A Motion Capture and Imitation Learning-based Approach to Robot Control

A Preprint

Peteris Racinskis[*†] peteris.racinskis@edi.lv

Janis Arents[*†] janis.arents@edi.lv

Modris Greitans[†] modris_greitans@edi.lv

June 26, 2022

Abstract

Imitation Learning is a discipline of Machine Learning primarily concerned with replicating observed behavior of agents known to perform well on a given task, collected in demonstration data sets. In this paper we set out to introduce a pipeline for collecting demonstrations and training models that can produce motion plans for industrial robots. Object throwing is defined as the motivating use case. Multiple input data modalities are surveyed, and motion capture is selected as the most practicable. Two model architectures operating autoregressively are examined – feedforward and recurrent neural networks. Trained models execute throws on a real robot successfully, and a battery of quantitative evaluation metrics is proposed. Recurrent neural networks outperform feedforward ones in most respects, but this advantage is not universal or conclusive. The data collection, pre-processing and model training aspects of our proposed approach show promise, but further work is required in developing Cartesian motion planning tools before it is applicable for production applications.

Keywords Imitation Learning · Motion capture · Robotics · Artificial neural networks · RNN

1 Introduction

Manipulator arms and other types of robots have become ubiquitous in modern industry and their use has been proliferating for decades, yet even now the primary method for programming these devices remains procedural code, hand-crafted by specially trained technicians and engineers. This significantly increases the cost and complexity of commissioning process nodes that utilize industrial robots. To this end, various alternative approaches grounded in machine learning have been proposed. Perhaps the most commonly studied are methods that fall under the umbrella of reinforcement learning, which optimize an agent acting on its environment through the use of an explicitly specified reward function [Sutton and Barto, 2018]. While offering strong theoretical guarantees of convergence, they require interaction with the environment for learning to occur. Furthermore, in practical settings the reward function for many tasks is very sparse, leading to large search spaces and inefficient exploration [Hester et al., 2018]. Finally, the reward function for a task may be unknown or difficult to specify analytically [Abbeel and Ng, 2004].

Imitation learning is a broad field of study in its own right that promises to overcome some or all of the aforementioned challenges, provided that it is possible to obtain a corpus of demonstrations showing how the task is to be accomplished. When dealing with industrial robotics, this is often the case as the tasks we wish to execute are ones that human operators already routinely perform. Therefore, we have developed an approach for recording actions taken by humans in the physical environment, programmatically producing a training data set structured in the form of explicit demonstrations augmented with additional, extrapolated state inputs and control signals, and training artificial neural network models to reproduce the trajectories therein as motion plans.

Given the inherent trade-off between task specificity and required model complexity when it comes to observation data modality and model output, motion capture has been selected as a convenient middle ground. It does not require a

[*]Robotics and Machine Perception Laboratory

[†]Institute of Electronics and Computer Science, Riga, Latvia

simulated environment as approaches that involve virtual reality do [Zhang et al., 2018, Dyrstad et al., 2018], obviates any issues that may come with having to learn motions in robot configuration space by operating in Cartesian coordinates, and does not necessitate the use of complex image processing layers as found in models with visual input modalities [Liu et al., 2018, Zhang et al., 2018]. The main drawbacks of this approach are that it requires motion capture equipment, which is more expensive than conventional video cameras, and Cartesian motion plans need an additional conversion step into joint space trajectories before it is possible to execute them on a real robot. Given the intended application of this system – programming robots to perform tasks in an industrial environment – the advantages of compact models, rapid training times and results that are possible to evaluate ahead of time were deemed to outweigh the drawbacks.

To help guide the development process and serve as a means of evaluation, a sample use case was selected – object throwing with the goal of extending the robot's reach and improving cycle times. In particular, the task of robotically sorting plastic bottles at a recycling plant was to be augmented with a throwing capability. While on the surface the task is almost entirely defined by elementary ballistics that can be programmed explicitly, it serves as a convenient benchmark for robot programming by way of demonstrations. The task was deemed sufficiently non-trivial when considered from the perspective of a Markov decision process or time-series model only given limited information about system state at any given time step, while also being suitable for intuitive evaluation by human observers due to its intuitively straightforward nature. Furthermore, the relationship between release position, velocity and target coordinates provides an obvious way to quantitatively evaluate model performance against training and validation data – extrapolated throw accuracy.

2 Preliminaries

In this article, an *agent* can be taken to mean the part of the system that acts based on state or observation information s in an *environment*, according to a *policy* π which is specified by a parametric *model* – a function $\pi_\theta(s)$ with parameters θ tuned in optimization and produces an output *action* a. In practice this means that references to the agent, model and policy are almost interchangeable in most contexts. A formalism common in both reinforcement and imitation learning contexts is the *Markov decision process* (MDP), formally given by [Attia and Dayan, 2018]

$$MDP = (S, A, T, R, I) \tag{1}$$

where S is the set of states s, A is the (formally discrete) set of actions a, $T : S \times A \to S$ is a state transition function that encapsulates the environment, $R : S \to \mathbb{R}$ is a reward function associated with each state and $I = p\,(s_0 \in S)$ represents the initial state distribution. In many imitation learning-related cases a reward function need not specified or considered. Moreover, if one is willing to break with the strict formal definition and give up the use of mathematical tools defined only on finite probability distributions, continuous action spaces can also be considered.

One important characteristic of the MDP is that it is history-agnostic – the future state distribution of the system is uniquely defined by its current state. While technically true for physical environments, it is often the case that instead of a complete state representation vector $\mathbf{s}$ we are instead operating on a more limited *observation* $\mathbf{o}$ (often interchangeably referred to as $\mathbf{s}$ for brevity), where each element o_i is given by some function $f_i(\mathbf{s})$. It is therefore possible that historical observations contain information about hidden system state variables even assuming the MDP formalism holds for the underlying environment. An example of this can be the case when individual observations contain only the current position of an object but not its derivatives, as in video data. In such situations it may prove beneficial to break with the formalism further, by redefining the policy to operate on sequences of $k + 1$ previous states/observations

$$t, k, n \in \mathbb{N}, \pi_\theta : \{(s_n)_{t-k}^{t}\} \to A \tag{2}$$

which, by allowing for input sequences of variable length, may become a function defined on the entire known state history:

$$\pi_\theta : \{(s_n)_{1}^{t}\} \to A \tag{3}$$

These adjustments allow for the employment of sequence-to-sequence predictor architectures also studied in other areas of machine learning such as recurrent neural networks (RNNs) [Rumelhart et al., 1985] and transformers [Vaswani et al., 2017]. Finally, if continuous action spaces are permitted it is no great stretch to also consider formats where the action corresponds to a predicted future state to be used for static motion planning or in a feedback controller

$$\pi_\theta : \{(s_n)_{1}^{t}\} \to S \tag{4}$$

which, when running the model on its own output, is equivalent to sequence building tasks encountered in domains such as text generation.

3 Related Work

In its simplest and most straightforward form – sometimes referred to as *behavioral cloning* – imitation learning can be reduced to a classification or regression task. Given a set of states and actions or state transitions produced by an unknown expert function, a model (*policy*) is trained to approximate this function. Even with very small parameter counts by modern standards, when combined with artificial neural networks this method has demonstrated some success as far back as the 1980s, provided a task with simple dynamics such as keeping a motor vehicle centered on a road [Pomerleau, 1989].

