TransDynaMo – Using Transformers to Learn Dynamical Models

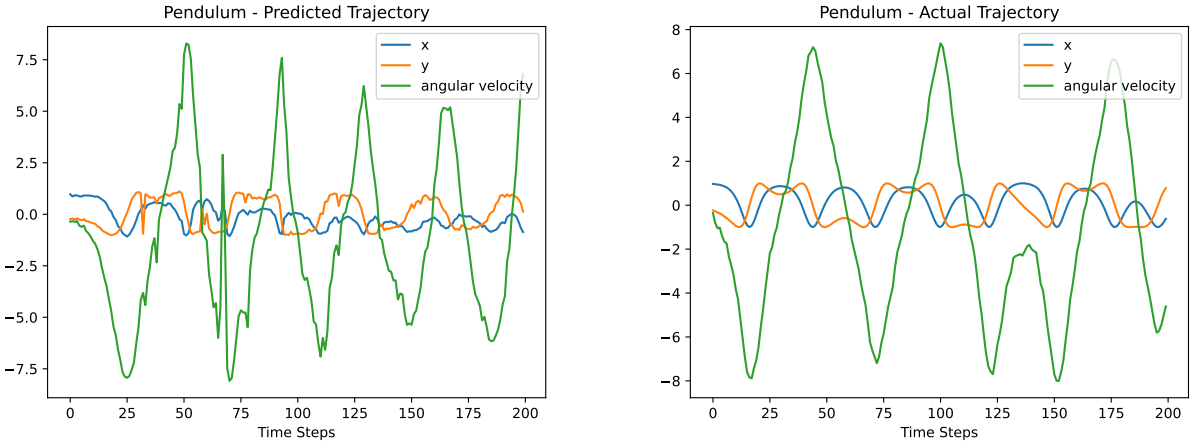
CS444 - Deep Learning for Computer Vision
Spring 2023

University of Illinois Urbana-Champaign

Authors:

Maulik Bhatt mcbhatt2@illinois.edu
Amogh Pandey amoghp3@illinois.edu

In this project, we aim to learn dynamical models of systems by leveraging the proven power of the Transformer architecture to perform sequence to sequence modelling tasks. We pose as an input sequence the trajectory of the system (which is a sequence of states and actions), and show that the output sequence produced by our Transformer Architecture is a set of states that would be the next states of the dynamical system, if the system started in the given initial state and took the given sequence of actions. The results of our effort are quite promising. When we compare a real trajectory from a dynamical system to the predicted trajectory (output of a transformer which has "learned" the dynamical model of the system), we see a resemblance between the two trajectories (1). This result shows that when one models the problem of learning dynamical models appropriately as a sequence to sequence modelling problem, a Transformer is a good candidate to capture the dynamics of the system. The codebase of this work can be found at - https://github.com/maulikbhatt585/TransDynaMo



(a) Pendulum Predicted Trajectory

(b) Pendulum Ground Truth

Figure 1: Comparison of predicted and ground truth trajectories shows that that predicted trajectories closely resembles the ground truth

Contents

1 Introduction

Now, more than ever before, robots have become increasingly prevalent in our society, finding themselves used in fields such as medicine, manufacturing, and even on the battlefield. However, a key challenge in developing robotic systems is to develop accurate models of their dynamics, which are necessary for control, motion planning, and safety analysis. For example, suppose we would like to use a robotic car to complete a way-point navigation task in an environment with obstacles (this is a key task in the autonomous navigation of vehicles). A common control method that an engineer may choose to achieve this task is known as model predictive control (MPC). However, in order to use the MPC algorithm, the control engineer will require a dynamics model $\dot{x} = f(x(t), u(t))$ of the car to be able to accurately model how the state of the system (car) evolves with respect to control actions. Similarly, for motion planning knowledge of the robot's dynamics is required to generate feasible motion plans. As a final example, consider the safety analysis of a robotic system, which would require knowledge of the system's dynamical model to define safety limits (for example set of safe/unsafe states) of the system, and to understand the "safe set" of control actions which keep the system in the safe set of states, and the "unsafe set" of control actions which breaches the safe set of states and possibly violating safety constraints. Our goal is to find a simple and efficient way to learn dynamic models of robotic systems.

Having understood the importance of knowing the dynamical models of the robotics systems being used, let us now take a look at some of the existing methods used for developing dynamical models. There are many different ways to model dynamical systems. Some of the most common methods include:

- Ordinary differential equations (ODEs) are a type of mathematical equation that describes the evolution of a system over time. ODEs are often used to model physical systems, such as the motion of a pendulum or the flow of water in a pipe.
- Partial differential equations (PDEs) are a type of mathematical equation that describes the evolution of a system in two or more dimensions. PDEs are often used to model physical systems, such as the weather or the flow of heat in a solid.
- Agent-based models (ABMs) are a type of simulation that models the behavior of individual agents and how their interactions affect the system as a whole. ABMs are often used to model social systems, such as the spread of disease or the evolution of cooperation.
- Machine learning (ML) is a field of computer science that uses statistical techniques to learn from data. ML can be used to model dynamic systems by learning the relationships between the system's inputs and outputs.

