Parallel Exploration of Reward Functions in Catan using Containerized Deployments

Wasif Ul Islam
islam25@purdue.edu
w.islam@obscuritylabs.com

John Michael Van Treeck
jvantree@purdue.edu

Diego Reyes Olmos
dreyesol@purdue.edu

December 06, 2024

Objective

To investigate varying reward functions and associated hyper-parameters for a reinforcement learning (RL) agent exploring a 2-player variant of the board game Settlers of Catan ("Catan"), while exploring the benefits and drawbacks of using state-of-the-art containerized software deployment in the training process.

Figure 1: Image of Catan Board Game - the basis of the Reinforcement Learning problem

Background and Motivation for Catan

Board games offer an effective way to investigate reinforcement learning algorithms since there is a clear agent – the player – and environment – opponents, the board, and the rules of the game. Catan is an interesting game to model because of its complexity and non-deterministic features. In Catan, up to four players compete to build settlements, roads, and cities on a board of hexagonal tiles. Each tile produces one of five possible resources: wood, brick, sheep, ore, and wheat. At the start of each turn, players will roll the dice, determining the resources they can collect, and trade with other players. The resources are used to either purchase development cards or build structures, which results in earning victory points. The game is won by the first player to reach 10 victory points (VPs). A Image of the board game can be seen above in Figure 1.

The game is non-deterministic for a variety of reasons: dice rolls, development cards, the "robber" mechanic, and player choices. When a player rolls the dice, the sum of the roll indicates which hex tiles produce resources, which adds stochasticity to the resources collected throughout the game. The resources in a player's hand and the board state will dictate which subset of actions is available to the player on their turn, like where they can build or if they can purchase a development card. If a player chooses to purchase a development card, it is randomly drawn from a stack of cards. When a 7 is rolled the "robber" token can be moved, eliminating a tile's resource production, and players with too many resources must remove half their resource cards. All these probabilistic events make the Markov Decision Process (MDP) highly stochastic, and no single policy or set of actions can create a predictable outcome. Since the game is multi-player, the state transitions are not only influenced by the agent but by the actions and negotiations of all players. Finally, the inability to see another player's cards means that there is hidden information, which introduces an element of partial observability in the environment, which can make it difficult to provide complete state representation of the MDP.

Prior Work

Some RL research has already been done on versions of the game, like optimized player trading [1], a 2-player variant [2], or a simplified board state [3]. Multiple researchers have used Actor-Critic methods, where the Actor evaluates the state of the game and takes an action, while the critic provides feedback to the

agent by estimating the value function [4]. Using the Actor-Critic method has both succeeded in beating a baseline heuristic AI called Jsettlers [2] and failed to achieve a successful win rate against an opponent selecting actions at random (Random AI) [3]. Although both researchers have used similar architectures, the results have varied between these two investigations significantly. We believe the outcome depends not only on the architecture selected, but also on how the reward function is designed. Researchers that have used other Deep Learning Architectures, like Deep Q-Networks (DQNs), have pointed out the dependence of the reward function [5], especially as DQN models relies directly on the Bellman consistency equation [6]. By investigating different reward functions and other hyper-parameters, we can improve on previous research and evaluate how to best approach large state-space strategies. Table 1 below is an example of some rewards investigated by other researchers using actor-critic and DQN architectures. Each investigation made simplifications to the game and board state, such as using 2 players instead of 4 players[2],[3], removing trading[3],[7], removing randomization of hex tile placement [7], reducing the number of hex tiles[3], or using heuristics to handle portions of the action space[5].

Architecture	**Reward Function**	**Win Rate**
Actor-Critic (2 players) [3]	±10 win/loss +1/VP, -0.02/pass	0.18 (Random Agent)
Actor-Critic (2 players, self-play) [2]	±0.75 win/loss, +0.02ΔVP	0.58 (*Jsettlers*)
DQN excluding placement* (4 players)[5]	+Resource gained +Resource Diversity +Building ± win	0.32 (colonist.io)
DQN with Attention (4 players) [7]	±500 win/loss, +5/VP, +1 robber, -0.3 discard	0.33 (Random Agent)

Table 1: Reward Functions and Performance of previous Models used for *Settlers of Catan*.
*[5] did not disclose reward values, only categories

Game and Environment Adaptations

To focus on different reward structures, some concessions were made to simplify the game and speed up training time. One main concession was to remove partial observability. Rather than hide all opponent information, the agent will have knowledge of opponent resource cards and number of development cards, but not what each development card is. This will significantly decrease the complexity of the environment and make it easier to develop an optimal policy.

Since Catan can have multiple players, the game could either be a multi-agent problem or the opponents can be integrated into the environment. Since we are focused on the reward structures, we chose to simplify the environment to one opponent that is a "random" AI. In literature, we found reports showing that training against an opponent following a heuristic model slightly improves win-rate compared to training against a "Random AI" when both compete against the heuristic model during testing [1]. So, we are confident that starting with a Random AI will provide good directional information. Next, we decided to remove player-player trading. Player-Player trading alone could be its own optimization problem, attempting to model when it's in the best interest for a player to make simple, complex, or decline a trade. However, since trading is such an integral part of the game, we left the ability to trade with the "bank" or harbors. Trading directly with the bank requires giving up 4 of a single resource for 1 of any other resource. Harbors are locations on the board that allow discounts on this ratio for a specific resource. For example, a Wheat harbor may offer a, 2:1 ratio instead of 4:1.

During our initial investigation, we found several GitHub repositories that used different versions of the Catan environment, but we found one that met the simplifications we were looking to make [8]. In addition to our simplifications, the author adapted the board game to go from 19 Hexagonal tiles to a grid system that is 21 x 11 squares. This still successfully incorporates all possible board locations but allows a matrix to easily represent the board state. A representation of the grid system can be seen in Figure 2.

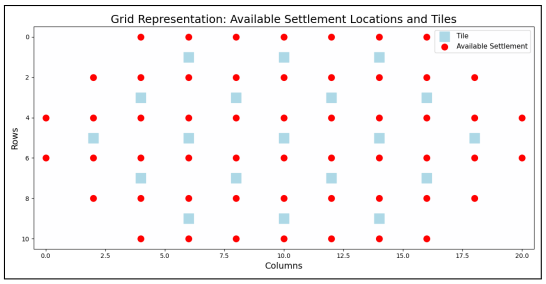


Figure 2: Grid Representation of the Catan Board Game, and all possible settlement locations highlighted with a red dot.

A few additional adaptations were made to the environment provided, mostly due to bugs in how the environment was operating and reducing the complexity of the environment. The first was to remove the option to control discarded resources. When a 7 was rolled, and a player had too many cards (> 7), they are required to remove half their cards. The implementation of this initially allowed both agents to select resources to keep; however, there was a bug in the mechanics that allowed the RL agent to continue to select the same resource card continuously and never end its turn. For time, we updated this game mechanic to be apart of the environment, and instead of selecting the resources to keep/discard, they were randomly discarded for both players. The second change was removing the *Road Builder* development card. This development card typically allows the player to automatically build 2 roads for free. Both the random agent and the RL agent had a predisposition to place both roads on-top of each other, since the board was not being properly updated in between actions. So for time, we removed it from the development card deck, and replaced it with knight cards.

With all of these changes in mind, the state and action space was analyzed before finalizing any architectures. The state space is broken up into 2 sections: a board state space, and a player state space. The information can be seen in Table 2. The board state is a 23x21x11 matrix, where harbor, resource, settlement, road, robber, dice probabilities, etc. locations are documented and updated. The player space consists of all information in the hand of the player and the opponent. This includes resources of both players, available roads, cities, settlements, development cards available to the player, and total development cards of the opponent. This means the encoded state space is 23x21x11 + 41 = 5,354. The categories of the state space can be seen in Table 2.The action space consists of all possible board and player actions. Board actions consist of placing the robber, city, settlement, or road on the 11x21 grid. So board actions alone total 4x11x21 possible actions. Player actions consist of all the other actions that don't take place on the board: buying development cards (1 action), playing a development card (14 possible actions), trading with the bank or a harbor (20 actions), and ending the players turn. This means the action space has 960 possible actions.

Board States	**Player States**
Resource tiles	Agent's victory points
Resource Probabilities	Each player's Resources
Trade harbor locations	Development Cards played
Settlement locations	Development Cards available
City locations	Sum of Opponent Development Cards
Road locations	Army size of all players
Robber location	Longest road of all players

Table 2: State Space of Catan Environment.

Agent Architecture and Training

Two architectures were provided in the initial repository: an asynchronous advantage actor-critic (A3C) method and a DQN [8]. The proposal for the project included plans to use an actor-critic architecture, but multiple asynchronous actors playing simultaneously increased the complexity and resource requirements for containerized deployment, so we shifted to fixing and implementing the DQN architecture. The DQN equation is based off the Bellman optimality equation, seen in Equation 1 [9].

$$Q^*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right] \quad (1)$$

Using an off-policy, temporal difference design, the DQN updates a loss function, $L(\theta)$, by comparing the results of the policy's neural network Q_θ to a target network Q_{θ^-}. We initially evaluated both a L_1 regularization and a mean square error (MSE) loss function, but found better results with MSE, so we kept it in our final

training environment. Equation 2 and equation 3, are the implementations we used for our target net and our loss function. Since the target network uses older θ values (denoted as θ^-), we chose to update the parameters every 100 episodes during training.

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a') \tag{2}$$

$$L(\theta) = \mathbb{E}_{(s,a,r,s' \sim D)} \left[(y - Q_\theta(s,a))^2 \right] \tag{3}$$

In order to pull from a distribution of trajectories when calculating the loss function, we implemented Replay, which allows us to store trajectory steps in a memory bank. During the loss calculation and optimization step, a batch of randomly selected trajectory steps are pulled from the memory bank and used to calculate the loss. By implementing batch sizes and Replay, the training would be much more stable.

