Ghost In the Grid: Challenges for Reinforcement Learning in Grid World Environments

Christopher David James Bamford

Ph.D. thesis

School of Electronic Engineering and Computer Science
Queen Mary University of London

2022

Abstract

The current state-of-the-art deep reinforcement learning techniques require agents to gather large amounts of diverse experiences to train effective and general models. In addition, there are also many other factors that have to be taken into consideration: for example, how the agent interacts with its environment; parameter optimization techniques; environment exploration methods; and finally the diversity of environments that is provided to an agent.

In this thesis, we investigate several of these factors.

Firstly we introduce Griddly, a high-performance grid-world game engine that provides a state-of-the-art combination of high performance and flexibility. We demonstrate that grid worlds provide a principled and expressive substrate for fundamental research questions in reinforcement learning, whilst filtering out noise inherent in physical systems. We show that although grid-worlds are constructed with simple rules-based mechanics, they can be used to construct complex open-ended, and procedurally generated environments.

We improve upon Griddly with GriddlyJS, a web-based tool for designing and testing grid-world environments for reinforcement learning research. GriddlyJS provides a rich suite of features that assist researchers in a multitude of different learning approaches. To highlight the features of GriddlyJS we present a dataset of 100 complex escape-room puzzle levels. In addition to these complex puzzle levels, we provide human-generated trajectories and a baseline policy that can be run in a web browser. We show that this tooling enables significantly faster research iteration in many sub-fields.

We then explore several areas of RL research that are made accessible by the features introduced by Griddly:

Firstly, we explore learning grid-world game mechanics using deep neural networks. The *neural game engine* is introduced which has competitive performance in terms of sample efficiency and predicting states accurately over long time horizons.

Secondly, *conditional action trees* are introduced which describe a method for compactly expressing complex hierarchical action spaces. Expressing hierarchical action spaces as trees leads to action spaces that are additive rather than multiplicative over the factors of the action space. It is shown that these compressed action spaces reduce the required output size of neural networks without compromising performance. This makes the interfaces to complex environments significantly simpler to implement.

Finally, we explore the inherent symmetry in common observation spaces, using the concept of *geometric deep learning*. We show that certain geometric data augmentation methods do not conform to the underlying assumptions in several training algorithms. We provide solutions to these problems in the form of novel regularization functions and demonstrate that these methods fix the underlying assumptions.

Acknowledgements

Firstly, I'd like to thank my family. My Mother and Father have been supportive and have listened to my worries and given me constant and unwavering support throughout the years of my PhD. My Sister and Brother have always been on the other end of the phone when I needed to vent about frustrations around writing up, or just needed a pick me up.

I'd like to thank my supervisor Simon Lucas, who gave me this incredible opportunity to pursue this PhD. Something I've always aspired to do in my life. Thanks for all the feedback on paper and ideas that have been thrown around over the past few years!

I want to thank all of the friends and colleagues I have made in the Queen Mary Game AI Lab and across the Intelligent Games and Games Intelligence (IGGI) network. In particular friends in the office that have been good fun: James Goodman, Sebastian Berns, Elena Gordon-Petrovskaya, Nathan John, Sara Cardinale, Martin Balla, Michael Aichmuller, Remo Sasso, Michelangelo Conserva, Linjie Xu and Sahar Mirhadi. Also I would like to thank Raluca Gaina, Diego Perez Liebana and Suzanne Binder for organising many games nights, conferences and other events made me really feel like part of a community.

A huge thank you goes out to the team at MetaAI where I did two internships and gained my biggest publication: Tim Rocktäschel, Minqi Jiang and Mikayel Samvelyan, Heinrich Kuttler and Ed Grefenstette. You helped me to present ideas in a much more principled and scientific way, and I have learned a huge amount from my time working with you.

I also want to say a thanks all the contributors and collaborators I have had on several open-source projects: Shengyi (Costa) Huang, Aaron Dharna Ville Kopio, Clemens Winter and Alvaro Ovalle.

Finally I'd like to thank Margaret Ainsley and our pet cat Poppy. I know at times supporting me was difficult but I appreciate everything that you have done for me.

Contents

Chapter 1

Introduction

For artificial general intelligence (AGI) to be realized, there are many hurdles that need to be overcome. These hurdles fall into an array of different areas of science and technology. One of the most promising directions of research is known as deep learning (DL). Many recent advances have been made in hardware, infrastructure, and software to support progress in deep learning. These advantages have led to rapid progress in many scientific fields.

The focus of this thesis surrounds a particular area of DL which is known as deep reinforcement learning. deep reinforcement learning concerns creating algorithms that control the actions of an agent within an environment. The agent can take many forms such as a mobile robot, a character in a game, or a more high-level control issuing hierarchical commands to other smaller low-level control agents. The environments in which these agents perform tasks can be real-world physical systems such as a robot arm, or simulated, where a virtual environment is implemented in software.

While real-world autonomous agents are a desirable goal for realizing the practical potential of these systems, they are particularly difficult to achieve. Deep learning is well known to require a large amount of data to train, especially when the goal of training is for a system to act *generally*. The most common methods of training DL systems still struggle to perform tasks in environments that are different from those seen during training. In real-world terms, the amount of required data or experience for a deep learning solution to achieve the same accuracy or effectiveness as a human counterpart is often intractable. Additionally, it is hard to control the parameters of real-world environments, such as noise in sensor readings, lighting conditions, and tolerances in manufacturing processes. This leads to large differences in the behavior of agents as they struggle to overcome small physical differences in hardware implementations.

In order to attempt to side-step the limitations of learning in physical environments, simulated environments can be used. However, when using these methods, it is still challenging to effectively transfer any simulation-learned behaviors to the real world [214]. High-fidelity realistic environments are also extremely difficult to build which would mean accurately reproducing a digital physical model of the environment in question.

On the other hand, simulated environments provide a configurable test bed for a wide variety of research questions. In scientific research and mathematical reasoning, a common approach is to take a particular problem and break it down into component parts in order to understand these fully and separately. As artificial intelligence is arguably one of the most complex scientific problems, breaking these problems down into milestones is crucial to making progress in understanding.

Instead of trying to replicate the dynamics of the real world, many simulated environments try to capture the important factors in particular problems while removing confounding factors that would otherwise be found in physical systems. One class of environments that we concentrate on in this thesis is that of grid worlds. Grid worlds can be described as environments where actions and states are typically discrete. Grid worlds can encompass a huge variety of different tasks and problems. In the simplest case, a grid world could be a task where an agent has to move to a particular goal state in the lowest number of up, down, left, and right movements. However, grid worlds can also encompass complex multi-agent strategy games where there are different classes of units, upgrades, and resources to manage.

Several general platforms for generating simulated environments have been created in order to build games for entertainment purposes. These are generally referred to as "game engines". However, these platforms, such as Unity [**?**] and Unreal Engine [83] are targeted at creating high-fidelity commercial games, and their support for research applications is limited. This lack of first-class support leads to environments that produce data inefficiently and integration with these environments usually requires significant engineering expertise to modify the interfaces to make them compatible with RL algorithms.

In this thesis, we aim to answer the following research questions:

- How can we design and build a game engine that provides a high volume of relevant data for a wide variety of RL research topics, while maintaining efficiency and functionality?
- What essential features should be incorporated into this game engine to ensure it can generate the most pertinent training data, along with diverse scenarios and adaptable formats?

- How can we ensure the data provided by the game engine can be effectively used by various types of algorithms?

In order to provide a platform for answering these questions, we introduce Griddly [29], a unified framework for grid-world environments. We describe how Griddly can be used to create many different kinds of environments for research, and show that Griddly provides a state-of-the-art combination of efficiency and flexibility. We then expand on the Griddly platform by introducing GriddlyJS which allows researchers to easily create, debug and run experiments in a web-based graphical user interface. Griddly and GriddlyJS are introduced fully in section 1.1 and described in detail in section 3.

Taking advantage of the flexibility of the Griddly platform, we then investigate several other areas of research. Firstly we concentrate on the interfaces between the environment and the agent itself. These interfaces are commonly known as action spaces and observation spaces. In section 1.2, we introduce the concept of *conditional action trees*, which are a method of generalizing complex discrete action spaces into a canonical form. We show that conditional action trees can simplify the action spaces while maintaining the accuracy of more complex action spaces. This leads to smaller and therefore more computationally efficient networks.

We also use the Griddly platform to perform additional experiments where we take advantage of natural priors of grid-world environments. More specifically, we develop the *neural game engine*, covered in section 5, which learns the local dynamics of many grid-world games and can reproduce the states and rewards of these games with near-perfect accuracy over long time-scales. This allows agents to learn complex puzzle games in their own *imagined* environments.

Another property of many grid-world environments is that they can have state and action symmetry. For example, given a set of actions to solve an environment, there may exist a linear transformation of the state and corresponding action transformation which can solve the transformed environment. We explore these ideas in section 6 using the mathematical framework of invariance and equivariance [199, 63, 299, 286] and show that under geometric data augmentations, certain RL methods that employ off policy corrections have certain requirements which are violated. We provide regularization methods that alleviate these issues and show that this greatly improves sample efficiency in training.

1.1 Griddly

Designing and implementing game environments to test the ability of different algorithms, such as reinforcement learning (RL), to reason, generalize and plan can be complex and time-consuming. Even simple environments require implementing a number of common components such as rendering, game mechanics, and optimization. A few solutions have been developed to abstract away the implementation details of environments and present researchers with a simplified interface. This allows researchers to concentrate on building the specifics of environments to test their algorithms.

In chapter 3 we introduce Griddly [29] which provides a highly configurable and optimized platform for building grid-world games for artificial intelligence research. Griddly uses a domain-specific language known as *Griddly description YAML* (GDY) which allows an unprecedented level of configurability in all of the key areas described above. Not only can GDY be used to create single-player puzzle games like those in GVGAI [222], MiniGrid [54], DMLab2D [35] and other toy problems, but it can be used to create multi-agent and RTS style games with partial observability and complex resource systems. GDY also provides multiple built-in observation representations such as sprite-based isometric rendering, simple shape-based tiles, and minimal state tensors. In addition to the wide array of configurability that Griddly provides, the underlying Griddly engine is heavily optimized in both computational speed and memory usage by taking advantage of hardware-accelerated rendering techniques.

1.1.1 GriddlyJS

On top of the Griddly engine, we introduce GriddlyJS [30], a web-based integrated development environment (IDE) based on a version of the Griddly engine which is integrated with the web browser using WebAssembly (WASM). GriddlyJS provides a simple interface for developing and testing grid-world environments using a visual editor, with support for highly complex game mechanics and environment generation logic. The visual editor allows the rapid design of new levels (i.e. variations of an environment) via a simple point-and-click interface. Environments can be tested via interactive control of the agent directly inside the IDE.

We use GriddlyJS to build a dataset of 100 challenging *escape room* game levels. We show that even with strong RL agents trained with state-of-the-art domain-randomization techniques, these human-designed levels are very difficult to solve. For each level, we also use GriddlyJS to record trajectories of how these levels can be completed. These trajectories can then be used in downstream reinforcement learning algorithms such as behavioral cloning.

1.2 Environment Interfaces

In addition to the tooling introduced in the previous two sections, we focus our efforts on the interface of the agents to these grid-world environments. More specifically, we study the effects of different configurations of action spaces in which the agent can interact with the environment.

In the grid-world environments that Griddly provides, the actions are typically discrete to match the nature of the deterministic mechanics, but creating a common interface that can encompass all possible actions has many considerations. For example, actions can be multi-factored, requiring the agent to select actions to be performed in a hierarchical sense, where an action "type" is selected before an action "parameter". In the case where an environment exposes *multiple* different action "types" for *multiple* different controllable units, the number of possible actions in this interface can increase exponentially. This section provides solutions to these issues.

1.2.1 Conditional Action Trees

Training reinforcement learning agents to solve environments with large, complex action spaces is a notoriously difficult task [163]. Several methods have been proposed to try to either reduce the space of actions by re-using model outputs for different action types [191, 292], provide auxiliary information to facilitate the exploration of large numbers of possible actions [240, 209, 156], or simplify the manipulation of the action spaces through action embeddings via mechanisms such as attention and graphs networks [4, 186, 5]. We propose the *conditional action tree* [28] as a paradigm to generalize several of these methods. *Conditional action trees* can be used to describe action spaces in a way that naturally reduces the required policy model output size whilst also allowing hierarchical action parameterized and action reduction using invalid action masking[138]. We show how many action spaces frequently found in single, multi-agent, and real-time strategy (RTS) games can be described using *conditional action trees.* We also show that agents that have access to *conditional action trees* as part of their state observations can learn high-performing policies.

Conditional action trees are presented in detail in section 4.2

1.3 Environment Modelling

The majority of this thesis is concerned with the usage of environment engines such as Griddly as the primary substrate in which an agent can learn to complete tasks. This is commonly known as "model-free" reinforcement learning as the agent does not have access to an *internal* model of the game engine itself. Model-based reinforcement learning refers to algorithms in which agents build or use an internal model of the environment itself.

Recent work has shown that in a situation where the agent has no access to a pre-defined model of an environment, it can be possible for the agent to learn its own model. This approach attempts to solve the problem of predicting future states of the environment given a set of actions. In situations where there where it is difficult or inefficient to collect lots of environmental data, sufficiently accurate models of the environment can be used. Learning a model of the environment has many potential benefits over traditional model-free reinforcement learning, and provides a gateway to much more complex and intricate learning algorithms. Model-based reinforcement learning techniques are discussed in section 5.1.7.

1.3.1 Neural Game Engine

In chapter 5 we introduce a technique for learning accurate deterministic grid-world games using a deep neural network architecture known as the *neural game engine* [26].

The *neural game engine* (NGE) can learn grid-based arcade-style games of any dimension with high accuracy from a limited number of game ticks (state transitions). Additionally, the architecture can scale to grid games of any number of tiles without loss of accuracy. The NGE engine is trained on several deterministic grid-world games from the GVGAI environment [222], an updated version of pyVGDL [243] which provides many grid-based games under the OpenAI gym wrapper [45].

We compare the *neural game engine* to several state-of-the-art methods used in predicting the transition function of grid-world games and show that it outperforms these methods in terms of compounding errors. These experiments are performed in the pre-cursor to the Griddly Engine, (GVGAI), but the environments used are available in Griddly with significant improvements in speed and flexibility.

In chapter 5 we introduce the *neural game engine* and associated experiments in more detail.

1.4 Equivariant Data Augmentation

Data augmentation is a promising new technique for RL that is regularly used in supervised learning [257]. Data augmentation adds training data, either via fixed transformations, or random perturbations, with the goal of increasing the training data distribution to improve generalization to unseen examples. Unfortunately, certain classes of data augmentation that are applied in supervised learning, are problematic when applied in RL settings, and are commonly avoided for this reason [181, 174]. In particular, certain geometric data augmentations such as rotation and flipping do not produce invariant policies with respect to the inputs. As an intuitive example, if an agent traverses from left to right, but that map is flipped during data augmentation, the output policy will now have to also flip so the agent traverses from right to left instead. This is not the case with augmentations that are invariant, such as those in supervised learning, where the output is usually a classifier that would still output the same classification, regardless of the transformation used.

In order to solve this problem, we analyze data augmentation with a mathematical approach known as *geometric deep learning. Geometric deep learning* investigates the usage of strong geometrically-based priors that occur frequently and naturally in many datasets. For example, in computer vision, it is common that transformations such as scaling and rotating input images should result in the same classifications [34, 299, 167]. This is shown to have many advantages over just increasing the amount of data using data augmentation.

In section 6 we apply the mathematical framework of *geometric deep learning* to the problem of data augmentation in reinforcement learning. We show that certain algorithms, such as the v-trace algorithm [86], which is used for off-policy corrections, have certain requirements which are not met when data augmentation is applied. We propose regularization functions to improve these methods and show empirically that these methods improve training under data augmentation.

1.5 Contributions

This section contains a list of contributions of both **peer reviewed academic publications**, **open source contributions** and **documentation and tutorials**.

1.5.1 Publications

GriddlyJS

Title GriddlyJS: A Web IDE for Reinforcement Learning
Venue NeurIPS Datasets and Benchmarks, 2022
Authors Bamford, Christopher and Jiang, Minqi and Samvelyan, Mikayel and Rocktäschel, Tim.
Date 2022-06-16
URL `https://arxiv.org/abs/2207.06105`

GriddlyJS introduces a set of tools which allow researchers to easily generate challenging reinforcement learning challenges for many different research questions. **GriddlyJS** includes interfaces for creating, debugging and evaluating environments, recording human trajectories for behavioural cloning and view trajectories of trained policies. **GriddlyJS** is explained in detail in chapter 3.11

Conditional Action Trees

Title Griddly: A Platform for AI Research in Games
Venue 2021 IEEE Conference on Games(CoG)
Authors Bamford, Christopher and Ovalle, Alvaro.
Date 2021-08-17
URL `https://ieeexplore.ieee.org/document/9619093/`

This paper introduces a canonical method of representing hierarchical action spaces where only sub-sets of actions are available at each time-step. Further detail on **Conditional Action Trees** can be found in chapter 4.2

Griddly

Title Griddly: A Platform for AI Research in Games
Venue Workshop on Reinforcement Learning in Games
Authors Christopher Bamford, Shengyi Huang, Simon Lucas.
Date 2021-02-08

URL `http://aaai-rlg.mlanctot.info/2021/papers/AAAI21-RLG_paper_34.pdf`

Griddly is an efficient and flexible game engine built for grid-world based research. **Griddly** underpins many of the contributions in this thesis, including **Conditional Action Trees** and **GriddlyJS**. **Griddly** is explained in detail in chapter 3.

Neural Game Engines

Title Neural Game Engine: Accurate learning of generalizable forward models from pixels
Venue 2020 IEEE Conference on Games (CoG)
Authors Bamford, Chris and Lucas, Simon.
Date 2020-10-20
URL `https://ieeexplore.ieee.org/document/9231688`

Previously to the creation of **Griddly**, experiments were undertaking in learning the mechanics of grid-world games, specifically those in the GVGAI framework. **Neural Game Engines** learn accurate models of several GVGAI games, and outperforms other methods in terms of accuracy of predicting many states into the future.

Neural Game Engines are discussed in chapter 5

Local Forward Modelling

Title A local approach to forward model learning: Results on the game of life game
Venue 2019 IEEE Conference on Games (CoG)
Authors Lucas, Simon M. and Dockhorn, Alexander and Volz, Vanessa and Bamford, Chris and Gaina, Raluca D. and Bravi, Ivan and Perez-Liebana, Diego and Mostaghim, Sanaz and Kruse, Rudolf.
Date 2019-03-29
URL `https://arxiv.org/abs/1903.12508`

In this paper, contributions were made to the comparisons of different forward model structures. More specifically a deep neural convolutional auto-encoder neural network was used to predict local changes in the forward model of the environment.

