CS 7643 Project Report: Behavior cloning for Lux AI Season 2

Stephen Huan* Daniel Lu* Vedaant Shah* Jerry Xiong*

1. Introduction

The Lux AI challenge is an AI programming competition in which two teams of robots face off in a resource-gathering simulation. Submissions range from heuristic-based algorithms to machine learning approaches, competing on a continually updating, global leaderboard. In this project, we experiment with a deep learning approach for imitating the behavior of strong-performing agents in Lux AI season 2, which ran on Kaggle from January to April 2023. The project repository can be found at this link.

1.1. Problem description

The competition environment is a two-dimensional grid, where each agent controls a teams of factories and robots (Figure 1). Robots collect resources such as ice and ore from set locations on the map, return them to the factories to refine them into water and metal, and use these resources to build and sustain additional robots. The game has three phases: a bidding phase, in which agents bid starting resources to determine the factory placement order; a placement phase, where agents take turns selecting initial factory locations; and finally, the main phase of the game, where robots are built and controlled. The final objective is to maximize the growth of a resource called lichen using water collected throughout the main phase.

1.2. Motivation

A number of competitors utilize heuristic-based agents. These approaches rely on significant understanding of the environment dynamics, as well as human intuition. For example, agents might rely on path-finding algorithms, or heuristics to delegate robot responsibilities. Such approaches can scale poorly in terms of computational cost as the number of agents increases. In addition, modifying such an approach requires manually changing the code of the underlying program.

In this project, we examine an deep learning approach based on imitating the behavior of other agents. This approach can not only leverage data from a wide variety of other teams' strategies, but also, since robot actions are select via a single pass of neural network inference, our approach has constant computation cost with respect to the number of agents. Additionally, by incorporating existing architectures designed for semantic segmentation of images, we can leverage a large body of knowledge relevant to pixel-level prediction on two-dimensional grid inputs.

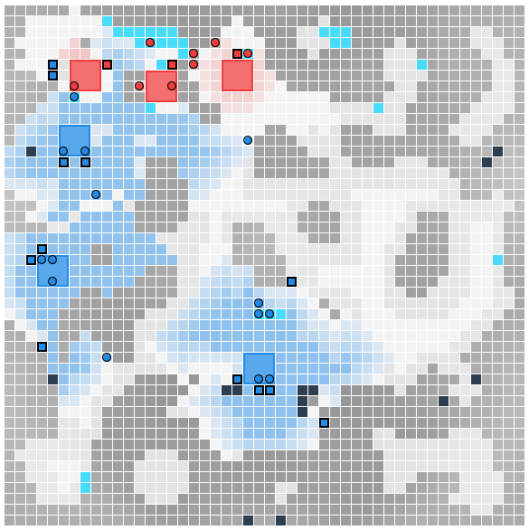

Figure 1. An example game board state. The resources are ice, ore, and rubble is shaded depending on whether it is light or heavy. Factories are the 3×3 squares with lichen growing from them, and robots are the 1×1 squares and circles (same for the other player).

We leverage the vast amount of replay data available through the public Meta Kaggle dataset [17]. By downloading replays through Kaggle's API, we can extract in-game trajectories of states and actions, as well as the agent ratings, numbers representing an approximate skill level used for matchmaking.

2. Approach

A single match between two teams is separated into three phases: the bidding phase, the setup phase, and the main phase. In the bidding phase, each team bids a certain amount of resources in order to decide the turn order for the setup phase. For the sake of simplicity, we choose to ignore the bidding phase, and simply bid 0 resources every game.

*Georgia Institute of Technology, Atlanta, GA

In subsection 2.1, we discuss a modeling methodology for the setup phase based on Gaussian Process regression. In Appendix C, we discuss a black-box hyperparameter tuning strategy that needs only games between agents. For the main phase of the game, we utilize an approach inspired by image segmentation, which we describe in subsection 2.2.

2.1. Kernel methods for setup phase

During the setup phase, players take turns selecting a 3×3 area which fits within the limits of the board, that does not directly overlap ice or ore, and is at least 6 tiles from the center of any other factory. At a high level, our goal during the setup phase is to pick the "best" factory locations that are close to desired resources, away from the opponent's factories, and so on.

Compared to the main phase of the game, which involves trajectories of up to 1000 states and actions, each replay only provides around 3-5 data points relevant to the setup phase. Out of 200,000 total states, we extracted only 203 starting states. Thus, it is prohibitively expensive to collect a large amount of data for the setup phase and so we considered neural networks infeasible at this level of data scarcity.

Instead, we decided to use a kernel method akin to *Gaussian processes* (GPs), a nonparametric model which directly induces a probability distribution over functional spaces through its *mean* and *kernel* function, in order to predict the score of each cell and place a factory at the highest scoring cells. Prior information and inductive biases like smoothness can be injected directly into the kernel information. As a result, GPs have seen widespread usage in machine learning [16] and can be also be seen as the limit of infinite-depth Bayesian neural networks [3].

