Cambridge University Engineering Department
Engineering Tripos Part IIA
PROJECTS: Interim and Final Report Coversheet

IIA Projects

TO BE COMPLETED BY THE STUDENT(S)

Project:	SF3 Machine Learning
Title of report:	SF3 Machine Learning: Final Report ~~Group Report~~ / Individual Report (delete as appropriate)

Name(s): (capitals)	crsID(s):	College(s):
JEEVAN SINGH BHOOT	jsb212	Trinity

Declaration for: ~~Interim Report 1~~ / ~~Interim Report 2~~ / Final Report (delete as appropriate)

I/~~we~~ confirm that, except where indicated, the work contained in this report is my/~~our~~ own original work.

Instructions to markers of Part IIA project reports:

Grading scheme

Grade	A*	A	B	C	D	E
Standard	Excellent	Very Good	Good	Acceptable	Minimum acceptable for Honours	Below Honours

Grade the reports by ticking the appropriate guideline assessment box below, and provide feedback against as many of the criteria as are applicable (or add your own). Feedback is particularly important for work graded C-E. Students should be aware that different projects and reports will require different characteristics.

Penalties for lateness: Interim Reports: 3 marks per weekday; Final Reports: 0 marks awarded – late reports not accepted.

Guideline assessment (tick one box)

A*/A	**A/B**	**B/C**	**C/D**	**D/E**

Marker:		Date:	

Delete (1) or (2) as appropriate (for marking in hard copy – different arrangements apply for feedback on Moodle):

(1) Feedback from the marker is provided on the report itself.

(2) Feedback from the marker is provided on second page of cover sheet.

	Typical Criteria	Feedback comments
Project Skills, Initiative, Originality	Appreciation of problem, and development of ideas	
	Competence in planning and record-keeping	
	Practical skill, theoretical work, programming	
	Evidence of originality, innovation, wider reading (with full referencing), or additional research	
	Initiative, and level of supervision required	
Report	Overall planning and layout, within set page limit	
	Clarity of introductory overview and conclusions	
	Logical account of work, clarity in discussion of main issues	
	Technical understanding, competence and accuracy	
	Quality of language, readability, full referencing of papers and other sources	
	Clarity of figures, graphs and tables, with captions and full referencing in text	

University of Cambridge

CUED IIA Project

SF3 Machine Learning: Final Report

Jeevan Singh Bhoot, jsb212
Trinity College

June 10, 2022

1 Introduction

This report focuses on the entirety of the four-week SF3 Machine Learning project, as part of the CUED IIA course, which involved investigating an inverted pendulum system (as shown in Figure 1) through a software simulation of a 'cart-pole'. The aim of the project was to develop a data-driven controller that could balance the pendulum on its unstable equilibrium, where the pole is vertically upwards ($\theta = 0$).

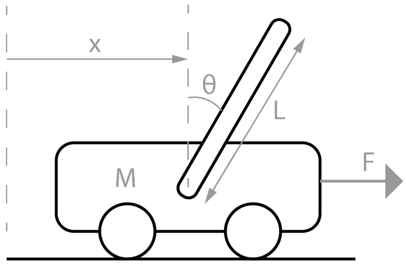


Figure 1: Diagram of an inverted pendulum on a cart ('cart-pole').

2 Week 1

The first week of the project involved familiarising oneself with the provided code, and using such code to simulate rollouts. Data for rollouts from different initial states was collected and analysed, and then used to train linear models, which predicted the change in states after one step.

2.1 Task 1.1

The initial task involved producing code to simulate a 'rollout'; a run of the cart-pole system for a specified number of steps, given a set of initial conditions, specifying the cart's position (x), the cart's velocity (x), the pole's angle (θ), and the pole's angular velocity (θ). Plots of the system's variables as functions of time were produced (although omitted from this report); time evolutions of the system's state for different initial conditions resulted in different mechanical behaviours: simple oscillation about the stable equilibrium ($X = [x, \dot{x}, \theta, \dot{\theta}] = [0, 0, \pi, 0]$), and complete rotation of the pole.

2.2 Task 1.2

This task involved exploring the change in the system's state after a singular call to the `performAction()` function.

2.2.1 Y = X(1) = State after 1 step

The system was initialised in a random state ($[4.2, -2.3, -1.7, 3.1]$), and then a scan across each variable in the initial state was conducted (one-by-one).

2.2.2 Y = X(1) - X(0) = Difference in states after 1 step

Here Y was set as the difference in states after 1 step (i.e. one call of the `performAction()` function). The same scans were conducted (from the same random initial state in 2.2.1) and plots of Y as a function of the scan were produced, as shown in Figure 2.

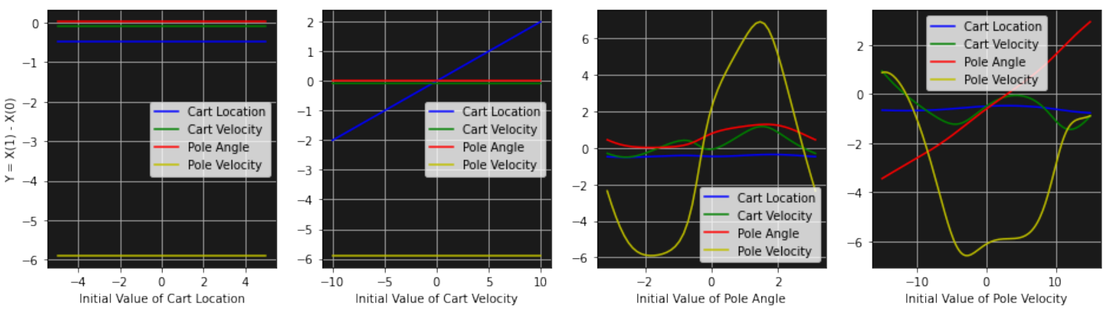


Figure 2: Plots of $Y = X(1) - X(0)$ as a function of scans over initial settings of the parameters (X).

In addition to the scans across single variables, scans across the initial settings of two variables (with the other two variables held constant) was conducted. For these sweeps across two variables, contour plots were produced for $Y = X(1) - X(0)$, shown in Figure 3. In total, 24 sweeps across two variables were conducted - however, only 12 have been shown; this set of 12 is representative of the whole, and shows that the data consists of both linearities and non-linearities. It can be seen clearly in the plots of Figures 2 and 3 that the initial value of the cart location has no impact on the parameters after one step. This was expected as the mechanical system in invariant under translation - shifting the cart-pole to a different location has no impact on the dynamics. It can also be seen (in Figure 3) that the pole angle and velocity have a non-linear impact on the system's dynamic state - the contours are visibly non-linear. The initial cart velocity has a linear impact on the four state parameters (which can be seen in the second plot of Figure 2) - however, this does not mean that the final cart velocity can be modelled linearly, as it still depends non-linearly on the pole's angle and angular velocity.

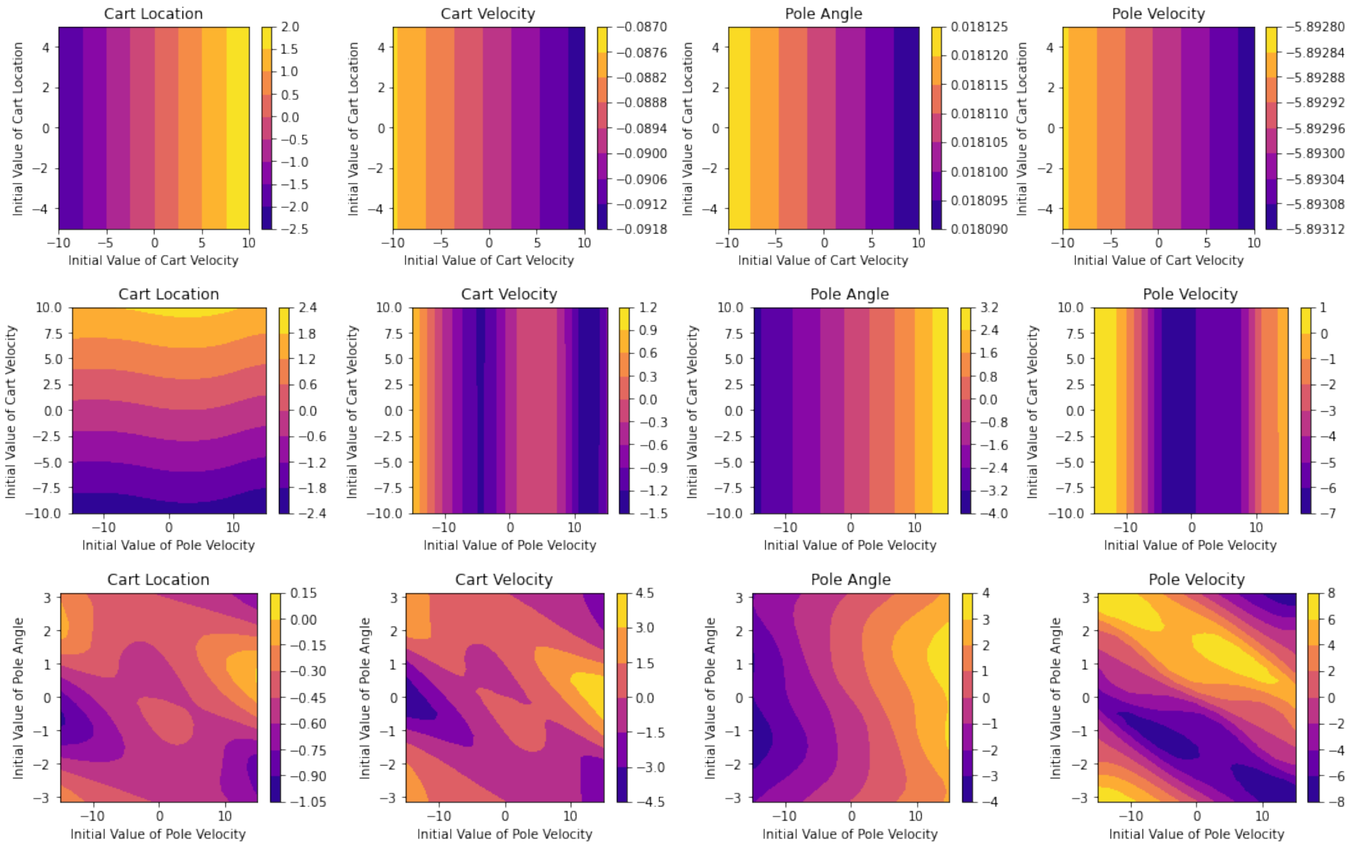


Figure 3: Contour plots for the change in each state parameter as a function of scans across initial values of two of the parameters.

2.3 Task 1.3

Although it had been deduced in task 1.2 that the system's state could not be modelled linearly, a simple linear model was used to predict the system's change in state, as a baseline for future comparisons.

Data for the predictive model was gathered by simulating the dynamic model; the system was initialised in a completely random state (within suitable ranges for each parameters) and run for a singular step. The inputs to the predictive model were the random initial states ($X(0)$), and the outputs the changes in state after one step ($Y = X(1) - X(0)$). 500 data points were obtained, with 20% set aside for the test set.

With the data set, linear regression was conducted, with the use of scitkit-learn (a machine learning library). The predictions of the linear regression model on the test set were compared to the ground truths, shown in Figure 4. The plots show the predicted final state against the true final state, with good predictions shown by straight lines. It can be seen that the predictions for the cart position and pole angle are superior over that of the cart velocity and pole angular velocity. This is expected as the velocities are non-linear functions of the state, and therefore could not be modelled accurately by a linear predictive model.

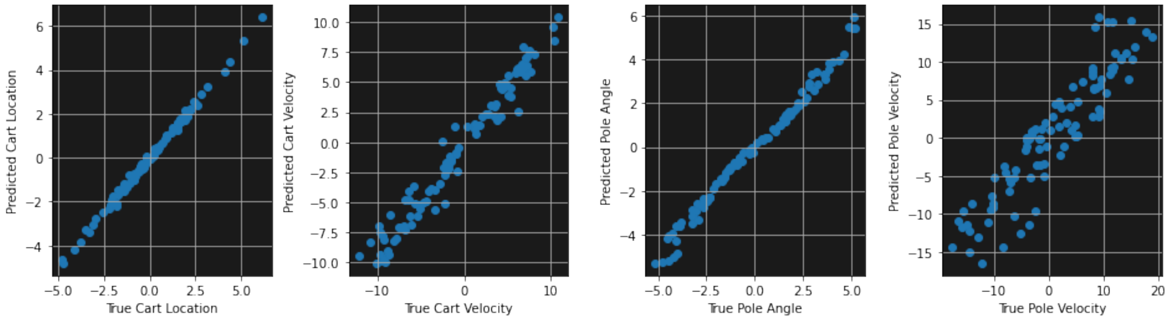


Figure 4: Predicted state parameter against true state parameter.

Figure 5 shows the scans from section 2.2.2 repeated to include the predicted change in state as a function of scans across the state variables. Again it can be seen that predictions are significantly worse for the velocities compared to the position and angle, and that the model can not fit to the non-linearities.

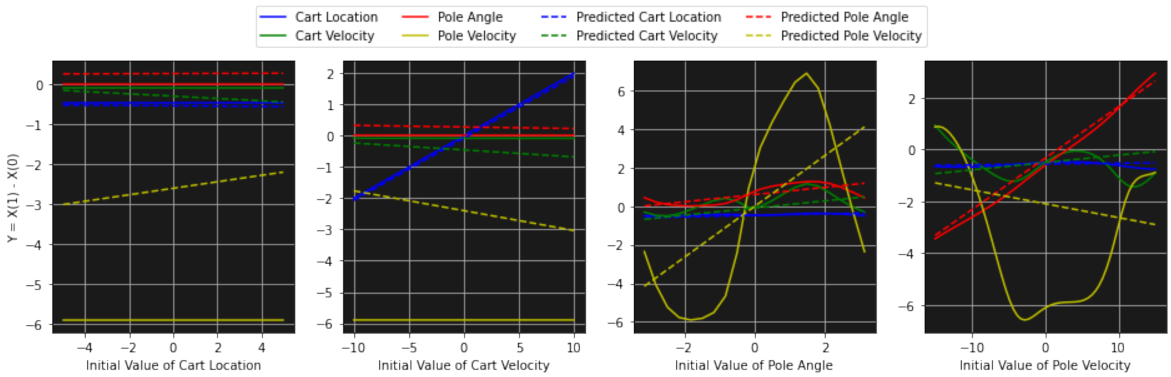
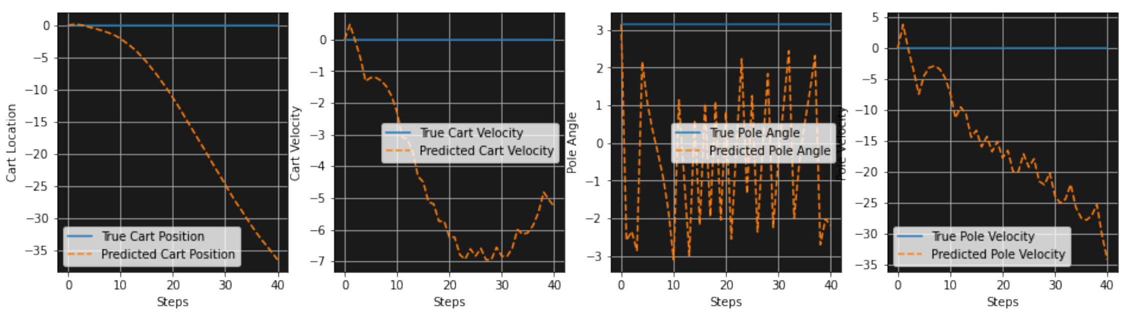


Figure 5: Plots of predicted and true $Y = X(1) - X(0)$ as a function of scans over initial settings of the parameters (X).

2.4 Task 1.4

To gain further insight into the predictive performance of the linear regression model, it was used to predict the time evolution of the cart-pole system from a random set of initial conditions. The predicted time evolution of the inverted pendulum was compared to the true dynamics. Figure 6 shows the true and predicted time evolutions of the cart-pole system from three different initial conditions: [0, 0, π, 0], [-0.169, 9.607, 2.557, -14.155], [0.738, -0.467, 3.068, 14.384], from top row to bottom row, respectively. It can be clearly seen that the linear model was not performing well - it was not accurately predicting the dynamics of the system over a number of steps. Even for a single step, the model was not so accurate; over a number of steps, the error in prediction at each step accumulates. This can be seen in the diverging cart positions, especially for the initial condition [0, 0, π, 0]. At the stable equilibrium, one would expect the cart-pole to remain at the equilibrium. However, the linear regression model predicts the system to accelerate and move away from its initial position, without any physical input. This model is clearly not suitable for further use.

Further experimentation was conducted by expanding the size of the dataset from 500 samples to 10000 samples. The model trained on the expanded dataset was used to predict time evolutions from the same initial states in Figure 6. There was a very slight improvement in performance. However, divergence still occurred - one would expect a greater boost in performance for the dataset size increase; the predictive performance was still poor. The model was underfitting the training data, as it could not map to the non-linearities.



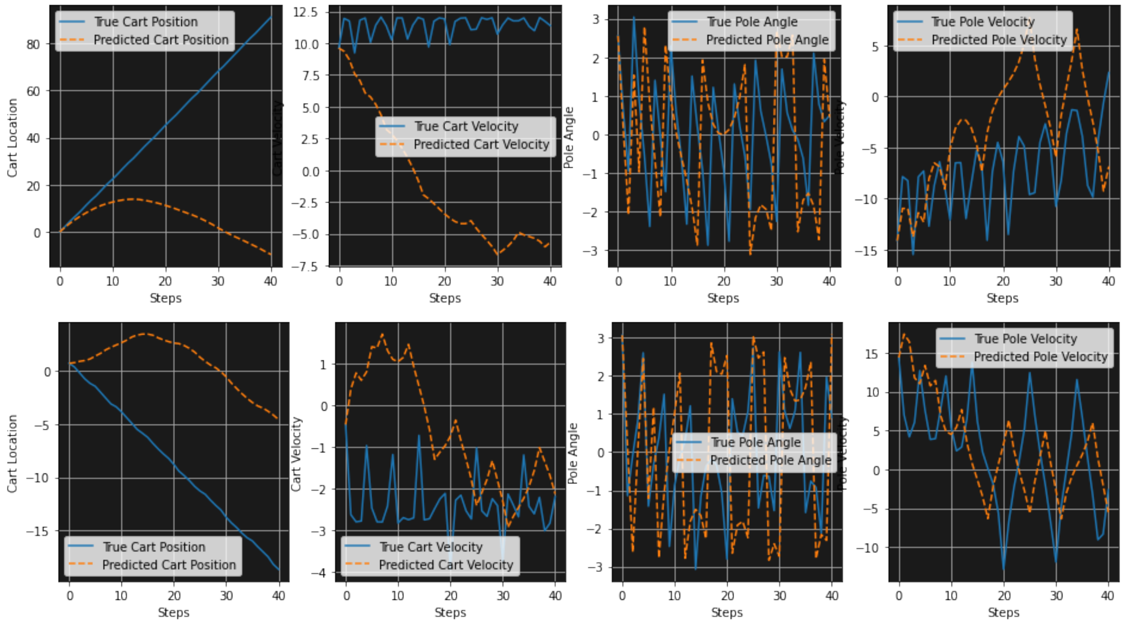


Figure 6: Predicted and true dynamics across 40 time steps, starting from a variety of initial states.

3 Week 2

The second week of the project involved taking action based on the discussions during the first week and conducting non-linear modelling, which was achieved with linear regression with non-linear basis functions. The force/action was then added to the state vector, which allowed for the development of a controller to control the cart-pole system. A linear policy function was developed, which defined what action should be taken, given the state variables, to reach the target state of $X_0 = [0, 0, 0, 0]$.

3.1 Task 2.1

As the linear predictive model's performance was poor, a non-linear model was built, using linear regression with non-linear basis functions. Given a data set of (X, Y) pairs, the model function is given by

$$f(X) = \sum_i \alpha_i K(X, X_i)$$

where α_i are the model coefficients and K is a Gaussian kernel function that defines the non-linear basis. X_i is a state vector within the state space (from the dataset), which is used to place the basis functions at some location in the state space. The kernel function is given by

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

where $X^{(j)}$ refers to the jth component of the state vector e.g. cart position or pole velocity. As pole angle θ is periodic, $(\theta - \theta')^2$ was replaced by $\sin^2((\theta - \theta')/2)$. σ_j are hyperparameters of the kernel function that represent length scales - these parameters were tuned to minimise mean squared error.

The model is given by the following linear system

$$K_{NN}\alpha_N = Y_N$$

with the coefficients of the model given by

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

The parameter λ is a regularisation hyperparameter, which also requires tuning. There are four models, one for each state parameter. N represents the number of training data points, and M represents the number of basis centres.

An initial non-linear model was trained on 1000 data points (with an 80-20 train-test split), with $M = 100$, the sigma values equal to the standard deviation of each state parameters in the training data set, and $\lambda = 1e-4$. Plots of predicted state parameter against true state parameter, for this initial non-linear model, is shown in Figure 7. Comparing this to Figure 4, one can already see a huge improvement in predictive performance.

