# Implementation

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## 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

Insert code figure for generating map, some example functions

### 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.

*(Insert environment constructor figure here)*

#### 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.

*(Insert observation space and reward settings figure here)*

#### 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 setings 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.

*(Insert step and reset function figure here)*

#### PCG Capability

A MapOptions object defines rparameters 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.

*(Insert map creation figure here)*

#### 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.

*(Insert heatmap rendering function figure here)*

*(Insert bad and good heatmaps)*

* **2.3 Models**
  + **PG Testing, basic map**
  + **PG Testing, seed 1**
  + **DQN, FDQN, DBDQN seed 1**
  + **A2C seed 1**
  + **PG PCG**
  + **A2C PCG**
  + **PG seed 1 Post Train PCG TEST**
  + **A2C seed 1 Post Train PCG Test**
  + **PG PCG Post Train PCG TEST**
  + **A2C PCG Post Train PCG Test**

### 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.

(Insert training configuration, model configuration, test and train pipeline figure here)

#### 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.

*(Insert build and train multiple models figure)*

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.

(Insert reward history plotting function figure here)

## Results

* + **PG Testing, basic map**
  + **PG Testing, seed 1**
  + **DQN, FDQN, DBDQN seed 1**
  + **A2C seed 1**
  + **PG PCG**
  + **A2C PCG**
  + **PG seed 1 Post Train PCG TEST**
  + **A2C seed 1 Post Train PCG Test**
  + **PG PCG Post Train PCG TEST**
  + **A2C PCG Post Train PCG Test**

#### 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.

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.

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 [Todo] shows a model with 256 Neurons exhibiting a lack of exploration and an extreme overconfidence in hugging the left wall, as opposed to Figure [Todo], 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 X – 1L 256N, Boltzmann 4 Figure Y – 1L 64N Boltzmann 3

##### 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.

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.

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.

##### A2C

##### Multiple Levels

Pg made less erratic with low decay Boltzmann, and baseline adjustment and running normalisation can slow the learning process but do seem to limit erraticism and lead to slow but positive learning trends.

##### Curriculum Learning

#### Post Training – Generalisation Capability

## Further Work

**- DQN working**

**- map generation more dependent on seed (fixed door, spawn, hazards?)**

**-Hierarchical Model**

**-Further testing with better models and CNN specifically**

**- I had CNN as a goal to add after converging with simpler networks, but the truth is that the DQN struggled despite a lot of attempts to improve it, and CNN may have helped with giving spatial structure meaning and leading to a good model**

**- Overfitting on coordinates and perhaps full map view, maybe better with moving vision grid and features that change based on surroundings and relation to exit for generalisation**

**-Curriculum learning using PCG system to tune levels during multiple level process, see if it leads to better rewards.**

A diagram of a company

Description automatically generated

***Figure 13 –*** *Testing plan*

The PCG System timeline was completed, unfortunately the actual training of model timeline was not fully realised.

## Technical Challenges

The process was not without difficulties, and it look a good amount of time to build a model that was able to converge on a solution for a very basic environment, before moving on to a more complex one.  
Partly due to my own inexperience in the setting and how certain parameters might be tuned, and also errors during implementation.

- time difficulties with running models

- it can be hard to discern what is going wrong, is it model tuning, observation space, reward shaping?

One of the biggest lessons coming out of this, is that honing in on what is troubling a reinforcement learning model can be a tricky process. So many factors are at play whether it be reward shaping, observation space, training hyperparameters or model structure, or even the training algorithms themselves, and the many adjustments and improvements that can be made to vastly affect a model’s learning.

A significant amount of time was spent changing parameters, messing with observations and features, adding, or tuning reward signals, and what felt like sometimes blindly trying to fit all of these puzzle pieces together into something that would work. It is often said that a lot of what goes on behind machine learning is dark arts and this definitely affirmed that for me. That being said the science of it, the search through hyperparameters and seeing the impacts that each change could have inspired me to keep going and assured me that with experience the process could be a smoother one.