1.5.2 Open Source Contributions

Griddly

Griddly is an entirely open source project which can be found at: `https://github.com/Bam4d/Griddly`. The python libraries for Griddly are automatically built and distirbuted on PyPi `https://pypi.org/project/griddly/` and available for Linux, Windows and MacOS development. Additionally GriddlyJS source code is provided in a sub-folder of GriddlyJS `https://github.com/Bam4d/Griddly/js`

Grafter

Grafter is an implementation of the Crafter [117] game using the Griddly Engine. Recreating Crafter using the Griddly engine has many benefits such as multi-agent support, faster rendering, and highly configurable observations spaces. `https://github.com/GriddlyAI/grafter`

Grafter Escape Rooms

The **Escape Rooms** project is the resulting dataset included in the original GriddlyJS Paper [30]. This dataset contains 100 challenging human-designed escape room levels. This dataset is designed specifically to be a highly complex challenge for Reinforcement Learning. The dataset also contains human trajectories for level for use in behavioural cloning or other methods such as verifying or measuring ppolicies.

Entity Neural Networks

Entity Neural Networks is an open source project that allows entity based observation spaces to be used in reinforcement learning algorithms. This project was developed mainly by Clemens Winter of OpenAI, but is natively supported by the Griddly Engine. `https://github.com/entity-neural-network`. Griddly was used to benchmark several different features of the Entity Neural Network library, and several Griddly environments are included as part of the collection of examples.

RLLib

RLLib contains an efficient and distributed implementation of the IMPALA [86] which is used in several Griddly-based projects such as the Conditional Action Trees paper [28]. During development of this project and other experiments, several bugs were found in the underlying implementation of IMPALA which were corrected, these contributions can be found here: `https://github.com/ray-project/ray/commits?author=Bam4d`

1.5.3 Documentation and Tutorials

Griddly Documentation

Alongside the code of Griddly, extensive documentation has been authored which can be found online at `https://griddly.readthedocs.io/en/latest/`.

A pdf of up-to-date documentation can also be downloaded from: `https://griddly.readthedocs.io/_/downloads/en/latest/pdf/`

Griddly Tutorials

In the Griddly documentation [27] there are tutorials for implementing various complex mechanics: **Making Sokoban**, **GDY Schema Tutorial**, **Proximity Triggers**, **Custom Shaders**, **Projectiles**, **Stochasticity**, **Level Design**, **A* Search**. In addition there are also addition to these tutorials which concentrate on implementation of Griddly environments, there are also articles explaining how to integrate Griddly with **Procedural Content Genration** algorithms and popular Reinforcement Learning Environments such as **RLLib** [182],

Chapter 2

Background

This chapter introduces the main concepts of Reinforcement Learning that lay the foundations of the contributions in this thesis.

It is assumed that the reader of this thesis is already familiar with many of the basics of neural networks such as stochastic gradient descent and information theory. An in-depth discussion of these concepts and their origins will not be given.

We provide a detailed background on Reinforcement Learning to provide context for the main chapters in this thesis. In addition, we provide further background at the start of each chapter to deepen the reader's understanding before each section.

Reinforcement Learning is one of the most active areas of research in artificial intelligence and is arguably a major step toward building Artificial General Intelligence. Reinforcement learning aims to reduce any agent in an environment to a set of functions that try to choose actions that the agent can undertake in order to maximize a particular reward. The most basic form of reinforcement learning tries to optimize one specific reward function in a particular environment, for example, trying to reach a high score on the Atari game Breakout [196]. However, the performance of Reinforcement Learning agents is strongly correlated to the distribution of the data it is trained with. If an agent is trained to perfect a small subset of training environments, it will likely fail to perform well in environments that it has never seen before. This *generalization gap* problem is yet to be solved. Other more complex methods of reinforcement learning attempt to learn higher-level hierarchical concepts such as points of interest in the game. Other algorithms attempt to solve problems given contextual markers such as textual hints or in-game prompts. This section aims to provide an in-depth review of the most impactful reinforcement learning techniques. More detail will be given on particular techniques that are relevant to the contributions of this thesis.

Reinforcement learning (RL) in its most simple case can be represented as a Markov Decision Process (MDP), defined as a tuple $\mathcal{M} = (S, A, O, \Omega, \mathcal{T}, R, \gamma)$, where S is the state space, A is the action space, O is the observation space, $\Omega : S \to O$ is the observation (or emission) function, $\mathcal{T} : S \times A \to S$ is the transition function, $R : S \to \mathbb{R}$ is the reward function, γ is the discount factor, and ρ is the distribution over initial states. At each time t, the RL agent takes an action a_t according to its policy $\pi(a_t|o_t)$, where $o_t = \Omega(s_t)$, and the environment then transitions its state to $s_{t+1} = \mathcal{T}(s_t, a_t)$, producing a reward $r_t = R(s_{t+1})$ for the agent. Both the observation and transition functions can provide a source of randomness, i.e. the observation function may return the observation o_t from a distribution $P(o_t|s_t)$ and the transition function may return the state from a distribution $P(s_{t+1}|s_t, a_t)$.

In many cases the observation function $\Omega(s_t)$ is a transformation of the state s or the identity function. In this case, from the perspective of an agent, the environment represented by the MDP M is known as fully observable. In cases where the observation only contains a subset of the information contained in s, for example, a cropped, or masked image, the environment is referred to as partially observable.

In both cases, it is common to see the state referred to as s_t, even if the environment returns a partially observable state. When RL equations and models are defined without the context of a particular environment, it is also common to use the state variable s_t in definitions instead of the observation variable o_t.

To put this into practical terms, a minimal example of a fully observable environment would be the game of chess, where each player can see the entire board. In contrast, the game battleships would be considered partially observable as both players have limited information about the positions of the other player's pieces.

Reinforcement learning algorithms attempt to learn to maximize a function such as the value function $V(s_t)$, or action-value function $Q(s_t, a_t)$.

The value function $V(s_t)$ is an estimate of the return of the environment. The return G_t, also known as the cumulative reward or the total discounted reward, is a measure of the total reward an agent receives over a sequence of time steps t. It represents the sum of all rewards from the current time step until the end of an episode:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \tag{2.1}$$

Action Value Function (Q-function) denoted as $Q(s_t, a_t)$, estimates the expected return when starting in a particular state s_t, taking a specific action s_t, and following a certain policy thereafter. It represents how valuable it is to take a particular action in a specific state.

Early experiments with deep neural networks to estimate value functions were proposed in [233], however, Deep reinforcement learning was more widely introduced in [196] where Q-learning was combined with a deep neural network to predict the value function directly from the pixels of the Atari learning environment [37].

In these experiments, the Q function is updated iteratively as the agent gathers experience in the environment. The simplest form of the Q-learning temporal difference update algorithm is given by:

$$Q_{new}(s_t, a_t) = Q_{current}(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q_{current}(s_t, a_t) \right] \tag{2.2}$$

Where r_t is the reward given by the environment at step t, γ is the discount factor which is effectively a measure of how a reward should be discounted over time and α is the learning rate. The temporal difference (TD-error) is calculated as the difference between the predicted and actual discounted reward.

Several modifications to Q learning have been proposed such as double Q learning [124, 288] which uses two Q functions, one to estimate the value and one to estimate the policy:

$$\begin{aligned} Q^A_{new}(s_t, a_t) &= Q^A_{current}(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q^B(s_{t+1}, a) - Q^A_{current}(s_t, a_t) \right] \\ Q^B_{new}(s_t, a_t) &= Q^B_{current}(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q^A(s_{t+1}, a) - Q^B_{current}(s_t, a_t) \right] \end{aligned} \tag{2.3}$$

Using two Q functions in this way allows much faster training and avoids the problem of overestimating the value of certain actions, which is a common problem in Q learning.

When using Q-learning as described above, the action space is required to be discrete as the maximum over the possible actions needs to be taken to update the Q-function. However, in complex control tasks where movement requires more fine-grained control, such as physics environments where forces have to be applied to joints or actuators, continuous actions have to be used. Games with continuous action spaces tend to have simulated physical environments, such as racing games with steering wheels, flight simulation games with continuous joysticks, or first-person shooters where the mouse is used to aim.

In these kinds of environments, *Policy Gradient Methods* can be used.

2.1 Policy Gradient Methods

Instead of learning an intermediate value function such as with Q learning, policy gradient methods attempt to learn a distribution of the actions $P(a|s_t)$ that gives a high probability of future rewards. Policy gradient theorem [276] shows the optimal policy can be found by gradient ascent of:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}[\nabla_\theta log \pi_\theta(a|s) Q^\pi(s,a)] \tag{2.4}$$

A common way of using deep neural networks with the policy gradient theorem is to use actor-critic architectures [73].

Actor-critic architectures contain two function approximators, one that learns the policy function $\pi_\theta(a|s)$ and one that learns the value function $Q^\pi(s,a)$. The value function can be estimated using TD error updates as in equation 2.2, and then gradient ascent can be used to learn the parameters for π_θ.

In episodic reinforcement cases where the value function can be easily calculated as the sum of discounted rewards over an episode, REINFORCE [302] can be used. This method, however, can produce updates with high variance as each action is judged based on long sequences of subsequent actions. This can also result in slower and more unstable training.

[258] shows that the stochastic policy term $\pi_\theta(a|s)$ in equation 2.4 can be replaced by a deterministic term $\mu_\theta(s)$. The deterministic policy function $\mu_\theta(s)$ is a limited case of the policy $\pi_\theta(a|s)$ where the variance is negligible.

Like Q-learning, deterministic policy gradient methods can be extended to use deep neural networks [183].

As stated before, the q-value function $Q^\pi(s,a)$ can be learned by using the TD error from episodes. However, the TD error itself can be estimated by learning an advantage function.

The advantage of an action is a measure of how much better or worse an action is compared to the average action taken in a particular state. It quantifies the impact of selecting a specific action rather than following the current policy.

Mathematically, the advantage of an action a in a state s under a policy π is defined as the difference between the action value function $Q(s,a)$ and the state value function $V(s)$:

$$\begin{aligned} A(s_t, a_t) &= Q(s_t, a_t) - V(s_t) \\ &= r_t + V(s_{t+1}) - V(s_t) \end{aligned} \tag{2.5}$$

This yields the Advantage Actor-Critic (A2C) algorithm [198] where the advantage is used instead of the Q-function:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}[\nabla_\theta log\pi_\theta(a|s)\mathbf{A(s,\ a)}] \tag{2.6}$$

Since A2C methods are learning an estimate of the TD error, this can introduce bias which can lead to convergence to poor policies or lack of convergence at all. This problem is addressed in [251], where a Generalized Advantage Estimator is introduced. Generalized Advantage Estimators attempt to stabilize training by implementing two techniques. The first is to use a discounted reward function, similar to Q-learning. The second technique uses an exponentially weighted average of the calculated advantage terms within an episode. Both techniques can be adjusted by hyperparameters γ and λ, respectively. These terms allow a tradeoff between bias and variance to be configured.

Variance in the updates of the policy and value estimation can also be reduced by modifying the update functions to reduce their step size intelligently. [250] introduces Trust Region Policy Optimization (TRPO), which in theory guarantees monotonic improvement to policies, however in practice, uses several approximations, and Proximal Policy Optimization (PPO) [252] which modifies TRPO to avoid the use of expensive computations. [279] generalizes [250] [252] by showing that they are in fact a case of Taylor Series Policy Optimization (TayPO) where only the first expansion term is used.

Variance in updates in other deep learning problems is generally reduced by using large batches and averaging updates over these. In [196], an experience replay buffer is used, which keeps a record of state-action pairs, and updates the policy intermittently by sampling batches from the experience replay buffer. Simple experience replay buffers only keep a certain amount of recent history from the agent's experience in the environment and disregard old history. This means that events in this history appear in the training set with the same proportion of time they would if there was no replay buffer. In reinforcement learning environments, the states that result in high rewards are usually rare. This would result in high TD errors as they are rarely encountered. [245] introduces Prioritized Experience Replay (PER), which uses a priority queue as a replay buffer using the TD error as the priority metric. This leads to states that have low TD errors being removed from the replay buffer and states that have high td errors being more likely to be used in policy updates.

2.2 Exploration

In most reinforcement learning algorithms, the agent initially has no experience and knows nothing about the environment in which it is placed. This makes the initial policy for the agent essentially random. The agent will randomly move around the environment, learning that the value of most states is essentially 0. Once the agent reaches a state that contains a non-zero reward, the discounted TD error can be applied to previous states, which will help the agent to follow or avoid similar trajectories in the future. In many environments, however, for the agent to reach states that contain rewards, the agent must perform a series of specific actions. In many cases, randomly choosing actions will not reach these states.

Exploration in reinforcement learning is a very difficult challenge with many possible solutions that may work in some environments but not others. This section covers model-free methods. However, many model-based methods are covered in section 5.1.7.

In model-free exploration with a deterministic policy that chooses actions by sampling the maximum Q value, an exploration policy must be implemented to stop the agent from taking the same actions every step. ϵ-greedy exploration is the most common technique, where ϵ is the probability that a random action will be taken instead of the action given by the maximum Q-value. In stochastic policies where the action is sampled from a distribution, random exploration arises naturally. In actor-critic reinforcement learning, soft actor-critic [116] methods can be used to aid exploration by adding a constraint that tries to increase the entropy of the policy.

Several methods of improving ϵ-greedy exploration involve repeating the randomly selected actions. For example, in [187], "sticky actions" are introduced, which repeat the previous action based on a probability $1 - \varsigma$. Similarly, in [71], actions are repeated for lengths of time defined by distributions.

In the Atari Learning Environment, several games are defined as "hard exploration" problems due to the complexity required to gain rewards. Montezuma's Revenge and Pitfall fall into this category. In both games, the agent controls a 2D avatar in a platformer-like environment in which the avatar can move from one room to another by passing through walls or doorways. Additionally, the order in which the avatar interacts with objects is important across large time scales. For example, moving across several rooms to pick up a key and backtracking to find a door that can now be opened.

Go-Explore [82] is one of the first methods to solve both Montezuma's Revenge and Pitfall with scores higher than the average human baseline. Go Explore explores randomly using standard ϵ-greedy methods and then stores states that are interesting and trajectories of how to get to those states. Interesting states can be states that are not visited very often or states that are near other interesting states. Once many interesting states are found, the environment is reset to those states, and exploration is performed to find new interesting states. In the second phase, the trajectories that find the interesting states are used to teach a policy that learns how to get to those states. Because go-explore uses many heuristics and involves state resets, it is impractical for many environments, such as those with highly stochastic transition functions or physical environments where state resetting is not possible or time-consuming. Environments have to be specifically compatible with the Go-Explore algorithm, which, in many cases, defeats the simplicity of maximizing the expected returns that reinforcement learning promises.

There are numerous techniques utilized to enhance exploration and make sure that agents encounter a wide range of possible game scenarios. Methods such as counting the number of times states have been visited can be used for intrinsic motivation [38]. For example, in [215], the observed states are encoded using a deep neural network as a pseudo-counting algorithm. The number of times these states are visited is then counted, and a reward inversely proportional to the count is used as an intrinsic reward. This encourages the agent to visit states that are not visited as often and avoid states that it has seen many times before.

Many exploration policies that work in some environments can be poor in others, [21] and [20] learn several different exploration policies which are selected depending on the task to be solved. The exploration policies give intrinsic rewards to the agent based on the novelty of states that are being observed by the agent, similar to [217] and [48]. These methods commonly learn models of their environments in which to predict certain useful factors such as expected future rewards or measures of information gain. These model-based methods are explored further in section 5.1.7

2.3 Parallelization and Distributed Methods

Currently, there is a challenge with deep neural networks, particularly deep reinforcement learning: They demand a significant amount of data to establish highly efficient policies. In reinforcement learning, many environment models can only process actions and produce their associated states relatively slowly. Many environments are not inherently designed to produce states efficiently for reinforcement learning as they are software emulated [37], [205], or are ports of existing games that contain a wrapper [292].

A method that is commonly used to generate more data for reinforcement learning algorithms is to parallelize the execution of the environments and use the experience from many agents running in parallel to update a global model. This section concentrates on these kinds of methods and their advantages and disadvantages.

[201] introduces a parallel architecture (Gorila) that uses several agents with

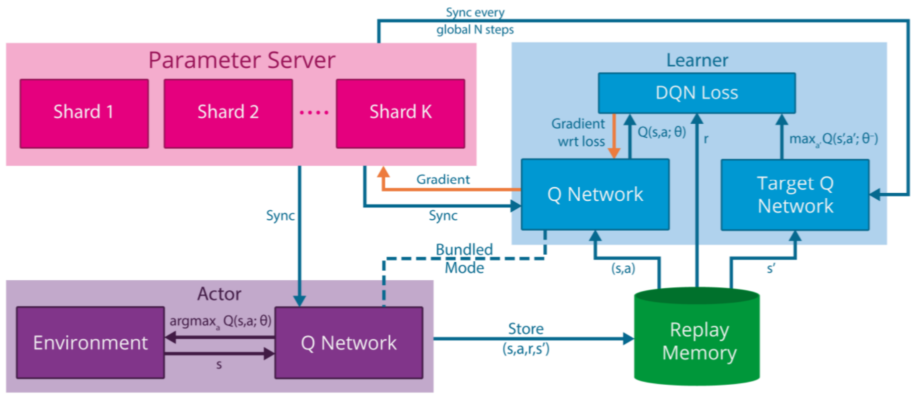


Figure 2.1: Image taken from [201] showing the architecture of the Gorila framework for distributed deep reinforcement learning.

their own experience replay buffers. The learning algorithm, instead of sampling from a single replay buffer, samples from the many different replay buffers generated by the individual agent's experience. This experience is then used to generate gradient updates to a global parameter store. The global parameter store contains the parameters for a Q-function which periodically synchronizes with the Q-functions being used by the individual agents. Figure 2.1 shows a diagram of this architecture.

[198] introduces a distributed version of the A2C algorithm, Asynchronous Advantage Actor Critic (A3C). Instead of using a shared experience replay buffer and many physical machines to produce experience data, A3C generates experience using several threads on the same machine, which reduces the latency of updates over a network. Additionally, as there is no replay buffer, different actors can generate different experiences using different policies as they evolve over time. These policies can periodically update a shared target network which is then distributed to all actors. The authors show empirical evidence that A3C scales linearly in computation time with respect to the number of actors that are producing experience.

IMPALA [86] improves upon parallel methods by updating a global policy network using GPU acceleration with data from multiple actors across many machines. Experience, policy distributions, and initial states are generated by asynchronous learners and are passed to a global queue mechanism which does large batched updates of the global policy. This global policy is then shared with the asynchronous actors to generate new experience information. One of the issues with this asynchronous training method is that asynchronous actors generating the experience and policy updates are always added to the end of the queue, meaning these updates are from old policies, which may conflict with the much newer global policy. In order to alleviate this issue, [86] introduces the V-trace algorithm, which uses importance sampling to correct off-policy updates.

Prioritized experience replay is extended to use a distributed architecture in [133]. The distributed architecture uses many actors to push experience into a global prioritized experience replay buffer which filters out states where the TD error is low.

Using LSTMs to help store information across many frames in reinforcement learning allows hidden states to be remembered or inferred in partially observable Markov decision processes [23] and time series data [96]. LSTMs can also provide large improvements in training in fully observable environments. Recurrent Replay Distributed DQN (R2D2) [165] uses the idea from Ape-X [133] to use an experience replay buffer, but instead of storing (state, action, reward, next state) tuples, stores sequences of (state, action, reward), which can then be trained as sequences using Recurrent Neural Networks (in this case LSTMs). R2D2 achieved much higher scores than both IMPALA and Ape-X on many Atari games and superhuman performance on 52 out of 57 of the Atari benchmark suite.