However, it has since become apparent that pure behavioral cloning suffers from distribution shift – a phenomenon whereby the distribution of states visited by the policy diverges from that of the original training data set by way of incremental error, eventually leading to poor predictions by the model and irrecoverable deviation from the intended task. To improve the ability of policies to recover from this failure mode, various more complex approaches to the task have been proposed. One major direction of research has been the use of inherently robust, composable functions known as *dynamic motion primitives*, employing systems of differential equations and parametric models such as Gaussian basis functions to obviate the problem of distribution shift entirely – ensuring convergence towards the goal through the explicit dynamics of the system [Pastor et al., 2009]. Though showing promising results, it must be noted that the model templates used are quite domain-specific. Others have attempted to use more general-purpose models in conjunction with interactive sampling of the expert response to compensate for distribution shift [Ross et al., 2011]. While theoretically able to guarantee convergence, the major drawback of such methods is that an expert function is required that can be queried numerous times as part of the training process. This severely restricts applicability to practical use cases, since it is impossible to implement when only given a set of pre-recorded demonstrations.

A different means of handling this problem is given by the field of *inverse reinforcement learning* [Abbeel and Ng, 2004]. Rather than attempting to model the expert function directly, it is assumed that the expert is itself acting in accordance with an unknown reward function. An attempt therefore is made to approximate this reward in a way that explains the observed behavior. From there this becomes a classical reinforcement learning problem, and any training method or model architecture developed in the space of this adjacent field of study can be employed. Where early approaches made certain assumptions about the form of this reward function – such as it being linear – more recent work has proposed that generative adversarial networks be used where the discriminator can approximate the class of all possible reward functions fitting the observations over an iterative training process [Ho and Ermon, 2016].

Perhaps the most promising results, however, come from the sequence modelling domain. Recurrent neural networks show up in earlier scientific literature periodically, such as in predicting a time-series of robot end effector loads in an assembly task [Scherzinger et al., 2019] and learning latent action plans from large, uncategorized play data sets [Lynch et al., 2020]. But current state-of-the-art performance across a wide variety of sequence prediction tasks – among them imitation learning in a robotics context – is given by combining a large, universal transformer model with embedding schemes specific to various data modalities [Reed et al., 2022]. These results strongly suggest that structuring one's approach to be compatible with general-purpose sequence predictor algorithms is preferable for ensuring its longevity.

When it comes to the use of motion capture data, previous work in the robotics and imitation learning corner is quite sparse, possibly due to the costly and specialized nature of the equipment involved. One previously considered direction has been the use of consumer-grade motion tracking equipment for collecting demonstrations [Jha et al., 2017]. Unlike our work, they employ a single, relatively inexpensive sensor unit to generate the motion tracking data, and the main focus of the work is on accurate extraction of the demonstrations rather than modelling and extrapolation – which is left as something of an afterthought, with cluster and k-nearest interpolation methods used for inference. Outside the imitation learning space, related work has been done in human motion modelling utilizing RNNs trained on motion capture data [Fragkiadaki et al., 2015]. The main difference between research in this direction and our work lies in the fact that we are looking for ways to translate human motion into paths taken by a robot, and do not consider the human kinematic model. Additionally, we place an emphasis on extracting particular implicit signals such as release timing and desired target coordinates.

Meanwhile, optimal execution of object throwing tasks has been previously studied in a reinforcement learning context, such as with *TossingBot* [Zeng et al., 2020]. Their approach is radically different from ours, in that it utilizes a reinforcement learning policy to predict a set of grasp and throw parameters. These serve as inputs for grasping and throwing motion primitives. The throwing primitive itself is not learned, but provided with a release position and velocity, which it then executes. Initial estimates are produced analytically, which are then combined with learned residuals. The main distinction between their work and ours, however, is that of purpose. We do not seek to perfect an approach to throwing in particular, but rather establish a method for using imitation learning to program industrial robots. There is an expectation of being able to further adapt them for a variety of tasks – with throwing serving primarily as a convenient benchmark for progress.

4 Proposed approach

Three crucial decisions need to be made when devising an imitation learning-based approach to robot control:

- Data collection – what will serve as the expert policy? How will data be obtained?
- Model architecture – what type of model template will be used to learn the policy? What are its inputs and outputs, what pre- and post-processing steps will these formats call for?
- Control method – how will model outputs be used in robot motion planning or feedback control?

Considering that the goal of this paper is not to examine any of these sub-problems in detail but rather to come up with a holistic approach to combine all of them, trade-offs that affect more than one step at a time need to be considered. Recording of human performances as the expert data set is also a constraint imposed by the objective we set out to accomplish. Therefore, when selecting the demonstration data modality, three possibilities were examined:

- raw video – using conventional cameras to record a scene and use these observations as model inputs directly;
- motion capture – obtain effector and scene object pose (position and orientation) data with specialized motion capture equipment that consists of multiple cameras tracking highly reflective markers affixed to bodies of interest;
- simulation – using a simulated environment in conjunction with virtual reality (VR) or other human-machine interface to obtain either pose data as with motion capture, or record the configurations of a robot model directly.

Raw video is the most attractive approach from a material standpoint – it does not involve motion capture or VR equipment, does not require simulated environments, and in principle presents the possibility of a sensor-to-actuator learned pipeline, if the video data were used directly in the model's input to predict joint velocities. In practice, however, not only does using image data call for the employment of more complex model architectures such as convolutional neural networks, images are inherently 2-dimensional representations of 3-dimensional space, leading to ambiguities in the data – and one quickly runs into difficult issues such as context translation that confound the issue further [Liu et al., 2018].

In contrast, unambiguous pose information is readily obtained with motion capture or simulation. Simulating a robot and collecting data in joint space allows for a simple final controller stage. But it very tightly couples the trained model to a specific physical implementation, and reasoning about or debugging model outputs obtained this way is not straightforward. Thus, the main comparison is to be drawn between object pose data as obtained in a simulation and with motion capture. Assuming motion capture equipment is available, setting the scene up for collecting demonstrations is a matter of outfitting the objects of interest with markers and defining them as rigid bodies to be tracked. However, there is a degree of imprecision in the observations, and information about different rigid bodies is available at different instants in time.

Simulation allows recording exact pose information directly at regular time intervals, but setting up the scene necessitates creating a virtual environment where the task can be performed. In addition, for realistic interactions a physics simulation and immersive interface such as VR is required, but even this does not present the human demonstrator with the haptic feedback of actually performing the task in the real world. As such we decided that the benefits of simple setup and inherently natural interaction with the physical scene outweigh the precision and cost advantages of simulated environments, and selected motion capture as the source of demonstration data. For extracting additional information specific to the throwing task, we also track the object thrown and use its pose information to annotate each observation with target coordinates extrapolated from its flight arc and a release timing signal determined by heuristics described below.

The next crucial decision to make is what the input and output spaces of the model will be. Setting aside the additional control parameters (target coordinates and release signal), when provided with a sequence of pose observations the main options are:

- Pose-Pose – predict the pose at the next discrete time step from the current observation;
- Pose-Derivatives – predict velocities or accelerations as actions;
- Joint state-Joint target – analogous to pose-pose, but in joint space;
- Joint state-Joint velocity – analogous to pose-derivates, but in joint space.

