# Implementation – Alastair R.D.

## Project Summary

The overall goal of the project is to assess the impact that training on procedurally generated content has on various reinforcement learning models, and how to improve their performance when generalising to unseen content.

The project implementation covers a system for procedurally generating 15x15 ‘grid-world’ levels, a custom gymnasium environment for the models to interact with, and a pipeline to easily build, train and analyse the performance of models.

## Core Features

### PCG

The procedural content generation pipeline was implemented first, making sure there were consistent training environments and allow reproducible testing for hyperparameter tuning and model evaluation. It uses a node-based network structure, supporting future graph-based algorithms and helping map design.

Levels are generated by first creating a lattice-like base map, with every second row and node designated as paths, and the rest as walls. A modified Kruskal’s algorithm is used to generate a perfect maze as a foundation, with further algorithms introducing cycles to create imperfect mazes. Parameters control how cycles are added, adjusting space size and wall removal bias to shape the environment.

A black and white background with white hashtags

AI-generated content may be incorrect.A black and white background with white hashtags

Description automatically generated

***Figure 1 –*** *Pre Kruskal level*  ***Figure 2 –*** *Post-Kruskal’s, ‘perfect’ maze, seed=79*

The system also scales hazard placement, enemy spawning, and objective distribution (keys and exits). Map generation is fully customisable by seed, ensuring reproducibility. Training primarily took place on a single seed (seed 1), targeting what I believed to be a roughly medium difficulty environment.

Using the MapOptions class built for defining these parameters, one sets the values for creating a map that essentially pertain to how open or closed a map is, and how hazardous. This can be roughly analogised to difficulty.



A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 3 – MapOptions settings used for most of training / map creation snippet.*

### Environment

#### Creation

To begin reinforcement learning training, a custom environment was built using the Farama Foundation’s Gymnasium library.

At its core, the environment defines an observation space, a step function to process agent actions and return rewards, and a reset function to ensure clean episodes without information leakage. These functions form the foundation of the training pipeline.  
A basic rendering system using Pygame was also implemented to allow human monitoring of training, helping to identify behavioural issues.

The environment constructor accepts a range of arguments, allowing for different observation spaces, reward settings, and rendering options. It also stores references to the current map and relevant objects for use during training.

*A screenshot of a computer program

AI-generated content may be incorrect.*

*Figure 4 – Example code from the environment constructor.*

#### Observation Spaces and Reward Settings

The observation spaces evolved over time, allowing many configurations: coordinates for key locations, one-hot or scalar encoded map states, agent health values, or A\* path lengths.  
Different observation spaces led to varied model performance, providing key insights into feature selection for reinforcement learning.

A custom reward settings object defines rewards for events like reaching the exit, hitting hazards, moving closer to the goal, or getting stuck. This allowed experimenting with both sparse and dense reward setups, or testing the effects of harsher punishments versus more lenient settings.

*A screen shot of a computer code

AI-generated content may be incorrect.*

*Figure 5 – Reward Settings class.*

*A screen shot of a computer program

AI-generated content may be incorrect.*

*Figure 6 – Example early observation spaces.*

#### Step / Reset

The step function processes an action, updates the environment, returns the new observation, reward, and episode termination flags. This is where the set custom reward settings object will play a part in how the step returns a reward signal.

The reset function fully resets the environment state to start a new training episode cleanly, avoiding contamination from prior episodes. This includes agent location, health, lists tracking visited locations or intentions, and even resets map-based variables in-case training is using procedurally generated content and needs to update to a new map.

*A computer screen shot of code

AI-generated content may be incorrect.*

*Figure 7 – Code snippet from step function.*

#### PCG Capability

A MapOptions object defines parameters for map generation, including hazards, enemies, keys, and exits. On environment initialisation, a map is generated using these options.  
Maps can be swapped during training using a list of seeds, allowing models to train across varied environments within a single environment instance, allowing study of how procedural content generation can effect a model’s ability to generalise.

This list of seeds is passed to the training loop during initialisation and training – which then call’s the environment’s helper functions. These functions allow for the generation and subsequent setting of maps for use during training within the environment.

This capability allows for curriculum learning capacity too – by specifying a different set of MapOptions, more open, ‘easier’ levels can be trained on initially. Changing the hyper parameters to create more dangerous and wall-filled maps allows for more difficult training regimes for models performing well on earlier sets.