The choice of modeling method depends on the specific system being modeled. For example, ODEs are often a good choice for modeling systems with a small number of variables, while PDEs are often a good choice for modeling systems with a large number of variables. ABMs are often a good choice for modeling systems with complex interactions between agents, while ML can be used to model systems where the relationships between the inputs and outputs are not known. No matter which modeling method is chosen, it is important to validate the model by comparing its predictions to real-world data. This helps to ensure that the model is accurate and can be used to make reliable predictions.

Traditional dynamical modeling methods rely heavily on advanced mathematical techniques and mathematical models. For example, Dynamic Mode Decomposition (DMD) [8] is based on dimensionality reduction. Other methods such as [1] rely on spectral submanifolds or Koopman Operators to learn ordinary differential equations (ODEs) governing system dynamics. In recent years, machine

learning and deep learning have been used to learn dynamical models. [5] uses transformers, but has a theory based on Koopman Operators, [11] leverages neural networks, but also uses Koopman Operator theory, and [10] uses transformers in conjunction with some data processing to extract certain geometrical properties of the data.

As foreshadowed by a few of the aforementioned works [5] [10], Transformers have seen some use in dynamics modeling, however, as previously mentioned, these works use transformers in conjunction with certain mathematical tools like Koopman Operators or rely on geometrical properties of the data. These methods are not very generalizable, and quite over-complicated.

We propose a simpler Transformer-based method, TransDynaMo, to learn dynamical models by exploiting a task that the Transformer is excellent at performing: sequence-to-sequence modeling. In our approach, we structure the problem of learning a dynamical model as a sequence-to-sequence task. Specifically, we pose as an input sequence trajectory of the system (which is a sequence of states and actions), and show that the output sequence produced by our Transformer Architecture is a set of states that would be the next states of the dynamical system if the system started in the given initial state and took the given sequence of actions (3).

2 Transformers

One of the most promising deep learning models for sequential data is the Transformer [12], which was originally developed for natural language processing tasks (such as machine translation) but has since been applied to various other tasks such as dealing with images [4] and speech [3]. The most powerful part of the Transformer architecture is actually the attention mechanism. The Transformer model uses self-attention mechanisms to capture dependencies between different parts of the input sequence. When an input sequence is given to a transformer, the self-attention mechanism generates attention weights for each element of the input sequence with respect to the other elements. These weights capture how related two elements of the input sequence are to each other. This self-attention mechanisms allow each element in the sequence to attend to other elements. This enables the model to capture non-linear relationships between different time steps, which is valuable for modeling the non-linear dynamics often present in time series. Therefore, we employ Transformers to model dynamical systems as follows.

3 Approach

In this section, we present our algorithm, TransDynaMo which presents the problem of modeling dynamical systems as a sequence-to-sequence modeling problem and solves it using Transformer architecture. We take motivation from the Decision Transformer [2] architecture which posits the reinforcement learning of accumulated rewards maximization as a sequence-to-sequence prediction problem. The Decision Transformer aims to solve an offline Reinforcement Learning (RL) problem. Given the past trajectory of the system, Decision Transformer learns to predict future actions through transformer architecture. Therefore the Decision Transformer is learning a policy and is solving a Model Free offline RL problem. In contrast, the problem we aim to solve is that of learning the dynamical model for a system. We outline a novel approach to applying the transformer architecture in a similar (but simpler fashion) as the Decision Transformer to the problem of learning dynamical models.

3.1 Notation

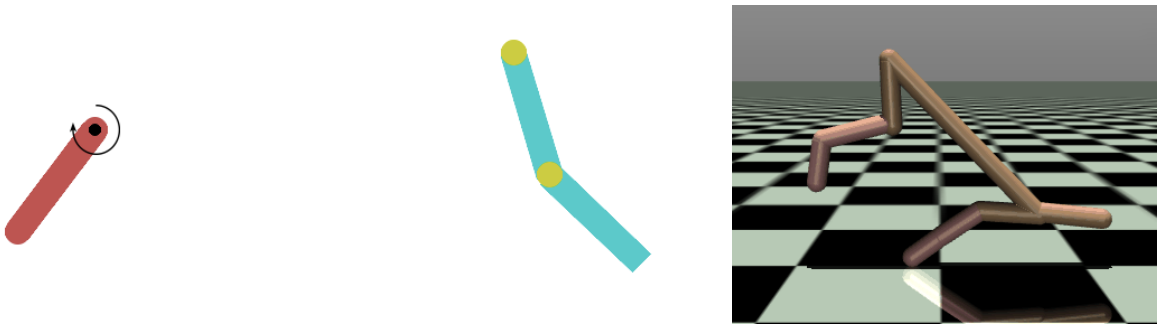
We consider the problem of modeling a discrete-time dynamical system. The state of the system at time step t is denoted by $x_t \in \mathcal{X}$ where $\mathcal{X}$ is the set of states of the system. The control action for the system at time step t is denoted through $u_t \in \mathcal{U}$ where $\mathcal{U}$ is the set of control inputs of the system. We assume that system evolves according to unknown dynamics $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ given by

$$x_{t+1} = f(x_t, u_t). \tag{1}$$

We denote the full trajectory of the system as $\tau = (x_0, u_0, x_1, u_1, \ldots, x_{T-1}, u_{T-1}, x_T)$ and trajectory of system up to time step t is denoted as $\tau_t = (x_0, u_0, x_1, u_1, \ldots, x_{t-1}, u_{t-1})$. Our aim is to obtain a learned estimate of the dynamics of the system $x_{t+1} \approx \hat{f}_\theta(\tau_t)$ parameterized by θ (in our case, parameters of the transformer neural network) that accurately predicts the next state of the system given the history of the trajectory of the system.

