ROBOTIC BIOMETRIC MOTION IDENTIFICATION: LEARNING PROPRIOCEPTION-ENABLED MORPHOLOGICAL REPRESENTATION FOR ROBOT META-SELF-MODELING

Yunzhe Wang
School of Engineering and Applied Science
Columbia University
New York, NY 10027, USA
yw3737@columbia.edu

ABSTRACT

Proprioception refers to the sense of self-movement captured by intrinsic sensors in limbs and joints, allowing humans to create mental imagery of their body parts without external stimuli. Such self-image has shown to be obtainable in robots using supervised computational *self-models* that predict body morphology from kinematic data, enabling generalized planning. Recent progress in self-modeling has targeted a more general schema – *meta-self-modeling* – by learning over the distribution of an entire robot family of similar structures. However, decoupling morphological features for individual robots within the family remains challenging. This paper presents a multiclass-multioutput robot morphology classifier that predicts a high-dimensional integer vector configuration code from motion dynamics to identify robots. Unlike prior studies that use a single robotic arm with relatively low complexity, the classifier generalizes over the distribution of a 12-DoF quadruped robot family with leg position and orientation variations. We discovered that prediction accuracy is best acquired when incorporating both the temporal and spatial dependency of collected motion data, where task-agnostic motor babbling action is sufficient for differentiation. In our experiment, 26.8% of the robot's configuration code can be fully predicted within 20 babbling cycles, while 81.3% achieves l_1 errors ≤ 3. Our result suggests a strong decodability from robot motor dynamics to morphology. We anticipate the learning procedure to be reusable beyond this well-defined robot family, and decoupling individual robots could serve as a critical aspect in future generalist robotic research.

1 INTRODUCTION

The ability to create a long-term body self-image through movement is natural for humans (Proske & Gandevia, 2012; O'Shaughnessy, 1995): we can perceive body (parts) positions and movements without seeing and thinking, thanks to proprioceptors located within muscles, tendons, and joints (Jankowska, 1992). Such spatial self-awareness is essential for robots to anticipate outcomes of motor action without trying them out in physical reality, where past research trying to replicate such capability through computational models have named the paradigm *robot self-modeling* (Kwiatkowski et al., 2022). It has been shown that a consistent self-model, once learned, allows robots to plan actions, avoid obstacles, optimize trajectories, and recover from real-world damage, showing improved performance and machine resiliency (Bongard et al., 2006; Kwiatkowski & Lipson, 2019; Chen et al., 2022; Hu et al., 2022).

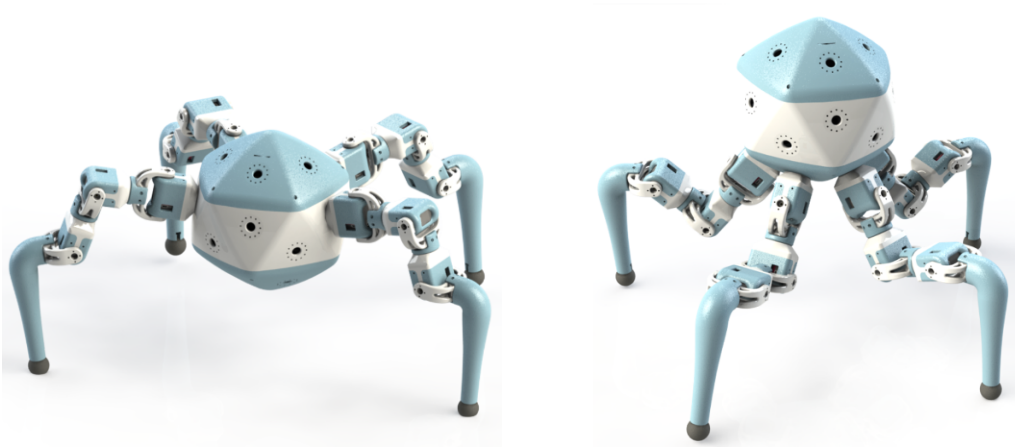
Recently, the concept has been extended to a more generalized perspective, *meta-self-modeling*, where the self-images of an entire robot family can be captured by a single model. The robot family here refers to a group of robots sharing similar mechanical structures and control systems, and individual robots can be uniquely represented via a signature code. Nonetheless, differentiating robots belonging to the same family without knowing their ground truth morphology remains a key

challenge. Such ability, once acquired, may enable generalized control over the distribution of the entire family.

In this work, we proposed a learning framework to identify individual robots from a well-defined quadruped robot family described in section 1.1. By predicting their configuration code through intrinsic kinematic data relating to body position, orientation, and joint angles, our learned neural representation demonstrate conditioning power for self-modeling robot with unseen configuration. The process is analogous to Biometric Motion Identification, where individual humans can be identified through motion captures in kinetic videos and 3D skeletal sequences (Boulgouris et al., 2005; Munsell et al., 2012; Han et al., 2017).

1.1 Isocahedrone Quadruped Robot Family and Its Configuration Code

To study a robot family with similar configurations, we designed a 12-DoF quadruped robot with URDF descriptions to be loaded in physics simulation engines and assembled in the real world. Its legs can be attached to any four faces of an icosahedron body, and each leg consists of three links where the connection point between links can be rotated to 12 different angles with 30 degrees separation like a clock. This gives a family of robots with a total number of $C_4^{20} \cdot (4 \cdot 3)^{12} \approx 4.32 \times 10^{16}$ possible configurations depending on how we assemble the robot. To uniquely describe each robot, we designed an integer vector coding $\mathbf{y} \in \mathbb{Z}^{12}$ such that $0 \leq y_f \leq 19$ for $f \in \{0, 4, 8, 12\}$, indicating the four faces with legs, and that $0 \leq y_l \leq 11$ for $l \in \{1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15\}$, indicating the angles of the twelves links. See examples and explanation in figure 1. We implemented a script to generate the corresponding URDF file given a configuration for simulation.



[10, 9, 9, 6; 11, 9, 9, 6; 13, 3, 3, 6; 14, 3, 3, 6]
L1 L2 L3 L4

[16, 9, 2, 1; 15, 9, 2, 4; 18, 3, 10; 8, 17, 3, 10, 11]

Figure 1: Examples of two robots as described by their configuration coding. Each code is separated into four legs by semicolons. Each group's first value (orange) indicates the face index (20 choices) to which the leg is attached. The three following numbers indicate the connected joints' rotation angle (12 choices). For instance, the code on the left indicates that the robot's four legs are attached to faces 10, 11, 13, and 14. The three joints of the leg in face 10 have connection angles of degrees 9×30, 9×30, and 6×30. Both robots are symmetric: the joint angles for mirrored legs sum up to a full circle $L1 + L4 = L2 + L3 = [12, 12, 12]$

2 Methods

To model the mapping from robot motion dynamics to their respective morphology, we developed a neural network classifier:

1. We randomly generated 250K different robots that are symmetric and can stand.

2. For each robot, we collected the state sequences for eight trials; each contains six steps of robot motor babbles.
3. We trained a multiclass-multioutput classifier with seven prediction heads (one for leg position and six for joint orientation) that predict the robot's structural coding given a sampled state sequence of varying lengths.
4. Once trained, the model can predict robots with unseen configurations by observing their motion dynamics on the fly.

2.1 Data Collection

Our data collection schema consists of two phases: 1) robot generation and 2) motor babbling state collection. We randomly generated 250K robots with different configuration coding, where each code maps to unique URDF files describing the robot's morphology. All generated robots are symmetric and can stand when loaded into the simulation. For each robot, we collected its kinematic state (position, orientation, joint angles) sequences while performing random motor babbling actions defined by the parametric sine-gait function in equation 1.

Algorithm 1: Robot Generation

Data: Left faces indices I_f, Number of robots N, Configuration mirror function m
Result: Robot Dataset $\mathcal{D}_r$ as a mapping from config code to URDF

$\mathcal{D}_r \leftarrow \emptyset$;
while $|\mathcal{D}_r| < N$ **do**
  $C_J \leftarrow$ `RandInt(`*range=12, n=6*`)`;
  $C_L \leftarrow \{C_L \subseteq I_f \mid card(C_L) = 2\}$;
  $C \leftarrow [C_L; m(C_L); C_J; m(C_J)]$;
  $R \leftarrow$ `URDF(`C`)`;
  if `collide(`R`)` *or* `slip(`R`)` **then**
    continue
  else
    $\mathcal{D}_r \leftarrow \mathcal{D}_r \cup \{C : R\}$;
  end
end

Algorithm 2: Kinematic Data Collection

Data: Robot Dataset $\mathcal{D}_r$, # of steps N, # of trials T, Sine-gait action function a.
Result: Kinematic Dataset $\mathcal{D}_k$

foreach $\{C : R\} \in \mathcal{D}_r$ **do**
  $\mathcal{D}_k^{(C)} \leftarrow \emptyset$;
  while $|\mathcal{D}_k^{(C)}| < T$ **do**
    $\mathcal{D}_t \leftarrow \emptyset$;
    for $n = \{1, ..., N\}$ **do**
      $\theta \leftarrow$ `Rand(`*n=10*`)`;
      R`.next_step(`$a(\theta)$`)`;
      if `slip(`R`)` **then**
        break
      end
      $\mathcal{D}_t \leftarrow \mathcal{D}_t \cup \{$`state(`$R$`)`$\}$;
    end
    $\mathcal{D}_k^{(C)} \leftarrow \mathcal{D}_k^{(C)} \cup \mathcal{D}_t$;
  end
end

Robot Generation We randomly generated 260K robots for the icosahedron quadruped family (section 1.1) dataset with different configuration codes as described in algorithm 1. For simplicity, we constrained the robot structure in two ways: 1) only 12 out of the 20 faces will be chosen to attach legs, where all of them locates in the middle and bottom layers of the icosahedron; 2) All robots are symmetric, meaning that given half of the configuration code (8 values) on one side of the robot, we are guaranteed to determine the other half through a mirror function m. This simplification drastically reduces the total number of possible configurations to $2 \cdot C_2^6 \cdot (2 \cdot 3)^{12} \approx 8.96 \times 10^7$.

We generated the corresponding URDF description file for each configuration coding and loaded them in the PyBullet Physics Engine for validation. We filtered out robots with self-collision and those that would slip over when loaded into the simulation. Slipping over is triggered when the robot's body roll or pitch value is greater than $\pi/2$ while it moves. After the selection process, we are left with 255K valid robot configurations.

Motor Babbling Kinematic Data Collection For each valid robot configuration, we collected their state sequence for eight trials, where each trial contains eight steps of motor babbling cycles. During random movement, the robot might also fall over before finishing six steps. If that happens, the trial would be aborted and rerun as shown in algorithm 2.

