Dice Stacking: A Dynamic Manipulation Task

Hunter Goforth
Robotics Institute
Carnegie Mellon University

Leonid Keselman
Robotics Institute
Carnegie Mellon University

***Abstract*—We present dice stacking as a highly dynamic, task that can serve as a model problem for robotic manipulation. While a seemingly simple human manipulation task, containing only a few rigid-body objects, it is shown to be rather challenging for standard algorithms in robotics. We believe this is due to the need to maintain high velocity while undergoing regular collisions. We have designed a simulator to model this task. Our simulator produces plausible behavior, and includes multiple user input mechanisms to allow humans to produce viable trajectories in simulation. We were successfully able to learn to perform the stacking task in our simulation using force/torque control, when the simulator is run at about half of real-time speed. We then investigate multiple methods for automatically discovering and learning this behavior, including black-box optimization, reinforcement learning, imitation learning, and planning. We finally propose a novel variant of imitation learning for control which is able to solve the task somewhat successfully.**

I. Introduction

Dice stacking, in the humble words of Wikipedia, is *a performance art, akin to juggling or sleight-of-hand, in which the performer scoops dice off a flat surface with a cup and then sets it down such that it stacks the dice into a vertical column*. One very clear visual example of this task is available on YouTube [14]. We selected this as an interesting manipulation task that expert humans are able to successfully perform. We found no previously published research on this topic, and decided to investigate this task via the use of a physics simulator instead of a completely theoretical analysis. As the task requires constant motion, possibly complicated inter-object behavior, and varying friction coefficients, a simulator is a good starting point to study this problem.

All of our code, and many of our successfully recorded runs are available at http://placholder

II. Related Work

There exist many simulation environments for robotic tasks. They range from general purpose physics simulators such as MuJoCo [31] and Bullet [7], to task-specific simulators such as GraspIt [22]. There is also a middle-ground of fairly general, robotics focused simulator environments such Drake [30] and Gazebo [16]. Others exist to focus on a specific technical approach to robotics, such as OpenAI's Gym [3], which phrases all robot learning tasks as reinforcement learn problems. Each of these simulation environments comes packaged with many example tasks. However, we found many of these examples were fairly simplistic and unable to serve as sufficiently sophisticated test environments for research into highly dynamic tasks.

In this case, we propose a specific new task, that of dice-stacking. Our motivation in this task is to develop a problem domain which is simple yet highly dynamic. In this case, we mean *dynamic* in the sense of velocities being highly important to the successful completion of the task. We believe this is an important area of research, as even some of the most sophisticated research in model-free robotics tends to focus on tasks that are effectively static. For example, in [20], many of the tasks focus on positioning challenge. The scope of dynamic tasks is limited to those solvable with center-of-mass planning [9]. While more recent work has focused on dexterous manipulation [24], these tasks tend to be much higher dimensional and still rarely have non-zero velocity for most objects. We propose dice stacking as a challenging task where objects only have non-zero velocity at the start and end of the simulation. This task is both simply and highly dynamic, making it an interesting new problem to tackle in manipulation research.

We should note that older work exists in some highly dynamic tasks, such as juggling. However, while there was much initial work in juggling controllers [1], it was later shown that that juggling is solvable with clever open-loop control alone [27]. We hope that dice stacking is somewhat more challenging task, despite being fairly similiar in number of moving parts.

III. Simulation

To investigate the problem of dice stacking, we built a 2D dice stacking simulation from scratch in Python. Some example screenshots from this 2D simulation are shown in Fig. 1. Representing the simulation in this way provides multiple benefits. The first benefit is that it is an inherently simpler, and only requires a 2D physics simulator framework. It is also easier to introduce a simple control scheme for a user to control the cup, and stacking the dice actually becomes much more achievable (with a bit of practice) than it might otherwise be in the real 3D world. We describe our design of user control schemes in Sec. III-A. Our concept of what it means to be successful in this task is described in Sec. III-C.

Our physics simulator of choice was Chipmunk 7 [19], through its Python interface Pymunk. Chipmunk implements a temporally coherent iterated impulse solver [5], which provides very fast simulation of a physical environment. The solver provides a simulation of real-world dynamics by iteratively applying changes in velocity (force over time is an impulse, and it produces a change in velocity). Any intersection between