Specifically, we say a function $f(\boldsymbol{x})$ is a Gaussian process with mean function $\mu(\boldsymbol{x})$ and kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$ if for any set of points $X = \{\boldsymbol{x}_i\}_{i=1}^N$, $f(X) \sim \mathcal{N}(\boldsymbol{\mu}, \Theta)$, where $\mu_i = \mu(\boldsymbol{x}_i)$ and $\Theta_{i,j} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Given features $X_{\text{Tr}} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]^\top$ with targets $\boldsymbol{y}_{\text{Tr}} = [y_1, \ldots, y_N]^\top$, we wish to predict the values at new points X_{Pr} for which $\boldsymbol{y}_{\text{Pr}}$ is unknown. To make predictions at X_{Pr}, we condition the desired prediction $\boldsymbol{y}_{\text{Pr}}$ on the known data $\boldsymbol{y}_{\text{Tr}}$. For Gaussian processes, the closed-form posterior distribution is

$$\mathbb{E}[\boldsymbol{y}_{\text{Pr}} \mid \boldsymbol{y}_{\text{Tr}}] = \boldsymbol{\mu}_{\text{Pr}} + \Theta_{\text{Pr,Tr}}\Theta_{\text{Tr,Tr}}^{-1}(\boldsymbol{y}_{\text{Tr}} - \boldsymbol{\mu}_{\text{Tr}}), \quad (1)$$

$$\mathbb{C}\text{ov}[\boldsymbol{y}_{\text{Pr}} \mid \boldsymbol{y}_{\text{Tr}}] = \Theta_{\text{Pr,Pr}} - \Theta_{\text{Pr,Tr}}\Theta_{\text{Tr,Tr}}^{-1}\Theta_{\text{Tr,Pr}}. \quad (2)$$

However, computing the conditional mean (1) requires inversion of the covariance matrix of the training data, which scales as $\mathcal{O}(n^3)$ for n training points. In principle we could accelerate this with state-of-the-art numerics [20], but for simplicity we propose the following approximation: first, without loss of generality we can assume the mean is $\mathbf{0}$ since a nonzero mean will shift every cell equally, preserving the relative rankings. Second we propose approximating

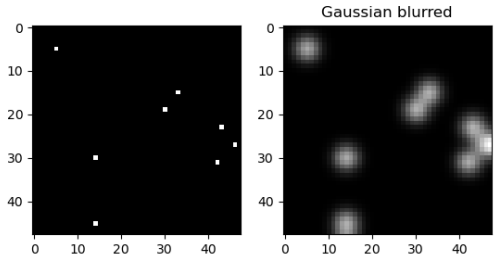

Figure 2. True factory positions (left) after smoothing (right).

$\Theta_{\text{Tr,Tr}}^{-1} \approx \text{Id}$ which is equivalent to assuming training cells are statistically independent. We can intuitively interpret

$$\mathbb{E}[\boldsymbol{y}_{\text{Pr}} \mid \boldsymbol{y}_{\text{Tr}}] \approx \Theta_{\text{Pr,Tr}}\boldsymbol{y}_{\text{Tr}} \quad (3)$$

as a weighted linear combination of training labels, with weights given by the similarity of each training point to the target point. We use a separate Matérn kernel for each salient feature of the board (ice, ore, and rubble) for

$$\boldsymbol{y}_{\text{Pr}}^{\text{final}} = c_{\text{ice}}\boldsymbol{y}_{\text{Pr}}^{\text{ice}} + c_{\text{ore}}\boldsymbol{y}_{\text{Pr}}^{\text{ore}} + c_{\text{rubble}}\boldsymbol{y}_{\text{Pr}}^{\text{rubble}}. \quad (4)$$

Other than the coefficient in (4), each Matérn kernel has two hyperparameters, a length scale which controls to what degree father points influences the prediction, and a smoothness which controls the regularity of sampled functions. We therefore have 12 hyperparameters to learn.

We implement both the estimator and the hyperparameter tuning in the `scikit-learn` [14] framework. We use random search with uniform priors over all coefficients:

- Coefficients: $[-1, 1]$
- Length scale: $[0, 20]$
- Smoothness: $\{1/2, 3/2, 5/2, \infty\}$

Since we do not have trainable parameters, we do not perform cross validation and simply take the highest scoring set of parameters. We score parameters by their mean squared error (MSE) from (4) to the one-hot encoded true factory positions. We use Gaussian smoothing to regularize the objective, encouraging the model to place factories at *similar* locations to the ground truth (Figure 2).

[coefficients, length scales, smoothness]	MSE
[0.061, 0.857, -0.001], [4.21, 1.91, 13.5], [0.5, 0.5, 1.5]	0.024
[-0.016, 0.460, -0.063], [9.15, 2.75, 0.218], [inf, inf, 0.5]	0.054
[0.648, 0.434, -0.170], [5.56, 2.33, 0.548], [2.5, 2.5, 0.5]	0.069
[0.660, 0.270, -0.319], [3.83, 11.3, 0.541], [0.5, 0.5, 0.5]	0.108

Table 1. Best hyperparameters out of 500 samples for 4 different seeds. For each parameter group, the order is [ice, ore, rubble].