The robot action gait is defined by a parameterized sine function as shown in equation 1, where A is the amplitude parameter, and ϕ is the phase-shift parameter. i indexes the three joints of the same leg (ordered as inner, middle, and outer joints), and j indexes the four legs (ordered as right-hind, right-front, left-front, and left-hind legs). t is the current timestep, and $\tau = 16$ is a predetermined period constant indicating the number of sub-steps within a cycle. The action a_{ij} indicates the targeted angle of the joint i in leg j at timestep t. During motor babbling, A and ϕ are chosen uniformly at random with values normalized to the motor action space. Meanwhile, robot state sequences are collected at each timestep.

$$a_{ij} = A_i \cdot \sin\left(\frac{t \bmod \tau}{\tau} \cdot 2\pi + \phi_j\right) + x_{ij} \tag{1}$$

At timestep t, the robot's state $S_t \in \mathbb{R}^{18}$, consists of the position and Euler-angles of the icosahedron body at the center of mass $[x, y, z, \psi, \theta, \phi]$ as well as the angles of the 12 joints. The kinematic state data over an entire babbling cycle can thus be seen as a multinomial time series of 18 channels with sequence length $\tau = 16$.

2.2 Neural Network Architecture and Training

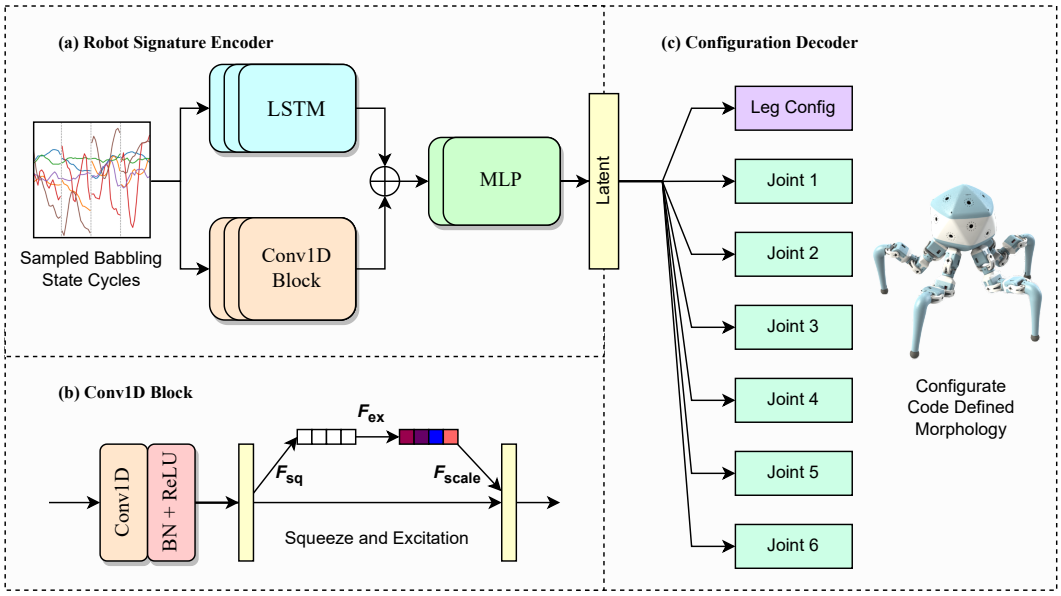


Figure 2: The model architecture of the classifier. The robot signature encoder (a) employs a Multinomial LSTM-FCN backbone, where sampled state sequences represented as piecewise multinomial time series are processed by a 3-layer LSTM network for temporal dependency and three 1D convolution blocks for channel-wise dependency. Each convolution block (b) employed the squeeze and excitation operation. The time and spatial features are then concatenated and encoded through a 2-layer MLP network into a latent vector. Lastly, the latent vector was decoded by seven prediction heads for leg positions and six joints on one side of the robot, where each one is a single fully-connected layer. The predicted indices for leg positions and joint angles are selected by taking $\arg\max$ over each head's output.

At its core, the model consists of two components: 1) a robot signature encoder and 2) a configuration decoder. The encoder handles both channel-wise and temporal dependencies of the collected state sequences, extracting a latent robot morphological representation. The decoder has seven classification heads; each is a single fully-connected layer and decodes the latent representation to the leg-face pattern and joint orientations.

Robot Signature Encoder Given the collected state-sequences of a robot, we aim to predict its configuration code to differentiate individual robots by observing its movement. We name this pro-

cess "robot signature encoding," similar to biometric motion identification, where a person can be recognized based on his/her motion capture sequence. Since the nature of the task is multinomial time series classification, we developed upon a baseline network, Multivariate Long Short Term Memory Fully Convolutional Network (MLSTM-FCN)(Karim et al., 2019), for our signature encoder Φ_{ENC} as shown in figure 2 (a). We also tried other encoder architectures relying on different data assumptions. The MLSTM-FCN Net consists of Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) layers for handling temporal dependency and 1D convolution layers with squeeze-and-excitation blocks (Hu et al., 2018) for capturing channel-wise dependencies as illustrated in figure 2 (b). The encoded features are concatenated and fed through Multi-Layer Perceptron (MLP) layers as a latent vector $\mathbf{z} = \Phi_{\text{ENC}}(\mathbf{x}) \in \mathbb{R}^d$ before decoding the configuration code.

Configuration Decoder To predict the configuration code vector $\mathbf{y}$ as defined in section 1.1 and figure 1, we leveraged a multi-output decoder Φ_{DEC} with seven prediction head as seen in figure 2 (c) that decodes $\mathbf{y}$ from the latent space $\mathbf{z}$. One head classifies the leg position pattern with 30 choices, and the other six heads classify the orientation pattern for the six legs on one side of the robot; each has 12 choices. Due to the symmetry constraint, the other half of the robot legs can be completely determined. Therefore, a 7-heads classifier is sufficient for our setup:

$$\Phi_{\text{DEC}} := (\Phi_{\text{LEG}}, \Phi^1_{\text{JNT}}, \Phi^2_{\text{JNT}}, \Phi^3_{\text{JNT}}, \Phi^4_{\text{JNT}}, \Phi^5_{\text{JNT}}, \Phi^6_{\text{JNT}})$$

Objective Function We applied the Categorical Cross-Entroy loss function for each classification head.

$$\mathcal{L}_{head} = -y \log \frac{\exp(\hat{y})}{\sum_{c=1}^{C} \exp(\hat{y}_i)} \tag{2}$$

Where $\hat{y}$ is the network output logits, y is the ground truth label, and C is the number of classes, where $C = 30$ if `head=leg` and $C = 12$ if `head=joint`. The total loss across all seven heads is aggregated by taking a weighted sum controlled by a ratio hyper-parameter λ due to the varying difficulties between classifying legs and joints.

$$\mathcal{L}_{total} = \lambda \mathcal{L}_{leg} + (1 - \lambda) \frac{1}{6} \sum_{i=1}^{6} \mathcal{L}^{(i)}_{joint} \tag{3}$$

Input Data Sampling For generalization, we adopted a sampling approach for the input state sequences during training to augment the data. For a particular robot, rather than taking all collected steps over one trial as input, we sampled without replacement non-consecutive steps over all trials of lengths 1 to 20. Since the input sequence has varying lengths, the robot can predict its morphology starting from the first move, slowly becoming more stable and accurate as it moves. The process is similar to acquiring and maintaining a self-image over time. For inference and evaluation, we input the state sequence over the entire trial rather than sampling since the model is intended to be used in real-time.

3 Experiment Setup

We followed the data-collection procedure as described in section 2.1 and collected 175K trials of robot walking data for 25K different robot configurations. Each trial contains a robot walking for six steps without slipping over, and each step is an 18-channel time series with 16 consecutive states. For each robot configuration training instance, we sampled 1-20 steps of state sequences as input $\mathbf{x}$, forward through the model to get the predicted logits $\hat{\mathbf{y}}$ and updated the gradient based on the configuration label $\mathbf{y}$ and the loss function $\mathcal{L}_{total}$ with a ratio hyper-parameter $\lambda = 0.7$ determined empirically. We split the data with an 8:2 ratio for training and validation.

Our model parameters are optimized using the PyTorch (Paszke et al., 2019) deep-learning framework and adaptively tuned the model hyperparameter using the HYPERBAND (Li et al., 2017)

	Leg	Jnt1	Jnt4	Jnt2	Jnt5	Jnt3	Jnt6
Acc.	0.936	0.747	0.752	0.526	0.529	0.407	0.414
Std.	0.008	0.005	0.012	0.025	0.025	0.026	0.022

Table 1: Prediction accuracy for individual heads. Jnt1 and Jnt4 are inner joints, Jnt2 and Jnt5 are middle joints, Jnt3 and Jnt6 are outer joints. We can observe a increasing difficulty for joints locate farther away from the body.

algorithm through the Weight-and-Bias (Biewald, 2020) package. We employed the Adam Optimizer (Kingma & Ba, 2014) with learning rate 3e-4 and weight decay 1e-5. We used the ReLU activation function, Batch Normalization (Ioffe & Szegedy, 2015), and a Dropout (Srivastava et al., 2014) probability of 0.3 for all convolution layers and the first MLP layer. We also applied gradient clipping with value 1 for the LSTM module to avoid gradient exploding. The training was done over one Nvidia RTX2080 GPU for 800 epochs with batch size 32, taking 38 hours in total.

We conducted several experiments regarding the input sequence length, data representations, and model architectures in their ability to achieve the best prediction accuracy, and analyzed our results quantitatively in section 4.1.

4 Main Results

Evaluation Metrics We monitored the following five metrics during training and evaluation. Even though we are ultimately interested in the classification accuracy of the entire configuration code (Tot-Acc), the model is intended to let the robot gradually become aware of its morphology and maintain the prediction. Since the input is a state sequence of varying step numbers, and we cannot expect the robot to be accurate at all steps, especially at the beginning when the sequence is too short, the best model is selected based on the leg prediction accuracy and the joints accuracy averaged over all time steps. During the evaluation, we used a l_1 distance error function between the prediction and the ground truth for a more intuitive view.

- **Leg-Acc:** Prediction accuracy for the leg position configuration.
- **Jnt[1-6]-Acc:** Prediction accuracy for the joint angle configuration for joint 1 to 6.
- **Jnt-Acc-Avg:** Average prediction accuracy over 6 joints.
- **Tot-Acc:** Prediction accuracy over legs and all 6 joints. A correct prediction means the model correctly predicts the leg positions and the angles for all 6 joint.
- **Err-Dist:** The l_1 distance between the predicted and real configurations integer vector.