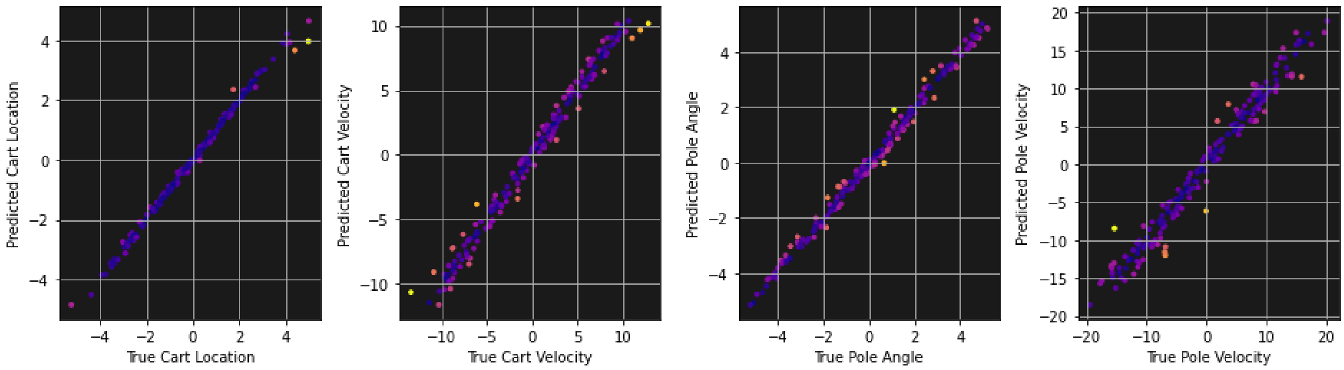


Figure 7: Predicted state parameter against true state parameter for the initial non-linear model.

Optimisation of the non-linear model begun by tuning the length-scale hyperparameters, σ_j, of the kernel functions. This was conducted by scanning over ranges for each σ_j (whilst keeping the others constant, equal to the standard deviation of the corresponding state parameter), and obtaining the mean squared error (MSE) in the predicted change of state for each state variable, with plots shown in Figure 8. It can be seen that each σ_j does not have a significant impact on predictions of cart position and pole angle (which the linear model was somewhat decent at predicting) compared to cart and pole velocity. The sigma values for cart position and cart velocity show a continuous decrease in MSE with increasing σ_j - hence, scans were conducted for extended ranges. The extended scans showed a very small decrease in MSE with increasing σ_0 and σ_1 beyond 20 and 25, respectively. The final selection for [σ_0, σ_1, σ_2, σ_3] was [20, 25, 0.5, 12].

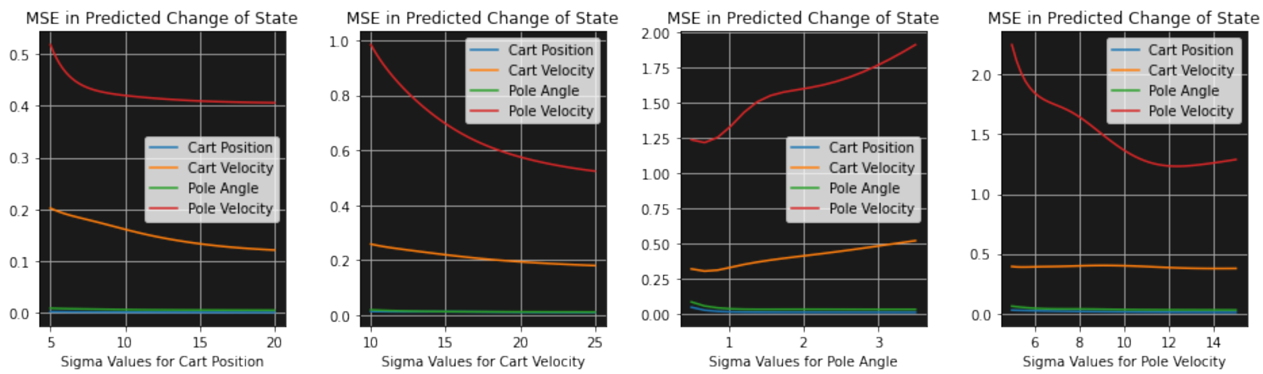


Figure 8: Mean squared error in predicted change of state as functions of scans across σ_j.

The value of λ was swept across the range of 10^{-6} to 10^{-1}, and for each value, the MSE in the model's prediction for the change of state after one call of `performAction()` was calculated and plotted, as shown in Figure 9. The plots show that between 1e-4 and 1e-6, there is practically no change in MSE. As λ is a regularisation parameter, in theory, the smaller its value, the closer the fit to the data, but the more unstable the linear system. Therefore, it was decided to continue with a value of 10^{-5} for λ.

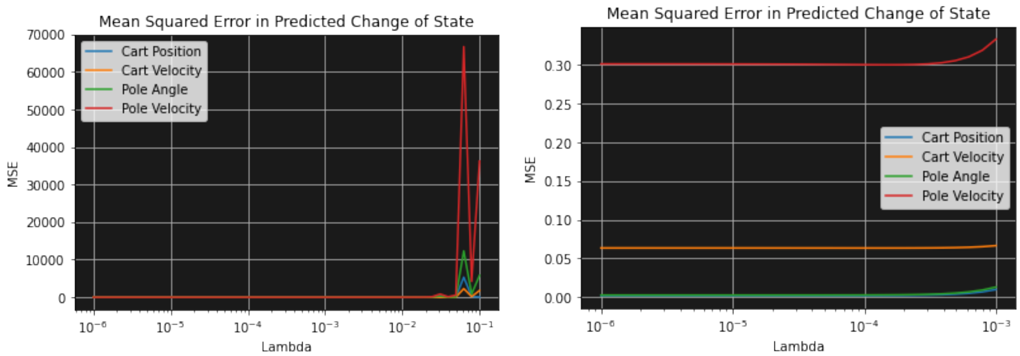


Figure 9: Mean squared error in predicted change of state against the regularisation hyperparameter, λ.

In theory, an increase in the amount of training data increases the performance of the model (at least, to an extent). However, when increasing the amount of training data, it had been found to be beneficial to also increase the number of kernel locations (M), which resulted in longer computation times. Therefore, it was decided to continue with $N = 5000$, with 20% of the data used for testing. For this value of N, a scan across the value of M was conducted - the value of M for the final model was set 500.

Table 1 shows the parameters of the final non-linear model. The mean squared errors in the predicted changes of each state variable for the initial model and the optimised model are given in Table 2 - a massive improvement in predictive performance can be seen. Plots of predicted state parameter against true state parameter, for this optimised non-linear model, are shown in Figure 10. The scatter points in these plots are tighter on the line $y = x$ than in Figure 7, highlighting the boost in predictive performance - there are also more points in this plot as the test set was expanded from 200 data points to 1000.

σ_0	σ_1	σ_2	σ_3	λ	N	M
20	25	0.5	12	1e-5	5000	500

Table 1: Parameters for the final, optimised non-linear model to predict changes in state variables after one step.

Model	**MSE in Cart Position**	**MSE in Cart Velocity**	**MSE in Pole Angle**	**MSE in Pole Velocity**
Initial	0.0316	0.4678	0.0831	2.5120
Final	6.732e-5	0.005559	4.010e-4	0.2035

Table 2: MSE in predicted changes of state variables for the initial and final non-linear models.

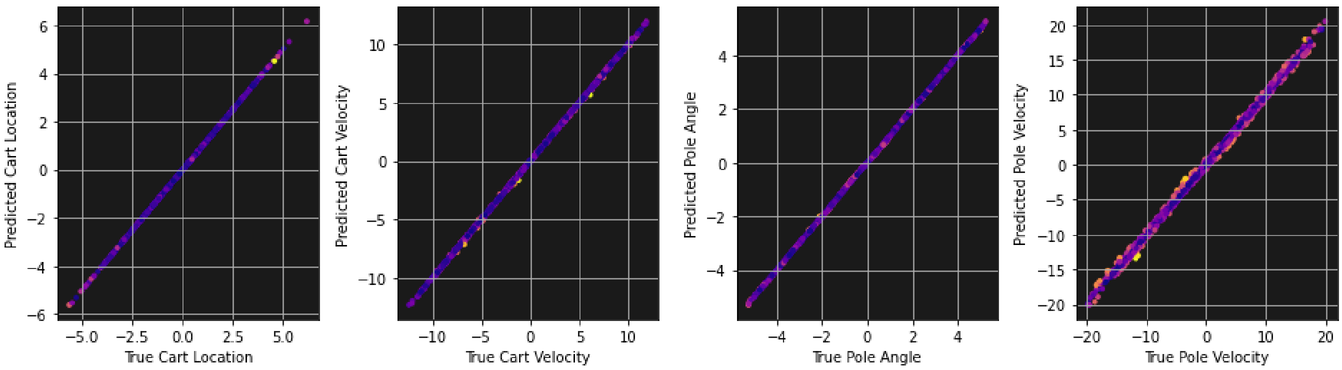


Figure 10: Predicted state parameter against true state parameter for the optimised non-linear model.

As in section 2.3, the 1D scans across the initial setting of each variable was repeated to include the predictions by the optimised non-linear model, shown in Figure 11. Comparing this set of plots to Figure 5 highlights the extent to which the non-linear model is superior to the linear model. This model, as expected, can fit to the non-linearities of the data.

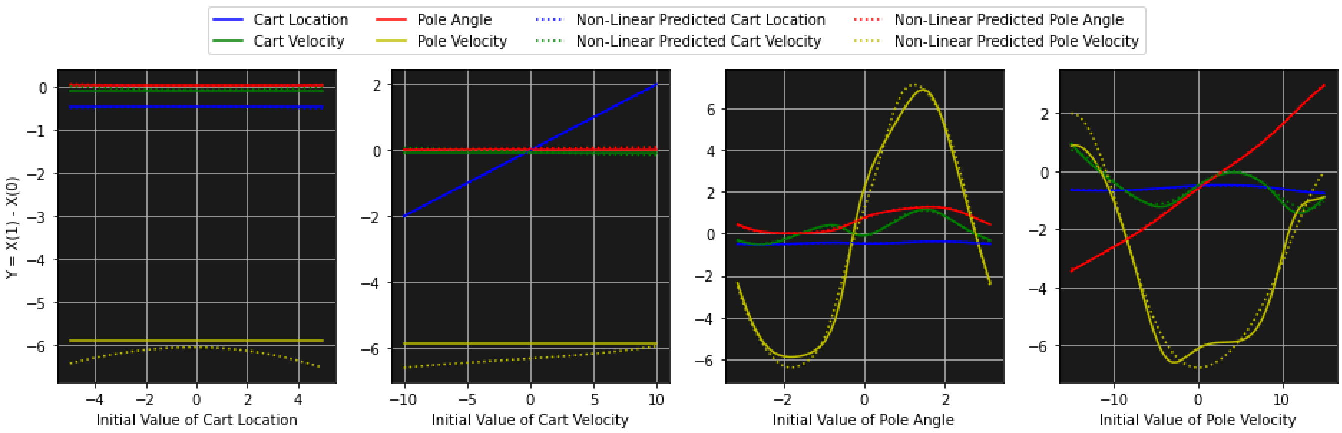


Figure 11: Plots of (non-linear) predicted and true $Y = X(1) - X(0)$ as a function of scans over initial settings of the parameters (X).

The scans across two variables from section 2.2.2 were repeated, but instead the predicted change in each parameter was plotted as contour maps, as can be seen in Figure 12. Comparing these contour plots to Figure 3, it can be seen that the non-linear model is generally fitting to the non-linearities correctly, which can especially be seen in the two bottom rows - the patterns are very similar! However, the top row is also showing non-linear patterns, whilst the true data is (almost) perfectly linear - this may be a case of overfitting. However, upon closer examination, it can be seen that the variation across the colour bars for the top row is very small i.e. the non-linear changes are quite small, and therefore will not have a massive impact on predictions.

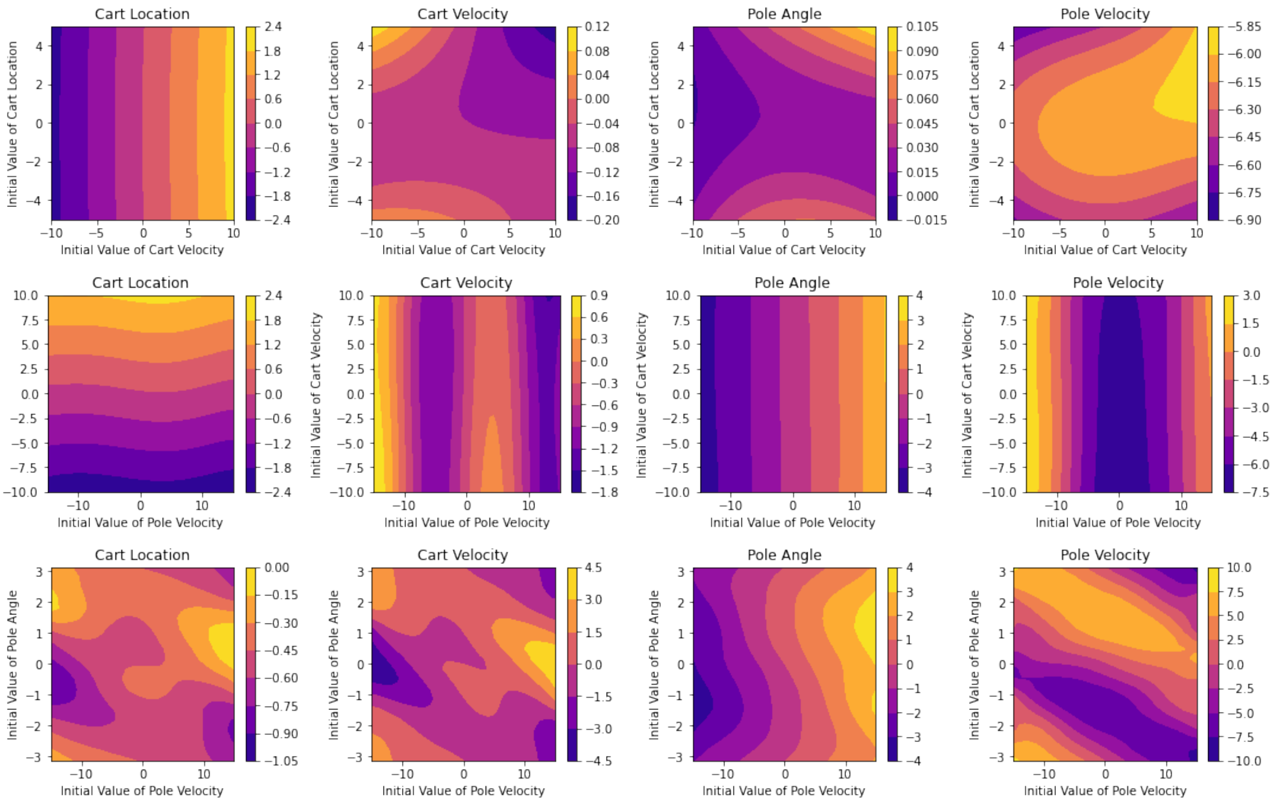
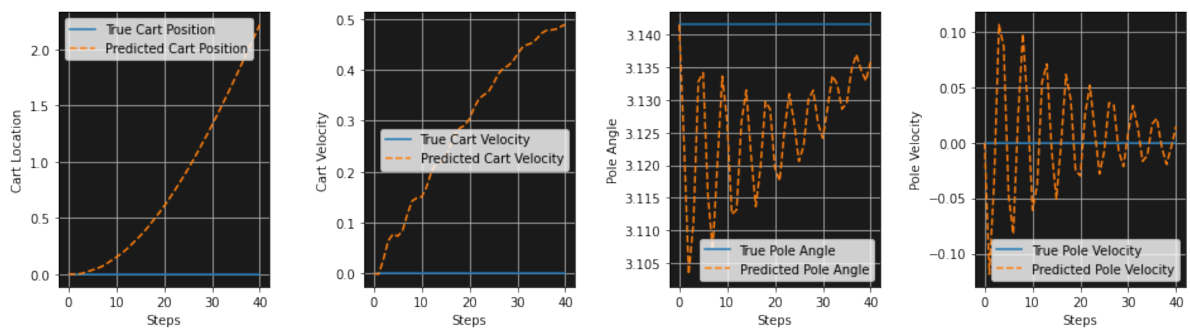


Figure 12: Contour plots for the (non-linear) predicted change in each state parameter as a function of scans across initial values of two of the parameters.

To fully evaluate the quality of the predictions by the non-linear model, it was used to predict rollouts of the cart-pole system, as shown in Figure 13. The first three rows of plots correspond to the same initial conditions in Figure 6: [0, 0, π, 0], [-0.169, 9.607, 2.557, -14.155], [0.738, -0.467, 3.068, 14.384], from top row to bottom row, respectively. The final set of plots shows the predicted time evolution of the dynamics of the inverted pendulum from the initial state [-0.322, 2.593, -0.073, -8.473], and was included due to the model's particularly good predictive performance for this particular initial state. Examining the plots, it can be seen that the predicted rollouts by the non-linear model are greatly superior to that of the linear model. The non-linear model can always predict 10 simulation steps (2 seconds) to a relatively good degree of accuracy, and in some cases can reach almost 40 steps (8 seconds). In the case where the initial state is the stable equilibrium, the model may be predicting a divergence, but the divergences are much less extreme than the predictions by the linear model, which predicted the position to reach -35 in 40 steps, whilst the non-linear model predicted a position of about 2 after 40 calls to `performAction()`. Divergence occurs due to an accumulation of errors; the model can accurately predict the state after one step, but each prediction is not 100% accurate - there is always a small error.



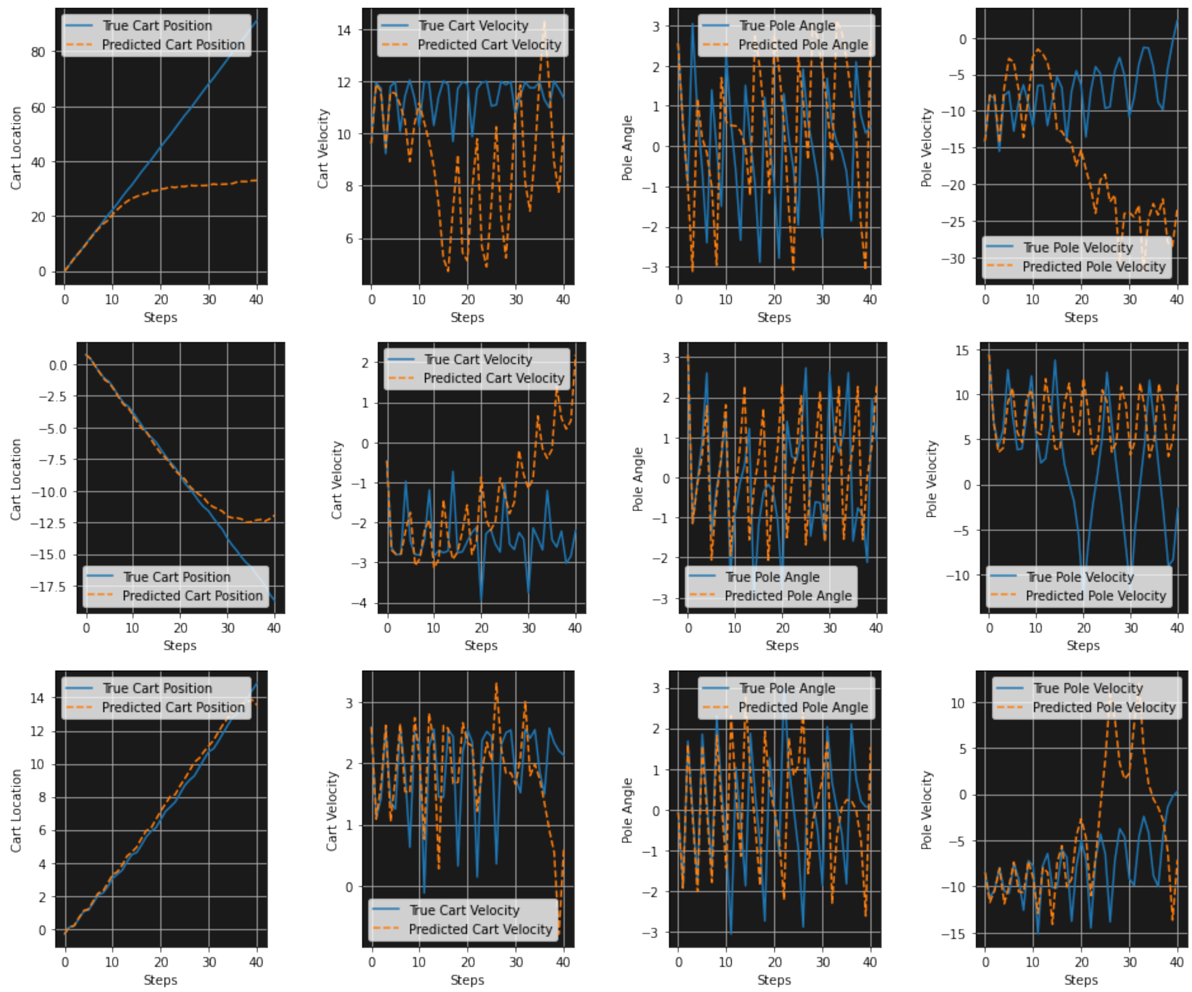


Figure 13: (Non-linear) predicted and true dynamics across 40 time steps, starting from a variety of initial states.

