CSC2626 Final Project Report: continuous controllability and autoencoder architectures for learned latent actions

Sergey Protserov
Department of Computer Science
University of Toronto
Canada
s.protserov@mail.utoronto.ca

Yujie Wu
Department of Mechanical and Industrial Engineering
University of Toronto
Canada
yujie.wu@mail.utoronto.ca

Abstract: Robotic arms and other manipulators have many important practical applications, such as assisting users with disabilities to perform everyday actions or helping clinicians perform surgeries. One of the important shortcomings of such systems is that they typically have many degrees of freedom (5 or more). While this allows for high dexterity, it also makes it challenging to operate such manipulators due to the high cognitive load of controlling every degree of freedom individually. One potential solution to this issue is to learn a mapping from low-dimensional controls to high-dimensional controls, which would allow a user to control the robotic arm using a low-dimensional controller, such as a joystick. Losey et al.[1] propose an algorithm for learning low-dimensional ”latent actions” and a mapping from ”latent actions” to high-dimensional actions from state-action pair datasets using variations of autoencoder neural network architecture. In this project, we reproduce parts of their work, and expand on it in two different ways. First, we experiment with an orthogonality constraint for the learned latent action space with the help of orthogonal autoencoders. Secondly, we analyze the possibility of moving a manipulator along an end-effector trajectory by issuing *continuous* actions in latent space, as opposed to latent actions potentially belonging to arbitrary parts of latent space, which would make them harder for human operator to control. Our experiments show that orthogonal autoencoders do not provide a significant and reliable improvement when compared to regular and variational autoencoders for latent spaces of low ($d = 2$) dimensionality. Our tests of *continuous* controllability show that this requirement significantly hinders the manipulator's ability to traverse synthetic end-effector trajectories, which may be a problem for real-life applications of low-dimensional latent actions.

Keywords: Shared autonomy, Dimensionality reduction, Feature extraction

1 Introduction

Assistive robot arms have been deeply involved in the lives of people with disabilities. They allow for grabbing and moving objects with only a minimum amount of movement required from one's real arm. For example, a well-known JACO arm system[2] developed by Kinova is a 7 DoF light-weighted mobile mountable robotic manipulator. This system consists of 6 linked segments with a three-fingered hand attached to the end, which facilitates performing various daily tasks. Similarly, Willow Garage PR2[3], which is primarily used to perform fetching and freighting tasks, also has 7 degrees of freedom, but a less flexible end-effector (gripper). Other products, such as KUKA LWR III[4] and Universal Robots UR5e[5] with 7 and 6 DoFs respectively, allow users to switch between multiple end-effector tools, which allow them to generalize to numerous daily and research scenarios. Unlike manipulators with few degrees of freedom that suffer from limited feasible action space, high-DoF arms are usually more difficult to operate. This raises the question about ways to control high-DoF manipulators with low-DoF controllers. Neural networks are powerful feature extractors which provide an easy and efficient way of learning mappings from and to spaces of different dimensionality. The optimization objectives, as well as any additional constraints, are expressed via loss functions. This allows neural networks to be used for learning low-dimensional embeddings of high-dimensional controls. These low-dimensional embeddings can then be manipulated by a user. Losey et al.[1] propose an algorithm for learning low-dimensional controls from datasets of task trajectory demonstrations using autoencoder-based neural network architectures, such as conditional autoencoders (cAE) and conditional variational autoencoders (cVAE), and also formulate three requirements that should hold for the latent action space and the mapping from latent action to ”full” action space to allow for convenient operation, namely:

- latent controllability: it should be possible to move between any two states in the dataset by applying a sequence of latent controls
- latent consistency: same latent action should have similar effects if applied from two states that are close to each other
- latent scaling: applying larger latent actions leads to larger changes in the state

In this project, we focus on the latent controllability requirement. We partially reproduce some of the experiments conducted by Losey et al., and also expand on their work in two ways:

- Following the idea proposed by Wang et al. in [6], we try using orthogonal autoencoders (OAE) for dimensionality reduction. OAEs incorporate a requirement for orthogonality of latent vectors, which, as we hypothesize, should improve controllability and intuitiveness of control using latent actions
- We extend the controllability requirement to a requirement of *continuous* controllability: it should be possible to move between any two states in the dataset by applying a sequence of latent controls, *in which every next latent control doesn’t differ much from the previous one*. We believe this requirement models the properties of real-life controllers, such as joysticks, which typically allow the operator to draw a continuous curve in the control space, instead of applying vectors from different regions of that space

In our project, we evaluate regular, variational and orthogonal autoencoders, as well as their conditional variants, from the point of view of controllability and continuous controllability on datasets comprised of trajectories for two ”toy” tasks: drawing a sine wave and drawing a circle in 3D space. An overview of the learned latent actions concept and our project’s setup is presented in Figures 1, 2.