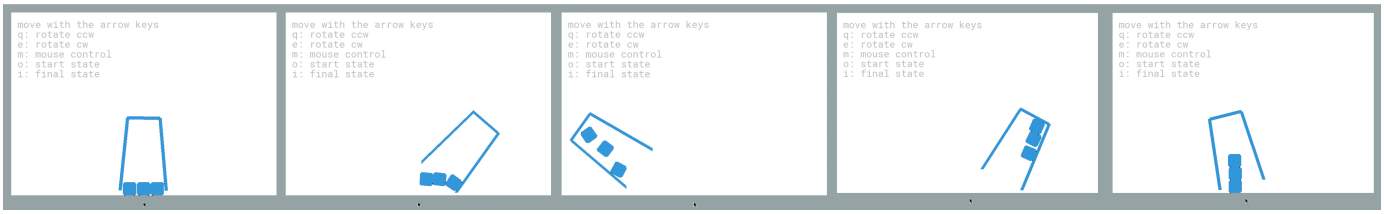

Fig. 1. A visual example of our 2D dice stacking simulation, successfully completing our designed task of taking dice laying flat on a table and stacking them on top of each other using a cup. Videos of our simulation are including with the submission but can also be seen on YouTube [14]

bodes is computed with GJK [10] and corrected. Finding the minimum number of simulations to run, while avoiding object interpenetration, required a small amount of tuning. For each timestep, we ran 5 simulation steps, with 10 solver iterations per simulation step. The use of an iterated impulse solver means that our physics simulation is not strictly deterministic, and may produce different simulation results on different runs due to how collision constraints are handled. This informed the design of many of our algorithms in Sec. V. Ultimately the benefit of this simulator is that is both sufficiently realistic and extremely fast (in our testing, we can do 40 simulations per second on the single core of a 2017 Macbook Pro. Each simulation here was 100 timesteps, which, as seen in Fig. 2, is long enough to solve the task).

The shapes in our simulation are all composed of simple convex primitives. The cup consists of three segments (a base and two walls) and has a slight outward tilt towards the cup opening. The dice are not squares, but instead octagons that resemble squares but have a slight lip to model the shape of real dice. The entire simulation has enclosing walls to keep everything in a contained environment.

We also performed significant tuning of our physics simulation settings. In the end we settled on the following constants for the physics simulation:

$$gravity = -4.0m/s^2 \tag{1}$$
$$\mu_{dice} = 0.6 \tag{2}$$
$$\mu_{table} = 1.0 \tag{3}$$
$$\mu_{cup} = 0.4 \tag{4}$$
$$m_{dice} = 10g \tag{5}$$

These settings were tuned based on some understanding of friction of real bodies and tuning to make the experience intuitive yet challenging for a player. For example, the cup has the lowest coefficient of friction, followed by the dice, and lastly the walls and floor surface. This is based on our assumption of smooth slippery dice and something like a smooth glass cup. Lowering gravity allows us to play the simulation in slow motion, and we picked a factor of about 40% of real-time speed. We found that the friction between the cup and dice needed to be low enough such that the dice would slide rather than roll down a tilted cup wall, as would be the case in the real world. Also, the friction between two dice needed to be tuned such that they would not become jammed in a non-stacked position within the cup. However, lowering the friction too much of either the cup or dice made it difficult to pick up the dice with the cup at all. The mass of the dice is based on a standard 22mm die which has a typical mass of around 10 grams. The frictional coefficients are the same for both static and moving friction, as Pymunk did not allow us to specify separate values. While these values were not based on any measured values, their behavior seemed realistic and allowed for users to sucessfully complete the stacking task with our control scheme.

A. *Design of control interface for users*

To generate valid solutions or example trajectories for this task, we developed many forms of user input. Primarily, we focused on methods which producing velocity changes, as a force/torque controller is an operation standard in physical robotic manipulators [15]. We tried many different forms of control schemes before settling on one that allowed for us to easily control the game. For example, we initially implemented gravity compensation for the cup but found that it made it harder to control the system. We finally settled on a control scheme where holding a key `up/down/left/right` for a timestep applies a fixed force in that direction. This is a form of force control. Likewise for the `q/e` keys and rotating the cup clockwise or counter-clockwise, where we apply a torque at the center of mass. When the user releases a key direction (either vertical, horizontal, or angular), we apply a corrective force to stop the cup in that dimension. This allows for the cup to behave predictably, otherwise we'd often find the cup moving even when the user released the keys.

In the final variation, the user may control the cup in various ways. The arrow keys provide up-down-left-right movement of the cup, with directions always defined in the global frame of reference. To instead enable orientation control via the mouse, the user can press `m`. When mouse control is active, the cup will always face the mouse pointer. The starting state of the simulation consists of the 3 dice laying flat (not stacked). The goal state is to have the 3 dice stacked.

