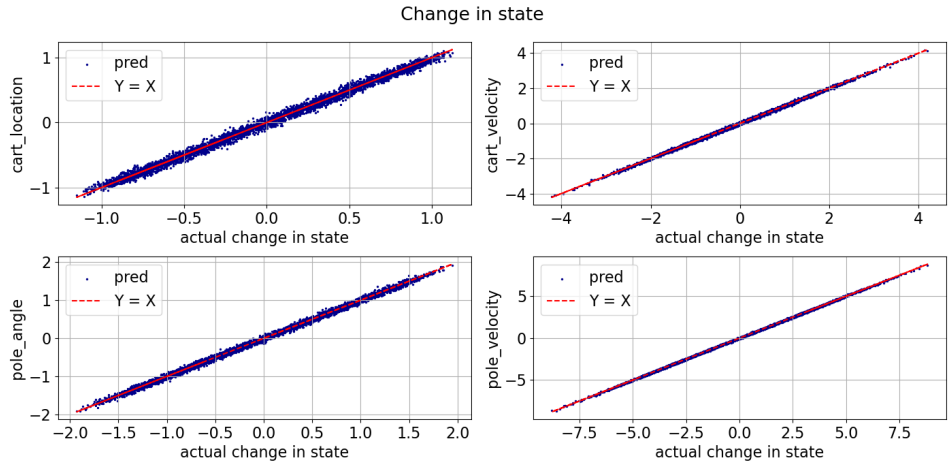


Figure 18: Non-linear model, Predicted vs Actual (with noise in actual dynamics)

The policy trained on this new model performed worse than that trained on observed noise. The rollout of a cartpole starting at nearly upright position is shown in Figure 18. This policy managed to keep the pole upright but the cart is moving at constant velocity. To achieve an upright angle, the policy compensated for position and velocity. The policy, however, performed very similarly when the cartpole began at the downward position. The code for adding noise to cartpole is provided in the appendix (Listing 4)

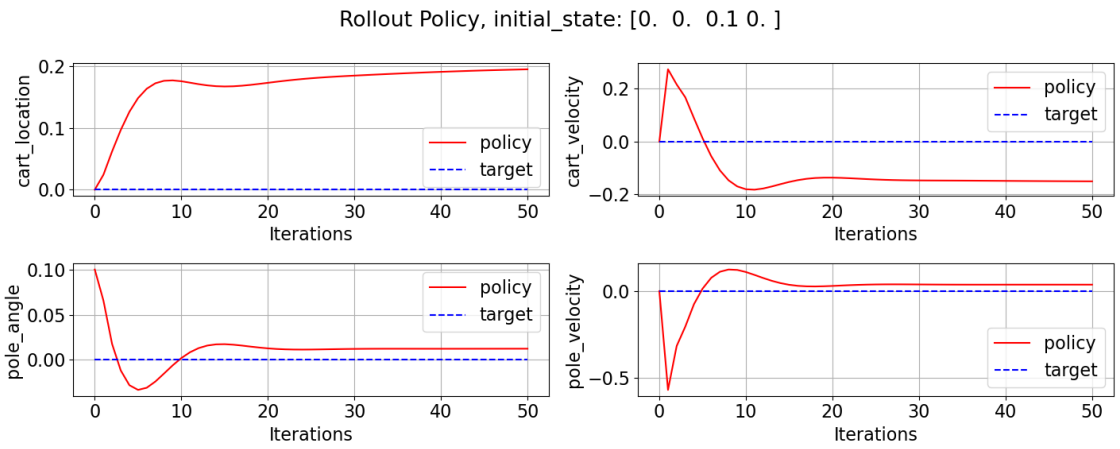


Figure 19: Linear policy trained on non-linear model (Pole initially downward)

# 6 Conclusions

In Task 1, it was observed that the linear model performed poorly. It converged to an unstable equilibrium instead of simple oscillations. The linear model is not robust enough to capture periodic motion.

In Task 2, the model improved when non-linearities through the basis functions were added. Tuning the hyperparameters with gradients also improved the model. Lastly, replacing angle for sine and cosine produced the best model. This model is able to model oscillations very well, up to 9s (90 iterations, 9 cycles).

In task 3, two linear policies were trained. The first policy, tarined on actual dynamics, performed the best. The second policy, trained on our non-linear model, failed to converge past a time frame of 25-50s.

In task 4, noise was added to observations and cartpole dynamics. Both resulted in policy degradation, with the latter being unable to meet the target state when the pole is initialised as being upright.