Approaches crossing the Cartesian-joint space domain boundary were not considered, as the model would be required to learn the inverse kinematics of the robot. The advantages of having a model operate entirely in joint space would be seen

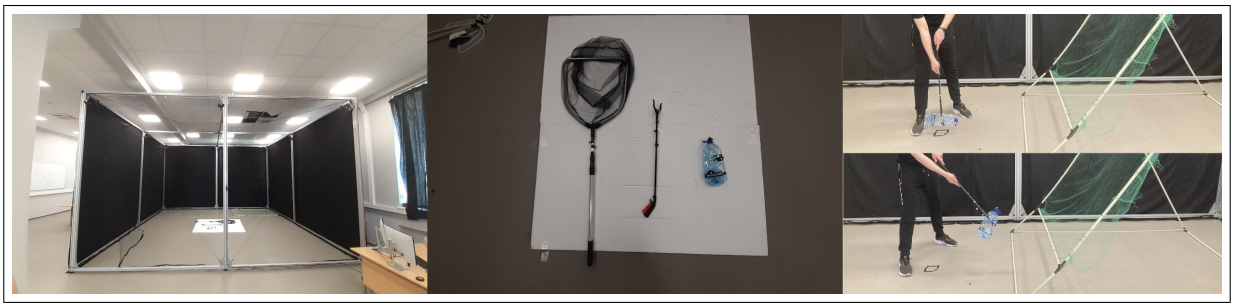

Figure 1: Motion capture equipment and process.

at the next stage – integration with the robot controller – as motion planning to joint targets is trivial. However, training such a model would require mapping the demonstrations to robot configuration space by solving inverse kinematics on them, much the same as with a model operating in Cartesian space. While computationally less expensive at runtime, this approach is more difficult to reason about or explain, complicating the hyperparameter discovery process. Any model obtained this way would also be tightly coupled to a specific robot. Hence, all models were trained in Cartesian space, with the motion planning step left for last.

Outputting pose derivatives (linear and angular velocities) is advantageous as it enables the use of servo controllers, should implementations of such exist, and input-output timing constraints are less stringent than they would be in a strict sequence model that assumes a constant time step. However, a model-in-the-loop control scheme is required – forward planning is not possible without a means to integrate model outputs with realistic feedback from the environment. Moreover, for training the model derivatives of the pose need to be estimated from sequential observations, highly susceptible to discretization error.

Predicting the future pose allows for using discrete-time pose observations directly, and, assuming a pose following controller is able to keep within tolerance, forward planning becomes a matter of running the model autoregressively (using its own outputs for generating a sequence). This approach naturally lends itself to the employment of recurrent model architectures. The main drawback is that pose derivative information is encoded in the time step, which imposes constraints on observation and command timing, complicating use of such a model in a real-time feedback manner. It also requires that a method for accurate time parametrization of Cartesian motion plans is available. Ultimately this is the approach that was selected, because it neatly decouples the model and its generated plans from a specific robot implementation, and enables us to generate, visualize and numerically evaluate motion plans ahead of time. Execution then becomes a matter of conventional motion planning given a Cartesian path.

5 Implementation

The system devised in this paper can be broken down into three main sections – the collection of observations in the physical environment (5.1), a pipeline for turning raw observation data into structured demonstrations with additional control signals (5.2) and neural network model implementations (5.3). Two general ways to evaluate system performance and obtain design feedback were employed. First, qualitative observations of the generated trajectories were made using spatial visualization in a virtual environment, followed by execution on simulated and real robots (5.4). When satisfactory performance was attained, a series of quantitative metrics were computed for comparing system outputs with training and validation datasets on different model architectures and hyperparameter sets (5.5).

5.1 Data collection

All demonstrations were recorded using *OptiTrack* equipment that consists of a set of cameras, highly reflective markers to be attached to trackable objects and the *Motive* software package which handles pose estimation and streaming. As shown in fig. 1, a cage with 8 cameras installed serves to hide external sources of specular reflections from the cameras and confine moving objects such as drones from leaving the scene. To simulate an industrial robot end effector, a gripping hand tool was equipped with markers. Since the motivating application calls for throwing plastic bottles, one such was also marked to serve as basis for extrapolating target coordinates and identify the release point in a demonstration. A start point is laid down on the floor to serve as a datum for automatically separating the recorded demonstrations during pre-processing. Holding the effector stand-in here for at least a second before each demonstration provides a consistent signal that can be identified programmatically.

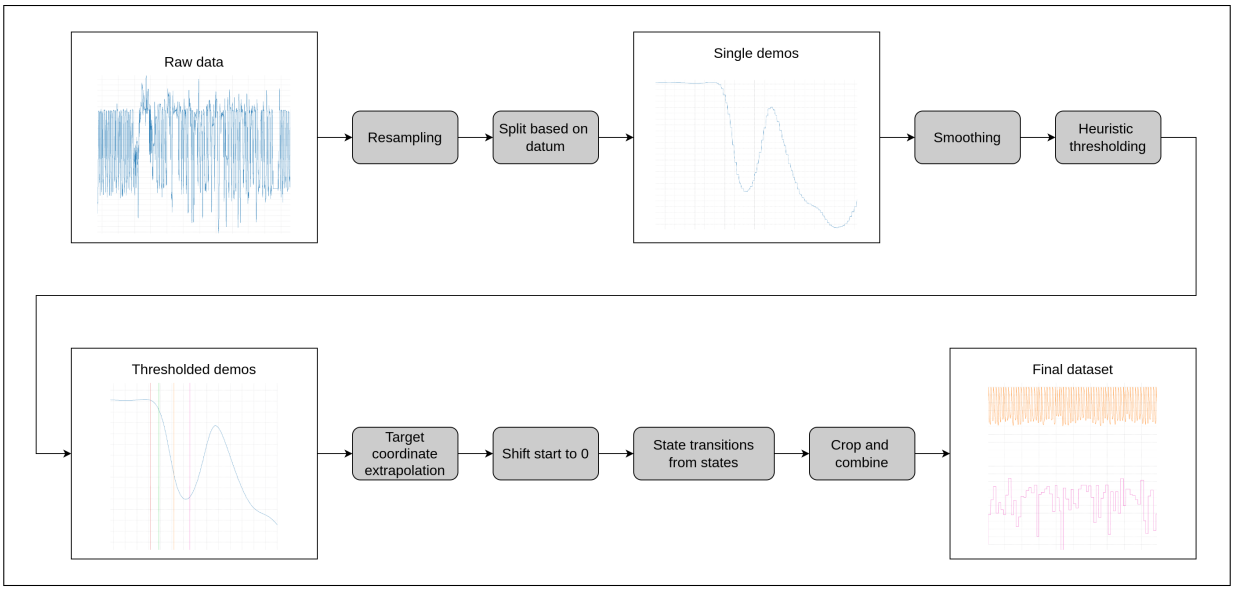

Figure 2: Pre-processing pipeline overview.

Nets were used to catch the thrown bottle and prevent damaging the markers. These were also marked to aid in delimiting the ballistic segment of the thrown object's flight. While in principle it is possible to detect an object in freefall by observing its acceleration, for the purposes of trajectory extrapolation this was deemed too prone to error – some impacts radically change the horizontal component of the object's velocity without breaking its fall, and the acceleration of the empty bottles proved to be affected by air resistance to a significant degree owing to their low terminal velocity.