B. *Unrealistic Solutions*

We first explored some methods of achieving the goal state in our simulation which do not reflect how the solution is normally achieved in the real world. Examples of these possible but unrealistic solutions are shown in Fig. 3 and Fig. 4. These generally involve using the vertical walls included in

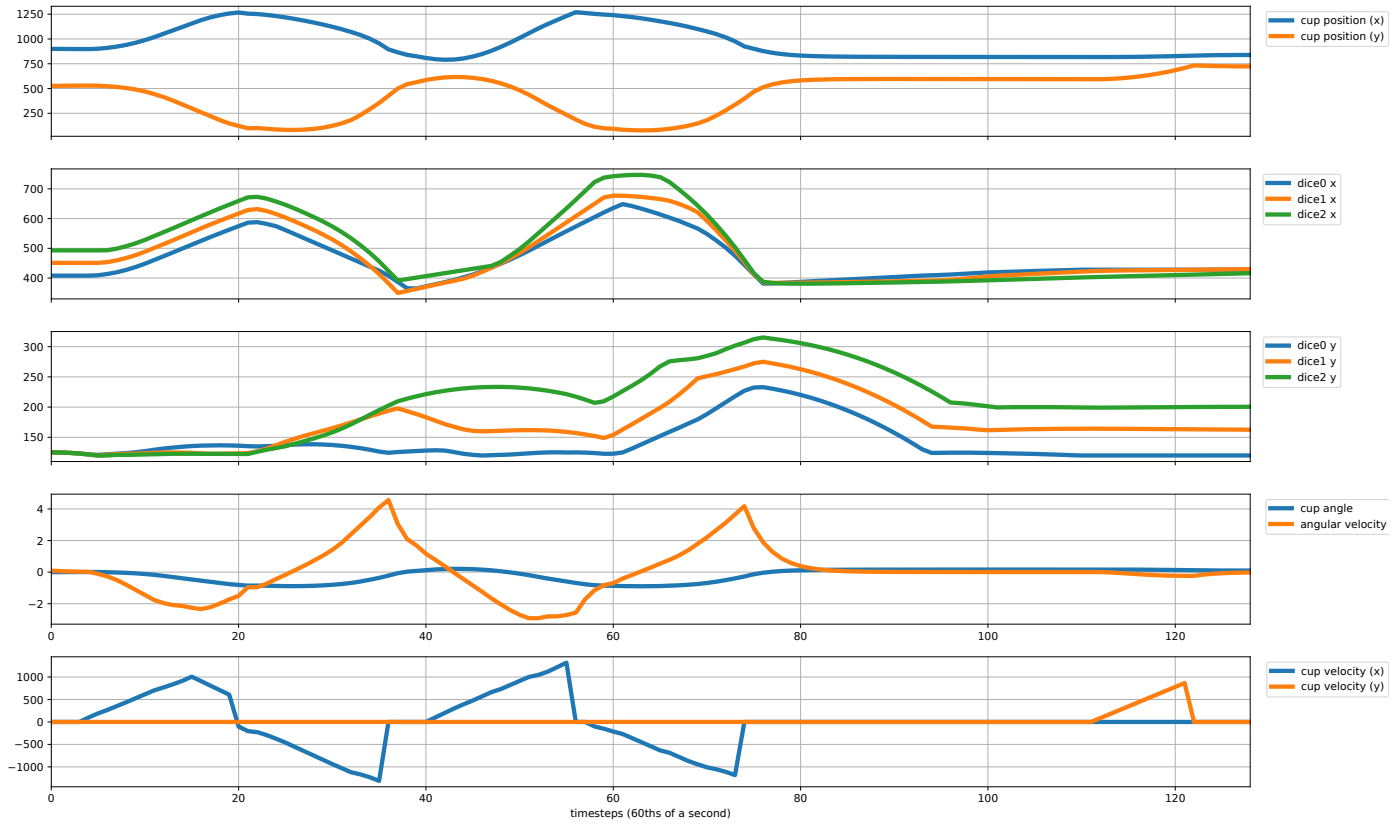

Fig. 2. The state of the simulation during a successful run (see Section III-C). The top row shows the position of the cup, and you can see our characteristic left-to-right swinging motion. The second and third rows show the positions of the dice, and we can see that the swinging motion allows the dice to stack on top of each other after the second swing. The last two rows show the angles and velocities of the cup, demonstrating what our force-control actions are actually doing to cup (force being proportional to acceleration, which produces a change in velocity). The entirety of this run corresponds to about two seconds of real-time.

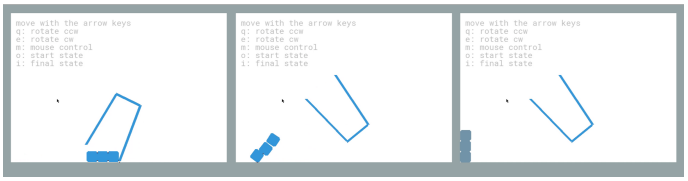

Fig. 3. Example of unrealistic stacking solution. The cup starts in the initial state (left) and rotates to fling the dice at the vertical wall where they settle in a stack. This is unrealistic because not only is it unlikely to happen in the real world, but it is also against the rules to use a vertical wall to aid stacking.