3.2 Task 2.2

This task involved adding the action taken (/force applied to the system) to the input state vector (i.e. X in the training data). This was required for latter stages of the project, where policy functions define actions to be taken to approach the unstable equilibrium [0, 0, 0, 0] - to predict the state after an action was taken, the model can not only be trained on data where the action is zero.

The introduction of the action/force to the input state vector also required the introduction of a fifth length-scale hyperparameter, σ_4. Tuning of this hyperparameter was conducted, and a suitable value was found to be $\sigma_4 = 14$.

Data was generated randomly, with the force being generated in the range of -20 to 20, as 20 was the maximum force set in the `CartPole()` class. As the addition of another input parameter expands the dimensionality of the state space, the dataset size was expanded to $N = 10000$ - ideally a greater number would be used to map across the whole space, but computation times would be massively increased. The train-test split was altered to 0.9-0.1, and M was increased to 1000. Training proceeded with all other hyperparameters set to the same as the optimised model in task 2.1.

The 1D scans across the initial value of each state variable were repeated for this model, but also including a sweep across the initial value of force (with the initial force value being 4.7). The plots can be seen in Figure 14. The model is performing well at predicting the change in state based on the initial force.

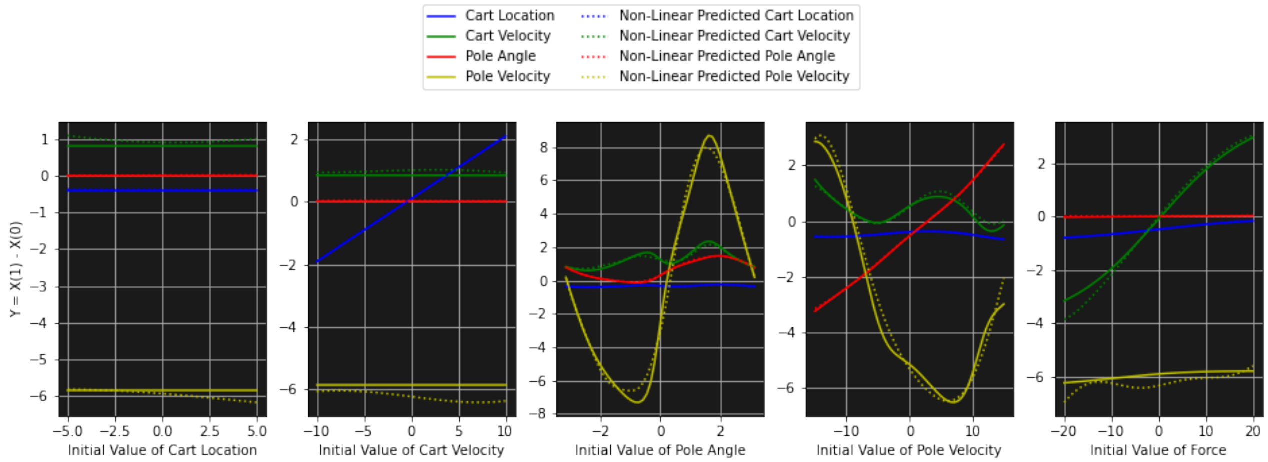


Figure 14: Plots of (non-linear) predicted and true $Y = X(1) - X(0)$ as a function of scans over initial settings of the parameters (X), including force.

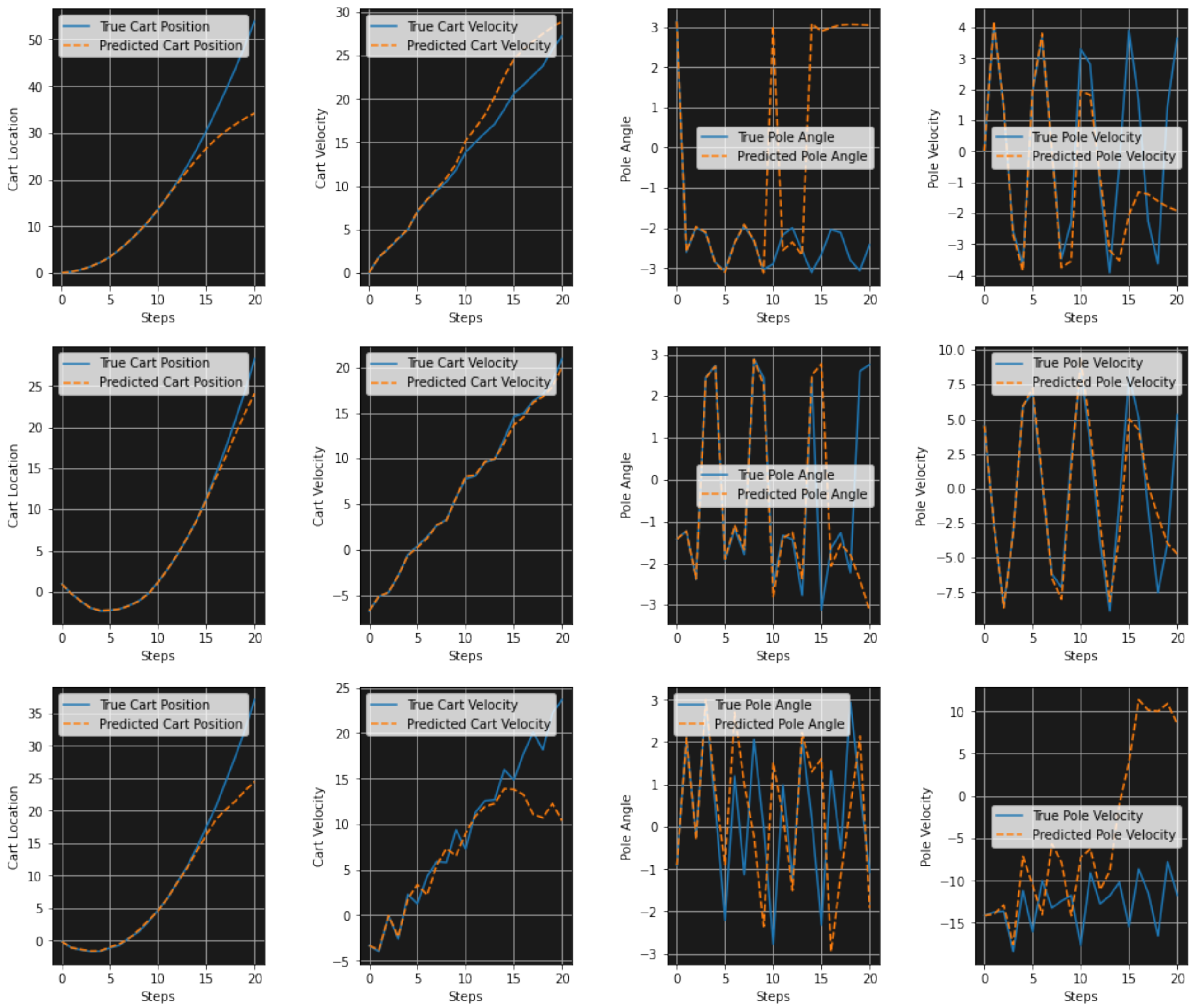


Figure 15: (Non-linear) predicted and true dynamics across 40 time steps, starting from a variety of initial states.

This model was also used to predict rollouts. For these rollouts, the action at each step was set to a constant - in this case, the value 7. The first initial state was set to [0, 0, 0, 0] (with force equal to 7), whilst the other were randomised. The predicted and actual system dynamics for 20 simulation steps can be seen in Figure 15. In some cases, the model can predict up to 20 simulation steps (4 seconds), but it can always predict 10 simulation steps (2 seconds) to a relatively high accuracy. Compared to the model which did not include force in the input state vector (i.e. force was always 0), which could in some cases predict up to 40 steps, the performance of the model including force is reduced. However, to reach the same performance as the previous model, one would require 5000 data points for each value of force - the curse of dimensionality! It was expected that simply doubling the amount of training data would not achieve the same standard of predictive performance. However, a model trained on more data would generally require a larger number of basis centres (to make the most out of the increase in data), which would greatly reduce the speed of the model for predictions (as well as training). For curiosity, a non-linear model (including force) trained on 20000 data points, with 2000 kernel centres, was developed - the predicted rollouts from the same initial conditions as in Figure 15 were produced, as shown in Figure 16. There is a small improvement in predictive performance, but not sufficient considering the great increase in computation demand. It was therefore decided to continue with the previous model, trained on 10000 data points.

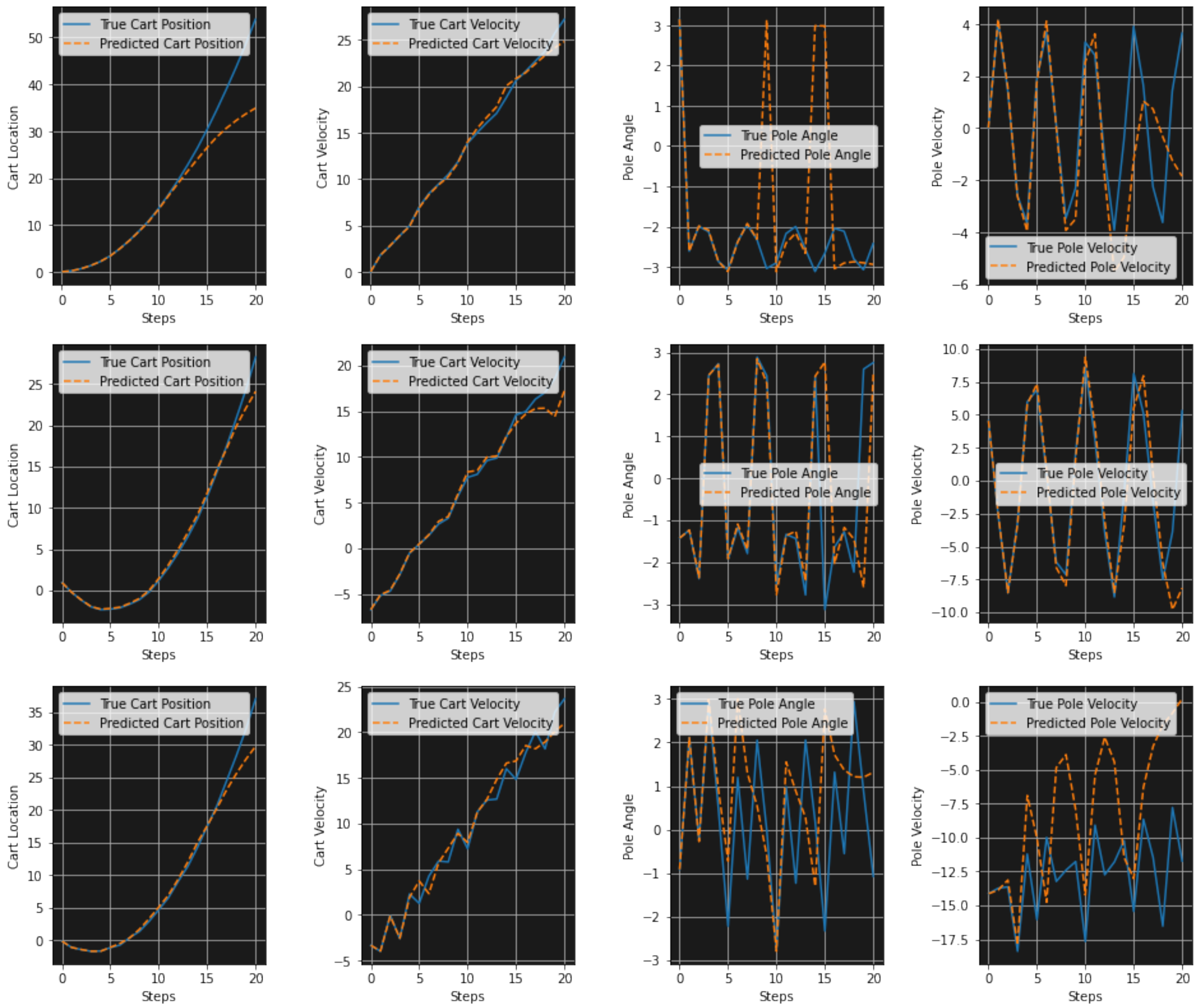


Figure 16: Predicted and true dynamics across 40 time steps, starting from a variety of initial states, for the non-linear model (with force) trained on 20000 data points.

3.3 Task 2.3

The aim of this task was to find a policy function that when enacted would give rise to the desired behaviour of the pole being balanced around its unstable equilibrium [0, 0, 0, 0]. A linear policy, $p(X)$, that defines what action (/force) should be taken given the state variables, is defined as

$$p(X) = \mathbf{p} \cdot \mathbf{X}$$

where $\mathbf{p}$ is a unknown coefficient vector, to be optimised. To optimise a policy, an objective (or loss) function is required. The provided code gives a loss function between a state, and is defined as

$$l(X) = 1 - e^{-|X-X_0|^2/2\sigma_l^2}$$

where σ_l is a scaling factor, and its original value in the provided code was 0.5. On trialling this value of σ_l, it was found that the resolution in the loss function was low, with the loss skewed towards a value of 1 for most states. Therefore, the value of σ_l was set to 10, which reduced the skew in loss. The provided function gives the loss for a singular state.

However, the aim is to develop a policy that can stabilise the system for a number of simulation steps. Therefore, code was produced to calculate the total loss of a trajectory, over a number of steps (in most cases, 20), from a specific state.

Before conducting optimisation of the policy, visualisations of the loss as functions of the parameters in **p** were produced. Each policy parameter was swept across the range -20 to 20, for different initial conditions and initial policies, with two sets of plots provided in Figure 17. The initial state and initial policy parameters for the top set of plots was [0, 0.07, 0, 0.12] and [2, 5, -3, 4], respectively; the two for the bottom set of plots was [0.14, -0.07, 0.11, 0.06] and [1, 2, -1.5, 0.5], respectively. This shows that the loss is dependant on the initial state, which will be a key consideration when optimising the policy.

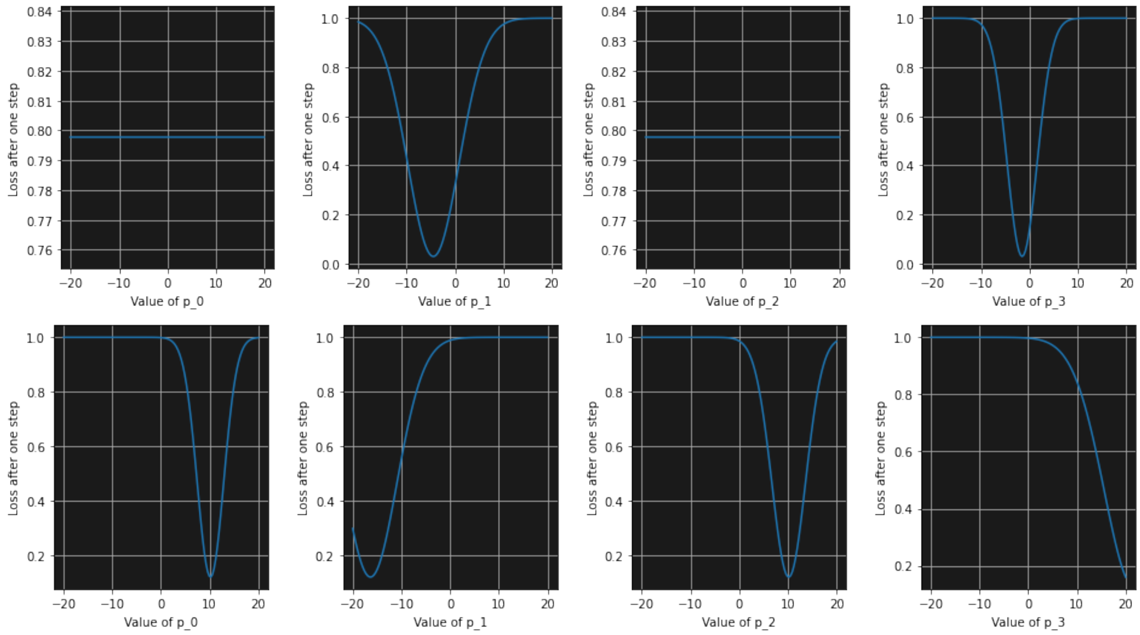


Figure 17: Loss as a function of scans across the four policy parameters.

In addition to the 1D scans above, 2D scans across combinations of policy parameters were produced - with a selection shown in Figure 18.

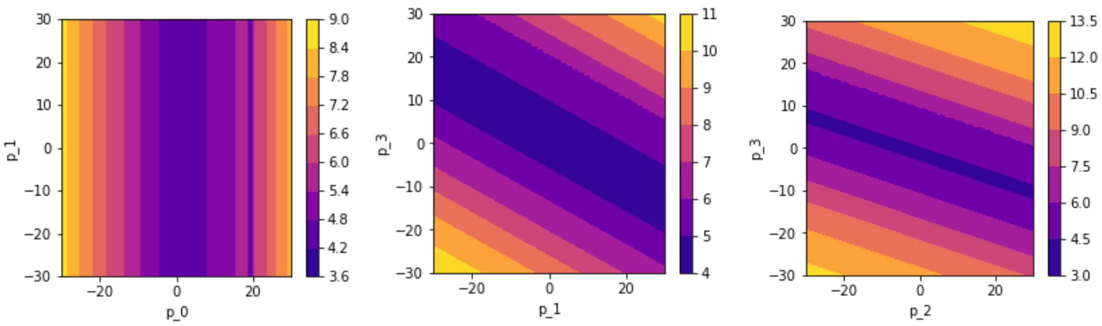


Figure 18: A selection of contour plots for the loss over 20 steps as a function of scans across policy parameters.

The process of optimisation consisted of using the Nelder-Mead method with an off-the-shelf optimiser in `scipy.optimize.minimize()`. Optimisations were conducted for an initial state of [0, 0, 0.2, 0], with a selection of randomised policy parameters - a new policy was obtained for each different setting of the initial policy parameters. The loss for each optimised policy was found, and thus the best policy was returned. The optimal policy was found to be [1.0431, 1.5229, 17.4288, 2.6746] - the plots for the dynamics of the system under the optimised policy, from the initial state of [0, 0, 0.2, 0] can be found in Figure 19. Even though this policy was optimised to minimise loss over 20 steps, it was able to stabilise on the unstable equilibrium for essentially infinite time, as can be seen in the plots, which extend to 50 time steps.

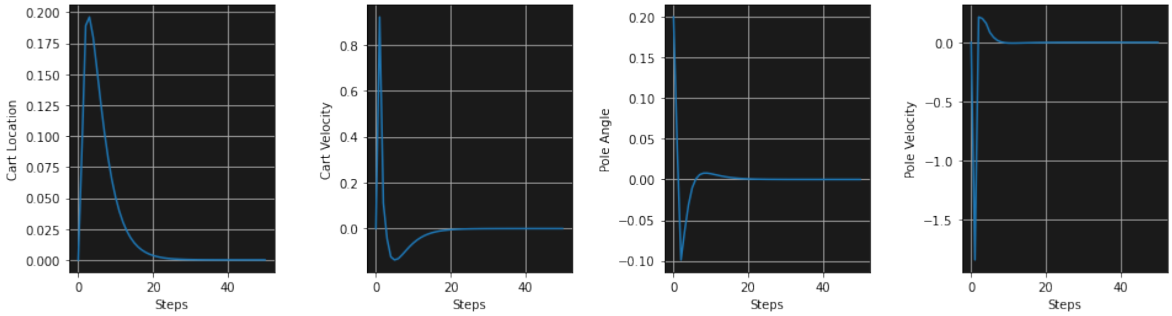


Figure 19: Dynamics of the cart-pole across 50 time steps, under the optimised policy, from the initial state [0, 0, 0.2, 0].

Furthermore, the policy was able to stabilise from angles larger than 0.2, in fact, it was able to stabilise for non-zero values of the other three state variables. The maximum values for the four state variables from which the policy was able to stabilise (whilst the other three were held at zero) were:

- Cart position - 2.56,
- Cart velocity - 1.81,
- Pole angle - 0.58,
- Pole velocity - 4.82.

Plots showing the stabilising capabilities of the optimised policy from initial states of [0, 0, 0.58, 0] and [0, 0, 0.59, 0] (top and bottom, respectively), are presented in Figure 20.