Using many CPUs, GPUs, and networking hardware can become extremely expensive and efforts have been made to make the process more efficient and cheaper. [87] Combines R2D2 and V-trace from [86] in an algorithm called Scalable Efficient Deep RL (SEED) which can train reinforcement learning agents with millions of frames per second, with significantly reduced costs.

2.4 Auxillary Losses

In [180], Deep Recurrent Q Networks [125] are used to play the 3D game Doom to a super-human ability. During training, the agent uses pixel information and additional game features such as the presence of certain items within the visual frame. The agent's network is trained to predict game features in each frame in addition to predicting the value function Q. A similar method named *Auxiliary Tasks* is used in [143] to navigate 3D mazes, the *Auxiliary Task* in this case is to learn the distance from the player to the wall ahead of it. [256] also uses several auxiliary functions, such as predicting the actions based on the current and next state (Also known as inverse dynamics), reconstructing the next state using auto-encoders, and predicting whether rewards will be positive, negative or 0. These methods were used in pre-training an RL agent and sped up the subsequent learning of the agent. In many cases forcing neural networks to predict underlying semantic information as part of training dramatically increases training speed and agent performance.

[98] shows that learning an accurate model of the game itself can drastically improve reinforcement learning on several Atari 2600 games.

2.5 Multi-Task Learning

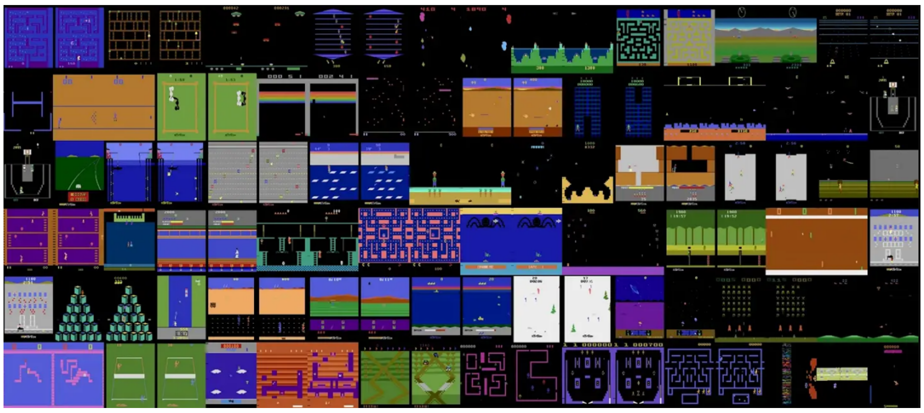


Figure 2.2: Image taken from [20] showing the 57 atari games that were solved using the combination of multi-agent learning, exploration, and meta-learning.

Deep Reinforcement Learning has been able to learn a single task or play single games with state-of-the-art performance. The results achieved are certainly impressive, but unfortunately, the methods used may not be suitable for scaling to other games or tasks. This is because reinforcement learning is trained on a specific state distribution and reward system, which means that the learned policy may not perform well on a state distribution that was not covered during the training phase. Additionally, it may not perform well on tasks defined by different reward schemes.

One method of addressing multi-task learning is to use universal value function approximators (UVFA) [244]. This method includes an extra variable in the value function Q which encodes the current task to be solved.

Normal value function for state and action (s, a):

$$Q(s, a) \tag{2.7}$$

Value function with state, action, and task identifier $\mathbf{z}$

$$Q(s, a, \mathbf{z}) \tag{2.8}$$

The value function in this case assumes that the environments and their structure are similar, but the tasks have different reward functions. This allows a single-value function to be trained on multiple tasks.

An improvement to this method combines universal function approximators with successor features [41].

In [31] uses successor features to train several tasks and then uses the successor features in linear combinations to generalize to new unseen tasks. This algorithm allows agents to infer policies of new tasks from tasks that have previously been learned without having to learn new tasks from scratch. The successor features for this new policy can then be used to generate new policies in future tasks that are unlike those previously seen. The authors argue that this has implications for life-long learning applications.

Using universal value function approximators, [21] learns several exploration value functions $Q(s, a, \beta)$ that give the agent varying degrees of intrinsic reward. The reward of the agent at a given time-step t is given by $r_t^{\beta_i} = r_t^e + \beta_i r_t^i$ where r_t^e and r_t^i are the extrinsic and intrinsic rewards at a given time step. The learned value functions for these rewards are parameterized over the β^i values. The number of beta values that are used is a hyperparameter N. In harder exploration problems, larger values of N were beneficial, giving a more complex exploration policy, but this was a detriment in simple games.

The same universal function approximators approach is used in [20], which achieves much higher scores on the same ALE games as [21] as well as being able the first algorithm to beat the human-level benchmark on all 57 ALE games. [20] claims that having a single Q-function for all values of β is difficult to learn as the different β values result in different scaling and sparsity of rewards. To alleviate this issue, two improvements are made, firstly, the discount γ factor is added as a parameterization of the Q-values and secondly, separate Q-functions are learned for exploration and exploitation. This gives an update Q-function as $Q(s, a, j) = Q(s, a, j; \theta^e) + \beta_j Q(s, a, j; \theta^i)$ where j is a one-hot vector representing which combination of β_j, γ_j is used. [184] uses a similar approach of decoupling the exploration reward and environment reward into separate policies with separate Q functions.

2.6 Transfer Learning

Transfer learning is another field of research in deep neural networks that deal with how to generalize learning between similar environments or use the knowledge learned in one environment to understand others in order to learn faster.

In the domain of reinforcement learning, one of the most promising areas of research is how to train agents in simulations in a way that can then be embodied in a physical machine and still retain high performance. It is extremely difficult to transfer from simulations to real life because there are so many simplifications in simulated environments that are unrealistic in physical worlds. For example, when controlling robot arms, in a simulation, the sensors that give state information to the agent will give perfectly accurate values. However, in the real world, the sensor information is subject to electrical noise, slew rates, inaccuracies, and drifting. This means the data from a physical robot will almost certainly not fall within the distribution of the training data given in the simulation. The agent will then struggle to perform even basic tasks.

One method of solving this issue is to train neural networks on the data read from the physical system itself [88], [289]. However, due to the inefficiency of deep neural networks, these methods are extremely slow to train, making the method impractical, as the training would take years for the neural networks to converge.

In [282], a method of training under uncertainty in simulated environments is introduced. Domain Randomization randomizes the parameters of the simulated environment observations in a way that forces the neural networks to ignore many distractions, such as visual noise, lightning conditions, and colors. [214] expands on this by extending the randomization to sensor readings in a robot hand. This allowed a robot hand trained in simulation to manipulate a cube block into different orientations to transfer to a physical robot hand.

Instead of training a control algorithm under a large amount of randomized data, [145] trains a model to first convert the randomized data into a canonical representation of the environment. This canonical representation is then used as the input to the agent which controls the environment, in this case, a robot arm. This method draws parallels with learning accurate latent state space models of complex environments and then using this state as the observation in reinforcement learning.

2.7 Imitation Learning

Imitation learning is the process in which an agent learns to perform tasks in an environment by attempting to mimic the actions of another agent it is observing. It is prevalent in nature, for example, when a child imitates their parents by trying to walk or trying to say words. Therefore, it is a natural research direction to teach artificial agents to imitate an example agent's actions in other environments. In many cases, the example data comes from humans, for example, in [255], [14], [293]. "Expert trajectories," which in most cases refers to the full state of the environment, including the data from all the sensors, are used to effectively seed the initial policy of the network so that the agent is biased to finding particular rewards or reacting to particular situations in certain ways. This avoids the initial stages of inefficient random exploration that the agent would otherwise follow.

Natural environments do not provide all of the information required to copy a policy from one agent to another. Suppose one agent tries copying another "demonstrator" by observing its actions. In that case, the only information the agent receives is visual, and it does not have access to any sensor information or noise data that the demonstrator is experiencing. In many cases, the agent may not have the same physical or embodied characteristics as the demonstrator whom it is trying to imitate. This problem is referred to as a *Domain Gap*, and there are many potential solutions to this problem [170], [255], [14], [57].

[255] and [14] Both use Time Contrastive Networks (TCN) in order to bridge the domain gap between multiple different videos of human demonstration. In the former, a human demonstrates tasks such as pouring liquid into a cup to be replicated by a robot arm. In the latter, an Atari game-playing agent attempts to imitate playing several games from YouTube videos. The YouTube videos in [14] are of several hard exploration games that are difficult to solve using standard exploration techniques as rewards are sparse and tend to require specific sequences of actions to realize. The main domain gap issue with learning games from YouTube is that in the process of uploading and encoding, the users may be using different encoding algorithms, aspect ratios, and in some cases, adding image overlays or transitions to the videos. TCNs are used alongside Cycle consistency algorithms to map the different videos into the same latent space which is then combined with an imitation reward function. The imitation reward function rewards the agent when the agent's actions produce observations that match those in the demonstrations.

In DeepMimic [220], task-based objectives are combined with imitation objectives to create a realistic movement of many humanoid, animal, and robot models. DeepMimic can generalize across similar tasks, such as jumping to and landing from different heights, having only seen examples of jumping across a level platform. Complex physical motions such as Spinkicks, cartwheels, and backflips can also be learned and are nearly indistinguishable from the appearance from the reference motion.

It is also useful for agents to learn policies from other agents. In this case, a technique known as policy distillation [69] [236] can be used.

2.8 Unsupervised Environment Design

Like any algorithm using deep learning, deep RL requires large amounts of data and is sensitive to overtraining. That is to say that if the agent is exposed to the same environment over many training trajectories, then the same set of states will always be encountered, thus leading to a policy that is trained specifically for that environment. In the game setting, training an agent to perform well in a single level will likely impede its ability to perform well or generalize to levels it has not seen. This problem is referred to as the *generalization gap* [61].

One of the most promising methods for agents to learn high-performing policies that still have high performance on un-seen levels is to use Procedural Content Generation (PGC) methods to create a much larger variety of experiences for the agent during training, with the hope that this experience is transferable to unseen scenarios.

Procedural content generation (PCG) refers to automatically creating environment assets or configurations such as textures, maps, audio, and even NPC (Non-player character) behavior.

PCG methods have existed mainly in the gaming industry to generate novel experiences for human players rather than for generating data for artificial intelligence training. Thus, many directions of PCG research focus on human experience [275, 268] such as controlling diversity, player enjoyment, and aesthetic perception [126]. In the RL setting, PCG can be used as a component of what is known as Unsupervised Environment Design (UED) [74].

In UED methods, the MDP introduced in section 2 has a slightly different parameterization, where the initial distribution ρ is replaced by an environment parameterization Θ which allows modification of the environment during trajectories $\mathcal{M} = (S, A, O, \Omega, \mathcal{T}, R, \gamma, \Theta)$. An example of this would be in domain randomization [214] where the environment parameters such as friction and gravity are modified at each time-step in order to improve sim-to-real transfer to a physical robot hand.

In game environments, it is much more likely that the environment parameterization is only used to modify the initial state of the MDP, which in practical terms, means that the level configurations are procedurally generated per episode. The generated levels depend on the particular algorithm being used in UED. In one of the simplest cases, the environments are generated using heuristic algorithms that rely on random number generation. In this case, the parametrization of the environment generator can be viewed as the parameterization. This is the case in environments such as ProcGen [61], MiniGrid [54], and Crafter [117].

Using these randomly generated levels can lead to better generalization across unseen levels. For example, in [281], the wave function collapse algorithm [111] is used to generate diverse sets of levels with various different goals. However, randomly generating levels and using them to train agents can lead to poor performance. Instead, in [281], a "dynamic task generation" method is introduced, where levels are only added to the training curriculum if they meet certain criteria. These criteria are based on the performance of a "control agent," which measures whether or not the levels are too simple or too complicated.

This issue is also addressed in [148], where a curriculum is automatically generated by scoring levels that have high learning potential and repeatedly exposing these levels to the agent and revising scores regularly. This method automatically generates a level curriculum, allowing the agent to learn more efficiently and with more effective policies that generalize better to unseen levels. This method is expanded in [216] where, instead of just using and scoring randomly selected levels, the levels are also modified using evolutionary algorithms. For example, in maze levels, walls can be added or removed. This work again improves learning efficiency and generalization across many environments.

Evolutionary algorithms are used as the primary method used in [295] to generate an evolving set of training environments alongside an evolving population of agents. This co-evolution strategy generates new sets of problems as the agents evolve to solve the existing problems. Like previously mentioned methods, this also naturally generates a curriculum of levels that allows the agents to learn in a more principled manner.

In [74], the environment parameterization controls a reinforcement learning policy that places environment objects. This policy is trained to maximize the regret between a protagonist and an antagonist agent. The *protaganist* agent is trained to minimize the regret, and the *antagonist* agent is trained to maximize the regret (similarly to the policy that generates the environment). This method, similar to [216], creates a natural curriculum whereas the protagonist learns to solve environments by minimizing regret, the antagonist and adversary policies have to generate harder levels in order to maximize it.

In other literature, the method of using an RL policy to generate levels is known as Procedural Content Generation Via Reinforcement Learning (PCGRL) [169]. PCGRL typically splits the problem of environment generation into three components, the *problem module*, the *the representation module*, and finally, the *Change Percentage*. The problem module defines the components of the environment generation, such as objects, goals, and reward functions. The representation module defines how the RL agent perceives the environment generation process, for example, the state representation of an environment under construction, and how the actions can modify this representation. The problem module contains rules for defining which state transitions in the representation module result in rewards. For example, if an action results in the addition of a goal object that is required for a valid level, the problem module calculates a positive reward for this action. Finally the change percentage is a hyperparameter which controls how many parameters of the environment can be modified per iteration. In 2d Grid worlds, this percentage represents the number of tiles that can be changed as a percentage of the total tiles in the environment.

In the implementation of PCGRL [169], the rules in the "problem module" have to be manually specified. For example, in the Sokoban problem module, a simple Sokoban solver has to be included to validate that the generated levels are not impossible. This can be problematic as proving that any environment or game has a solution is an open problem. In several other works, this problem of generating solvable levels is also encountered [314, 265, 81].

In section 3, we introduce the Griddly framework, which contains many features to support UED and PCGRL. For example, Griddly environments can be configured that allow RL agents to design an environment itself. This environment can then be exported and played by another agent.

2.9 Alternate methods

In this section, we will discuss alternative techniques that do not rely on deep neural network models to train policies or understand the dynamics of environments for model-based reinforcement learning. The methods here use either heuristic methods or evolutionary computation algorithms.

For example, in [114] the game engine of Mario is learned by learning how to map the set of given sprite images to each frame and then learning a set of heuristic rules as a sprite transition function. This method works very well, but the mechanics of the game and the features, in this case, sprites, are specifically engineered for this single game.

[75] and [76] also learn rule-based hierarchical knowledge bases that learn very fast and can adapt to unseen states quickly.

[222] introduces Rolling Horizon Evolutionary algorithms, which, similarly to MCTS require a model of the environment and perform rollouts to find the best actions. [94] Shows that in some environments when the population size of the evolutionary algorithm underlying is larger, it can outperform MCTS. [95] improves on these results by using another genetic algorithm N-Tuple Armed Bandit Evolutionary Algorithm (NTBEA), to optimize the large hyperparameter space of RHEA.

Particularly interesting areas of research are those that use neural networks but combine these with evolutionary strategies.[202] uses randomly initialized neural networks and an evolutionary strategy that, instead of using backpropagation, learns the parameters of Hebbian learning rules. This allows each connection between neurons to be configurable in terms of their learning gradient and learning rules.

With this broad context of reinforcement learning, we can now focus on the research questions in section 1.

Chapter 3

Griddly

In the research setting, it is common to use video games as the primary substrate in which to perform experiments. Arguably without video games, many of the recent advances in reinforcement learning would not have been possible. In this chapter, we cover the challenges of building simulation environments specifically for reinforcement learning. As new architectures, methods, and representations are developed in deep learning, the way that agents interact and perceive environments may have to change to accommodate these advances. For example, with the introduction of transformer models and object-centric methods, the ability to enumerate objects and their associated features is required, but many game environments do not support this. Many reinforcement-learning specific environments have been created, many based on real environments, or wrap existing games such as Starcraft. Unfortunately, many of these environments suffer from slow execution speed and large memory overhead, making research progress only available to organizations with large experimental resources. In this section, we first delve into current research methods and approaches aimed at creating efficient and accessible environments to enhance research productivity. We then introduce Griddly [29] as a software package that aims to provide rich functionality for many research directions, without sacrificing features or efficiency.

3.1 Background

Deep Reinforcement Learning typically requires large amounts of experience data to train expressive and general policies. However, collecting this data from physical sensors on an embodied agent such as a humanoid robot or a self-driving car is particularly inefficient. Additionally, this data will have a significant amount of noise as a result of the physical dynamics of the real

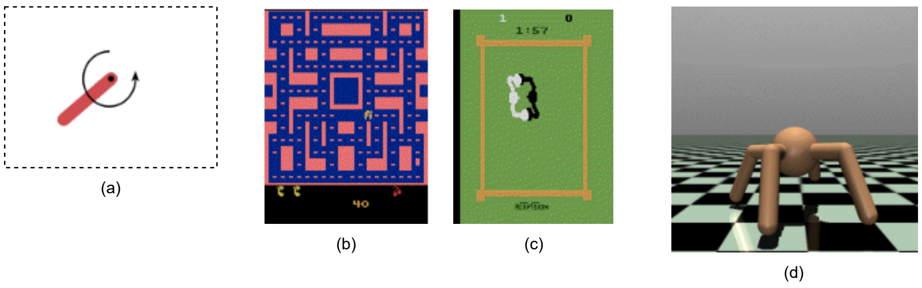


Figure 3.1: Some of the most popular reinforcement learning simulated environments. a) the "pendulum" classical control environment. b) and c) are the "Ms. Pac-Man" and "Boxing" environments from the Arcade Learning Environment (ALE). Finally d) is the "ant" environment from MuJoCo.

world, such as friction, temperature, and vibration. Only very recent research has shown that it is possible to learn policies for physical robots [305, 267] within a reasonable amount of time. These methods, although impressive, are limited to learning robust motor control over different terrains, and do not involve tasks that require complex reasoning or planning.

It is for this reason that the majority of Reinforcement Learning environments are simulated, that is, they exist in software. This has many advantages, such as allowing the environments to be easily configured and controlled. Simulated environments also have large advantages over physical environments for research: most environments are free and open source; do not require specialized hardware; are not limited by physical time constraints (i.e can be trained at speeds much higher than required in inference) and are far safer as they do not risk physical injury.

Using simulated environments also allows the concept of embodiment to transcend physical restrictions. In simulated environments, we can give agents as little or as much information about the environment as possible and also define the form and structure of that information. We can also completely control how the agent interacts with the environment: a simulation could be interacted with by setting forces on simulated actuators, or interactions could be far more abstract such as "find the shortest path to a destination", where an inbuilt heuristic algorithm is initiated by a simple decision.

In this section, we explore the different types of simulation environments that have been proposed for reinforcement learning research and describe the methods and interfaces used to embody these agents.