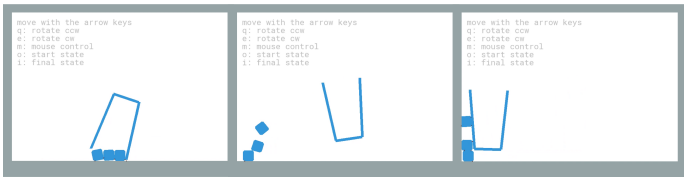

Fig. 4. Example of unrealistic stacking solution. The cup starts in the initial state (left), rotates to fling the dice at the vertical wall, and then hits the dice to cause them to settle in a stack.

the simulation. These are unrealistic and not very much in the original spirit of the game, as real-life dice stacking does not include the use of any vertical walls. Some other similarly unrealistic methods could be devised, such as scooping motions or flinging motions. We wanted to explore these possible but unrealistic ways of stacking in order to understand better the full suite of manipulations possible with the cup, and also to understand what might happen when we start trying to train a computer to accomplish the stacking task.

C. Realistic Solution

A more realistic solution is shown in Fig. 1 and in the accompanying video. This solution obeys the original rules of challenge. Specifically, neither the cup nor dice make contact with the vertical walls. The cup is moved in the same sort of smooth, back-and-forth motion that we see in the slow motion video. The dice are picked up by the cup via the cup sliding along the table toward the dice at an angle which knocks the dice upward, while the cup rotates in order to catch the dice as they fly. The cup is moved left and right, all while maintaining an orientation pointing toward the center-bottom of the game screen (where the mouse pointer is located). Using this technique, the dice eventually become stacked within the cup as the cup is moved left and right. Finally, we can slow down the cup motion gradually, and stop the cup when it is

approximately vertical. The dice drop down, stacked. We plot some parameters of the game state during this realistic solution, such as cup position and velocity, in Fig. 2.

D. How does our simulation compare to the real world?

We can compare our two dimensional simulation with a slow-motion capture of what occurs in real-world three-dimensional dice stacking. We found a good example at [11]. We find that the basic behavior of our simulation and the real-world user is very similar. The user uses the cup's velocity to knock the dice up and into the cup. Then a series of left-right motions allow the dice to be knocked into the top of the cup. If the cup is kept nearly vertical, the dice will largely stack in one of the two corners in a vertical fashion. Afterwards, with a small amount of finesse, the user slowly stops the cup and lets the dice slide down the the cups interior and onto the table, remaining vertical. This type of behavior is seen in both the 2D simulation and the real-world stacking video.

However, there is at least one limitation to our two dimensional simulation. In the video referenced above, the real-world player is able to pick up dice on each left/right action by rotating their wrist and changing which corner of the cup the dice are being stacked in. This rotational action is not possible in our two-dimensional simulation, and if we were to pick up dice one-by-one, we would require consistently using a left-to-right or right-to-left action to knock the dice into our cup.

IV. Producing Viable Trajectories

We initially did not have a sufficiently good user control scheme, nor the experience, to perform dice stacking successfully in simulation. We thus attempted a few strategies of solving this task without knowing how to do it ourselves; these attempts are outlined in Sec. V. After these methods failed to provide us desirable solutions and we also got good at playing the game, we tried imitation learning techniques to play the game successfully; these results are in Sec. VI.

V. Automatic strategies, without guidance

A. State space

To use algorithms to automatically learn the dice stacking task, we required a suitable state space within which to operate. We chose to define a 24-dimensional state space which includes:

- Position and angle for the cup (3)
- Linear and angular velocity for the cup (3)
- Position and angle for 3 dice (9)
- Linear and angular velocity for 3 dice (9)

We allow the same 3 dimensional input to the simulation as the user is allowed. That is, an algorithm may apply force control to change the cup position and angle. While the user's choice of forces is binary (either they are applying a fixed-sized force or not), the computational methods predict a real-valued force, up to the order of magnitude of the user input forces. Additionally, we don't include "stopping" mechanic that human players get, see Sec. III-A.

B. Optimization

Our initial formulation of the problem was simply to construct a function to perform the task, return a result fidelity, and optimize the input with a black-box optimizer. The input control was parameterized as $x \in \mathbb{R}^{n\times 3}$, with n timesteps of input, and a linear and angular force to apply at each timestep. We treated all dimensions as largely existing in the unit hypercube of $x_{ij} \in [-1, +1]$, where we applied per-dimensional scaling afterwards. We had scaling of $100, 10, 0.5$ in the x, y, θ dimensions respectively. This is because, as seen in Fig. 2 we largely use horizontal motion, with little vertical motion. We did runs of 30 timesteps with gravity at 3.0x the settings in Eq. 1 (and respectively up-scaled force limits). This was to simulate the behavior of a user, where they might only be able to change their input 10 times per second, or roughly every 3 time-steps. This gave us a 90-dimensional optimization problem.