3.2 Data Generation

In order to generate for our experiments, we use OpenAI's Gymnasium API [7]. Gymnasium is an open-source Python library that contains various environments for testing various machine learning and reinforcement learning algorithm. In particular, we generate data for three different systems namely Pendulum, Acrobot, and Half Cheetah(see Figure 2).



(a) Pendulum **(b)** Acrobot **(c)** Half Cheetah

Figure 2

These three environments are chosen in increasing complexity to test the capabilities of our approach.

- The classical **Pendulum** system consists of a pendulum attached at one end to a fixed point, and the other end being free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position, with its center of gravity right above the fixed point. The state space is of dimension 3 and the action space is of dimension 1. This is a relatively simple system. The dynamics are generated for 200 timesteps.
- **Acrobot** consists of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated. The goal is to apply torques on the actuated joint to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards. Therefore, the state space is of dimension 6 and the action space is of dimension 3. The dynamics are generated for 500 timesteps.
- **Half Cheetah** is a 2-dimensional robot consisting of 9 links and 8 joints connecting them (including two paws). The goal is to apply torque on the joints to make the cheetah run forward

(right) as fast as possible, with a positive reward allocated based on the distance moved forward and a negative reward allocated for moving backward. The torso and head of the cheetah are fixed, and the torque can only be applied on the other 6 joints over the front and back thighs (connecting to the torso), shins (connecting to the thighs), and feet (connecting to the shins). Therefore the state space is of dimension 17 and the action space is of dimension 6. This is an extremely difficult system to learn the dynamics of. The dynamics are generated for 200 timesteps.

Due to computational constraints, we generate 100000 number of trajectories for each of the systems and 95% of them are used in the training set and 5% of them are used in testing.

3.3 TransDynaMo

As mentioned earlier, the architecture of TransDynaMo is inspired by Decision Transformer. Similar to Decision Transformer, we aim to utilize off-the-shelf Transformer architecture with our own modifications particular to our problem. We use GPT-2 transformer model [9] as our base transformer architecture with modifications on embedding as we add our own embeddings for that. As shown in Figure 3, we use an embedding layer to obtain tokens for states and controls of trajectories and we further add positional embeddings where we add timesteps embeddings to each of the embedded vectors. The obtained tokens are passed through a standard transformer to predict the next states of the dynamical system. We also added an encoder attention mask to prevent the transformer from having access to future states and cheating during the training process. A pseudo-code of our approach is presented in Algorithm-1

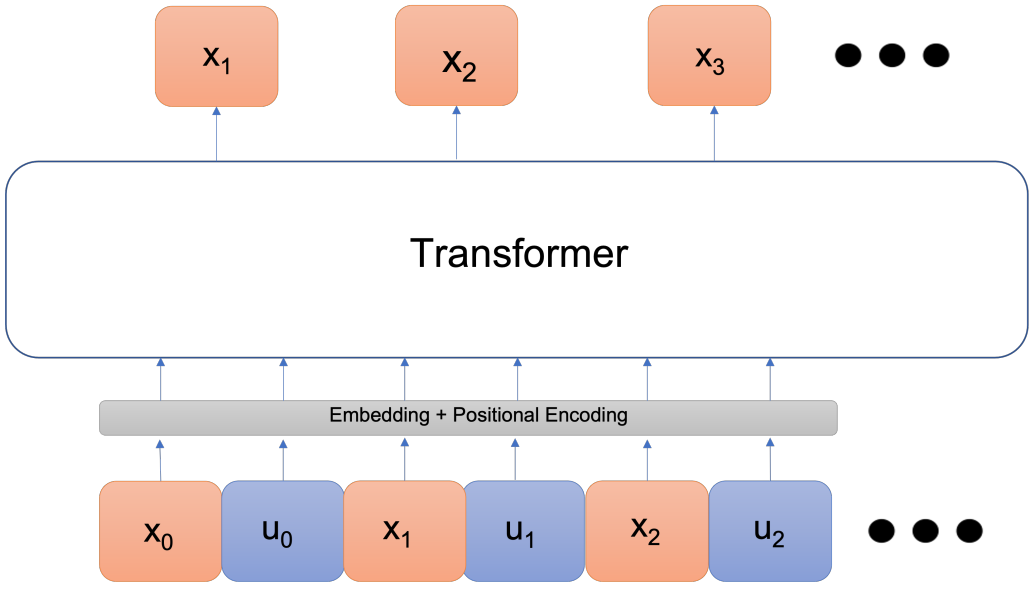


Figure 3: TransDynaMo Input/Output Model

3.3.1 Training

During each training step, given the dataset of trajectories, we sample a mini-batch of trajectories. As mentioned earlier, the trajectories are passed the TransDynaMo architecture to predict the state

trajectories of the system while using the encoder attention mask to avoid cheating by the transformer. The losses between predicted and actual trajectories are computed through either mean squared error loss or smooth L1 loss. The model parameters are updated through a backward pass.

