In *The 40th International Conference on Robotics and Automation* (ICRA 2023) *Workshop on Multi-Robot Learning* London, UK, 2023

Program Embeddings for Rapid Mechanism Evaluation

Sai Kiran Narayanaswami[1] David Fridovich-Keil[1] Swarat Chaudhuri[1] Peter Stone[1,2]

Abstract— **Mechanisms such as auctions, voting and traffic control systems incentivize agents in multiagent systems in order to achieve or optimize certain global outcomes such as safety, productivity, and welfare. Designing mechanisms in the form of programs would bring several benefits such as interpretability, transparency and verifiability. However, program synthesis for finding suitable mechanisms from a search space represented by programs leads to computationally expensive two-level optimization problems, where mechanisms need to be evaluated in an inner loop, which entails learning agent responses for each mechanism. Multi-Task Learning (MTL) approaches efficiently learn parameterized agent strategies that can be used to act in various tasks, here mechanisms defined by programs. Such multi-task agent policies allow for rapid evaluation of mechanism performance for a given program by bypassing the need to learn agent responses from scratch, thus allowing for efficient evaluation of mechanisms. In this work, we use learned program representations (or *embeddings*) to provide suitable task contexts that make MTL feasible with combinatorially large programmatic search spaces of mechanisms. We demonstrate experimentally that program embeddings generated by an off-the-shelf Code2Vec model are sufficient for reconstructing matrix games based on only a programmatic description. The embeddings are also able to serve as task context to guide a set of agent strategies to act near-optimally in mechanisms from the search space.**

I. Introduction

Mechanism design aims to facilitate safe and productive interactions and coordination between agents, both human and artificial. Thus, it is a critical problem in multi-robot systems, human-robot interaction, autonomous vehicles, traffic management and many other domains. Automating mechanism design is crucial for being able to design effective mechanisms in settings involving complex environments and interactions between many agents. At the same time, these complex mechanisms have many human participants and stakeholders, for whom it is important to be able to inspect and analyze the behavior of a mechanism. That is, some form of *interpretability* is also a necessity.

The use of Machine Learning (ML) for mechanism design in complex environments, particularly Deep Reinforcement Learning (RL) has recently gained traction [15], [2], [6], [12], [4], [3]. Such approaches, however, design mechanisms in the form of deep Neural Networks (NN), which are difficult to interpret and analyze. Program Synthesis has been proposed as a solution to this problem. Mechanisms (multiagent environments) are represented as programs in a high-level Domain-Specific (programming) Language (DSL), which can be interpreted by humans much more easily than neural networks, and can also be analyzed and formally verified. Mechanism design is then cast as the problem of finding such a programmatic mechanism from a search space defined by the DSL, such that the mechanism it represents maximizes a global outcome (such as social welfare) when agents learn to maximize their own personal payoffs.

Techniques to convert learned NN policies into programs, notably NDPS [13], open up possibilities for leveraging the efficiency of Deep RL approaches for performing program synthesis. NDPS maintains a pool of candidate programs whose output closely matches that of the target NN policy. The performance of this pool is then evaluated and improved until a program is found that matches the network's performance. However, such evaluation in the setting of mechanism design faces a challenge that stems from the fact that even a small change in the mechanism can cause a large change in the agents' responses. In general, it is also expected that it will not be possible to match the behavior of a given neural mechanism exactly using any program from the DSL. These issues lead to compromised evaluations of candidate programs if agent responses are kept fixed. An alternative is to learn agent behaviors from scratch for each candidate program, so that an accurate evaluation of the mechanism is obtained, which is extremely expensive due to the combinatorial nature of search spaces of programs.

The above dilemma captures the need for a solution to the problem of rapidly finding agent responses to any given mechanism from a programmatic search space. Having a way to do so enables rapid evaluation of any mechanism from the search space, which in turn makes the improvement step in NDPS more tractable.