Our simulation used the L_2 variant of Eq. 9 as our distance function $d(\vec{x}, \vec{y})$ as it's distance function between states. We implemented two ways to stabilize this optimization. The first technique was the use of multiple evaluations for every single function evaluation to handle our non-deterministic physics simulator. Without this, the optimizer would often find minima that would actually be worse than the initial conditions upon re-running them. The second was the use a variant of discounted rewards. Our total error can be written as

$$\text{Cost} = \sum_{i=1}^{N}\sum_{j=1}^{T} \gamma^{T-j} d(\overrightarrow{f(x_j|x_{j-1}, \ldots, x_0)}, \vec{y}) \qquad (6)$$

This is done over N samples, with $x_j \in \mathbb{R}^3$, $\vec{y}$ as the goal state, $d(\vec{x}, \vec{y})$ as described above, $f(x)$ returning simulator state (as described in Sec. V-A) and discount factor $\gamma \in [0, 1]$. The discount factor allows us to penalize all states, with larger weights on the final states. If we wish to only penalize the last state, we can set $\gamma = 0$; if we wish to penalize all states equally, we can set $\gamma = 1$. In our experience a setting between 0.1 and 0.5 is best, as it allows for smooth optimization and a focus on the final states.

We experimented with multiple black-box optimizers, and rank them by our subjective experience of using them below.

1) CMA-ES [12] was the best, and often generated solutions where it would flip the cup sideways, launching the dice such that the final state of the simulation often had them nearly in the right position (see Fig. 5). However, none of these input sequences were ever stable. Compared to the other solvers, we believe that CMA's iterative re-fitting algorithm led to good handling of the many-timestamp issues, and its logarithmic dependence on optimization dimension meant that it ran rather quickly.
2) Differential Evolution [28] sometimes produces results that looked like the CMA-ES results, but often simply shook the cup side to side. We believe this is because our sampling strategy kept track of the best result so far, and hence was affected by an unusually good sample

being in the batch, even if was an unstable result where the dice happened to bounce just right.

3) Bayesian Optimization [2] only ran for a few dozen time-steps before the cost of running the optimizer began to dominate the per-iteration compute time. For results with only a few function evaluations, it produced good objective values but we never ran it as long as the aforementioned solvers, which happily handled thousands of simulations.
4) Basin-Hopping with L-BFGS [4, 34] performed very poorly. The basin hopping would often sample a random configuration and the local optimizer would do something unexpected due to the stochastic cost function. The new random samples were sometimes good, but the local optimization step produced no benefits.

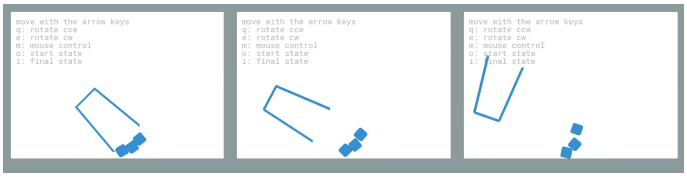

Fig. 5. Example of an optimizer-based input sequence, flipping the cup such that the final state closely resembles the goal state for the dice. See Sec. V-B

C. Reinforcement Learning

Using the same cost function as in Sec. V-B, we also implemented a reinforcement learning strategy. Since this formulation was aware of different states and actions, we thought it would be more stable than simply using a black-box optimizer. We implemented a policy gradient method, namely REINFORCE [35], from scratch. We used a per-timestep exponential moving average baseline with an $\alpha = 0.99$ and the following update equation

$$b_t = (1 - \alpha)b_t + \alpha G_t \tag{7}$$

We additionally used discounted rewards [29] such that

$$G_t = \sum_{t' \geq t}^{T} \gamma^{t'-t} r_t \tag{8}$$

where, in our case, $r_t = -d(\vec{x}, \vec{y})$. We implemented our function approximator as a three layer neural network, with hidden size 32, a discount factor of 0.99, and LeakyReLU activations ($\alpha = 0.05$) [36], and Gaussian Noise [29] to handle the case of exploration in a regression environment ($\sigma = 0.3$). We used a $\tanh$ function to produced the same $x_i = [-1, +1]$ values for force-control of the cup as in Sec. V-B.