*A screen shot of a computer code

AI-generated content may be incorrect.*

*Figure 8 – Environment’s map setting function.*

#### Rendering

Rendering is not required for training but was implemented for better insight into why a model may be struggling during training, and what behavioural trends it might be exhibiting.  
The environment can render not only live episodes but also replay full training histories, allowing for unrendered training for efficiency, and playback at a customisable FPS if desired.  
Additionally, a heatmap feature was added to summarise agent behaviour over full training runs quickly, highlighting movement patterns and problem areas. The heatmap allows for a rudimentary understanding of whether a model has learnt good or bad behaviours from a glance.

*A computer screen with text on it

AI-generated content may be incorrect.* **A screenshot of a game

AI-generated content may be incorrect.**

*Figure 9 – Heatmap rendering snippet and example heatmap.*

### Testing Pipeline

#### Overview

A custom testing and logging pipeline was developed to support efficient experimentation across environment configurations, PCG setups, model architectures, and training parameters. This system enables quick and reproducible testing and easy hyperparameter tuning.

The process begins by defining the environment with specific reward shaping and map options.

Once this is in place, the next step is to define a **TrainingConfig()** and **ModelConfig()** object.  
The model configuration class specifies architecture details—layer count, neurons, inputs and outputs and descriptive metadata.   
The training configuration class defines how the training loop operates, covering iteration count, episodes per iteration, decay policies, learning rates, and model type, along with much more. The training step of the pipeline takes lists of these configurations, building and training models one by one based on the settings.

A screen shot of a computer code

AI-generated content may be incorrect.

*Figure 10 – Model and Training configuration classes.*

#### Training & History

A ***build\_and\_train\_multiple\_models()*** function handles the full pipeline execution. It takes model and training configurations along with an environment reference, builds each model accordingly, and runs training based on the provided settings. This allows differentiation between Policy Gradient, DQN variants and A2C training loops.

*A screen shot of a computer code

AI-generated content may be incorrect.*

*Figure 11 – Function for training and building multiple models.*

During training, a history object is built that logs not only reward progression but also agent positions per episode—enabling later use of rendering and heatmap features in the environment. There are two classes here, an **EpHistory()** class which stores specific history for a single episode, and a **TrainingHistory()** class which stores all histories over the course of training, featuring a list of lists of EpHistories.

Once training concludes, the **plot\_reward\_history()** function can be used to graph training rewards across iterations or average rewards per episode. This became a primary tool for evaluating the effect of changes to model structure, training parameters, reward shaping, and observation space.

There are also several post training analysis functions, for building a pandas dataframe, and subsequently creating scatterplots, bar charts, and correlation heatmaps for examining results of trained models upon unseen seeds.

### Models

Each of the types of model’s training processes follow roughly the same logic.

The process starts with setting an optimiser and initialising global variables necessary such as for storing history.  
If training on multiple levels, the correct map is generated and set in the environment.

The training will loop through the set number of iterations as per the training configuration, scaling the exploration policy if applicable based on how far into the loop it is and relevant settings. For Policy Gradient and Actor Critic methods, each iteration comes with 10 episodes.  
Following a **play\_multiple\_episodes()** function, the environment will reset for each episode, and the observation space will be processed into numerical inputs. From there each step plays out with a **play\_one\_step()** function, which in turn calls the environment’s **step()** function with the chosen action. This continues until an episode is terminated and truncated. Loss is calculated, rewards and gradients are gathered each step, and at the end of the episode an **EpHistory** object containing location and reward history is stored in a list to be returned at the end of the iteration.

With the end of the iteration, the relevant history is stored and gradients are applied by the optimiser. The cycle continues with further iterations.  
DQN training structure is slightly different, relying on singular episodes instead of batched in iterations. Training may also include multiple models in both DQN and A2C methods, and could rely on a replay buffer or staggered updating of weights for the secondary model.

In the case of A2C, state values from the critic are also returned with each iteration, for calculation of loss and gradients for the actor.

A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

*Figure 12 & 13 – Policy Gradient – play\_one\_step() / play\_multiple\_episodes()*

When all is said and done, a **TrainingHistory** object contains all the history stored from each iteration and their episodes, ready for plotting and analysis.

## Results

#### Training - Policy Gradient

Testing began with a simple Policy Gradient model set up, using relatively small model architectures and relatively standard hyperparameters, aiming to sweep through ranges and identify values that worked and areas of failing.  
Very early it was evident that even though this task was more complex than I was used to in cartpole, models were still prone to overfitting and overreacting to rewards.

A graph of a graph

AI-generated content may be incorrect.

*Figure 14 – Early policy gradient testing, Boltzmann exploration and learning rate tests.*

This early chart shows 6 models of varying architecture and 3 feature a lower learning rate of 0.005 rather than 0.01.  
The best performer was the simplest model, with the lowest learning rate. Important to note is the way models can catastrophically forget, as taming this erratic spiking is important for training a stable model able to generalise.