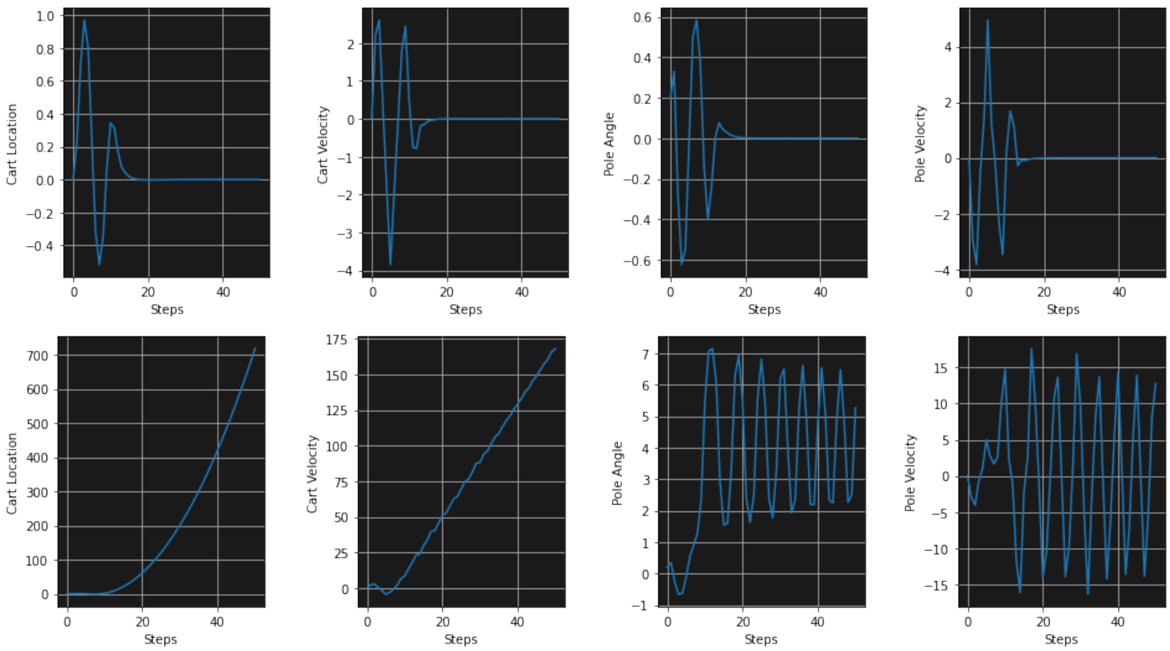


Figure 20: Dynamics of the cart-pole across 50 time steps, under the optimised policy, from the initial states [0, 0, 0.58, 0] (top row) and [0, 0, 0.59, 0] (bottom row).

In an attempt to further increase the maximum ranges in which the policy could stabilise the cart-pole system, further optimisation was conducted. The `scipy.optimize.minimize()` function was called, with the initial state being [3, 0, 0, 0], and the initial policy being the prior optimal policy. This returned the following policy parameters [1.1583, 1.1683, 20.0619, 2.8489]. This indeed increased the stabilising ranges of the policy, but reduced the stabilising intensity - rather than the state approaching [0, 0, 0, 0] and staying there, in most cases, the system would oscillate about the unstable equilibrium. Plots of the dynamics of the inverted pendulum under the new policy, from three different initial states, are shown in Figure 21. The top row of plots shows the states of the system under the new policy, from the initial state [0, 0, 0.58, 0] - the same initial state as in the top row of plots in Figure 20. This shows the oscillatory behaviour of the new policy. The second row of plots in Figure 21 shows the dynamics starting from the state [0, 0, 0.72, 0] - this policy has larger ranges than the previous, which would have diverged from this initial state. The last row of plots shows the time evolution of the system's state from the initial state [3, 0, 0, 0] - this is the state that the policy was optimised for, hence the perfect stabilisation. For positions of 2.9 and 3.1, the policy resulted in oscillatory behaviour.

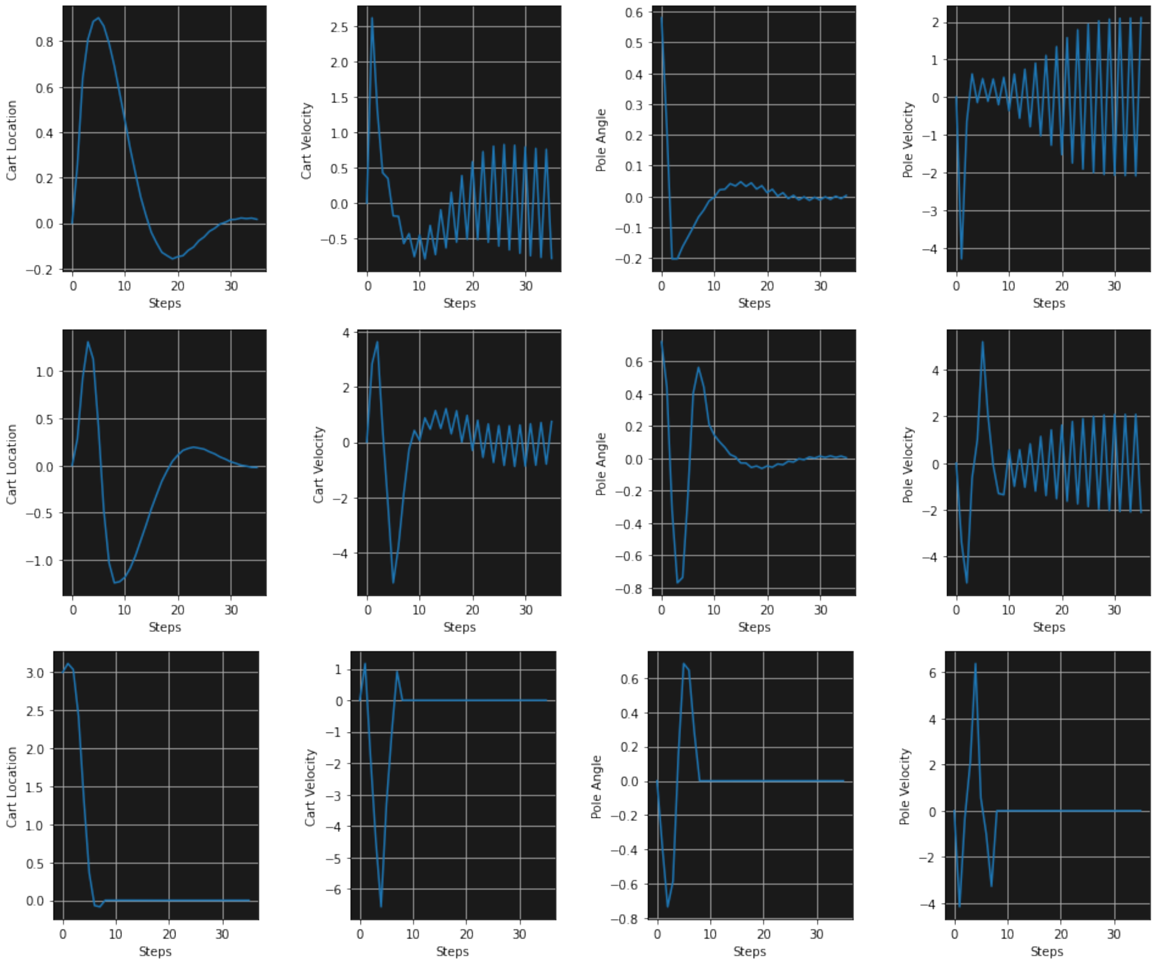


Figure 21: Dynamics of the cart-pole across 50 time steps, under the newly optimised policy, from the initial states [0, 0, 0.58, 0] (top row), [0, 0, 0.72, 0] (middle row), and [3, 0, 0, 0] (top row).

3.4 Task 2.4

The aim of this task was to implement model predictive control; applying a policy to predicted states of a model. Figure 22 shows the first optimal policy (i.e. the one that is more stabilising, but with worse ranges) applied to the predicted states by the non-linear model from task 2.2. The policy was applied to the predicted state at each step, and the true state due to this action was also found. Figure 23 shows a slightly different picture - here the policy was applied to both the predicted and true state at each step. Studying both figures, it can be deduced that both the non-linear predictive model and the policies were working well together; model predictive control was successful.

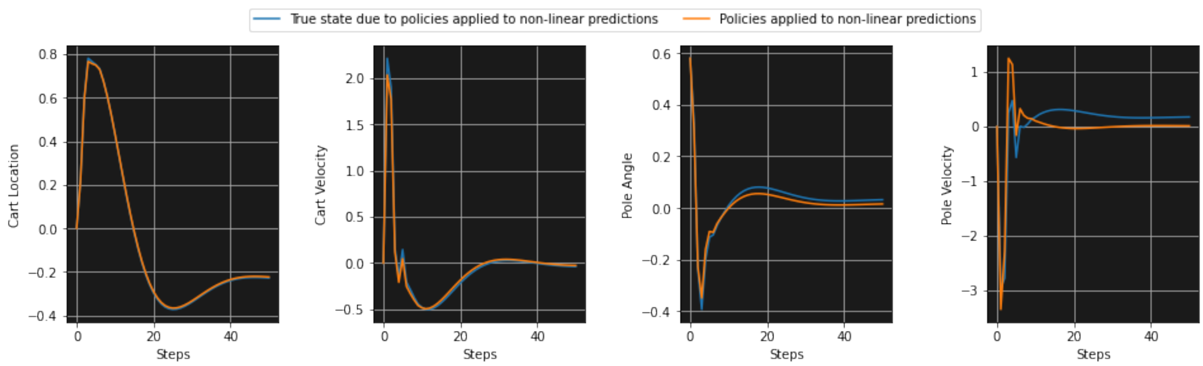


Figure 22: Model predictive control from initial state [0, 0, 0.58, 0] - policy only applied to predicted states.

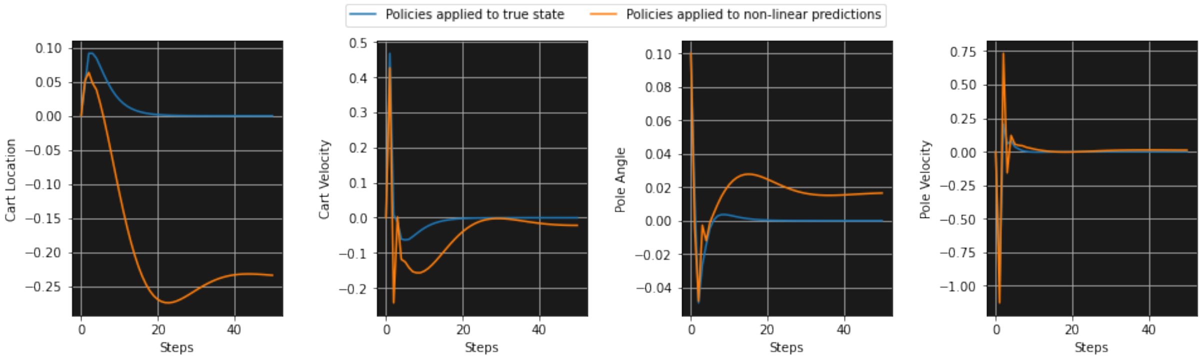


Figure 23: Model predictive control from initial state [0, 0, 0.58, 0] - policy applied to both predicted states and true states.

4 Week 3

The penultimate week of the project involved repeating previous tasks, but introducing noise in various forms, and observing its impact.

4.1 Task 3.1

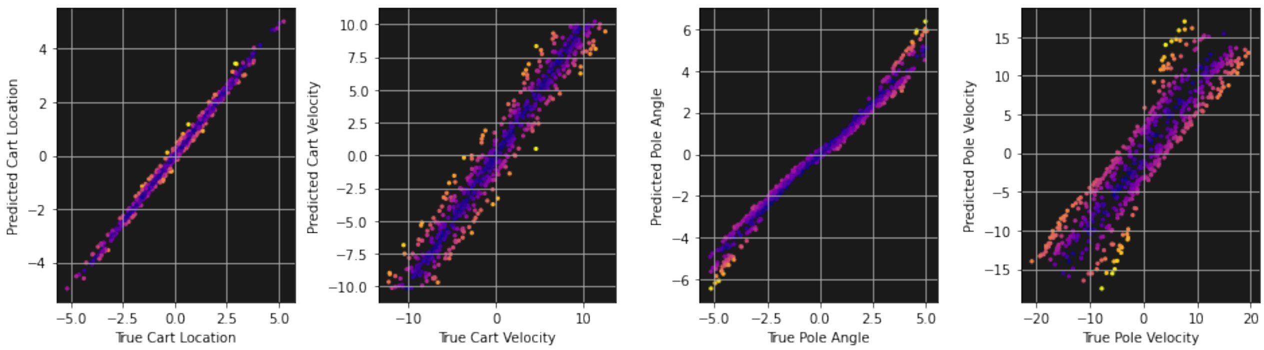
In task 3.1, noise was introduced to the observed dynamics (not the actual dynamics). This was achieved by adding zero-mean Gaussian noise to each state variable after each call to `performAction()` - Listing 1 shows a code snippet for adding noise to the observed states. The standard deviation of the noise for each state variable was set to the standard deviation of the distributions for the generated data, and the `noise_frac` variable allowed for tuning of the extent of the noise. In reality, the noise would not depend on the actual data; the noise would be a property of the sensors. However, the addition of `noise_frac` allowed for the mimicry of sensor noise.

```
current_state += np.random.normal(0, [1.5, np.sqrt(1/12)*20, np.sqrt(1/12)*2*np.pi, np.sqrt(1/12)*30]) * noise_frac
```

Listing 1: Code snippet for adding noise to the observed states

4.1.1 Linear Model

Data was generated with the addition of noise to the observed states i.e. noise was added to Y. Figure 24 shows how noise impacted the predictions of the linear model, with noise_frac for the top row being 0.05, and 0.2 for the bottom row. Without noise, the linear model was a poor predictor of the change in state after one step - the addition of noise worsened performance further.



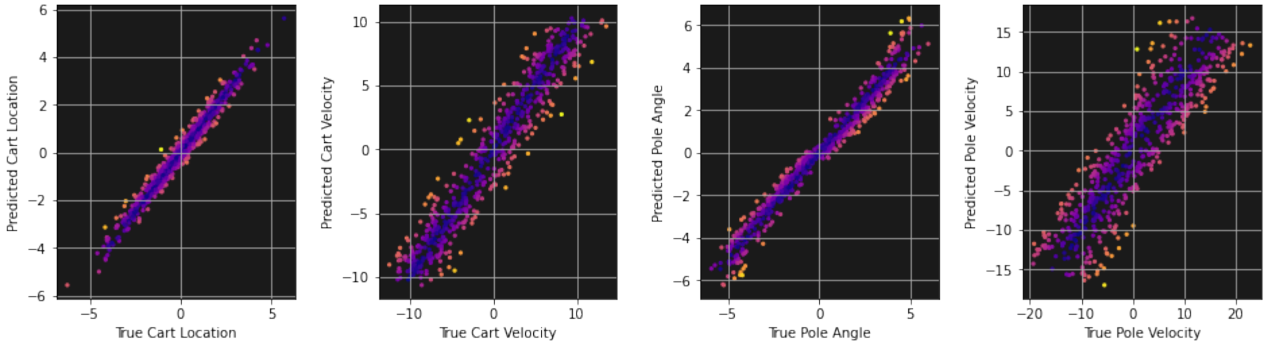


Figure 24: Predicted vs true state parameters for linear models trained on noisy data: noise_frac = 0.05 (top) and noise_frac = 0.2 (bottom).

4.1.2 Non-Linear Model

Noise was applied to the training data for both the non-linear model including and excluding the force as an input state variable. The impact of noise on both was the same, and therefore, only results for the model including force will be presented.

Figure 25 shows predicted state variables against true variables for noise multipliers of 0.05 and 0.2, in the top and bottom rows of plots, respectively. This shows how noise impacts the predictive performance of the non-linear model. The impact of noise on the non-linear model is more severe than the impact on the linear model. The reduction in performance from `noise_frac` values of 0.05 to 0.2 is more severe for the non-linear model than the linear model - the linear model is slightly less tight about $y = x$ (given that it was already quite poor), whilst the non-linear model is dramatically less tightly fitting. For a noise multiplier of 0.2, the performance of the non-linear model is quite poor, which is further backed up by Figure 26, which shows the predicted and true changes in state as functions of scans over initial values of the state parameters. The bottom set of plots in this figure shows a quite severe degradation in performance - the model is not fitting to the true trends in the data as well.

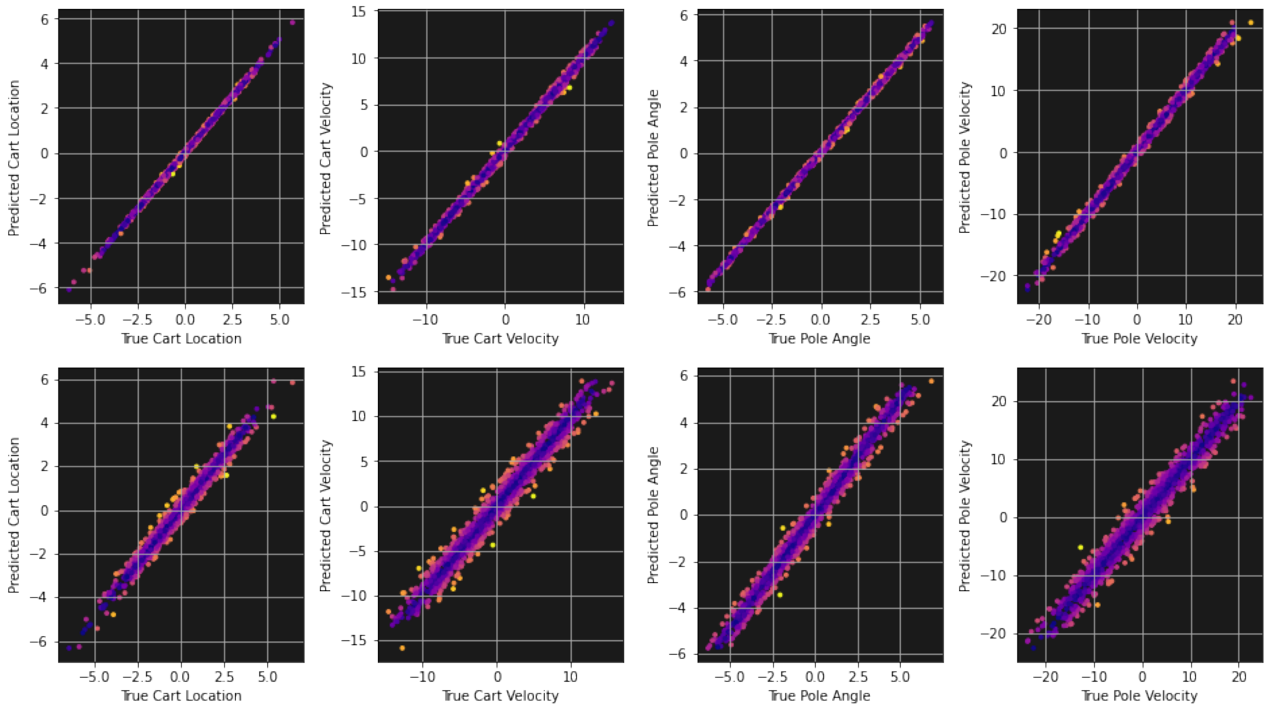


Figure 25: Predicted vs true state parameters for non-linear models trained on noisy data: noise_frac = 0.05 (top) and noise_frac = 0.2 (bottom).

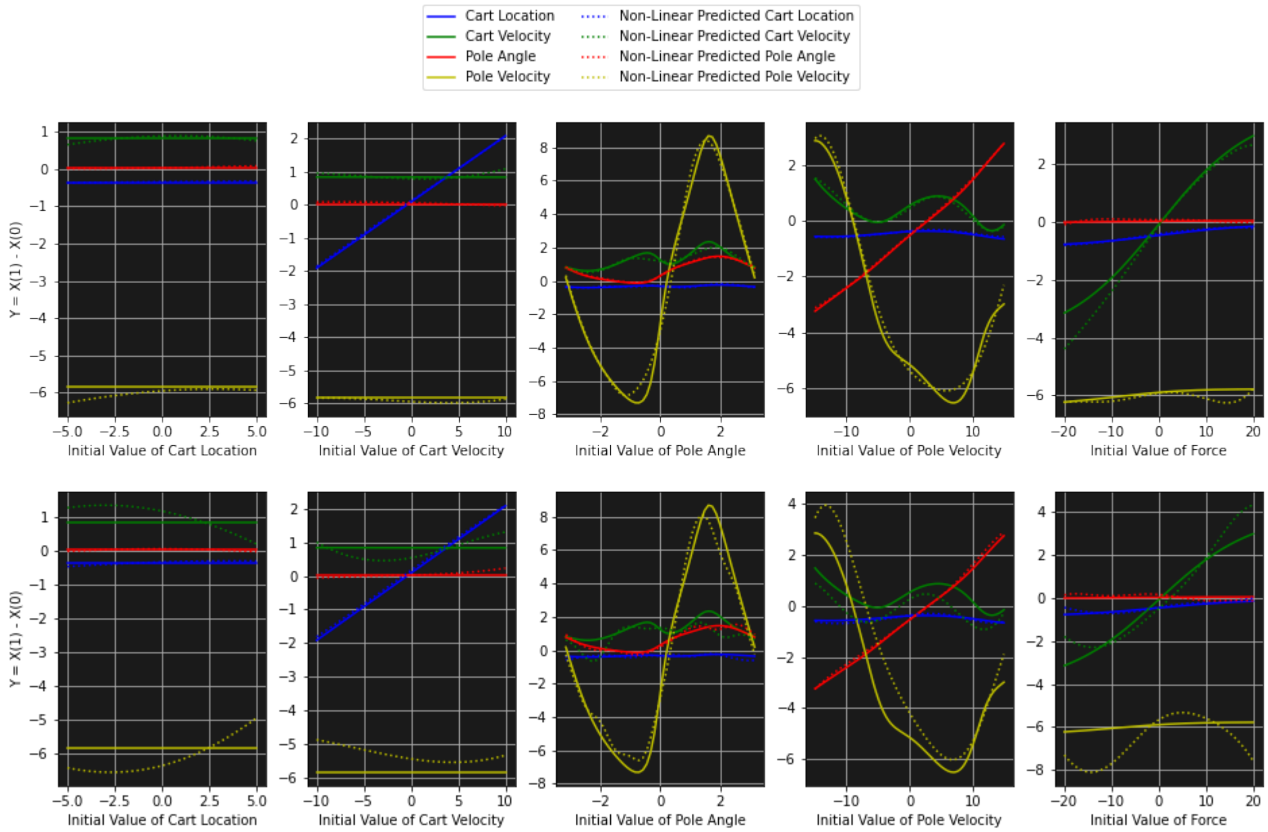


Figure 26: Plots of (non-linear) predicted and true $Y = X(1) - X(0)$ as a function of scans over initial settings of the parameters (X), for noise fractions of 0.05 (top) and 0.2 (bottom).