2 Project diagram

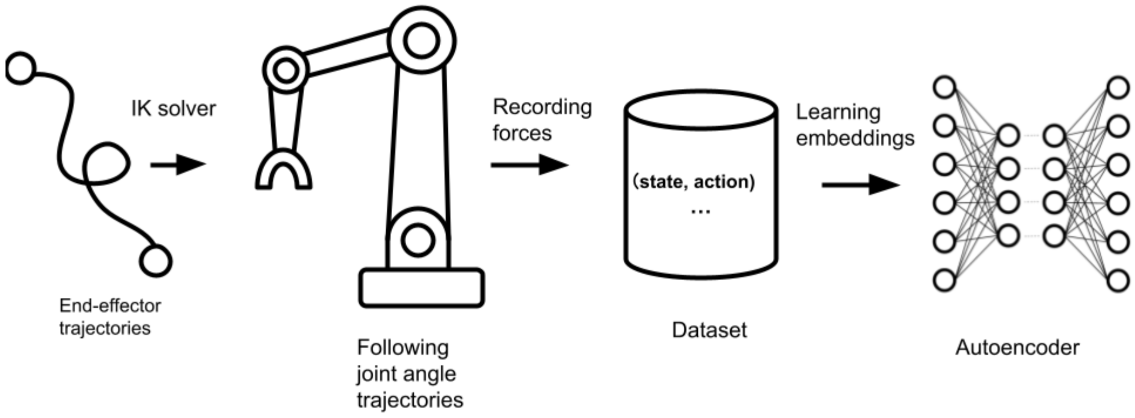

Figure 1: Data collection and training stages

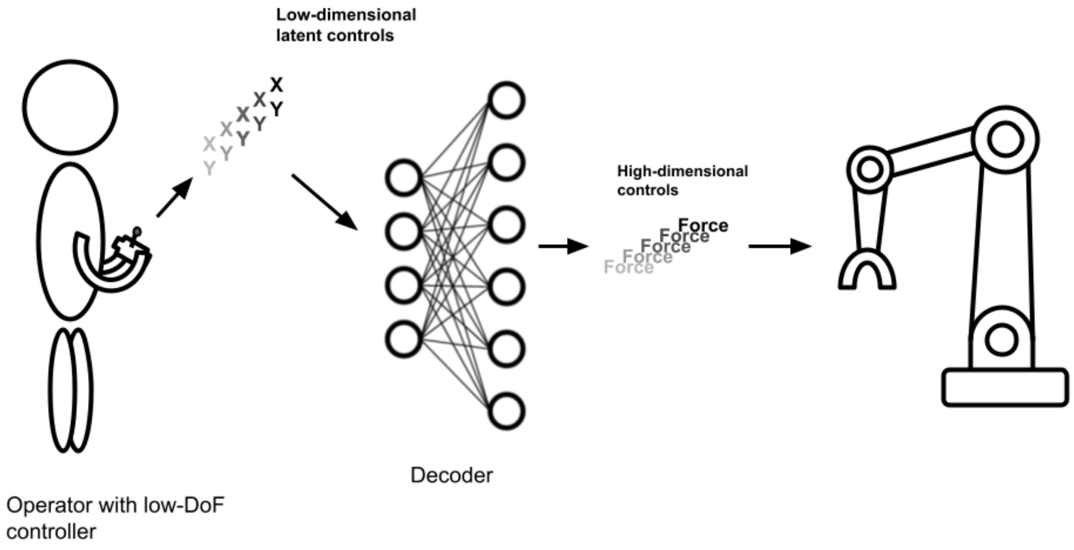
Low-dimensional latent controls
X Y
High-dimensional controls
Force
Operator with low-DoF controller
Decoder

Figure 2: Application of the trained model

3 Related Works

The paper [1] that we base our project on is one of the papers in a large sub-domain of imitation and reinforcement learning, called shared autonomy. This subdomain focuses on how to best combine human input with robot policy for successful task completion. Here we briefly go over some of the prior art in this field. Rakita et al.[7] improve upon traditional inverse kinematics (IK) solvers to ensure that the robot actions maintain motion feasibility when generating a sequence of solutions, while also ensuring that the end-effector pose matches the goal throughout the motion. Mehr et al.[8] present an algorithm for automatically inferring task constraints, such as end-effector moving along a particular plane, or keeping a particular object oriented in a specific way, from online operator input, and assisting them in maintaining these constraints. Broad et al.[9] propose a Koopman operator-based approach to learning system dynamics, and combine it with optimal control methods to augment the user's input and improve the system's stability. Finally, while most works focus on single-arm systems, Tung et al.[10] explored how well shared autonomy scales to multi-arm systems. The authors show that global coordination of multiple arms is only required during specific subtask elements, and further propose and evaluate an approach that mixes centralized and decentralized controls.

4 Methods: overview

Assuming access to a dataset $D = \{(s_i,\, a_i)\}_{i=0}^{n}$ of state-action pairs coming from expert trajectory demonstrations, where $s_i \in S \subseteq \mathbb{R}^m$, $a_i \in A \subseteq \mathbb{R}^k$, and a given latent space dimensionality $d < k$ our goal is to learn a latent space $L \subseteq \mathbb{R}^d$ and a mapping $\phi :\ S \times L \to A$ or $\hat{\phi} :\ L \to A$, depending on whether the current state is taken into account when decoding the latent action. We use autoencoder-based neural network models for learning L, ϕ. An operator will then be able to send low-dimensional controls from L, which are decoded by the model to "full" actions $\in A$ and applied to the manipulator.

5 Methods: data

We have collected our own trajectory dataset for the purposes of this project. We used a simulated version of UR5e[5] robotic hand in a MuJoCo[11] environment as our manipulator. The simulated version of UR5e was taken from MuJoCo Menagerie [12]. We defined two broad groups of toy tasks for our manipulator to perform: draw sine curves in 3D and draw circles in 3D with the manipulator's end-effector. For the sine curve drawing task possible parameters were starting Cartesian coordinate, direction of wave spread, range of the argument change, amplitude, frequency and the rotation angle around the wave direction axis. For the circle drawing task possible parameters were center Cartesian coordinate, radius, rotation axis and angle in 3D. Functions for generating these trajectories are defined in the ***gen_trajectories.py*** file in our codebase, that we upload with this report. Overall we have defined 101 different trajectories for sine curve and 72 different trajectories for circle drawing, please see lines 46 through 89 in the ***main.py*** file in our codebase for the task parameter sets we used in dataset generation. In order to record the "golden standard" controls that would make our manipulator follow these trajectories, we used the following procedure for each trajectory:

- use inverse kinematics solver as implemented in dm_control library [13] to find joint angles that would bring the end-effector to the required Cartesian coordinates. Within one trajectory, after we find joint angles for a required end-effector position, we set the manipulator to these angles before finding angles for the end-effector Cartesian coordinates in the next trajectory step
- position the manipulator at the beginning of the trajectory and make it follow the joint angle trajectory computed on the previous step. MuJoCo engine takes care of finding what forces need to be applied to joints in order to arrive at the given joint angles. We set a new joint angle target (next point in the trajectory) only after the manipulator is within the

specified tolerance (10^{-1} for joint angles that change in ranges $[-2\pi,\ 2\pi]$ or $[-\pi,\ \pi]$) from the current target.
- record manipulator states (joint angles and velocities) and actuation forces that MuJoCo applies to the manipulator while following the trajectory