As tests progressed, the environment was made to be more complex with ‘v2’ and ‘v3’ observation spaces which included the entire map grid encoded as scalars, as well as agent health. Larger models were tested to match the new input size that came with this.

A graph with different colored lines

AI-generated content may be incorrect.

*Figure 15 – Full map with hazards, early model testing.*

In this example test, again noting that some-what smaller architecture and lower learning rate wins out in stability. This test also highlights that sometimes models were collapsing early into bad behaviours and unable to recover.

With development of the heatmap rendering function, we can examine this visually, with examples of models consistently hugging walls and corners, perhaps as a ‘safe’ option, where explorative policy isn’t forcing enough exploration to break out of it. Figure [16] shows a model with 256 Neurons exhibiting a lack of exploration and an extreme overconfidence in hugging the left wall, as opposed to Figure [17], a single layer model with just 64 neurons, exhibiting far more explorative behaviour and ability to path find.

A screenshot of a game

AI-generated content may be incorrect. A screenshot of a game

AI-generated content may be incorrect.

*Figure 16 & 17 – 1L 256N and 1L 64N policy gradient heatmaps on seed 1.*

#### Training - DQN

Three variants of deep Q Learning were attempted. Those being a regular DQN, a Fixed-Target DQN and a Double DQN. Initial testing showed some level of success early in training, but with notable instability and catastrophic forgetting.

Below features the first attempt at a Fixed-Target DQN, reaching high rewards after an initial target update, and getting worse with each further update.

A graph showing a number of points

AI-generated content may be incorrect.

*Figure 18 – Initial attempts at Fixed Target DQN.*

Unfortunately I was not able to bring deep queue networks to a point of stability and convergence.

I attempted to implement many changes to the DQN training loop, in an effort to achieve some semblance of consistent learning and stability, such as prioritised experience replay, using one hot encoded observation spaces, learning rate scheduling and more – but the best I could achieve was a level of neutral survival, with intermittent spikes of high score followed generally by total catastrophic forgetting.

A graph of blue and orange lines

AI-generated content may be incorrect.

*Figure 19 – Catastrophic forgetting behaviour in Double DQN.*

The models are clearly capable of learning – but very susceptible to what seems like over correction during training. Given more time I would like to explore this more as I believe that it’s possible to make work.

#### Training - A2C

Initial results for Actor Critic models showed promise with ability to learn, albeit sometimes erratic. Performance could either converge extremely well or catastrophically collapse as seen in the figure below.

A graph showing a number of data

AI-generated content may be incorrect.

*Figure 20 – Early A2C, instability tests.*

Results fared better with an added high entropy co-efficient, incentivising more entropic probability distributions.

Stability could be found by also simplifying the model architecture, which seemed to calm the erratic spikes in behaviour somewhat.

A graph of a graph

AI-generated content may be incorrect.

*Figure 21 – Early A2C, instability tests.*

#### Training - Multiple Levels

Training with multiple levels is a naturally unstable setting for training. Where the policy gradient models may converge on a navigation task, now each level can throw the model off, and the results oscillate heavily.

A graph of blue and orange lines

AI-generated content may be incorrect.

*Figure 22 – A2C Multiple Level Training (PCG) – Policy Collapse.*

Early tests with A2C models across 500 levels showed that a higher learning rate could lead to pretty catastrophic collapse in policy.

Implementation of the entropy coefficient to encourage more entropic action probabilities, along with keeping learning rate lower led to more stable outcomes. Early training sees an upward trend, but a lull in upwards trajectory in later training.

Also notably smaller models seemed to lose themselves less in erratic spikes.

A graph of different colored lines

AI-generated content may be incorrect.

*Figure 23 – A2C Multiple Level Training (PCG) – Stability changes.*

Policy gradient models saw instability with initial attempts across 500 levels. The figure below features two models using Boltzmann exploration at a temperature of 2, which decays quickly to greedy policy over this many episodes.

By decreasing the rate at which this Boltzmann temperature decays at, and decreasing learning rate – the models showed increased stability and a slow upwards trend in rewards.

A graph of blue and orange lines

AI-generated content may be incorrect.

*Figure 24 – Policy Gradient Multiple Level Training (PCG) – Initial instability.*

A graph of a graph

AI-generated content may be incorrect.

*Figure 25 – PG PCG training – stability through lowered exploration decay.*

Furthermore, adjusting rewards by a baseline and normalising rewards using running variables between iterations was found to in some cases slow the learning process but do seem to limit erraticism and lead to slow but positive learning trends, seen below in Figure 26.