To fully understand the impact of noise on the non-linear model, the two models trained on data with noise fractions of 0.05 and 0.5 were used to predict rollouts. Figure 27 shows the predicted time evolutions of the system's state from the initial state [0, 0, 0, 0], with a constant force of value 7 at each step, for models trained on noisy data. The bottom row of plots shows the predictions from the model trained on data with a noise fraction of 0.2 - this further emphasises the impact of noise on the non-linear model.

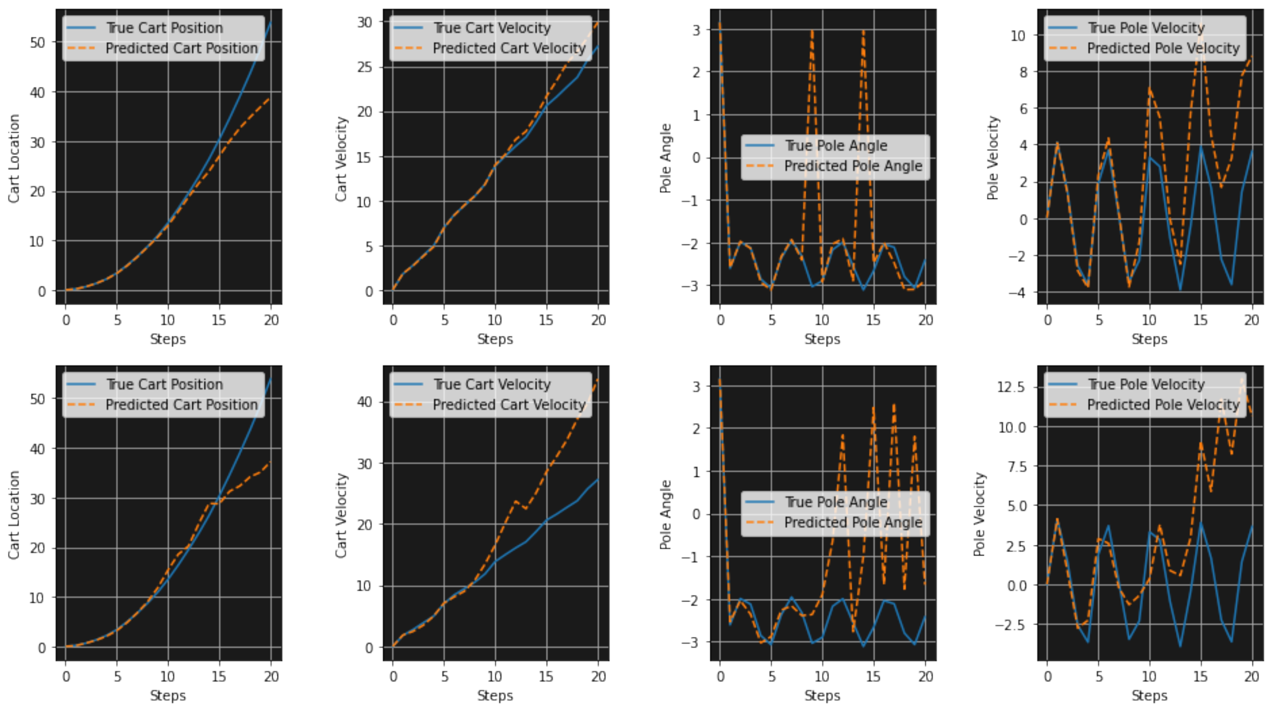


Figure 27: Predicted dynamics of the cart-pole system from the initial state [0, 0, 0, 0], with a constant force of value 7 at each time step, for non-linear models trained on data with noise multipliers of 0.05 (top) and 0.2 (bottom).

4.1.3 Linear Policy

The optimised policies were applied to noisy data to analyse whether or not the policies could still stabilise the system, and if so, to what degree. The first optimal policy (which had the smaller ranges, but better stabilisations) was applied to noisy observed states. The true and noisy states were plotted over time, as shown in Figure 28. Here, the noise fraction was set to 0.05 and the initial state was [0, 0, 0.58, 0] - the maximum angle which the policy could stabilise without noise. With this amount of noise, the policy sometimes stabilised (although oscillatory), as can be seen in the top row of plots, but sometimes diverged, as shown in the bottom set of plots. The policy applied to noisy data could sometimes stabilise (although not perfectly) angles greater than 0.58 - in some cases, the noise was favourable. However, in general, noise was a detriment to the performance of the policy - the policy could no longer bring the model to stay perfectly at [0, 0, 0, 0], as expected.

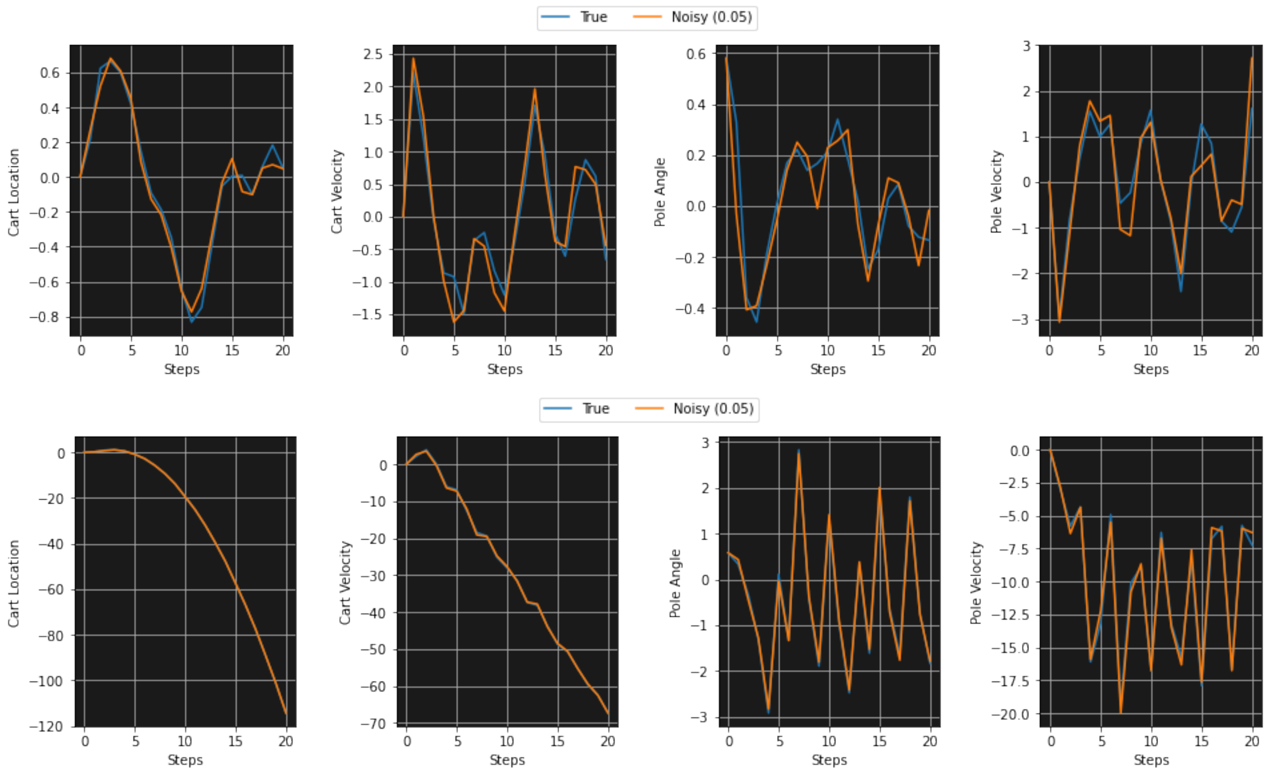


Figure 28: Policy applied to noisy states (noise_frac = 0.05), from an initial state of [0, 0, 0.58, 0].

The policy was applied to a noise multiplier of 0.2 and was never able to stabilise, even when starting from [0, 0, 0, 0], as shown in Figure 29. The critical value of the noise multiplier, above which the policy was never able to stabilise the system from [0, 0, 0, 0], was found to be 0.12.

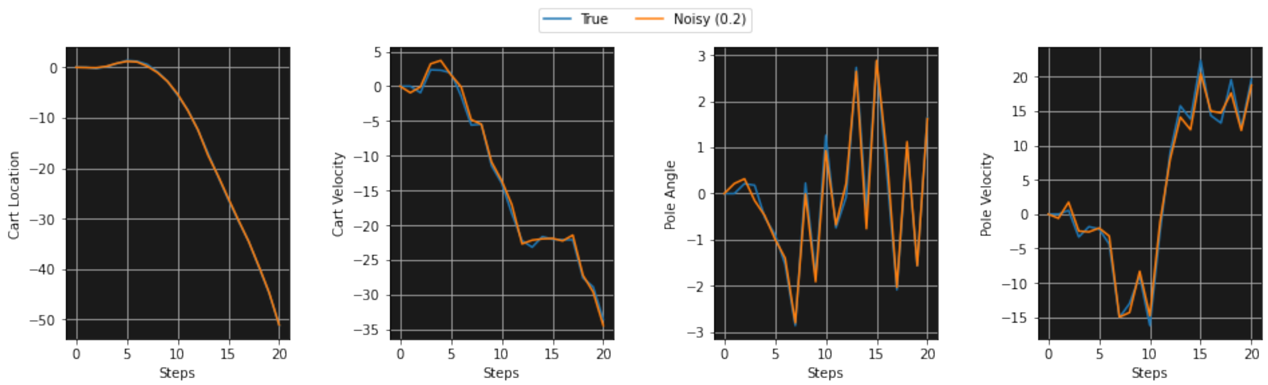


Figure 29: Policy applied to noisy states (noise_frac = 0.2), from an initial state of [0, 0, 0, 0].

4.2 Task 3.2

Rather than adding noise to the observed states, task 3.2 involved adding noise to the actual dynamics of the cartpole system. This was achieved by adding noise to the system's cart velocity and pole velocity, before being input to the equations of motion in each simulation step of `performAction()`, as shown in Listing 2. Again, a variable was included to alter the amount of noise added to the system. The noise was added to the velocities of the cart and pole, which fed into the accelerations of both, which then fed back into the velocities. Therefore, the noise propagated through the system, and a noise multiplier of 0.05 had a much larger impact than in task 3.1, where the noise was only added at the end.

```
self.cart_velocity += np.random.normal(0, np.sqrt(1/12)*20) * self.noise_frac
self.pole_velocity += np.random.normal(0, np.sqrt(1/12)*30) * self.noise_frac
```

Listing 2: Code snippet for adding noise to the actual dynamics of the system.

4.2.1 Linear Model

Data was generated with the addition of noise to the dynamics of the system and used to train a linear regressor to predict the change in state after one call to `performAction()`. The results and plots for this were almost identical to that of section 4.1.1, except for the fact that lower noise multipliers were required to achieve the same impact on performance, due to the propagation of noise, as explained previously. Figure 30 shows predicted state parameters for a linear model trained on noisy data with the noise multiplier equal to 0.2 against true variables - the scatter points are very sparsely dispersed. Comparing this to Figure 24 shows that noise multipliers of equal magnitude have a much greater impact when applied to the actual dynamics as opposed to the observed states, as expected. Therefore, to achieve the same impact, a lower multiplier must be used.

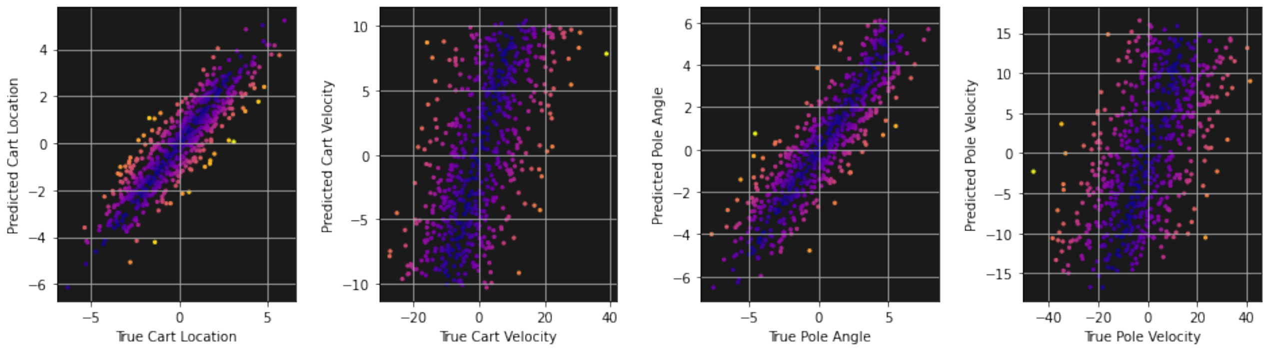
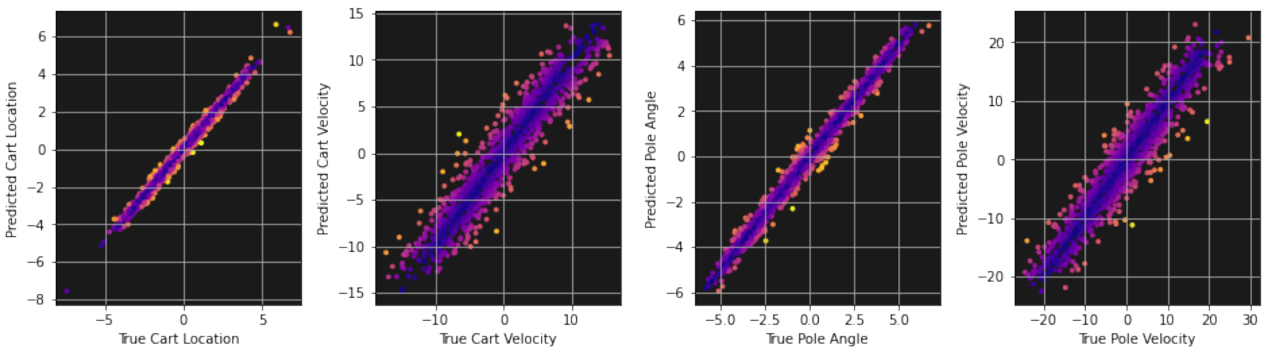


Figure 30: Predicted state parameter against true state parameter for a linear model trained on noisy data, where noise is added to the dynamics of the system and noise_frac = 0.2.

4.2.2 Non-Linear Model

The same plots as in section 4.1.2 were repeated here, except that different values for the noise multipliers were used, due to the propagation of noise issue. The top set of plots in each of Figures 31 to 33 shows predictions for a non-linear model trained on noise generated in the dynamics of the system with a noise multiplier of 0.05, whilst the bottom row in each figure uses a noise factor of 0.02. It can be seen that even with a noise factor of 0.02, the model is still being greatly impacted, with reduced predictive performance.



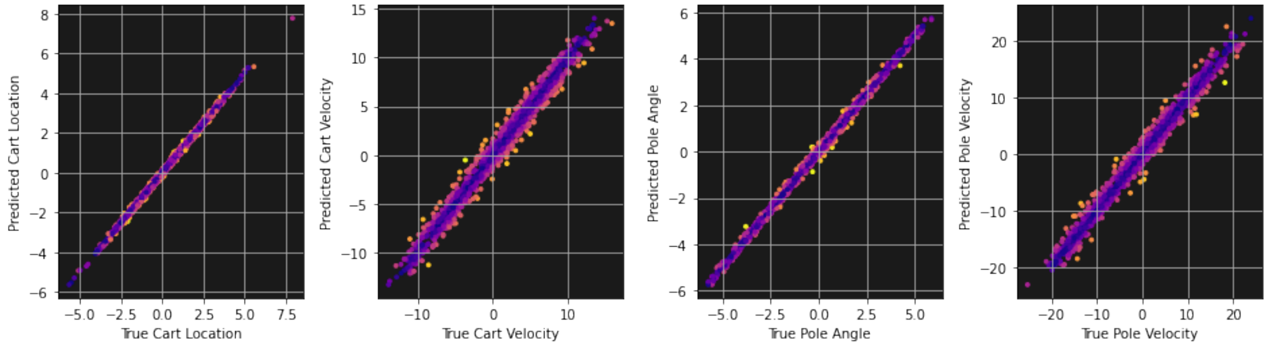


Figure 31: Predicted vs true state parameters for non-linear models trained on noisy data: noise_frac = 0.05 (top) and noise_frac = 0.02 (bottom).

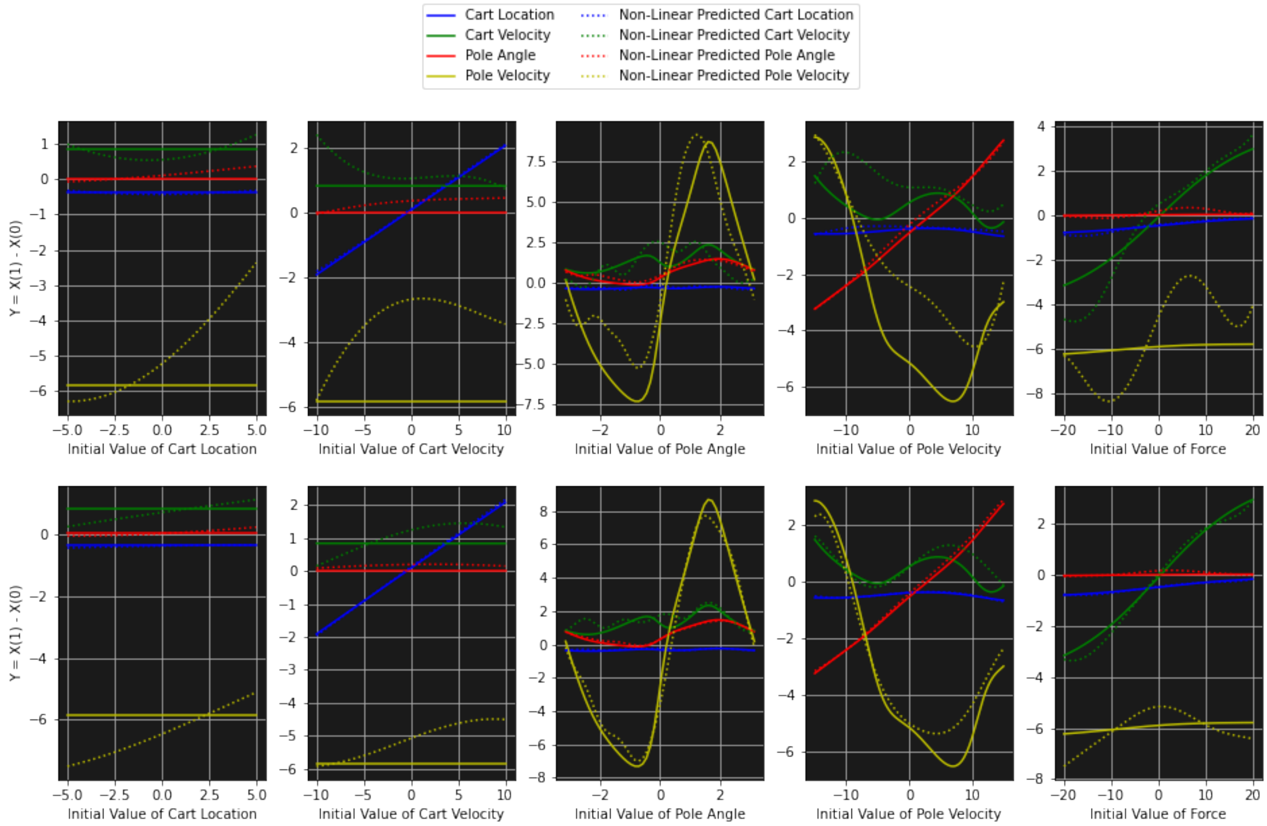
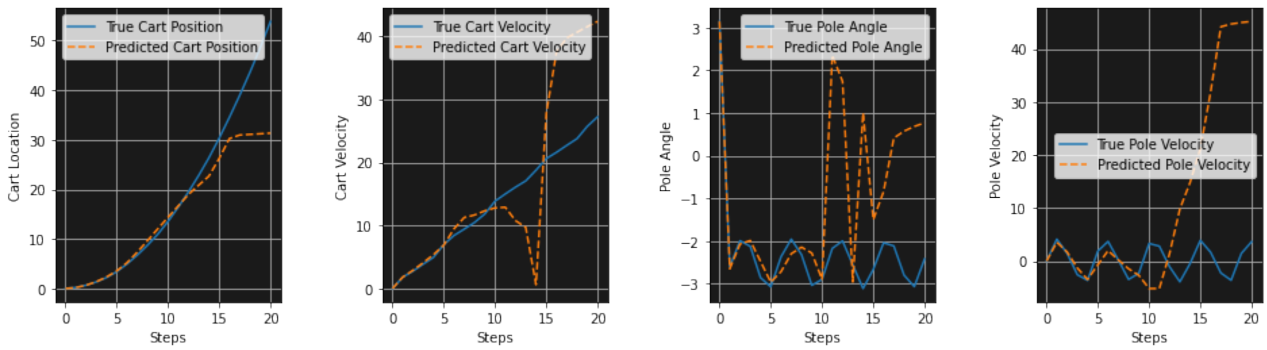


Figure 32: Plots of (non-linear) predicted and true $Y = X(1) - X(0)$ as a function of scans over initial settings of the parameters (X), for noise fractions of 0.05 (top) and 0.2 (bottom).