3.3.2 Testing

During the evaluation process, a mini batch of trajectories from the offline data is sampled. For each of the trajectories, we start with the initial state and control from the trajectories. We give them as inputs to the transformer with appropriate padding and attention mask to predict the next state of the system. This predicted next state and previous state are concatenated to obtain new state array and control of next time from the sampled trajectory are concatenated to obtain the new control array. With new arrays and appropriate padding and attention mask, the next state is predicted again. This is repeated until the end of the sequence. In the end, the predicted state array is compared against the actual array to evaluate the performance of our approach.

3.4 Hyperparameters

Key hyperparameters for our experiments are mentioned below. Rest of them are set to their default values from the original GPT paper.

Hyperparameter	Value
Batch size	64
Optimizer(AdamW) initial lr	0.1
Optimizer(AdamW) weight decay	1e-4
Training epochs	600
Embedding dimension size	128
Number of attention heads	4
Number of hidden layers	3

4 Results

All experiments were performed using NVIDIA Tesla V100 GPU on GCP.

4.1 Pendulum

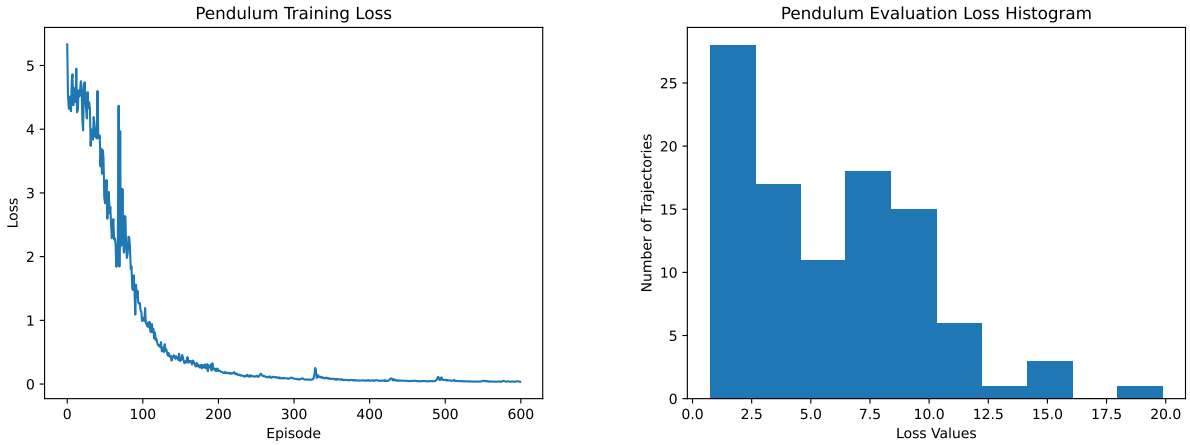
We generated a total of 100,000 trajectories, of which 95,000 were used in training and 5,000 were used for testing. During each epoch, we sample a mini-batch of size 64 trajectories from the training set. We ran training for 600 epochs.

Algorithm 1 TransDynaMo

function TransDynaMoTransformer($x, u, t, attention\_mask, encoder\_attention\_mask$)
  $pos\_embedding \leftarrow$ embed_t(t) ▷ Per-timestep positional embedding
  $x\_embedding \leftarrow$ embed_x(x) + $pos\_embedding$ ▷ State embedding
  $u\_embedding \leftarrow$ embed_u(u) + $pos\_embedding$ ▷ Action embedding
  $input\_embeds \leftarrow$ stack($x\_embedding, u\_embedding$) ▷ Concatenated tokens
  $attention\_mask$ ▷ Mask to differentiate between input sequence and padding
  $encoder\_attention\_maks$ ▷ Encoder mask to avoid access of future states during training
  $pred\_states \leftarrow$ transformer($input\_embeds, attention\_mask, encoder\_attention\_mask$)
  return $pred\_states$ ▷ State prediction
end function

procedure TrainingLoop(dataloader)
  for (x, u, t) in dataloader **do** ▷ Batched training data
    Compute appropriate $encoder\_attention\_mask$
    $attention\_maks = None$ ▷ No padding required during training
    $x\_preds \leftarrow$ TransDynaMoTransformer($x, u, t, attention\_mask, encoder\_attention\_mask$)
    $loss \leftarrow$ mean($(x\_preds - x)^2$) ▷ MSE loss
    optimizer.zero_grad()
    $loss$.backward()
    optimizer.step()
    scheduler.step()
  end for
end procedure

procedure EvaluationLoop
  (x^{traj}, u^{traj}, ts) ▷ Trajectory Sampled from Test data
  $x = x_0^{traj}, u = u_0^{traj}$ ▷ Initial States and Controls
  for t in ts **do**
    Compute $attention\_maks$ according to the appropriate padding
    $encoder\_attention\_mask = None$ ▷ No need for during evaluation
    $x\_new \leftarrow$ DecisionTransformer(x, u, t)[-1] ▷ Sample next action
    $(x, u, t) \leftarrow (x + [x\_new], u + [u^{traj}[t+1]], t)$ ▷ Append new tokens
  end for
  $loss \leftarrow$ mean($(x^{traj} - x)^2$) ▷ MSE loss of predicted traj wrt actual
end procedure