We noted that during this procedure, for some ”theoretical” end-effector trajectories the manipulator failed to follow computed joint angle trajectories with the specified accuracy. We dropped these trajectories from our dataset. For other end-effector trajectories, the manipulator was able to follow joint angle trajectories with sufficient accuracy, but the ”empirical” end-effector trajectory still differed from the ”theoretical” one due to allowed tolerance or physics/engine limitations. Please see Appendix B for plots of some of the ”theoretical” and ”empirical” end-effector trajectory pairs. Plots for all trajectories in use are also included under ***data*** directory in our codebase. We observed that making our tolerance stricter significantly reduced amount of joint angle trajectories that the manipulator was able to follow. Finally, we observed that forces that MuJoCo actuation engine applies to the 6th joint of UR5e have extremely low variance in our dataset, due to the fact that the end-effector is in the center of a circle, and the 6th joint is responsible for rotating this circle. Because of that, we chose to remove this joint, which resulted in a 5-DoF manipulator. Our dataset resulted in having 86 trajectories for sine curve and 72 trajectories for circle drawing tasks. For the purposes of autoencoder model training/validation and final model evaluation we have split all trajectories into three subsets, train, valid, and test, having 110, 24 and 24 trajectories respectively. Train and valid subsets are then treated as sets of $(\mathrm{state},\ \mathrm{control})$ pairs, where state is a vector of joint angles and velocities and control is a vector of joint forces. Train and valid datasets consist of 444949 and 115754 such points respectively. Test set is not used for model training and validation, and is reserved for the (continuous) controllability evaluation.

6 Methods: models

Following [1], we use autoencoder-based models for learning $L,\ \phi$ from trajectory demonstations. Specifically, the models we use are:

6.1 Autoencoders (AEs)

An autoencoder[14] is an unsupervised dimensionality reduction neural network model that is trained to reconstruct its inputs while passing the data through a low-dimensional bottleneck. As a result, it learns a transform from its input space to a low-dimensional embedding space and a transform from a low-dimensional embedding space back to the input (output) space. It has been shown in numerous machine learning papers that autoencoders are powerful feature extractors. For the purposes of our project, we train AEs with $(\mathrm{state},\ \mathrm{control})$ pairs as inputs and $\mathrm{controls}$ as outputs, while passing the data through a bottleneck of dimensionality $d = 2$. A typical loss function for AE training is mean squared error (MSE). A scheme of autoencoder is shown in Figure 3.

6.2 Variational autoencoders (VAEs)

A variational autoencoder[15] is a probabilistic generative model that encodes its inputs to mean and variance vectors and further decodes a vector sampled from a normal distribution with these mean and variance. VAEs are trained with a loss function that is a combination of a reconstruction loss (typically, MSE) and Kullback-Leibler divergence between the latent vector distribution and $\mathcal{N}(\mathbf{0}, I)$. This allows any latent code sampled from $\mathcal{N}(\mathbf{0}, I)$ to serve as a valid decoder input and be decoded to a vector that is meaningful for the given domain.

6.3 Orthogonal autoencoders (OAEs)

Following the idea proposed by Wang et al. in [6], we implement orthogonal autoencoders that expand on AEs by adding an extra term to the loss function to ensure that latent vectors are closer to being orthogonal to each other. Given a matrix $Z = \{z_i\}_{i=0}^{\mathrm{batch\_size}}$ of latent vectors obtained as

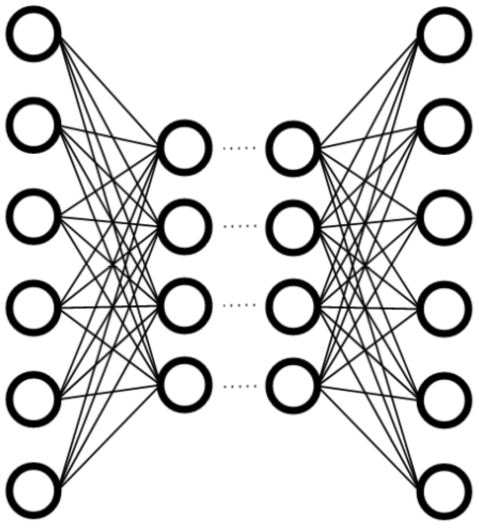
Figure 3: Autoencoder architecture scheme

encoder outputs, orthogonality term is computed as $\|Z^T Z - I\|_F^2$ and added to the reconstruction loss with some weight.

6.4 Conditional autoencoders (cAEs, cVAEs, cOAEs)

Losey et al. have shown that using only latent action vectors as decoder inputs to obtain ”full” action vectors yields poor results in terms of controllability. Far better results were obtained with the help of conditional autoencoders that use both latent action vectors and current state information as the input of decoder part of the model. This extension can be trivially implemented for AEs, VAEs and OAEs. In our experiments we used autoencoders without conditioning, autoencoders conditioned on current manipulator joint angles, and autoencoders conditioned on manipulator joint angles and velocities.

6.5 Architecture implementation details

Our implementations of these architectures are based on repository [16]. In all our models we passed the latent vectors through a hyperbolic tangent (tanh) layer to ensure that their values are constrained to range $[-1, 1]$, which is crucial for our controllability evaluations. Architecture definitions with all the details are available in the ***models_losses.py*** file in our codebase.