The best parameters out of 500 samples for 4 different seeds are shown in Table 1. The model generally encourages placement near ice and ore while discouraging proximity to rubble. Through the length scale the model considers distant ice, nearby ore, but only extremely close rubble. However, significant variation in the absolute magnitude of parameters as well as in smoothness suggests the problem is severely underspecified, probably due to data scarcity.

In addition, we believe that the model may be underscoring valuable resources like ice and ore due to its inability to encode nearby factories. In the course of an ordinary game, the first player might place their factory near ice and ore, thus blocking them from the second player. Thus factories are artificially far from each other. In future work we may want to collect data at a more fine-grain level or model existing factories as electric charges and add an energy penalty.

2.2. Data and model architecture

For the main phase of the game, we utilize a behavioral cloning[15] approach. Specifically, our goal is to predict the actions taken by each robot that a top-ranking submission would likely perform. As such, we use the Kaggle API to download replays of games involving submissions from the top 100 ranked competitors on the leaderboard. The replay data contains the actions taken by each robot across all time steps in the game. We thought that this would lead to a successful strategy since we would be imitating the already successful, though unpublished, strategies based on their replays. To achieve this, note that during the main phase of the game, each robot on the board takes actions as determined by the overall agent/player. As such, our model's objective can be formulated as to correctly predict the actions taken by the robots for the replays in our dataset.

Our model needs to predict the appropriate action taken by each robot on the board. Specifically, this involves predicting the type of action (e.g. a move to an adjacent position, a transfer of resources from one robot to another, a pickup of resources from a factory, etc.), the type of resource for which the action is being performed, and the quantity of the resource involved (for transfer/pickup actions). The treat the prediction of action and resource types as classification tasks, whereas resource quantity is treated as a regression problem.

As such, the structure of our problem is that we are given a representation of the current board state, which includes the locations of the factories, resources, and robots. Clearly, this leads to a 2D-grid structure spread across multiple channels. Likewise, the output is to predict the values described above for certain positions in the board, also resulting in a 2D-grid structure. This problem can naturally be interpreted as a form of image segmentation, which is generally the idea of predicting from a discrete class of values for each pixel in an image.

Thus, we design our model in a similar method to image segmentation, except with the modification that we have three output heads mentioned above, and not just a pixel-wise single classification objective. Additionally, rather than a 3-channel input like with images, our input contains 37 channels due needing to account for all items on the board.

Therefore, we structure our model with an image segmentation backbone architecture, such as U-Nets [18] and LRASPP [8], followed by three fully-connected portions for the output heads at each position in the board. For the backbone, we consider using a simple linear model as a baseline, which is implemented via a single convolutional layer with a kernel size of 1. We also consider U-Net, which is an improvement over encoder-decoder fully connected convolutional networks for segmentation due to the presence of skip connections and multiple up-sampling layers rather than only 1. Lastly, we consider LRASPP, which is a lightweight image segmentation model designed to be run on mobile devices. We do so since we are not using the pre-trained version of any models (as it does not make sense to use models trained on actual images for a problem that is not involving images) in this problem, and thus the larger U-Net may not be able to learn efficiently on our dataset.

In total, all components related to the segmentation and output heads are learnable. The only non-learnable parameters of this model include our pre-processing to convert the raw replays into PyTorch tensors for the model to take as input and output (for training), as well as the logic for converting the model's output into decisions to make in-game, which can be done by using the argmax for the classification outputs and rounding to the nearest integer for the regression output.

The main problem that we anticipated with approach was feature scaling for real-valued inputs in the board, such as the number of a given resource on a position. Since these values could potentially get large and thus hinder the model's accuracy, we instead used the log of all real-valued features, a trick which has been used in other settings [6]. Additionally, we anticipated an issue with balancing our loss function to accurately account for all 3 output heads, which we addressed by weighing the loss in section 3. Lastly, there was a class imbalance as certain actions were taken much more frequently than others in our replays. We attempted to address this using the recall-based class weighting approach suggested in [21], but found that this approach did not have a significant impact on performance.

3. Training

For training our model on the main phase of the game, we use the same loss function regardless of the underlying segmentation architecture. Specifically, since our model needs to predict the type of action, resource involved, and

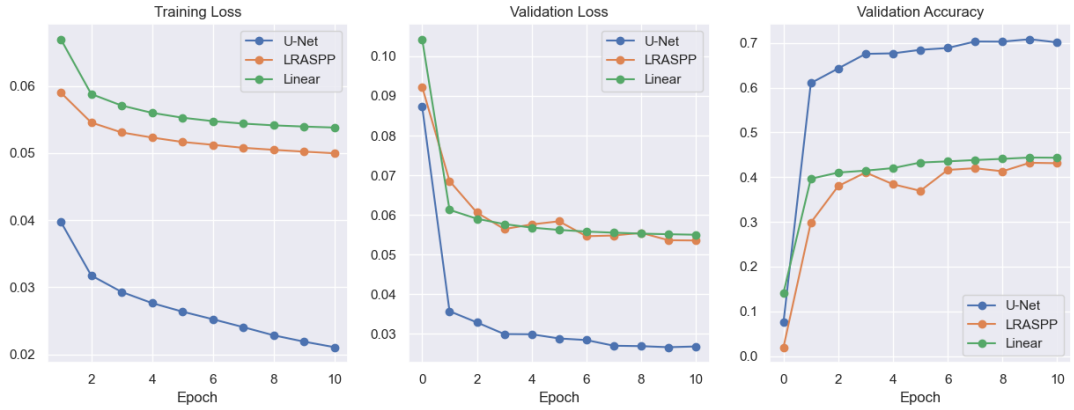

Figure 3. The training loss, validation loss, and validation accuracy of our model for each of the three architectures.