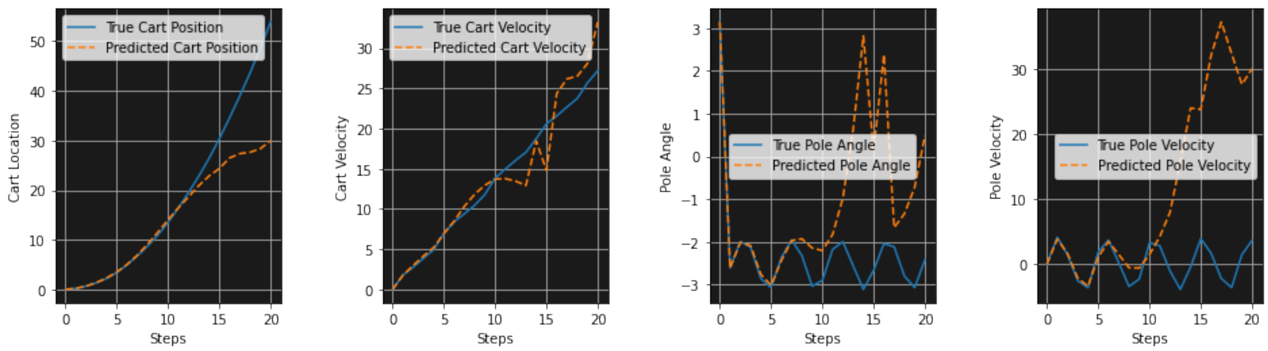


Figure 33: Predicted dynamics of the cart-pole system from the initial state [0, 0, 0, 0], with a constant force of value 7 at each time step, for non-linear models trained on data with noise multipliers of 0.05 (top) and 0.02 (bottom).

4.2.3 Linear Policy

For the same linear policy used in section 4.1.3 (i.e. policy parameter values of [1.0431, 1.5229, 17.4288, 2.674]), the critical noise factor, above which the policy was unable to stabilise the system from the unstable equilibrium itself, was found to be 0.015, when the noise was added to the dynamics of the system, rather than simply to the observed state. For a noise multiplier of 0.015, the vast majority of the time, the policy resulted in divergence. However, on some rare occasions, the policy was able to stabilise the system under this amount of noise, with one of these occasions shown in Figure 34. Above 0.015, the policy was never able to stabilise the inverted pendulum on a cart.

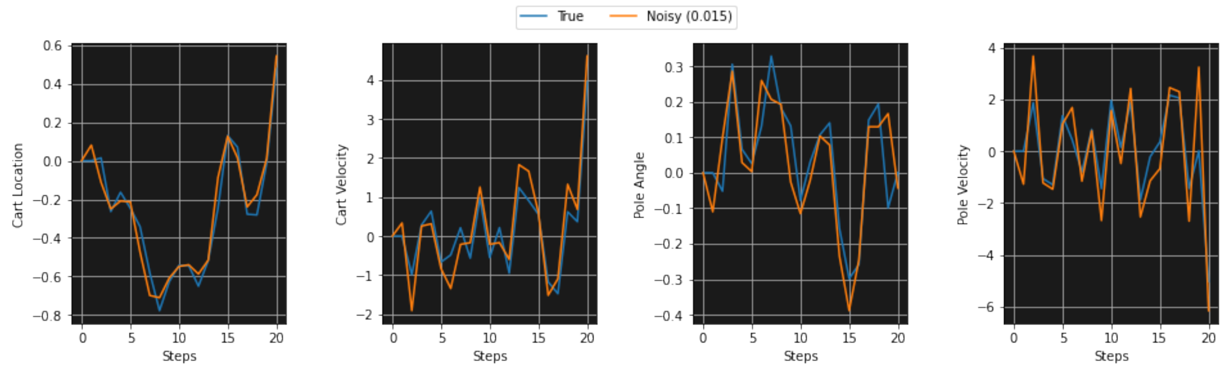


Figure 34: Policy applied to noisy states (noise_frac = 0.015), where noise is added into the equations of motion, from an initial state of [0, 0, 0, 0].

5 Week 4

5.1 Task 4.1

The aim of this task and the fourth and final week was to obtain a non-linear policy to stabilise the system at its unstable equilibrium, with the pole facing vertically upwards, when starting with the pole facing vertically downwards (i.e. the stable equilibrium), in the noise-free case. The non-linear policy function was defined as

$$p(X) = \sum_i w_i e^{-0.5(X-X_i)^T W (X-X_i)}$$

where the weights w_i, the basis function centres X_i, and the elements of the 4x4 symmetric matrix W are free parameters to be optimised. It was decided to use 20 basis functions, and therefore, the total number of parameters that required optimising was 116, compared to four parameters for the linear policy. As a result, each optimisation took about 4.5 minutes on a PC, with an AMD Ryzen 5 1600 CPU, and about 6.5 minutes on a HP laptop, with an AMD Ryzen 4 4500U CPU (technically, APU).

This task was found to be very difficult. A lot of experimentation was conducted to obtain a policy which could swing the pole up, and then keep it balanced. The linear policies were able to stabilise the system from angles of 0.5 to 0.7. Although the non-linear policies should be superior over the linear policies, it was still a challenge to achieve a policy that could stabilise the system from an angle of π (= 3.14).

Initial optimisation was conducted by randomly setting the initial policies: the weights and coefficients of W were set as random samples from a uniform distribution over $[0, 1)$ with `np.random.rand()`, and the locations X_i in the same ranges as generated data for the predictive models i.e. $X_{i,0}$ sampled from a zero-mean Gaussian, with variance 1.5, and $X_{i,1}$, $X_{i,2}$, and $X_{i,3}$ sampled from uniform distributions over the ranges $[-10, 10)$, $[-\pi, \pi)$, and $[-15, 15)$, respectively. The same loss function as in task 2.3, with $\sigma_l = 10$, was used. The matrix W was made symmetric by taking its transpose and multiplying by itself (i.e. $W^T W$). All of the policy parameters were added to a one-dimensional array, which was then optimised using the Nelder-Mead method, making use of the `scipy.optimize.minimize()` function. More than 50 random initial policies were optimised, for the initial state [0, 0, π, 0]. This returned some policies achieving losses of less than one over 20 steps - these policies kept the cart position, cart velocity and pole velocity at almost zero for the entire 20 steps (4 seconds), and kept the pole angle at π. This behaviour was not desired, and an example of this behaviour can be seen in Figure 35 - take notice of the axes labels for cart postion, cart velocity, and pole angular velocity. Further investigation led to the discovery that the value for σ_l in the loss function was too high at 10; this value of σ_l did not penalise angle sufficiently. When the value of σ_l was reduced to 2, the loss over 20 states for the same policies were found to be around 15. Other policies (which achieved higher losses) when σ_l was set to 10 were also studied - some got somewhat close to the desired angle (although the velocities were quite high), but were unable to completely stabilise. Further optimisations continued with σ_l set to a value of 2.

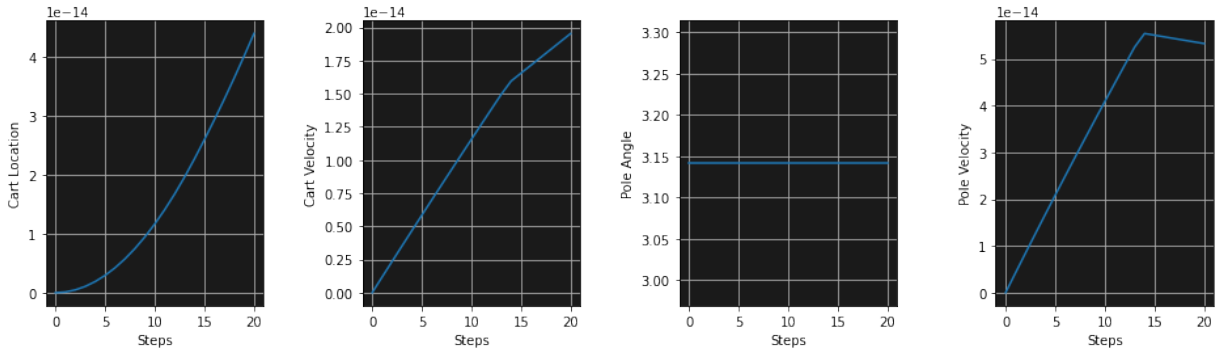


Figure 35: Policy which achieved lowest loss when σ_l was set to 10.

Another issue encountered during optimisations was overflow occurring, especially during the matrix multiplications in the policy function. This lead to the policy function returning 'NaN' ('Not a Number') values, which led to NaN values in the state parameters and in the loss. Some initial settings of the policy parameters lead to overflow in the first (few) iterations. Therefore, when a policy function returned NaN during the first 4 simulation steps, the policy was discarded. If a policy returned NaN in the latter iterations, the action was set to a value of zero, and evaluation of these policies was still conducted.

Further experimentation was conducted by:

- Varying initial policy parameters.
- Including periodicity in the non-linear policy function by replacing $(X - X_i)$ with $\sin(X - X_i)$ for the pole angle (similar to the non-linear model for predicting the next state).
- Varying σ_l.
- Constructing the 4x4 symmetric matrix W in a different method - reflecting the upper triangular, thus only requiring 10 values to construct a 4x4 (16 values) symmetric matrix, with the function shown in Listing 3.
- Testing different initial states. The aim of the task was to stabilise the system from [0, 0, π, 0]. However, special initial states with non-zero (but small) cart positions, cart velocities and pole velocities were used e.g. [0.5, 0, π, 0].
- Varying the number of basis functions between 5 to 20 (although one would expect that optimal performance would be achieved with the greatest number of basis functions).

```python
def make_44_sym_matrix(vals):
    """Make 4x4 symmetrix matrix from upper triangular.
    Hard coded for 4x4
    Vals should be a Numpy array of length 10.
    """
    indices = [[], [1], [2, 5], [3, 6, 8]]
    matrix = []
    summ = 0
```

```
    for i in range(4, 0, -1):
        num = 4-i
        lst = []
        for index in indices[num]:
            lst.append(vals[index])
        matrix.append(lst + vals[summ:i+summ].tolist())
        summ += i
    return matrix
```

Listing 3: Function for generating a 4x4 symmetric matrix from 10 given values

Hundreds of optimisations were conducted, with devices running for several nights, but a non-linear policy that could stabilise the cart-pole, even for just a few simulation steps, could not be obtained. However, policies that could get quite close to the desired state, even for just a single simulation step, were found. It was thus decided to combine a non-linear policy with a linear policy - the non-linear policy would swing the pole up, and the linear policy would 'catch' the state of the system, and stabilise it. In fact, two linear policies were combined with a non-linear policy - both linear policies mentioned in task 2.3 were used i.e. the one with good range but worse stabilisation, and the one with worse range but good stabilisation. The policy with better range was used to bring the state even closer to the unstable equilibrium, and the policy with good stabilisation was then able to hold the desired state, [0, 0, 0, 0]. Plots showing these policies enacted on the system, starting from [0, 0, π, 0], are shown in Figure 36. This combined policy was in fact able to stabilise the system at the unstable equilibrium, with the pole vertically upwards, for infinite time. Testing was conducted to check if this combined policy was also able to stabilise the system when the other initial state parameters were non-zero. The maximum values for the three state variables from which the combined policy was able to stabilise the cart-pole system, whilst the other two were kept at zero and the angle kept at π, were:

- Cart position - 3.25,
- Cart velocity - 4.6,
- Pole velocity - 0.94.

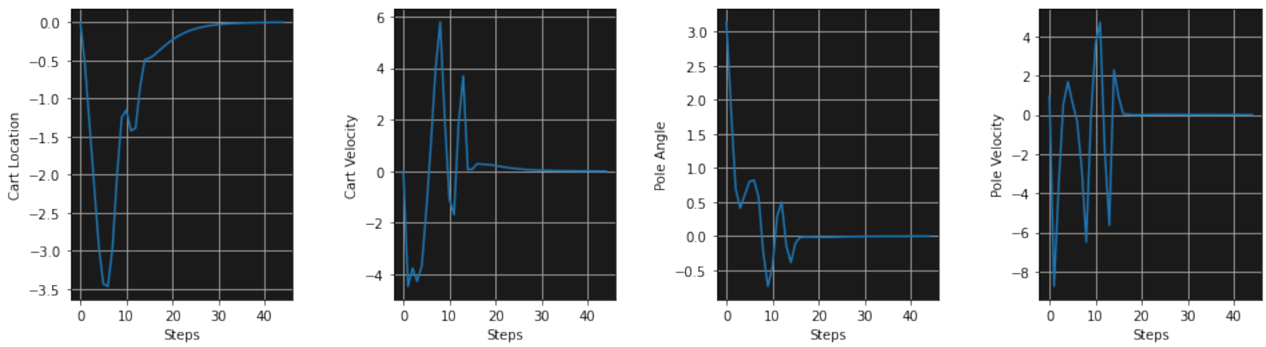


Figure 36: Best non-linear policy combined with the two optimised linear policies, to achieve stabilisation about the unstable equilibrium, when starting with the pole vertically downwards.

6 Conclusion

The first quarter of the project involved investigating the dynamics of the cart-pole system, and thus attempting to fit a linear regressive model to predict the next state after a single simulation step. When investigating the dynamics of the system, it was found that the dynamics were non-linear functions of the system's state, and therefore, as expected, the predictive peformance of the linear model was poor.

Building on the conclusions regarding the linear model, the second week of the project involved developing a non-linear model, by employing linear regression with non-linear Gaussian basis functions, to expand the dimensionality of the input state. Even when unoptimised, the non-linear model blew the linear model out of the water. Optimisation of the hyperparameters of the non-linear model, by conducting scans of mean squared error against the hyperparameters, led to a great boost in performance. The non-linear model was used to predict rollouts from various initial conditions - the model was always able to predict the first 10 simulation steps (2 seconds) to a relatively high degree of accuracy, and in some cases 40 steps! On the other hand, the linear model would always diverge.

The action taken (/force) was then added to the input state vector - all previous work assumed the force was always zero. The range of the force was -20 to 20 - thus the addition of force to the input state vector massively increased the dimensionality of the data. Non-linear models were trained on data including random forces - the dataset size was double from 5000 to 10000. Although not as impressive as the performance of the non-linear model on data omitting force (i.e. force is set to 0 for every state), the non-linear model trained on data including force achieved good performance. The model was always able to predict 10 steps of system dynamics, and in some cases could reach 20 steps. To achieve the same standard of performance as the model ignoring force, 5000 data points would be required for each value of force - clearly, this is not feasible.

The project then continued by introducing a linear policy, which would define the action that should be taken to give rise to the desired behaviour of the pole being balance vertically upwards (i.e. the unstable equilibrium, [0, 0, 0, 0]). Optimisation of policies led to the discovery of two policies with good performance. The first was able to stabilise the system and hold the state at the unstable equilibrium (for essentially infinite time), but in a tight range about [0, 0, 0, 0]. The range of the second policy was better, but its stabilising intensity was reduced - instead, the system would oscillate about the desired state, rather than hold the system at the state. The policies were applied to predicted states by the non-linear model, thus employing model predictive control, which gave rise to good results.

The third week of the project involved repeating all previous investigations, but adding noise. Noise was added in two different ways: to the observed states, and to the actual dynamics of the system (the equations of motion). Noise was of detriment to the all of the linear model, non-linear model, and linear policy.

The final week was to obtain a controller, through a non-linear policy, to stabilise the system at the unstable equilibrium, when starting with the pole facing vertically downwards (i.e. the stable equilibrium). Many experiments were conducted to find a suitable non-linear policy. However, a non-linear policy that could hold the pole vertically upwards could not be found. Instead, a decent non-linear policy was combined with the two optimised linear policies from the second week of the project; the non-linear policy flicked the pole upwards, and the linear policies kept the pole at this angle. The combined policy was very effective.

The goal of the project was achieved; a data-driven controller that could balance the pendulum on its unstable equilibrium was found.

A Code

Edits to the CartPole() class to include noise are shown in Listing 4.

```
def __init__(self, visual=False, noise=False, noise_frac=0.05):
    self.cart_location = 0.0
    self.cart_velocity = 0.0
    self.pole_angle = np.pi # angle is defined to be zero when the pole is upright, pi when hanging vertically down
    self.pole_velocity = 0.0
    self.visual = visual
    self.noise = noise
    self.noise_frac = noise_frac

def performAction(self, action = 0.0):
    # 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):
        if self.noise:
            self.cart_velocity += np.random.normal(0, np.sqrt(1/12)*20) * self.noise_frac
            self.pole_velocity += np.random.normal(0, np.sqrt(1/12)*30) * self.noise_frac

        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

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

Listing 4: Edits to the CartPole() class.

The rest of the code used to conduct all of the tasks in this project is provided in Listing 5.

```
def simulate(steps=1, initial_state=[0, 0, np.pi, 0], action=0, remap__angle=False, noise_frac=0, visual=False):
    """
    Simulate the cartpole system for a number of specificed steps from a specified initial state.
    Returning an array containing all the states (at each step), including the initial state.
    noise = fraction of signal
    Noise can be added to the observed state.
    """
    cp = CartPole(visual=visual) #Create CartPole object
    cp.setState(initial_state) #Initialise CartPole object with given initial state
    states = initial_state.copy() #Create copy of initial state array

```

```
    for step in range(steps): #PerformAction for a given number of steps
        cp.performAction(action)
        if remap__angle: #remap the angles to range [-pi, pi] if True
            cp.remap_angle()
        current_state = cp.getState() #Find state after one performAction
        current_state += np.random.normal(0, [1.5, np.sqrt(1/12)*20, np.sqrt(1/12)*2*np.pi, np.sqrt(1/12)*30]) * noise_frac
        states = np.vstack((states, current_state)) #Create stacked array with state after each performAction
    return states

def simulate_dynamicnoise(steps=1, initial_state=[0, 0, np.pi, 0], action=0, remap__angle=False, noise_frac=0):
    """
    Simulate the cartpole system for a number of specificed steps from a specified initial state.
    Returning an array containing all the states (at each step), including the initial state.
    noise = fraction of signal
    Here noise is not added to the observed state, but added to the dynamics of the cart pole,
    specifically the cart and pole velocities, which feed into the accelerations and positions.
    The accelerations feed back into the velocities.
    """
    if noise_frac != 0:
        noise = True
    else:
        noise = False
    cp = CartPole(noise=noise, noise_frac=noise_frac) #Create CartPole object
    cp.setState(initial_state) #Initialise CartPole object with given initial state
    states = initial_state.copy() #Create copy of initial state array

    for step in range(steps): #PerformAction for a given number of steps
        cp.performAction(action)
        if remap__angle: #remap the angles to range [-pi, pi] if True
            cp.remap_angle()
        current_state = cp.getState() #Find state after one performAction

        states = np.vstack((states, current_state)) #Create stacked array with state after each performAction
    return states

def display_plots(states, model=False, model_states=None):
    """
    Display plots of each variable (position, velocity, pole angle, pole velocity) against time (/number of steps).
    If predicted states from a model are provided, the predicted dynamics are plotted alongside the true time evolutions.
    """
    positions = states[:,0]
    velocities = states[:,1]
    angles = states[:,2]
    pole_vels = states[:,3]
    if model:
        assert len(model_states) == len(states)
        pred_pos = model_states[:,0]
        pred_vel = model_states[:,1]
        pred_ang = model_states[:,2]
        pred_pol_vel = model_states[:,3]

    time = range(len(states))

    fig, axs = plt.subplots(1, 4, figsize=(16, 4))
    axs[0].plot(time, positions, label='True Cart Position')
    axs[1].plot(time, velocities, label='True Cart Velocity')
    axs[2].plot(time, angles, label='True Pole Angle')
    axs[3].plot(time, pole_vels, label='True Pole Velocity')
    if model:
        axs[0].plot(time, pred_pos, '--', label='Predicted Cart Position')
        axs[1].plot(time, pred_vel, '--', label='Predicted Cart Velocity')
```