Three variants of the DQN architecture shown in Figure 3 was proposed for investigating different reward structures. Each variant of the architecture takes in the board and player encoded state space as two different inputs, mixes them through different layers, and generates board actions and player actions. Convolutional layers were leveraged because of their parameter sharing to evaluate the large board space without having an extremely large set of fully connected layers. The board space also had a residual connection before being flattened and sent through a final fully connected layer. The convolutional layers used batch normalization and ReLU activation. The player state space vector (34x1) was fed through a set of fully connected layers with ReLU activation. The output of the DQN model was a concatenation of the board and player actions, providing state-action values for each of the 960 possible actions.The first variant, deemed "Small DQN" can be seen in Figure 5, and totaled 366,644 parameters. The second variant added a max pooling layer in the convolutional blocks and an average pooling layer in the Residual blocks. This was done to try and find different ways to emphasize the board state. The "Small DQN with Pooling" has 366,955 parameters. The final DQN Variant "Medium DQN" was a scaled up version of the Small DQN, adding a few extra connected layers, and increasing the size of the hidden layers and output dimensions of the convolutional layers. The Medium DQN has a total of 2,124,472 parameters. All of these are a reasonable size for training locally if necessary.

During initial testing, training periods would run thousands of episodes as the game tried to learn to play legal actions as well as a winning strategy. For time, we decided to implement a masking system, which would remove illegal moves from the resulting output of state-action values, so as to only allow the RL agent to make legal decisions. In addition to creating a mask, we also created a scheduled rate of exploration, shown in Equation 4. This way the environment can be explored heavily early and then slowly settle to a reasonable amount of exploration, like 5%. An example of this can be seen in Figure 4.

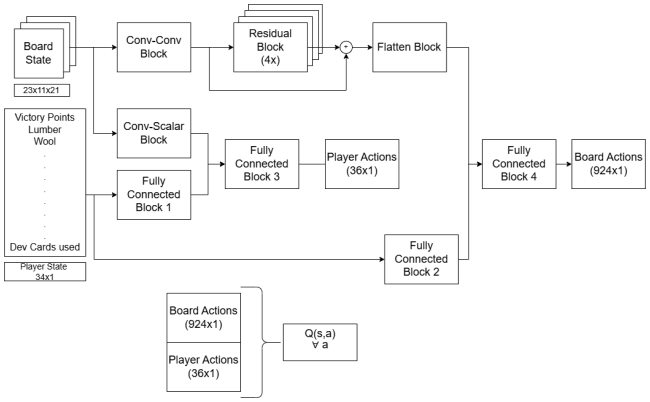


Figure 3: DQN architecture broken into building blocks

$$\varepsilon = \varepsilon_{Final} + (\varepsilon_0 - \varepsilon_{Final}) e^{(-\frac{Episode}{DecayRate})} \tag{4}$$

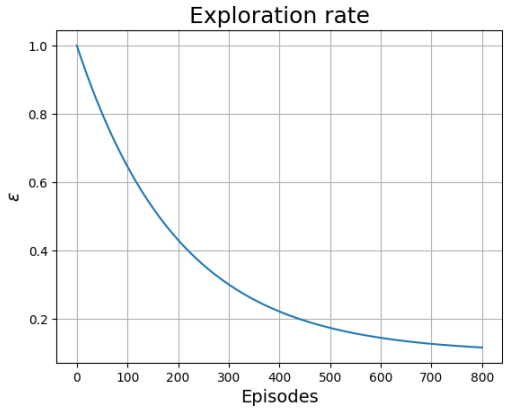


Figure 4: $\varepsilon(t)$, when $\varepsilon_{Final} = 0.1, \varepsilon_0 = 1$, and $DecayRate = 200 Episodes$

Proposed Reward Structures

In addition to the baseline sparse reward structure(+1 for winning the game, and -1 for losing the game), the reward functions displayed in Table 3 were investigated. The reward functions were selected to investigate different levels of sparsity and magnitude. We wanted to understand how much influence increasing feedback would influence the training of the RL agent.

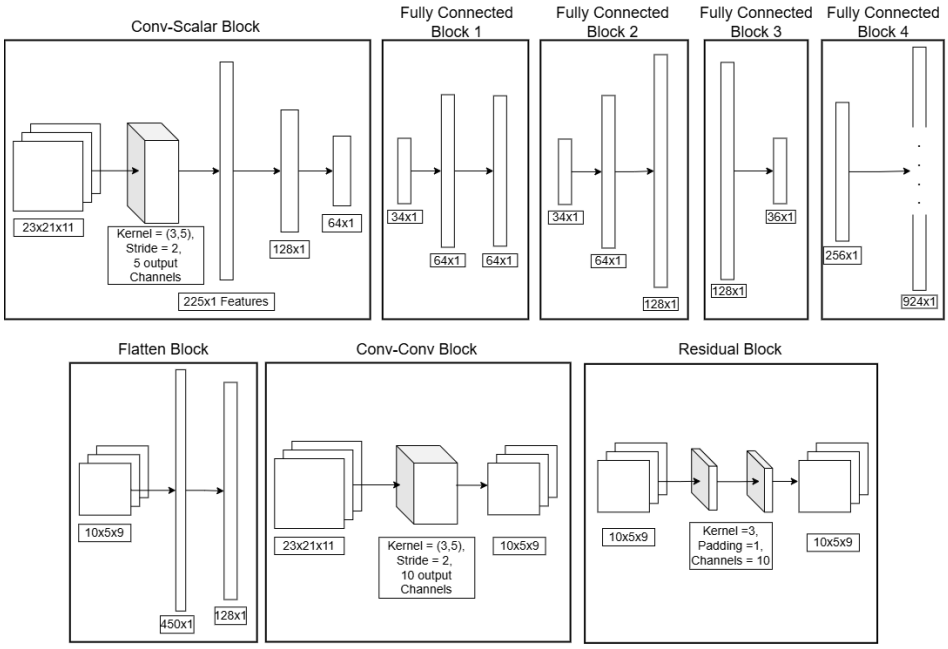


Figure 5: Each DQN building block used the Small DQN.

Reward Structure	**Rewards**	**Rationale**
Large Magnitude	+10 win -10 defeat	Increase responsiveness of rewards by making it larger and balance sparsity with magnitude
Differential VP	$\pm(0.02(\Delta VP)+0.75)$	Differentiate reward by how ample the victory was. Losing by a few VP is better than by a larger margin. Encourages RL agent to get more Victory points than opponent
Incremental VP	$+1$ for win, -1 for defeat, $+0.1(\Delta VP)$ for change in victory points	Maximize reward by having more victory points than opponents, create a rubber-band effect to encourage "catching up" when behind, and reduce sparsity in reward function
Rewards for actions	$+0.001$ for Resources, $+0.01$ for Dev Cards, $+0.01$ for Roads, $+0.01$ for Knights, $+0.1$ for Settlements , $+0.2$ for Cities, ± 5 for win/lose,	Provide rewards all the time of various scale to add perception of value to all consequences of actions. Does reducing sparsity in this way help?

Table 3: Proposed Reward Structures for RL Agent.

Stochastic Policy Implementation

After some initial tests with the default configurations, it was observed that using a deterministic policy often conducted to a drastic reduction on the win-rate, usually a result from having the end your turn action hold a high value, as this action needs to be performed regularly to win the game, which leads a deterministic policy to fixate on this action. To mitigate this situation, the idea of using a stochastic policy was suggested, which would allow the agent policy to constantly end its turn to obtain resources, while some times picking other high value actions, with increased probability the more valuable they are judged by the DQN. This also has a positive effect on the exploration vs exploitation

rate, as the decrease in epsilon reduces the exploration of actions, but the stochastic policy keeps the exploration of the actions already considered "valuable". Another positive effect is on the training stability, as the expected reward in the following state is stabilized by the use of an expectation over the policy, instead of the maximum possible reward in the next state, which often has outliers due to the sparse nature of the rewards in the game, and destabilizes the gradient calculation. to select an action with the stochastic version of the policy, equation 5 is used to create the stochastic policy. The softmax function is used to calculate probabilities for each of the legal actions based on the predicted values of each action for the current game state, and then randomly select an action from with a distribution created with the calculated probabilities.

$$\pi(a_i|s) = \frac{e^{Q_\theta(a_i,s)}}{\sum_j e^{Q_\theta(a_j,s)}} \tag{5}$$

The proposed stochastic variation to train each of the DQN Networks, is to use a stochastic version of the training and target policy. The deterministic policy uses Equation 2 to calculate the labels for training and the training loss, while the stochastic version of the policy uses equation 6.

$$y = r + \gamma \mathbb{E}_{(a' \sim \pi^T)} \left[Q_{\theta^-}(s', a') \right] \tag{6}$$

once the y labels for training are calculated, the stochastic version also uses equation 3 to calculate the loss used in the back-propagation algorithm for training the DQN.

Experiment Design

To perform the agent training in the parameterized Kubernetes environment, the base code was heavily redesigned to improve readability, modularity, and other performance enhancements that arose from experimenting with the code:

- **Benchmarking Method**: A benchmarking method was created to simulate games using the trained DQN networks. It generates policies and plays against a random agent. This benchmarking calculates the trained policy win-rate and can be configured with a minimum win rate to abort training if the policy reaches an unviable win-rate.
- **Containerized Training Loop**: The train loop was containerized in another method, allowing for Training, Benchmarking, or running a Train-Benchmark loop in the environment.
- **Policy Strategies and Reward Functions**: The code was modified to use different policy strategies (deterministic or stochastic) and configurable reward functions.
- **Main Function with Hyper-parameters**: A main function that can be called with arguments was added. These arguments are the hyper-parameters to be changed on each node environment to test the effects on the trained agent. The hyper-parameters added to the main function are as follows:
  - **TRAINING_LOOPS**: Number of times the experiment will train and benchmark an agent.
  - **EPISODES_PER_LOOP**: Number of games that will be played in each training loop.
  - **GAMES_PER_BENCHMARK**: Number of games that will be played to benchmark the agent policy against a random policy.
  - **MEMORY**: Number of past state-action pairs that will be stored for training. The batch for model optimization will be randomly picked from this memory.
  - **MODEL_SELECT**: The DQN model to be trained. Three models have been defined: small, small with pooling, and medium.
  - **REWARD_FUNCTION**: The reward function to be used for training. It can be Basic, Large Magnitude, Differential VP, Incremental VP, and All Incremental.
  - **STOCHASTIC**: Whether the policy generated from the DQN model predictions will be stochastic or deterministic.
  - **LR_START**: Initial learning rate at the start of the experiment.
  - **EPS_START**: The initial epsilon for the e-greedy policy used during exploration.
  - **EPS_END**: The end value of the exponential decay for the epsilon.
  - **EPS_DECAY**: The decay rate for the epsilon. The larger the value, the slower it will decay.
  - **GAMMA**: The discount for each step on the expected reward calculation.
  - **BATCH_SIZE**: The number of state and expected reward pair samples that will be used to run an optimization of the DQN step.
  - **PRINT_ACTIONS**: Whether the actions will be printed in the training log, which can help with debugging issues.