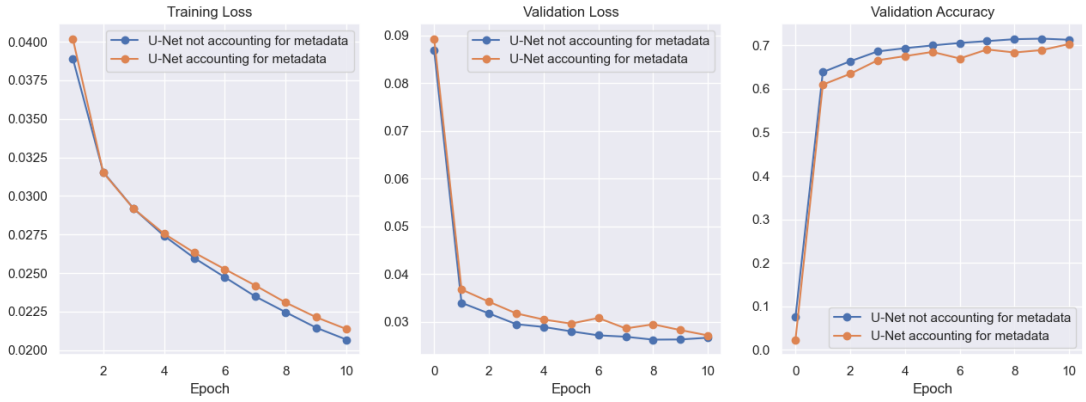

Figure 4. Curves for training loss, validation loss, and validation accuracy of our approach with the U-Net architecture, comparing performance with and without metadata.

quantity at each position, our loss function can be written as

$$L = L_{\text{action type}} + L_{\text{resource}} + \alpha L_{\text{quantity}}. \tag{5}$$

Prediction of action type and resource are both classification problems, so we perform pixel-wise cross-entropy loss for both, taking the mean across the entire board and applying a 0/1 mask, so that the loss only includes positions at which there is a robot. For predicting the quantity, we have a regression problem, so we use pixel-wise mean squared error, taking the sum across the entire board and applying a 0/1 mask again. α is a non-negative hyperparameter used to weigh the MSE loss, in order to prevent it from dominating the cross-entropy loss.

The hyperparameters of our model mainly include the learning rate, MSE weight term, and the coefficient of weight decay. We tune hyperparameters via a coarse grid search up to 1 step per decade. We trained using the Adam optimizer with a minibatch size of 16 for 10 epochs. We ended up using a learning rate of a learning rate of 1e-3, weight decay of 1e-6, and $\alpha = 1/100$, which we found performed relatively well on all architectures.

We used `PyTorch` [13] for our model implementation, and obtained implementations of the baseline semantic segmentation architectures from `Torchvision` [12]. We did not use pretrained weights in any form, as our domain does not deal with a visual image-based input in the traditional sense.

3.1. Conditioning on Demonstrator Quality

We consider passing in various types of metadata available in the replay files, in addition to the board state itself. This includes information about the two players' ratings, as well as the difference in final score (the accumulated lichen), which determines the game winner. We add these scalars to the input by broadcasting over the entire input plane and appending them as additional image channels. This approach was loosely inspired by the Return-To-Go (RtG) input provided during the training of Decision Transformers [4]. Essentially, by treating the problem as a sequence modeling task and conditioning on the reward

received in the remainder of the episode, the decision transformer learns to model both effective and ineffective actions. By including data about player ratings and the future match outcome into our training data, and conditioning on *high* player ratings or *winning* match outcomes at test time, we hope to attain similar benefits.

Remark We initially planned on using a reinforcement learning approach for this problem rather than the presented supervised learning model, and in Appendix C, we experimented with a more rigorous dueling bandit approach to hyperparameter tuning. However, we shifted away from reinforcement learning due to challenges with the LuxAI environment, and thus for this supervised learning problem, simply evaluating hyperparameters on a validation set is a more scalable approach with our magnitude of training data.

4. Experiments and Results

Our primary measure of success was to consider the accuracy of our model on the validation dataset in predicting the action type, resource, and quantity (see above) for each robot on the board. We considered this metric under the various semantic segmentation architectures mentioned earlier. The training and accuracy curves can be seen in Figure 3. It should be worth noting that we are not using the pretrained version of these models since they are pretrained for images; although this task involves a pointwise prediction on grid-like structures, they are not images. As such, we consider all such architectures rather than relying on whichever performs best on benchmark image datasets. Based on the figure, the U-Net architecture has the strongest performance, while the performance of LRASPP is somewhat similar or slightly worse than the linear model.