The aim of this work is to investigate ways to utilize Multi-Task Learning (MTL) as a means to this end. MTL concerns learning policies that are able to perform any task from a given set of tasks. Particularly of interest is the task-conditioned setting where tasks have parameters or equivalently, some form of "task ID" that are given as additional context input to the policy, as these tend to be the most reliable and the easiest to implement. Here, each mechanism from the search space is a task. Rather than just one policy in single agent settings, we would want to learn a task-conditioned policy for each agent.

Although there appear to be a very large number of tasks due to the combinatorial nature of search spaces of programs, it is not necessary to treat each of them as separate tasks. Just like natural language and other data, it is possible to learn vector *embeddings* for programs that serve as compact

[1]The University of Texas at Austin
[2]Sony AI
Email: nskiran@utexas.edu

representations of their structure and behavior [1], [14], [11]. We investigate if these program embeddings could serve as the task context that guides the agents' policies.

The key insight is that by training only one set of multi-task agent policies guided by program embeddings, it would be possible at test time to evaluate any given programmatic mechanism by simply providing its embedding to the policies, and rolling out episodes using them. The embedding network that maps a given program (as source code) to its vector embedding can also be trained offline. Further, once the multi-task policies and embedding network have been trained, they can be reused to evaluate mechanisms according to any performance criterion.

The contributions of this work are as follows:

1) We propose a novel method to enable rapid evaluation of mechanisms from a programmatically described search space.
2) We develop a domain of matrix games for our experiments whose entries are populated by programs from a simple DSL. Although we consider 2-player (bi)matrix games here, our approach is applicable to games with more than two players (tensor games), as well as more general multiagent environments such as Markov games.
3) In this domain, we experimentally demonstrate that:
    - Program embeddings generated by an off-the-shelf, general-purpose embedding network pre-trained using the Code2Vec [1] approach are able to accurately describe mechanisms from a programmatic search space, without any additional training of the embedding network.
    - These embeddings also hold sufficient information for agents to act near-optimally in the mechanism using only the embeddings as task context.

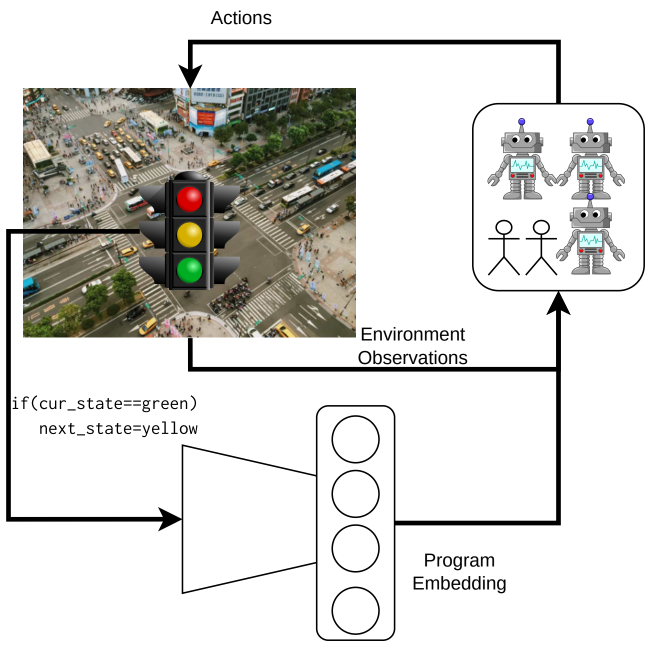

Fig. 1: An illustration of agent learning and acting in the proposed framework. The program representing the mechanism (here defining the operation of a traffic signal at an intersection) is encoded by a neural network (trained separately using standard techniques) into a vector embedding. Agents learn and act with task-conditioned policies that take in both observations from the environment as well as the program embedding as task context. When evaluating a new mechanism after learning, the agent policies can be executed in the same way without the need for learning a separate set of policies.

II. Related Work

Task Conditioned Multi-task RL: Solving multiple tasks using information about the task as additional context input to a policy has been explored in different forms. Goal-conditioned RL [8] considers settings in which each task is to reach a goal. IMPALA [5] is an approach to scale up general task-conditioned RL that also uses a learned representation for natural language instructions as task context. In Multiagent RL (MARL) settings, [10] develops an approach for multi-task learning using recurrent deep Q-networks and a concurrent experience replay scheme.