```
        axs[2].plot(time, pred_ang, '--', label='Predicted Pole Angle')
        axs[3].plot(time, pred_pol_vel, '--', label='Predicted Pole Velocity')

    axs[0].set_ylabel('Cart Location')
    axs[1].set_ylabel('Cart Velocity')
    axs[2].set_ylabel('Pole Angle')
    axs[3].set_ylabel('Pole Velocity')

    for i in range(4):
            if model:
                axs[i].legend()
            axs[i].set_xlabel('Steps')
            axs[i].set_facecolor((0.1, 0.1, 0.1))
            axs[i].grid()
    plt.subplots_adjust(wspace=0.45)

def phase_portraits(states):
    positions = states[:,0]
    velocities = states[:,1]
    angles = states[:,2]
    pole_vels = states[:,3]

    fig, axs = plt.subplots(1, 2, figsize=(12, 4))
    axs[0].plot(positions, velocities)
    axs[0].set_ylabel('Cart Velocity')
    axs[0].set_xlabel('Cart Position')
    axs[1].plot(angles, pole_vels)
    axs[1].set_ylabel('Pole Velocity')
    axs[1].set_xlabel('Pole Angle')
    axs[0].set_facecolor((0.1, 0.1, 0.1))
    axs[0].grid()
    axs[1].set_facecolor((0.1, 0.1, 0.1))
    axs[1].grid()

def plot_y_1step(initial_state, ranges):
    """
    Vary the initial value of each variable one-by-one (i.e. keeping the other 3 constant, set by initial_state)
    One step of performAction for each variable value.
    Plot of Y for different initial values of each variable.
    Y is the system's state after one step of performAction.
    Four plots are produced.
    """
    fig, axs = plt.subplots(1, 4, figsize=(16, 4))
    for i in range(4): #Iterate over the four state variables
        positions = []
        velocities = []
        angles = []
        pole_vels = []
        rng = ranges[i]
        for val in ranges[i]: #Iterate over the specified range for the current variable (perform scan)
            initial_state_copy = initial_state.copy()
            initial_state_copy[i] = val
            y = simulate(initial_state=initial_state_copy)[1] # y = next state
            positions.append(y[0])
            velocities.append(y[1])
            angles.append(y[2])
            pole_vels.append(y[3])

        axs[i].plot(ranges[i], positions, 'b-', label='Cart Location') #Plot each variable as function of scan
        axs[i].plot(ranges[i], velocities, 'g-', label='Cart Velocity')
        axs[i].plot(ranges[i], angles, 'r-', label='Pole Angle')
        axs[i].plot(ranges[i], pole_vels, 'y-', label='Pole Velocity')
        axs[i].set_facecolor((0.1, 0.1, 0.1))
        axs[i].grid()
        axs[i].legend()
```

```
    axs[0].set_xlabel('Initial Value of Cart Location')
    axs[1].set_xlabel('Initial Value of Cart Velocity')
    axs[2].set_xlabel('Initial Value of Pole Angle')
    axs[3].set_xlabel('Initial Value of Pole Velocity')
    axs[0].set_ylabel('Y = X(1)')

def plot_y_diff(initial_state, ranges, linear_model=None, nonlinear_model=None, figsize=(16,4)):
    """
    Vary the initial value of each variable one-by-one (i.e. keeping the other 3 constant, set by initial_state)
    One step of performAction for each variable value.
    Plot of Y for different initial values of each variable.
    Y is the difference between the system's state after one performAction and the initial state!!!
    Four plots are produced.
    If a model is provided, the model's predictions are also plotted (on the same axes)
    """
    fig, ((axs1, axs2, axs3, axs4)) = plt.subplots(1, 4, figsize=figsize)
    for i, axs in enumerate([axs1, axs2, axs3, axs4]):
        positions = []
        velocities = []
        angles = []
        pole_vels = []
        if linear_model:
            pred_pos = []
            pred_vel = []
            pred_ang = []
            pred_pol_vel = []
        if nonlinear_model:
            pred_pos2 = []
            pred_vel2 = []
            pred_ang2 =[]
            pred_pol_vel2 = []
        for val in ranges[i]:
            initial_state_copy = initial_state.copy()
            initial_state_copy[i] = val
            y = simulate(initial_state=initial_state_copy)[1] - np.array(initial_state_copy)
            positions.append(y[0])
            velocities.append(y[1])
            angles.append(y[2])
            pole_vels.append(y[3])
            if linear_model:
                y_pred = linear_model.predict([initial_state_copy])[0]
                pred_pos.append(y_pred[0])
                pred_vel.append(y_pred[1])
                pred_ang.append(y_pred[2])
                pred_pol_vel.append(y_pred[3])
            if nonlinear_model:
                alphas, kernel_centres, sigma = nonlinear_model[0], nonlinear_model[1], nonlinear_model[2]
                preds, preds_final = get_preds(np.array([initial_state_copy]), kernel_centres, sigma, alphas)
                pred_pos2.append(preds[0])
                pred_vel2.append(preds[1])
                pred_ang2.append(preds[2])
                pred_pol_vel2.append(preds[3])
        if i==0: #One legend is produced for whole subplot. This prevents same labels appearing in legend multiple times.
            label1, label2, label3, label4 = 'Cart Location', 'Cart Velocity', 'Pole Angle', 'Pole Velocity'
            label1x, label2x, label3x, label4x = 'Linear Predicted '+label1, 'Linear Predicted '+label2, 'Linear Predicted '+label3, 'Linear Predicted '+label4
            label1y, label2y, label3y, label4y = 'Non-Linear Predicted '+label1, 'Non-Linear Predicted '+label2, 'Non-Linear Predicted '+label3, 'Non-Linear Predicted '+label4
        else:
            label1 = label2 = label3 = label4 = None
            label1x = label2x = label3x = label4x = None
            label1y = label2y = label3y = label4y = None

        axs.plot(ranges[i], positions, 'b-', label=label1)
```

```
        axs.plot(ranges[i], velocities, 'g-', label=label2)
        axs.plot(ranges[i], angles, 'r-', label=label3)
        axs.plot(ranges[i], pole_vels, 'y-', label=label4)
        if linear_model:
            axs.plot(ranges[i], pred_pos, 'b--', label=label1x)
            axs.plot(ranges[i], pred_vel, 'g--', label=label2x)
            axs.plot(ranges[i], pred_ang, 'r--', label=label3x)
            axs.plot(ranges[i], pred_pol_vel, 'y--', label=label4x)
        if nonlinear_model:
            axs.plot(ranges[i], pred_pos2, 'b:', label=label1y)
            axs.plot(ranges[i], pred_vel2, 'g:', label=label2y)
            axs.plot(ranges[i], pred_ang2, 'r:', label=label3y)
            axs.plot(ranges[i], pred_pol_vel2, 'y:', label=label4y)
        axs.set_facecolor((0.1, 0.1, 0.1))
        axs.grid()
    fig.legend(loc='upper center', bbox_to_anchor=(0.5, 1), ncol=4)
    axs1.set_xlabel('Initial Value of Cart Location')
    axs2.set_xlabel('Initial Value of Cart Velocity')
    axs3.set_xlabel('Initial Value of Pole Angle')
    axs4.set_xlabel('Initial Value of Pole Velocity')
    axs1.set_ylabel('Y = X(1) - X(0)')

def plot_contours(initial_state, ranges):
    """
    Vary the initial values for TWO parameters, keeping the other two constant (set by initial_state)
    Contour plots of Y (= change in state after one step) as function of two scans.
    Four plots produced for each index pair (i.e. each scan).
    """
    titles = ['Cart Location', 'Cart Velocity', 'Pole Angle', 'Pole Velocity']
    for index in index_pairs: #obtain a pair of indices (e.g. [1, 3]) from a list of index pairs
        fig, (axs1, axs2, axs3, axs4) = plt.subplots(1, 4, figsize=(18, 3))
        i, j = index[0], index[1]
        x = y = np.zeros((len(ranges[i]), len(ranges[j]), 4))
        for k in range(len(ranges[i])): #Scan over the two specified parameters
            for l in range(len(ranges[j])):
                val1, val2 = ranges[i][k], ranges[j][l]
                initial_state_copy = initial_state.copy()
                initial_state_copy[i] = val1
                initial_state_copy[j] = val2
                x[k,l] = initial_state_copy
                state = simulate(initial_state=initial_state_copy)[1]
                y[k,l] = state - np.array(initial_state_copy)
        for m, axs in enumerate([axs1, axs2, axs3, axs4]):
            cs = axs.contourf(ranges[j], ranges[i], y[:,:,m], cmap='plasma')
            axs.set_title(titles[m])
            axs.set_xlabel('Initial Value of ' + titles[j])
            axs.set_ylabel('Initial Value of ' + titles[i])
            fig.colorbar(cs, ax=axs)
        plt.subplots_adjust(wspace=0.4)

def generate_data(n, train_prop=0.8, remap__angle=False, action_flag=False, noise_frac=0):
    """
    Generate n data points for training and testing a predictive model.
    The proportion of data set aside for training is set by train_prop (default = 80%)
    The input (x) is a random initial state.
    The output (y) is the change in state after a singular step.
    """
    x_stack = []
    y_stack = []
    for i in range(n):
        if action_flag:
            action = np.random.uniform(-20, 20)
        else:
            action = 0
        initial_state = [np.random.normal(loc=0, scale=1.5), np.random.uniform(-10, 10),
                        np.random.uniform(-np.pi, np.pi), np.random.uniform(-15, 15)] #Create random initial
                            ↪ state
```

```python
        x1 = simulate(initial_state=initial_state, remap__angle=remap__angle, action=action, noise_frac=noise_frac)[1] #Obtain state after one step
        y = x1 - np.array(initial_state) # y = change in state
        if action_flag:
            initial_state.append(action)
        x_stack.append(initial_state)
        y_stack.append(y)
    x_train, x_test = x_stack[:int(n*train_prop)], x_stack[int(n*train_prop):] #Split into proportion for training
    y_train, y_test = y_stack[:int(n*train_prop)], y_stack[int(n*train_prop):] #and testing
    return np.array(x_train), np.array(y_train), np.array(x_test), np.array(y_test)

def generate_data_dynamicnoise(n, train_prop=0.8, remap__angle=False, action_flag=False, noise_frac=0):
    """
    Generate n data points for training and testing a predictive model.
    The proportion of data set aside for training is set by train_prop (default = 80%)
    The input (x) is a random initial state.
    The output (y) is the change in state after a singular step.
    """
    x_stack = []
    y_stack = []
    for i in range(n):
        if action_flag:
            action = np.random.uniform(-20, 20)
        else:
            action = 0
        initial_state = [np.random.normal(loc=0, scale=1.5), np.random.uniform(-10, 10),
                         np.random.uniform(-np.pi, np.pi), np.random.uniform(-15, 15)] #Create random initial state
        x1 = simulate_dynamicnoise(initial_state=initial_state, remap__angle=remap__angle, action=action, noise_frac=noise_frac)[1] #Obtain state after one step
        y = x1 - np.array(initial_state) # y = change in state
        if action_flag:
            initial_state.append(action)
        x_stack.append(initial_state)
        y_stack.append(y)
    x_train, x_test = x_stack[:int(n*train_prop)], x_stack[int(n*train_prop):] #Split into proportion for training
    y_train, y_test = y_stack[:int(n*train_prop)], y_stack[int(n*train_prop):] #and testing
    return np.array(x_train), np.array(y_train), np.array(x_test), np.array(y_test)

def create_initialStates(n, action_flag=False):
    """
    Create a set of initial states.
    First two states are already specificed (first is stable equilibrium)
    Next n states are randomly generated (within training set ranges)
    """
    initial_states = [[0, 0, np.pi, 0], [-0.169, 9.607, 2.557, -14.155], [0.738, -0.467, 3.068, 14.384]]
    for i in range(n):
        initial_states.append([np.random.normal(loc=0, scale=1.5), np.random.uniform(-10, 10),
                               np.random.uniform(-np.pi, np.pi), np.random.uniform(-15, 15)])

    return initial_states

def plot_ModelVsTrue_OverTime(steps, initial_states, model):
    """
    Plot true and predicted time evolutions (dynamics) of the cart-pole system
    for a range of given initial states.
    """
    for i in range(len(initial_states)):
        initial_state = initial_states[i]
        pred_states, initial_state_copy = initial_state.copy(), initial_state.copy()
        true_states = simulate(initial_state=initial_state, steps=steps, remap__angle=True) #Simulate for n steps
        for step in range(steps): #Predict n times using given model, starting from initial state
            initial_state_copy[2] = remap_angle(initial_state_copy[2])
            next_pred = model.predict([initial_state_copy])[0] + initial_state_copy
```

```python
            next_pred[2] = remap_angle(next_pred[2])
            pred_states = np.vstack((pred_states, next_pred))
            initial_state_copy = next_pred
        #print(initial_state)
        display_plots(true_states, model=True, model_states=pred_states)

def get_kernel_centres(m, X):
    n = len(X)
    m_indices = []
    while len(m_indices) < m:
        num = np.random.randint(0, n)
        if num in m_indices:
            continue
        else:
            m_indices.append(num)
    kernel_centres = []
    for i in m_indices:
        kernel_centres.append(X[i])
    return kernel_centres

def kernel(X, X_prime, sigma):
    summ = 0
    for i in range(len(X)):
        if i != 2:
            val = ((X[i] - X_prime[i])**2)
        else:
            val = (np.sin((X[i]-X_prime[i])/2))**2
        summ += val/(2*sigma[i]**2)
    return np.exp(-summ)

def get_K_matrix(kernel_centres, X, sigma):
    n = len(X)
    m = len(kernel_centres)
    matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            matrix[i, j] = kernel(X[i], kernel_centres[j], sigma=sigma)

    return matrix

def train_alpha(X, y, kernel_centres, sigma, lamda):
    m = len(kernel_centres)
    K_mm = get_K_matrix(kernel_centres, X[:m], sigma=sigma)
    K_nm = get_K_matrix(kernel_centres, X, sigma=sigma)
    alpha = np.linalg.lstsq((np.matmul(K_nm.T, K_nm) + lamda*K_mm), np.matmul(K_nm.T, y), rcond=None)[0]
    return alpha

def get_preds(x_test, kernel_centres, sigma, alphas):
    K_nm_test = get_K_matrix(kernel_centres, x_test, sigma)
    preds = []
    for i in range(4):
        preds.append(np.matmul(K_nm_test, alphas[i]))
    preds_final = []
    for i, pred in enumerate(preds):
        pred_final = np.add(pred, x_test[:,i])
        preds_final.append(pred_final)
    return preds, preds_final

def plot_predicted_against_true(preds_final, y_test_final):
    fig, ((axs1, axs2, axs3, axs4)) = plt.subplots(1, 4, figsize=(16, 4))
    for i, axs in enumerate([axs1, axs2, axs3, axs4]):
        c = np.abs(preds_final[i] - y_test_final[:,i])
        axs.scatter(y_test_final[:,i], preds_final[i], c=c, cmap='plasma', s=8)
        axs.set_facecolor((0.1, 0.1, 0.1))
        axs.grid()

```

```
    axs1.set_xlabel('True Cart Location')
    axs1.set_ylabel('Predicted Cart Location')

    axs2.set_xlabel('True Cart Velocity')
    axs2.set_ylabel('Predicted Cart Velocity')

    axs3.set_xlabel('True Pole Angle')
    axs3.set_ylabel('Predicted Pole Angle')

    axs4.set_xlabel('True Pole Velocity')
    axs4.set_ylabel('Predicted Pole Velocity')

    plt.subplots_adjust(wspace=0.3)

def find_best_lambda(lambdas):
    labels = ['Cart Position', 'Cart Velocity', 'Pole Angle', 'Pole Velocity']
    n=1000
    x_train, y_train, x_test, y_test = generate_data(n, train_prop=0.8)
    sigma = [10, 16, 0.5, 11]
    m = 100
    kernel_centres = get_kernel_centres(m, x_train)
    K_nm_test = get_K_matrix(kernel_centres, x_test, sigma)
    errors = []
    for l in lambdas:
        alphas = []
        errors_for_given_l = []
        for i in range(4):
            alpha = train_alpha(X=x_train, y=y_train[:,i], kernel_centres=kernel_centres, sigma=sigma,
              ↪ lamda=l)
            pred = np.matmul(K_nm_test, alpha)
            errors_for_given_l.append(mse(y_test[:,i], pred))
        errors.append(errors_for_given_l)
    errors = np.array(errors)
    fig, axs = plt.subplots(1, 1, figsize=(6, 4))
    for i in range(4):
        axs.plot(lambdas, errors[:,i], label=labels[i])
    axs.set_xscale('log')
    axs.set_facecolor((0.1, 0.1, 0.1))
    axs.grid()
    axs.set_title('Mean Squared Error in Predicted Change of State')
    axs.set_ylabel('MSE')
    axs.set_xlabel('Lambda')
    axs.legend()

def find_best_N(Ns, x_test, y_test, m):
    labels = ['Cart Position', 'Cart Velocity', 'Pole Angle', 'Pole Velocity']
    sigma = [10, 16, 0.5, 11]
    errors = []
    for n in Ns:
        n = int(n)
        x_train, y_train, ignore1, ignore2 = generate_data(n, train_prop=1)
        kernel_centres = get_kernel_centres(m, x_train)
        K_nm_test = get_K_matrix(kernel_centres, x_test, sigma)
        alphas = []
        errors_for_given_n = []
        for i in range(4):
            alpha = train_alpha(X=x_train, y=y_train[:,i], kernel_centres=kernel_centres, sigma=sigma,
              ↪ lamda=1e-5)
            pred = np.matmul(K_nm_test, alpha)
            errors_for_given_n.append(mse(y_test[:,i], pred))
        errors.append(errors_for_given_n)
    errors = np.array(errors)
    fig, axs = plt.subplots(1, 1, figsize=(8, 6))
    for i in range(4):
        axs.plot(Ns, errors[:,i], label=labels[i])
    axs.set_facecolor((0.1, 0.1, 0.1))
    axs.grid()
```

```python
    axs.set_title('Mean Squared Error in Predicted Change of State')
    axs.set_ylabel('MSE')
    axs.set_xlabel('N')
    axs.legend()

def find_best_N2(Ns, x_test, y_test):
    labels = ['Cart Position', 'Cart Velocity', 'Pole Angle', 'Pole Velocity']
    sigma = [10, 16, 0.5, 11]
    errors = []
    for n in Ns:
        n = int(n)
        m = int(0.1*n)
        x_train, y_train, ignore1, ignore2 = generate_data(n, train_prop=1)
        kernel_centres = get_kernel_centres(m, x_train)
        K_nm_test = get_K_matrix(kernel_centres, x_test, sigma)
        alphas = []
        errors_for_given_n = []
        for i in range(4):
            alpha = train_alpha(X=x_train, y=y_train[:,i], kernel_centres=kernel_centres, sigma=sigma,
                ↪ lamda=1e-5)
            pred = np.matmul(K_nm_test, alpha)
            errors_for_given_n.append(mse(y_test[:,i], pred))
        errors.append(errors_for_given_n)
    errors = np.array(errors)
    fig, axs = plt.subplots(1, 1, figsize=(8, 6))
    for i in range(4):
        axs.plot(Ns, errors[:,i], label=labels[i])
    axs.set_facecolor((0.1, 0.1, 0.1))
    axs.grid()
    axs.set_title('Mean Squared Error in Predicted Change of State')
    axs.set_ylabel('MSE')
    axs.set_xlabel('N')
    axs.legend()

def find_best_M(Ms, n, sigma, lam):
    labels = ['Cart Position', 'Cart Velocity', 'Pole Angle', 'Pole Velocity']
    x_train, y_train, x_test, y_test = generate_data(n, train_prop=0.8)
    errors = []
    for m in Ms:
        kernel_centres = get_kernel_centres(m, x_train)
        K_nm_test = get_K_matrix(kernel_centres, x_test, sigma)
        alphas = []
        errors_for_given_m = []
        for i in range(4):
            alpha = train_alpha(X=x_train, y=y_train[:,i], kernel_centres=kernel_centres, sigma=sigma,
                ↪ lamda=lam)
            pred = np.matmul(K_nm_test, alpha)
            errors_for_given_m.append(mse(y_test[:,i], pred))
        errors.append(errors_for_given_m)
    errors = np.array(errors)
    fig, axs = plt.subplots(1, 1, figsize=(8, 6))
    for i in range(4):
        axs.plot(Ms, errors[:,i], label=labels[i])
    axs.set_facecolor((0.1, 0.1, 0.1))
    axs.grid()
    axs.set_title('Mean Squared Error in Predicted Change of State')
    axs.set_ylabel('MSE')
    axs.set_xlabel('M (Number of Kernel Centres)')
    axs.legend()

def plot_model_contours(initial_state, ranges, nonlinear_model, index_pairs):
    """
    Vary the initial values for TWO parameters, keeping the other two constant (set by initial_state)
    Contour plots of Y (= change in state after one step) as function of two scans.
    Four plots produced for each index pair (i.e. each scan).
    """
    titles = ['Cart Location', 'Cart Velocity', 'Pole Angle', 'Pole Velocity']
```