4.1. Ablation: metadata conditioning

We conduct an ablation study of the additional metadata input channels described in subsection 3.1 by comparing performance including the metadata and performance without (where the initial channels are set to zero). To isolate the impact of this change, we test with the only best-performing U-Net architecture.

The training curves comparing the two approaches are depicted in Figure 4. There does not seem to be a significant difference in performance with and without the metadata. We discuss possible interpretations and alternative approaches in section 5.

5. Discussion

Of the 3 semantic segmentation architectures tested with our model for the main phase, the U-Net performed the best. The poor performance of the baseline linear model is unsurprising, due to the limited model capacity. However, LRASPP surprisingly performed quite poorly, even relative to the linear approach. One possible explanation is that during inference, the LRASPP architecture predicts at a lower resolution before bilinearly interpolating the output to a higher resolution. Although this may be a reasonable architectural decision for typical tasks involving semantic segmentation of images (after all, adjacent pixels likely belong to the same object class), in this problem space it is quite common for robots in adjacent tiles to take entirely distinct actions, thus reducing LRASPP's effectiveness.

In subsection 4.1 we observed that conditioning on submission ratings and match outcomes did not improve training or validation performance. This outcome is not entirely unexpected, as we do not take the sequence modeling approach from [4], so similar modeling decisions may not lead to strong improvements in the domain of single-step modeling. In addition, our source of data was replays from the top 100 agents, during a snapshot of the leaderboard relatively early on in the competition. At that point, most of the agents were likely deterministic, heuristic-based agents, whose behavior did not change much regardless of how well the team was doing at a particular phase of the game, and was of course fixed regardless of the submission's current rating.

5.1. Analysis: post-competition comparison to other approaches

By combining the kernel methods for the setup phase with the segmentation approaches for the main phase of the game, we technically have an agent capable of competing in the Lux AI environment. However, despite the high overall accuracy on the offline dataset, we ran into several difficulties when deploying the model online. For example, the agent would often fail to mine sufficient quantities of ice to survive past turn 150.

After the competition ended on April 24th, we can compare our approach with various solutions other teams decided to publish. We noticed that the imitation-based solution posted here for example, was quite similar to our work. Notable difference from our method include

- A simplification of the action space: they only predicted action classes, and used hardcoded parameters for those actions (target resource and quantity).
- No demonstration heterogeneity: all of their replays came from a single, reinforcement-learning based submission, while we sampled uniformly from the top 100 players.
- More sophisticated postprocessing: they masked out invalid actions, as well as friendly inter-robot collisions.

- Heuristic algorithm for the setup phase, based on human intuition.

Essentially, for the main phase of the game, their approach puts strict limitations on the output space of the model, reducing the difficulty of the problem. A lack of demonstration heterogeneity may lead to reduced strategic diversity, but also meant that their targets would be more consistent across their dataset.

6. Conclusion

In this project, we explored a deep imitation learning approach to the LuxAI Season 2 competition. Specifically, we used a kernel method approach for the setup phase and an image segmentation approach to the main phase. We experimented with multiple segmentation architectures and found that U-Nets worked best for this task. Furthermore, we also considered usage of the game's metadata values in our model to see if there would be further improvement.

References

[1] Arpit Agarwal, Rohan Ghuge, and Viswanath Nagarajan. An Asymptotically Optimal Batched Algorithm for the Dueling Bandit Problem, Sept. 2022. 9

[2] Arpit Agarwal, Rohan Ghuge, and Viswanath Nagarajan. Batched Dueling Bandits, Feb. 2022. 9

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York, 2006. 2

[4] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *CoRR*, abs/2106.01345, 2021. 4, 5

[5] Moein Falahatgar, Yi Hao, Alon Orlitsky, Venkatadheeraj Pichapati, and Vaishakh Ravindrakumar. Maxing and Ranking with Few Assumptions. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. 9

[6] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models, 2023. 3

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. 10

[8] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019. 3

[9] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, May 2007. 10

[10] Junpei Komiyama, Junya Honda, Hisashi Kashima, and Hiroshi Nakagawa. Regret Lower Bound and Optimal Algorithm in Dueling Bandit Problem, June 2015. 9

[11] Junpei Komiyama, Junya Honda, and Hiroshi Nakagawa. Copeland Dueling Bandit Problem: Regret Lower Bound, Optimal Algorithm, and Computationally Efficient Algorithm, May 2016. 9

[12] TorchVision maintainers and contributors. TorchVision: PyTorch's Computer Vision library, Nov. 2016. 4

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. 4

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011. 2

[15] Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991. 3

[16] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, Mass, 2006. 2

[17] Megan Risdal and Timo Bozsolik. Meta kaggle, 2022. 1

[18] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. 3

[19] Aadirupa Saha and Pierre Gaillard. Versatile Dueling Bandits: Best-of-both-World Analyses for Online Learning from Preferences, Feb. 2022. 9, 10

[20] Florian Schäfer, Matthias Katzfuss, and Houman Owhadi. Sparse Cholesky factorization by Kullback-Leibler minimization. *arXiv:2004.14455 [cs, math, stat]*, Oct. 2021. 2