4.1 Quantitative Analysis

We generated 50K new robots and collected their kinematic sequence data over 20 consecutive motor babbling steps without slipping over. We input the cumulative state sequence at each step and predict the configuration coding. The change of l_1 distance error distribution is shown in figure 3 (left), where we observe a strong correlation between the prediction accuracy and the step length: the more the robot babbles, the more accurate the prediction. We believe this is because a longer recorded state sequence captures more information about the robot's morphology for which the robot can be identified. Figure 3 (right) shows the distribution of the lowest error that each robot obtained over the 20 steps. Over 20 babbling steps, we can see that 26.8% of the robots can be fully identified, 30.5% of the robots have missed only one digit in l_1 distance, 17.1% have error 2, and 7.6% have error 3, summing up to 82.0%. Even though more steps generally mean less error, for all robots that have been fully identified, we observed that the average number of steps needed is 10.8

Accuracy for individual head We evaluated the accuracy for individual prediction heads and observed a spatial correlation in the prediction difficulty as shown in table 1. The leg position configuration always obtains the highest accuracy. For the six joints, the inner joints (Jnt1 and Jnt4) are relatively more manageable than the middle joints (Jnt2 and Jnt5), which is then easier than the outer joints (Jnt3 and Jnt6).

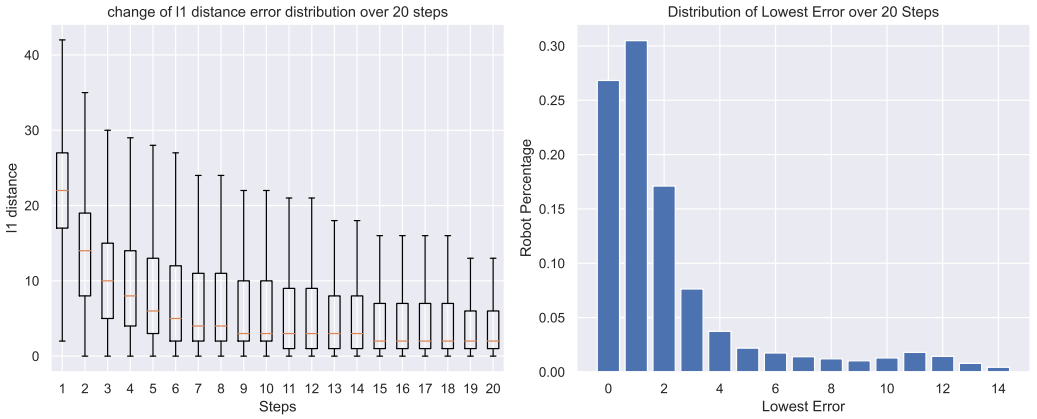


Figure 3: Left: Box plot distribution of the testing l_1 distance error of predicted configuration code over 20 steps, with a sample size of 50K unseen robots. Outliers greater than 1.5 IQR are hidden from the plot. Generally, the further the robot moves, the more accurate the prediction is. Right: the distribution of the lowest prediction error achieved by each robot within 20 steps. An error of 0 means a fully correct prediction, for which there are 26.8% of such robots. The long distribution tail for error greater than 15 is not shown.

Signature Encoder Ablation Study We retrained and reevaluated our model by taking off parts of the signature encoder. We obtained a 0.468 total accuracy by removing the Conv1D module, a 0.594 total accuracy by removing the LSTM module, and a 0.617 total accuracy with both components kept. This result indicates that LSTM and Conv1D are both needed to achieve the best prediction, capturing temporal and spatial dependencies. Nonetheless, the Conv1D blocks' contribution is more significant than that of LSTM in our problem formation.

5 Conclusion

We have developed a neural network classifier that predicts the configuration coding for an icosahedron quadruped robot family given motor babbling kinematic data sequences. We experimented with architecture incorporating both the temporal and/or spatial dependencies replying to different data assumptions. Our results suggest a strong relationship between motor dynamics and morphology, and we anticipate the learning paradigm to be applicable in more generalized scenarios for other robot families.

6 Discussion

Similar to biometric motion identification, where individual humans can be recognized, the model is an identification tool to distinguish individual robots. Such robot identity information can be applied to improve downstream controlling tasks. For instance, its most prominent application is for robot predictive control through robot self-modeling, which aims to optimize robot action trajectories on the fly in unseen scenarios. Having the ability to distinguish individual robots, it is possible for a single self-model to generalize over the distribution of an entire robot family. To achieve such an effect, we anticipate that the dense latent vector from the last encoder layer (figure 2) of a close prediction, not necessarily correct, is sufficient. Nonetheless, more experiments need to be done regarding this question as future work.

References

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

Josh Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121, 2006.

Nikolaos V Boulgouris, Dimitrios Hatzinakos, and Konstantinos N Plataniotis. Gait recognition: a challenging signal processing technology for biometric identification. *IEEE signal processing magazine*, 22(6):78–90, 2005.

Boyuan Chen, Robert Kwiatkowski, Carl Vondrick, and Hod Lipson. Fully body visual self-modeling of robot morphologies. *Science Robotics*, 7(68):eabn1944, 2022.

Fei Han, Brian Reily, William Hoff, and Hao Zhang. Space-time representation of people based on 3d skeletal data: A review. *Computer Vision and Image Understanding*, 158:85–105, 2017.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141, 2018.

Yuhang Hu, Boyuan Chen, and Hod Lipson. Egocentric visual self-modeling for legged robot locomotion. *arXiv preprint arXiv:2207.03386*, 2022.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

E Jankowska. Interneuronal relay in spinal pathways from proprioceptors. *Progress in neurobiology*, 38(4):335–378, 1992.

Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Samuel Harford. Multivariate lstm-fcns for time series classification. *Neural Networks*, 116:237–245, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Robert Kwiatkowski and Hod Lipson. Task-agnostic self-modeling machines. *Science Robotics*, 4 (26):eaau9354, 2019.

Robert Kwiatkowski, Yuhang Hu, Boyuan Chen, and Hod Lipson. On the origins of self-modeling. *arXiv preprint arXiv:2209.02010*, 2022.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

Brent C Munsell, Andrew Temlyakov, Chengzheng Qu, and Song Wang. Person identification using full-body motion and anthropometric biometrics from kinect videos. In *European Conference on Computer Vision*, pp. 91–100. Springer, 2012.

Brian O'Shaughnessy. Proprioception and the body image. *The body and the self*, pp. 175–203, 1995.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Uwe Proske and Simon C Gandevia. The proprioceptive senses: their roles in signaling body shape, body position and movement, and muscle force. *Physiological reviews*, 2012.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

LEGGED ROBOT META SELF-MODEL
2022 SUMMER RESEARCH REPORT

Ruibo Liu, Zhou Shen, Jiong Lin, Yunzhe Wang
School of Engineering and Applied Science
Columbia University
New York, NY 10027, USA
{rl3233,zs2489,jl6017,yw3737}@columbia.edu

ABSTRACT

Self-model gives robots the ability to foresee itself into the future. In locomotive robot, this can mean the ability to predict motor and kinematics states at future timesteps. Previous work have demonstrated the power of self-model in manipulative robot arms with relatively low degree of freedom, but few experiment was done on locomotive legged-robot with complex morphology. In this work, we present the meta-self-model: a single self-model that captures the motion dynamics of an under-actuated legged robot with 12 degrees of freedom with $C_4^{20} \cdot 12^{12}$ possible number of configurations. At the current stage, we simplify the state space by optimizing forward sine-gate motion of 1000 robot configurations through hill-climbing, and build the meta-self-model upon data collected with the optimized baseline behavior. Our experiment shows that, the meta-self-model could already improve the forward task and zero-shot learn to backward and turning tasks via planning in noised action spaces. We identified several modifications and directions that could be made regarding data processing, pipeline design, and deployment for improved performance.

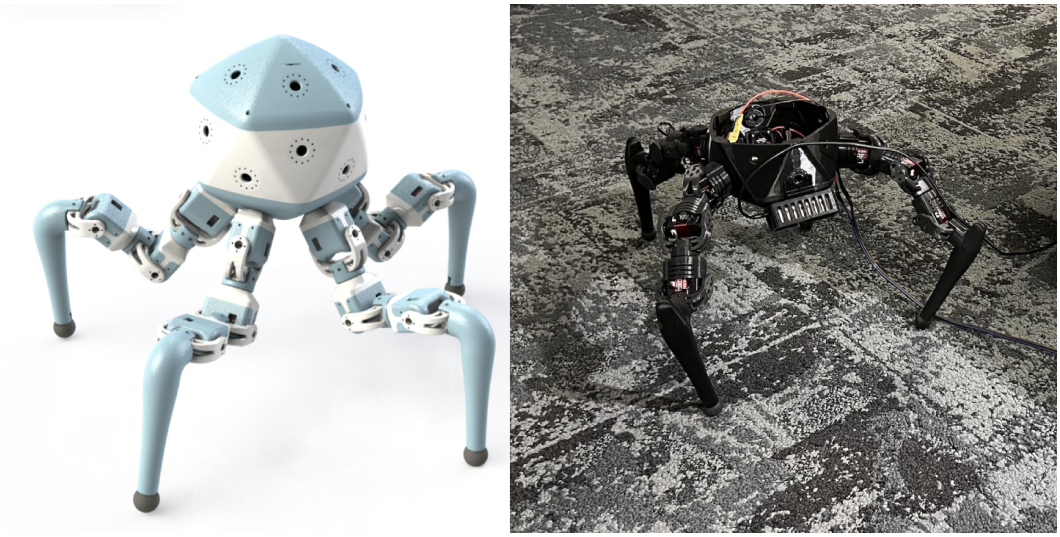


Figure 1: The customizable icosahedrone four legged-robot that we are going to build meta self-model upon. It can be modeled in both physics simulation environment (left) and in real world (right)

CONTENTS

1 INTRODUCTION

1.1 CUSTOMIZABLE ICOSAHEDRON LEGGED ROBOT

We made a icosahedron legged robot that has 20 faces with four legs that can run in simulation by loading URDF in PyBullet engine and in real world (figure 1). The four legs can be plugged into any 20 choose 4 faces. Also, each leg has three links where the connection point can be rotated in 12 different angles with 15 degrees each. Therefore, the total number of possible configurations is $C_4^{20} \cdot 12^{12}$. To simplify the problem, we randomly generated 1000 different configurations and optimized all of them such that they can perform some preliminary forward sine-gate motion.

1.2 ROBOT SELF-MODELING

We humans acquires the ability to model ourselves as infant: through playing around in the world we lived in, we acquire senses of our body capabilities. Before performing real-world tasks, with varying cognitive complexity from jumping over a gap to preparing a presentation, we play simulations in our mind imagining what could happen in all sorts of scenarios, accessing risks and gains as if we have done the actions. We call the model that allow robot to predict itself in a future state a "self-model". In its simplest form, a self-model is a regression model that predicts values in future time steps given past information and can be implemented using feed-forward neural networks. The process of building the self-model requires robot performing various (task-agnostic) actions in exploring its physical constraints and motion dynamics. Recent researches shows that through task-agnostic neural self-model, robot can plan motion, reach goals, avoid obstacles, and recover from real-world damage, leading to improved machine resiliency. Bongard et al. (2006); Kwiatkowski & Lipson (2019); Chen et al. (2022). Previous work (unpublished) also demonstrated that the value of a self-model increases as the complexity of the systems increases and the self-model doesn't have to be accurate to make improvements but cannot create a deflated self-image.