We spent a few hours trying to tune the settings and make the algorithm balance exploration, smooth convergence, but we never got anything better than a random strategy that shook the cup and knocked the dice around. While this may be a more promising technique with additional run-time (our longest training run was around 10 minutes or 5000 training steps), in our experiments this was never able to produce a useful policy.

D. Rapidly Exploring Random Trees

Additionally, We implemented a planning-based solution to this problem. We implemented our own variant of single-directional, non-connected RRT [17, 18]. Our specific contribution in this case was, every time RRT sampled a new target state, we ran an optimization function (Sec. V-B) to produce actions for a few time-steps, trying to perform the desired state-to-state transition. Then we added the result of the final optimized section of actions and the state transition that it produced into our RRT tree.

However, we found that it took several hundred iterations before we found any iterations closer to the goal state than the initial state. Thus, we never found a good plan from the initial to the goal state. We suspect that this is due to our very high dimensional state space (Sec. V-A) and uniform sampling in said state-space being incredibly inefficient.

VI. Automatic Strategies, with Guidance

We each practiced executing the realistic solution until we were good enough to stack the dice successfully approximately 1 in 10 times. This took several thousand attempts, and we recorded about 100 successful runs in our equivalent of bag files. We wanted to become good enough at executing the stacking so that our solutions could provide supervised training data to an imitation learning algorithm. Our attempts at teaching a computer to stack dice with our simulator are described in the next sections.

A. Playback Imitation

The most straightforward method of using our successful games is to simply play back the control input of a successful user trial. However, we found this strategy was not entirely successful, not only because it couldn't handle things like placing the dice in different positions, but also because our simulator is non-deterministic (Sec. III). We tried playing back a few of our runs, and while one of the authors often generated runs that played back successfully, the other author was unable to get any of their runs to play back correctly.

In order to create a more generalized policy, we decided to learn a classifier to predict our actions based on a large family of successful behaviors. This can be seen as supervised learning for imitation learning, or a variant of DAgger [26] with $\beta_i = 0 \ \forall i$.

B. Choice of Regression Class

We attempt to train various regressors in a supervised setting, in order to learn the dice stacking task. The input to the regressor is the 24-dimensional game state, and they must predict the 3-dimensional force control output at each time step consisting of the linear and angular forces to apply to the cup. We experiment with using Support Vector Regression (SVR) with Radial Basis Function (RBF), a Multi-Layer Perceptron (MLP), a Random Forest regression, a stochastic gradient descent linear SVR implementation [25], and the gradient-boosted trees [6]. We use sklearn [23] and XGBoost [6] for regression implementations.

To generate training data for the regressors, we played the simulation successfully almost 70 times using the solution method described in Section III-C. We recorded the 24-dimensional state, and our 3-dimensional velocity control inputs, for each success. We fit each of the regressor types to this data, and then handed control of the simulation over to the regressor to observe the result. The results for training error, testing error, and a qualitative description of performance are shown in Table I. For training and testing error, we report a variant of Mahalanobis distance [21]. Specifically, we use a diagonal covariance and L_1 norm:

$$d(\vec{x}, \vec{y}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{3} \frac{\left| \vec{x}_j^{(i)} - \vec{y}_j^{(i)} \right|}{\sigma_j^{(i)}} \tag{9}$$

For N examples in the dataset, dataset labels $\vec{x}$, regressor predictions $\vec{y}$, and σ the standard deviations from the training dataset. One interesting finding for the regressors is that they seem to benefit from an extra force input from the user in the initial state, pushing the cup either left or right. Without this initial push, most regressors have a tendency to remain still or do only small motions. We include videos of RBF SVR, and gradient-boosted trees in our video. In each of the videos, at the beginning of a simulation run, we input an extra push to the left which jump-starts the regressor to begin the characteristic swinging motion. Unfortunately, none of our regressors were able to fully complete the goal, although some came close.

In general, we found that the tree-based regressors had the best performance in terms of training and testing error. We hypothesize that this is true because of the very high correlation between the features (the 24-dimensional state space) e.g. the position and velocity of the dice and cup are highly interdependent. Tree-based methods are often the best in problems with this type of feature overlap. The parameterized methods struggle to achieve good numerical results, most likely due to it being difficult to generalize to a distribution over the relatively small number of training examples (about 70). We do however see an expected result, that RBF SVR performs better than linear SVR, due to nonlinear interactions between state vector and control input.

C. Design of Regression Function

When determining hyperparameters for our regressors, we focused on finding parameters which seemed to best avoid over-fitting while also providing low training error. For RBF SVR we use $C = 1.0$. For the MLP, we use 5 hidden layers with 16 neurons each and ReLU between each layer. We use the Adam solver and train for 100 iterations. For the Random Forest, we used 8 trees with a maximum depth of 12. For the linear SGD SVR, we use epsilon-insensitive loss which ignores errors below $\epsilon = 1e^{-3}$ and is linear past that, the same loss used by RBF SVR. We limit SGD to 2000 iterations. For gradient boosted trees (XGBoost), we use 100 trees with max tree depth of 12.