– **MLFLOW_ADDRESS**: The address for logging the artifacts from the experiment using MLFLOW.

Workflow Scaling Motivation

Generally, academic training is an intrinsically serial process where the model is defined, the hyper-parameters are defined as a configuration object, and the training and validation workflow executes on the defined model and hyper-parameters. In general, training progress is tracked per serial run completed. This implementation offers a technical demonstration of deploying the training workflow which decreases total training and validation time of the model, increased frequency on validation parameter updates, and improved batch artifact storage. The work attempts to execute on the above functional requirements by satisfying the following non-functional requirements:

- Parallel concurrency on training and validation workflows
- Distributed compute platform over event driven messaging between server, client, and task executors
- Standalone state tracking platform for model workflows
- Build and environment isolation

Although parallel workflow scaling is a very attractive concept in the current context, historically it is a challenging prospect when considering the software development overhead of implementing a process safe, and thread safe workflow, especially when considering transitive, or cross-platform dependencies in the overarching system architecture. As a result, an alternate approach to achieving the concurrency requirement is fielded. Horizontal scaling of compute resources at the operating system with respect to quantized workflow definitions allow for single-threaded/processed workflows to be scaled. While there are multiple implementation methods that could be used, such as hardware virtualization, runtime containerization is used as the implementation path. Orchestration of containerized runtime will allow for necessary controls to scale according to defined behavior. In general, the aim is to scale the number of training and validation workflows by a user defined amount using multiple hyper-parameter requests to a web-server accepting job requests. Details of the implementation will be discussed later.

Kubernetes was selected as the container orchestration platform for its production heritage and its extensible implementation of the orchestrator control loop. The simplest deployment of Kubernetes require a single host system running a Linux based operating system with an applicable container runtime (such as containerd, podman, or CRI-O), known as a cluster node. The control plane of the node provides interfaces where developers can define how the node can measure current state, identify desired state, and perform reconciliation actions. Applications or workflows can be declaratively defined as runtime containers, which are built in isolated environments while adhering to design principles allowing for the runtime orchestrator to apply the reconciliation loop. For the purpose of this work, it allowed for the web server accepting model hyper-parameters, model workflows, and the Event-Driven Architecture (EDA) to be defined separately in isolation. Additional resource types are considered to aid in augmenting the project's ability to properly train and track the training of feature approximators. Kubernetes allows for custom resource extensibility, allowing for custom logic within the declarative control loop, with respect to both applications and the underlying resources. These objects are known as custom resource definitions (CRDs)[10]. Often, a CRD may contain multiple resource objects. However, the value in defining a CRD lies in the operator definition, the resource type within Kubernetes responsible for enforcing custom controls on declarative application/workflow definition.

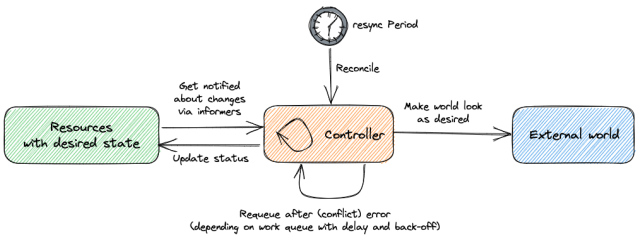


Figure 6: CRD Control Loop [11]

System Architecture

As prescribed in the previous section, the deployment of the training and validation workflow will follow a EDA based on a publisher and queue consumer implementation. The user will interact with a back-end web-server over a compliant HTTP client (either `curl` over a CLI or a web-browser). For the current implementation of the deployment, the user will send a `POST` request

with the defined hyper-parameters to the web server. The web server will then publish a message containing the hyper-parameter encoding to a deployed message queue. Registered task runners will consume available messages on the message queue, and process the associated training and validation task. As part of the processing, the results will be published to an experiment tracking server allowing for visualization. In addition, once the job is finished, the registered task runner will send back a status message through the message queue to the web-server, which gets relayed back to the client ensuring full task completion. The software architecture below outlines the above process:

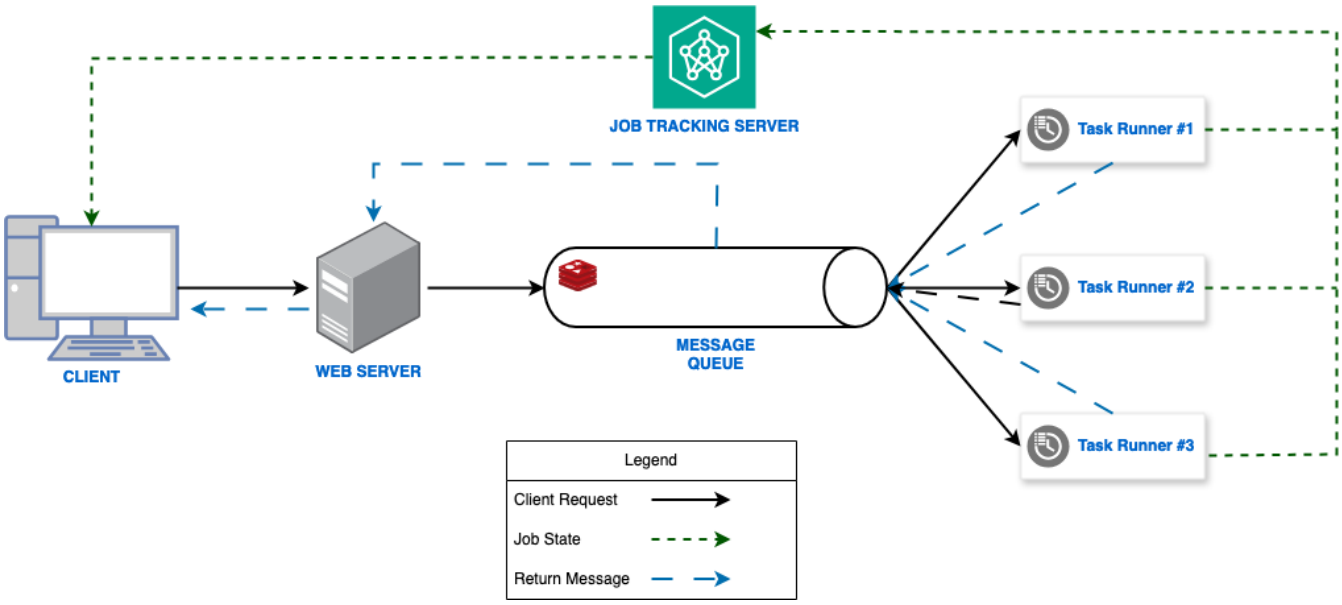


Figure 7: Software Architecture: Persistent Resource

In the above scenario, the task runners are not scalable as discrete resource allocation is required. If three task runners are defined, then that is the maximum level of parallelization that can be achieved without Kubernetes. In addition, if the user does not require parallel workflow execution capabilities, the resources consumed by the task runners cannot easily be reallocated for other tasks or spun down. This is where the desired state reconciliation behavior of Kubernetes comes into play. By defining a CRD that can poll the message queue, we can scale the task runners based on how many messages are available in the message queue. In addition, we can set parameters on how many messages before scaling triggers, maximum number of jobs to scale to, and availability period before resource reclamation by node management. The control plane of the cluster handles the networking stack for all discrete resources, so each task runner can dynamically communicate with the tracking server without networking configurations.

Workflow Scaling Implementation

A Linux virtual machine was defined within an existing workstation. The following are the VM workstation specifications:

- **VM Software**: VMWare Workstation Pro 17
- **CPU**: 8 Logical Cores
- **RAM**: 16 GB
- **Persistent Storage**: 100 GB
- **Networking**: VMWare NAT (Bridged to Host)

NOTE: NAT networking is necessary in order to access services exposed by the virtual machine on the host operating system.

As Kubernetes is an SDK supporting the development of a container orchestrator, there exists different distribution of Kubernetes based on application use case. For example, RedHat OpenShift, or OpenSUSE Rancher are enterprise cluster distribution where there are a catalog of features built to provide support at a large enterprise level. K3s is a production grade low resource

overhead distribution generally used in small and simple application deployments. As a result, the current cluster implementation is a K3s distribution. The setup process is automated by using vendor provided quick-start script. The script provides an entry-level, single-node cluster instantiated as a `systemd` service [12]. As a result of working with a VM, and working with host OS DNS resolution implementation, there maybe some troubleshooting steps necessary to ensure the `coredns` service can appropriately resolve host names from the internet. Steps to remedy the problem are included in the code base documentation.

Helm as a Kubernetes resource package manager is used to define the application templates, known as helm charts, as part of the full deployment. Each discrete deployment has its own resource definitions. These templates are Kubernetes resource objects defined as YAML manifests with a singular YAML template values file used to reconcile the template variables. Pre-built helm charts are also available to download from a chart repository for dependencies that are already defined by product vendors. For example the message queue implementation, ML tracking server, and the CRD performing the EDA control loop are open source helm charts initialized as a dependency to the larger system helm chart.

The project helm chart is defined as *auto-catan*, and the following are its components:

- **External Helm Charts**
  - **Redis**: Message Queue
  - **KEDA**: EDA Auto-Scaler CRD
  - **Minio**: Local S3 Cache
- **Applications**
  - **catan-web**: Web server accepting user training requests with hyper-parameters
  - **mlflow**: Training and validation tracking server
  - **catan-ml**: Dramatiq asynchronous task runner

Note: A separate library is defined as *catan-dramatiq* where the actual business logic is located, including dramatiq actor functions. The task runner initialized by the docker runtime imports the actor functions and executes the logic defined in the consumed queue message.