7 Methods: model training & validation

We follow typical machine learning workflow for model training and validation. Model inputs are concatenated vectors of manipulator state and actuation forces in case of conditional autoencoders and just actuation forces in case of autoencoders without conditioning. Model outputs are actuation forces. Data preprocessing is done by computing mean and standard deviation of every feature separately over the train dataset. We then perform Z-score normalization with these mean and standard deviation for the entire dataset. For model training we use PyTorch [17] and PyTorch Lightning [18] frameworks. Conditional autoencoders are trained for 200 epochs each, autoencoders without conditioning are trained for 100 epochs each. We use Adam optimizer [19] and OneCycle learning rate scheduler [20]. We tried to keep weight amounts as close as possible between different models that we use, but conditional autoencoders inevitably have more connections between input layer and hidden layer due to higher input dimensionality, and variational autoencoders have more connections in the encoder due to the fact that they encode both mean and variance vectors. Detailed information about amount of features extracted by every layer is available in the ***train_valid.html*** file in our codebase. Our model training process establishes $L = \mathbb{R}^{[0,\,1] \times [0,\,1]}$ as the latent control space. Overall, we have trained 6 models: for each of the AE, VAE, OAE architectures we trained a version conditioned on current joint angles and a version without conditioning. We realized early in our experiments that including conditioning on current joint velocities leads to extremely poor results, and chose not to pursue this direction. We do not describe the model training/validation results and obtained values of loss functions here, because they are irrelevant for the goal of our project. For every model type, we used validation loss to select the best parameter set.

8 (Continuous) controllability evaluation

For our (continuous) controllability evaluation we use 24 end-effector trajectories from the test subset, and try to make our manipulator follow them using only learned latent controls. It is important to note that instead of "theoretical" trajectories we use "empirical" ones, as recorded during simulated runs, because we can't guarantee that it is possible for our manipulator to follow the "theoretical" ones ideally. Our requirement of *continuous* controllability is formalized using $\mathrm{latent\_control\_eps}$ parameter which defines the allowed distance between every next latent control vector and the previous one. In our evaluations $\mathrm{latent\_control\_eps} \in \left\{2,\, 5 \cdot 10^{-1},\, 10^{-1}\right\}$. For every model, every value of $\mathrm{latent\_control\_eps}$ and every end-effector trajectory we perform the following experiment:

1. position the manipulator at the beginning of the trajectory
2. solve an optimization problem to find which vector $z_{\mathrm{next}} \in L$, if passed through a decoder and applied to our manipulator for one timestep, brings the end-effector to a position that is closest to the next position in the trajectory in terms of L_2 norm
3. take a step defined by vector z_{next} and re-name it to z_{prev}
4. repeat steps 2 and 3 with a modification to step 2: for every step after the first one, it is required that $\|z_{\mathrm{next}} - z_{\mathrm{prev}}\|_1 \leq \mathrm{latent\_control\_eps}$

These steps are repeated until either the trajectory is successfully traversed or the MSE between expected and factual end-effector positions is greater than 10^{-2}, in which case we consider that the experiment diverged. Since $L = \mathbb{R}^{[0,\,1] \times [0,\,1]}$, $\mathrm{latent\_control\_eps} = 2$ effectively means no continuity requirement. For every such experiment, we record if this experiment ended successfully, what part of the expected trajectory was traversed until divergence and what were the values of MSE between expected and factual end-effector positions on every step. We use Optuna library[21] for solving optimization problems in step 2. Code for this evaluation process is available in the ***optim_traverse.py*** file.

9 Results

We present our results in Table 4. Some plots of the end-effector trajectories acquired during evaluations are presented in Appendix C, all plots are available under ***optim_traverse_results*** directory in our codebase. First, as we mentioned in Section 7, our early experiments showed that conditioning the decoder on joint velocities leads to extremely poor results in terms of controllability. Table 4 shows that conditioning the decoder on joint angles also tends to significantly hinder controllability. Because of that, we chose not to evaluate *continuous* controllability of conditional autoencoders. We hypothesize that the reason for poor performance of conditional autoencoders in our project is that we intentionally use different trajectories for model training/validation and for controllability evaluation. As a result, the manipulator state component of decoder inputs is almost always out-of-distribution, which results in controls being inadequate for the trajectory being followed. Secondly, as we expected, introducing the continuity requirement, and further making it stricter, negatively affects the average portion of trajectories traversed before divergence. Even for $\mathrm{latent\_control\_eps} = 5 \cdot 10^{-1}$, which is a not very strict limitation (recall that $L = \mathbb{R}^{[0,\,1] \times [0,\,1]}$), "avg traversed trajectory portion" drops by 34 to 58 percentage points. However, these results do not allow us to conclude whether or not our models allow for continuous controllability in real-life

model	latent_control_eps	# successes	avg MSE	avg traversed trajectory portion
AE	2	14	0.0002	80%
VAE	2	12	0.0003	76%
OAE	2	9	0.0002	62%
cAE	2	0	0.0005	10%
cVAE	2	0	0.0010	16%
cOAE	2	0	0.0006	10%
AE	$5 \cdot 10^{-1}$	0	0.0010	22%
VAE	$5 \cdot 10^{-1}$	0	0.0007	32%
OAE	$5 \cdot 10^{-1}$	0	0.0006	28%
AE	10^{-1}	0	0.0015	7%
VAE	10^{-1}	0	0.0016	9%
OAE	10^{-1}	0	0.0010	7%

Figure 4: Results of continuous controllability evaluation. ”Model” column shows autoencoder type, models whose names start with ”c” are conditioned on current joint angles. ”# successes” shows the amount of trajectories (out of 24) successfully traversed from the beginning to end. ”avg MSE” shows MSE between expected end-effector Cartesian coordinates and actual ones, first averaged over steps within each individual trajectory and then over entire trajectories. ”avg traversed trajectory portion” shows the proportion of trajectory traversed before experiment termination due to successful finish or divergence, averaged over all trajectories.

scenarios and with a human operator, who may be able to get an ”intuitive” feeling for how controls from different parts of L affect the manipulator and act accordingly. Next, it doesn’t seem like orthogonal autoencoders (OAEs) reliably provide an improvement over AEs or VAEs in our experiments. It is still possible that latent space orthogonality is more beneficial for latent spaces of higher dimensionality, but testing this hypothesis is beyond the scope of our project. Finally, we noticed that ”circle-drawing” trajectories tend to be traversed far more successfully. We believe this is due to the fact that they are far smoother than the ”sine-drawing” ones.