It took quite some time to reach a model that converged at all on simply a basic environment, even longest still on a complex environment. Deep Q Learning did not converge no matter what was tried.

Given the nature of reinforcement learning coming with a large amount of computation, there was time constraints with simply how long it would take to run each model. At first the most basic iteration of the map and an incredibly simple policy gradient model did not take too long to run, a span of 10 minutes or less. However models like double DQN, or training on hundreds of different seeds, It could quite easily take many hours.

Some of this is due to errors in implementation that sometimes led to a lack of efficiency and increased computation time, and indeed there’s probably many ways the code could still be improved.

If doing this again I would look at unity, and ways to vectorise the learning process – training time became a pain, when id like to run sweeping, duplicated tests to track results but am held back by efficiency.

Overlooked simpler models when increasing observation space, was naïve and could have seen different results otherwise

**1. Overview**

Implementation involved the construction of multiple interweaved parts to create a reuseable reinforcement learning pipeline. To facilitate the training and testing of procedurally generated content on a model’s ability to generalise, a code structure for generating solve-able levels, that are reproduceable for further testing was developed.  
Alongside this, a custom environment for gymnasium was needed to define the training environment and how models interact with it.  
Once these were in place, models were built with Keras and trained within the environment. Focus was put on two different styles of model, being Deep Q learning and Policy Gradient.   
This lead to the creation of a model construction and training pipeline that allowed to quick creation and tuning of hyperparameters for continued testing of models across the environment as it grew in complexity. Finally, Actor Critic models were developed and tested.

The procedural content generation pipeline was a critical element to tackle early on, to ensure consistency across training, and enable reuseable testing environments to gather informed results when tuning hyperparameters and testing model structures and modifications.  
It was designed to use a node based network structure, to allow for the use of graph pathfinding algorithms, both for model use in the future and for the designing of maps.

Levels are generated by first creating a ‘base’ maze-like level that looks a little like a lattice, every second row, every second node is a path node, while all

other nodes are walls.

Using Kruskal’s algorithm a ‘perfect’ maze is built as a foundation for the rest of the level design process.  
A perfect maze is one featuring no cycles, essentially a path describes a minimum spanning tree across a graph.  
By slightly modifying the algorithm, levels are created where nodes also serve as walls, and a further algorithm is used to create cycles within the perfect maze, creating imperfect mazes that are customise-able in their nature.  
By tuning the algorithm parameters, levels can be generated in different ways, adding cycles in different manners leading to large open spaces, or smaller pockets of space. The process can be tuned to reach desired levels of cycle addition, and how biased towards wall removal removing neighbouring walls too.  
The process can also scale a level of hazards introduced into map generation, and features the ability to spawn enemies, keys to lock the exit until gathered, and of course the exit itself.  
By way of seed, the entire process is reproduceable given the same parameters and seed are used.  
Because of this, most training took place on a singular seed, attempting to replicate a medium difficulty environment.

* **2.2 Environment**
  + **Creation**

**To get started with the reinforcement learning training we need to define an environment for the models to interact with using the Farama Foundation’s gymnasium library, we can do just this.**

**The core of the environment is providing an observation space and well defined functions On how the environment the reinforcement learning agent model is in reacts and interacts with the agent. This really makes up the foundation of the whole training pipeline, with a step function that assesses the agent’s action its impact on the environment and the reward signal that flows back to the agent. There must also be a well-defined reset function that keeps this environment fresh for each new training episode without letting any Leakage of prior information into future training episodes and overall keeping the process running smoothly.**

**Alongside this comes rendering which allows for a visual pie game window to watch the training process unfold and better assess from a human perspective any potential pitfalls in behaviour and generally allowing more insight.**

**The environment that has been defined for this project allows for a constructor that can take a variety of arguments that will tailor make the environment for a particular training run. It allows for different observation spaces and reward shapings as well as defining how we’d like to render the training process if at all And stores references to the map currently being used in training the agent and all relevant information that may be needed.**

* + **Observation Spaces and Reward Settings**

**The observation space and reward settings are incredibly important for reinforcement learning process. In this environment the observation spaces available kept growing as testing continued, allowing for many different variations, whether that be coordinates for important locations, map states encoded either via one hot encoding or scalar interpretation, or numerical features such as agent health or using a star to input path length to the model.**