The KEDA auto-scaler aims to take the *catan-ml* application deployment, and scale the runtime based on the number of messages in the queue. The scaling triggers start at 2 messages in order to stop KEDA from self-consuming the first replica in the deployment, or the infinitely scale pods until the cluster crashes. Each scaled runtime then communicates with the MLFLOW tracking server to published results of model benchmarks. The user can manually collect training artifacts or download the respective artifacts by interacting with the MLFLOW API (*example script provided*)

Results - Model Performance

Each Model-Reward function pair was trained with the goal of changing the hyper-parameters between each reward function as little as possible. The main hyper-parameter that was changed was γ, which we increased with reward functions that were more sparse like the Basic or the Large Magnitude reward function. Each model was trained using a 30-100 train-Benchmark loop. Training for 30 episodes, and then benchmarking the model by running a 100 game test. The peak benchmark results are reported for the Medium, Small, and Small Pooling Architectures in Tables 4, 5, and 6 respectively. The stopping criteria used during this investigation was either letting the model run out to 810 episodes, or stopping early when win rate collapsed or started to fall into the lower 40%. Since Each episode average 450-490 turns, that means each model went through over 360,000 optimization steps.

Reward Functions	γ	Win Rate	Victory Points	Average Loss
Basic	0.9995	0.58	7.65	0.00061
Differential VP	0.999	0.61	7.95	0.00074
Reward Actions	0.95	0.57	8.1	0.03282
Incremental VP	0.99	0.59	7.83	0.00130

Table 4: Performance results of Medium DQN across 4 reward functions. "Large Magnitude" reward function not tested with Medium DQN. Batch size = 32, Replay Memory = 20,000, ε Decay rate = 114 episodes, and learning rate = 0.0001 for all runs.

Looking at Table 4, it appears that the Differential Victory Point, which is a sparse reward function, had the best win rate during training, but the average victory points of the Reward Actions reward function was higher. Similarly, the Small DQN results in Table 5 had good performance with both sparse and frequent re-

ward systems. Although the Differential Vicotry Point option was the best for both the Small and Medium DQN, using an $\alpha = 0.05$, with a sample of 100 runs, we can't say that the improvement in win rate between Differential Victory Point reward function and the Reward Actions reward function is statistically significant.

Looking at the Small DQN with Pooling results in Table 6, a proportion test yields a p-value of 0.02, which indicates there is statistically an improvement in using the Reward Actions reward function compared to the Incremental Victory Point reward function (for Small DQN with Pooling). We also see that the average victory points over the benchmark was the highest under Reward Actions, meaning the RL agent consistently performed better than some of the other reward function options. Seeing the results across all three Models, its clear that model architecture plays a bigger role in performance than the reward function. For all possible reward functions, the Small DQN with Pooling performed in the lower 60%s, while the medium DQN performed worse across all reward functions.

Reward Functions	γ	Win Rate	Victory Points	Average Loss
Basic	0.9995	0.6	7.82	0.0024
Large Magnitude	0.9995	0.59	7.7	0.36234
Differential VP	0.999	0.64	8.18	0.00183
Reward Actions	0.97	0.59	7.71	0.0344
Incremental VP	0.99	0.58	7.92	0.0018

Table 5: Performance results of Small DQN across 5 reward functions. "Large Magnitude" reward function used learning rate of 0.00001, all else used learning rate =0.0001. Batch size = 32, Replay Memory = 10,000, ε Decay rate = 45-56 Episodes for all runs.

Reward Functions	γ	Win Rate	Victory Points	Average Loss
Basic	0.9995	0.61	8.02	0.0022
Large Magnitude	0.9995	0.6	7.88	0.38424
Differential VP	0.999	0.61	8.15	0.00249
Reward Actions	0.95	0.67	8.4	0.06995
Incremental VP	0.99	0.57	7.51	0.00334

Table 6: Performance results of Medium DQN across 5 reward functions. "Large Magnitude" reward function used learning rate of 0.00001, all else used learning rate =0.0001. Batch size = 32, Replay Memory = 10,000, ε Decay rate = 45-56 Episodes for all runs.

Looking at all the test benches conducted across the different runs using the Small DQN with Pooling architecture, the team noticed that each run seemed to perform well as the level of exploration dropped to near 5%. Afterwards, the model would struggle to improve, oscillating around the .55 winrate, which is just marginally better than the random agent. The results can be seen in 8.

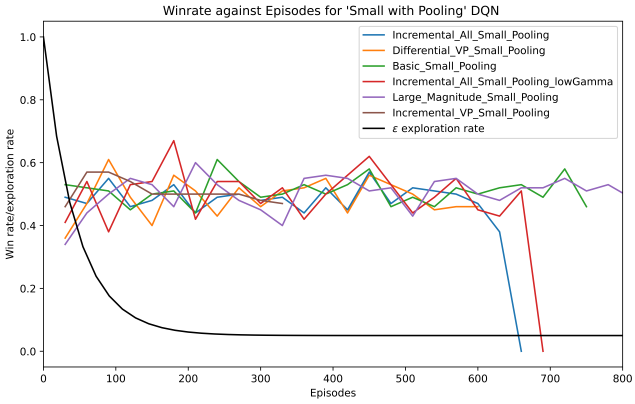


Figure 8: Performance results after every 30 episodes of training for all Small with Pooling DQN runs

The oscillations could be happening for a couple of reasons. Since exploration is fully minimized after around 200 episodes, we could be over-training the model, which is why we see some runs start to collapse down to 0. Another reason is the instability of DQN coupled with the only updating the target every 100 episodes. The optimizer has already taken close to 50,000 optimization steps before the target is updated. we could be at the point of over training well before we update our target. Resulting in a stale target and cause overestimation bias.

We plotted each game's total discounted reward, training rate, and target update schedule to see if there was any relationship between the three. This can be seen in Figure 9. From Figure 9, each decrease in win rate (orange) and large swings in the total discounted reward (blue) are closely coupled to when the target net is being updated. To improve this, we could decrease the update schedule, or we could move to a different architecture like a Dueling or Double DQN to help improve stability, so these oscillations don't occur.

Figure 9: Total Discounted reward per episode, Benchmark win rate, and Target update schedule all plotted against episodes of Small with Pooling DQN and the reward function that Rewards all Actions.

Results - Deployment Performance

To test the auto-scaling behavior, a test configuration was curated. It is reflective of a typical training/validation run, but tweaked to provide quick completion of workflow. 3 clients are initialized over a terminal (through the use of *curl*) and a POST request to the appropriate back-end route was sent, with a JSON object, encoding the test hyper-parameters.

```
NAME                                                READY   STATUS    RESTARTS   AGE
catan-ml-849889cdb6-fvn8n                           1/1     Running   0          81s
catan-ml-849889cdb6-hs4r8                           1/1     Running   0          2m8s
catan-ml-849889cdb6-qnwjt                           1/1     Running   0          2m6s
catan-ml-849889cdb6-vnnl2                           1/1     Running   0          2m6s
catan-web-7569f9cfdf-mn4h2                          1/1     Running   0          3m17s
chart-minio-7b649fbccc-jrr4f                        1/1     Running   0          3m17s
chart-redis-master-0                                1/1     Running   0          3m17s
chart-redis-replicas-0                              1/1     Running   0          3m17s
chart-redis-replicas-1                              1/1     Running   0          2m47s
chart-redis-replicas-2                              1/1     Running   0          2m26s
keda-admission-webhooks-56f95fc494-mg7rl            1/1     Running   0          3m17s
keda-operator-7dd47fffcd-lmftk                      1/1     Running   0          3m17s
keda-operator-metrics-apiserver-76c54b4f46-f9qgq    1/1     Running   0          3m17s
mlflow-595998694c-nhtld                             1/1     Running   0          3m17s
```

Figure 10: *kubectl* Showing Successful Pod Auto-Scaling

Upon inspection of cluster pod utilization, we can see the KEDA CRD Operator was able to successfully spin up the appropriate number of worker pods. Three client request results in 3 working task runners with a fourth runner awaiting the message queue on stand-by. The following are the runtime performance of the 3 task runners.

Parallel Run Results:

```
Runner #1

{
  "Server:": "Device: cpu, URI: http://
    mlflow:8250, TRAIN_SUCCESS: True,
    ELAPSED_TIME: 204.72695016860962"
}

Runner #2

{
  "Server:": "Device: cpu, URI: http://
    mlflow:8250, TRAIN_SUCCESS: True,
    ELAPSED_TIME: 229.49895858764648"
}

Runner #3

{
  "Server:": "Device: cpu, URI: http://
    mlflow:8250, TRAIN_SUCCESS: True,
    ELAPSED_TIME: 237.5023548603058"
}
```

Each pod as part of the scaling deployment was allotted 2vCPU. Cluster CPU utilization was hovering around 87% at full load with 3 active runners performing the training/validation loop. As the cluster CPU utilization saw an increase, so did completion time. Performance priority was given to the first task runner launched. Further pod scaling need was not possible due to CPU resource constraints. Applying 4 concurrent task runners in addition to a 5th runner for stand-by caused the control plane to taint the cluster and significantly increased cluster instability.

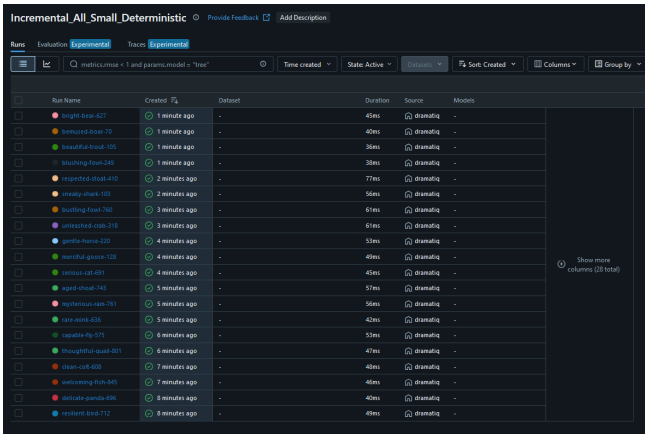


Figure 11: MLFLOW Output of Parallel Runs

The tracking server was able to handle concurrent task runners publishing data to itself. Inspecting runtime logs show it is able to handle up-to 4 concurrent connections before having to synchronously throttle accepting connections. Completion of the deployment test yield 20 total epoch completion in under 5 minutes total run-

time. A secondary test was performed under the same hardware envelope with the KEDA EDA auto-scaler turned off, yielding total runtime of 20 epochs to be above 12 minutes. The performance value proposition scales as the number of concurrent requests increase. Models with large hyper-parameter space would benefit heavily from such a deployment.