10 Limitations

First of all, we acknowledge that our project is only a partial reproduction of the paper by Losey et al., and deviates from their work in many places, most importantly:

- trajectory task definitions
- robotic manipulator choice
- definition of control (Losey et al. use joint velocities, we use forces)

In general, Losey et al. didn’t go into much detail about their experimental setup, which limited our ability to reproduce their work.
Secondly, we observed that during the simulated runs our manipulator failed to accurately match ”theoretical” end-effector trajectories, so the demonstrations we collected should be considered ”noisy”, but we didn’t do anything to mitigate this problem.
Then, we acknowledge that selection of model architecture and training process parameters is a complex task, which is out of scope of our project. Because of that, we did only limited hyperparameter tuning, guided more by our own experience and intuition than by a systematic process.
Next, we were surprised by the fact that including conditioning on current joint angles affected the controllability results so gravely. Though we do have a hypothesis for why this may be the case, we didn’t debug this issue properly. However, even if in our case poor performance was caused by something other than manipulator states quickly going out-of-distribution, we believe this is going to be a problem in real-life applications and needs to be mitigated.
Finally, our results are directly influenced by how well the optimization problems in step 2 of continuous controllability evaluation algorithm are solved, but we didn’t have time to explore different approaches to optimization. The most important reason why we chose L_1 norm for our continuity requirement is that it is easy to model it in Optuna optimization routines. Besides that, the default

optimization mode used by Optuna is very computationally intensive, and so we had to run only a limited amount of optimization runs for every optimization problem, which could have affected our results.

11 Conclusion

In this project, we partially reproduced the work [1] by Losey et al., which explores the feasibility of embedding high-dimensional control commands into low-dimensional latent space. We also expanded on this work by experimenting with orthogonality constraints for autoencoders and discovered that it doesn't bring noticeable improvements in our setup. Finally, we formulated a *continuous* controllability requirement for the latent control space, which is stronger than controllability requirement formulated by Losey et al., and, we believe, more closely represents the demands of real-life human operation. Our experiments show that this requirement significantly hinders the ability of our optimization-based ”operator” to traverse the trajectories not seen by the model during training/validation stage.

References

[1] D. P. Losey, K. Srinivasan, A. Mandlekar, A. Garg, and D. Sadigh. Controlling assistive robots with learned latent actions. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 378–384. IEEE, 2020.

[2] V. Maheu, P. S. Archambault, J. Frappier, and F. Routhier. Evaluation of the jaco robotic arm: Clinico-economic study for powered wheelchair users with upper-extremity disabilities. In *2011 IEEE International Conference on Rehabilitation Robotics*, pages 1–5. IEEE, 2011.

[3] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dymesich. Fetch and freight: Standard platforms for service robot applications. In *Workshop on autonomous mobile service robots*, 2016.

[4] R. Bischoff, J. Kurth, G. Schreiber, R. Koeppe, A. Albu-Schäffer, A. Beyer, O. Eiberger, S. Haddadin, A. Stemmer, G. Grunwald, et al. The kuka-dlr lightweight robot arm-a new reference platform for robotics research and manufacturing. In *ISR 2010 (41st international symposium on robotics) and ROBOTIK 2010 (6th German conference on robotics)*, pages 1–8. VDE, 2010.

[5] P. M. Kebria, S. Al-Wais, H. Abdi, and S. Nahavandi. Kinematic and dynamic modelling of ur5 manipulator. In *2016 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 004229–004234. IEEE, 2016.

[6] W. Wang, D. Yang, F. Chen, Y. Pang, S. Huang, and Y. Ge. Clustering with orthogonal autoencoder. *IEEE Access*, 7:62421–62432, 2019. doi:10.1109/ACCESS.2019.2916030.

[7] D. Rakita, B. Mutlu, and M. Gleicher. Relaxedik: Real-time synthesis of accurate and feasible robot arm motion. In *Robotics: Science and Systems*, pages 26–30. Pittsburgh, PA, 2018.

[8] N. Mehr, R. Horowitz, and A. D. Dragan. Inferring and assisting with constraints in shared autonomy. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 6689–6696. IEEE, 2016.

[9] A. Broad, T. Murphey, and B. Argall. Learning models for shared control of human-machine systems with unknown dynamics. *arXiv preprint arXiv:1808.08268*, 2018.

[10] A. Tung, J. Wong, A. Mandlekar, R. Martín-Martín, Y. Zhu, L. Fei-Fei, and S. Savarese. Learning multi-arm manipulation through collaborative teleoperation. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9212–9219. IEEE, 2021.

[11] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi:10.1109/IROS.2012.6386109.

[12] M. M. Contributors. MuJoCo Menagerie: A collection of high-quality simulation models for MuJoCo, 2022. URL `http://github.com/deepmind/mujoco_menagerie`.

[13] S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, N. Heess, and Y. Tassa. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020.

[14] M. A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *Aiche Journal*, 37:233–243, 1991.

[15] L. Pinheiro Cinelli, M. Araújo Marins, E. A. Barros da Silva, and S. Lima Netto. *Variational Autoencoder*, pages 111–149. Springer International Publishing, Cham, 2021. ISBN 978-3-030-70679-1. doi:10.1007/978-3-030-70679-1_5. URL `https://doi.org/10.1007/978-3-030-70679-1_5`.

[16] A. Subramanian. Pytorch-vae. `https://github.com/AntixK/PyTorch-VAE`, 2020.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[18] W. Falcon and The PyTorch Lightning team. PyTorch Lightning, 3 2019. URL `https://github.com/Lightning-AI/lightning`.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] L. N. Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[21] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Appendices

A Setup instructions

Versions of all dependencies used in our project are specified in requirements.txt file in our code base. Note, however, that it may have excessive entries, not required for running our project. Running our code requires some changes to the dm_control code base, here we provide setup instructions:

- Note the directory where dm_control was installed, for example, virtual_env/lib/python3.10/site_packages/dm_control
- Copy ur5e.py, ur5e.xml files and ur5e_assests directory from our code base to the dm_control/suite directory
- Replace the dm_control/suite/__init__.py file with __init__.py from our codebase
- Replace the dm_control/suite/base.py file with base.py from our codebase
- Replace the dm_control/suite/rl/control.py with control.py from our codebase

B End-effector trajectory plots: data collection

Note that these plots are in non-linear time: since we allowed the simulated manipulator to take as many steps as needed to arrive at every next trajectory point, we then had to align ”theoretical” (”expected”) trajectory steps with ”empirical” ones in such a way that they are comparable on our plots. This explains irregularities in ”theoretical” (”expected”) trajectories plotted here.