[21] Junjiao Tian, Niluthpol Chowdhury Mithun, Zachary Seymour, Han-Pang Chiu, and Zsolt Kira. Striking the right balance: Recall loss for semantic segmentation. *CoRR*, abs/2106.14917, 2021. 3

[22] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, Mar. 2020. 10

[23] Michael L. Waskom. Seaborn: Statistical data visualization. *Journal of Open Source Software*, 6(60):3021, Apr. 2021. 10

[24] Huasen Wu and Xin Liu. Double Thompson Sampling for Dueling Bandits, Oct. 2016. 9

[25] Yisong Yue and Thorsten Joachims. Beat the mean bandit. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 241–248, Madison, WI, USA, June 2011. Omnipress. 9

[26] Julian Zimmert and Yevgeny Seldin. Tsallis-INF: An Optimal Algorithm for Stochastic and Adversarial Bandits, Mar. 2022. 9

[27] Masrour Zoghi, Zohar Karnin, Shimon Whiteson, and Maarten de Rijke. Copeland Dueling Bandits, May 2015. 9

Student Name	Contributed Aspects	Details
Stephen Huan	Setup phase, hyperparameter tuning, report writing	Kernel methods, dueling bandit algorithms
Daniel Lu	Training, experimentation, report writing	Tuning framework, architecture, initial classifier
Vedaant Shah	Training, architecture, report writing	Overall training and U-Net implementations
Jerry Xiong	Dataset, preprocessing, report writing	Created the dataset and preprocessing pipeline

Table 2. Contributions of team members.

A. Source code

Code for all experiments can be found at the repository https://github.com/jxiong21029/LuxS2.

B. Work division

The contributions of each team member is as follows. See also Table 2 for a high-level summary.

Stephen Huan I primarily worked on analyzing factory placement in the setup phase and developing a possible hyperparameter tuning strategy. For the setup phase I decided to use a kernel method inspired by Gaussian processes, and coded the estimator, hyperparameter tuning, and integration into the final agent. I also worked on developing the hyperparameter tuning algorithm with dueling bandits, implementing four different papers. Correspondingly, I wrote subsection 2.1 and Appendix C of this report.

Daniel Lu I worked on training and hyperparameter tuning. This involved integrating our chosen models, dataset, and parameter specification into a custom tuning framework, as well as performing the training and creating visualizations. Additionally, I worked on a classifier for an early iteration of the setup phase inspired by Monte-Carlo Tree Search and contributed towards applying the chosen architectures. In the report, I focused on the introductory section 1 (subsection 1.1 and subsection 1.2).

Vedaant Shah I worked on creating the logic for training and evaluating our model for the main phase of the game. This included writing the loss function, the training loop over the dataset, and the evaluation function I also implemented the U-Net architecture in our code as well as the logic for the model using the metadata. I trained and obtained the results for the model with the U-Net architecture both with and without the metadata. As such, I wrote significant portions of subsection 2.2, section 3, and section 4.

Jerry Xiong In the early phases of the project, I explored a deep reinforcement learning based approach for this environment. The bandit-based hyperparameter tuning approach I encouraged Stephen to work on was an offshoot of this effort. However, we eventually decided that this was outside the scope of what we could accomplish in this timeframe. After pivoting to an imitation learning approach, my main contributions were downloading and processing the dataset into a format consumable by neural networks, including the board state and metadata inputs and the action targets. I also experimented with the recall-based action class-weighting approach. I contributed to section 1, subsection 3.1, subsection 4.1, and section 5 of the report.

C. Dueling bandits for hyperparameter tuning

Initially, we implemented a reinforcement learning approach for the main phase of the game with an agent playing and learning from its actions. Due to challenges with the Lux AI environment, we switched to the segmentation-based supervised learning approach discussed in subsection 2.2. In this setting, different agents can be compared by their supervised loss. Here we present our initial ideas of comparing agents with only black-box game evaluations.

Given two agents, the most natural way to compare them is to play them against each other. However, because of the inherent stochasticity in the game and agent strategies, this is a relatively noisy comparison, perhaps requiring multiple comparisons to determine which is better. We consider determining the best agent from K agents in the least number of comparisons, as each comparison requires an computationally expensive simulation of the game. These K agents will eventually be models with different hyperparameters.

We model this in the *multi-armed dueling bandit* framework. While we could use existing probably approximately correct (PAC) algorithms for maxing and ranking [5], often they make stringent assumptions on the problem structure like strong stochastic transitivity (SST). Instead, we will seek nearly assumptionless algorithms. Since we can run multiple games in parallel, the *batched* algorithms of [2, 1] are compelling, but we believe the overhead of adapting a sequential algorithm to the parallel setting is not significant.

Following the setup of [24], consider K arms. For the t-th timestep the algorithm selects two arms $(a_t^{(1)}, a_t^{(2)})$ and receives win (1) or loss (0) based on some preference matrix p such that $\mathbb{P}\{i \succ j\} := p_{i,j}$ where $i \succ j$ means that i won. We assume $p_{i,j} + p_{j,i} = 1$ and $p_{i,i} := 1/2$ and we say an arm i *beats* arm j if $p_{i,j} > 1/2$.