1.3 GOALS AND MOTIVATION

Given the customizable robot configurations and high-degrees of freedom, we want to build a single model that predicts the robot's future states to all its configurations thereby having full motion dynamics knowledge to make planning in all sorts of locomotive tasks. The ultimate goal of this project is two-folds:

- **Predict Goal:** Build a model that have sufficient knowledge about robot state-action space for all different configurations. This allow us to do the following things: Given an unseen robot configuration, let it perform some predefined actions (sampling a surface from its whole state-action space), it will have knowledge to what itself is (self-representation/self-model). Then, we can build a predictive model given an observed state and proposed action, conditioned on the robot's self-representation (what the robot is), to predict the its next state. Abstractly, the model can predict the future given past experience given any robot.
- **Control Goal:** Given a working predict model. After a robot build its self-representation through some predefined action, given the state it's currently in, by sampling some promising actions and predict the next states as if the robot perform those actions, and given a reward metric based on the change of states, we can search a optimal action trajectory which maximizes reward. The reward metric defines the type of the action. (e.g. state

change in x-coordinate represents forward movement). This model could possible allow us to use a single model to control all robot configurations in performing all sorts of actions.

2 Robot Configuration

2.1 URDF generating script

The first step to exploring different configurations of legged robots is making a script to generate all possible configurations automatically. The body of our robot is an icosahedron, and using a rotation matrix with a given icosahedron dimension is able to calculate the geometrical relation, such as radius, roll, pitch, and yaw, to transfer the leg component from one face to another faces.

$$
\begin{aligned}
Rotation &= R_z(\alpha)R_y(\beta)R_x(\gamma) \\
&= \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\gamma & -sin\gamma \\ 0 & sin\gamma & cos\gamma \end{bmatrix} \\
&= \begin{bmatrix} cos\alpha cos\beta & sin\alpha sin\beta cos\gamma - sin\alpha cos\gamma & cos\alpha sin\beta cos\gamma + sin\alpha sin\gamma \\ cos\beta sin\gamma & sin\alpha sin\beta cos\gamma + cos\alpha cos\gamma & cos\alpha sin\beta sin\gamma - sin\alpha cos\gamma \\ -sin\beta & sin\alpha cos\beta & cos\alpha cos\beta \end{bmatrix}
\end{aligned}
$$

2.1.1 Robot Encode method

To distinguish the different configurations of the robot, we define the icosahedron faces from index 0 to 19, and three parts of leg component rotation from index 0 to 11. With this encode method, the generating program can easily assign some random numbers to create random robot

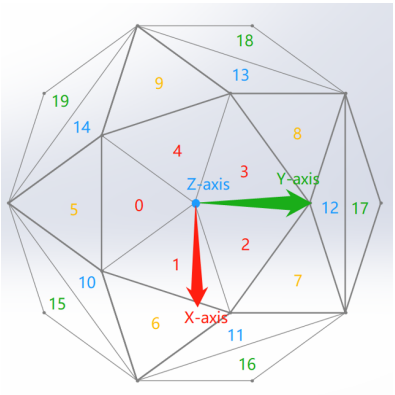


Figure 2: Index 0-19 of the 20 faces robot body

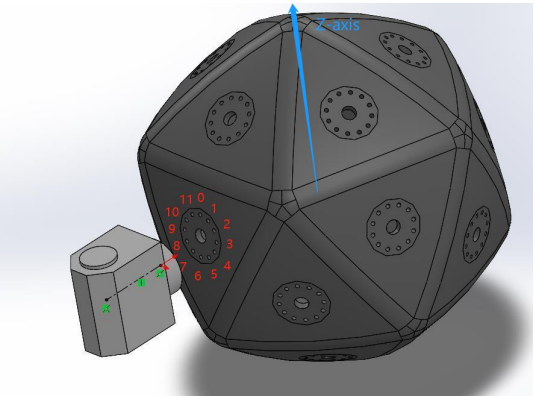


Figure 3: Index 0-11 of the leg component rotation

For a four-legged robot, we can encode the robot's name to a 16-digit number to show which four faces are connected to the robot's leg and what kind of leg rotation is. For example,

2.1.2 Different initial angle of variety of configuration

In addition, after generating a URDF file, the program will assign random initial angles to each joint of the robot and check if there are any unexpected collisions between different components of the robot. With no collision happening, the program will test if the feet of each leg point to the ground and are under the lowest point of the robot's body. And the height of each leg should be at a similar

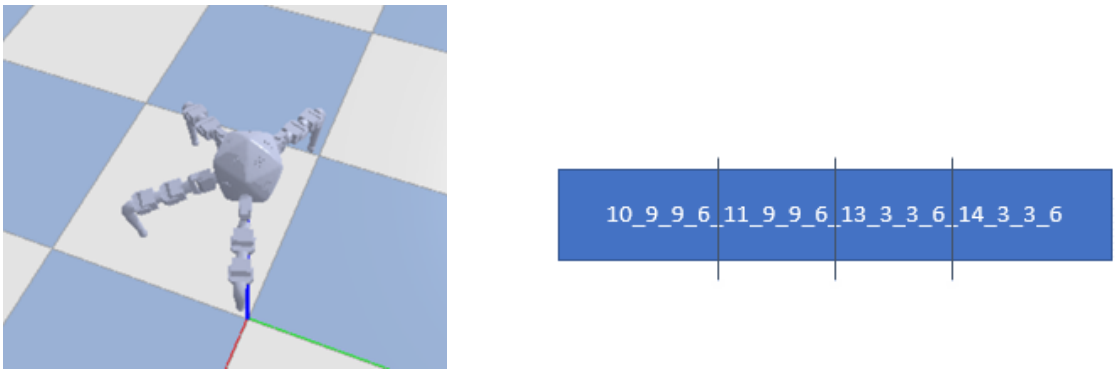


Figure 4: Example Robot:“10_9_9_6_11_9_9_6_13_3_3_6_14_3_3_6”

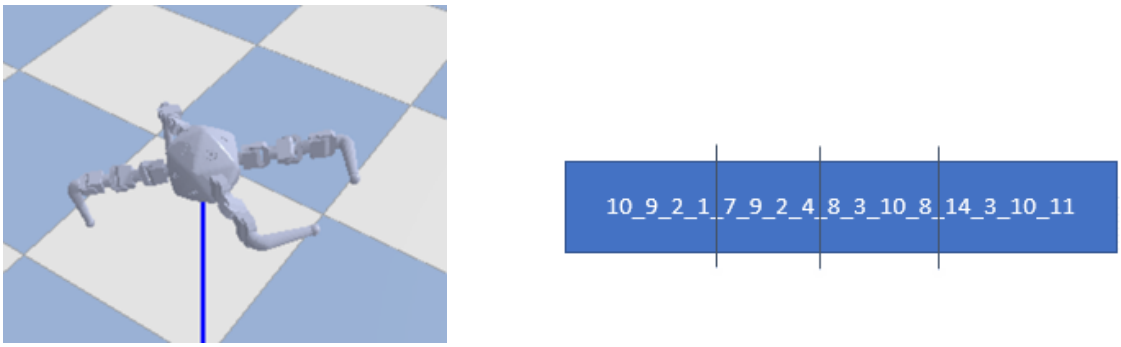


Figure 5: Example Robot:“10_9_2_1_7_9_2_4_8_3_10_8_14_3_10_11”

level. I also tried to use different mathematic methods to make sure the projection of the mass center of the robot’s body is always inside the triangle created by three of the robot’s feet. However, the criterion is too strict, and the method is not able to provide the desired result. Therefore, I tried inverse kinematic methods to adjust the robot’s joint angles, which passed the previous test and had a highly possibility of a solvable result. With the inverse kinematic method, the program will search a 10mm * 10mm * 10 mm cube space of each foot from initial angles to find if there is a way to let those feet be at the same altitude with no collision. If the inverse kinematic is solvable, the program will assign the solved result as a new initial angle and apply it to the robot. The last step of the examining process is that the simulator will drop the robot from 25 cm height to the ground. If the robot is still able to stand on the ground, the program will save the robot as an urdf file and initial angles as a csv file.

2.2 Sin Gait exploration

To explore the gait of different robot motions, I tried using the sin gait method with 10 random parameters to optimize the best sin gait function to control the robot and its success movement. Here is the simple algorithm of the sin gait function:

This method is able to find an acceptable parameter to control the robot motion, but the learning rate of the process is too low because the whole process is random, sometime I may spend more than 5 hours to find only a set of parameters that allow the robot to move forward. So, I decided to do hill climber with evolution algorithm to keep the best genre of previous parameters and “breed” new parameters based on the best genre. The new algorithm shows there:

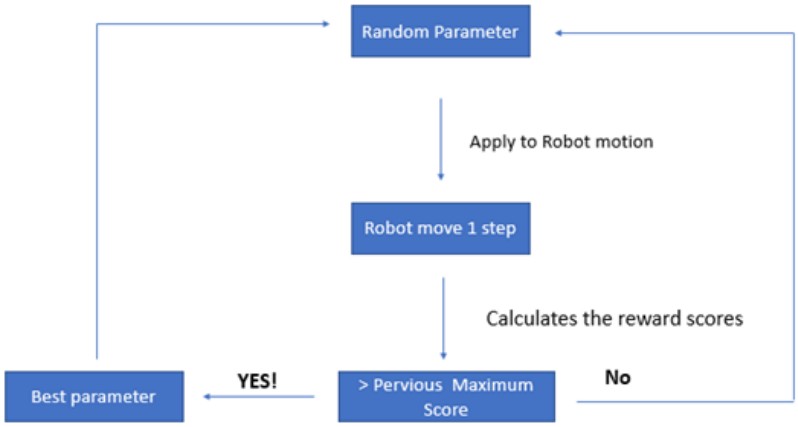


Figure 6: The first version of Sin gait algorithm

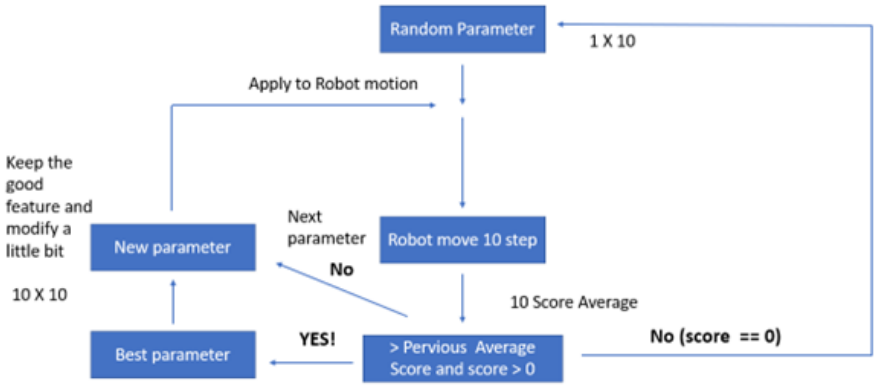


Figure 7: The second version of Sin gait algorithm with evolutionary algorithm

2.3 Domain Randomization

The efficiency of the searching process is improved a lot, but a new obstacle appears. When I applied the excellent parameters to a simulator, the robot's movement was smooth and accurate and able to execute the plan of motion. But using the parameters of a real-world robot, the result can be a disaster. The real-world robot may move backward and turn around for the moving forward parameters. To solve this sim2real problem, I tried to update the properties of the robot in a simulator, such as the mass and moment inertia of each component, motor's force, and velocity, to make the simulator properties as much as closer to real-world properties. To get the real-world data, I used weight to measure each component of the robot and created a program that can return a graph with the change in the motor's position. Then I can adjust the motor's force and velocity in a simulator to make the simulated robot has the same motor changing as the real world. I also read the paper, "SOLVING RUBIK'S CUBE WITH A ROBOT HAND," edited by OpenAIOpenAI et al. (2019). And they described a method called domain randomization, which allows people to create different simulated environments with randomized properties. If the model can find a good result in those randomized environments, the robot may have a high chance of making the same motion in the real world. Therefore, I updated my algorithm again to allow the robot to run a set of parameters in the different randomized environments and find the set of parameters that has a highest average score.