Future Improvements

After reviewing the results from the experiments, some ideas were discussed to improve the performance of the trained agents, the training environment and training infrastructure:

Deployment Infrastructure:

While developing the MVP for the deployed version of the training workflow, several aspects of the design was identified as key iteration section to provide a more valuable experience to scientists. Architecture improvements are split into the following categories:

- Training and Validation Time
- User Experience
- Deployment Initialization Time
- System Reliability and Availability

GPU pass-through is an important aspect of the current deployment strategy that was not implemented due to the lack of hardware availability and development time. As of the time of this writing, current implementation of PyTorch use NVidia CUDA and AMD ROCm as front-ends to OS level APIs to access GPU resources. The hardware the system was developed on did not support either of the SDKs so consideration was not provided to allocate development time for GPU pass-through [13]. Acquiring a GPGPU capable of using the compute SDK would allow for the use of GPU scheduling object. Allocations of GPUs are defined under resource affinity definitions, which is outlined on a per deployment object basis.

In addition to performance improvements, the cluster health can be improved by defining a dead-letter-queue(DLQ) for handling task runner exceptions. For example, if a task runner is unable to be scaled due to KEDA operator errors, Redis will redirect the message to a DLQ for later consumption. Currently Redis will wait the full task time-out period for a return status from a consumer with the message ID, which is not desirable behavior.

Container build and deployment times are also fairly long as the container size is exceptionally large as a result of including the entire PyTorch toolkit (including CUDNN APIs). Any container images with a dependency on $catan - dramatiq$ will see image sizes larger than 5GB. There are distroless [14] build techniques that can be harnessed to reduce image size by 20-30%, along with optimized layering techniques of containers. In addition, the model can be served over the tracking server or an S3 object store to further reduce the container size.

Finally, client side interaction with the web-server has limited interaction and tolerance to unexpected behavior. At the time of this writing, the application handling requests operate on a synchronous execution pattern, causing the client to be blocked from further execution before the task runner sends back a status code (which in theory can take hours). Asynchronous handling of client connections in addition to alternative forms of status updates (such as closing client connection at job run time and sending updates over tracking server) would allow for a more streamlined user experience.

Agent Model:

on the Policy network side, interesting developments that we considered but could not implement due to time constraints are the redesign of the DQN into a Double DQN or even an implementation of rainbow DQN. The double DQN consist in a separate network for estimating the value of the actions and for creating the policy for a given state. Both networks can then be updated symmetrically, creating a more stable learning that does not fail due to the tendency of single DQNs to overestimate[15]. Further, a Rainbow DQN would combine a double DQN with other improvements like Dueling DQNs.

We also consider the implementation of a convex combination of Target and Agent DQNs as a possible solution to improve training stability and reduce overestimation.

Another area that where we have ideas for improvement is in the training methodology. From the original author work, an improvement we made was to fix the bad implementation of the Adam optimizer. Original implementation forced the learning rate to be the

calculated decaying learning rate for all outputs when running the optimizer algorithm. However, the Adam optimizer by design is meant to automatically adjust this based on the gradient moments of each parameter. Forcing the learning rate to be a fixed value for all parameter effectively converts Adam back into standard stochastic gradient descent. Future work would be to integrate a scheduler and the optimizer with a strategy that allows learning rates to more finely tuned, increasing the learning rate when stuck in suboptimal minimums while decreasing it when risk of exploding gradients is present.

Finally, another idea to combat overestimation by changing the training methodology is to change the algorithm to use approximate policy iteration instead of TD. by unrolling entire trajectories in the game instead of updating on each action taken, the overestimation caused by the DQN does not propagate via the target network estimates of the next state action pair, but instead a true estimate calculated with sample trajectories is used, giving more reliable data to the DQN to use for training. This is specially effective because the game simulation accurately replicates the real world behavior, and itself requires minimal computational effort, so collecting data to train the DQN with a Monte-Carlo methodology would not represent a challenge.

References

[1] H. Cuayáhuitl, S. Keizer, and O. Lemon, "Strategic Dialogue Management via Deep Reinforcement Learning," NIPS'15 Workshop on Deep Reinforcement Learning, 2015. [Online]. Available: https://arxiv.org/pdf/1511.08099.

[2] Q. Gendre and T. Kaneko, "Playing Catan with Cross-dimensional Neural Network," arXiv, 2020. [Online]. Available: https://arxiv.org/pdf/2008.07079. [Accessed 2024].

[3] G. van der Kooij, "Actor-Critic Catan: Reinforcement Learning in High Strategic Environments," Utrecht University, 2020. [Online]. Available: https://studenttheses.uu.nl/bitstream/handle/20.500.12932/40638/Reinforcement%20Learning%20Catan%20Thesis.pdf?sequence=1. [Accessed 2024].

[4] T. P. Lilicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," arXiv, 2019. [Online]. Available: https://arxiv.org/pdf/1509.02971. [Accessed 2024].

[5] C. C. Kim and A. Y. Li, "Re-L Catan: Evaluation of Deep Reinforcement Learning for Resource Management under Competitive and Uncertain Environments," Stanford: CS230 Final Project, 2021. [Online]. Available: https://cs230.stanford.edu/projects_fall_2021/reports/103176936.pdf. [Accessed 2024].

[6] P. McAughan, A. Krishnakumar, J. Hahn, S. Kulkarni, "QSettlers: Deep Reinforcement Learning for Settlers of Catan," Georgia Institute of Technology, 2023. [Online]. Available: https://akrishna77.github.io/QSettlers/. [Accessed 2024]

[7] H. Charlesworth, "Learning to Play Settlers of Catan with Deep Reinforcement Learning," Rowden Technologies, 2022. [Online]. Available: https://settlers-rl.github.io/#none. [Accessed 2024].

[8] GJSProgo, "Reinforcement Learning for Catan: DQN/A3C Implementation," 2024. [Online]. Available: https://github.com/GSJProgo/RL-Catan. [Accessed 2024].

[9] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang, "A Theoretical Analysis of Deep Q-Learning," 2020. [Online]. https://arxiv.org/abs/1901.00137. [Accessed 2024]

[10] Kubernetes, "Extend the Kubernetes API with CustomResourceDefinitions," Kubernetes, 2024. [Online]. Available: https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/. [Accessed 2024].

[11] Kubernetes, “Kubernetes,” Kubernetes, 2024. [Online]. Available: https://kubernetes.io/. [Accessed 2024].

[12] K3s, ”Quick-Start Guide”, 2024. [Online]. Available: https://docs.k3s.io/quick-start. [Accessed 2024]

[13] Pytorch Documentation, The Linux Foundation, 2024. [Online]. Available: https://pytorch.org/docs/stable/index.html. [Accessed 2024].

[14] Google Inc, ”Distroless” Container Images, 2024. [Online]. Available: https://github.com/GoogleContainerTools/distroless. [Accessed 2024].

[15] Hado van Hasselt, Arthur Guez, David Silver “Deep Reinforcement Learning with Double Q-learning,” 2020. [Online]. https://arxiv.org/pdf/1509.06461. [Accessed 2024]

Parallel Exploration of Reward Functions in Catan using Containerized Deployments

Wasif Ul Islam, John Michael Van Treeck, Diego Reyes Olmos

PURDUE UNIVERSITY | Elmore Family School of Electrical and Computer Engineering

Problem: Can We Create a Winning Catan Agent?

- Catan is a very complex game
- Win by being the first to 10 Victory points (VP)
- Dice rolls, randomness to board setup, piece placement, and partial observability all impact the MDP
- Highly non-deterministic
- Using our encoding structure:
  - the state spaces has 5,354 different elements
  - Action space has 960 different possible actions
- Using A Deep Q network, we wanted to investigate the effectiveness of different reward Structures
- Other researchers have done Actor-Critic Models, focused on just player trading, or developed combinations of Heuristics and AI.

Environment Overview and Adaptations

- Simplifying the Game:
  - Allowing the RL Agent to be aware of opponent Resources
  - Only 2 players, not 3 or 4
  - Removing Development Card longest Road
  - Playing against an agent that makes random moves
  - Remove Player-Player Trading
  - Selecting Resources to discard removed
- State space has both board states and player states
- Action space has actions that interact with the board and actions that interact with players

Board States	Player States
Resource Tiles	Number of Victory Points
Resources Probabilities	Resources of each players
Trade Harbor Locations	Development cards played
City locations	Development cards available
Road Locations	Opponent number of Development Cards
Robber locations	Army Size of each player
	Longest Road of each player