# References

[1] *Matrix Condition Number.* Nist.gov, 2025 (Accessed 21 May 2025). Available at www.itl.nist.gov.

# 7 Appendix

```python
def generate_data_random(num_steps, initial_force):
    env = CartPole(visual=False)
    env.reset()
    x_data = {
        'cart_location': [],
        'cart_velocity': [],
        'pole_angle': [],
        'pole_velocity': []
    }
    y_data = {
        'cart_location': [],
        'cart_velocity': [],
        'pole_angle': [],
        'pole_velocity': []
    }
    for i in range(num_steps):
        initial_state = [np.random.uniform(-10, 10), np.random.uniform(-10, 10),
                         np.random.uniform(-np.pi, np.pi), np.random.uniform(-15, 15)]
        env.reset()
        env.setState(initial_state)
        env.performAction(initial_force)
        # remap the angle to be between -pi and pi
        env.remap_angle()

        next_state = env.getState()
        x_data['cart_location'].append(initial_state[0])
        x_data['cart_velocity'].append(initial_state[1])
        x_data['pole_angle'].append(initial_state[2])
        x_data['pole_velocity'].append(initial_state[3])

        y_data['cart_location'].append(next_state[0] - initial_state[0])
        y_data['cart_velocity'].append(next_state[1] - initial_state[1])
        y_data['pole_angle'].append(next_state[2] - initial_state[2])
        y_data['pole_velocity'].append(next_state[3] - initial_state[3])

    X = convert_dict_to_array(x_data)
    Y = convert_dict_to_array(y_data)
    return X, Y

def linear_regression(X, Y, intercept=False):
    """
    Args:
        X (numpy.ndarray): The input data.
        Y (numpy.ndarray): The output data.

    Returns:
        W (numpy.ndarray): The parameters of the model
        Y_pred (numpy.ndarray): The predicted output data
    """
    # Add a column of ones to X for the intercept term
    if intercept:
        X = np.hstack((np.ones((X.shape[0], 1)), X))

    # Calculate the weights using the normal equation
    # W = np.linalg.inv(X.T @ X) @ X.T @ Y
    W = np.linalg.lstsq(X, Y)[0]

    # Make predictions
    Y_pred = X @ W

    return W, Y_pred
```

Listing 1: Linear regression

```python
@jax.jit
def kernel_expanded_j(X, X_prime, sigma):
    # create new X where 2 additional dimensions are added,
    # replacing the angle with sin and cos
    # angle dimension is removed
    X_new = jnp.hstack((X[:,0:2], jnp.sin(X[:, 2:3]), jnp.cos(X[:, 2:3]), X[:, 3:]))
    X_prime_new = jnp.hstack(
        (X_prime[:,0:2], jnp.sin(X_prime[:, 2:3]),
        jnp.cos(X_prime[:, 2:3]), X_prime[:, 3:])
    )  # (M, D+1)

    X_e = jnp.expand_dims(X_new, axis=1)  # (N, 1, D+1)
    X_prime_e = jnp.expand_dims(X_prime_new, axis=0)  # (1, M, D+1)

    diff = X_e - X_prime_e  # (N, M, D+1)
    scaled_squared_diff = (diff ** 2)/(2 * sigma ** 2) # (N, M, D+1)

    K = jnp.exp(-jnp.sum(scaled_squared_diff, axis=-1))  # (N, M)
    return K

N_train, N_test, M = 4096, 2048, 1024

X, Y = generate_data_random(num_steps=N_train+N_test, initial_force=0)
X, Y = jnp.array(X), jnp.array(Y)
X_prime = X[:M]

X_train, Y_train, X_val, Y_val = X[:N_train], Y[:N_train], X[-N_test:], Y[:N_train]

def loss(parameters):
    lamb = parameters[0]
    sigma = parameters[1:]
    # train model
    alpha, X_prime, _= train_nonlinear_models_j(
        X_train, Y_train, M=M, lamb=lamb,
        sigma=sigma, kernel_fn=kernel_expanded_j
    )
    # predict using validation set
    K_val = kernel_expanded_j(X_val, X_prime, sigma)
    Y_pred = K_val @ alpha

    mse = jnp.mean((Y_val - Y_pred) ** 2)
    return mse

# create a function that calculates the gradient
# of the loss function using jax.grad
grad_loss = jax.grad(loss)

initial_lamb = 1E-4

x_sigma = get_std(X)
# initial_sigma = jnp.array([6, 6, 0.5, 0.5, 6])
initial_sigma = jnp.array([x_sigma[0], x_sigma[1], std_sine, std_cos, x_sigma[-1]])
initial_hyperparameters = jnp.array(
    [initial_lamb, initial_sigma[0], initial_sigma[1],
    initial_sigma[2], initial_sigma[3], initial_sigma[4]]
)

losses = [loss(initial_hyperparameters)]

bounds = [(1E-6, 1E-1)] + [(0, 10)] * 5  # bounds for lamb and sigma
res = scipy.optimize.minimize(loss, x0=initial_hyperparameters, method='L-BFGS-B', jac=
    grad_loss, bounds=bounds)
```