Type	**Train Error**	**Test Error**	**Qualitative Performance**	**Example**
Linear SVR	0.79	0.86	Poor	No dice in cup
MLP	0.63	0.76	Poor	No dice in cup
RBF SVR	0.49	0.61	Okay	3 dice in cup
Rand. Forest	0.28	0.49	Poor	2 dice in cup
XGBoost	0.13	0.17	Very Poor	No dice in cup
Linear SVR w/ DaD [33]	–	–	Poor	No dice in cup
k-NN w/ DaD (Sections VI-E & VI-E2)	–	–	Okay	3 dice in cup

TABLE I

REGRESSOR PERFORMANCE FOR DICE STACKING. NONE OF THE REGRESSORS ACCOMPLISHED THE FINAL GOAL OF STACKING THE DICE, AND IN FACT, THEY HAD VARYING LEVELS OF SUCCESS EVEN BEING ABLE TO GET DICE INTO THE CUP. AN INTERESTING RESULT IS THAT GRADIENT-BOOSTED TREES (XGBOOST) ACHIEVES THE LOWEST TRAINING AND TESTING ERROR BY FAR, BUT ONE OF THE WORST QUALITATIVE PERFORMANCES. THE BEST QUALITATIVE PERFORMER IS RBF SVR, WHICH WE FOUND WAS ABLE TO REPLICATE THE SMOOTH, SWINGING LEFT-AND-RIGHT MOTION CHARACTERISTIC OF HUMAN SOLUTIONS. SEE OUR VIDEOS OF RBF SVR AND GRADIENT-BOOSTED TREES IN OUR ACCOMPANYING VIDEO.

D. Supervision with DAgger

While we've implemented many supervised learning approaches to imitation learning, we'd like to augment our solver with DAgger [26]. DAgger allows for experts to correct the classifier over time, fixing the types of errors that the classifier tends to make. However, while we've implemented the technical code path for this approach, our own inability to play the game with a 100% success rate means that our initial few runs of DAgger simply learned bad behavior. This would be correctable by getting better at the game and developing editing tools that would allow us to generate better trajectories in non-real-time.

E. Supervision with DaD

Additionally, we implemented a novel variant of DaD [33, 32], where we learn how to correct the classifier trajectories. This method is unlike the previously published variant for control, where a linear dynamics model was learned and then solved with an optimal linear controller [32]. Instead, we propose a variant of DaD similar to our modified version of RRT in section V-D, where an optimizer is used to learn corrective actions from unusual trajectories. To do this, we first start with a parametric policy as developed in section VI. Then, we automatically increase our dataset. This is done by performing

$$X = X \cup D \tag{10}$$

where X is initialized with our initial imitation learning dataset. The set of labels D is generated by running the policy as in section VI-A, and then using an optimizer (Sec. V-B) to generate examples that correct for mistakes in the classifier trajectory. During each iteration, we generated a new augmented dataset

using

$$D_i = \underset{x_n}{\operatorname{argmin}} ||f(x_n|x_{n_1} \cdots x_0) - \tau_n||_2^2 \quad \forall N \geq n \geq 1 \quad (11)$$

Where x_n is an applied force at time-step n, $f(x)$ is our simulator and τ_n is our recorded trajectory. This can be interpreted as learning (via optimization) a correction that allows each of our recordings to play back corrected (via a learned policy). This formulation allows for the use of any arbitrary function approximator, even a nonlinear one. In theory, if we were to use an arbitrary function approximator and used a sufficiently powerful optimizer, this would allow for automatic correction of any trajectory artifacts seen in using a classifier-based regression function. This can also be seen as a form of automatic data augmentation [13] in a controls setting. In practice, we used 10 windows of increasing size and perform 3 iterations for each window, similar to the time window annealing process used in DaD [32].

1) Parametric DaD: As this method requires a fast classifier, our only parametric experiments were run with The MLP and Linear SVR methods. Unfortunately, even after running this method for an hour, our performance hadn't improved. It is unclear if our choice of function approximator was poor and lacked the ability to learn a sophisticated policy, or if our optimizer returned corrections that we insufficient in the computational time available.