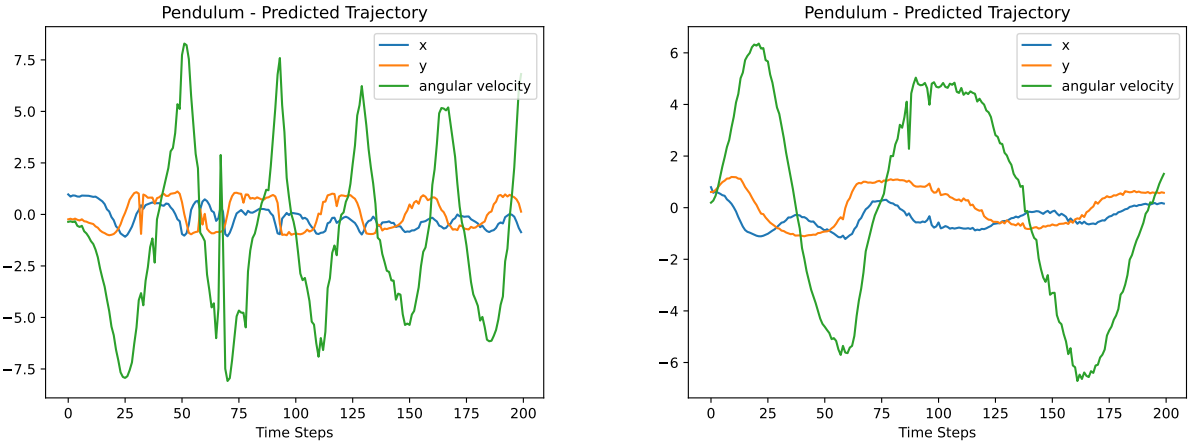
(a) Pendulum Training Loss (MSE)

(b) Pendulum Evaluation Loss Histogram

Figure 4: Pendulum Training and Evaluation plots shows that during Training the loss seems to converge to zero and during testing also about 50% of the trajectories have loss values less than 5.

Figure 4a shows the training loss for the Pendulum, which clearly seems to converge near zero. After training, we sampled 100 trajectories for evaluation from the test set, and calculated the loss for each trajectory. Figure 4b shows that majority of the trajectories had relatively low loss, although there did seem to be a few outliers.

We also realized that there exist quite a few types of losses that we could use to measure performance of our model. We chose to compare MSE loss and Smooth L1 Loss. Figure 5 shows that the predicted trajectory using MSE Loss is better (closer to the actual trajectory) than the predicted trajectory using Smooth L1 loss. Due to this observation, we decided to use only MSE loss for the remaining environments and experiments.



(a) Pendulum Predicted Trajectory - MSE Loss

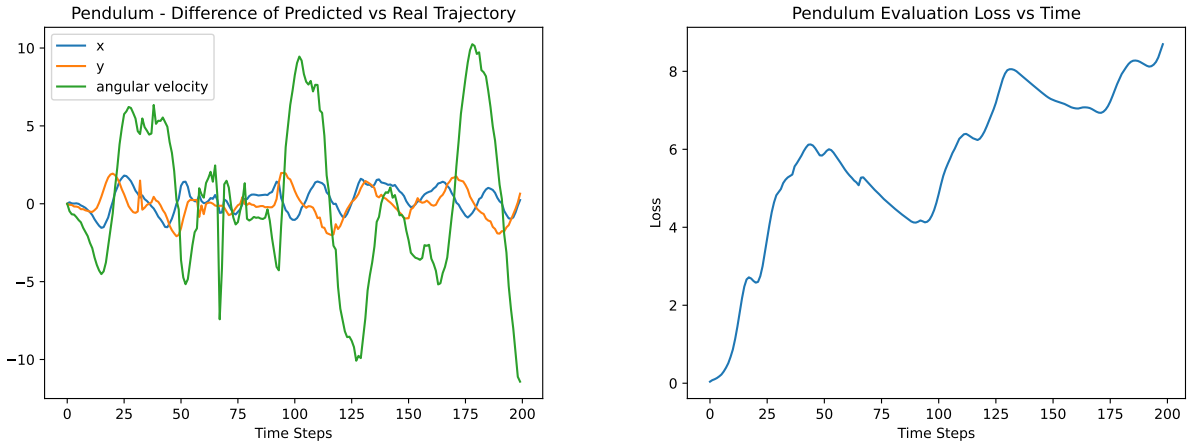
(b) Pendulum Predicted Trajectory - Smooth L1 Loss

Figure 5: The comparison of qualities of trajectories for MSE-Loss vs smooth L1-loss suggests that MSE trajectories are performing better. The actual trajectories are shown in Figure 1b

Figure 1a shows the predicted trajectory, and compares it to the Fig 1b, the ground truth (actual trajectory). To make the difference between the predicted and ground-truth trajectory easy to

visualize, we plotted their difference in Figure 6a.

Another interesting statistic we decided to consider was how much loss a predicted trajectory accumulates over time, shown in Figure 6b. Interestingly, at some points, the accumulated loss starts to decrease. However, overall the loss does not seem to be bounded.