```
    for index in index_pairs: #obtain a pair of indices (e.g. [1, 3]) from a list of index pairs
        fig, (axs1, axs2, axs3, axs4) = plt.subplots(1, 4, figsize=(18, 3))
        i, j = index[0], index[1]
        x = y = np.zeros((len(ranges[i]), len(ranges[j]), 4))
        for k in range(len(ranges[i])): #Scan over the two specified parameters
            for l in range(len(ranges[j])):
                val1, val2 = ranges[i][k], ranges[j][l]
                initial_state_copy = initial_state.copy()
                initial_state_copy[i] = val1
                initial_state_copy[j] = val2
                x[k,l] = initial_state_copy
                alphas, kernel_centres, sigma = nonlinear_model[0], nonlinear_model[1], nonlinear_model[2]
                preds, preds_final = get_preds(np.array([initial_state_copy]), kernel_centres, sigma, alphas)
                y[k,l] = preds
        for m, axs in enumerate([axs1, axs2, axs3, axs4]):
            cs = axs.contourf(ranges[j], ranges[i], y[:,:,m], cmap='plasma') #Plot contours
            axs.set_title(titles[m])
            axs.set_xlabel('Initial Value of ' + titles[j])
            axs.set_ylabel('Initial Value of ' + titles[i])
            fig.colorbar(cs, ax=axs)
        plt.subplots_adjust(wspace=0.4)

def plot_ModelVsTrue_OverTime2(steps, initial_states, nonlinear_model, action=0):
    """
    Plot true and predicted time evolutions (dynamics) of the cart-pole system
    for a range of given initial states.
    """
    alphas, kernel_centres, sigma = nonlinear_model[0], nonlinear_model[1], nonlinear_model[2]
    for i in range(len(initial_states)):
        initial_state = initial_states[i]
        pred_states, initial_state_copy = initial_state.copy(), initial_state.copy()
        true_states = simulate(initial_state=initial_state, steps=steps, remap__angle=True, action=action) #Simulate for n steps
        for _ in range(steps): #Predict n times using given model, starting from initial state
            initial_state_copy[2] = remap_angle(initial_state_copy[2])
            if action != 0:
                initial_state_copy.append(action)
            preds, preds_final = get_preds(np.array([initial_state_copy]), kernel_centres, sigma, alphas)
            next_pred = [0, 0, 0, 0]
            for i in range(4):
                next_pred[i] = preds_final[i][0]
            next_pred[2] = remap_angle(next_pred[2])
            pred_states = np.vstack((pred_states, next_pred))
            initial_state_copy = next_pred
        display_plots(true_states, model=True, model_states=pred_states)

def plot_y_diff2(initial_state, initial_force, ranges, linear_model=None, nonlinear_model=None, figsize=(14,10)):
    """
    Vary the initial value of each variable one-by-one (i.e. keeping the other 3 constant, set by initial_state)
    One step of performAction for each variable value.
    Plot of Y for different initial values of each variable.
    Y is the difference between the system's state after one performAction and the initial state!!!
    Four plots are produced.
    If a model is provided, the model's predictions are also plotted (on the same axes)
    Includes force!!!!
    """
    fig, ((axs1, axs2), (axs3, axs4), (axs5, axs6)) = plt.subplots(1, 5, figsize=figsize)
    for i, axs in enumerate([axs1, axs2, axs3, axs4, axs5]):
        positions = []
        velocities = []
        angles = []
        pole_vels = []
        if linear_model:
            pred_pos = []
```

```
            pred_vel = []
            pred_ang = []
            pred_pol_vel = []
        if nonlinear_model:
            pred_pos2 = []
            pred_vel2 = []
            pred_ang2 =[]
            pred_pol_vel2 = []
        for val in ranges[i]:
            initial_state_copy = initial_state.copy()
            if i != 4:
                initial_state_copy[i] = val
                action = initial_force
            else:
                action = val
            y = simulate(initial_state=initial_state_copy, action=action)[1] - np.array(initial_state_copy)
            positions.append(y[0])
            velocities.append(y[1])
            angles.append(y[2])
            pole_vels.append(y[3])
            if linear_model:
                y_pred = linear_model.predict([initial_state_copy])[0]
                pred_pos.append(y_pred[0])
                pred_vel.append(y_pred[1])
                pred_ang.append(y_pred[2])
                pred_pol_vel.append(y_pred[3])
            if nonlinear_model:
                alphas, kernel_centres, sigma = nonlinear_model[0], nonlinear_model[1], nonlinear_model[2]
                initial_state_copy.append(action)
                preds, preds_final = get_preds(np.array([initial_state_copy]), kernel_centres, sigma, alphas)
                pred_pos2.append(preds[0])
                pred_vel2.append(preds[1])
                pred_ang2.append(preds[2])
                pred_pol_vel2.append(preds[3])
        if i==0: #One legend is produced for whole subplot. This prevents same labels appearing in legend multiple times.
            label1, label2, label3, label4 = 'Cart Location', 'Cart Velocity', 'Pole Angle', 'Pole Velocity'
            label1x, label2x, label3x, label4x = 'Linear Predicted '+label1, 'Linear Predicted '+label2, 'Linear Predicted '+label3, 'Linear Predicted '+label4
            label1y, label2y, label3y, label4y = 'Non-Linear Predicted '+label1, 'Non-Linear Predicted '+label2, 'Non-Linear Predicted '+label3, 'Non-Linear Predicted '+label4
        else:
            label1 = label2 = label3 = label4 = None
            label1x = label2x = label3x = label4x = None
            label1y = label2y = label3y = label4y = None

        axs.plot(ranges[i], positions, 'b-', label=label1)
        axs.plot(ranges[i], velocities, 'g-', label=label2)
        axs.plot(ranges[i], angles, 'r-', label=label3)
        axs.plot(ranges[i], pole_vels, 'y-', label=label4)
        if linear_model:
            axs.plot(ranges[i], pred_pos, 'b--', label=label1x)
            axs.plot(ranges[i], pred_vel, 'g--', label=label2x)
            axs.plot(ranges[i], pred_ang, 'r--', label=label3x)
            axs.plot(ranges[i], pred_pol_vel, 'y--', label=label4x)
        if nonlinear_model:
            axs.plot(ranges[i], pred_pos2, 'b:', label=label1y)
            axs.plot(ranges[i], pred_vel2, 'g:', label=label2y)
            axs.plot(ranges[i], pred_ang2, 'r:', label=label3y)
            axs.plot(ranges[i], pred_pol_vel2, 'y:', label=label4y)
        axs.set_facecolor((0.1, 0.1, 0.1))
        axs.grid()
    fig.legend(loc='upper center', bbox_to_anchor=(0.5, 1), ncol=2)
    axs1.set_xlabel('Initial Value of Cart Location')
    axs2.set_xlabel('Initial Value of Cart Velocity')
    axs3.set_xlabel('Initial Value of Pole Angle')
```

```
    axs4.set_xlabel('Initial Value of Pole Velocity')
    axs.set_xlabel('Initial Value of Force')
    axs1.set_ylabel('Y = X(1) - X(0)')
    axs6.set_visible(False)

def plot_loss_rollout(steps, initial_state, action=0, plot_states=False):
    losses = []
    total_loss = []
    losses.append(loss(initial_state))
    total_loss.append(loss(initial_state))
    states = initial_state.copy()
    for i in range(steps):
        #If action, same action used at each step.
        next_state = simulate(steps=1, initial_state=initial_state, action=action)[1]
        initial_state = next_state
        losses.append(loss(next_state))
        total_loss.append(loss(next_state) + total_loss[i])
        states = np.vstack((states, next_state))
    fig, ((axs)) = plt.subplots(1, 1, figsize=(6, 4))
    axs.plot(range(steps+1), losses, label='Loss at each step')
    axs.plot(range(steps+1), total_loss, label='Total loss of trajectory')
    axs.legend()
    axs.set_ylabel('Loss')
    axs.set_xlabel('Steps')
    axs.set_facecolor((0.1, 0.1, 0.1))
    axs.grid()
    if plot_states:
        display_plots(states)

def policy(p, X):
    return np.dot(p, X)

def plot_policy_scans(initial_p, initial_state, policy_range):
    fig, ((axs1, axs2, axs3, axs4)) = plt.subplots(1, 4, figsize=(16, 4))
    for i, axs in enumerate([axs1, axs2, axs3, axs4]):
        losses = []
        for val in policy_range:
            p = initial_p.copy()
            p[i] = val
            p_X = policy(p, initial_state)
            next_state = simulate(steps=1, initial_state=initial_state, action=p_X)[1]
            losses.append(loss(next_state))
        axs.plot(policy_range, losses)
        axs.set_ylabel('Loss after one step')
        axs.set_xlabel('Value of p_' + str(i))
        axs.set_facecolor((0.1, 0.1, 0.1))
        axs.grid()
    plt.subplots_adjust(wspace=0.35)

def plot_policy_contours(initial_p, initial_state, policy_range, index_pairs, steps=1):
    fig, ((axs1, axs2, axs3), (axs4, axs5, axs6)) = plt.subplots(2, 3, figsize=(14, 8))
    for x, axs in enumerate([axs1, axs2, axs3, axs4, axs5, axs6]):
        index = index_pairs[x]
        i, j = index[0], index[1]
        grid = np.zeros((len(policy_range), len(policy_range)))
        for k, val1 in enumerate(policy_range): #Scan over the two specified parameters
            for l, val2 in enumerate(policy_range):
                p = initial_p.copy()
                p[i] = val1
                p[j] = val2
                loss_ = 0
                initial_state_copy = initial_state.copy()
                for _ in range(steps):
                    p_X = policy(p, initial_state)
                    next_state = simulate(steps=1, initial_state=initial_state_copy, action=p_X)[1]
                    loss_ += loss(next_state)
                    initial_state_copy = next_state
```

```
                grid[k][l] = loss_
        cs = axs.contourf(policy_range, policy_range, grid, cmap='plasma')
        axs.set_xlabel('p_' + str(i))
        axs.set_ylabel('p_' + str(j))
        fig.colorbar(cs, ax=axs)
    plt.subplots_adjust(wspace=0.35, hspace=0.3)

def training_loss(p, initial_state, steps=20):
    loss_ = 0
    initial_state_copy = initial_state.copy()
    for _ in range(steps):
        p_X = policy(p, initial_state_copy)
        next_state = simulate(steps=1, initial_state=initial_state_copy, action=p_X)[1]
        loss_ += loss(next_state)
        initial_state_copy = next_state
    return loss_

def objective_function(p, initial_state):
    return training_loss(p, initial_state)

def train_policy(initial_state, n):
    """
    Train linear policy to achieve [0, 0, 0, 0] from given initial state.
    Finds policy [p_0, p_1, p_2, p_3] which minimises loss.
    Start from a random initial p.
    Attempt n times from different settings of initial p, to find a good optimum.
    Returns results from each trial.
    """
    losses = []
    initial_ps = []
    opt_ps = []
    for _ in range(n):
        initial_p = [np.random.uniform(-25, 25), np.random.uniform(-25, 25),
                     np.random.uniform(-25, 25), np.random.uniform(-25, 25)]
        initial_ps.append(initial_p)
        opt_p = scipy.optimize.minimize(objective_function, initial_p, args=(initial_state),
            ↪ method='Nelder-Mead')['x']
        opt_ps.append(opt_p)
        losses.append(objective_function(opt_p, initial_state))
    return losses, initial_ps, opt_ps

def train_policy2(initial_state, n, objective_function):
    """
    Train linear policy to achieve [0, 0, 0, 0] from given initial state.
    Finds policy [p_0, p_1, p_2, p_3] which minimises loss.
    Start from a random initial p.
    Attempt n times from different settings of initial p, to find a good optimum.
    Only returns the best policy.
    """
    opt_p = [0, 0, 0, 0]
    best_loss = objective_function(opt_p, initial_state)
    for _ in range(n):
        initial_p = [np.random.uniform(-25, 25), np.random.uniform(-25, 25),
                     np.random.uniform(-25, 25), np.random.uniform(-25, 25)]
        p = scipy.optimize.minimize(objective_function, initial_p, args=(initial_state),
            ↪ method='Nelder-Mead')['x']
        if objective_function(p, initial_state) < best_loss:
            best_loss = objective_function(p, initial_state)
            opt_p = p
    return opt_p, best_loss

def model_predictive_control_loss(p, initial_state, nonlinear_model, steps=20):
    alphas, kernel_centres, sigma = nonlinear_model[0], nonlinear_model[1], nonlinear_model[2]
    loss_ = 0
    initial_state_copy = initial_state.copy()
    for _ in range(steps):
        p_X = policy(p, initial_state_copy)
```

```
        initial_state_copy.append(p_X)
        preds, preds_final = get_preds(np.array([initial_state_copy]), kernel_centres, sigma, alphas)
        next_state = []
        for i in preds_final:
            next_state.append(i[0])
        loss_ += loss(next_state)
        initial_state_copy = next_state
    return loss_

def mpc_objective_function(p, initial_state, nonlinear_model):
    return model_predictive_control_loss(p, initial_state, nonlinear_model)

def get_policy_true_pred_states(p_opt, initial_state, nonlinear_model, steps=20):
    """Model predictive control.

    Apply given policy to predictions of states by given non-linear model.
    The true and predicted states are obtained and plotted against each other.
    """
    initial_state_copy = initial_state.copy()
    true_states = initial_state.copy()
    pred_states = initial_state.copy()
    alphas, kernel_centres, sigma = nonlinear_model[0], nonlinear_model[1], nonlinear_model[2]

    for _ in range(steps):
        p_X = policy(p_opt, initial_state_copy)
        next_true_state = simulate(steps=1, initial_state=initial_state_copy, action=p_X)[1]
        initial_state_copy.append(p_X)
        preds, preds_final = get_preds(np.array([initial_state_copy]), kernel_centres, sigma, alphas)
        next_state = []
        for i in preds_final:
            next_state.append(i[0])
        true_states = np.vstack((true_states, next_true_state))
        pred_states = np.vstack((pred_states, next_state))
        initial_state_copy = next_state
    return true_states, pred_states

def plot_set_of_states(set_of_states, labels):
    fig, axs = plt.subplots(1, 4, figsize=(16, 4))
    for i, states in enumerate(set_of_states):
        positions = states[:,0]
        velocities = states[:,1]
        angles = states[:,2]
        pole_vels = states[:,3]

        time = range(len(states))

        axs[0].plot(time, positions, label=labels[i])
        axs[1].plot(time, velocities)
        axs[2].plot(time, angles)
        axs[3].plot(time, pole_vels)


        axs[0].set_ylabel('Cart Location')
        axs[1].set_ylabel('Cart Velocity')
        axs[2].set_ylabel('Pole Angle')
        axs[3].set_ylabel('Pole Velocity')

    for i in range(4):
            axs[i].set_xlabel('Steps')
            axs[i].set_facecolor((0.1, 0.1, 0.1))
            axs[i].grid()
    plt.subplots_adjust(wspace=0.45)
    fig.legend(loc='upper center', bbox_to_anchor=(0.5, 1), ncol=2)

def noisy_training_loss(p, initial_state, steps=20, noise_frac=0.05):
    loss_ = 0
    initial_state_copy = initial_state.copy()
```

```
    for _ in range(steps):
        p_X = policy(p, initial_state_copy)
        next_state = simulate(steps=1, initial_state=initial_state_copy, action=p_X,
            ↪ noise_frac=noise_frac)[1]
        loss_ += loss(next_state)
        initial_state_copy = next_state
    return loss_

def noisy_objective_function(p, initial_state, noise_frac=0.05):
    return noisy_training_loss(p, initial_state, noise_frac=noise_frac)

def get_policy_true_noisy_states(p_opt, initial_state, noise_fract, dynamic=False):
    """Policy applied to noisy observed states.
    Also obtain the true states.
    """
    initial_state_copy = initial_state.copy()
    true_states = initial_state.copy()
    noisy_states = initial_state.copy()

    for _ in range(20):
        p_X = policy(p_opt, initial_state_copy)
        next_true_state = simulate(steps=1, initial_state=initial_state_copy, remap__angle=True,
            ↪ action=p_X)[1]
        if dynamic:
            next_noisy_state = simulate_dynamicnoise(steps=1, initial_state=initial_state_copy, action=p_X,
                ↪ noise_frac=noise_fract)[1]
        else:
            next_noisy_state = simulate(steps=1, initial_state=initial_state_copy, action=p_X,
                ↪ noise_frac=noise_fract)[1]
        true_states = np.vstack((true_states, next_true_state))
        noisy_states = np.vstack((noisy_states, next_noisy_state))
        next_noisy_state[2] = remap_angle(next_noisy_state[2])
        initial_state_copy = next_noisy_state
    return true_states, noisy_states

def non_linear_policy(X, weights, W, centres):
    p_X = 0
    for i, centre in enumerate(centres):
        val = X - centre
        power = -0.5 * np.matmul(val.T, np.matmul(W, val, dtype=np.float64), dtype=np.float64) #np.matmul
            ↪ overflow
        p_X += weights[i] * np.exp(power, dtype=np.float64)
    #if np.isnan(p_X):
        #p_X = 0
    return p_X

def non_linear_policy2(X, weights, W, centres):
    """Include periodicity."""
    p_X = 0
    for i, centre in enumerate(centres):
        val = []
        for j in range(len(X)):
            if j != 2:
                val.append(X[j] - centre[j])
            else:
                val.append(np.sin(X[j] - centre[j]))
        val = np.array(val)
        power = -0.5 * np.matmul(val.T, np.matmul(W, val))
        p_X += weights[i] * np.exp(power)
    #print(p_X)
    return p_X

def make_sym_matrix(vals):
    """Make 5x5 symmetrix matrix from upper triangular.
    Hard coded for 5x5.
    vals should be an array of length 15.
    """
```

```
    indices = [[], [1], [2, 6], [3, 7, 10], [4, 8, 11, 13]]
    matrix = []
    summ = 0
    for i in range(5, 0, -1):
        num = 5-i
        lst = []
        for index in indices[num]:
            lst.append(vals[index])
        matrix.append(lst + vals[summ:i+summ].tolist())
        summ += i
    return matrix

def make_44_sym_matrix(vals):
    """Make 4x4 symmetrix matrix from upper triangular.
    Hard coded for 4x4
    vals should be an array of length 10.
    """
    indices = [[], [1], [2, 5], [3, 6, 8]]
    matrix = []
    summ = 0
    for i in range(4, 0, -1):
        num = 4-i
        lst = []
        for index in indices[num]:
            lst.append(vals[index])
        matrix.append(lst + vals[summ:i+summ].tolist())
        summ += i
    return matrix

def split_non_linear_policy(p, num_centres=20):
    weights = np.array(p[:num_centres], dtype=np.float64)
    W = np.array(p[num_centres:num_centres+16], dtype=np.float64).reshape(4,4)
    centres = np.array(p[num_centres+16:], dtype=np.float64).reshape(num_centres, 4)
    #W = make_sym_matrix(W) #Make W symmetric
    W = W.T * W
    return weights, W, centres

def nonlinear_training_loss(p, initial_state, non_linear_policy, loss_func, steps=20, num_centres=20):
    weights, W, centres = split_non_linear_policy(p, num_centres=num_centres)
    loss_ = 0
    initial_state_copy = initial_state.copy()
    for j in range(steps):
        p_X = non_linear_policy(initial_state_copy, weights, W, centres)

        if j < 3 and np.isnan(p_X):
            break
        elif np.isnan(p_X):
            p_X = 0

        next_state = simulate(steps=1, initial_state=initial_state_copy[:4], action=p_X)[1]
        loss_ += loss_func(next_state)
        #next_state = np.append(next_state, p_X)
        initial_state_copy = next_state
    return loss_

def train_nonlinear_policies(initial_state, no_policies, path, loss_func):
    os.makedirs(path, exist_ok=True)
    for _ in range(no_policies):
        num_centres = 20
        centres = []
        for i in range(num_centres):
            centres.append(np.array([np.random.normal(loc=0, scale=3), np.random.uniform(-7.5, 7.5),
                                     np.random.uniform(-np.pi, np.pi), np.random.uniform(-12, 12)]))
        centres = np.array(centres, dtype=np.float64)
        centres2 = centres.flatten()

        W = np.random.rand(4, 4).astype(np.float64)
```

```
        W2 = W.flatten()

        weights = np.random.rand(num_centres).astype(np.float64)

        p = np.hstack((weights, W2, centres2)).astype(np.float64)

        nonlinear_p_opt = scipy.optimize.minimize(nonlinear_training_loss, p, args=(initial_state,
            ↪ non_linear_policy, loss_func),
                                                  method='Nelder-Mead')['x'].astype(np.float64)
        np.save(path + '/loss=' + str(nonlinear_training_loss(nonlinear_p_opt, initial_state,
            ↪ non_linear_policy, loss_func=loss_func)),
                nonlinear_p_opt)
        del centres, centres2, W, W2, weights, nonlinear_p_opt
        gc.collect()

def loss2(state):
    """Penalises angle more."""
    sig = 10
    sum_loss = 0
    for i, x in enumerate(state):
        if i != 2:
            sum_loss += -np.dot(x,x)/(2.0 * sig**2)
        else:
            sum_loss += -np.dot(x,x)/(2.0 * np.pi**2)
    return 1 - np.exp(sum_loss)

#Loading a policy
name = 'loss=14.146216677630209.npy'
path = './nonlinear_policies_initialstate1_noforce_sigmanew/'

pol = np.load(path + name, allow_pickle=True)
initial_state = np.array([0, 0, np.pi, 0])
print(nonlinear_training_loss(pol, initial_state, non_linear_policy, loss3))
weights, W, centres = split_non_linear_policy(pol)

#Plotting results of loaded policy
#%matplotlib
#visual = True
visual = False
initial_state_copy = [0, 0, np.pi, 0]
states = initial_state.copy()[:4]
for _ in range(20):
    p_X = non_linear_policy(initial_state_copy[:4], weights, W, centres)
    next_state = simulate(steps=1, initial_state=initial_state_copy[:4], action=p_X, visual=visual)[1]
    print(next_state)
    states = np.vstack((states, next_state))
    next_state = np.append(next_state, p_X)
    initial_state_copy = next_state
display_plots(states)
```

Listing 5: Code for this project.