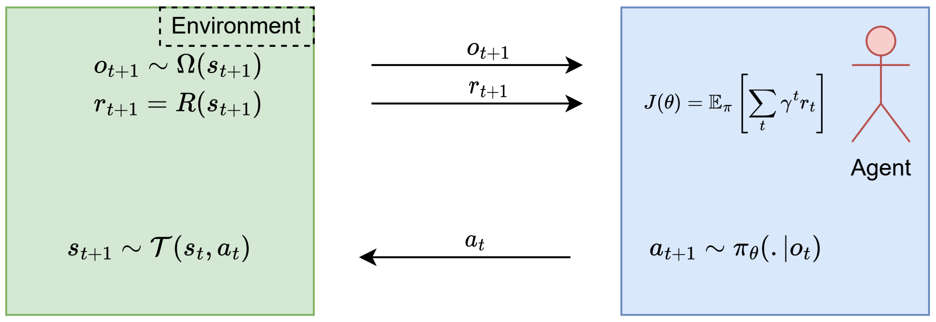

Environment
$o_{t+1} \sim \Omega(s_{t+1})$
$r_{t+1} = R(s_{t+1})$
$s_{t+1} \sim \mathcal{T}(s_t, a_t)$
o_{t+1}
r_{t+1}
a_t
$J(\theta) = \mathbb{E}_\pi\left[\sum_t \gamma^t r_t\right]$
Agent
$a_{t+1} \sim \pi_\theta(.\,|o_t)$

Figure 3.2: A diagram of the perception-action loop used to describe the way reinforcement learning is modeled as a Markov Decision Process. The agent produces actions a_t in response to observations o_t. The environment then produces the next state s_{t+1} using the transition function $\mathcal{T}$. Observations The next observation o_{t+1} is determined by an observation function Ω.

3.1.1 Perception-Action Loop

In section 2, we introduced the Markov Decision Process, as how reinforcement learning is modeled, and in this section, we will concentrate on particular components of the MDP which relate to the observations $o_i \in O$, rewards $r_t = R(s_t)$ and actions $a_i \in A$ within the perception-action loop. These components are shown in figure 3.2.

3.1.2 Observation Spaces

We review the many different forms that observations can take in reinforcement learning environments. More formally, we explore the different possible configurations of observation functions Ω that have been used in recent research and their relative merits.

Observations of environments refer to the way that an agent senses an environment. This could be using proprioceptive sensors, vision, audio, or external memory.

Some of the most well-known environments introduced in [45] are the "classical control environments". These environments typically consist of simple toy problems such as balancing a pole on a moving cart, pushing a car up a mountain, or balancing a pendulum vertically. All of these environments use observation spaces that can be described as proprioceptive, for example, the pendulum environment's observation space consists of 3 variables; the x and y position of the end of the pendulum, and the angular velocity of the pendulum. The pendulum environment can be seen in Figure 3.1.

Other popular environments, such as the Arcade Learning Environment [37] expose only the image of the screen. The image is encoded using RGB with a fixed width and height, $O \in \mathbb{R}^{210\times160\times3}$. In many experiments, however, this image is cropped, and down-scaled [196]. It has also been shown that using the Random Access Memory of the internal game state can be used as an observation space [8].

In some cases, the observation space of environments is configurable, meaning that the environment provides different observation spaces depending on the application. The MuJoCo environment [284] (shown in Figure 3.1 for example, can be trained directly from pixels [307] or using proprioceptive information such as angular velocities and positions of joints [80].

Aside from physical or visual observation spaces, there are several simulation environments that use natural language as the observation space [146, 70]. In these environments, the agent must learn to understand the environment via textual descriptions and perform actions based on these alone. These works commonly use transformer neural networks, which are the most popular method of language understanding. [300, 290].

In many game environments, observation spaces are split into several different components. For example, there may be visual components, components that represent maps and text representations in the same observation space. The observation space of the NetHack Learning environment [177] consists of a 2D description of what the agent can currently see, a vector of agent statistics, a text-based message input for messages from the game, and a padded tensor consisting of the inventory of the agent. Similarly in some mini-grid environments [54], textual messages such as describing environment goals are given which can be interpreted using language models.

Another particularly successful observation space related to textual observation spaces is that of *entity observation spaces*. In this thesis, we refer to any observation space that consists of a set of unordered feature vectors, which correspond to various environment components as *entity observation spaces*. A simple example of an entity observation space would simply be a set of vectors $e \in E$, where each individual vector corresponds to one object in the environment. Each individual vector would only contain an x,y coordinate, and a one-hot encoding of the object type. In more complicated environments, there can be several distinct sets of entity types, where different objects have different features, or, for example, there may be a set of entities that correspond to the inventory of the player. These entity features are used in environments such as NeuralMMO [273], StarCraft [293], and Dota [213]. Additionally, open-source frameworks exist which are designed to train agents with these kinds of observation spaces [303].

These entity observations are typically embedded into tokens and observed using neural networks such as transformers, in a similar way to language models

[33, 315, 303], this allows the neural network to learn relationships between the objects and attend to important factors in the observation space.

In this thesis, we concentrate mainly on grid-world environments, where observation states are discrete in nature as environmental objects have discrete positions. In many cases, pixel-based observations using sprites can be used as an observation space [54, 117, 35, 80, 221, 29], however as these environments generally have a distinct number of objects and positions, *Vectorized* observations may be used. We use the term *Vectorized* observations to refer to observations which are of the form $O \in R^{H \times W \times C}$ where H and W refer to the height and width and C refers to the number of *channels* at each x,y location. We use *channels* to refer to the information which is stored at each location. The scalars in each *channel* refer to features of the environment, such as which objects are present and the variables that might be associated with those objects. The configuration of the channels is usually defined by the particular environment. A simple example of a common configuration of channels is to use a one-hot encoding scheme to encode the locations of different object types. More complex environments can allow objects to occupy the same location, and in some cases, these objects might have observable variables such as health. In multi-agent games or games with multiple controllable units, the channels might also contain information on which agent or "team" the objects belong to [208, 137, 139, 157].

It is also common practice to embed pixel-based observations into a similar space by using convolutions that have a kernel size the same dimensions as tiles, and associated padding. This has the effect of creating an observation space similar to *Vectorized* observations. However, the embedding only contains information that is visible in the pixel observations, which in most cases is just the object type. Directly converting from pixels to an embedding can also have a negative effect on generalization if, for example, the image changes slightly due to effects unseen during training [271].

Partial Observability

Partial observability in an MDP refers to the implementation of an observation function Ω, which only includes a subset of the full state of the environment, which can be subject to additional transformations. In many game environments, this partial observability is inherent to the environment. For example, 3d worlds only show the camera view of the agent, so there are occluding walls, objects, and other features such as fog. In simple grid-world games, full observability is common for example in games like Sokoban or Zelda [222].

In more complicated, or larger grid world environments, it is common to use an *egocentric* partial view of the environment, for example, a 5x5 grid around the agent itself. This allows the agent to exist within a large complex environment, but have a consistent and canonical observation space. Consistent vector and pixel observation spaces are also much more suited to neural networks as they require fixed numbers of parameters to learn policies.

In terms of generalization, having a fixed representation also limits the size of the number of parameters required to learn policies, and thus can lead to policies that are much more successful, and generalize better to unseen level configurations [311].

As Reinforcement Learning research studies both fully observable and partially observable environments, many environments provide inherent configuration or tooling to customize the observation space of the environments. In-built tooling is far more efficient than post-processing, as only the specified observation is generated, rather than a complex observation which is then filtered by an additional process such as image manipulations or transformations [29].

Auxiliary Information

One advantage of simulated environments such as games is that additional high-level information can be shared from environments that would not typically be part of an observation space.

One example of this type of information is that of *Invalid Action Masking.*

Invalid action masking (IAM) [138] is a technique used to stop agents from sampling actions that are invalid in a particular game state. IAM is useful in environments where the action space is large, and some of the actions are only available in certain states. For example, in RTS games [293, 292, 240], the agent's action may consist of selecting a unit or units from a large list and then issuing commands to those units. The commands sent to those units can also be unique to particular unit types. This results in a large number of options in the action space that are invalid. In policy gradient and actor-critic methods in deep RL, IAM is applied to the output logits $\mathbf{l} \in \mathbb{R}^n$, of the policy produced by a neural network, by replacing the logits corresponding to invalid actions with large negative numbers. This forces the probability of selecting those actions to tend toward 0.

For instance, let us assume a compound policy constituted by K independent components, such that $\pi(a|s) = \prod_k^K \pi(c_k|s)$. This type of action policy could be described by:

$$\pi(a|s) = [\pi(c_0|s), \pi(c_1|s), \ldots, \pi(c_k|s)] \tag{3.1}$$

For each of the components in equation 3.1, a value is selected following a softmax sub-policy. We can create a mask to modify the logits to assign large negative numbers to actions deemed as non-viable or inaccessible. The modified logits result in $\hat{\mathbf{l}} = \mathbf{l} + \mathbf{m}$ where $-\infty < m_i \ll 0$. It then follows that the *masked* logits alter the probability of a value of c_i of being sampled:

$$\pi(c_i|s) = \begin{cases} 0 & \text{if } m_i \longrightarrow -\infty \\ \frac{e^{l_i}}{\sum_j^N e^{l_j}} & \text{if } m_i \quad = \quad 0 \end{cases}$$

In PySC2 [292], μRTS [209] and BotBowl [157] action masks can be constructed from lists of available actions that are provided by the environment implementations. However, these action masks do not consider that the masking of some sub-actions can depend on the sampled values of others. For example, in an environment with units selected by coordinates and the set of available actions for each unit is disjoint, the mask for the available actions depends on the unit's selection. Masks that are naively constructed using these lists can still lead to select actions that are not available, as the list does not consider the unit's selection. [139] introduces a two-step method for generating masks where the unit location is selected using masked logits and then a second mask is generated based on that selection. This significantly improves training as the mask for unit actions depends on the selected unit.

3.1.3 Action Spaces

In this section, we review several of the most common methods that an agent can affect the environment through their *action space.* In an environment modeled by an MDP (shown in figure 3.2), at each time step, an action $a_t \in A$ to perform is chosen by the agent. This next state s_{t+1} is then returned from the transition function $\mathcal{T}$, which takes the current state s_t and the chosen action a_t as inputs. The set of possible actions A an agent can take in an environment is known as the *action space.*

Action spaces can be structured in several ways, but most commonly, they take the form of a 1 Dimensional space $A \in \mathbb{R}^n$ where n is the number of possible action components. The choice of action can be discrete or continuous, depending on the environment. For example, in Mujoco environments (d) and Classic Control (a) in Figure 3.1, the actions set the torque values of the joints, and the torque values can be set within specific ranges. In this case $A = (a_0, ...a_n)$ where n is the number of joints to control, and $a_n \in [a_{min}, a_{max}]$

In Atari Environments (b and c) in Figure 3.1, the actions are discrete, as in the agent will select a distinct action such as "up," "down," "left," or "right." This translates to a single-valued action space $A = (a)$ where a is an integer value representing the index of an action to take $a \in [0..3]$.

We cover several action space configurations in detail in section 4.4.2

3.1.4 Rewards

A final part of the perception-action loop that we will cover is that of the reward. The reward given to an agent in any environment is the signal that the agent uses to learn any given task. Rewards are usually configured as part of the environment itself and are usually calculated as a function of the state of the environment $r = R(s_t)$. In Reinforcement Learning, described in section 2, the goal is to maximize the expected return, which is calculated as the sum over the discounted rewards $J(\theta) = \mathbb{E}_\pi \left[\sum_t \gamma^t r_t\right]$. It is commonly argued that this method of maximizing expected return is adequate even in complex environments to produce complex behaviors and general intelligence [261].

In many environments, the rewards configured by the environments themselves are very simplistic, for example, only giving a single reward when a goal is reached [54]. More complex environments add rewards for completing sub-tasks or tracking homeostatic variables such as health levels [117]. In many experiments, the reward function is modified to attempt to provide the agent with more information. This is especially useful in environments with sparse rewards [321, 108].

Another common modification to rewards is to provide intrinsic rewards rather than extrinsic (provided by the environment) rewards. Intrinsic rewards are those that are generated by the agent itself, usually generated by a particular algorithm that looks for novelty or high information gain [224, 217].

3.1.5 Engineering Considerations

A large factor in research success using simulated environments is the iteration speed of experiments [310]. Reinforcement learning usually requires hyperparameter searches to find the most optimal set of learning rates, batch sizes, and other parameters. It is not uncommon to find state-of-the-art results by just performing wide hyperparameter searches [271]. One of the largest bottlenecks in reinforcement learning, however, is the execution speed of the environment. Using high-fidelity game environments such as StarCraft, Dota and Minecraft [164, 113] can require large amounts of CPU and GPU resources to train neural networks in a reasonable amount of time. But faster training comes at a huge cost. These types of environments, as they are not designed with reinforcement learning in mind, lack optimizations that would make training significantly more efficient.

Hardware Accelleration

More recently, complex environments have been designed specifically with parallelism and sample efficiency as design requirements. These environments commonly take advantage of GPU acceleration or multi-threading capabilities to parallelize the computation of environment dynamics or rendering.

In [92] and [188], the computation of the mechanics of rigid body dynamics is offloaded entirely to the GPU. This means that several thousand agents can be run in parallel on a single piece of hardware. Additionally, the overhead of transferring memory from the CPU to the GPU can be avoided as the calculation of neural network gradients is also located on the GPU. Similarly, in [72], the Atari emulator is ported to the GPU which has similar benefits.

Another bottleneck that is common to reinforcement learning environments is due to the fact that the most common language used for experimentation, python, has no support for threading. Instead of parallelizing environments using efficient multi-threading methods, the most common method for running many environments in parallel is to create multiple processes, which then communicate through Remote Procedure Calls (RPC), which carry significant overhead. In some environments, typically those written in C++, this can be avoided by handling the parallelism natively, so only one Python process is required [61, 301]. Another method is to create a small number of Python processes (typically one per CPU) but then host multiple environment instantiations in each process [182].

Configurability

Many reinforcement learning environments are specific to particular tasks. There are many common parts of reinforcement learning environments that can, however, be re-used, for example, rendering and interfacing code for RL algorithms. In order to attempt to create a more general approach to generating environments to either build simple platforms such as GVGAI [222], use existing games [242, 113] or game engines such as Unity [153, 152].

GVGAI contains a simple scripting language that allows grid-world games to be built. However, the rendering and parallelization capabilities of GVGAI are not optimized for reinforcement learning, making it unsuitable for this particular use. A limited scripting language is also used in the NetHack Learning Environment [177, 242], which allows NetHack style levels to be generated, this scripting language gives access to hundreds of in-game items, NPCs and various configurations of behaviors. The NetHack Learning Environment is particularly complex in nature and is fairly well optimized, making it a very flexible choice for RL research.

Game Engines such as Unity, although offering high levels of flexibility and ease of use, RL interfaces are not designed to be optimal for RL research and commonly suffer from slower execution speed and high memory usage. Additionally, the flexibility of the environment allows researchers to write poor-performing code, which again can cause slow execution.

Some specifically designed engines for research have also been created, such as DMLab [36] and DMLab2D [35]. These sit between the full-featured game engine and the scripting approach. Both of these projects enable the creation of games using the Lua language. Lua is quite flexible, but researchers still need to invest time and effort to engineer new games. Additionally, these approaches are CPU bound and do not use GPU acceleration to parallelize game computation. However, they still achieve high sample rates, making them a popular choice for research.

In Chapter 3, we introduce the Griddly framework, which combines many of the above approaches into a state-of-the-art combination of flexibility and efficiency. We also show that the architecture of Griddly allows it to be integrated with web-based interfaces, which adds to its rich feature set. This integration with web technology makes Griddly a useful tool for sharing and disseminating research.

3.2 The Case for Grid Worlds

Grid worlds are environments corresponding to MDPs with discrete actions and states that can be represented as a 3D tensor. Note that while the state is constrained to be a tensor with dimensions $M \times N \times K$, where $M, N, K \in \mathbb{Z}^+$, the actual observations seen by the agent may be rendered differently, e.g. in pixels or as a partial observation. Typically, in 2D grid worlds, each position in the grid, or *tile*, corresponds to an entity, e.g. the main agent, a door, a wall, or an enemy unit. The entity type is then encoded according to a vector in R^K. By constraining the state and action space to simpler, discrete representations, grid worlds drastically cut down the computational cost of training RL agents without sacrificing the ability to study the core challenges of RL, such as exploration, planning, generalization, and multi-agent interactions.

Indeed, many of the most challenging RL environments, largely unsolved by even the latest state-of-the-art methods, are PCG grid worlds. For example, Box-World [315] and RTFM [319] are difficult grid worlds that require agents to perform compositional generalization; many games of strategy requiring efficient planning such as Go, Chess, and Shogi may be formulated as grid worlds; and many popular exploration benchmarks, such as MiniHack [242] and MiniGrid [54] take the form of grid worlds. A particularly notable grid world is the

NetHack Learning Environment [NLE; 177], on which symbolic bots currently still outperform state-of-the-art deep RL agents [122]. NLE pushes existing RL methods to their limits, requiring the ability to solve hard exploration tasks and strong systematic generalization, all in a partially-observable MDP with extremely long episode lengths. Crafter is a recent grid world that features an open-world environment in which the agent must learn dozens of skills to survive, and where the strongest model-based RL methods are not yet able to match human performance [117]. To demonstrate the potential of GriddlyJS, we use it to create a Griddly-based Crafter in Section 3.12.

Their common grid structure and discrete action space allows for grid worlds to be effectively parameterized in a generic specification. GriddlyJS takes advantage of such a specification to enable the mass production of diverse PCG grid worlds encompassing arbitrary game mechanics. Thus, while GriddlyJS is limited to grid worlds, we do not see this as significantly limiting the range of fundamental research that it can help enable. Still, it is important to acknowlege that grid worlds can not provide an appropriate environment for all RL research. In particular, grid worlds cannot directly represent MDPs featuring continuous state spaces, including many environments used in robotics research such as MuJoCo [283] and DeepMind Control Suite [280]. Nevertheless, as we previously argue, grid worlds capture the fundamental challenges of RL, making them an ideal testbed for benchmarking new algorithmic advances. Further, many application domains can directly be modeled as grid worlds or quantized as such, e.g. many spatial navigation problems, video games like NetHack [177], combinatorial optimization problems like chip design [194], and generally any MDP with discrete state and action spaces.