Listing 2: Train Non-linear model

```
g = 9.8  # gravity
mass_cart = 0.5
mass_pole = 0.5
mu_c = 0.001
mu_p = 0.001
l = 0.5  # pole length
max_force = 5.0
delta_time = 0.1
sim_steps = 50
dt = delta_time / sim_steps

@jax.jit
def remap_angle2(theta):
    return jnp.mod(theta + jnp.pi, 2 * jnp.pi) - jnp.pi

@jax.jit
def cartpole_dynamics(state, force, max_force=max_force):
    force = max_force * jnp.tanh(force/max_force)
    x, x_dot, theta, theta_dot = state

    s = jnp.sin(theta)
    c = jnp.cos(theta)
    m = 4.0 * (mass_cart + mass_pole) - 3.0 * mass_pole * c**2

    cart_accel = (2.0 * (l * mass_pole * theta_dot**2 * s + 2.0 * (force - mu_c * x_dot)) -
                  3.0 * mass_pole * g * c * s + 6.0 * mu_p * theta_dot * c / l) / m

    pole_accel = (-3.0 * c * (2.0 / l) * (l / 2.0 * mass_pole * theta_dot**2 * s + force -
    mu_c * x_dot) +
                  6.0 * (mass_cart + mass_pole) / (mass_pole * l) *
                  (mass_pole * g * s - 2.0 / l * mu_p * theta_dot)) / m

    x_dot_new = x_dot + dt * cart_accel
    theta_dot_new = theta_dot + dt * pole_accel
    theta_new = theta + dt * theta_dot_new
    x_new = x + dt * x_dot_new

    return jnp.array([x_new, x_dot_new, remap_angle2(theta_new), theta_dot_new])
    # return jnp.array([x_new, x_dot_new, remap_angle2(theta_new), theta_dot_new])

@jax.jit
def cartpole_step(state, force, max_force=max_force):
    def body_fn(i, state):
        return cartpole_dynamics(state, force, max_force=max_force)

    return jax.lax.fori_loop(0, sim_steps, body_fn, state)
```

Listing 3: JAX support for cartpole

```
def performAction_noise(self, action, std):
        # prevent the force from being too large
        force = self.max_force * np.tanh(action/self.max_force)

        # integrate forward the equations of motion using the Euler method
        for step in range(self.sim_steps):
            s = np.sin(self.pole_angle)
            c = np.cos(self.pole_angle)
            m = 4.0*(self.cart_mass+self.pole_mass)-3.0*self.pole_mass*(c**2)

            cart_accel = (2.0*(self.pole_length*self.pole_mass*(self.pole_velocity**2)*s
    +2.0*(force-self.mu_c*self.cart_velocity))\
                -3.0*self.pole_mass*self.gravity*c*s + 6.0*self.mu_p*self.pole_velocity*c/
    self.pole_length)/m

            pole_accel = (-3.0*c*(2.0/self.pole_length)*(self.pole_length/2.0*self.pole_mass
    *(self.pole_velocity**2)*s + force-self.mu_c*self.cart_velocity)+\
                6.0*(self.cart_mass+self.pole_mass)/(self.pole_mass*self.pole_length)*\
                (self.pole_mass*self.gravity*s - 2.0/self.pole_length*self.mu_p*self.
    pole_velocity) \
                )/m

            # Update state variables
            dt = (self.delta_time / float(self.sim_steps))
            # Do the updates in this order, so that we get semi-implicit Euler that is
    simplectic rather than forward-Euler which is not.
            self.cart_velocity += dt * cart_accel
            self.pole_velocity += dt * pole_accel
            self.pole_angle    += dt * self.pole_velocity
            self.cart_location += dt * self.cart_velocity

        noise = np.random.normal(0, std, 4)
        self.cart_location += noise[0]
        self.cart_velocity += noise[1]
        self.pole_angle    += noise[2]
        self.pole_velocity += noise[3]

        if self.visual:
            self._render()
```

Listing 4: Add noise to cartpole