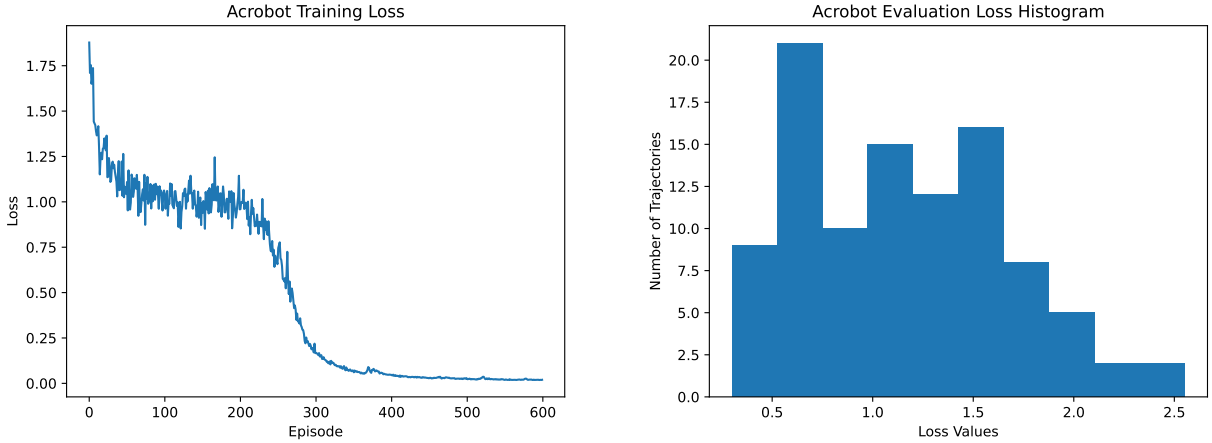
(a) Difference b/w Predicted and Ground Truth

(b) Evaluation Loss vs Time

Figure 6

4.2 Acrobot

Figure 7a shows the training loss for the Acrobot, which clearly seems to converge near zero. After training, we sampled 100 trajectories for evaluation from the test set, and calculated the loss for each trajectory. Figure 7b shows that the first 2-3 bins of the histogram do not contain majority of the trajectories, unlike 4b. This is the first indication that the transformer has not learned the Acrobot dynamics model as well as it had learned the Pendulum model.

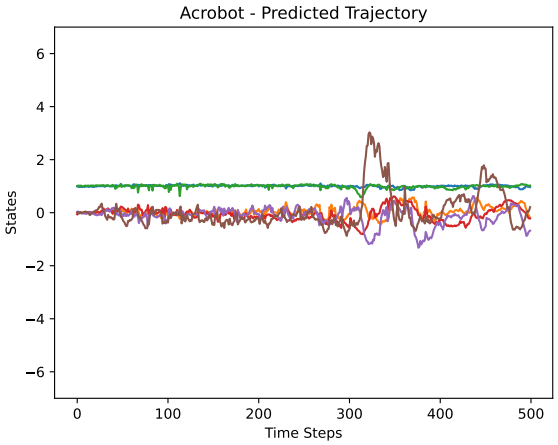


(a) Acrobot Training Loss

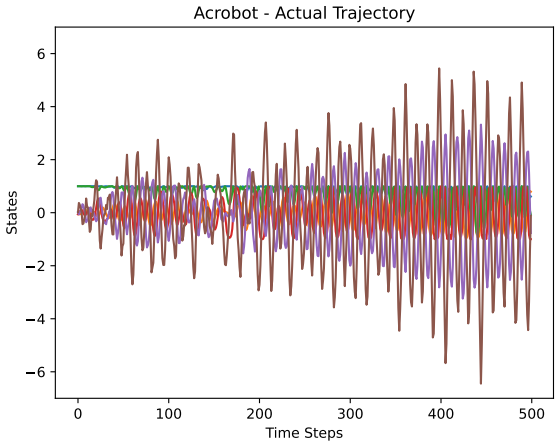
(b) Acrobot Evaluation Loss Histogram

Figure 7: Acrobot Training and Evaluation plots shows that during Training the loss seems to converge to zero and during testing also about 50% of the trajectories have loss values less than 1.75.

Figure 8a shows the predicted trajectory, and compares it to the Fig 8b, the ground truth (actual trajectory). To make the difference between the predicted and actual trajectory easy to visualize, we plotted their difference in Figure 9a. It is interesting to see that the transformer has learned the dynamics of some states very well (the state trajectory shown in green), and others not so much (for example, the purple state trajectory).

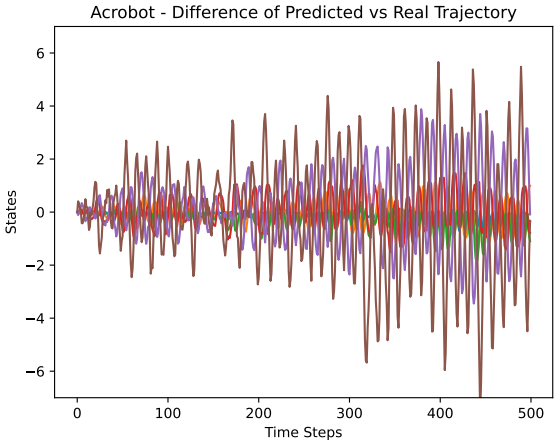


(a) Acrobot Predicted Trajectory

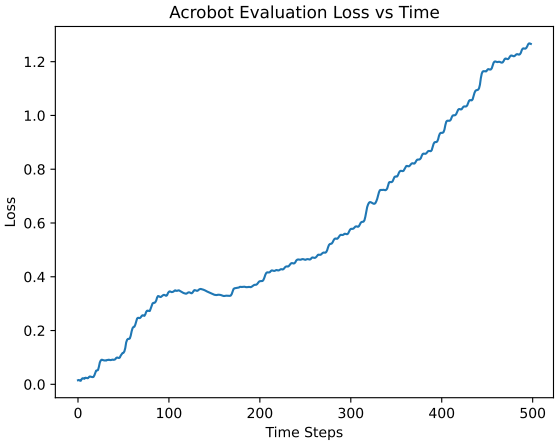


(b) Acrobot Ground-Truth

Figure 8: As we can see from the figure, the Acrobot ground truth is very chaotic for some states and TransDynaMo fails to capture those chaotic changes while it successfully captures states that are less chaotic.