Deep Program Representation Learning: Program representation learning is the problem of mapping a program (in the form of source code) to a vector representation, also known as a *program embedding*. Code2Vec [1] was one of the earliest works to use deep neural networks to successfully learn meaningful representations of code based on both syntax and semantics. It uses attention over paths in the abstract syntax tree of an input program to generate an embedding. A classifier is trained to predict the purpose of a program method based on this embedding using a labeled corpus of 1 Million Java programs. It is shown in [1] that although trained for a single task, these embeddings, also referred to as *code vectors*, hold information that can be used for more general downstream tasks. Approaches that improve on Code2Vec by more strongly incorporating semantics have also been proposed, such as Liger [14], which uses program execution traces, and NPM [11], which learns embeddings for Hoare Triples of the program.

Mechanism Design using ML: Many recent works have explored the use of deep learning for mechanism design. Deep Learning is used to design auctions using a hand-designed differentiable objective in [4]. Deep RL has also gained traction for being able to handle more generic and sophisticated settings and objectives with minimal manual intervention in the design process [15], [2], [6], [12], [3]. A majority of these works [2], [6], [12], [3] has involved settings where agent responses have been modeled a priori. In contrast, [15] proposes the AI Economist, an approach that does not require pre-defined models for agent behavior. It treats the mechanism as yet another agent acting concurrently with the other agents to optimize the mechanism design objective, and approximates a Nash solution using a MARL algorithm. Thus, it approximates the Stackelberg equilibrium we consider in this work, where the mechanism makes the first "move" to fix its behavior such that the resulting agent behavior leads to desirable outcomes. Doing so brings benefits in that it not only enables the modeling of agents' responses to a mechanism as they learn to maximize their own payoffs, but also circumvents the need for multiple levels of learning (that is, for each different mechanism encountered, learning the agents' behaviors from scratch), thus making it computationally tractable.

III. Methods

A. Problem Formulation

We consider the context of mechanism design over a search space of programs $\mathcal{F}$, as described in [9]. Each mechanism $f \in \mathcal{F}$ is represented by a program from a specific DSL. Strategy profiles and strategies of agents are denoted as $\boldsymbol{\pi} = (\pi_1, \pi_2 \dots \pi_k)$. Mechanism design is then played as a Stackelberg game where the designer makes the first move by choosing the mechanism, and the agents respond by arriving at a strategy profile according to an *(agent) behavior generator*, $\mathcal{B}$, that maps mechanisms from the search space to a distribution over strategy profiles. This behavior generator encodes assumptions about how agents respond to a given mechanism.

An example, which we use in our experiments, is a Nash solver, which outputs some Nash point from the set of all Nash points. The Nash solver could also be probabilistic and approximate, such as a multiagent RL algorithm. Like the above examples, many effective behavior generators cannot provide the entire output distribution, or any associated derivatives. Therefore, we assume that the only available access to the behavior generator is through sampling.

The objective function for the design problem, denoted $\mathcal{J}$ decides how good the mechanism is based on the outcomes when agents act according to the behavior generator. Thus, the mechanism design problem can be stated as finding

$$\arg\max_{f \in \mathcal{F}} \mathcal{J}\left(f, \mathcal{B}(f)\right)$$

In this work, we are concerned with finding an efficient way to evaluate $\mathcal{B}(f)$ for any $f \in \mathcal{F}$. That is, we require a parameterized strategy profile (in the MARL setting, a collection of of multi-task policies for all agents) $\boldsymbol{\pi}(.)$ that takes as input the program associated with a mechanism f, and returns a strategy profile $\boldsymbol{\pi}(f)$ such that $\boldsymbol{\pi}(f)$ is of maximum likelihood under $\mathcal{B}(f)$. In practice, we mainly need $\boldsymbol{\pi}(.)$ to be representative of the equilibrium behavior, and by extension, provide good estimates of $\mathcal{J}\left(f, \mathcal{B}(f)\right)$. Note that by training $\boldsymbol{\pi}(.)$ once, it can be reused to evaluate mechanisms for any choice of the mechanism objective $\mathcal{J}$.