In software, rigid body definitions were created for all object to be tracked. These were then streamed on the local network, relayed over *Robot Operating System* (ROS) topics corresponding to each rigid body using a pre-existing package ROS and recorded into a bag file to be converted into a .csv data set. Over the course of this project, two data sets with roughly 150 and 50 demonstrations each were collected. The first was used in the development process but proved to contain throws beyond the capabilities of the robot hardware used. Therefore, a second data set of throws with less pronounced swings was produced to fit within the working volume of the robot arm. A notable feature of the second data set is that initial orientations were randomized to a much lesser degree ($\pm 30°$) than in the original corpus ($\pm 90°$), and the range of estimated target coordinates for the throws was also more tightly constrained ($1.5...1.8$m along the x-axis as opposed to $1.6...3.6$m), likely contributing to the better estimated throw accuracy metrics as illustrated in 6.

5.2 Pre-processing, extraction of implicit control signals

Figure 2 provides a schematic overview of the main steps involved in the data preparation process. For illustrative purposes, graphs of the effector x-coordinate with respect to time are used, but the final picture also contains the estimated target coordinate of each throw.

After each recording session, an entirely unstructured corpus is available. It contains timestamped pose observations collected at whatever intervals they happened to have been generated by the tracking software. Each observation only contains information about a single tracked body. A peculiarity of this particular recording method is that observations about each body tend to arrive in bursts:

$$s^a_{t1}, s^a_{t2}, ..., s^b_{tm-1}, s^b_{tm} \tag{5}$$

where $s^a_{t\in\mathbb{R}}$ corresponds to the observed state of object a at time t. Thus, to obtain sequential data in discrete time, it is necessary to resample the positions and orientations of all relevant bodies at a constant time step:

$$(s^a_1, s^b_1, s^c_1), ..., (s^a_k, s^b_k, s^c_k) = s_1, ..., s_k \tag{6}$$

A sampling frequency of $100Hz$ was selected to obtain spatial precision on the order of centimeters given the velocities involved. Fortuitously, the intervals between observed bursts in the actual data are generally on this order as well.

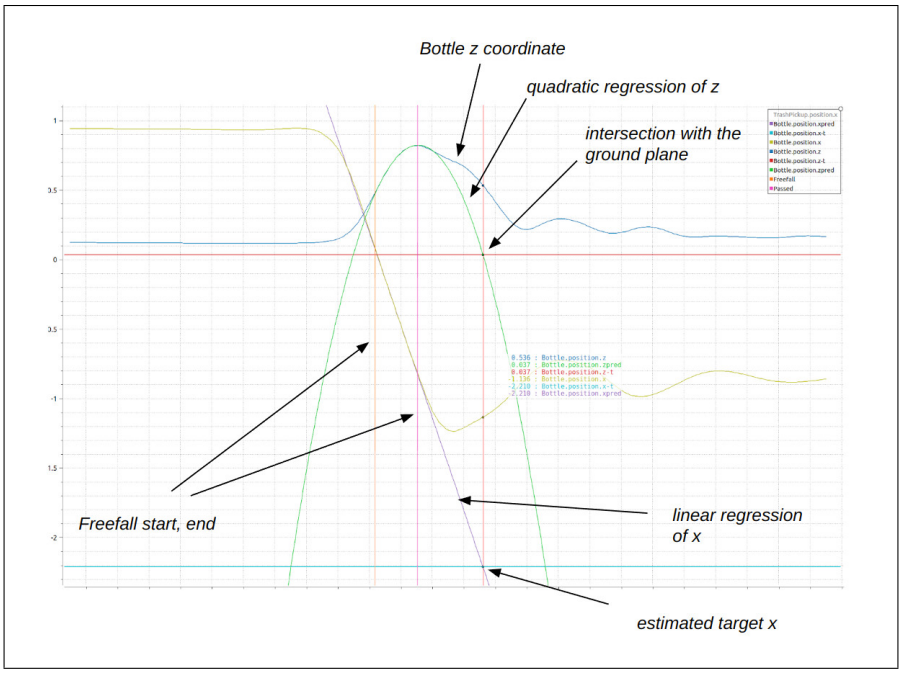

Figure 3: Estimation of the throw target coordinates.

Resampling is accomplished using forward fill interpolation, which introduces a slight sawtooth oscillation, but this is removed in the subsequent smoothing step along with other discontinuities.

After resampling, individual demonstrations are separated using the aforementioned consistent starting position. This may vary from recording session to recording session, so it needs to be identified and configured manually. Specifically, the condition used to identify demonstration start points is as follows:

$$\begin{aligned} t_{start} \in [t]_1^k : t < t_{start} \wedge (t_{start} - t < \Delta t_{max}) \Rightarrow \\ \Rightarrow x_t \in (x_0 - \delta x, x_0 + \delta x), y_t \in (y_0 - \delta y, y_0 + \delta y) \end{aligned} \tag{7}$$

Which is to say that every observation no more than Δt steps before t_{start} has to lie within $\pm\delta x, \pm\delta y$ of the datum (x_0, y_0). Each demonstration then consists of observations

$$(s_{t_{start}}, ..., s_{t_{start}+steps}), steps \in \mathbb{N} \tag{8}$$

The constants $\Delta t, \delta x, \delta y, steps$ are determined experimentally to produce consistent observations for the particular task and may require tuning if the manner in which demonstrations are recorded changes. The newly split demonstrations are saved in separate files, thus allowing demonstrations generated in different sessions to be processed at once.

As further thresholding steps rely on estimates of pose derivatives, a smoothing step is first employed to remove noise and the slight sawtooth oscillation induced by the resampling step. This is accomplished by applying a rolling average kernel to the trajectory. Nevertheless, it was found that any estimate of derivatives would exacerbate what noise there was in the dataset, leading to inconsistent results. So a simplified estimator $\overline{x'_t}$, combining a correlate of the derivative x'_t with a rolling average filter, was used with satisfactory results:

$$x'_t \propto \overline{x'_t} = \sum_{i=t}^{t+m} x_i - \sum_{i=t-m}^{t} x_i \tag{9}$$

This was then used to compute threshold functions corresponding with the start of motion, actuator release and unfettered freefall of the bottle:

$$f_{moving}(t) = \begin{cases} 0 \text{ if } \forall u < t, \|\overline{[\boldsymbol{r}_{Effector}(u)]'_u}\| < \overline{v}_{moving} \\ 1 \text{ otherwise} \end{cases} \tag{10}$$

Table 1: Model hyperparameters

Parameter	Feedforward	RNN
Architecture	2 fully-connected hidden layers, ReLU	GRU, fully connected linear output
Parameter counts	128-1024 perceptrons per layer	128-512 perceptrons in the unit
Training epochs	20-100	300-1200
Batch size	64	32
Optimizer	Adam	
Learning rate	10^{-4}	$10^{-3}, 10^{-4}$

$$f_{release}(t) = \begin{cases} 0 \text{ if } \forall u < t, \overline{[\|\boldsymbol{r}_{Bottle}(u) - \boldsymbol{r}_{Effector}(u)\|]}''_u < \overline{a}_{release} \\ 1 \text{ otherwise} \end{cases} \tag{11}$$

$$f_{freefall}(t) = \begin{cases} 0 \text{ if } \forall u < t, \overline{[\|\boldsymbol{r}_{Bottle}(u) - \boldsymbol{r}_{Effector}(u)\|]}'_u < \overline{v}_{freefall} \\ 1 \text{ otherwise} \end{cases} \tag{12}$$

where $\boldsymbol{r}$ corresponds to the position component of the observation vector. Note that in the first case, the norm of a vector derivative is used, whereas in the subsequent two it is the scalar derivative of a vector norm. As discussed in 5.1, the other terminating condition for the freefall segment of the bottle's trajectory is determined using the position of the net:

$$f_{passed}(t) = \begin{cases} 0 \text{ if } \forall u < t, x_{Bottle}(u) \geq x_{Net}(u) \\ 1 \text{ otherwise} \end{cases} \tag{13}$$