(a) Difference b/w Predicted and Ground Truth



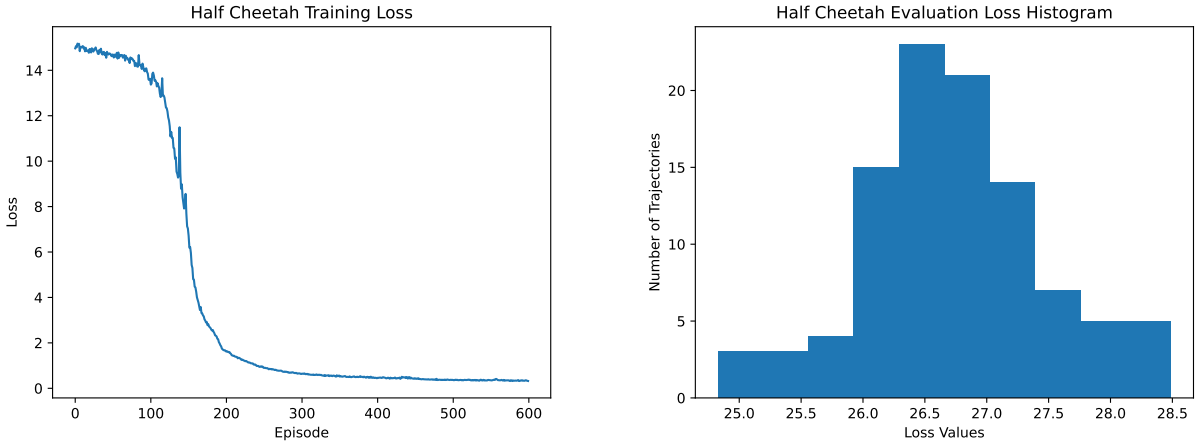
(b) Evaluation Loss vs Time

Figure 9

We notice that the Transformer has not learned the Acrobot dynamics model as well as it learned the Pendulum dynamics. This is likely due to the fact that the Acrobot dynamics are more complex than the pendulum.

4.3 Half Cheetah

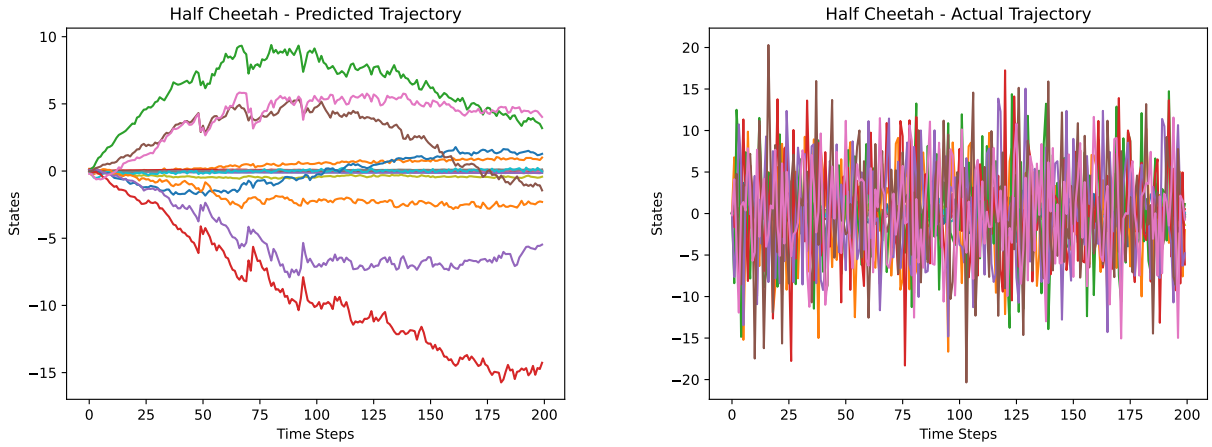
Figure 10a shows the training loss for the Half Cheetah, which clearly seems to be converging near zero. After training, we sampled 100 trajectories for evaluation from the test set, and calculated the loss for each trajectory. Figure 10b shows that majority of these 100 trajectories do not actually have low loss at all. In fact, the bins for this histogram do not start off at zero, unlike those for the Pendulum (Figure 4b) and the Acrobot (Figure 7b). Figure 11 and Figure 12 clearly demonstrate that the Transformer was not able to learn a meaningful model for the Half Cheetah dynamics. This is expected, as the Half Cheetah is a more complex model than Pendulum and Acrobot.