B. Solution

1) Program Representations: A network $\mathcal{M}$ takes in the text source code of the program representing mechanism f, and returns a vector *embedding* $v_e = \mathcal{M}(f)$ of dimension d_e. In this work, we don't consider the training of this network, instead choosing to use a network that has been trained offline, as detailed in the experiments. That is, the weights of the network are not changed during our experiments.

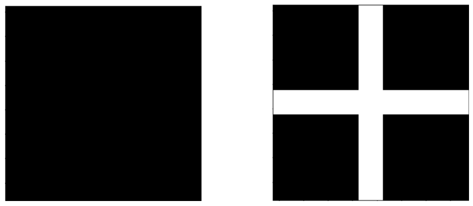
Fig. 2: Visualization of the matrix domain: Matrices are shown as images with black=0 and white=1. The matrix on the right is obtained by executing the program `set(7, 9, 0, 15, 1); set(0, 15, 7, 9, 1);`

2) Task-Parameterized Agent Strategies: Each agent uses a policy network $\pi_{\boldsymbol{\theta}_i}$ that takes as input a state, and the program embedding as task context, and returns action probabilities $\pi_{\boldsymbol{\theta}_i}(a|s, v_e)$. Weights $\boldsymbol{\theta}_i$ could be shared between agents, or independent as we use in our experiments. These networks enable the agent to make use of the program embedding to act in the mechanism.

3) Training: In the general setting, training can be done using MARL with each agent acting to maximize its return. Each episode of training occurs in a different mechanism, sampled at random from the search space. The mechanism's embedding only needs to be evaluated once per episode and can also be cached. A schematic of these steps is shown in Fig. 1. Other training algorithms can also be used, such as Behavioral Cloning or Imitation Learning when an expert (such as an analytical solver, or a planner like MCTS) is available.

IV. Experiments

A. Components

1) Matrix Game Domain: For our experiments, we use a domain of two-player, zero sum games (matrix games), whose entries are populated by programs from a DSL described in the next section. Here, player 1 chooses a row and seeks to minimize the value chosen, whereas player 2 chooses a column and seeks to maximize the value chosen. The matrices are of size 16x16, and each entry is either 0 or 1.

Possible mechanism design objectives in this domain could include functions of the players' costs at Nash equilibrium, and could also take into account how much the program changes the matrix (cost of modifying incentives in the real world).

An advantage of this domain is that a Nash solution can always be found exactly using standard Linear Programming solvers such as simplex and interior point methods, a fact we make use of in our training setup described later on.

2) Domain Specific Language: We use a simple DSL consisting of a sequence of 3 statements of the form:
`set(row_min, row_max, col_min, col_max, value);`

Each of these statements sets the block of the matrix specified by the ranges in the arguments to the given value. The base matrix that it modifies is assumed to be the zero matrix. An example of the execution of a program is shown in Fig 2. An interpreter for the programs was implemented using the ANTLR4 framework.

3) Program Representation: For the program embedding network, we use the Code2Vec [1] approach described earlier. We use the pre-trained model released with [1] without any further training. As our DSL is syntax-compatible with Java, the pre-trained model is able to accept programs from the DSL without any modifications. For this model, the size of the code vectors produced is $d_e = 384$.

4) Search Space: In the following experiments, we restrict our attention to programs with 3 statements as described above (with row and column ranges being being within bounds). As the Code2Vec model is quite large and takes time to evaluate, we use the previously suggested strategy of caching the code vectors for a fixed set of 1 Million programs generated uniformly randomly from the search space.

B. Matrix Prediction

In this first experiment, we investigate whether the code vectors of programs from the search space are able to describe the matrix that would result from the program's execution. To test this, we train a network using supervised learning to predict the resulting matrix using the code vector as the sole input.

1) Network: The network takes the code vector as input and outputs a 16x16 matrix (as a 1-channel image). Its architecture is similar to the decoder parts of standard autoencoder architectures: A linear layer converts the size-384 code vector into 384 2x2 images. A series of transposed convolutional layers with 128, 64 and 32 filters Leaky ReLU activations and Batch Normalization upsample it to size 16x16, and a regular convolutional layer then outputs the predicted 16x16 matrix.