The particular contents of the thresholding functions used are application-specific, but the method of detecting discrete events based on relative and absolute derivatives should prove broadly applicable. As with the split step, constants for thresholding were determined by way of inspection and would certainly be unique to each type of task. The output of $f_{release}$ serves as the gripper actuator signal estimate. f_{moving} is used in the final align, crop and combine steps to determine the first observation to include in the dataset. $f_{freefall}, f_{passed}$ are used in estimating the desired target coordinates of each throw to give models the capability to be aimed. This is done by applying quadratic regression to the z-coordinate of the object thrown, finding its intersection with the ground plane in corresponding x and y-coordinates – a geometric illustration of this process can be found in fig. 3. The actuator signal and target coordinates are concatenated to each observation vector, with target coordinates being constant within each demonstration. A time signal is also added to each observation as this was found to improve the performance of feedforward models. Finally, when combining the demonstrations into a training data set, all position vectors are shifted so that the start of motion corresponds to the origin of the coordinate system. This way the models are trained to operate relative to the effector starting position.

5.3 Models

We studied two classes of parametric models as part of this project – simple feedforward neural networks and RNNs, operating autoregressively. The choice was motivated by two factors:

- the dynamics of the problem were deemed to be simple enough that even small models would be able to model them adequately – making it possible to quickly train on development machines with low parameter counts, using pre-existing code libraries;
- given the recent advances in employing sequence-to-sequence models for imitation learning tasks, it was decided that models with broadly similar footprints and characteristics should be used to enable further research in this direction.

After an initial hyperparameter discovery process, the values in table 1 were arrived at for both model types respectively. A range of model sizes and training epoch counts was compared for both models. In the case of the recurrent neural network, performance with different learning rates was also evaluated. As the feedforward models were developed first, performance with and without a time signal in the input data was also compared. For the RNN architecture, GRU was selected as prior research suggests that it outperforms LSTM when dealing with small data sets of long sequences [Yang et al., 2020].

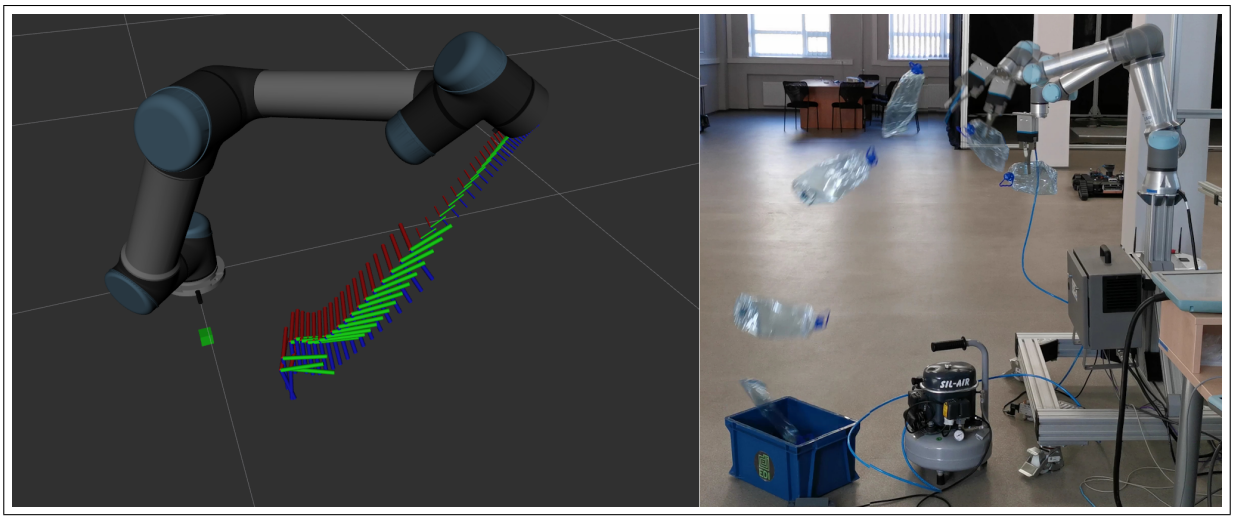

Figure 4: Visualization of pre-computed trajectories; execution of a throw on a real Ur5e robot.

The basic model footprint is given as follows:

$$(\boldsymbol{r}_{t+1}, \boldsymbol{q}_{t+1}, g_{t+1}) = \pi_\theta \left(\frac{t}{f_{sample}}, \boldsymbol{r}_t, \boldsymbol{q}_t, g_t, \boldsymbol{r}_t^{target} \right) \tag{14}$$

where $\boldsymbol{r}_t, \boldsymbol{q}_t, g_t$ represent the end effector translation vector, orientation quaternion and gripper actuator signal respectively at time step t in both the input and output. The input is augmented with a time/phase signal (discrete time step divided by the sampling frequency, in this case $100Hz$) relative to the start of the demonstration or generated trajectory, as well as the target coordinate vector $\boldsymbol{r}_t^{target}$ which corresponds to the extrapolated target coordinates in the demonstration data set and commanded throw coordinates at inference. The time signal was added to the input as it was found that trajectories generated by feedforward networks were liable to diverge without it, and the data sets thus modified were used for all training thereafter. The gripper control signal has values $\{0, 1\}$ in the training data set.

For training both types of networks, state transitions $\left(\left(t, s_t, r_t^{target} \right), s_{t+1} \right)$ are constructed. In the case of the feedforward network, these state transitions are then shuffled and batched independently. For the RNN, pairs of feature-label sequences are formed corresponding to complete demonstrations, and the loss function is computed on the entire predicted output sequence. To hold these variable length sequences, a ragged tensor is used, which is batched along its first axis and ragged along the second. With both types of networks, the Huber loss function is utilized.

5.4 Visualization and execution

As discussed in the introduction, an important part of the reason for selecting throwing as our motivating application was the fact that it is easy for humans to judge the qualitative performance aspects of this task. To accomplish this, a means of visualizing the model outputs was required. When operating in open-loop mode (without interfacing with the physical or simulated environment – running the model on its own previous outputs) it is possible to precompute trajectories and simply save them as sequences of robot states. To aid in estimating whether these trajectories were feasible, a tool for visualizing these sequences in the robot coordinate system was developed (fig. 4, left).

Given a policy that synchronously predicts the state of the system at the next time step, there are multiple possible ways to use it in robot control. The simplest approach is static trajectory planning – precompute a sequence of states and plan the motion between them. This is somewhat hindered by the lack of existing tools for precise time-parametrized Cartesian path planning in the ROS ecosystem. An alternative is a real-time pose following servo controller. The advantage of the latter approach is that closed-loop control is possible – with state observations taken from the environment, potentially allowing the model to compensate for offsets in real time. However, this presents the challenge of tuning controller gains. The method that was ultimately employed was open-loop planning of Cartesian paths – with timing approximated through a combination of time optimal trajectory generation [Kunz and Stilman, 2012], followed by scaling the trajectory to the correct total duration. To control the gripper, a threshold value was set, the point in the trajectory at which this value was crossed was found, the corresponding joint state was computed, and a callback function was set up to trigger when this joint state was reached within some tolerance.