3.3 The Griddly Engine

Griddly[1] [29] is a game engine designed for the fast and flexible creation of grid-world environments for RL, with support for both single and multi-agent environments. Griddly simplifies the implementation of environments with complex game mechanics, greatly improving research productivity. It allows the underlying MDP to be defined in terms of simple declarative rules, using a domain-specific language (DSL) based on Yet Another Markup Language (YAML). This is a similar approach to GVGAI [221], MiniHack [242] and Scenic4RL [15], where a DSL language is used to define environment mechanics. Griddly's DSL is designed to be low level in terms of interactions between defined objects, but does not go as far to allow the user to define physical models unlike Scenic4RL. This

[1] Documentation, tutorials, examples, and API reference can be found on the Griddly documentation website `https://griddly.readthedocs.io`, This can also be found in the Appendix

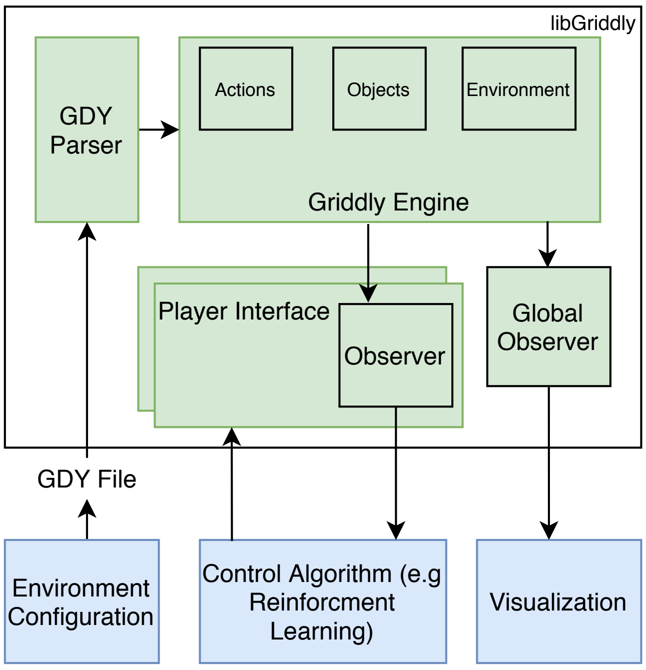


Figure 3.3: A high-level diagram of main components of the Griddly architecture showing the separation of the main components. Multiple player interfaces with configured observers can be attached to the Griddly engine to control any number of players in the environment. Additionally, a global observer can be configured to monitor the environment as a whole or analyze the performance of any algorithms from a global perspective. The environment configuration is decoupled from observation and agent control, meaning that generative algorithms for creating different environment layouts can be used.

is similar in regards to GVGAI and MiniHack as they are both grid-world games. Griddly's DSL contains higher level functions such as A* search and proximity sensing, which can be configured to build higher-level behaviours for NPCs. MiniHack's DSL gives access to all of the objects within the base NetHack game, and allows them to be configured at a high level using modifiers such as "hostile" or "peaceful". The choice of using YAML is also more flexible, as YAML is a common DSL with supporting libraries in many different languages. This allows the generation and manipulation of GDY files without requiring the construction of parsers or serializers. Integrations of higher level tooling such as GriddlyJS are made possible due to this. We provide a simple example Griddly environment implementation in Appendix 3.7.

3.3.1 Architecture

Griddly provides environments configured with user-defined Griddly Description YAML, which is explained in detail in Section 3.4. An Environment E can be thought of as an MDP with specific transition function $\mathcal{T}$, state S, space A, observation O and reward R spaces. In addition, Griddly can be configured to provide multiple observation functions Ω.

An environment's state contains *objects* which are distinct components arranged in the environment's grid layout. For example, an environment like a maze will contain walls, an avatar, and a goal. The walls, avatar, and goal are configured as *objects*.

objects contain *variables* which can be modified by the transition function of the environment, such as the x and y position of the *object* or other user-defined state variables. The state can also contain *global variables* accessible and modifiable by the transition function.

The transition function $\mathcal{T}$ in Griddly environments are configured by *action behaviors*. *action behaviors* are sets of instructions performed when the agent interacts with the environment. The action space A of the MDP represented by the environment is configured by these *action behaviors*. The states of the environment in which rewards are given are also defined in the *action behaviors*. More specifically, there may be an instruction in an *action behavior*, which sets a reward on execution.

Reinforcement learning agents may be able to interact with multiple *objects* in an environment or may be embodied in a single *object*. In the case where an agent can control multiple *objects* these *objects* are referred to as *units*, whereas if the agent is embodied in a single *object* this object is referred to as an *avatar*.

The observation space O of the environment is configured using *observers*, which are equivalent to the MDP observation function Ω. Griddly provides several different configurable formats of *observer*, which are described in section 3.5.1. In visual observation spaces, we commonly refer to *objects* as *tiles*. These *tiles* are images that represent the objects in the environment.

3.4 Griddly Description YAML (GDY)

Griddly Description YAML (GDY) is a schema-oriented domain-specific language (DSL) that allows great flexibility in creating grid-world environments. Any DSL requires a certain amount of pre-existing knowledge; however, with industry-standard configuration languages, there are many tools available such as syntax highlighting, schema validation, and linting, which can reduce the barrier to entry when writing DSLs. The use of YAML as a syntax allows schema validation to be used for syntax highlighting, validation, and auto-completion. Many IDEs such as Visual Studio Code, IntelliJ, and PyCharm support YAML validation out-of-the-box using JSON schemas. This makes the development of new games using Griddly simpler as the IDE will provide feedback on the game description's syntax and structure.

Full documentation of the schema of GDY can be found in the Griddly Documentation `https://griddly.readthedocs.io/en/latest/reference/GDY/index.html`.

3.4.1 Environment Configuration

The environment Section of the YAML contains configuration options for several high-level concepts for a Griddly game. The three most important of these are the *Player*, *Termination* and *Levels* options. The Player options define how the players will interact with the environment and which, if any, avatar object the player will control. Player partial observability can also be configured in the *Observer* subsection here using options such as: *RotateWithAvatar*, which causes the environment representation to rotate if the avatar rotates; *TrackAvatar*, which enables egocentric rendering (the agent is put at the center of the observation and *OffsetX* and *OffsetY* can be used to offset the agent from the center) and finally *Height* and *Width* which determine the height and width of the observable window.

Termination conditions, such as determining if the episode is complete, or determining the winner in a multi-player game are also set in the environment section. Termination options can use any variable defined at a global level and also several special variables that allow calculations such as counts of specific objects in the environment. Levels are defined using strings of characters, referred to as *Level Strings*. the characters that are used are defined in the *objects configuration*. An example of an environment configuration is shown below. An example of configured egocentric partial observability with an Isometric Renderer can be seen in figure 3.4

3.4.2 Action Behaviour Configuration

Instead of having a fixed set of actions, GDY allows the user to define any number of actions and how they will interact with other objects. Actions have the ability to modify any variables associated with the objects involved and any global variables.

Actions in Griddly are defined in two parts, the **Input Mapping** and the **Behaviours**. The Input Mapping maps a set of distinct integers to a *Description*, *OrientationVector* and *VectorToDest*. The OrientationVector and Vector-

ToDest are parameters that can then be used by the **Behaviours** to define how different objects react when the action is performed on them by another object. The object that is performing the action is referred to as the *source* object and object that is the target of the action is referred to as the *destination*. The defined behaviours in actions can also be subjects to conditions on the variables of objects or global variables.

3.4.3 Object Configuration

Objects in environments are defined individually. The object definition allows individual objects to contain encapsulated variables, for example; hit points, resources and possessions such as keys. Rendering information is also defined per-object and passed to the rendering engine at run-time.

3.5 Observation Space Configuration

In this section we outline the numerous ways in which the observation spaces of Griddly environments can be configured. With this flexibility of observations spaces, Griddly can replicate the observations of many different grid-world environments, but also add the benefits of flexibility, which allows these environments to be re-configured quickly and easily whilst maintaining efficiency.

3.5.1 Observers

In Griddly, observation function configurations are referred to as **observers**. Each different observer represents a subset of possible observation functions Ω that can be used to either render the environment for viewing and debugging policies or provide an observation space for an agent.

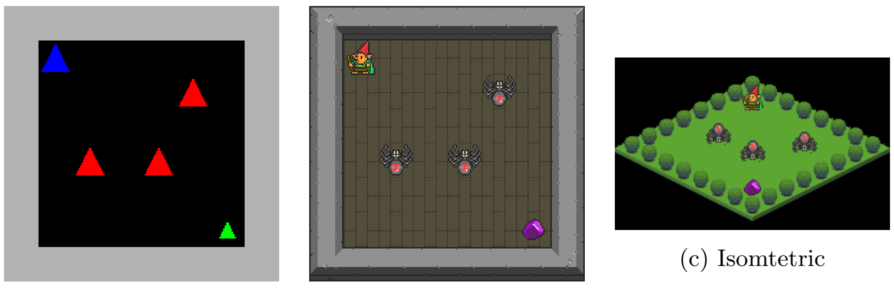

(a) Block2D (b) Sprite2D (c) Isomtetric

Figure 3.4: Visualization of the three observers, *Block*, *Sprite* and *Isometric* configured on the "Spiders" environment. The 4th *Vector* observer does not have a natural visual representation.

Block2D observers are a simplistic visual observation space that renders only shapes and colours. The shapes, scale and colors can be customized per-object and can be modified by GDY behaviours. The observations space is RGB $\mathbb{R}^{H\times W\times 3}$ where H and W are the height and the width of the observation space. An example of these observations can be seen in Figure 3.4a

Sprite2D observers are similar to Block2D observations in terms of their available configuration properties, however they allow images and tilesets to be used instead of simple shapes. This allows more visually rich environments to be generated. Again the observation space is RGB encoded $O \in \mathbb{R}^{H\times W\times 3}$. The Sprite2D rendering of the spiders environment can be seen in 3.4b

Isometric observers (Figure 3.4c are also RGB encoded $O \in \mathbb{R}^{H\times W\times 3}$, and are configurable in the same way as Sprite2D and Block2D observers. Isometric observers differ in visual presentation in that they render the game *isometrically*, which is a technique common to rogue-like games. Isometric observations are more visually appealing, as they appear 3D to the viewer but are rendered using 2D sprites.

Vector observers consist of a set of features for each tile in the observable grid, that is to say, the observation space can be arranged as a single tensor $O \in \mathbb{R}^{H\times W\times C}$ where H and W are the height and the width of the observation and C is the "channel" dimension which represents the features at each location. The channel dimension consists of a one-hot encoding of the "type" of object, followed by optional features such as object rotation, agent id, object variable values and global variable values.

ASCII observers present the environment as a string of ASCII characters, each character representing an object. ASCII representation can be particularly useful when building PCG environments as the observations are compatible with the level descriptions used to define environment layouts. This can assist in building PCGRL [169] environments where agents design or modify level layouts.

Entity observers represent each object in the environment as a set of features. Each feature set contains variables such as x and y coordinates and object variables can be provided also. Given that the environment has a layout the same as those in figure 3.4, the environment would consist of the following set of objects and their respective features: 30 "wall" objects $W \in \mathbb{R}^{30\times f_w}$, 3 spider objects $S \in \mathbb{R}^{3\times f_s}$, one goal $G \in \mathbb{R}^{1\times f_g}$ and one "gnome" avatar $A \in \mathbb{R}^{1\times f_a}$.

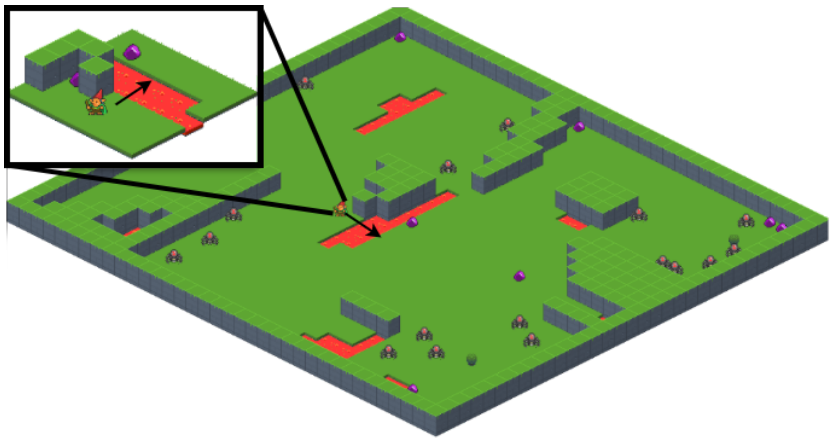


Figure 3.5: An example of the rendering of the Griddly environment *spider nest* with a specific player's egocentric partially observable view that an agent will see during training (inset)

Entity observations are typically embedded using a linear neural network and then fed to a multi-headed attention transformer [303].

3.5.2 Partial Observability

Many of the markov-decision processes used in research provide an observation space that is only a subset of the environment. These partially observable environments are significantly more realistic and provide much more challenging scenarios, especially in multi-agent games where the agents are required to predict or pre-empt the strategy of other agents in order to compete or collaborate.

In the Griddly Engine there are several options available for partial observability. Firstly, the height and width of the observation space can be modified to reduce the perceptual field of the environment. Offsets can also be applied to specifically place the perceptual field at any part of the environment. Egocentric partial observability can also be configured, which will "track" the agent as it moves around the environment. The "tracking" mechanism can also be configured to rotate around the agent's location, so that the observations are relative to the angle of the agent itself.

Partial observability can be configured globally for all defined observers, or it can be defined differently per each observer. For example there could be a fully observable Block2D observation function used as a mini-map alongside a vector or entity observation space.

We show an example of an isometric observer with partial observability in Figure 3.5.

3.5.3 Custom Shaders

The visual observers (Sprite, Block and Isometric) are implemented using Vulkan Graphics Pipelines [2], which support the SPIR-V shader language. This high-performing shader language not only provides fast GPU rendering support, but also allows customization of the rendering. When configuring custom shaders, variables can be made available in the graphics pipeline, meaning that they can be referenced in the shader. This allows significantly flexibility when producing visual representations. Custom shaders can be used to add noise, fog-of-war, lighting, or even "juice" effects to make the visualizations more appealing.

Figure 3.6 shows an example of a custom shader which renders coloured bars on agents in a multi-agent game.

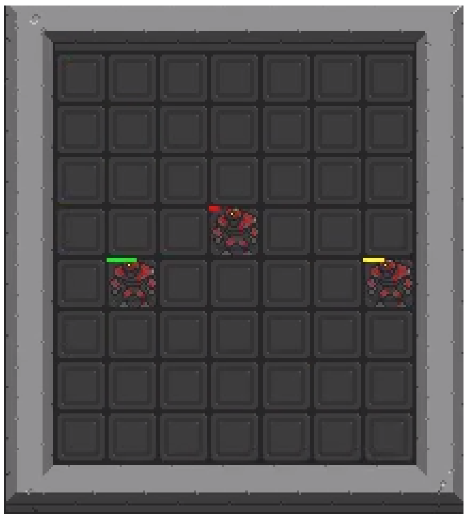


Figure 3.6: A Custom shader rendering a local object variable (health) using Signed Distance Functions in a custom shader.

3.6 Action Space Configuration

Action spaces in Griddly follow the **Conditional Action Tree** representation which is described in more detail in chapter 4.2. In general, Griddly will automatically map the action space to the actions that are defined in the GDY [2]. We will define a few specific cases for action spaces here.

[2]More details with code examples can be found in the Griddly documentation `https://griddly.readthedocs.io/en/latest/getting-started/action%20spaces/index.html#examples`

3.6.1 Single Agent

Firstly, we consider a single agent that controls the movement of an avatar in an environment, there are only 4 "move" commands that can be issued: up, down, left, right. To set the object that will be controlled by the agent, an **avatar** object must be defined in the Environment section of the GDY. (In this example the object name is "gnome"):

```
Player:
  AvatarObject: gnome
```

As there is only a single action defined with 4 inputs, and the environment is set control a single avatar object, then only a single integer is required for the action space. In addition to the 4 possible action inputs, Griddly also automatically adds a NOP as the first action. This makes the full action space $a \in [0..4]$.

3.6.2 Single Agent - Multiple Object

If the AvatarObject is not defined, then the Griddly Engine will assume that the agent can control multiple objects in the environment. In order to make an object controllable, the object in the level string must be followed by the integer referring to which agent the object belongs (in the single agent case this would only be 1). For example "w1" will be controllable by the agent and "w" will be a fixed object. The object to be controlled is chosen by it's x and y coordinate, then the action to perform follows this.

Assuming we have an environment with up, down, left and right actions for each controllable object type, we first have to select the object and then issue the direction command. This action is of the form $a = [a_w, a_h, a_{id}]$ where $a_w \in [0..W]$, $a_h \in [0..H]$, and $a_{id} \in [0..4]$.

To select an object at 5,6 to go down, the action would be specified as $[5, 6, 2]$. To control multiple objects at once a vector of actions can be used e.g: $[[5, 6, 2], [3, 4, 3]]$

3.6.3 Multi-Agent

Multi-agent environments can be configured simply by adding a PlayerCount Key which has a value greater than 1:

```
Player:
  PlayerCount: 3
  AvatarObject: gnome
```

In the above code snippet, we have specified that there are three agents in our environment. We can define which object is owned by which player by adding a player id next to the map character in the level description. For example, to place three "gnome" objects with map character "g", each is assigned to a different agent:

```
Levels:
  - |
    w  w  w  w  w  w  w
    w  g1 .  .  .  .  w
    w  .  .  .  .  .  w
    w  .  .  g2 .  .  w
    w  .  .  .  .  .  w
    w  .  .  .  .  g3 w
    w  w  w  w  w  w  w
```

The action space for multiple agents is defined in the same way as single agents, and methods such as self-play can be achieved by parallelising actions to multiple agents at the same time.

3.6.4 Multi-Agent - Multiple Object

If there are multiple objects that can be controlled by multiple different agents then the action space becomes a combination of both the previous examples. Objects to be controlled must be given a number that assigns the ownership of that object, and then actions must include the x and y coordinates of the object to perform the action.

3.6.5 Multiple Action Types

A final part of the action space configuration is that of when there are multiple "action types" that objects can be assigned. For example there may be a "move" action, but there also may be non-directional action such as "collect", "build", "harvest" etc... these "action types" are automatically generated by Griddly when the GDY is parsed. If there are multiple "action types", the "action type" precedes the action itself.

In an environment with a single agent that can perform actions "move (up, down, left, right)" and "push (up, down, left, right)" the action space would be $a = [a_t, a_{id}]$ where $a_t \in [0..1]$ (move/push) and $a_{id} \in [0..4]$

In the multiple-object case this would expand with the a_w and a_h terms to $a = [a_w, a_h, a_t, a_{id}]$.

3.7 Griddly GDY Example - Sokoban

To provide an intuition on how environments can be implemented using Griddly, we provide a brief tutorial of the GDY below which recreates the popular game of Sokoban. Once the GDY is completed, Griddly's environment wrappers can easily map it to a Gym environment.

3.7.1 Objects

In this section we describe the objects of the game and the ways they can be rendered in Griddly.

The game of Sokoban is composed of four objects: **avatar**, **box**, **hole**, and **wall**. **avatar** is the main decision-making object which can move around and push boxes into holes. Walls are immovable objects. The goal of the **avatar** is to push all boxes into the holes.[3]

Each object needs to have a unique name, which can serve as references to that object in other parts of the GDY code. Let us start the `Objects` section and define the **avatar** object as follows:

```
Objects:
 - Name: avatar
   Z: 2
   MapCharacter: A
   Observers:
     Sprite2D:
       Image: images/gvgai/oryx/knight1.png
```

`MapCharacter` is used to define the ASCII character of an object and can be used to mark the initial positions of objects in concrete levels defined later in the `Environment` section.

The property *Z* can serve as the third dimension of the cells in the grid. It allows to define objects that can occupy the same location of the grid, as long as they have different *Z* values. It also defines the order of objects when rendered in Griddly, i.e., higher *Z* values indicate that the objects will be rendered on top.