2) Training: It is trained for 40 epochs on the above cached dataset (5% validation set) to minimize the Mean Square Error loss between the ground truth matrix resulting from the program, and the network's prediction.

3) Results: The prediction results are shown in Fig. 3 for programs that were not in the training or validation sets. We observe that:

- The network is able to predict the regions that have value 1 with a high degree of accuracy in number, position, shape and size.
- It is confounded minimally by the presence of blocks of zeros in the background.

The above results show that the code vectors are sufficient to accurately describe the behavior of the program from our DSL, despite the Code2Vec model being used off-the-shelf without any further training.

C. Equilibrium Strategies

Now, we investigate whether the code vectors encode information in a way that enables the agent to act optimally in the mechanism. We test this by training agent policies using behavioral cloning to imitate (mixed) Nash equilibrium strategy profiles based on only the code vector.

1) Network: The input to the network is again just the code vector, as there is only a single state in this domain. The output is a distribution over the number of rows/columns, which are the actions that each agent chooses from.

The architecture used in this experiment uses a similar series of transposed convolutions as in the matrix prediction experiment, but with filters of sizes 32, 64, 128 and 256. This is then followed by a series of strided convolutional layers that downsample the image and result in logits of the policy probabilities. The output is a log-softmax layer representing the action probabilities for each action.

2) Training: Similar to the prediction experiment, it is trained for 80 epochs on the cached data. The loss function being minimized is the KL divergence between the agent policy probabilities and a ground truth Nash strategy for each agent.

We use the `TensorGames.jl` library[7] to exactly solve for a Nash equilibrium, thus allowing us to study the effectiveness of prediction based on the code vectors separately from the multi-task learning component.

Although a large number of Nash equilibria might exist, the aim is to learn to choose any one of them. When designing mechanisms, it is intractable to account for, or even simply characterize the entire space of equilibria. However, a small subset of equilibria can be informative enough, especially when agents are of bounded rationality (e.g limited knowledge and computational power).

3) Results: The predicted strategies for a held-out set of programs are visualized in Fig. 4 for both players. We observe that whenever a row of zeroes is present, the learned strategies for P1 avoid the rows where there are ones present, even when they are very few in number. Similarly, the learned strategies for P2 avoid columns where zeroes are present whenever possible. That is, both players learn to recognize a dominant strategy when one is present. This result shows that the code vector is informative enough to precisely guide the actions of the agents. We emphasize again that the agents need to rely on the code vector as the only available information to choose their actions.

In Fig. 5, we plot the average deviation of player 1's cost under the learned policy from its true value at Nash equilibrium (which is constant across all Nash equilibria) over the course of 3 training runs, evaluated on a set of 1000 random test programs. We find that the best model iterations exhibit strategies that deviate by only 1-2% from Nash strategies in value on average. These results provide further evidence for the viability of using agent strategies guided by program embeddings to evaluate mechanisms.

V. Conclusions

The above results establish that general-purpose program embeddings such as from Code2Vec can offer concise descriptions of a programmatic mechanism in vector form that can be acted upon by neural policies. Particularly, they were successfully used as task context to learn parameterized agent strategies that are representative of equilibrium behavior across a large search space of programmatic mechanisms.

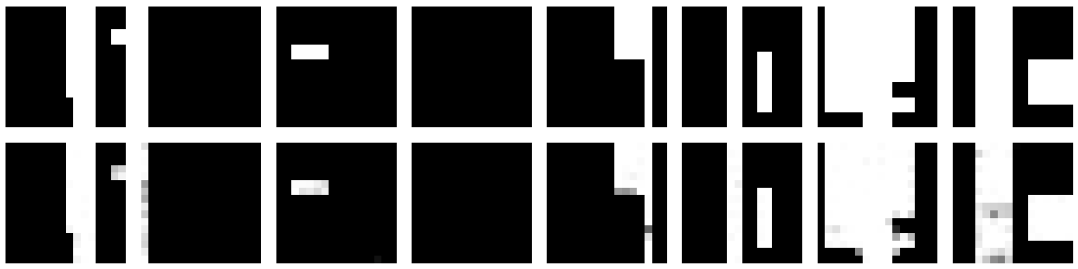
Fig. 3: Matrix prediction results visualized for a set of randomly held-out test programs. The top row shows the matrices generated by the programs (which are not given as inputs to the network), and the bottom row visualizes the predicted matrices. The reconstructions closely resemble the ground truth matrices.