While this is adequate for evaluating the positioning of the generated trajectories, and approximates the velocity profile closely enough for successful throws to be executed (fig. 4, right), a fully-featured time-parametrized Cartesian path planner or correctly tuned pose-following controller would be required to judge the throwing accuracy on real hardware and generalize our approach to other tasks. However, the development of such general purpose tools was deemed to be outside the scope of this project, the focus of which is on collection of demonstration data and imitation learning methods.

5.5 Evaluation metrics

To draw comparisons between models of different architectures, trained with differing hyperparameter sets, quantitative evaluations need to be computed. As this is not a standard task with agreed-upon metrics and benchmarks, some exploratory work was required to arrive at quantifiers that agree with intuitively self-evident characteristics. While in principle it should be possible to judge models based on throw accuracy, this is not a very helpful in the early stages of research when most models fall short of attaining the desired objective. Furthermore, limitations imposed by the aforementioned time parametrization issues with Cartesian motion planning in ROS make such a comparison as yet infeasible.

Hence, it was decided to compare model outputs with the demonstration data set. As judging against the training data set only informs us to the extent to which the model has been able to overfit, a smaller validation data set was set aside for out-of-distribution comparisons. The outputs to be compared were obtained by executing the models autoregresively on each demonstration's initial state, for as many steps as were present in the corresponding recorded demonstration. This can be formally stated as

$$\mathcal{D}_{eval} = \big((\tau_1^d, \tau_1^g), ..., (\tau_k^d, \tau_k^g)\big) \tag{15}$$

$$\tau^d, \tau^g = (s_1^d, ..., s_m^d), (s_1^g, ..., s_m^g); s_1^d = s_1^g \tag{16}$$

where $\mathcal{D}_{eval}$ refers to the evaluation data set of a single model with respect to either the validation or test demonstration set, but τ_i^d, τ_i^g are the demonstration and generated trajectories (sequences of state observations s_j) sharing the same initial state respectively.

Three broad classes of evaluation metrics were computed: data set-wise (global), step-wise and throw parameters. The first consist of vector similarity measures such as Pearson's correlation coefficient, cosine similarity and distance metrics applied to the entire data set and the trajectories generated against it as concatenated vectors:

$$f_{global}(\mathcal{D}_{eval}) = f : (\tau_1^d, ..., \tau_k^d) \times (\tau_1^g, ..., \tau_k^g) \to \mathbb{R} \tag{17}$$

The second class involves applying various measures to the observation/state variables at each time step:

$$f_{stepwise}(\mathcal{D}_{eval}) = f : \left[\sum_{\tau^d, \tau^g \in \mathcal{D}_{eval}} \sum_{i=1}^{m} \big(g : s_i^d \times s_i^g \to \mathbb{R}\big) \right] \to \mathbb{R} \tag{18}$$

where the inner function g corresponds to metrics such as position error, rotation error (quaternion angular distance) or categorical crossentropy in the release signal. Finally, a release error metric was computed trajectory-wise. To do this, the release point of each trajectory was found, the corresponding position found and velocity vector estimated. When considering ways to combine these two terms into a single quantitative error estimator, it was decided that a simple ballistic extrapolation and resulting miss distance along the ground plane would serve as a decent first order approximation of throw accuracy *vis-à-vis* the training or validation data set.

Specifically, the release position $\mathbf{r}_0 = (x_0, y_0, z_0)$ and estimated release velocity $\mathbf{v}_0 = (v_{0x}, v_{0y}, v_{0z})$ were used as parameters in the equations:

$$x(t) = x_0 + v_{0x}t \tag{19}$$

$$y(t) = y_0 + v_{0y}t \tag{20}$$

$$z(t) = z_0 + v_{0z}t - \frac{g}{2}t^2 \tag{21}$$

The ground plane intersect time $t_{intersect}$ was found by setting eq. 21 equal to the ground plane coordinate z_{target} and finding the positive root. Then ground plane intersect points were found as

$$r_{intersect} = (x(t_{intersect}), y(t_{intersect}), z(t_{intersect})) \tag{22}$$

and the trajectory-wise throw error metric computed by:

$$f_{throw}(\mathcal{D}_{eval}) = \frac{1}{k} \sum_{\mathcal{D}_{eval}} \|r^g_{intersect} - r^g_{intersect}\| \tag{23}$$

with $r^d_{intersect}, r^g_{intersect}$ being the demonstration and generated throw intersect positions respectively. Seeing as not every model would output a release signal that crossed the threshold (0.5) for every trajectory, this error term was set to be infinite in cases when no throw was defined.

6 Results

Over the course of our research, numerous models were trained on both of the data sets described in 5.1, in both of the architectures discussed in 5.3. After the initial trial and error process, a systematic training regimen produced the following:

- feedforward networks – a total of 48 models, varying the training data set, presence of a time signal in the features, perceptron counts per hidden layer and training duration;
- RNNs – 36 models with data set, learning rate, model size and training length being the variable parameters.

Fig. 4 illustrates how model outputs were visualized to use their qualitative aspects in guiding hyperparameter choice, and a preliminary implementation on a real robot was achieved – executing trajectories generated by the simpler feedforward network, with target coordinates sampled from normal distributions corresponding to their values in the training data set. As can be seen, when these outputs were used to generate motion plans for a *Universal Robotics Ur5e* robot equipped with a pneumatic gripper, a plastic bottle was successfully thrown into a target container located outside the robot's reachable volume. Adequate output sequences were attained with both model architectures at multiple hyperparameter combinations, so to draw specific conclusions the quantitative evaluation metrics discussed in 5.5