There are many notions of a "best" arm, here we consider two. A *Condorcet* winner is an arm that beats all other arms, and may not exist for every preference matrix. A natural relaxation that always exists is a *Copeland* winner, which is the arm(s) that beats the most number of other arms. Note that a Condorcet winner (when it exists) is always a Copeland winner but the converse is not true.

Specifically, define the (normalized) Copeland score to be $\zeta_i := \frac{1}{K-1}\sum_{j\neq i} \mathbb{1}(p_{i,j} > 1/2)$ where $\mathbb{1}(\cdot)$ is the indicator function. Let $\zeta^* := \max_{1\le i\le K} \zeta_i$ be the maximum score; an arm i is a Copeland winner if $\zeta_i = \zeta^*$ and a Condorcet winner if $\zeta_i = 1$. We define the expected cumulative *Copeland regret* of a dueling bandit algorithm to be

$$R(T) := \zeta^* T - \frac{1}{2}\sum_{t=1}^{T} \mathbb{E}\left[\zeta_{a_t^{(1)}} + \zeta_{a_t^{(2)}}\right] \tag{6}$$

or intuitively, the accumulated difference from the optimal arm over T timesteps. Since the regret (6) penalizes unnecessary comparisons, it can be applied to many notions of regret. We also define the similar *Condorcet regret* [19] where the arm $a^{(\mathsf{cw})}$ is the Condorcet winner as

$$R^{\mathsf{cw}}(T) := \frac{1}{2}\sum_{t=1}^{T}\left[\Delta(a^{(\mathsf{cw})}, a_t^{(1)}) + \Delta(a^{(\mathsf{cw})}, a_t^{(2)})\right] \tag{7}$$

where $\Delta(i,j) := p_{i,j} - 1/2$ is the optimality gap between arm i and j. We now give an overview of existing methods:

- Beat the mean (BTM) [25]
- Scalable Copeland bandits (SCB) [27]
- Relative minimum empirical divergence (RMED) [10]
- Copeland winners RMED (CW-RMED) [11]
- Double Thompson sampling (DTS) [24]
- Versatile dueling bandits (VDB) [19]

BTM assumes a total ordering, i.e. $a_1 \succ \cdots \succ a_K$, a (relaxed) stochastic transitivity in the sense that for some $\gamma \ge 1$ and triplet of arms $a_i \succ a_j \succ a_k$, it holds $\gamma\Delta(i,k) \ge \max\{\Delta(i,j), \Delta(j,k)\}$, and a stochastic triangle inequality in the sense that $\Delta(i,k) \le \Delta(i,j) + \Delta(j,k)$. It has regret $\mathcal{O}(\frac{\gamma^7 K}{\Delta^*}\log T)$ where $\Delta^* := \min_{i\neq a^{(\mathsf{cw})}} \Delta(a^{(\mathsf{cw})}, i)$ which can be adapted to find a (ε,δ)-PAC optimal bandit $\hat{a}$ in the sense that $\mathbb{P}\{\Delta(a^{(\mathsf{cw})}, \hat{a}) > \varepsilon\} \le \delta$ with sample complexity $\mathcal{O}(\frac{\gamma^6 K}{\varepsilon^2}\log\frac{KN}{\delta})$ for $N := \lceil \frac{36\gamma^6}{\varepsilon^2}\log\frac{K^3 N}{\delta}\rceil$.

With probability $1-\delta$ SCB finds a ε-Copeland winner $\hat{a}$ in the sense that $1-\zeta_{\hat{a}} \le (1-\zeta^*)(1+\varepsilon)$ with accumulated total regret of the form $\mathcal{O}(K\log K\log T)$.

RMED finds a Condorcet winner with (asymptotically optimal) regret $\mathcal{O}(\sum_{i\neq a^{(\mathsf{cw})}} \frac{\log T}{\Delta(a^{(\mathsf{cw})},i)})$ but includes an undesirable $\mathcal{O}(K^2)$ dependency. CW-RMED is an extension of RMED to the Copeland winner with similar performance.

DTS uses Thompson sampling twice to select both arms. DTS+ is a slight improvement which cleverly breaks ties. DTS achieves a Copeland regret of $\mathcal{O}(K^2\log T)$ but this analysis is probably not tight as it performs well in practice.

VDB uses a simple reduction from dueling bandits to independent standard multi-armed bandits. It uses the "best-of-both-world" algorithm Tsallis-INF [26] that achieves both the optimal (pseudo-)regret $\mathcal{O}(\sum_{i\neq a^{(\mathsf{cw})}} \frac{\log T}{\Delta(a^{(\mathsf{cw})},i)})$ in the stochastic case and $\mathcal{O}(\sqrt{KT})$ regret in the adversarial case where at every timestep the preference matrix $p_{i,j}$ is allowed to change in response to the bandit algorithm. VDB basically inherits these regret bounds but its analysis depends significantly on the existence of a Condorcet winner.

We consider four variants of VDB depending on whether reduced-variance (RV) estimators are used for the loss [26] and whether the bandits share information by re-using the same online mirror descent update (remark 2 of [19]).