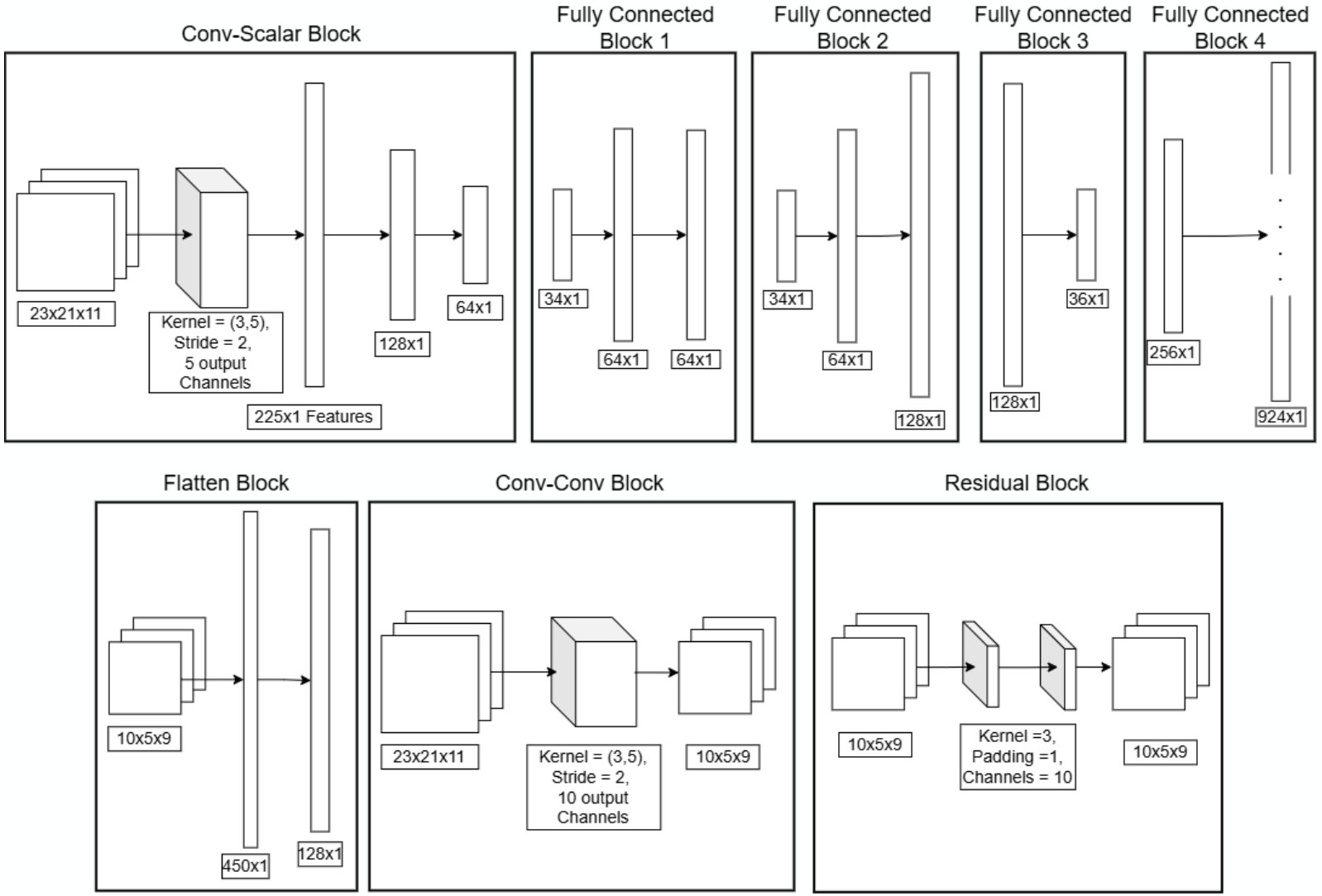
Overview of Training and DQN

- Bellman Optimality: $Q^*(s,a) = E\left[r + \gamma \max_{a'} Q^*(s',a') \,\middle|\, s,a\right]$
- Target: $y = r + \gamma \max_{a'} Q_{\theta^-}(s',a')$
- Loss: $L(\theta) = E_{(s,a,r,s'\sim D)}\left[\left(y - Q_\theta(s,a)\right)^2\right]$

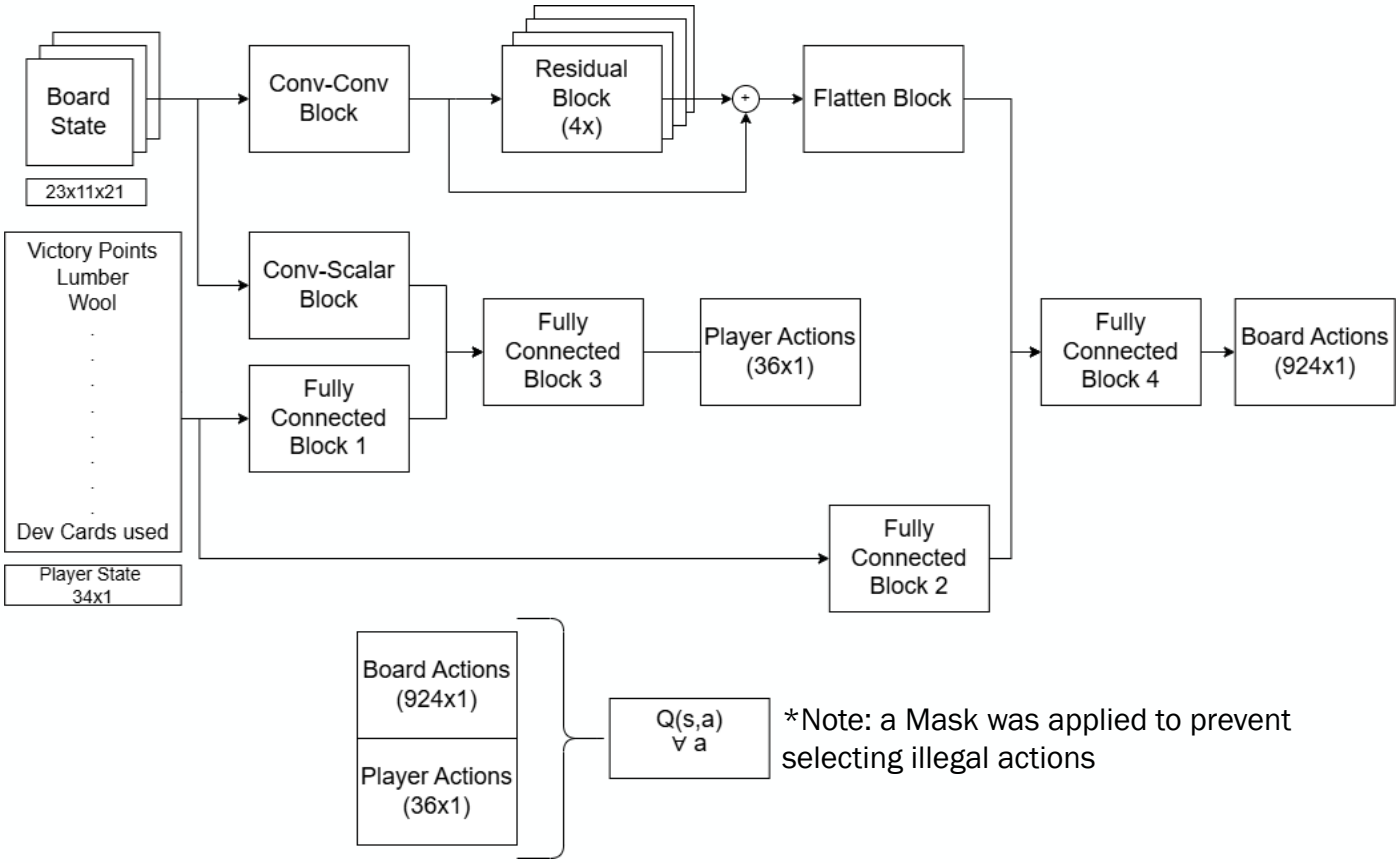
Steps for selecting action (Pseudo Code):

1. Calculate ϵ (decreases from 1 to 0.05)
2. If $\text{random}(0,1) > \epsilon$, Step 3, else: select random action
3. Calculate $Q_\theta(s,a)\ \forall a \in A$
4. Remove Illegal actions.
   - Deterministic Variant: $\pi(a|s) = \text{argmax}\left(Q_\theta(s,a)\right)$
   - Stochastic Variant: $\pi(a|s) = \frac{exp(Q(s,a)}{\sum_{a' \in A} exp\left(Q(s,a')\right)}$
5. Run Optimizer
6. If $Episode\ \%\ 100 == 0$: $Q_{\theta^-} \leftarrow Q_\theta$

Building Blocks of Architecture



Overview of the Archictecture



*Note: a Mask was applied to prevent selecting illegal actions

Reward Structures Proposed:

Baseline	Large Magnitude	Differential VP	Incremental VP	Rewards for Actions
+ 1 for Win - 1 for loss	+ 10 for Win - 10 for loss	+ (0.75 +ΔVP) for Win - (0.75 +ΔVP) for loss	+ 1 for Win - 1 for loss +0.1(ΔVP) at each update	+0.001 for Resources +0.01 for Dev Cards +0.01 for Roads +0.01 for Knights +0.1 for Settlements +0.2 for Cities +5 for win -5 for Loss
Reasoning: • Very easy baseline	**Reasoning:** • Increase responsiveness of Rewards • Amplify difference in Temporal Difference Error • Balance sparsity with magnitude	**Reasoning:** • Maximize reward by having more victory points than opponent • Losing by a little is better than losing by a lot • Encourages RL agent to get more Victory points than opponent	**Reasoning:** • Maximize reward by having more victory points than opponent • Create a rubberband effect to try and catch up when losing • Reduce sparsity in reward function	**Reasoning:** • Can we provide rewards all the time of various scale to encourage always wanting to take an action • Does reducing sparsity a lot help?