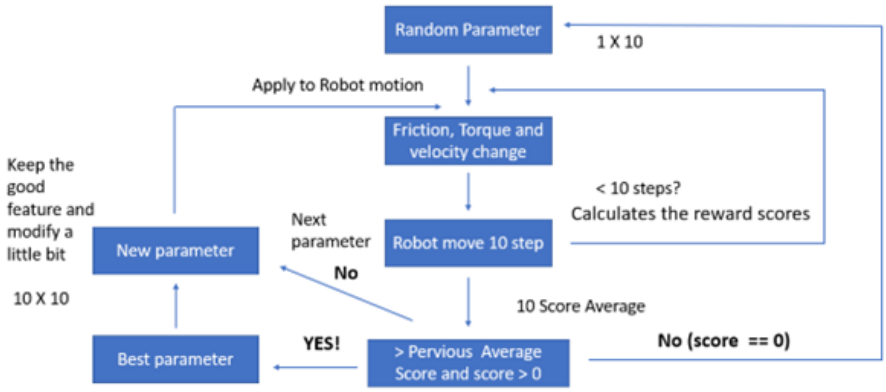


Figure 8: The third version of Sin gait algorithm with Domain Randomization

Compare the result with DR and the result without DR in the simulator:

	With DR		
		x	y
	0.5	0.146091	4.193737
	0.6	0.123622	4.212219
	0.7	0.070106	4.210204
	0.8	-0.06661	4.190717
	0.9	-0.14866	4.172972
Std Dev		0.12745	0.016032
	Without DR (== 0.7)		
	0.5	-1.35308	4.543038
	0.6	-0.7093	4.758384
	0.7	-0.02187	4.839723
	0.8	-0.34784	4.791108
	0.9	-0.49332	4.761923
Std Dev		0.49678	0.114195

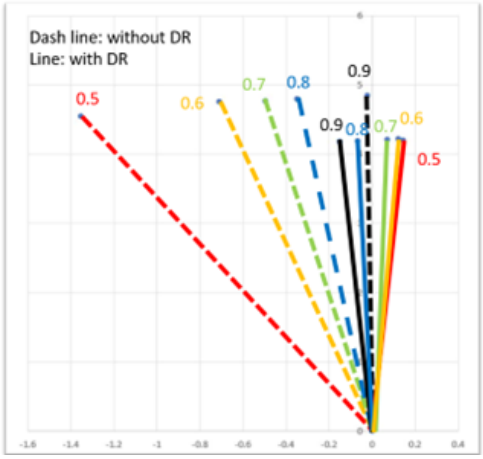


Figure 9: Moving distance with DR and without DR

From the chart shown above, we can see that the result of using Domain Randomization has much less value of standard deviation, which means the robot is more likely to be able to run in a different real-world situation. To exam if the simulated and real-world robots are able to make similar motions. I applied the demo robot model, ID 222, with a set of trained parameters to control the robot moving forward. The results show that:

	Y-direction(forward)(meter)	X-direction(meter)
simulation(friction = 0.6)	2.06	0.246
simulation(friction = 0.8)	2.044	0.032
real world	$\sim$1.82	$\sim$0.26

Although the actual result and simulated result are still different, the robot is able to finish the same purpose by sin gait control with the same parameters. The next step we can take is to keep updating the properties in a simulator and minimize the gap between the real world and simulation by using the DR method. The simulated model can also be improved because the joints in the simulator have zero weight, but in the real-world robot, the motor placed in the joint position is heavy. So,

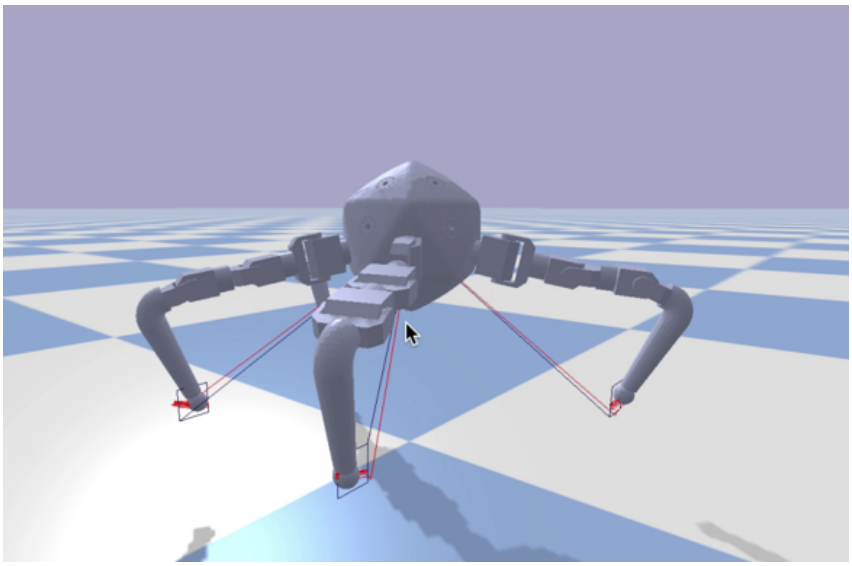


Figure 10: Inverse kinematics function in PyBullet fails

sometimes when we compare the simulated motion and real-world motion, the real-world robot is not able to raise the robot's leg but only slides on the ground with a similar motion.

2.4 Socket development

We are able to collect millions of State-Action-State data from the simulation, but the simulated result is not good enough for building a self-model robot. To break the limit of simulated data, I used the Socket package that allows the robot to communicate with my PC anytime. The robot would return the motor position to the PC during the motion, and the real-sense camera returns the position and orientation change of the robot each step. And the program will process the real-world data to SAS form and apply it for further learning use.

3 Gait Optimization

After we have explored different configurations of legged robots and generated the the URDF files for each configuration.The next step is to design some gaits so that the robot can walk. In this section, we figured out the approach to make the robot walk towards the targets in simulation.

3.1 Handcrafted Walking Gaits

At the beginning, I manually designed some simple feet trajectories and simulated them in the PyBullet. Given the trajectories of the feet, I applied the built-in function $calculateInverseKinematics(IK)$ in PyBullet to generate the desired joint positions of the bus servos. However, it failed no matter what kind of trajectories the robot has. The problem is that the IK function in PyBullet only returns the correct joint configuration if it is close to the IK solution. Figure 2.110 shows that there is a big difference between the actual feet trajectories (red line) and expected feet trajectories (black line).Even though we have tried different shape of trajectories, the actual trajectories are always different from the expected trajectories. Finally, we gave up and moving forward using hill climbing algorithm and evolutionary algorithm.

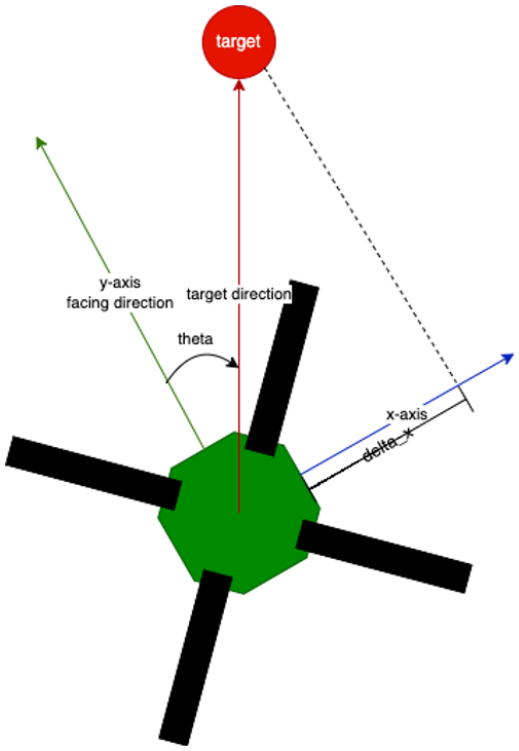


Figure 11: The range and bearing schematic

3.2 Gaits from Hill Climbing

We tried using the sin gait method with 10 random parameter to do the hill climbing algorithm. Some of the parameters are shared to mimic the crossed gait of the animal. We only choose the best parameters to evolve so that the robot would walk faster and faster. We use random parameters only but the gaits we finally generated were good enough.

3.3 Self-correctable Gaits

Then, I optimized the self-correctable gaits to let the robot walk properly and follow the targets. Once the robot can walk, we need the gait itself can cancel off the range and bearing offsets. Figure 11 shows the metrics to determine the offset of the robot. The purpose of this part is to reduce the sim-to-real gap and adapt the robot itself to walk within an acceptable range. In this part, I modified the sin gait method to a closed-loop function by adding more parameters in joint movement functions. Based on he range and bearing the robot detected, the joint movement functions will adjust themselves.