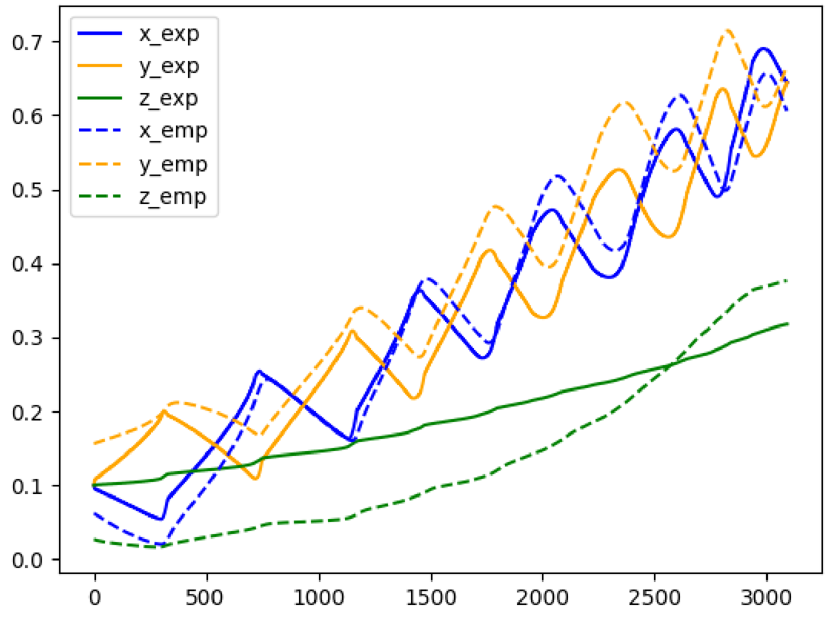

Figure 5: Sine-drawing trajectory, data/0.png, ”theoretical” trajectories in solid lines, ”empirical” trajectories in dotted lines, non-linear time to align the two

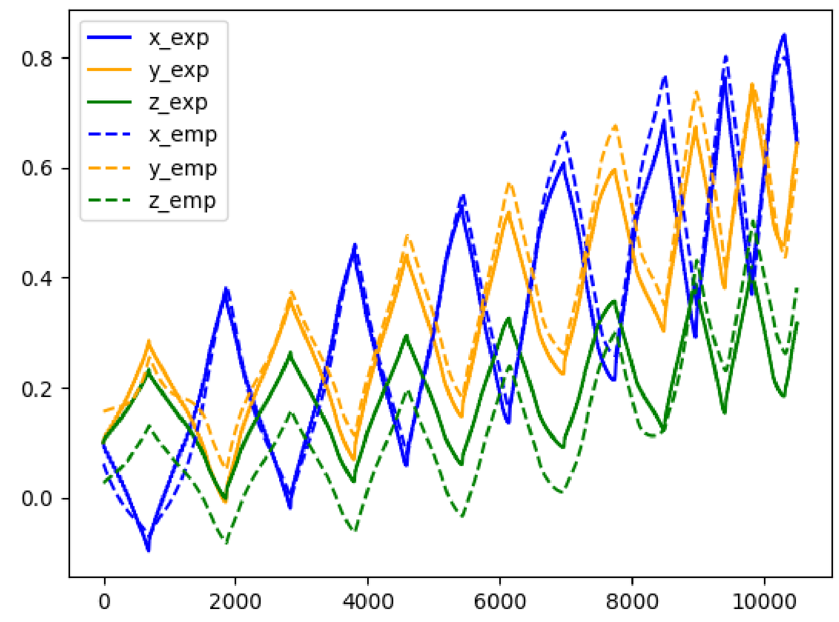

Figure 6: Sine-drawing trajectory, data/29.png, ”theoretical” trajectories in solid lines, ”empirical” trajectories in dotted lines, non-linear time to align the two

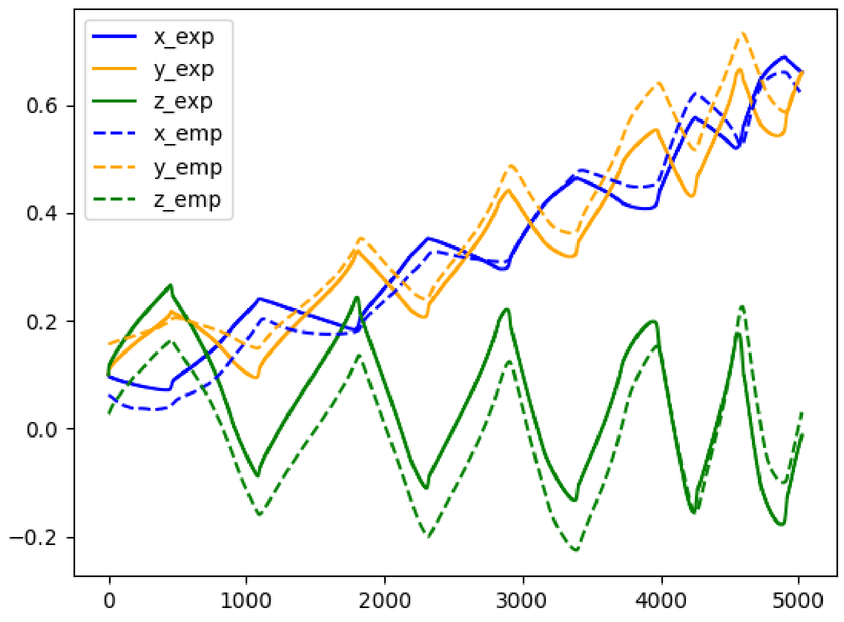

Figure 7: Sine-drawing trajectory, data/87.png, ”theoretical” trajectories in solid lines, ”empirical” trajectories in dotted lines, non-linear time to align the two