(a) Half Cheetah Training

(b) Evaluation Loss Histogram

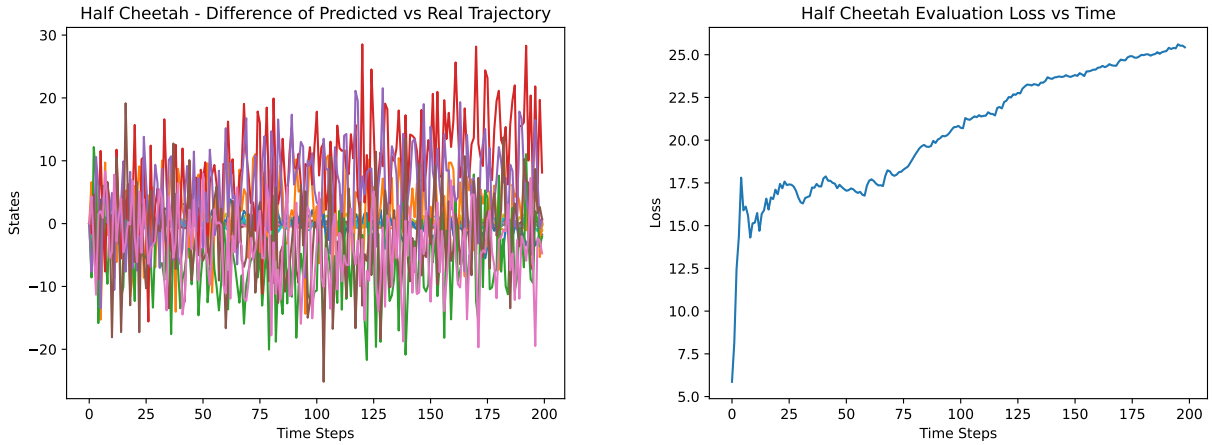
Figure 10: Half Cheetah Training and Evaluation plots shows that during Training the loss seems to converge to zero however during testing the loss values are very high due to higher complexity in dynamics and randomness states from one time step to another



(a) Half Cheetah Predicted Trajectory

(b) Half Cheetah Ground-Truth

Figure 11: As we can see, the predicted trajectory is very off from the ground truth because of the aforementioned reasons



(a) Difference b/w Predicted and Ground Truth

(b) Evaluation Loss vs Time

Figure 12

5 Conclusion and Future Directions

In conclusion, our project demonstrates the effectiveness of utilizing Transformer architecture for learning dynamic models of systems through sequence-to-sequence modeling. By treating the system's trajectory as an input sequence and generating the next states as the output sequence, we have successfully trained a Transformer to predict the future states of the system based on its initial state and a sequence of actions. The comparison between real and predicted trajectories reveals a significant resemblance, indicating that the Transformer has effectively captured the dynamics of the system. These promising results highlight the suitability of Transformers for modeling dynamic systems in the context of sequence-to-sequence modeling tasks.

However, as we observed, the transformer fails to capture high dimensional and system dynamics. We suspect the following reasons behind this.

- Insufficient training data for complex systems: Due to computational constraints, the amount of training data available might not have been enough, especially for high-dimensional and complex systems like Acrobot or Half Cheetah. More data would have been beneficial for improving the performance of the Transformer model.
- Potential for improved results with increased model complexity: Given access to more computational power, increasing the complexity of the Transformer model could have potentially yielded better results. However, this was not possible within the existing constraints.
- Limited modeling of temporal dependencies: Transformers, while proficient at capturing patterns within a sequence, do not explicitly model the temporal dependencies inherent in dynamical systems. As a result, accurately capturing the evolution of states over time can be challenging for Transformers.
- Fixed-length context windows: Transformers typically operate on fixed-length context windows, which restricts their ability to capture long-term dependencies and fully understand the complete history of a system's dynamics. This limitation can impact their performance in modeling dynamical systems accurately.

- Additionally, it is worth noting that employing standard data pre-processing techniques, as discussed in [6], could have potentially improved the results obtained from using Transformers in this context.

In the future, our aim is to explore more complex models and leverage improved pre-processed data to delve deeper into this problem and potentially achieve superior results.

6 Statement of Individual Contributions

- Maulik contributed towards ideation of how to think of learning dynamics models as a sequence to sequence tasks. He also contributed towards data generation from Mujoco-Gym environments, understanding and developing the transformer architecture, running experiments, and preparing the report.
- Amogh also contributed towards the ideation of how to think of learning dynamics models as a sequence to sequence tasks. He also contributed to the data generation, modifying and troubleshooting the TransDynaMo Architecture, running experiments, and preparing the report.

References

[1] Mattia Cenedese, Joar Axås, Bastian Bäuerlein, Kerstin Avila, and George Haller. Data-driven modeling and prediction of non-linearizable dynamics via spectral submanifolds. *Nature Communications*, 13(1), feb 2022.

[2] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

[3] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888, 2018.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[5] Nicholas Geneva and Nicholas Zabaras. Transformers for modeling physical systems. *Neural Networks*, 146:272–289, 2022.

[6] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[7] OpenAI. Gymnasium: A collection of reinforcement learning environments for openai gym. https://github.com/openai/gymnasium, 2021.

[8] Joshua L. Proctor, Steven L. Brunton, and J. Nathan Kutz. Dynamic mode decomposition with control, 2014.

[9] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[10] Anna Shalova and Ivan Oseledets. Tensorized transformer for dynamical systems modeling, 2020.

[11] Haojie Shi and Max Q. H. Meng. Deep koopman operator with control for nonlinear systems, 2022.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.