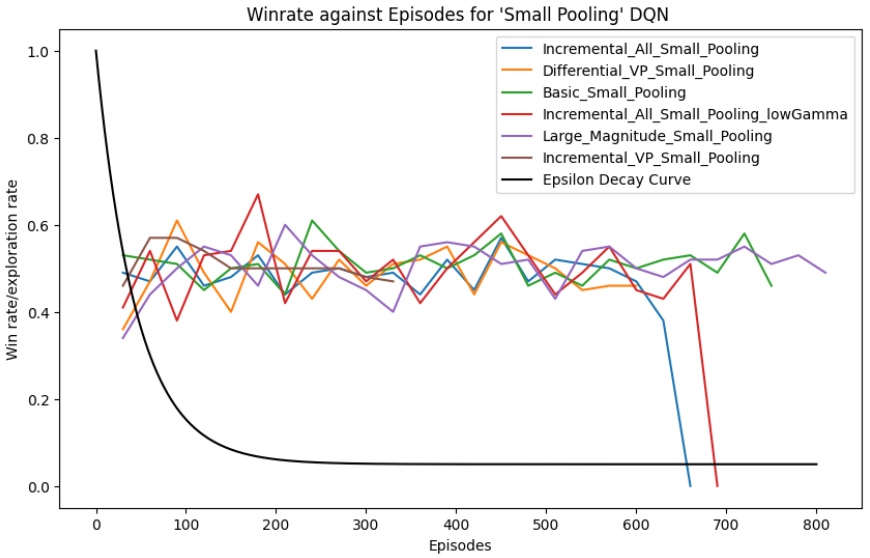
Results

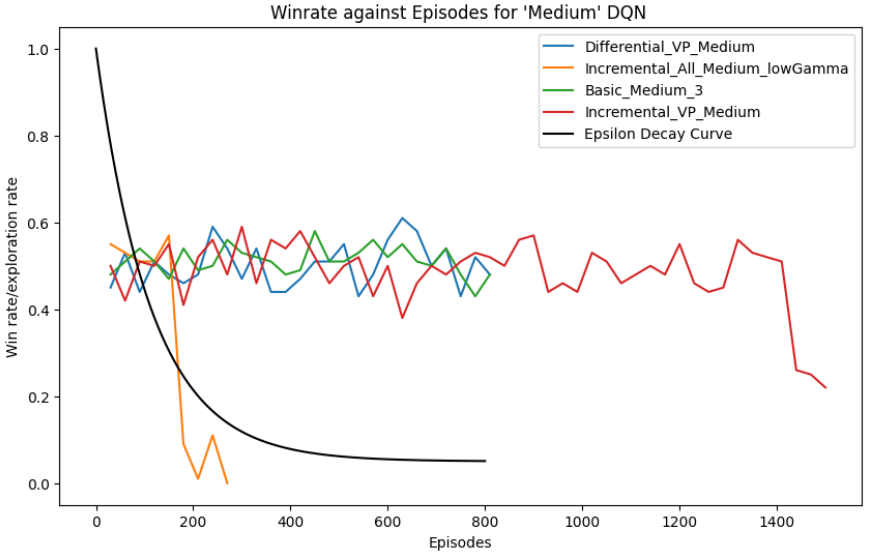
Model Selected	Batch Size	Memory size	Reward Structure	Epsilon Decay rate (episodes)	Learning Rate	Gamma	Peak winrate	Episodes trained at peak winrate	Average VP	Average model loss	Average game length
Medium	32	20000	Basic	114	0.0001	0.9995	0.58	450	7.65	0.00061	428
			Differential VP	114	0.0001	0.999	0.61	630	7.95	0.00074	470
			Reward Actions	114	0.0001	0.95	0.57	150	8.1	0.03282	490
			Incremental VP	114	0.0001	0.99	0.59	300	7.83	0.00130	472
Small	32	10000	Basic	56	0.0001	0.9995	0.6	180	7.82	0.00240	447
			Differential VP	45	0.0001	0.999	0.64	210	8.18	0.00183	387
			Reward Actions	136	0.0001	0.97	0.59	240	7.71	0.03438	552
			Incremental VP	45	0.0001	0.99	0.58	810	7.92	0.00180	518
			Large Magnitude	56	1.00E-05	0.9995	0.59	90	7.7	0.36234	499
Small Pooling	32	10000	Basic	56	0.0001	0.9995	0.61	240	8.02	0.00220	498
			Differential VP	45	0.0001	0.999	0.61	90	8.15	0.00249	429
			Reward Actions	45	0.0001	0.999	0.57	450	7.8	0.05589	534
			Reward Actions	45	0.0001	0.95	0.67	180	8.4	0.06995	489
			Incremental VP	45	0.0001	0.99	0.57	90	7.51	0.00334	517
			Large Magnitude	56	1.00E-05	0.9995	0.6	210	7.88	0.38424	468

Results

Takeaway:

- Models seemed to peak out in performance right as exploration reached its minimum.
- Any continued training, and models began to oscillate, become unstable, or overfit the data.
- The Architecture had a significantly higher influence than reward structure on the results
- There was some inherent instability by only updating the DQN target every 100 episodes.

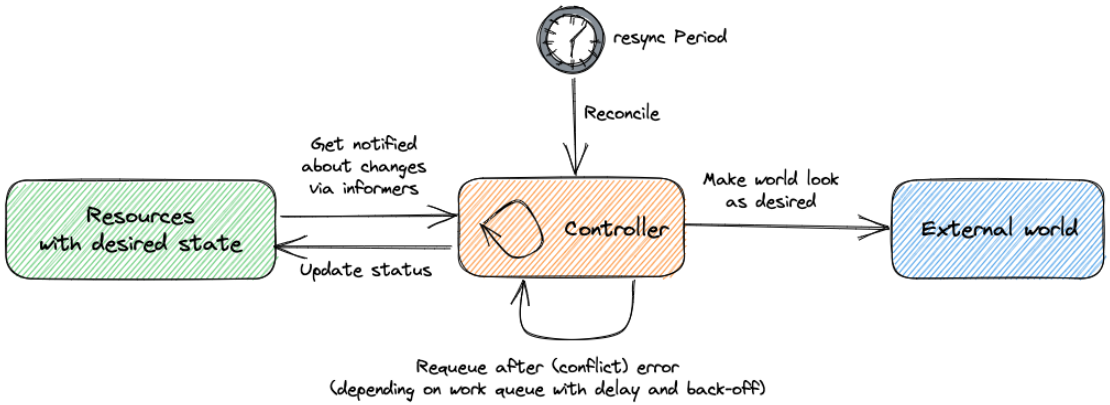




Workflow Scaling Motivation

Scaling Motivation:

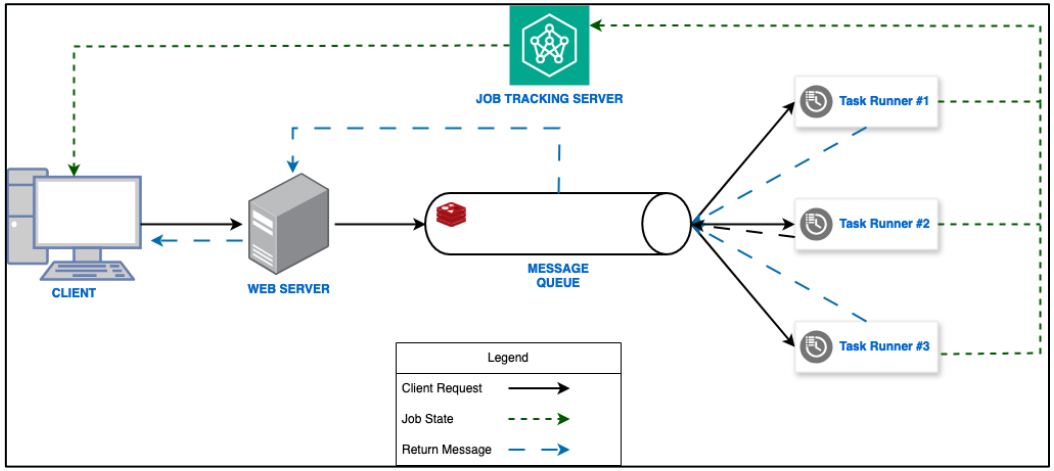
- Model training intrinsically a serial process.
- Multiple runs of varying hyper-parameters may take a long time.
- Implementation aims to be a technical demonstration of parallel training/validation workflow using Kubernetes.
- Objective is to reduce total computation time.



System Architecture

EDA System:

- Simple architecture consisting of a web server accepting client requests in the form of required hyper-parameters as a JSON object
- Web server publishes message to a message queue
- Task runner consume the message and execute the training loop with given hyperparameters



Workflow Scaling Implementation

Scaling Setup:

- Helm Charts:
  - Use vendor definitions for supporting systems
    - Redis
    - KEDA – CRD
  - Custom Definition for RL Workflow
    - Web-Server
    - Tracking Server
    - Distributed Task Runner

- KEDA allows for the distributed task runner to scale with number of messages in message queue

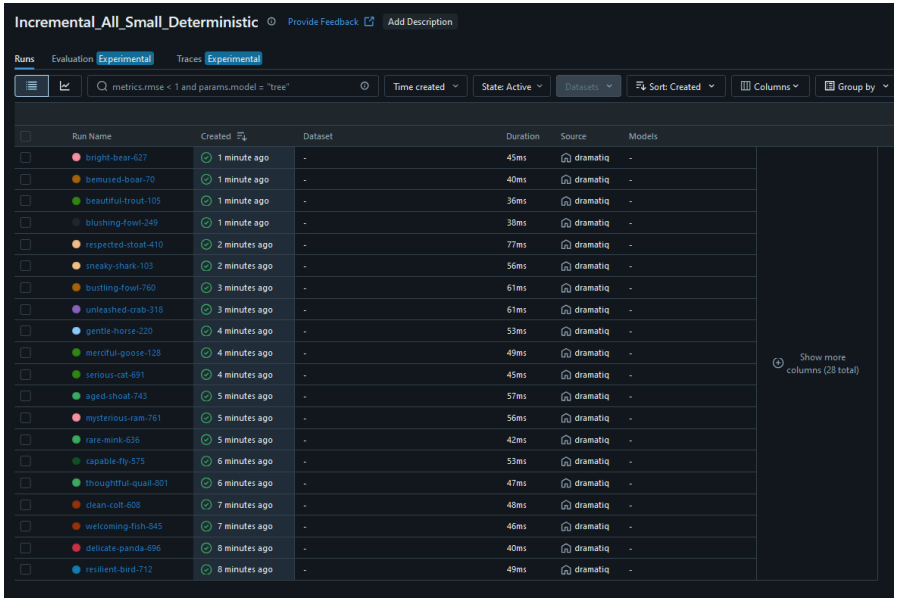
Results - Deployment

Scaling Tests:

- Performance on par with expectation:
  - **~64% reduction** in total training time vs sequential training.
  - Average completion time for 3 concurrent runs: **223.906 seconds**
  - Sequential Completion Time: **615.23 seconds**

- Value proposition for parallel training/validation execution scale as number of pods scaling increase
- Hardware is main limiting factor

```
Runner #1

{
  "Server:": "Device: cpu, URI: http://
    mlflow:8250, TRAIN_SUCCESS: True,
    ELAPSED_TIME: 204.72695016860962"
}

Runner #2

{
  "Server:": "Device: cpu, URI: http://
    mlflow:8250, TRAIN_SUCCESS: True,
    ELAPSED_TIME: 229.49895858764648"
}

Runner #3

{
  "Server:": "Device: cpu, URI: http://
    mlflow:8250, TRAIN_SUCCESS: True,
    ELAPSED_TIME: 237.5023548603058"
}
```



Future work

Kubernetes Training infrastructure improvements	Policy Network Improvements	Training Methodology Improvements
• Add GPU pass through for Kubernetes deployment. • asynchronous handling of client requests. • container build optimizations. • message queue disruption fault tolerance. • Improve model storage.	• DQN change to Double DQN or Rainbow DQN to further improve stability. • Convex combination of Target and Agent DQNs to make updates more smooth. • Add Actor-Critic implementation.	• Proper scheduler implementation to improve Adam and learning rates. • Create a Monte-Carlo Version of the training algorithm using approximate policy iteration instead of TD.