To find the self-correctable gait, I applied the evolutionary algorithm to evolve the gait. We applied several strategies in evolving the gait. First, we added a slightly uneven terrain to simulate the real-world floor. Second, in each epoch, the robot needs to complete a series of tasks. In Figure **??**, the robot has to move towards the multiple targets. If the robot can reach the targets in different directions with a high reward, the robot will have a high variance to adapt to different collision and obstacles in the environment. Finally, we adjusted the mutation, crossover and parent selection in evolutionary algorithm so that the model can explore more in parameters. Finally the result is good, the robot itself can adjust its gait automatically even collision happens. In Figure13, the blue lines

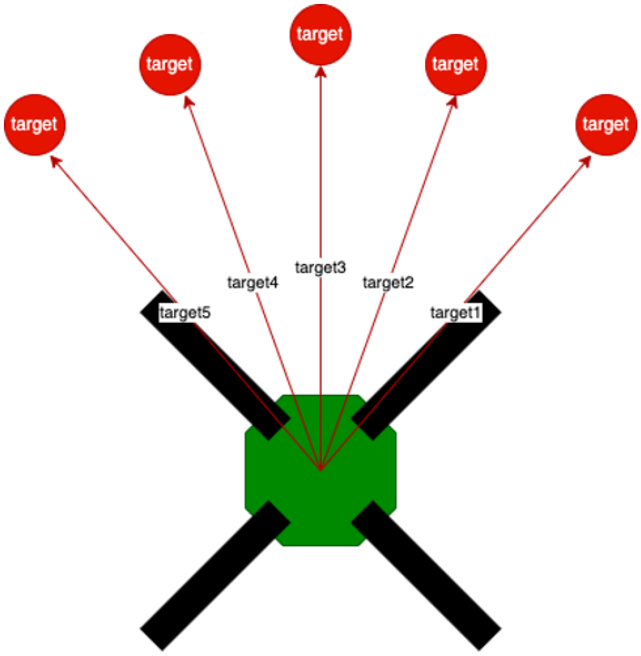


Figure 12: In each epoch, the robot will move towards multiple targets

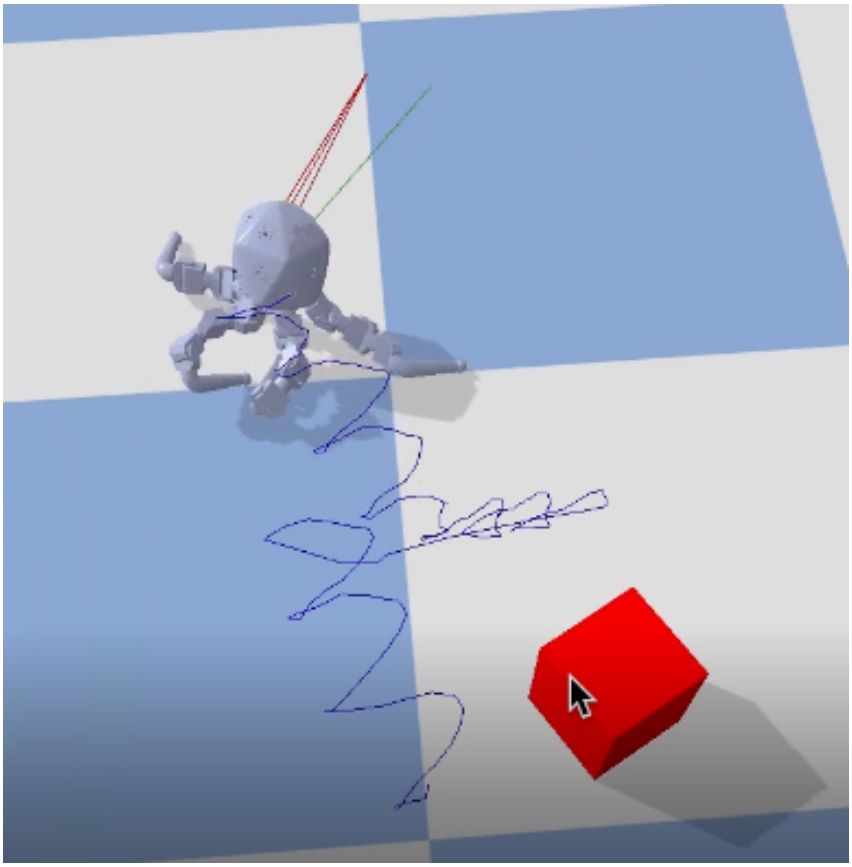


Figure 13: In each epoch, the robot will move towards multiple targets

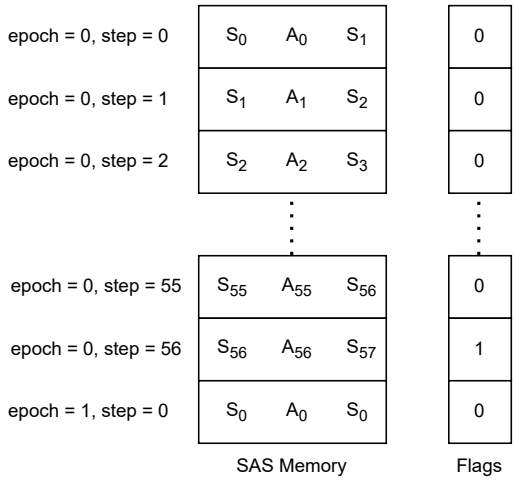


Figure 14: A sequence of SAS memory and robot flags

are the tracks of robot movement. From the track, the red obstacle collides with the robot in the half way and pushes the robot to the right side, but the robot itself can adjust its gait can cancel off the range and bearing offsets.

4 STATE-ACTION-STATE MEMORY

Following previous self-modeling research experience Kwiatkowski & Lipson (2019), we formulate the robot memory as a sequence of State-Action-State (SAS) memory, where a SAS memory piece at any given timestep t is a concatenation of the robot's current state S_t, the action a_t it takes at S_t, and the resulting next state S_{t+1}. The robot's guided actions at any given time follows a deterministic sine-gate baseline policy $a_t = \pi_\theta(t)$, where θ is a set of sine-gate parameters optimized by hill-climbing. Each state is a vector of size 18, consisting of the change of position, change of rotation, and current joint angles. Each action is a vector of size 12, stating the target joint position. Actions are made by driving the actuator a fixed amount of timestep in the physical simulation considering factors such as joint forces, velocities, and environment frictions, so the resulting next state joint angle isn't a simple addition of the current joint angle plus joint action, but an observation from the simulating environment.

To collect the SAS memory sequence, we run the simulation for a total fixed amount of timestep seperated by epoch while the robot performs the baseline action. We reset the environment if it reaches the maximum steps within an episode or the robot triggers a flag, which usually happens when it is tripped off. Figure 14 shows the structure of a SAS memory sequence where the robot tripped off during epoch 0 at timestep 56.

5 SELF-MODEL: DATA SAMPLING AND TRAINING

5.1 THE PIPELINE OF SELF-MODEL FOR ONE ROBOT

As is shown in Figure15, the complete pipeline of the self-model includes four parts. First, for one configuration of robot, we can select a example gait, using optimization methods like hill climber and evolutionary algorithms. Second, using different data sampling approaches, we can collect our

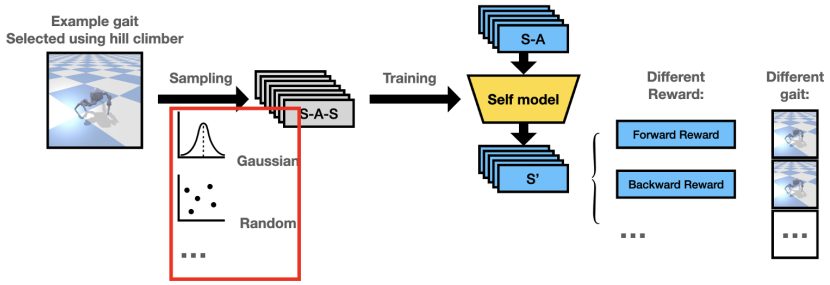


Figure 15: The pipeline of training and applying self-model on one robot

training dataset, which is sets of state-action-state data. Next, we can train the self-model that can predict the next state with current state and action input. Finally, with different reward functions and the self-model, we can pick the gaits that moving in four directions.

5.2 Data sampling test

We hope to find a data sampling method that has High data efficiency. As is shown in table1, we are doing experiments with: random sampling, adding different sized Gaussian noise. We can see that Gaussian sampling has a better performance than random sampling. Gaussian 0.1 sampling has the smallest loss, but it converges very fast and is easy to overfit to one kind of gait. Gaussian 0.2 sampling is good enough and can find different gaits.

	Random	Gaussian, sigma=0.2	Gaussian, sigma=0.1
20k	0.009924	0.000916	0.000339
50k	0.006396	0.000831	0.000324
100k	0.003094	0.000796	0.000323

Table 1: data sampling and testing loss

5.3 Model evaluation

After training the self-model, we also did the model evaluation by comparing the state predicted by the self-model and the state given by Simulator. This is the result we collected in 50 steps: Figure16. This graph shows the difference between the self-model and the simulator in x, y, z and row, pitch, and yaw. Our self-model can give a decent prediction in x, y, roll and pitch, while it does not do as well in z and yaw.

6 Predictive Self-Model

Given the current state (and some past states history) and the proposed actions, we build a neural network that predicts the next state as if the proposed action is executed. The model architecture is shown in figure 17. The state history is a sequence of states at past timesteps with varying length from (1-8), it is then encoded to a fixed length embedding (green) through taking the hidden state last layer of a 3-layer LSTM network. The action is encoded to a embedding (red) through 2 MLP layers. The state embedding and action embedding is then passed through 3 MLP layers to predict the next state. Each MLP layer block consists of a linear layer, a ReLU activation, and a one-dimensional batch normalization Ioffe & Szegedy (2015) layer.

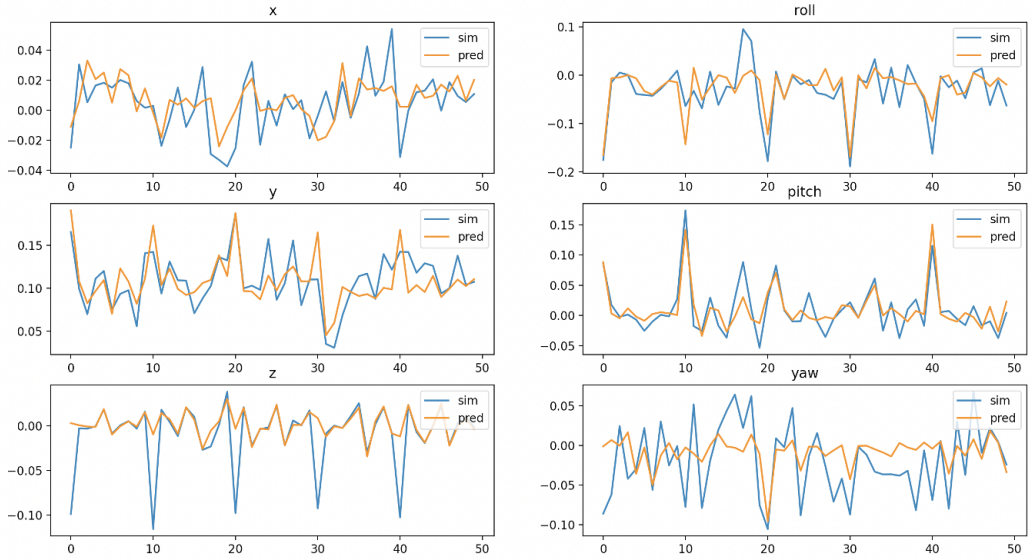


Figure 16: The predicted states by self-model, ground truth: Pybullet

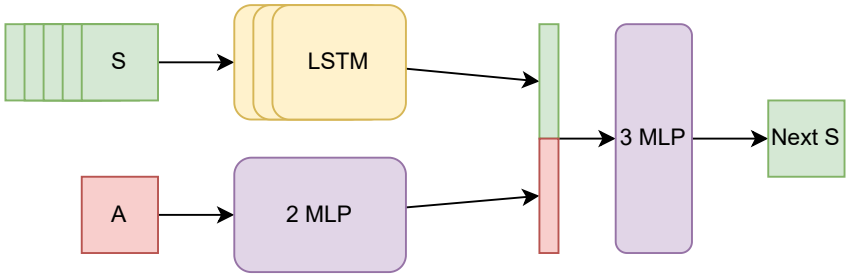


Figure 17: Predictive self-model architecture: predicting next state given state history and proposed action

In most cases, the model was trained using an Adam optimizer with learning rate $3e-4$ and weight decay of $1e-5$. The model was trained with a batch size of 32. Both the state and action embedding has size 32. Model gradient are clipped with threshold 1 at every optimization step. We empirically found that taking 20K memory steps gives relatively good prediction result.