The `Observers` property determines how each observer type will render this particular object. Here, the **avatar** object only includes a Sprite2D observer, but Griddly supports additional forms of observations, including those shown in Figure 3.4. Sprite2D observers expect an image for rendering, thus we select a knight icon from Griddly's selection of icons.[4]

[3]There are many variations of the game of Sokoban. In our particular implementation the agent can push boxes into any hole on the environment. There also exist versions where only a single box can be pushed into each hole.

[4]Griddly allows users to easily upload new custom icons for their own environments.

Having finished the description of the **avatar** object, we proceed to the **wall** object. Here we provide 16 different images for walls to correspond to different positions of walls, such as horizontal or vertical locations, corner pieces, T-pieces, etc.

```
- Name: wall
  MapCharacter: w
  Observers:
    Sprite2D:
      TilingMode: WALL_16
      Image:
        - images/gvgai/oryx/wall3_0.png
        - images/gvgai/oryx/wall3_1.png
        - images/gvgai/oryx/wall3_2.png
        - images/gvgai/oryx/wall3_3.png
        - images/gvgai/oryx/wall3_4.png
        - images/gvgai/oryx/wall3_5.png
        - images/gvgai/oryx/wall3_6.png
        - images/gvgai/oryx/wall3_7.png
        - images/gvgai/oryx/wall3_8.png
        - images/gvgai/oryx/wall3_9.png
        - images/gvgai/oryx/wall3_10.png
        - images/gvgai/oryx/wall3_11.png
        - images/gvgai/oryx/wall3_12.png
        - images/gvgai/oryx/wall3_13.png
        - images/gvgai/oryx/wall3_14.png
        - images/gvgai/oryx/wall3_15.png
```

We do not provide this object with a `Z` value given that nothing should interact with this object. WALL_16 tiling mode is used to make sure all 16 wall icons are rendered correctly for each location.

```
- Name: box
  Z: 2
  MapCharacter: b
  Observers:
    Sprite2D:
      Image: images/gvgai/newset/block1.png

- Name: hole
  Z: 1
  MapCharacter: h
  Observers:
    Sprite2D:
      Image: images/gvgai/oryx/cspell4.png
```

The **box** and **hole** objects can be defined similar to the **avatar** objects, except that the **hole** objects have a different `Z` value allowing the avatar to move on top of them.

3.7.2 Actions

`Actions` define the mechanics of the game and interactions between objects in Griddly. Each individual action includes two entities: **source** and **destination**.

source is the object which performs a particular action, whilst the **destination** is the object that is affected by this action. Firstly, we define the movement action of the **avatar** as follows.

```
Actions:
 # Define the move action
 - Name: move
   Behaviours:
   # The agent can move around freely in empty space and over holes
     - Src:
         Object: avatar
         Commands:
           - mov: _dest
       Dst:
         Object: _empty
```

Given that the Src key includes **avatar** object as its value, it can be inferred that this is an action performed by the **avatar**. The Dst key with the object value _empty indicates that the behaviour only applies when the action is performed on a space with no objects on it.

The Commands property in the Src field includes a list of instructions that will be executed to the Src object once this action is performed. The `mov: _dest` commands moves the object to the destination of the action.

Next, we define the box pushing actions. Firstly, we define the ability of **box** objects to move to empty locations. Then, we allow the **avatar** object to interact with the **box** object.

```
# Boxes can move into empty space
- Src:
    Object: box
    Commands:
        - mov: _dest
  Dst:
    Object: _empty

# The agent can push boxes
- Src:
    Object: avatar
    Commands:
        - mov: _dest
  Dst:
    Object: box
    Commands:
        - cascade: _dest
```

Here we make sure that the **box** object is moved in the same direction as the **avatar** object, the source of the action. We achieve this by using the `cascade: _dest` which re-apply the same action on the destination object, namely the **box**.

Finally, we define the mechanics of the **box** being pushed onto a **hole**. We achieve this by defining our last action with **box** as its source and **hole** as its destination:

```
# If a box is moved into a hole remove it
 - Src:
     Object: box
     Commands:
       - remove: true
       - reward: 1
   Dst:
     Object: hole
```

Here, the `remove: true` command removes the source **box** from the grid once pushed into a hole. Furthermore, the `reward: 1` commands Griddly to provide the agent with the reward of 1 once this event is triggered.

3.7.3 Environment

The `Environment` section defines the environment description, such as its name, as well as the observation and action spaces of the MDP.

Below we provide the `Environment` section for our Sokoban example. Firstly, we indicate that the **avatar** object will serve as the decision-making agent in our environment:

```
Player:
  AvatarObject: avatar
```

We then describe the termination condition which determines when the episode is complete and whether the agent wins or loses. For Sokoban, an episodes is considered won if all the boxes are pushed into the holes, i.e., the number of boxes in the environment is equal to 0:

```
Termination:
    Win:
     - eq: [box:count, 0]
```

Our next step is to defines levels for our Sokoban game. The layout of each level can be defined using a sequence of strings that uses the `MapCharacter` characters of each object defined above. The dot character `.` indicates an unoccupied grid cell.

```
Levels:
    - |
      wwwwwww
      w..hA.w
      w.whw.w
      w...b.w
      whbb.ww
      w..wwww
      wwwwwww
    - |
```

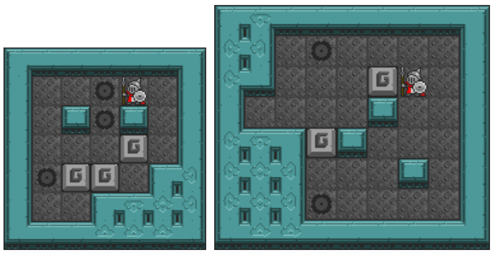


Figure 3.7: Custom Sokoban levels defined in the GDY example.

```
        wwwwwwwww
        ww.h....w
        ww...bA.w
        w....w..w
        wwwbw...w
        www...w.w
        wwwh....w
        wwwwwwwww
```

The two defined levels produce the environment renderings illustrated in Figure 3.7.

Lastly, we specify the size of the tiles in pixels and the background image using the `MapCharacter` and `BackgroundTile` fields. We also provide the environment with a unique name.

```
Environment:
   Name: sokoban
   TileSize: 24
   BackgroundTile: images/gvgai/newset/floor2.png
```

3.7.4 Putting It All Together

Figures 3.8 and 3.9 provide the full implementation of the Sokoban example in Griddly.

```
Environment:
   Name: sokoban
   TileSize: 24
   BackgroundTile: images/gvgai/newset/floor2.png
   Player:
     AvatarObject: avatar
   Termination:
     Win:
       - eq: [box:count, 0]
   Levels:
     - |
       wwwwwww
       w..hA.w
       w.whw.w
       w...b.w
       whbb.ww
       w..wwww
       wwwwwww
     - |
       wwwwwwwww
       ww.h....w
       ww...bA.w
       w....w..w
       wwwbw...w
       www...w.w
       wwwh....w
       wwwwwwwww
Actions:
- Name: move
  Behaviours:
    - Src:
        Object: avatar
        Commands:
          - mov: _dest
      Dst:
        Object: [_empty, hole]
    - Src:
        Object: box
        Commands:
            - mov: _dest
      Dst:
        Object: _empty
    - Src:
        Object: avatar
        Commands:
          - mov: _dest
      Dst:
        Object: box
        Commands:
          - cascade: _dest
    - Src:
        Object: box
        Commands:
          - remove: true
          - reward: 1
      Dst:
        Object: hole
```

Figure 3.8: Full implementation of Sokoban in Griddly (part 1).

```yaml
Objects:
 - Name: box
   Z: 2
   MapCharacter: b
   Observers:
     Sprite2D:
       Image: images/gvgai/newset/block1.png

 - Name: wall
   MapCharacter: w
   Observers:
   Sprite2D:
     TilingMode: WALL_16
     Image:
       - images/gvgai/oryx/wall3_0.png
       - images/gvgai/oryx/wall3_1.png
       - images/gvgai/oryx/wall3_2.png
       - images/gvgai/oryx/wall3_3.png
       - images/gvgai/oryx/wall3_4.png
       - images/gvgai/oryx/wall3_5.png
       - images/gvgai/oryx/wall3_6.png
       - images/gvgai/oryx/wall3_7.png
       - images/gvgai/oryx/wall3_8.png
       - images/gvgai/oryx/wall3_9.png
       - images/gvgai/oryx/wall3_10.png
       - images/gvgai/oryx/wall3_11.png
       - images/gvgai/oryx/wall3_12.png
       - images/gvgai/oryx/wall3_13.png
       - images/gvgai/oryx/wall3_14.png
       - images/gvgai/oryx/wall3_15.png

 - Name: hole
   Z: 1
   MapCharacter: h
   Observers:
     Sprite2D:
       Image: images/gvgai/oryx/cspell4.png

 - Name: avatar
   Z: 2
   MapCharacter: A
   Observers:
     Sprite2D:
       Image: images/gvgai/oryx/knight1.png
```

Figure 3.9: Full implementation of Sokoban in Griddly (part 2).

3.8 Baselines

In this section we introduce a set of experiments on an array of Griddly environments. The goal of these experiments is to:

- Provide a set of baselines for any future improvements to be compared against.

- Showcase the flexibility of Griddly by using many different engine features.
- Highlight the ability for Griddly to provide a challenging substrate for reinforcement learning research.

Firstly we chose 10 environments that have been ported from the GVGAI environment that are particularly difficult for RL algorithms to solve. Each environment contains 5 levels of different sizes and varying difficulty. We trained each level from each environment separately using three different observers *Vector*, *Block* and *Sprite*. Additionally two of the environments are configured with egocentric partial observability. This baseline suite consists of 150 experiments in total. Each level is trained for 1.28M frames, and the average score for the 100 final episodes during training is measured. Table 3.1 shows the results of these experiments.

The second baseline uses a subset of the previous experiments, but all configured with egocentric partial observability. These experiments have an observation space of 5×5 tiles, with the agent centered at the bottom of this grid. This allows the agent to see 4 squares ahead of it in the direction of travel, and 2 squares either side. Additionally, instead of training each level separately, 3 of the levels are used as a training set and then the 2 remaining levels are used to evaluate generalization. These experiments are all performed using only the *Vector* observer to produce game states.

The neural network's architecture for both sets of experiments is the same. Both networks are trained using Proximal Policy Optimization [252] and Random Network Distillation [48] (RND).

All environments in the baseline examples are deterministic. Griddly does allow stochastic behaviors to be implemented but this is beyond the scope of the baselines in this section.

3.8.1 Random Network Distillation

Random Network Distillation (RND) is a technique used in reinforcement learning, particularly in exploration strategies. Its central idea revolves around the concept of "curiosity", where the agent gets rewards for exploring unfamiliar states. This idea makes RND useful in solving environments with sparse rewards.

The mechanics of RND are as follows:

Two networks are trained a fixed randomly initialized target network and a predictor network. Both of these networks have the same architecture, which usually consists of several layers of a neural network.

Observation processing The state or observation from the environment goes through these networks. The target network produces a random feature vector from the observation, while the predictor network tries to match this output. The predictor network is updated based on a loss function (typically mean squared error) between its output and the target network's output.

Intrinsic reward calculation The difference between the output of the two networks is used to form an intrinsic reward signal. If the predictor network struggles to predict the output of the fixed random network (meaning the agent is in a state it hasn't seen much before), then the intrinsic reward is high. Conversely, if the predictor can easily predict the output (meaning the agent is in a state it's familiar with), the intrinsic reward is low.

Policy update This intrinsic reward is then used to guide the policy of the agent. In other words, the agent is motivated to go to places where the predictor network struggles to mimic the random network. This motivates the agent to explore unseen or less familiar states in the environment.

RND is effective for encouraging exploration in environments with sparse rewards because it generates an auxiliary reward signal based on the novelty of states. This allows the agent to discover and learn from new experiences even when the environment does not provide any explicit rewards.

3.8.2 Network Architecture

The agent network, RND target network and RND prediction network share the same architecture per experiment. The only difference in networks occurs in the first few layers as the observation space differs between different observers used. Experiments using *Sprite* and *Block* observers begin with a convolutional layer with kernel size and stride the same as the tile size. This effectively embeds each tile into tensor $\mathbb{R}^{32\times W\times H}$ where C is the number of channels, and H and W are the height and width of the game level in terms of tiles. Alternatively experiments using *Vector* observers embed the vector representations using a 1×1 kernel with 32 channels into the same resulting shape $\mathbb{R}^{32\times W\times H}$. The embedded representation is then fed through the following layers: A convolutional layer with 32 input channels, 64 output channels, kernel size 3 and padding 1. Then a convolutional layer with 64 input and output channels, with kernel size 3 and padding 1. A global average pooling (GAP) layer is then used to reduce the network size to a linear layer of size 2048, this is then reduced by a further linear layer to a vector size 512. Single linear layers are used after this point for the various heads such as the actor and critic.

The global average pooling layer allows different H and W for different level sizes to output the same shape vector (in our case 2048) [25].

The action space of the fully observable environments consists of 5 actions *do nothing (no-op)*, *move up*, *move left*, *move down*, and *move right*. In contrast, the partially observable setting the agent has access to 4 actions, *do nothing (no-op)*, *turn left*, *turn right* and *move forward*.

3.9 Results

3.9.1 Per-Environment

As shown in the results table 3.1 and training plots 3.10, 3.11, there are levels of specific environments that fail to gain any score. These environments are sometimes difficult to solve even for humans, as they require precise planning, and making single mistakes result in states that cannot be reversed. For example, in the "Zenpuzzle" the agent gains a reward every time it moves over tiles of a certain color, but once it has moved over these blocks, it can not move onto them again. This can cause the agent to get "stuck" surrounded by tiles that it has already passed. The agent can also easily stop itself from being able to reach certain tiles by blocking the path to them. Although the agent scores highly in the "Zenpuzzle" environment, it rarely covers "all" the tiles to reach the maximum score possible. In the game "Clusters" the agent must push colored blocks into groups and score a point each time a block is "grouped". There are other obstacles in the environment that the agent must avoid pushing the blocks into. In the "Clusters" experiments, the agent rarely learns to make a single group.

We test two partially observable versions of the games "Labyrinth" and "Zen Puzzle" to see if more general approaches, such as wall-following strategies, can be learned. As expected in "Labyrinth", a simple maze game with a reward at the destination, the agent performs better and can solve some of the mazes. We observe that in some of the games where a "wall following" approach can solve the maze, the agent learns this strategy. In other levels, the agent does not learn a strategy and fails to find the goal. It is also interesting to note that the full observable maze levels could not be learned by this method.

In the "Sokoban" environment, some levels are consistently solved. However, on other levels, the agent consistently fails to score a single point. This highlights that the structure of a level and the strategy required to solve the "Sokoban" levels can require different approaches to training, even when the mechanics of the game are consistent across levels.

Game	Level	Max Reward	Observer		
			Vector	Block	Sprite
	0	8	-0.06 ± 0.24	0.00 ± 0.00	2.00 ± 0.00
	1	8	-0.02 ± 0.14	3.00 ± 0.00	0.00 ± 0.00
Clusters	2	6	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	3	9	-0.04 ± 0.20	1.00 ± 0.00	1.00 ± 0.00
	4	6	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	0	25	0.08 ± 0.57	8.00 ± 0.00	0.00 ± 0.00
	1	25	1.47 ± 1.93	8.00 ± 0.00	0.00 ± 0.00
Cook Me Pasta	2	25	-0.04 ± 0.20	4.00 ± 0.00	0.00 ± 0.00
	3	25	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	4	25	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	0	5	5.00 ± 0.00	5.00 ± 0.00	5.00 ± 0.00
	1	7	0.00 ± 0.00	1.00 ± 0.00	0.00 ± 0.00
Bait	2	12	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
	3	42	7.04 ± 0.28	23.00 ± 0.00	25.90 ± 0.36
	4	12	7.00 ± 0.00	7.00 ± 0.00	7.00 ± 0.00
	0	5	**5.00** ± 0.00	**5.00** ± 0.00	3.98 ± 2.02
	1	7	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Bait (Keys)	2	12	1.00 ± 0.00	1.00 ± 0.00	2.00 ± 0.00
	3	42	7.00 ± 0.00	5.78 ± 0.65	26.90 ± 0.30
	4	12	7.00 ± 0.00	6.69 ± 0.91	7.00 ± 0.00
	1	4	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	2	3	0.00 ± 0.00	**3.00** ± 0.00	2.00 ± 0.00
Sokoban	3	4	2.00 ± 0.00	**4.00** ± 0.00	2.00 ± 0.00
	4	3	2.00 ± 0.00	2.00 ± 0.00	1.00 ± 0.00
	5	2	0.00 ± 0.00	**2.00** ± 0.00	0.00 ± 0.00
	0	4	2.00 ± 0.00	2.00 ± 0.00	2.00 ± 0.00
	1	3	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Sokoban 2	2	6	5.00 ± 0.00	4.92 ± 0.27	4.96 ± 0.20
	3	2	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
	4	2	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	0	34	27.24 ± 3.11	33.00 ± 0.00	**34.00** ± 0.00
	1	34	30.53 ± 1.74	**34.00** ± 0.00	33.00 ± 0.00
Zen Puzzle	2	33	26.14 ± 2.27	30.31 ± 1.68	28.98 ± 0.14
	3	23	19.94 ± 0.24	18.00 ± 0.00	20.00 ± 0.00
	4	27	26.90 ± 0.30	25.98 ± 0.14	25.00 ± 0.00
	0	1	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	1	1	-0.04 ± 0.20	0.00 ± 0.00	0.00 ± 0.00
Labyrinth	2	1	-0.02 ± 0.14	0.00 ± 0.00	0.00 ± 0.00
	3	1	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	4	1	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	0	1	**1.00** ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	1	1	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Labyrinth ^{po}	2	1	**1.00** ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	3	1	0.00 ± 0.00	0.00 ± 0.00	**1.00** ± 0.00
	4	1	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	0	34	**34.00** ± 0.00	31.02 ± 1.67	30.00 ± 0.00
	1	34	27.00 ± 0.00	32.00 ± 0.00	31.88 ± 0.48
Zen Puzzle ^{po}	2	33	29.00 ± 0.00	31.00 ± 0.00	28.00 ± 0.00
	3	23	19.84 ± 0.37	19.90 ± 0.30	19.63 ± 0.48
	4	27	22.00 ± 0.00	22.00 ± 0.00	22.00 ± 0.00

Table 3.1: This table shows the results of training 5 levels of 10 GVGAI environments that have been ported to the Griddly platform using GDY. Each level from each environment is trained for 1.28 million steps, and the average reward of the final 100 episodes is reported for each observer used. (Vector, Block, and Sprite). In most cases the results are consistent across all the representations, but due to the unstable nature of RL based on random starting seeds, some errors in consistency are expected. The two environments marked with ^{po} are configured with egocentric partial observability. Due to computational constraints, we were only able to train a single model per environment