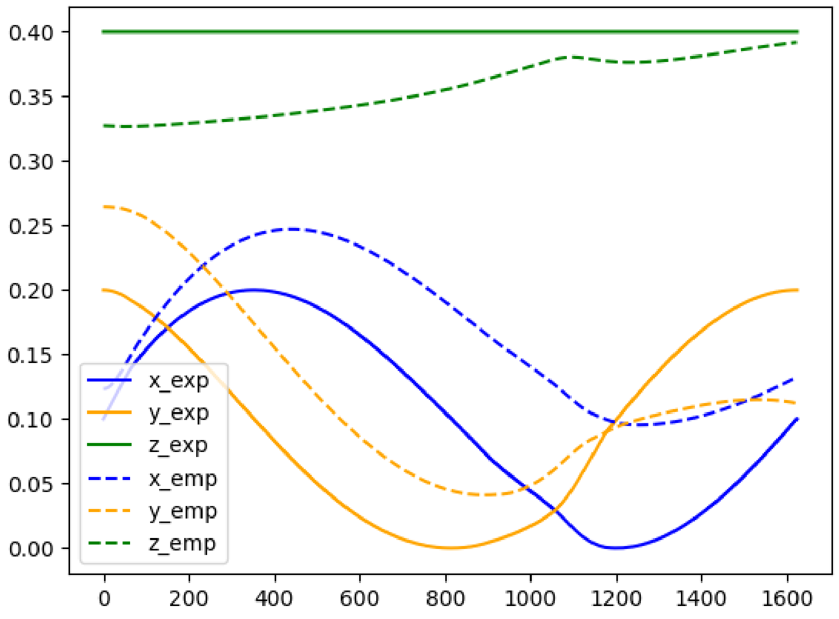

Figure 8: Circle-drawing trajectory, data/108.png, ”theoretical” trajectories in solid lines, ”empirical” trajectories in dotted lines, non-linear time to align the two

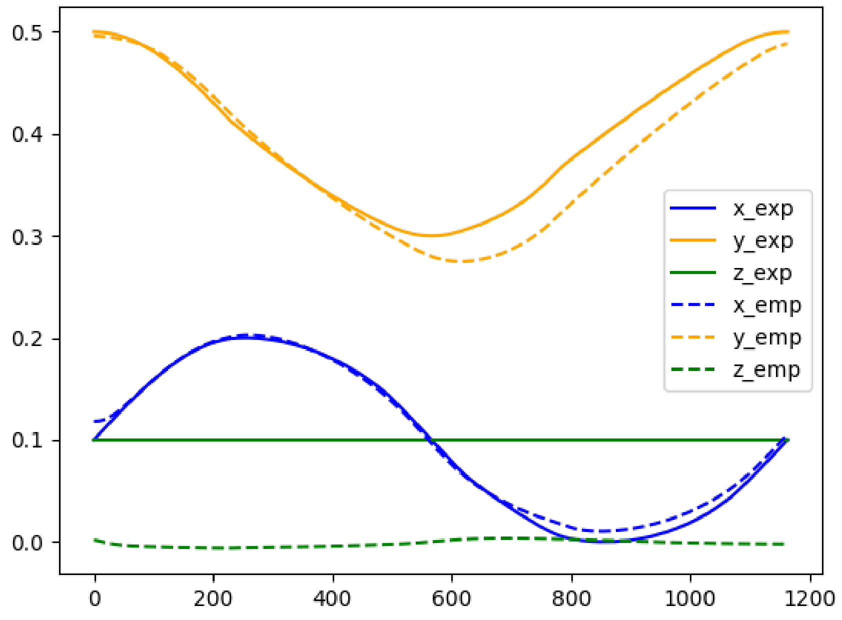

Figure 9: Circle-drawing trajectory, data/116.png, ”theoretical” trajectories in solid lines, ”empirical” trajectories in dotted lines, non-linear time to align the two

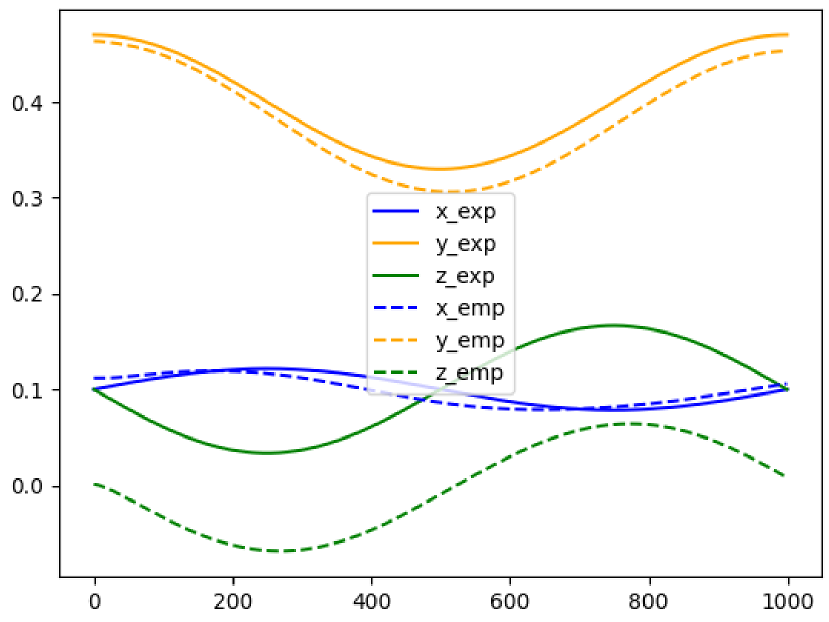

Figure 10: Circle-drawing trajectory, data/147.png, ”theoretical” trajectories in solid lines, ”empirical” trajectories in dotted lines, non-linear time to align the two

C End-effector trajectory plots: optimization-based traversing with low-dimensional controls