This predictive model does not have any information about the robot's configuration, and it works pretty well for self-modeling a single robot. This model can also be used to learn the motion dynamics for multiple robots, where the state and action spaces contains SAS samples collected from different robot configurations. However, if the predictive model can be conditioned on some unique robot identify information, such as its morphological encoding, the model should have stronger predictive power across different robot configurations. We therefore introduces the Meta-Self-Model and robot signature-encoding.

7 META-SELF-MODEL

The predictive model shown in figure 17 works well for self-modeling a single robot. We call this predictive model, without any robot identity information, the baseline predictive model. Nonetheless, if we can encode some robot identity information and make predictions conditioned upon it,

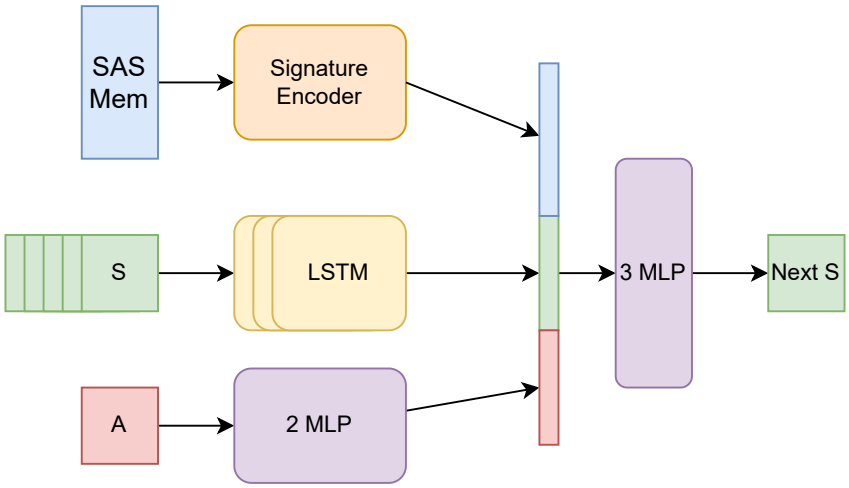


Figure 18: Predictive Meta Self-Model architecture: predicting next state given state history and proposed action conditioned on the robot signature encoding

we are expected to see improvements. This is based on the observation that robot with different morphology should innately possess different physical capabilities. For example, given two robots, one with longer legs than the other, and asks the question: let both robots make one step forward, which one is expected to move further? Since we have the extra information of robot's morphology, that one robot has longer legs than the other, we can make better a prediction that the robot with longer legs will be most likely step further away.

However, the direct robot configuration/morphology encoding information may not be directly accessible given a new robot. Also, we want the model to be able to generalize to unseen robot configuration while there are infinitely many of them. We thereby introduces a robot signature encoder which implicitly acquires robots identity information purely learned from the collected robot SAS memory, that hopefully contains information about the robot's configuration that help us make better predict future states.

The proposed network architecture can be seen in figure 18, where an additional signature encoding from the SAS memory (blue) is added for the next-state prediction. The signature encoder E_{sig} is essentially a mapping function that encodes a sampled SAS memory piece into a fixed length vector $E_{sig} : R^{k \times 48} \rightarrow R^{d_{sig}}$, where k is the sample size and d_{sig} is the signature encoding dimension. We experimented with various types of signature encoder architecture discussed in section 7.1.

7.1 Robot Signature Encoder

The robot signature encoder is an additional model added to the predictive network which encodes a sampled SAS memory block to a meaningful vector representation that somehow tells the identity of the robot. One assumption that we make about the encoding is that the encoder should handle SAS memory in a permutation-invariant manner. That is, given SAS instants at different timesteps without any particular order, the encoded vector should remain the same. A common way of creating permutation-invariant encoding is to use the attention mechanism and pooling operation, and it has been studied quite extensively in the point cloud literature.

To be specific, we experimented with a GRU encoder Chung et al. (2014), a multi-head attention Vaswani et al. (2017) encoder, a Set-Transformer Lee et al. (2019) encoder, and a Point-Cloud TransformerGuo et al. (2021) encoder. Model architectures can been seen in figure 19.

Unfortunately, even with extensive experimentation regarding the robot signature encoding, we are constantly facing overfitting problem such that the predictive network could not adapt well to validation dataset with unseen robot configuration, and the performance of the meta-self-model (with signature encoder) could not perform better than the baseline self-model (without signature encoder). We further noticed the model, that simply takes the mean over the time dimension as pooling followed by a few MLP layers, could slightly out-perform the baseline self-model.

We hypothesis that the actual problem lies in the training data being collected, where not all robot configurations possess optimized sine gate to collect sample-efficient data for self-model training. The exploration process needs further investigation.

For the SAS memory collected for each robot, we split the sequence in 4:1 ratio for training and validation dataset split. We then further split the dataset (either train or validation) in 1:1 ratio for signature memory sampling and the predictive model training. For example, a SAS memory sequence of 20K timesteps will be divided into 16K for training and 4K for validation. The 16K training data will be further divided into 8K for memory sampling and 8K for predictive model training.

7.2 Memory Sampling and Position Embedding

Given a sequence of SAS memory, we don't need to use all of them to training the signature encoder, as it could contain redundant information and increases computational complexity. Instead, we should sample a subset of instances that best describes the overall structure of the entire memory sequence. The sampling method itself remains an interesting research topic itself. For example, researchers have came up with methods such as further-est point sampling Qi et al. (2017) and multi-view partial sampling Mohammadi et al. (2021) from the point cloud literature.

To examine memory ordering information, we experimented with two types of shuffling methods and positional embedding. After shuffling, we take the first k memory instances as the memory sample, and sum with the sinusoidal positional encoding introduced by Vaswani et al. (2017). We attempted two types of shuffling: 'shuffle by step' mix up the order of the entire sequence; 'shuffle by epoch' keeps the steps ordering within each epoch, and mix up the order across different epochs. We also attempted two types of positional encoding shown in figure 21.

8 Locomotive Trajectory Planning with Self-Model

The self-model gives robots a sense of its motion dynamics. Having this 'extra sensing' information we may build algorithms that improves robots ability to perform various locomotive tasks. While having a somewhat optimized sine-gate baseline action, our result shows that the addition of the self-model allows robot to perform local optimization by searching through noised action spaces in real time. It also allow robots to perform tasks such that the robot has zero knowledge about by crafting a reward function. In this section, we present several reward functions that allows robot to improve its forward motion tasks, and zero-shot learn to unlearned tasks such as backward motion, turning left or right. The planning is based on a local search that finds the highest cumulative reward in the "imaged action trajectory" in self-model.

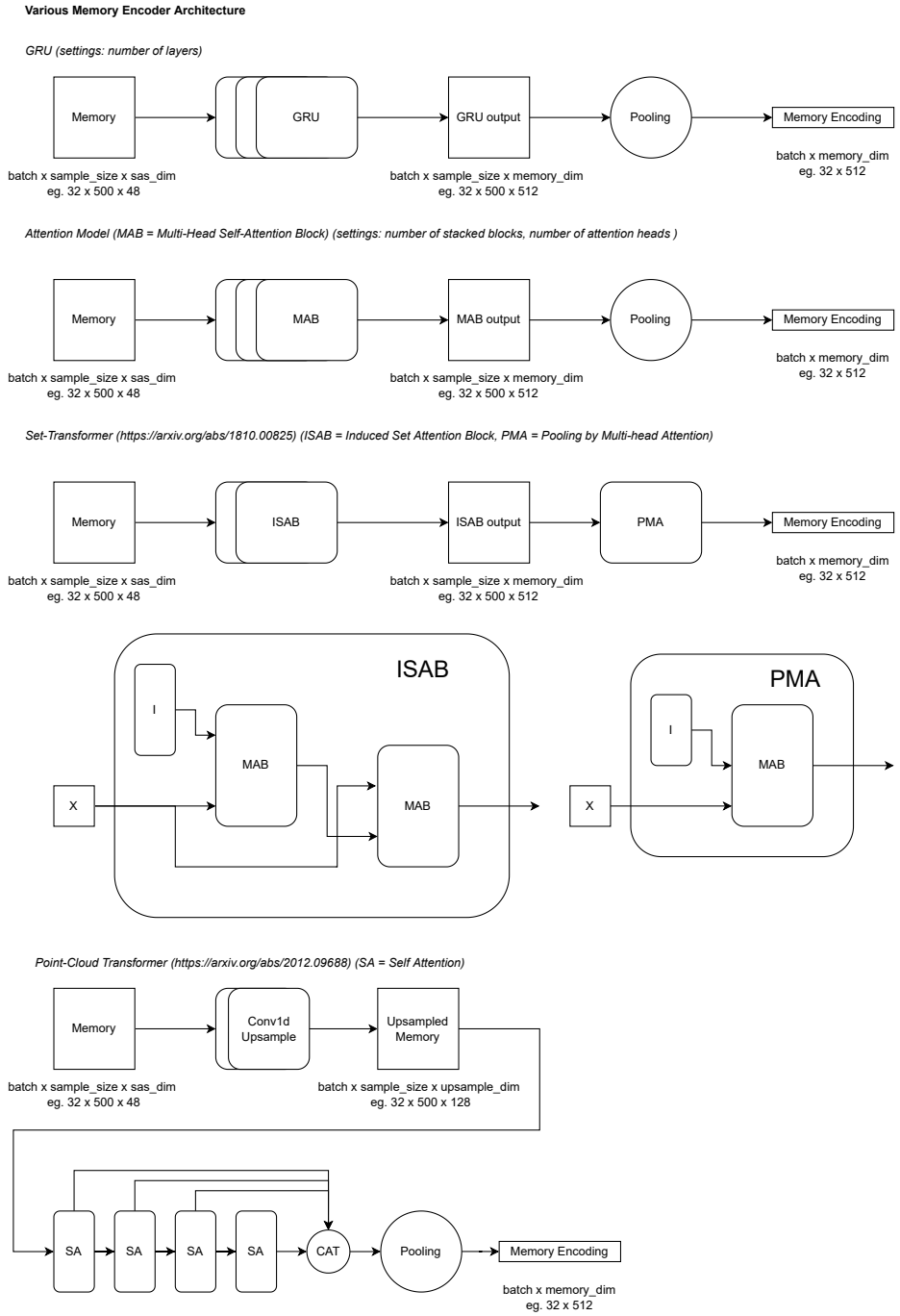


Figure 19: Various robot signature encoder experimented.