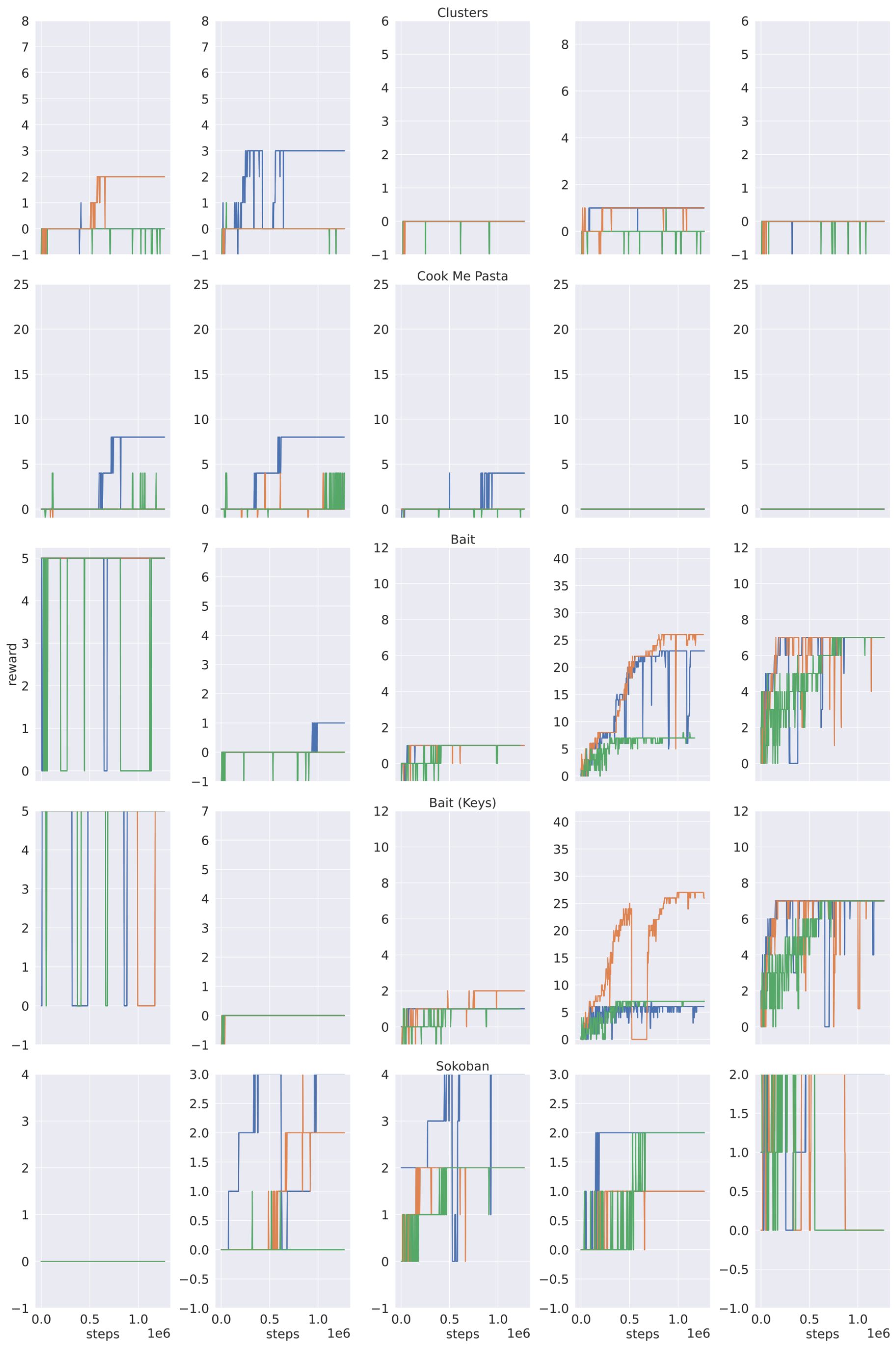


Figure 3.10: Training curves for the first 5 games shown in table 3.1

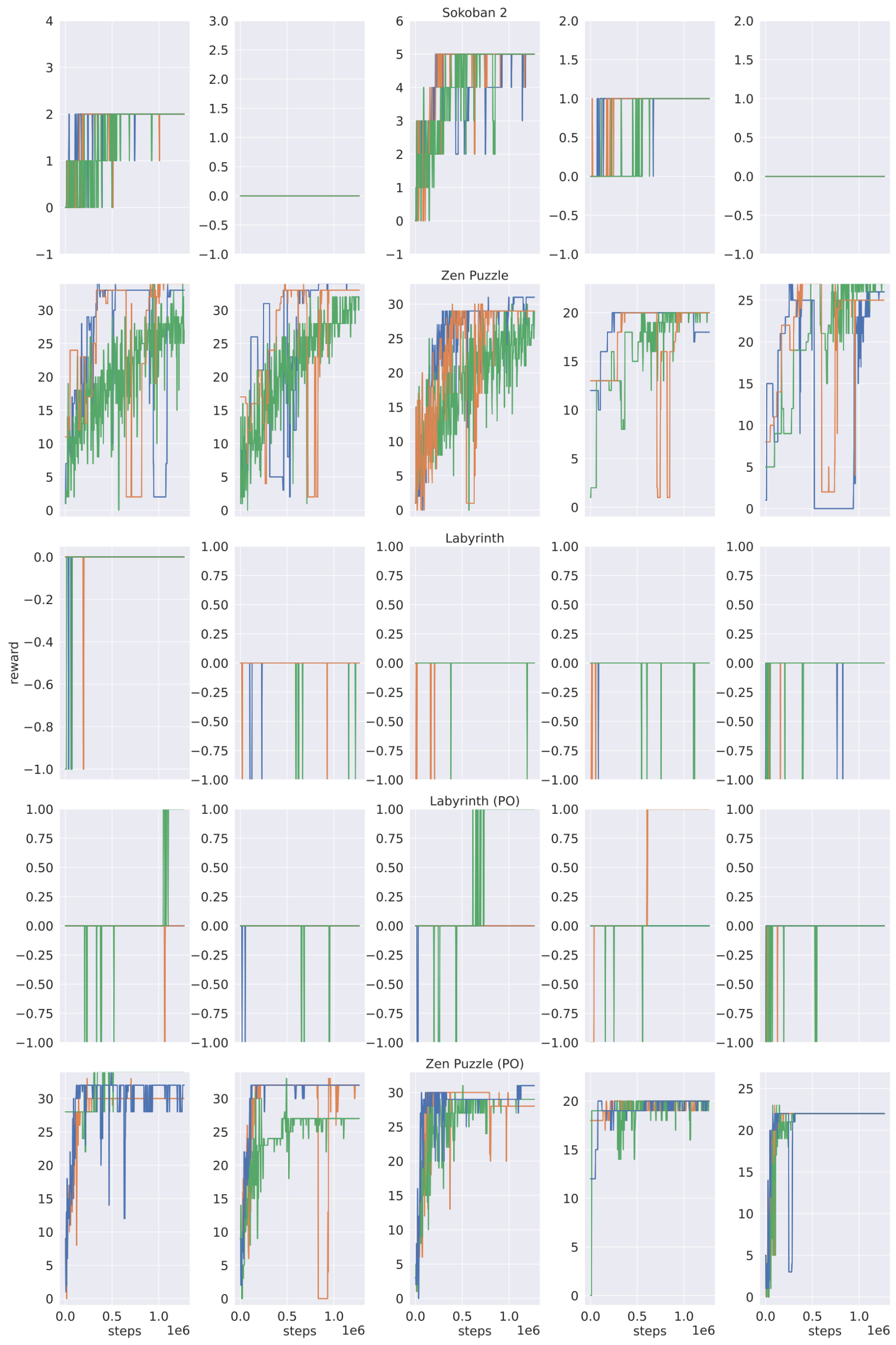


Figure 3.11: Training curves for the first 5 games shown in table 3.1

Game	Evaluation Level 1	3
Clusters	0.00 ± 0.00	0.7 ± 0.46
Cook Me Pasta	4.00 ± 0.00	0.00 ± 0.00
Bait	-0.09 ± 0.29	1.78 ± 0.42
Sokoban 2	0.00 ± 0.00	0.00 ± 0.00
Zen Puzzle	23.00 ± 0.00	10.9 ± 5.01
Labyrinth	0.00 ± 0.00	**1.00** ± 0.00

Table 3.2: This table shows that a simple neural network is able to generalize when trained on three levels and evaluated on two unseen levels. The agent can only see an egocentric, partially observable view of the environment.

3.9.2 Generalization

The three training levels used in each experiment are 0,1,4, and the evaluation levels are 1 and 3. Table 3.2 shows the results of these experiments. These experiments were also trained for 10M steps instead of the 1.28M used in the single environment experiments. In some levels, the agent can achieve some rewards but can only solve a single unseen "Labyrinth" level out of the evaluation set. The result of the "Labyrinth" experiment trained per level also shows that partial observability seems less challenging for the RL agent than having access to the whole level. The maximum episodic rewards for these environments can be seen in Table 3.1.

3.10 Framework Comparison

In this section, we provide two comparisons between several frameworks. The first comparison is a feature matrix showing the differences in features between Griddly and its closest grid-based relatives. The second comparison is between several games from popular frameworks that have been re-implemented using the GDY language.

3.10.1 Features

Table 3.3 shows how the features offered by Griddly compare with a selection of other environments; since this chapter is about Griddly, it is presented to highlight what Griddly offers that other environments do not. Griddly is most closely related to GVGAI and DMLab2D [35], but with various extensions to provide faster rendering and support for multi-agent and grid-based RTS games similar to μRTS [208].

Although ALE is not technically a grid world, we've included it in the table for comparison. NetHack Learning Environment (NLE) is built around the

		Griddly	GVGAI	gym-microrts	MiniGrid	ALE	NLE	DMLab2D
	Vector	x		x	x	x	x	x
	Block	x		x	x	x		
Observation	Sprites	x	x					x
	Isometric	x						
	ASCII	x	x				x	x
	Entity	x		x				
	Configurable Partial Observability	x					x	
	GPU Accelerated Rendering	x						
	Description Language	x	x					x
	Procedural Content Generation				x		x	x
	Copyable Forward Model	x	x	x	x	x		
	Single	x	x		x	x	x	x
Player Modes	Multi	x	x					x
	RTS	x		x				

Table 3.3: Feature matrix comparing Griddly with other environments.

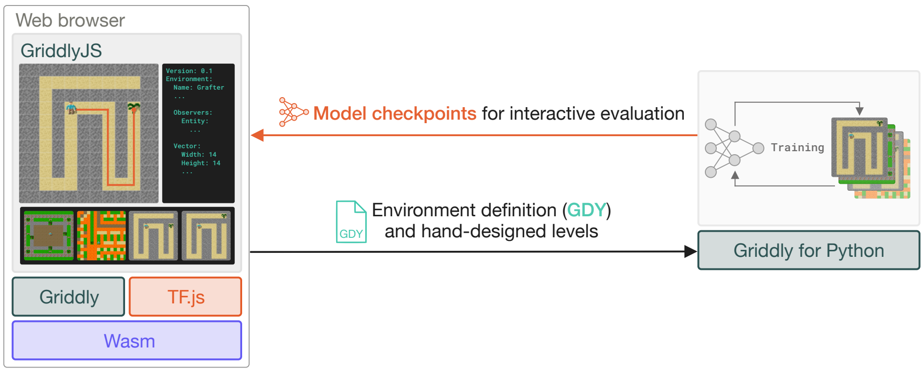
Platform	FPS (Rendered) ± std.	FPS (Vector) ± std.	Max Memory (MB)
Griddly	5023 ± 268	**72790** ± 2474	95
DMLab2D (10x10)	**12815** ± 3863	20562 ± 6658	**94**
Griddly	**3769** ± 124	**65056** ± 1736	138
DMLab2D (50x50)	984 ± 116	17036 ± 5341	**98**
Griddly	**1936** ± 76	**60232** ± 691	371
DMLab2D (100x100)	318 ± 16	8577 ± 2370	**146**
Griddly	**5012** ± 244	**73134** ± 839	**106**
GVGAI GYM	19 ± 5	-	365
Griddly	**3799** ± 170	**61101** ± 4186	106
Minigrid	95 ± 3	1228 ± 25	**49**
Griddly	**1160** ± 157	**32130** ± 419	**106**
gym-microRTS	177 ± 12	1906 ± 272	278

Table 3.4: Speed and memory footprint of Griddly compared to similar environments. All environments are tested using the python OpenAI gym interface except from DMLab2D which has its own equivalent python interface. In each double-row, the Griddly entries are for the same or similar game running on each of the platforms.

classic NetHack Rogue-like game, and although just one grid-based game, it offers great variety due to procedural level generation. Although not included in the table, ProcGen is similar to ALE in scope but offers endless variety through procedural level generation.

3.10.2 Efficiency

As the focus of the Griddly Engine is currently to improve the data rate of RL in grid-world environments, a benchmark comparison of the available Python gym interfaces for some of the most popular grid-based environments is shown in Table 3.4. The benchmark consists of running the original environment and the equivalent Griddly version with a random agent for 1000000 frames and calculating the rendered states' average frames-per-second (FPS) and maximum memory usage. Rendering the pixels of the environments is the most demanding method of producing game states, so it provides a useful bottleneck to test. Additionally, if available, we compare the game engines' vectorized versions of the states. The games and maps used for the tests are GVGAI - Sokoban, MiniGrid - FourRooms, gym-microrts [137] - MicrortsMining-v4. We also provide three separate comparisons to Deepmind 2D lab, which is the most closely related to Griddly. These three comparisons are on three "Pushbox" game levels with sizes 10x10, 50x50, and 100x100. We also configured the tile size to be consistent in Griddly and other platforms.



```
Version: 0.1
Environment:
  Name: Grafter
  ...

  Observers:
    Entity:
      ...

  Vector:
    Width: 14
    Height: 14
    ...
```

Figure 3.12: An overview of the human-in-the-loop environment development workflow enabled by GriddlyJS, built on top of the Griddly engine in Wasm and Tensorflow.js (TF.js). Environments and custom-designed levels can be loaded into Griddly for Python for training, and model checkpoints can be directly loaded into GriddlyJS for visual evaluation.

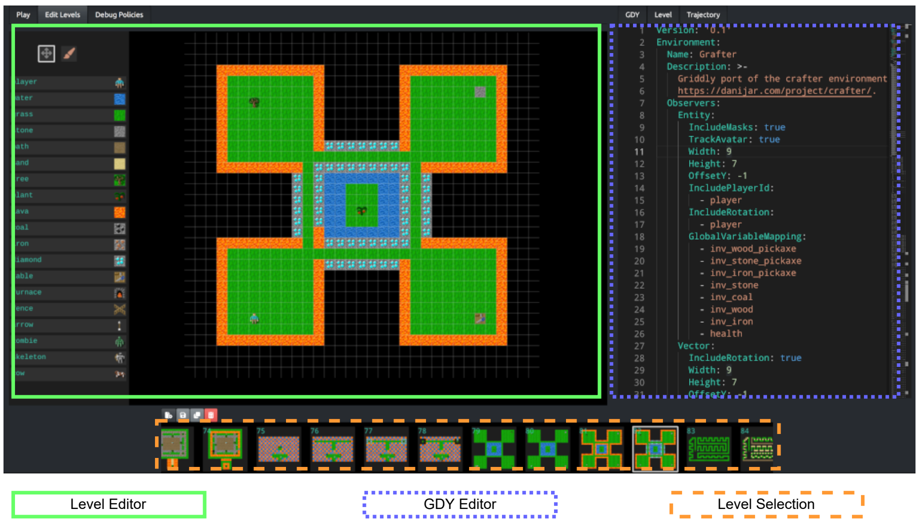
3.11 GriddlyJS

Recently, procedural content generation (PCG) has emerged as the standard paradigm for developing environments that can vary throughout training, enabling the study of systematic generalization and robustness in RL [234, 179, 62, 54, 152, 241]. The more complex programming logic entailed by PCG algorithms, i.e. creating probabilistic programs that specify distributions over environment configurations, adds considerable engineering overhead to the creation of new RL environments.

As research progresses and simpler environments no longer require adequate challenges, there is a need for environments with higher complexity. Due to this, researchers are required to create sufficient challenges and thus spend more time on environment design. These complex environments often come with additional difficulty in the reproduction of results.

Most environment implementations are focused on solving single tasks from particular directions and do not offer additional tooling to enable or assist many approaches. Visualising, evaluating, and recording agent trajectories, for example, usually requires additional effort from the researcher if these techniques are required, and this comes with the additional cost of maintaining these features for further work.

To address these challenges, we introduce GriddlyJS, a web-based integrated development environment (IDE) based on a WebAssembly (Wasm) version of the Griddly engine [29]. GriddlyJS provides a simple interface for developing and testing arbitrary, procedurally-generated grid-world environments in Griddly using a visual editor and domain-specific language based on YAML, with support for highly complex game mechanics and environment generation logic. The visual editor allows rapid design of new levels (i.e. variations of an environ-



```yaml
Version: '0.1'
Environment:
  Name: Grafter
  Description: >-
    Griddly port of the crafter environment
    https://danijar.com/project/crafter/.
  Observers:
    Entity:
      IncludeMasks: true
      TrackAvatar: true
      Width: 9
      Height: 7
      OffsetY: -1
      IncludePlayerId:
        - player
      IncludeRotation:
        - player
      GlobalVariableMapping:
        - inv_wood_pickaxe
        - inv_stone_pickaxe
        - inv_iron_pickaxe
        - inv_stone
        - inv_coal
        - inv_wood
        - inv_iron
        - health
    Vector:
      IncludeRotation: true
      Width: 9
      Height: 7
      OffsetY: -1
```

Figure 3.13: GriddlyJS has three main components: The level editor allows rapid design of custom levels with a code-free, visual interface; rendered levels are fully interactive via keyboard control. The GDY editor allows editing of the underlying GDY specification of the core environment mechanics. The level selection component lists previously designed levels. Users can select levels for further modification or deletion.

ment) via a simple point-and-click interface. Environments can be tested via interactive control of the agent directly inside the IDE.

GriddlyJS produces Griddly game description YAML files (GDY), which define environments and custom levels. GDY files can be loaded directly into Griddly for Python, producing a Gym-compatible environment. In addition, any agent model can be loaded into the GriddlyJS IDE, once easily converted to the TensorFlow.js [TF.js; 266] format, allowing visualizing, evaluating, or recording performance. The integrated development and visualization provided by GriddlyJS enables a whole new mode of closed-loop development of RL environments, in which the researcher can rapidly iterate on environment design based on the behavior of the agent. This allows designing environments that specifically break state-of-the-art RL methods, thereby assisting in pushing the field forward.

GriddlyJS provides a fully web-based integrated development environment (see Figure 3.13) composed of simple and intuitive user interfaces wrapping the core components of the Griddly engine. As such, the GriddlyJS IDE can be used inside any modern browser, without the need for installing complex dependencies. Running Griddly directly inside of the browser is made possible by transpiling the core components of Griddly into WebAssembly (Wasm). The interface itself is written using the React library. Inside the GriddlyJS IDE, GDY files can be directly edited with any changes to the environment's mechanics immediately reflected. Moreover, specific levels of the environment can be designed using a simple visual editor, allowing levels to be drawn directly inside the IDE. Previously designed levels can be saved locally into a gallery and instantly reloaded into the environment instance running inside the IDE and played via keyboard controls. Taken together, the features of GriddlyJS allow for rapid environment development, debugging, and experimentation. We now discuss the major highlights of GriddlyJS in turn. For a detailed walkthrough of these features, see Appendix A.1.