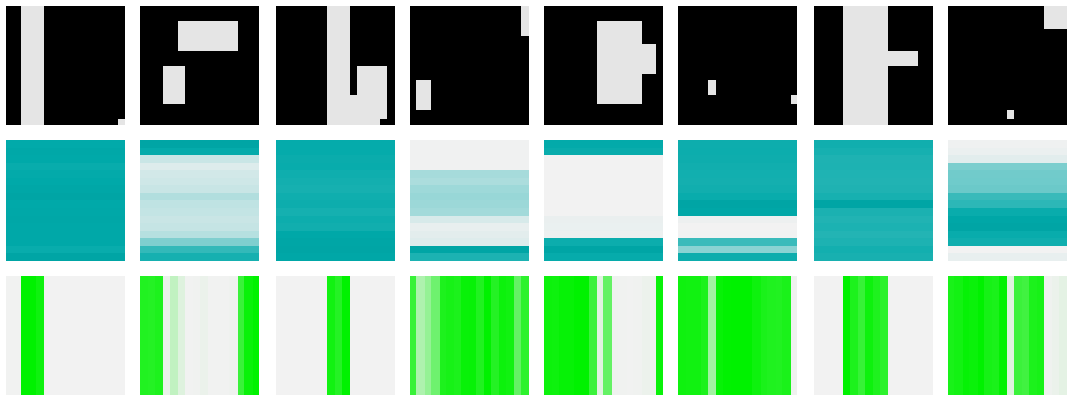
Fig. 4: Learned policy probabilities for a set of random test programs: The top row shows the matrix generated by the programs (note that these are not given as inputs to the network). The middle row visualizes the policy probabilities for player 1. The intensity of the color of each horizontal line represents the probability of player 1 choosing that row of the matrix. The probabilities are normalized so that the largest value for any particular strategy is 1 (darkest). Similarly, the bottom row visualizes the policy probabilities for player 1 for choosing columns of the matrix. We see that both agents have learned to act near-optimally by avoiding rows/columns where the other agent can always win (whenever possible). They do so even when such blocks are very small, such as the rightmost matrix or the third from the right, where player 1 is able to choose.

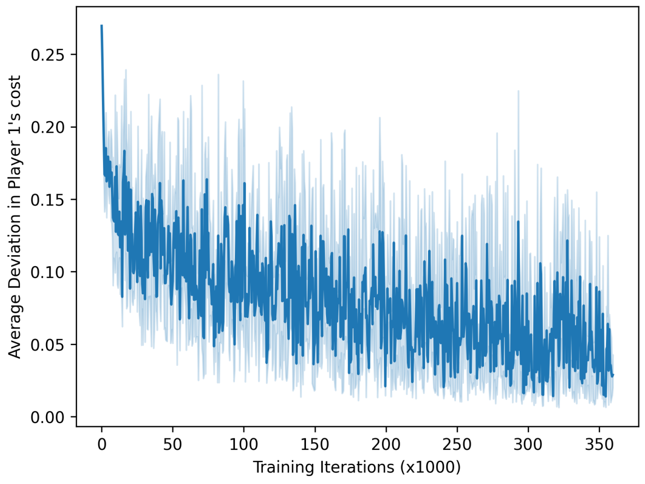

Fig. 5: Average deviation in player 1's cost between learned strategies and the true Nash equilibrium cost across 1000 random test programs over the course of training.

Thus, our results indicate that it will be possible to use such a method to rapidly and accurately evaluate mechanisms, even without using custom-trained program representation networks. Further, since the above experiments studied the properties of the embeddings independently from the learning aspect, the results could possibly carry over to full-fledged sequential multiagent environments.