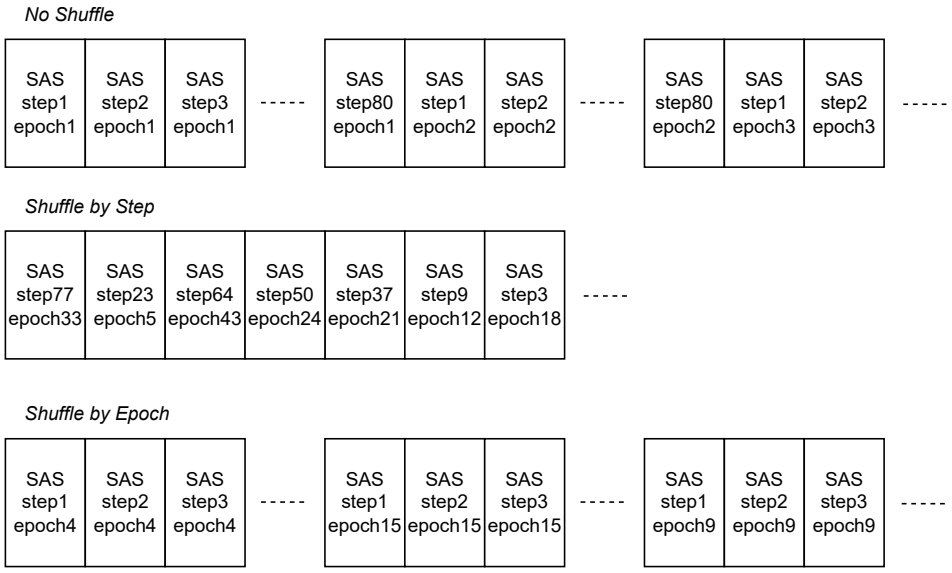


Figure 20: Various memory sampling/shuffle method with regard to SAS ordering information experimented.

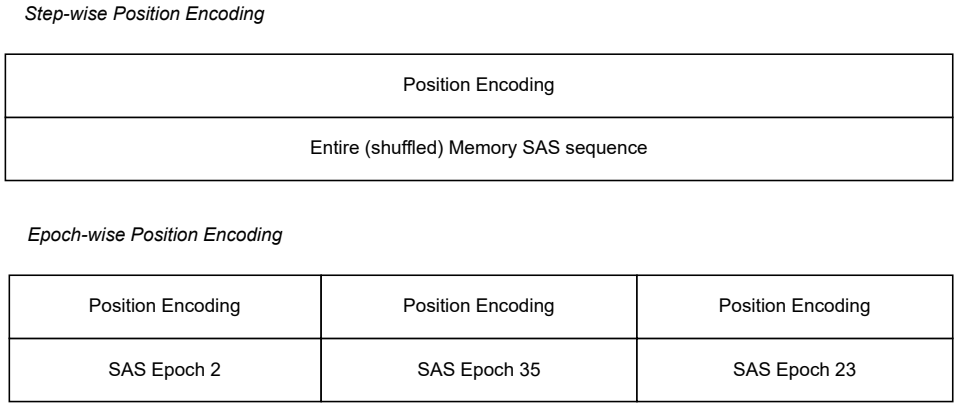


Figure 21: Two types of sinusoidal positional encoding experimented.

8.1 REWARD FUNCTIONS

The state observed by the robot contains change of position Δx, Δy, and Δz and change of rotation roll $\Delta\phi$, pitch $\Delta\theta$, and yaw $\Delta\psi$ information. In our simulation, the forward direction matches with the y axis, so a simple forward reward function should encourages shift in the y direction while a backward reward function should encourages shift in the $-y$ direction. A turning reward function could therefore control the change of yaw as defined below.

$$
\begin{aligned}
R_{forward} &= \widehat{\Delta y} \\
R_{backward} &= -\widehat{\Delta y} \\
R_{left} &= \widehat{\Delta\psi} \\
R_{forward} &= -\widehat{\Delta\psi}
\end{aligned}
$$

Nonetheless, a reward function can also be composed of various factors. For example, the forward reward function below encourages shifts in the y direction, avoids offsets in the x direction, and avoids ψ rotation to make sure the forward motion maintains in a straight line.

$$
R_{forward} = \lambda_1 \cdot \widehat{\Delta y} - \lambda_2 \cdot |\psi + \widehat{\Delta\psi}| - \lambda_3 \cdot |x + \widehat{\Delta x}|
$$

We also discovered that the reward function doesn't have to include reward explicit to the task. For example, a reward that avoids change of rotation at all could make a good forward reward function.

$$
R_{forward} = -|\psi + \widehat{\Delta\psi}|
$$

8.2 LOCAL TRAJECTORY PLANNING/OPTIMIZATION

The self-model gives robots the ability to predict its future states. Once the self-model is trained, it can be applied to the robot in many ways. Here we demonstrate two types of searches that locally optimizes the guided action. The greedy search looks ahead states at one future timesteps, it is relatively computationally efficient but myopic. Algorithm 1 shows the planning procedure done by the greedy search. Given the guided policy π_θ with parameters optimized through hill-climbing and the current timestep t, we first obtain the baseline guided action a_g. We them propose k extra actions by adding Gaussian noises to a_g. For each action, we predict the future state with the self-model as if the action is executed and select the action, that results in the highest reward based on the current state and the predicted future state, as out greedy action.

One of the problem of the greedy search is that it only looks one timestep into the future. While the consecutive execute of two actions, where each one of them might not give the biggest gain along but could gives the maximized reward when doing both. To make sure we don't miss actions like these, we borrowed the idea of Beam search that's been widely used in decoding text in neural language models. Instead of finding the best action at a specific timestep, the beam search finds the top k actions, and maintains k action trajectories with the highest reward overall, looking further into the future. However, since it needs to compute more timesteps and the prediction error amplifies at each

timestep, it is relatively and prone to error. One way to optimize the beam search is to maintain a priority queue, and we will leave that to future work.

Algorithm 1: Greedy Trajectory Planning through Self-Model

Input: self-model SM, guide policy π_θ, Gaussian deviation σ^2, current timestep t, reward function R, past d states $S_{t:t-d}$

Output: Best action planned through self-model a^*

$a_g \leftarrow \pi_\theta(t)$

$\vec{\epsilon} \leftarrow [\epsilon_1, \epsilon_2, ..., \epsilon_k], \epsilon_i \sim \mathcal{N}(0, \sigma^2)$

$\vec{a} \leftarrow [a_1, a_2, ..., a_k, a_{k+1}], a_i \in \{a_g + \vec{\epsilon}\} \cup \{a_g\}$

$a^* \leftarrow \underset{a_{i:0 \le i \le k+1}}{\arg\max} R(S_t, SM(S_{t:t-d}, a_i))$

9 Discussion and Future Work

9.1 Looking Deeper into the Future

The current self-model only predict the state at future one timestep. Nonetheless, it is interesting to us if we could predict more states that goes further into the future. This way, the self-model can learn more robust representations of motion dynamics with long-term dependencies. To do so, we want to borrow the sequence-to-sequence model which can be implement with both RNN or the Transformer architecture from natural language process literature. The task will be similar to dialog system or machine translation where the model outputs a sentence given an input sentence. The SAS memory shares similar structure: the SAS instance at each timestep could be seen as a word token, and the SAS memory sequence in one epoch is analogous to a sentence.

9.2 Life-long Learning

One limitation of our current data-collection and training pipeline is that: the baseline sine-gate optimization process is separated from the self-model training. We first optimize the gate behavior to a certain degree that enables the robot to walk forward and hope that, by adding certain amount of action noise, the collected state-action experience could be sample-efficient and represent most states that the robot might seen. The reality is, our current self-model only explores a tiny fraction of the possible states spaces since the behavior is already so pre-defined, and the prediction accuracy is relatively low to unseen situations.

We hypothesis that a robust self-model training paradigm should combine the model training process with the gate optimization process, with both module providing feedback with each other, and train the self-model using state-action data generated on the fly. The self-model trained using this online learning pipeline contains life-long dynamics experience, thereby could handle unseen scenario much better.

REFERENCES

Josh Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121, 2006.

Boyuan Chen, Robert Kwiatkowski, Carl Vondrick, and Hod Lipson. Fully body visual self-modeling of robot morphologies. *Science Robotics*, 7(68):eabn1944, 2022.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R Martin, and Shi-Min Hu. Pct: Point cloud transformer. *Computational Visual Media*, 7(2):187–199, 2021.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Robert Kwiatkowski and Hod Lipson. Task-agnostic self-modeling machines. *Science Robotics*, 4 (26):eaau9354, 2019.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pp. 3744–3753. PMLR, 2019.

Seyed Saber Mohammadi, Yiming Wang, and Alessio Del Bue. Pointview-gcn: 3d shape classification with multi-view point clouds. In *2021 IEEE International Conference on Image Processing (ICIP)*, pp. 3103–3107. IEEE, 2021.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019. URL `https://arxiv.org/abs/1910.07113`.

Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

A WHO DID WHAT

- Ruibo Liu
  - Auto URDF generating script
  - Sin Gait exploration with hill climber and parallel evolutionary algorithm
  - Using Domain Randomization to reduce the gap between simulator and real world robot
  - Socket communication development for robot and PC
  - Improving and build robot's inner structure
  - Assembling different configuration robots
- Zhou Shen
  - Manual gait exploration such as circle, triangle trajectories in PyBullet environment
  - Using hill climber to generate diagonal symmetric gait
  - Remotely control the robot and collect data from the sensors like Intel Realsense Camera
  - Applied concepts in evolutionary algorithm to deal with range and bearing offset problem in simulation
- Jiong Lin
  - Gait selection with random search
  - Data (State-action-state) collection for 10 different robot
  - Self-model training
  - Data sampling test (random sampling and adding Gaussian noise)
  - Self-model evaluation, ground truth: Pybullet
- Yunzhe Wang
  - Collect SAS memory on gate-optimized robot, 1024 robots with 20K steps each having action noise 0.1, 0.2, and 0.5. (Total $1024 \times 20000 \times 3$ instances)
  - Experimented with various robot signature encoder architecture (GRU, MLP, multi-head attention, set transformer, point cloud transformer), pooling methods, SAS memory sampling/shuffling methods, positional encoding, and training objectives (auxiliary tasks and multi-tasks learning) for meta-self-model.
  - Implemented greedy and beam search for action trajectory planning and local optimization using predicted future states from trained self-model