A graph showing a number of data

AI-generated content may be incorrect.

*Figure 26 – Baseline adjustment and Running normalisation in Policy Gradient.*

#### Training - Curriculum Learning

Some models were trained not just on multiple levels, but instead multiple levels separated by tiers of difficulty. These models performed just as well as their non-curriculum counterparts – but could use further testing. There was no system in place to ‘checkpoint’ models, ensuring the next stage of training happens with a model that’s as good as it can be from the prior stage of training.A graph of a graph

AI-generated content may be incorrect.

*Figure 27 – Policy Gradient curriculum training (easy levels).*

#### Post Training – Generalisation Capability

Analysis of post training performance took place upon 100 seeds (101-200) unseen to any model during any of the training. These were all generated using the same ‘medium’ difficulty settings used for most of the training.  
Models were tested with versions trained only one seed 1, trained with 500 levels (501-1000), or 1000 levels (501-1501).

Seeds were attempted twice, once using greedy model predictions, and another using a small level of stochasticity, with a 10% chance to choose a random action. Stochasticity proved to be a very helpful addition in post training situations. No model was good enough for its purely greedy actions to govern it entirely. Random actions improved completions, reduced deaths and improved score across the board.

The goal of this analysis is to test how well each model can generalise to their new environment.

*A screenshot of a computer screen

AI-generated content may be incorrect.*Results showed overwhelmingly that models trained on a single seed, failed catastrophically to generalise to new levels. A2C models performed slightly better than the best of the Policy Gradient models trained.

A screen shot of a black and white screen

AI-generated content may be incorrect.*Figure 28 – Base Post training summary (greedy policy)*

*Figure 29 – Stochastic Post training summary (greedy policy + 10% random)*

Every model, when trained on 500 seeds, proved to be much more effective at generalising to these unseen seeds.   
Overall, the highest average scoring models were A2C models. When stochasticity is added, Policy Gradient models catch up a little bit in terms of performance. Notably A2C models die a lot less, seeming to have learnt to avoid hazards and walls better than Policy Gradient counterparts.

Following the trend of stochasticity being helpful, Policy Gradient models trained on multiple levels with a Boltzmann temperature of 6 performed better than those with 4, insinuating the extra exploration helps to round out behavioural training.

Generalisation capability tended to improve when increasing the levels trained on to 3000, and the best performing models were both models trained on 3000 levels.

Curriculum learning also noticeably improved Policy Gradient models, but the A2C model did not see an improvement. This is possibly due to the fact that in this quick implementation of Curriculum Learning, there was no waiting for a model to show convergence on a tier of difficulty, and in this case the curriculum trained 1000 level A2C model did see an erratic drop in performance at the end of its level 2 training, which may have affected its performance for future learning in level 3.

A graph showing a number of blue lines

AI-generated content may be incorrect.

*Figure 30 – A2C Curriculum Training – drop at the end (LVL2)*

The highest performance overall in terms of completion rate was from the Curriculum Learning, 3000 Seed trained Policy Gradient model, with stochastic injection in post train action choice.

This model achieved 466 / 1000 possible completions, with 78% of levels seeing at least one completion. It failed to reach the highest average reward, totalling at -9.7 – the highest being the smaller of the two A2C models, trained on 1000 seeds with a mean reward of 19.  
Comparitively, the A2C model only had a success rate of 36% of levels with 231 / 1000 completions possible.

This suggests that the Curriculum trained PG model moved more aggressively to its death in many cases, but overall possessed a better ability to generalise to its new environments and find the exit.

## Technical Challenges

Initial convergence with models proved difficult, even when beginning on basic environments, and required extensive trial and error. This was in part compounded by my own experience with Reinforcement Learning and effectively tuning hyperparameters. This definitely slowed the process of testing, and was a learning process for myself.

Machine learning can be referred to a bit like the ‘dark arts’ sometimes, and identifying what is wrong with a model can feel like searching in the dark. Is it your reward shaping? The observation space? Possibly the learning rate or exploration policy – or none of that at all?  
This was felt heavily during the DQN testing, and ultimately I could not get substantial progress with any DQN model. I suspect I have overlooked something, and with more time and further work I could fix this.

Training cycles on that note, could be very time consuming. Some runs taking multiple hours to complete, and proved a natural constraint on the extent of what I could allow myself to do. A code change might have to wait an entire day to show any results, as I swept through various hyperparameters with any changed code.

## Note:

I have not directly referenced anything in this report and was not sure if it needed a bibliography of sorts for any resources I’ve used to help me write code, so for now I’ve kept them off this document ready for the final report instead.