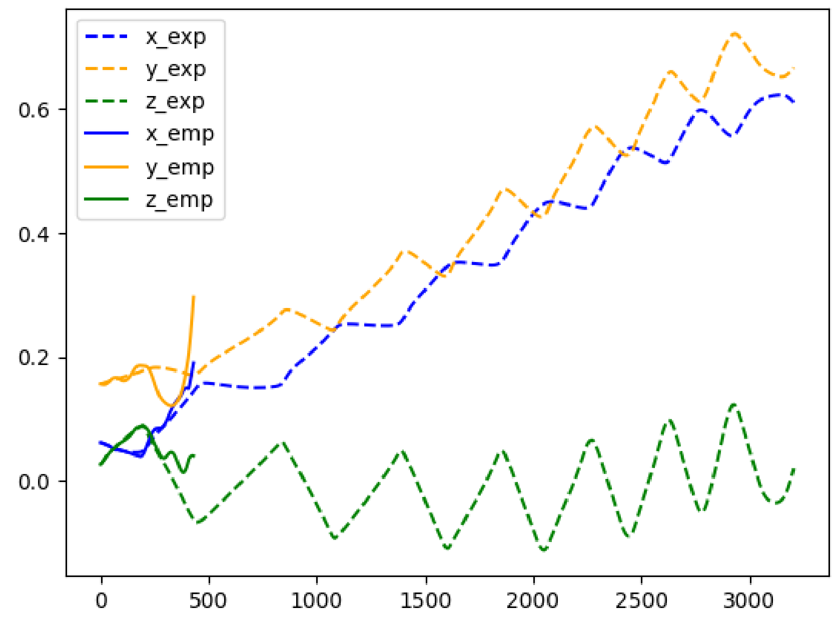

Figure 11: Sine-drawing trajectory, VAE with $\mathrm{latent\_control\_eps} = 5 \cdot 10^{-1}$, optim_traverse_results/without_qstate/vae_tanh_5e-1/79.joblib.png, "expected" trajectories in dotted lines, "empirical" trajectories in solid lines

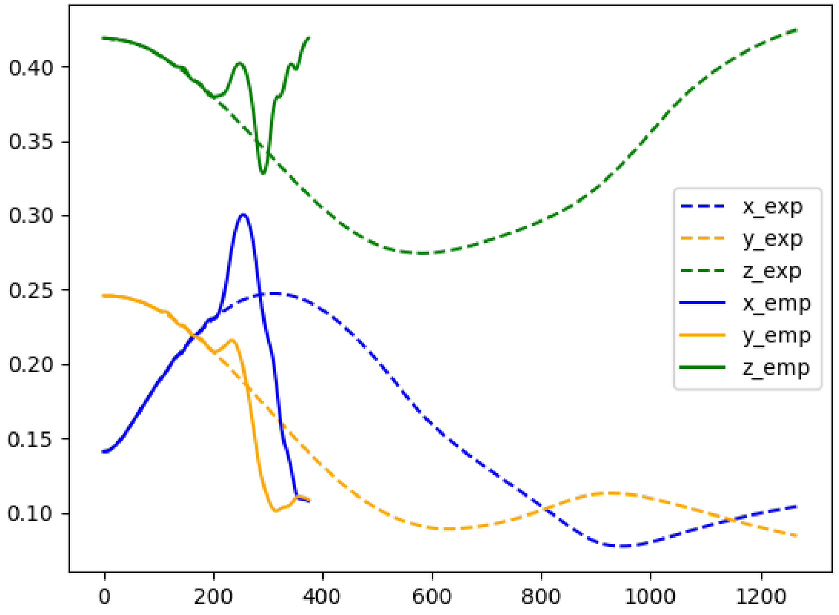

Figure 12: Circle-drawing trajectory, VAE with $\mathrm{latent\_control\_eps} = 5 \cdot 10^{-1}$, optim_traverse_results/without_qstate/vae_tanh_5e-1/110.joblib.png, ”expected” trajectories in dotted lines, ”empirical” trajectories in solid lines

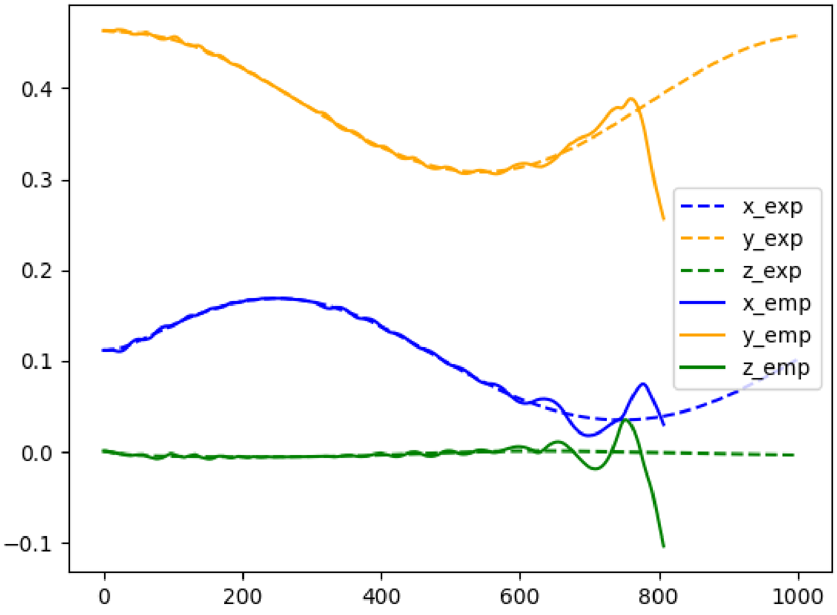

Figure 13: Circle-drawing trajectory, VAE with $\mathrm{latent\_control\_eps} = 5 \cdot 10^{-1}$, optim_traverse_results/without_qstate/vae_tanh_5e-1/140.joblib.png, ”expected” trajectories in dotted lines, ”empirical” trajectories in solid lines

D Contributions

Sergey Protserov: code for ”empirical” trajectory recording, code for model training/validation, code for optimization-based continuous controllability evaluation, running model training/validation, running continuous controllability evaluation experiments Yujie Wu: MuJoCo model preparation (some models ended up not being used in the final version of the project), code for ”theoretical” trajectory generation, code for running rendered simulations with ability to control joints through GUI (ended up not being used in the final version of the project), running model training/validation, running continuous controllability evaluation experiments, code for aggregating continuous controllability evaluation results