Therefore, this work has taken important first steps towards realizing a program synthesis approach for mechanism design that is scalable to complex multiagent environments. A natural next step is to obtain similar results through learning from interaction, e.g. MARL, so that the embedding-guided strategies can be learned without the need for expert policies, and in more complex environments. Doing so would allow for fully automated mechanism design via the NDPS procedure as described earlier.

Another question not considered in this work is learning suitable program embeddings for the search space. Large Language Models continue to improve dramatically, making it reasonable to assume that they will improve large-scale, general-purpose program representation learning. Although our results indicate that general-purpose embeddings can be quite informative about the mechanism, it is likely that using embeddings that are more tailored to the search space would result in better scalability to more complex search spaces and environments. One way to train such embeddings is to fine-tune a pre-trained embedding network such as the Code2Vec model we use by training it further on programs from the search space. We might also expect benefits from incorporating not only program syntax and semantics, but also the dynamics of the mechanism into the embedding, possibly by predicting state transitions and rewards as an auxiliary task.

VI. Acknowledgements

This work has taken place in the Learning Agents Research Group (LARG) at the Artificial Intelligence Laboratory, and the Trishul Laboratory at the Department of Computer Science, The University of Texas at Austin. LARG research is supported in part by the National Science Foundation (FAIN-2019844), the Office of Naval Research (N00014-18-2243), Army Research Office (W911NF-19-2-0333), DARPA, Bosch, and Good Systems, a research grand challenge at the University of Texas at Austin. Research in Trishul on this topic was supported by the National Science Foundation (CCF-1704883 and CCF-1918651). The views and conclusions contained in this document are those of the authors alone. Peter Stone serves as the Executive Director of Sony AI America and receives financial compensation for this work. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[2] Gianluca Brero, Alon Eden, Matthias Gerstgrasser, David Parkes, and Duncan Rheingans-Yoo. Reinforcement learning of sequential price mechanisms. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(6):5219–5227, May 2021.

[3] Roberto Capobianco, Varun Kompella, James Ault, Guni Sharon, Stacy Jong, Spencer Fox, Lauren Meyers, Peter R. Wurman, and Peter Stone. Agent-based markov modeling for improved COVID-19 mitigation policies. *The Journal of Artificial Intelligence Research (JAIR)*, 71:953–92, August 2021.

[4] Paul Duetting, Zhe Feng, Harikrishna Narasimhan, David Parkes, and Sai Srivatsa Ravindranath. Optimal auctions through deep learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1706–1715. PMLR, 09–15 Jun 2019.

[5] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1407–1416. PMLR, 10–15 Jul 2018.

[6] Raphael Koster, Jan Balaguer, Andrea Tacchetti, Ari Weinstein, Tina Zhu, Oliver Hauser, Duncan Williams, Lucy Campbell-Gillingham, Phoebe Thacker, Matthew Botvinick, et al. Human-centred mechanism design with democratic ai. *Nature Human Behaviour*, 6(10):1398–1407, 2022.

[7] Forrest Laine. TensorGames.

[8] Minghuan Liu, Menghui Zhu, and Weinan Zhang. Goal-conditioned reinforcement learning: Problems and solutions. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 5502–5511. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Survey Track.

[9] Sai Kiran Narayanaswami, Swarat Chaudhuri, Moshe Vardi, and Peter Stone. Automating mechanism design with program synthesis. In *Proceedings of the Adaptive and Learning Agents Workshop (ALA)*, May 2022.

[10] Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P. How, and John Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2681–2690. JMLR.org, 2017.

[11] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1093–1102, Lille, France, 07–09 Jul 2015. PMLR.

[12] Pingzhong Tang. Reinforcement mechanism design. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 5146–5150, 2017.

[13] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5045–5054. PMLR, 10–15 Jul 2018.

[14] Ke Wang and Zhendong Su. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 121–134, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Stephan Zheng, Alexander Trott, Sunil Srinivasa, David C. Parkes, and Richard Socher. The ai economist: Taxation policy design via two-level deep multiagent reinforcement learning. *Science Advances*, 8(18):eabk2607, 2022.