2) Nonparametric DaD: Our final experiment, we tested to see if model capacity was the problem with our previous section and changed our classifier to a k-Nearest Neighbors model [8]. We used $k = 3$ and repeated our proposed method. We started with three very different imitation examples and learned a nearest neighbor classifier to imitate this behavior. In practice, this classifier has a slight mismatch in velocities/times and is unable to perform the task correctly, similar to our results in Section.VI-A. To perform DaD, we run this model over our imitation examples and generate more data by learning corrections by solving equation 11. In this case, our optimizer is CMA-ES run and it is run for 50 iterations. The resulting process learns to perform a shaking and stacking operation that left all three dice in a tower, as shown in figure 6. However, this technique did not work perfectly and resulted in the use of a wall to accomplish this task.

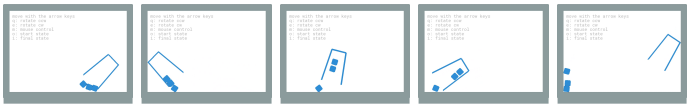

Fig. 6. Example of our nonparametric DaD result. See Sec. VI-E2

VII. Conclusion

We present, to the best of our knowledge, the technical analysis of this interesting manipulation task, dice stacking. These include a dice stacking simulation built using a realistic physics simulator. The simulation is open-source and user-playable, and we describe how we tuned the physics and designed a control scheme to allow a player to intuitively stack dice. We describe experiments involving a variety of supervised and unsupervised methods for teaching a computer to perform dice stacking using our simulation. These include a novel variant of using Data-As-Demonstrator (DaD) for a control task, which is able to provide adequate performance when given a dataset of imitation examples.

VIII. Future Work

To extend these results to better match the physical simulations [14] we're trying to resemble, much more work can be done. For example, we could develop more robust policies which would handle random initial dice and/or cup configurations and still achieve the goal state. Additionally, while we currently apply forces directly to the cup, it would be interesting to instead hook the arm up to a force-controlled robotic arm. In two dimensions we could use a 3 link (4 degrees-of-freedom) arm with a cup as an end effector. Lastly, we could extend the simulation to be a full three dimensional simulation by moving to Bullet [7] instead of Chipmunk [19]; all of the techniques proposed in this paper are general enough to also handle the higher dimensional case.

References

[1] E. W. Aboaf, S. M. Drucker, and C. G. Atkeson. Task-level robot learning: juggling a tennis ball more accurately. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 1290–1295 vol.3, May 1989. doi: 10.1109/ROBOT.1989.100158.

[2] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010. URL http://arxiv.org/abs/1012.2599.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16 (5):1190–1208, 1995.

[5] Erin Catto. Iterative dynamics with temporal coherence. In *Game Developer Conference*, 2005. http://www.continuousphysics.com/ftp/pub/test/physics/papers/IterativeDynamics.pdf.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[7] Erwin Coumans. Bullet physics engine. http://bulletphysics.org, 2018.

[8] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[9] S. Feng, X. Xinjilefu, C. G. Atkeson, and J. Kim. Optimization based controller design and implementation for

the atlas robot in the darpa robotics challenge finals. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 1028–1035, Nov 2015. doi: 10.1109/HUMANOIDS.2015.7363480.

[10] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, 1988.

[11] Sascha Gortz. Dice stacking - super-slowmotion with clear cup - 500fps. YouTube, 2009. https://youtu.be/j6xKTEMPTpw.

[12] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Jonathan Hub. dice stacking. YouTube, 2016. https://youtu.be/7WR9VTCbsE0.

[15] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[16] N Koenig and A Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE.

[17] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998.

[18] Steven M. Lavalle, James J. Kuffner, and Jr. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2000.

[19] Scott Lembcke. Chipmunk2d - a fast and lightweight 2d game physics library. https://github.com/slembcke/Chipmunk2D, 2018.

[20] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[21] P. C. Mahalanobis. On the generalised distance in statistics. In *Proceedings National Institute of Science, India*, volume 2, pages 49–55, 1936. URL http://ir.isical.ac.in/dspace/handle/1/1268.

[22] Andrew T Miller and Peter K Allen. Graspit! a versatile simulator for robotic grasping. *IEEE Robotics & Automation Magazine*, 11(4):110–122, 2004.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. In *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018. doi: 10.15607/RSS.2018.XIV.049.

[25] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[26] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[27] S Schaal and CG Atkeson. Open loop stable control strategies for robot juggling. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 913–918. IEEE, 1993.

[28] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[29] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 2018. ISBN 0262193981. URL http://www.worldcat.org/oclc/37293240.

[30] Russ Tedrake and the Drake Development Team. Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems, 2016. URL https://drake.mit.edu.

[31] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[32] Arun Venkatraman. *Training Strategies for Time Series: Learning for Prediction, Filtering, and Reinforcement Learning*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, October 2017.

[33] Arun Venkatraman, Martial Hebert, and J. Andrew (Drew) Bagnell. Improving multi-step prediction of learned time series models. In *Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*, January 2015.

[34] David J Wales and Jonathan PK Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.

[35] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[36] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.