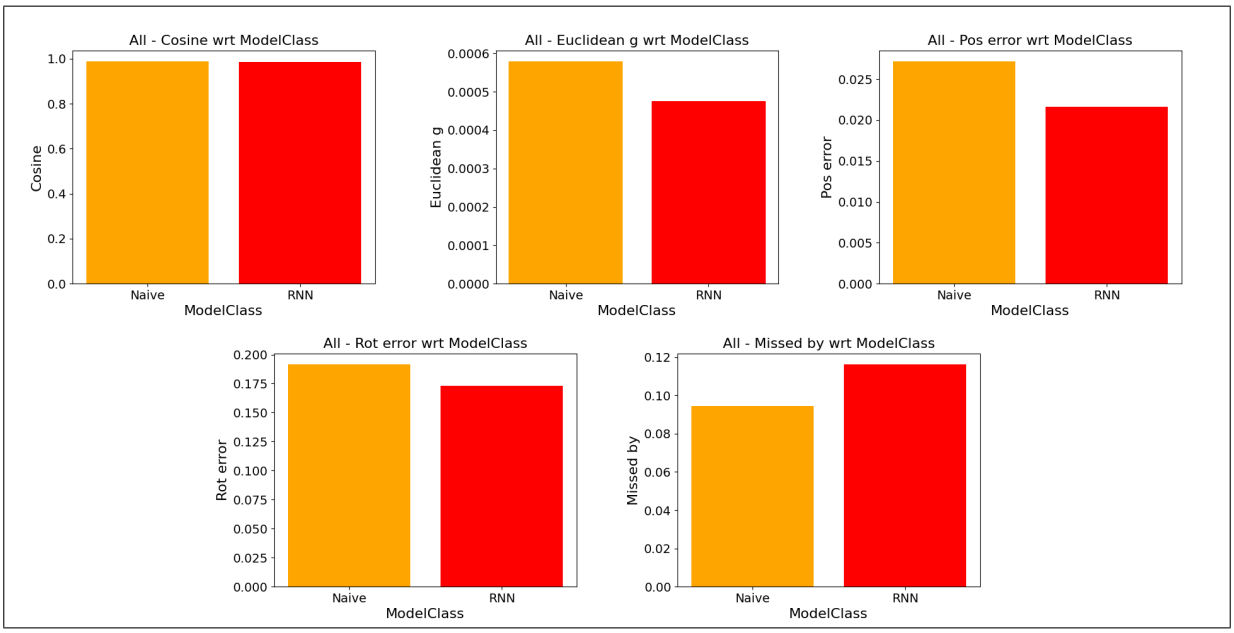

Figure 5: Results – comparison between model architectures (*Naive* – refers to the feedforward architecture, also "naive behavioral cloning"), the best performance attained in each class. Top row metrics – cosine similarity, Euclidean distance (global); mean position error (stepwise). Bottom row – mean rotation error (stepwise), mean throw error.

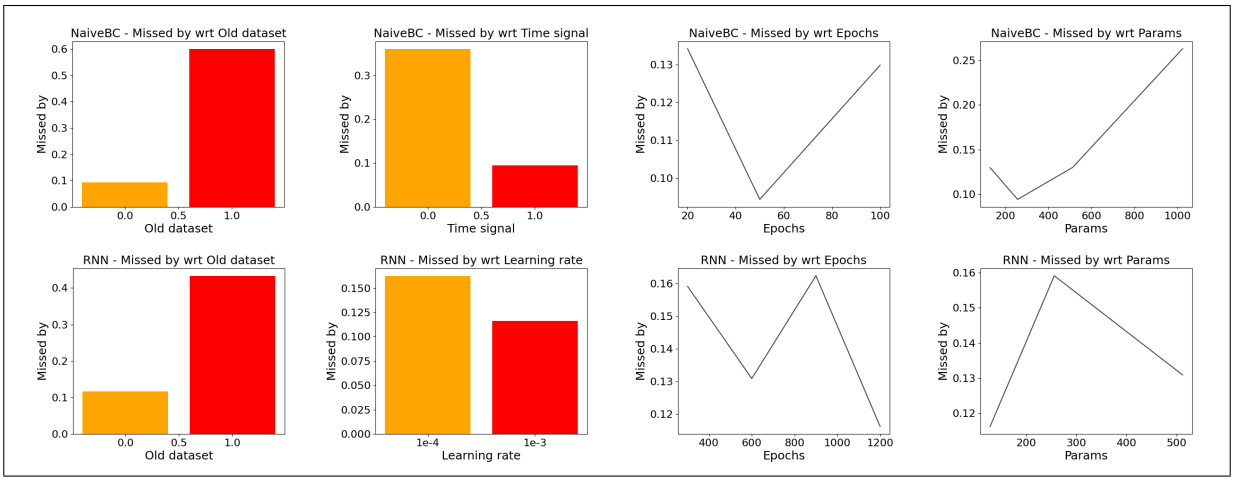

Figure 6: Results – best performance for each hyperparameter value. The top row shows feedforward model results, the bottom – RNN. In both cases, the smaller, newer dataset with less variance in orientation and target coordinates results in better throw evaluations. Introducing the time signal significantly improves feedforward model performance, whereas for the RNN improved results could be attained at higher learning rates. In the case of the feedforward model, training for too long and making the model too large initially appears to be detrimental, while for the RNN no clear trend was observed in this respect.

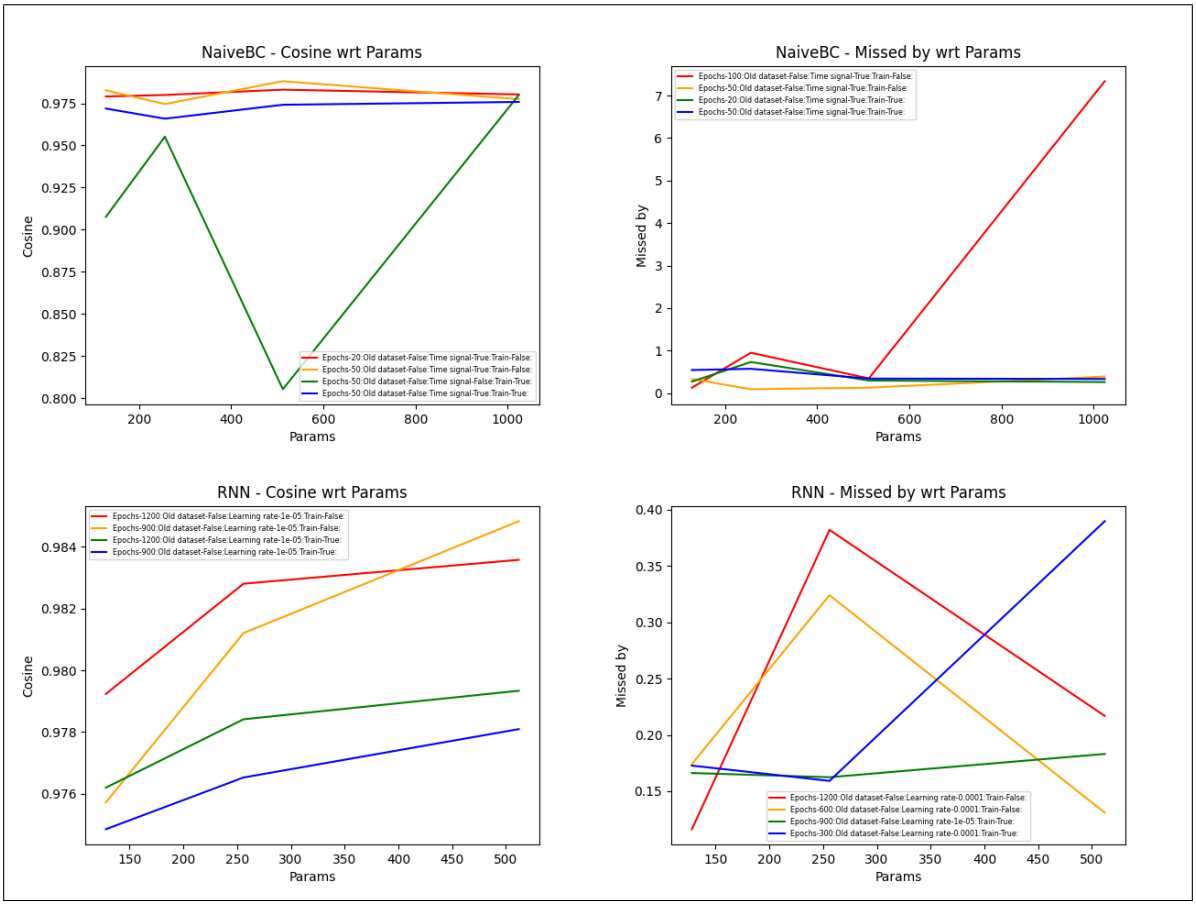

Figure 7: Results – cosine and throw accuracy metrics for feedforward and recurrent models only varying in the perceptron count parameter (others held constant) – best 2 on train and validation sets each.

were produced for the 84 systematically trained models described above. Of the feedforward models, 18 out of 48 (37.5%) successfully triggered a gripper release on every evaluation trajectory, enabling mean throw error estimates to be computed. Among the RNNs this was true for 24 out of 36 (66.7%).