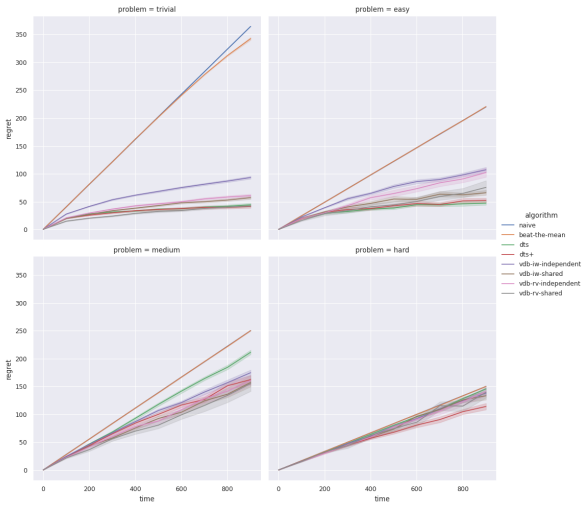

Figure 5. Regret for $T = 1000$, 50 trials per timestep.

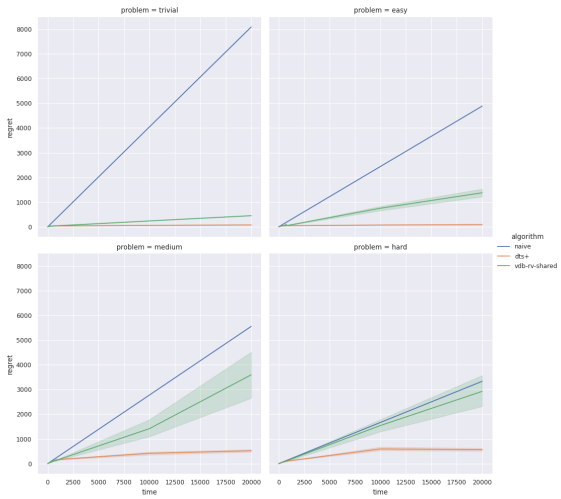

Figure 6. Regret for $T = 20,000$, 10 trials per timestep.

We implemented BTM, SCB, DTS and DTS+, and the four aforementioned variants of VDB. Results for SCB are not shown due to difficulties with its experimental results.

We consider four experimental setups, in order of decreasing structural assumptions on the preference matrix p:

- "trivial": We use the Bradley–Terry–Luce (BTL) or Plackett-Luce model [19]. Given a list of real scores $r_1, \ldots, r_K$, the probability arm i beats arm j is defined as $e^{r_i}/(e^{r_i}+e^{r_j})$. This guarantees a total ordering (and therefore a Condorcet winner), stochastic transitivity, and the stochastic triangle inequality. We sample r_i independently and identically (i.i.d) from $\text{Unif}(0, 100)$.

- "easy": We generate a preference matrix p with entries taken i.i.d. $p_{i,j} \sim \text{Unif}(0, 1)$ for $2 \leq i < j \leq K$. We then take $p_{1,j} \sim \text{Unif}(1/2, 1)$ for $2 \leq j$, guaranteeing that the arm $i = 1$ is a Condorcet winner.

- "medium:": We generate random matrices with entries i.i.d. $p_{i,j} \sim \text{Unif}(0, 1)$ for $1 \leq i < j \leq K$ until p has multiple (> 1) Copeland winners.

- "hard": We generate a random matrix like medium but without resampling for multiple Copeland winners.

For all setups we take $K = 10$ arms and shuffle the preference matrix after every trial to remove possible bias from the ordering. Experiments were implemented using standard Python scientific libraries `numpy` [7] and `scipy` [22]. Plots were produced with `matplotlib` [9] and `seaborn` [23], which generated the shaded confidence intervals.

As shown in Figure 5, DTS+ performs the best or close to the best over all setups, and of the VDB variants the reduced-variance estimator with shared information is the best. BTM is barely better than a naive strategy which simply samples every pair of arms evenly. For the more difficult setups, the regret of DTS+ and VDB look linear despite their theoretical regret of $\mathcal{O}(\log T)$. Running for a longer time horizon (Figure 6) shows that DTS+ achieves log scaling while VDB's regret still looks linear.

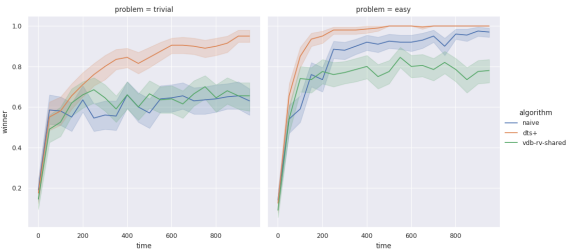

Figure 7. Winner recovery rate for $T = 1000$, 200 trials.

Finally, Figure 7 shows the percentage chance the algorithm recovers the Condorcet winner. DTS+ performs significantly better than the naive strategy, while VDB is about the same or even worse on the harder problem setup.

We think that existing algorithms focus too heavily on (1) regret minimization and (2) asymptotic scaling. In our use case, we only want to recover the best arm and do not mind making poor comparisons along the way. We also cannot afford taking $T \to \infty$, instead, we need good performance for $T \ll 10^3$. Whether an algorithm with these performance characteristics exists is an open question.