**Insert observation space figure here**

**Different models responded to different observation spaces, and it provided a very interesting outlook for testing How input features can greatly influence training.**

**Equally as important is how you shape the reward signal for a model, And this environment allows for a custom reward settings object to be posted, which defines reward amounts for all of the possible interactions with the environment and agent can have.**

**These rewards range from highly sparse rewards such as reaching the exit or the agent dying when touching hazards too often, but also more dense piecemeal rewards such as moving towards the goal or away from it giving both positive and negative smaller rewards, or punishments for being stuck in one place and running into walls.**

**Insert reward settings figure here**

* + **Step / Reset**

**The step function is arguably the most important function within the environment. It takes the proposed action of the model and assesses the outcome on this interaction with the environment, relaying back a reward signal, along with another observation of the environment as it now stands post action, and whether or not the training episode has finished either due to success, failure, or truncation due to time.**

**The reset function he’s also very important and is there to cleanly set the environment back into a state ready for another training loop with no leakage of information from any previous training episode, with all moving parts ready for a training loop.**

**Insert step and reset function figure here**

* + **PCG Capability**

**As the goal of this project was to assess procedural content generation when used within a training loop to improve generalisation, the environment needs to have some sort of capability to provide this.**

**Levels are created in the environment using a map options object, which defines a series of hyperparameters for the procedural content generation system to create a map of the desired settings and most importantly seated for reproducibility. The environment when initialised will take a map options object and itself create the map based on these options, there is another function that takes this created map and sets it to the map currently being used by the environment. From this point all other logic relates to this map, be it the step function or reset function or rendering.**

**Using this given a list of seeds, it is possible for the environment to simply change map at any point in time, and thus a training loop can be defined and performed over many maps using a singular environment.**

**Insert map creation figure here**

* + **Rendering**
    - **Heatmaps**

**While rendering is not necessary for the training loop, it does provide a useful insight into the training process and allows for human analysis of training.**

**I aimed to further this capability by not just providing a render function that allowed for rendering during training, but rather to render the history of training, allowing the rewatch of an entire training segment or simply parts of it. Further to this I developed a addendum that allowed for instead of rendering a training history episode by episode, rather it could deliver and save a heat map which gives a insightful and speedy overview of an agent’s behaviour across an entire training loop.**

**Insert heat map rendering function here**

* **2.4 Testing Pipeline**
  + **Customiseable, efficient testing and logging pipeline for graphs, models, histories and allowing for quick copy and paste hyperparameter tuning**
  + **Now that there was many moving parts such as the environment PCG system, model building pipeline, training loop functions Analysis functions, a pipeline developed which allowed for ease of consistent reproducible and easily tuneable testing. The process begins with defining an environment and instantiating it with the desired reward shaping and map options. Then using model configuration classes models can easily be structured with a desired layers neurons, hyperparameters, and descriptive information for future data saving and any sort of analysis and charting functions. Alongside this training configuration classes allow for a tailored training loop specific to each model, covering training parameters such as learning rate, exploration policies, the extent to which those exploration policies will decay or not, which particular type of model we are working with to provide the right training loop, how many iterations to run for, how many episodes per iteration, and much more.**

**Insert training configuration, model configuration, test and train pipeline figure here**

**Then comes a build and train multiple models function. This takes both the training configuration model configurations and a reference to the environment and builds the model in the desired manner and completes the training process given all the specified settings. This function returns a history of training as well as the trained models themselves for saving and future use. The training history object is built during the training process and accounts not just for reward history, but also agent locations throughout each episode, which enables the environments history rendering functionality.**

**Once training is complete the plot reward history function allows for an easy plotting of rewards across the training loop for each iteration, or in the case where multiple episodes happen per iteration an average across those episodes. This was a primary tool in assessing the success of each test and any modifications to a model and its training parameters, reward shaping, or observation space.**