Figure 5 shows a broad comparison between the RNN and feedforward model architectures, with each of the evaluation metric classes being present, though not all the specific metrics discussed – as there was found to be a considerable amount of duplication in their findings. All figures shown correspond to the best result attained by either type in each of the performance indicators, irrespective of other variables. Figure 6 elaborates upon each model class, showing the best attained results at various hyperparameter values. In the case of the feedforward model, the input observations did not initially contain a time signal, which was rectified early on. In the case of the RNN, deviating from the default learning rate was found to be beneficial. For both architectures, the best performance at each epoch count and parameter count is also plotted. Figure 7 shows the impact of only varying the crucial size (perceptron count) – in each case the two best performing models at each fixed parameter combination are selected and their performance at different values of the independent variable is graphed.

7 Discussion

Comparing the two proposed model architectures by their best attained values, in terms of distribution similarity measures such as cosine similarity and Euclidean distance, the results are generally quite good for both, with the RNN having the edge in most distance measures but notably one of the feedforward models demonstrating the highest cosine similarity. In terms of step-wise metrics, RNN models are typically better show better results – which is perhaps to be expected, given that their inputs contain all previous states in the trajectory rather than a single observation.

The same is true for the throw error metric in general, and it is quite apparent when comparing the percentage of valid throw trajectories (ones where has been commanded) that RNNs have an overall easier time learning the release timing aspect of the task. However, the best single result was attained by an outlier feedforward model – an average error of around 0.09m, as opposed to around 0.11m for the best recurrent model, both results attained on the validation data taken from the corrected, smaller demonstration collection (see 5.1). This result should be taken with a grain of salt, though, considering the small size of the validation data set (5 demonstrations, set aside from a collection of 45). The performance of the same feedforward model compared against the remaining training data set is actually much worse – an error of 0.57m – compared to the best RNN model, which retains an error of 0.26m. Moreover, at other hyperparameter combinations, RNN models can be seen to retain sub-0.2m error on both data sets (such as a 256-perceptron model trained for 300 epochs at a learning rate of 10^{-3}, which attains 0.16m and 0.18m error on the training and validation data sets, respectively).

An important thing to note is that this throw error estimate should not be taken as equivalent to a simulation or an actual throw – as has already been discussed in 5.1, the terminal velocities of the bottles thrown are low enough to affect the observed trajectories, and in any case instantaneous velocity vector estimates derived from sequences of discrete position measurements are bound to have a degree of error. This is further exacerbated by the fact that the interactions between the gripper and the work object would impart some delay between the release command and full separation. Nevertheless, this estimator does model a large part of the non-linear relationship between the parameters that define a throw – release position and velocity vectors – in a way that is likely to explain a large degree of destination variance among throws performed in the physical world. It is reasonable to assume that lower values of this error term would be strongly correlated with higher accuracy when deployed and tested on physical hardware.

Regardless of model type and parameters, the observed performance on the older data set, uncorrected for robot working volume – with a much greater variance in throw shape, timing, starting orientation and target coordinates, but only marginally greater size – is significantly worse. None of the feedforward networks attain a throw error estimate under 0.6m, while RNNs bottom out at around 0.4m. In the case of feedforward networks, augmenting the input vector with a time signal showed qualitatively observable advantages, and these are also present in the numeric evaluations. An apparent trend of better results being obtainable with smaller models that aren't trained for too long also exists, however, for the reasons discussed above this may well be spurious. Certainly no such conclusion can be confidently drawn with respect to the RNNs' performance, as comparatively high performance is achieved by some hyperparameter combinations at every scale explored. A slight, but consistent edge in performance was attained by increasing the learning rate of the Adam optimizer from the default value of 10^{-4} to 10^{-3}.

Observing the impact of hyperparameters in isolation yields the clearest insights in the cases of global distribution similarity measures (in the graphs shown, cosine similarity). In feedforward models, a high similarity is attained even at the lowest parameter counts, and making the models larger does not yield unambiguous improvements. Notably, at some settings there is a considerable degree of variance in this metric. Among RNNs, the perceptron counts examined show a slightly yet still monotonically increasing trend, suggesting that performance gains at larger model sizes may still be

made. When it comes to throw accuracy estimates, neither model architecture shows a clear response to perceptron count, and the feedforward architecture again exhibits outlier results. Interestingly, multiple RNN configurations show initially declining performance on validation data with increased model size, followed by an improvement – potentially evoking the double descent phenomenon [Nakkiran et al., 2021]. However, there is still a comparable degree of unexplained variance in the results, which forces us to be cautious in proposing any concrete explanations.

8 Conclusions

References

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, pages 60–77. MIT press, 2018.

Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Xi Chen, Ken Goldberg, and Pieter Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5628–5635. IEEE, 2018.

Jonatan S Dyrstad, Elling Ruud Øye, Annette Stahl, and John Reidar Mathiassen. Teaching a robot to grasp real fish by imitation learning from a human supervisor in virtual reality. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7185–7192. IEEE, 2018.

YuXuan Liu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Imitation from observation: Learning to imitate behaviors from raw video via context translation. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1118–1125. IEEE, 2018.

Alexandre Attia and Sharone Dayan. Global overview of imitation learning. *arXiv preprint arXiv:1801.06503*, 2018.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY . . . , 1989.

Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *2009 IEEE International Conference on Robotics and Automation*, pages 763–768. IEEE, 2009.

Stéphane Ross, Geoffrey J Gordon, and J Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. In *In AISTATS*. Citeseer, 2011.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29:4565–4573, 2016.

Stefan Scherzinger, Arne Roennau, and Rüdiger Dillmann. Contact skill imitation learning for robot-independent assembly programming. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4309–4316. IEEE, 2019.

Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. Learning latent plans from play. In *Conference on Robot Learning*, pages 1113–1132. PMLR, 2020.

Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

Abhishek Jha, Shital S Chiddarwar, Rohini Y Bhute, Veer Alakshendra, Gajanan Nikhade, and Priya M Khandekar. Imitation learning in industrial robots: a kinematics based trajectory generation framework. In *Proceedings of the Advances in Robotics*, pages 1–6. 2017.

Katerina Fragkiadaki, Sergey Levine, Panna Felsen, and Jitendra Malik. Recurrent network models for human dynamics. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *IEEE Transactions on Robotics*, 36(4):1307–1319, 2020.

Shudong Yang, Xueying Yu, and Ying Zhou. Lstm and gru neural network performance comparison study: Taking yelp review dataset as an example. In *2020 International workshop on electronic communication and artificial intelligence (IWECAI)*, pages 98–101. IEEE, 2020.

Tobias Kunz and Mike Stilman. Time-optimal trajectory generation for path following with bounded acceleration and velocity. *Robotics: Science and Systems VIII*, pages 1–8, 2012.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12): 124003, 2021.