All Source Code Published Here:
https://github.com/OnlyUseMeRocket/auto-catan

Thank You

PURDUE UNIVERSITY

Elmore Family School of Electrical and Computer Engineering

auto-catan

Academic technical demonstration of agentic RL implementation of game AI playing `Catan`. It is aimed to showcase a prototype pipeline starting from data acquisition, model training, and model inference

> NOTE: Codebase available at: https://github.com/OnlyUseMeRocket/auto-catan

Table of Content

Installation

> *NOTE: Currently, the installation of the cluster using defined IAC only supports Linux (Tested on Ubuntu 24.04 Server). YES YOU WILL NEED ROOT ACCESS TO YOUR MACHINE*

Pre-requisites

For effective reproduction, please setup a `linux/amd64` VM with `NAT` networking and with `Ubuntu 24.04` Server. Perform a minimal install with SSH server (no additional packages selected from snap) and install the following packages:

```
apt-get install git build-essential curl ca-certificates
```

Following that, clone the `main` branch onto the home directory of your VM

Install and Start Cluster

1. Create Local Executable Directory

```
mkdir -p $HOME/.local/bin
```

2. Add the above directory to path

```

```

```
echo "export PATH=$PATH:$HOME/.local/bin" >> $HOME/.bashrc
source $HOME/.bashrc
```

3. Run the scripts in `iac` to bootstrap cluster

```
./1-install-kubectl.sh
./2-install-k3s.sh
...
...
```

4. Refresh your terminal to realize system changes

```
source $HOME/.bashrc
```

5. Check status of Cluster: *Cluster status should be active*

```
sudo systemctl status k3s
```

The more comprehensive way of checking if the cluster is healthy is to check deployment and pod status under the `kube-system` namespace. For K3s, `coredns`, `local-path-provisioner`, `metrics-server`, and `traefik` should be all running and ready.

```
kubectl get all -A
```

We want to see the **non-installer** pods to be `Running` and `Ready`. The **installer** pods should not be running, and should be shown as `Completed`.

6. Create the `catan` namespace for the deployment, and set it as the default namespace

```
kubectl create namespace catan
kubectl kubectl config set-context --current --namespace=catan
```

Install Application Helm Chart

1. Once the cluster is running and cluster health can be verified, we can start deploying our application. Start by changing directories to root of the project and executing the following commands

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo add kedacore https://kedacore.github.io/charts
```

```
helm dependency build ./deploy
```

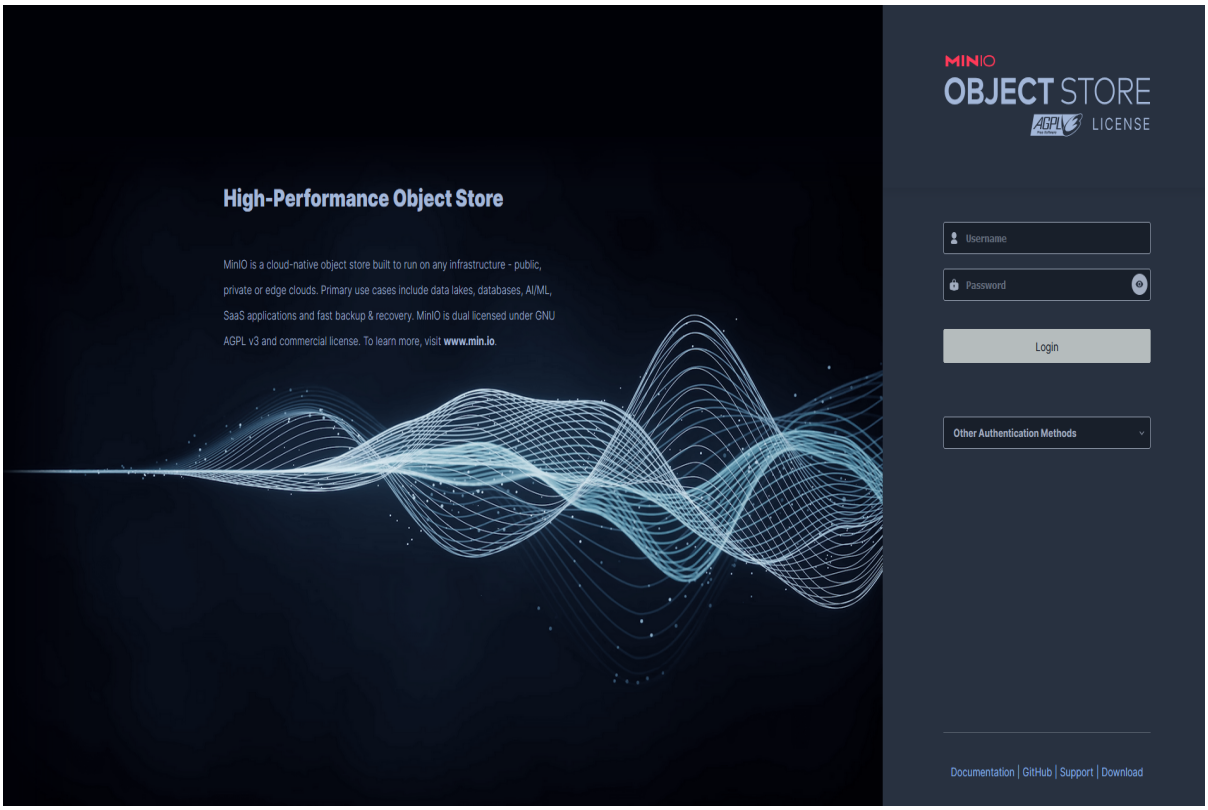
This will install our custom helm chart that contain RL specific applications and its dependencies.

2. Once the helm dependencies are installed you can bring up the deployment using `tilt`

```
tilt up --host 0.0.0.0
```

3. You can check if the deployment is properly working by looking at the `Tilt` status and navigating to `Minio`

> *Tilt showing all deployments are operational*



> *Splash Screen for Minio*

Running Parallel Trainings (*NO UI*)

Unfortunately, (*as of the current release*), we do not have asynchronous handling of client requests to the web server or a UI. As a result, the requests can be sent to the backend using `curl` or the swagger UI at `<ip-address>:8251/docs`. The `config.md` document contain valid hyper-parameters to submit to the backend.

> *Example `curl` command*:

```
curl -X POST <vm-ip-address>:8251/inf/run_experiment \
-H "Content-Type: application/json" \
-d <json-object-from-config.md>
```

If using the Swagger UI, then multiple tabs will need to be opened in order to submit multiple jobs, since asynchronous handling of requests, or a client side application capable of handling asynchronous requests are not implemented yet.

Alternative Execution

If you do not feel like going through all of the steps and troubleshooting and just want to run the model training experiment, go to the `rl_catan` folder, and just run `main.py`. If you would like to change the hyper-parameters, change the constant definitions under `Configurations.py`. In order to run, execute the following commands:

1. Download the `uv` package manager:

> For Linux and MacOS

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

> For Windows

```
powershell -ExecutionPolicy ByPass -c "irm
https://astral.sh/uv/install.ps1 | iex"
```

2. Execute `main.py`

```
cd rl-catan
uv sync
source .venv/bin/activate
python3 main.py
```

Installation Troubleshooting

1. CoreDNS `CrashBackOffLoop`

If you are running into pods in `coredns` experiencing `CrashBackOffLoop` due to the control plane not being able to `curl` the health probe, it is most likely because you do not have a DNS server configured with your host machine in `/etc/resolv.conf`

Execute the following command:

```
$ kubectl edit configmap coredns -n kube-system
```

Remove the following field and its assocaited predicates (if there are any)

```
 forward . /etc/resolv.conf
```

Add the following in its place

```
forward . 8.8.8.8 8.8.4.4
```

After the edit is complete, restart the `coredns` deployment

```

```

```
$ kubectl rollout restart deployment.apps/coredns -n kube-system
$ kubectl rollout restart deployment.apps/traefik -n kube-system
$ kubectl rollout restart deployment.apps/metrics-server -n kube-system
```

2. Docker Failing to Build Images (`deb.debian.org` not found):

If you are developing `Dockerfiles` that do not have an image already built, and you are in an Ubuntu VM, and chose to install `docker` from the OS installation menu, there is a good chance `docker` was installed using `snap`.

If `docker` is having a hard time resolving QDN's you will have to manually add google's DNS server to the docker config in the following location

```
/var/snap/docker/current/config
```

Edit the file `daemon.json` (note: you will have to be sudo to run this) and add the following lines:

```
{
    "dns": ["8.8.8.8", "8.8.4.4"]
}
```

This should now allow docker to properly build docker files when before it might have been having issues resolving hosts.

Dev Dependencies

In order to assist in reducing reproducability issues, a list of the following dependencies used to develop the project is provided:

- Ubuntu Server 24.04 *(Kernel: 6.8.0-45-generic)*
- Docker *(Version 24.0.5, Build ced0996)*
- Helm
- Bash *(Used in IaC scripting)*
- Tilt

Some Comments Regarding the Repo

- If you use this, and you are experiencing issues, you are largerly on your own. If you email me I can possibly try to help out (islamwasif3@gmail.com)
- Yes, the steps to get it up is bad/redundant/buggy. I apologize as the project was made in a pinch with some classmates

Authors:

1. Wasif Islam

    - islam25@purdue.edu
    - w.islam@obscuritylabs.com
    - islamwasif3@gmail.com

2. John Michael Van Treeck

    - jvantree@purdue.edu

3. Diege Reyes Olmos

    - dreyesol@purdue.edu