**Insert plot reward history**

**The process of building the models from scratch it started instead of from a place of rough guidelines tutorials and exploration to try and figure out what was going to be the best starting point for my environment a lot of grid world examples out there used really small grids whereas mine was starting off with a 15 by 15 grid testing and building started from first creating the procedural content generation system then moving on to uh the environment which I used to create a simple setup that generated a bare bones basic map with the agent started in one corner and the exit was located in an opposite corner from top left to bottom right.**

**Models were built and tested on this bare bones map to get a first idea of the model structure and rough hyper parameters needed to get us very simple policy gradient model converging on the most basic implementation of my grid world.**

**From their exploration ensued trying out various depths and sizes of layers learning rates exploration policy’s from epsilon greedy to implementing Boltzmann exploration experimenting with the temperature decay rates pencil. Press enter once an initial model was found to converge I had a better understanding of what approach to take with a more complex environment from there the environment was developed further to allow for the procedural content generation system to provide customised maps complete with hazards and varying complexities of wolves. This was not the full extent of the procedural content generation system or my plans for the environment but this was the next step up in difficulty from that bare bones map. This is where a majority of the project lay in trying to explore hyperparameters and get models to converge on this more complex setting. This proved to be a much more difficult task then simply converging in a bare bones environment. Indeed deep Q learning never truly converged in this environment with my testing whether it be using straight deep Q networks or deep U networks with fixed targets or double deep queue networks. Policy gradient models however did converge after extensive testing and hyperparameter searching.**

**During this testing process it was not just model structure or hyperparameter tuning but also side by side developing new features to add to the training configuration or model configuration allowing for Allowing for various techniques like learning rate scheduling or experience replay.**

**Alongside this was also the development of the environment and the observation space that could be fed to the models as well as how rewards were delivered and allowing for more reward shaping. This can be seen with the reward settings class here.**

**Insert reward settings class figure here**

**Insert training config model config figure here**

**The process of tuning every enforcement learning model has been an enlightening one and often it’s very hard to know exactly what is either holding your model back or driving improvements is it your reward shaping and the signal that’s being given back from the actions it takes? Is it the hyperarameters you set for your training loop the learning rate how fast its exploration policy decays exactly how it decides to explore? Is it the depth you’ve allowed the model and its ability to fit or over fit the environment? Or is the input and what you’ve allowed the model to see giving everything it needs to or too much?**

**With hundreds of models developed and tested, hyperparameters tuned, model structures tried, observation spaces developed, reward shapes fed- I did manage to bring some models to converging on this more complex environment.**

**By this point given how long the tests take I found myself running out of time to explore everything fully, and found that deep queue networks not converging was holding me back from attempting the rest of the project.**

**At this point it was determined the best possible route was to continue with the original goal of the project and put the procedural content generation system to work by testing what converged models I did have on a set of 100 unseen seeds.**

**Up until this point every model had been trained on seed one for consistency through training. At this point I chose to test the performance of an already trained model on 100 unseen levels. Seeds 101 to 200 we used for the post training runs.**

**As expected these baseline models trained on one environment only over fit massively but they did manage to score well on some levels.**

**Insert post train figures here**

**It can be seen from the figures above that even with extensive overfitting the models still were able to perform on a few levels this surprisingly did not correlate extensively with sheer similarity to the original seed one. There was the slightest correlation with With the distance that the door had to the original location of the door in seed one where the model was trained.**

**Duplicate model testing show luck plays some part in exploration and initial weights. Further work should include averaging multiple simulations.**

**Early DQN testing showed good initial results, but unstable over time.**

**A2C benefitted greatly from lower learning rates, and entropy rate tuning,**

**1000 multi level a2c showed instability during pcg process, further fine tuning of parameters could help – but also smaller models showed lowering complexity also helped with regularising.**

**The results for the first 1000 level a2c model come from a model that failed catastrophically during training, overreacting to an iteration, possibly exploding gradients, catastrophically forgetting and collapsing. This shows the need for stability in hyper parameters and model architecture and training choices when faced with a changing environment.**

**Having some system of early stopping would possibly help with curriculum learning too**