Environment Specification Designing new environment dynamics can be time-consuming. For example, adding and testing a new reward transition requires recompiling the environment and adding specific test cases. With GriddlyJS changes can be coded directly inside the GDY in the browser, where it will be immediately reflected in the environment. The designer can then interactively control the agent to test the new dynamic. Moreover, the environment's action space is automatically reanalyzed on all changes, and environment actions are assigned sensible key combinations, e.g. WASD for movement actions. Similarly, newly defined entities inside the GDY are immediately reflected in the visual level editor, allowing for rapid experimentation.

Level Design Given any GDY file, GriddlyJS provides a visual level editor that allows an end user to design environment levels by drawing tiles on a grid. Objects from the GDY file can be selected and placed in the grid by pointing and clicking. The level size is automatically adjusted as objects are added. The corresponding level description string, which is used by the Griddly environment in Python to reset to that specific level, is automatically generated based on the character-to-object mapping defined in the GDY. New levels can then be saved to the same GDY file and loaded inside the Python environment.

Publish to the Web As GriddlyJS is built using the React library, the environment component itself can be encapsulated inside a React web component. Moreover, GriddlyJS supports the loading and running of TF.js models directly inside the IDE environment instance. Taken together, this allows publishing Griddly environments and associated agent policies in the form of TF.js models directly to the web, as an embedded React web component. By allowing researchers to directly share interactive demos of their trained agents and environments, GriddlyJS provides a simple means to publish reproducible results, as well as research artifacts that encourage the audience to further engage with the strengths and weaknesses of the methods studied.

Recording Human Trajectories Recording human trajectories for environments is as simple as pressing a record button in the GriddlyJS interface and controlling the agent via the keyboard. Recorded trajectories are saved as JSON, consisting of a random seed for deterministically resetting the environment to a specific level and the list of actions taken. They can easily be compiled to datasets, e.g. for offline RL or imitation learning. The recorded trajectories can also be replayed inside of GriddlyJS.

Policy Visualization Visualizing policy behavior is a crucial debugging technique in RL. Policy models can be loaded into GriddlyJS using TF.js and run in real-time on any level. In this way, the strengths, weaknesses, and unexpected behaviors of a trained policy can be quickly identified, providing intuition and clues about any bugs or aspects of the environment that may be challenging in a closed-loop development cycle. These insights can then be used to produce new levels that can bridge the generalization gap and thus improve the robustness of the agent.

3.12 Proof-of-Concept: Escape Room Puzzles

We now demonstrate the utility of Griddly and GriddlyJS by rapidly creating a complex, procedurally-generated RL environment from scratch. After developing this new environment, we then use GriddlyJS to quickly hand-design a large, diverse collection of custom levels, as well as record a dataset of expert trajectories on these levels, which can be used for offline RL. We then load this new environment into Griddly for Python to train an RL agent on domain randomized levels—whose generation rules are defined within the associated GDY specification—and evaluate the agent's performance on the human-designed levels.

3.12.1 Rapid Environment Development

We consider an environment, resembling a 2D version of MineCraft, in which the agent must learn a set of skills related to gathering resources and constructing entities using these resources in order to reach a goal. While prior works have presented environments with similar 2D, compositional reasoning challenges [9, 315, 319], we specifically model our `EscapeRoom` environment after the complex state and transition dynamics of Crafter [117], with the key difference being that `EscapeRoom` episodes terminate and provide a large reward upon reaching the goal object, a cherry tree, in each level.[5] The dynamics inherited

[5]A full description of how `EscapeRooms` deviates from Crafter is provided in Appendix 3.13.

```
- Name: do
  ...
  Behaviours:
    ...
    - Src:
        Object: player
        Preconditions:
          ...
        Commands:
          - add:
              - inv_wood
              - 1
          - if:
              Conditions:
                lt:
                  - ach_collect_wood
                  - 1
              OnTrue:
                - set:
                    - ach_collect_wood
                    - 1
                - reward: 1
      Dst:
        Object: tree
        Commands:
          - remove: true
          - spawn: grass
```

Figure 3.14: GDY for an environment transition for picking up wood.

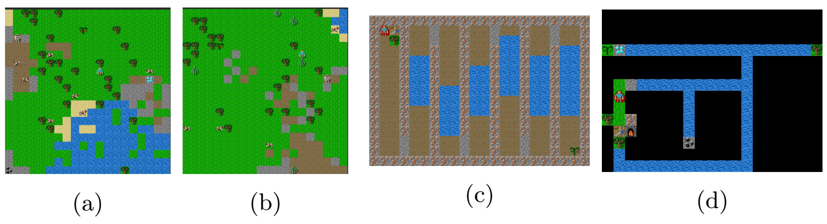


Figure 3.15: Example `EscapeRoom` that are procedurally-generated (a, b) and human-designed (c,d). The agent must collect resources to build tools and structures to reach the goal cherry tree, while surviving the environment.

from Crafter entail harvesting raw resources such as wood and coal in order to build tools like furnaces, bridges, and pickaxes required to harvest or otherwise clear the path of additional resource tiles like iron and diamond.

Mastering this environment presents a difficult exploration problem for the agent: Not only must the agent reach a potentially faraway goal, but it must also learn several subskills required to reliably survive and construct a path leading to this goal. Success in this environment thus requires exploration, learning modular subskills, as well as generalization across PCG levels. Meanwhile, implementing these rich dynamics presents a time-consuming challenge for the researcher. Such a complex environment typically entails knowledge of many disparate modules performing functions ranging from GPU-accelerated graphics rendering and vectorized processes for parallelized experience collection. Further, the researcher must implement complex logic for executing the finite-state automata underlying the environment transitions, as well as that handling the rendering of observations.

GriddlyJS abstracts away all of these details, allowing the researcher to focus exclusively on defining the underlying MDP through a succinct GDY specification. In particular, the researcher can simply define all entities (i.e. Griddly `objects`) present in the game, each with an array of internal state variables, as well as the agent's possible actions. Then, all transition dynamics are simply established by declaring a series of local transition rules (i.e. Griddly actions) based on the state of each entity in each tile, as well as any destination tile, acted upon by the agent's action. For example, after declaring the action of `do` (i.e. interact with an object) along with the possible game entities and their states (e.g. the agent is the `player`, which can be `sleeping`) we can simply define the transition dynamic of receiving +1 wood resource upon performing `do` on a `tree` using the simple sub-block declaration shown in Figure 3.14. More complex dynamics can be implemented by calling built-in algorithms like A* search or nesting Griddly `Action` definitions. Further, arbitrary PCG logic can be easily implemented by writing Python subroutines that output level strings corresponding to the ordering of the level tiles.

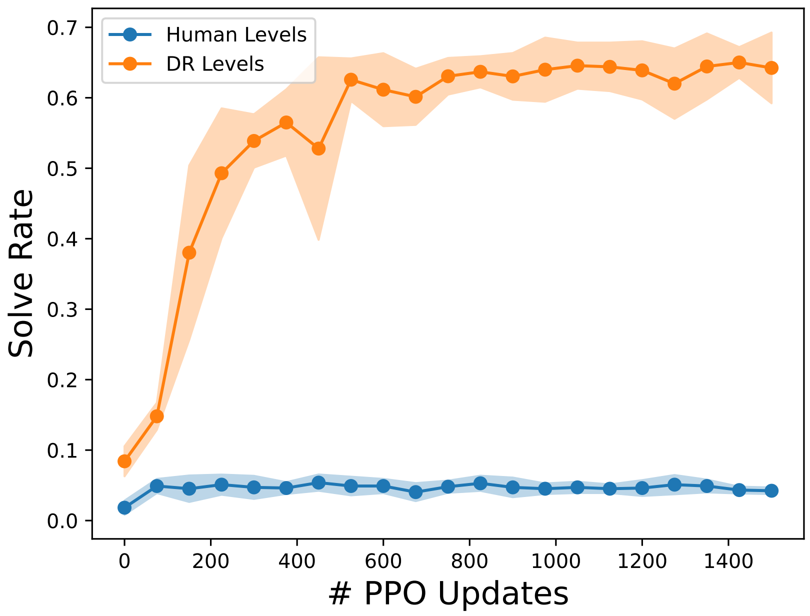


Figure 3.16: Mean and std of solve rate on DR levels (orange) and human-designed levels (blue).

3.12.2 Human-in-the-Loop Level Design

Given the rich design space of the `EscapeRoom` environment, randomized PCG rules defined by the Griddly level generator are unlikely to create challenging levels that push the boundaries of the agent's current capabilities. Rather in practice, designing such challenging levels for such puzzle games rely heavily on human creativity, intuition, and expertise, which can quickly hone in on the subsets of levels posing unique difficulties for a player or AI. Indeed, based on recent works investigating the out-of-distribution (OOD) robustness of RL agents trained on domain-randomized (DR) levels [74, 147], we do not expect agents trained purely on randomized PCG levels to perform well on highly out-of-distribution, human-designed levels, without the usage of such adaptive curricula. However, it can be costly to collect a large set of diverse and challenging human-designed levels necessary to encompass the relevant challenges, and thus, most prior works test on limited sets of human-designed OOD levels.

GriddlyJS allows us to quickly assess OOD generalization on a large number of human-designed levels. With its visual level editor and interactive, browser-based control of agents, we can rapidly design and iterate on new and challenging levels. In particular, we created 100 diverse environments in roughly eight hours, many featuring environment and solution structures that are highly unlikely to be generated at random. These levels can be seen in Figure 3.17

We then use PPO to train a policy on domain-randomized `EscapeRoom` levels. We checkpoint the policy at regular intervals in terms of the number of PPO

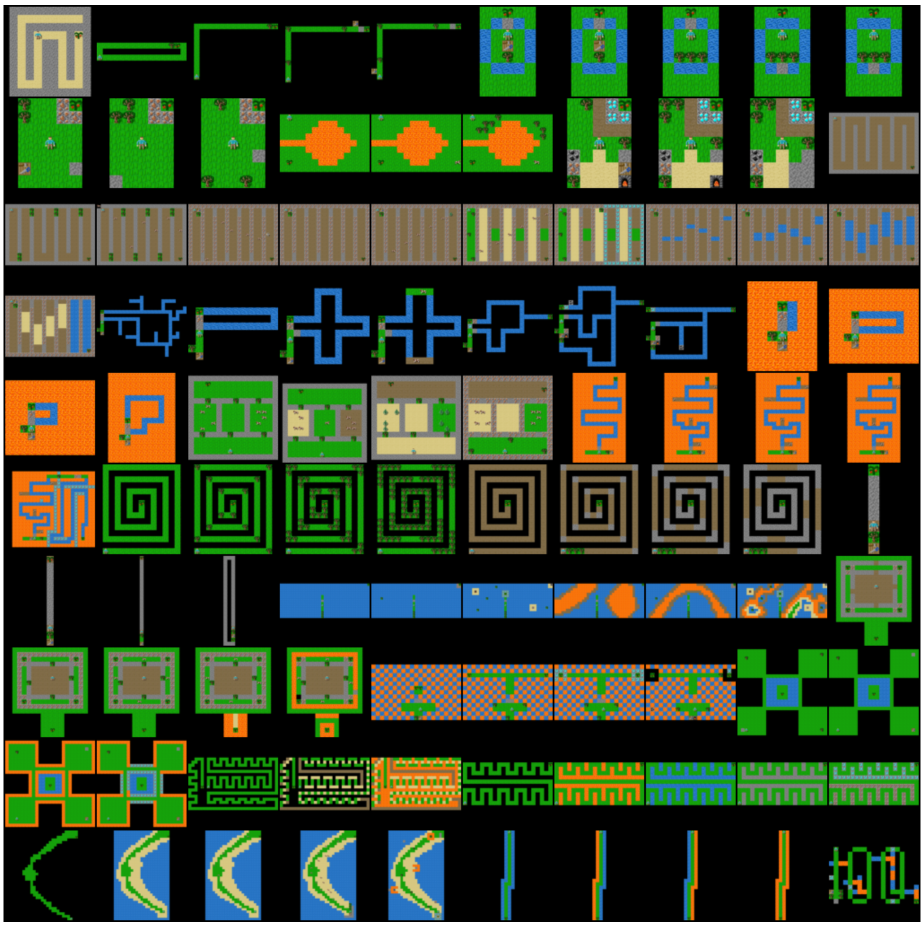


Figure 3.17: All 100 human-designed EscapeRoom levels, made using GriddlyJS.

updates and evaluate the performance of each checkpoint on all 100 human-designed levels, as well as 100 DR levels. We see in Figure 4.6 that, throughout training, the resulting policy solves DR levels at a significantly higher rate than human-designed levels, highlighting the distinct quality of human-designed levels. We also note here that the human-designed levels are exponentially unlikely to fall inside the distribution of the generated levels. This highlights a significant limitation of PCG, which does not exist with levels generated by a human. Generating complex puzzle levels using PCG methods is an ongoing area of research. Tuning generators to create unique levels often results in levels that are invalid or unsolvable, and vice versa [275, 314, 265, 81, 74]. The full details on our choice of model architecture and hyperparameters is provided in Appendix 3.13.

Furthermore, as GriddlyJS loads TF.js models for policy evaluation and visualization directly inside the IDE, human-in-the-loop level design can be performed in a closed-loop, adversarial manner: The policy, first trained on DR levels, is successively evaluated on additional sets of human-designed levels, the most challenging of which are added to the agent's training set, thereby robustifying the agent's weaknesses. Given the success of these methods and that of recent adversarial adaptive curricula methods for RL [148] in producing robust models, we expect human-in-the-loop adversarial training to lead to similarly significant gains in policy robustness. Importantly, by developing an RL environment in GriddlyJS, this mode of training is immediately made available to the researcher. GriddlyJS enables TF.js policies and environments to be directly published on the web, thus allowing such adversarial methods to be tested at high scale, potentially leading to highly robust policies and collecting unique datasets of adversarial levels useful to future research in generalization and the emerging field of unsupervised environment design [74].

3.12.3 Recording and Controlling Trajectories

GriddlyJS makes it easy to record trajectories for any level directly inside the IDE, enabling a wide range of downstream use cases. For example, such recorded trajectories can be associated with human-designed levels during human-in-the-loop adversarial training to ensure solvability. Further, such recorded trajectories naturally serve as datasets for offline RL and imitation learning—especially useful for more complex multi-task or goal-conditioned environments where it may be important to ensure the dataset has sufficient coverage of the various tasks or goals [219].

Moreover, the interactive control feature in the presence of a loaded TF.js policy enables the study of human-AI interaction, in which the agent may hand over control to a human when the policy is uncertain. Further, as levels can be edited directly inside the IDE, GriddlyJS allows researchers to perform a controlled evaluation of policy adaptation to environment changes that occur mid-episode.

Deep Learning Framework Support

GriddlyJS supports any deep learning model that can be converted into the ONNX format [207]. This includes many popular frameworks such as **PyTorch**, **Tensorflow**, **JAX**, **Caffe**, and **Chainer**. Once converted to the ONNX format, these models can be converted to TensorflowJS and used in the debugging view.

As our experiments in section 3.12 are trained using PyTorch, we include example scripts to convert these models to ONNX and then to TensorflowJS. These scripts and documentation on how to load and use the converted models can be found at:

`https://github.com/GriddlyAI/escape-rooms#using-checkpoints-in-griddlyjs`

3.13 Experimental Details and Hyperparameters

Table 3.5 summarises the hyperparameters we chose to sweep. Other hyperparameters while sweeping were those shown in Table 3.6.

Table 3.6 summarises our final hyperparameter choices for our PPO agent. The final choice was made by taking the highest average (calculated across the seeds) level completion rate.

Table 3.5: Hyperparameter sweep values

Parameter	Values
λ_{GAE}	0.65, 0.8, 0.95
Adam learning rate	5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4
Student entropy coefficient	0.2, 0.1, 5e-2, 1e-2, 5e-3, 1e-3
Seeds	0 1 2 3 4 5 6 7 8 9

3.13.1 Architecture

We use the PPO implementation from CleanRL [140] with the *ImpalaCNN* [86] architecture as this is commonly used with grid-world environments.

Table 3.6: Hyperparameters used for training the PPO model.

Parameter	Values
γ	0.99
λ_{GAE}	0.95
PPO rollout length	128
PPO epochs	4
PPO minibatches per epoch	4
PPO clip range	0.2
PPO number of workers	256
Adam learning rate	1e-3
Adam ϵ	1e-5
PPO max gradient norm	0.5
PPO value clipping	yes
Return normalization	no
Value loss coefficient	0.5
Student entropy coefficient	0.05

3.13.2 Training And Evaluation

All training and evaluation episodes are limited to 500 steps. The agent receives no penalty for reaching this limit. We trained our models with 10 different seeds for 50 million environment steps. All training is performed using our modified Crafter level generator as described in the next section. All results are averaged across these 10 seeds. All code for our experiments and descriptions on how to use the training and evaluation scripts can be found in the escape-rooms repository: `https://github.com/GriddlyAI/escape-rooms`

3.13.3 Modified Crafter Environment

Griddly's GDY format contains a restricted set of commands that allow complex mechanics to be realized. However, when translating from many environments into GDY format, there are some caveats that may mean behaviors are slightly modified from the original versions.

Grafter

Grafter Github repository: `https://github.com/GriddlyAI/grafter`

Before generating the Escape Room environments, Crafter was first translated directly to GDY to create as close replication of the original environment as possible. This replication (Nicknamed Grafter) had several features that could not be directly translated. These translation artifacts between the environment implementations are explained below:

Chunk Balancing The spawning and despawning of Non-player characters (NPC) i.e zombies, cows, and skeletons in order to balance their numbers across the environment is not possible using the current features of Griddly, so objects are only spawned at the start of the episode. Defeating NPCs removes them permanently from the environment.

Day and Night Cycles Changing the brightness of pixels in order to simulate a day and night cycle is relevant only when pixel observations are being used. Griddly supports several other observation spaces where day and night cannot be easily modeled, such as `Vector` and `Entity`. Day and night are still implemented as part of the `Sprite2D` observations, but the associated behavioral changes for NPCs are not present.

Chasing Behaviour Zombies and skeletons use Griddly's built-in A* pathfinding implementation, whereas zombies and skeletons in Crafter use a simple rule-based method.

Observation Spaces All observation spaces are configured to contain the same information as the original crafter environment and have equivalent observability dimensions.

Sprite2D observers configured to be the same as the original Crafter environment, the inventory display, and day/night cycle are produced by a custom shader. In this implementation, the Voronoi pixel noise used in the original environment is omitted.

Vector observers contain a 7x9x51 (WxHxC) observation space, where the channels C represent object types, orientations, playerIds, and the set of global variables which represent the inventory, which are repeated across the height and width dimensions.

Entity observers contain a list of features for each object type in the 7x9 space around the agent and additionally include a *global* entity which contains the inventory variables.

Multi-Agent Support is naturally introduced as part of the features that come with Griddly, agents gain an additional achievement if they defeat other agents. The number of agents that are spawned in the environment is configurable in the GDY.

Domain Randomization

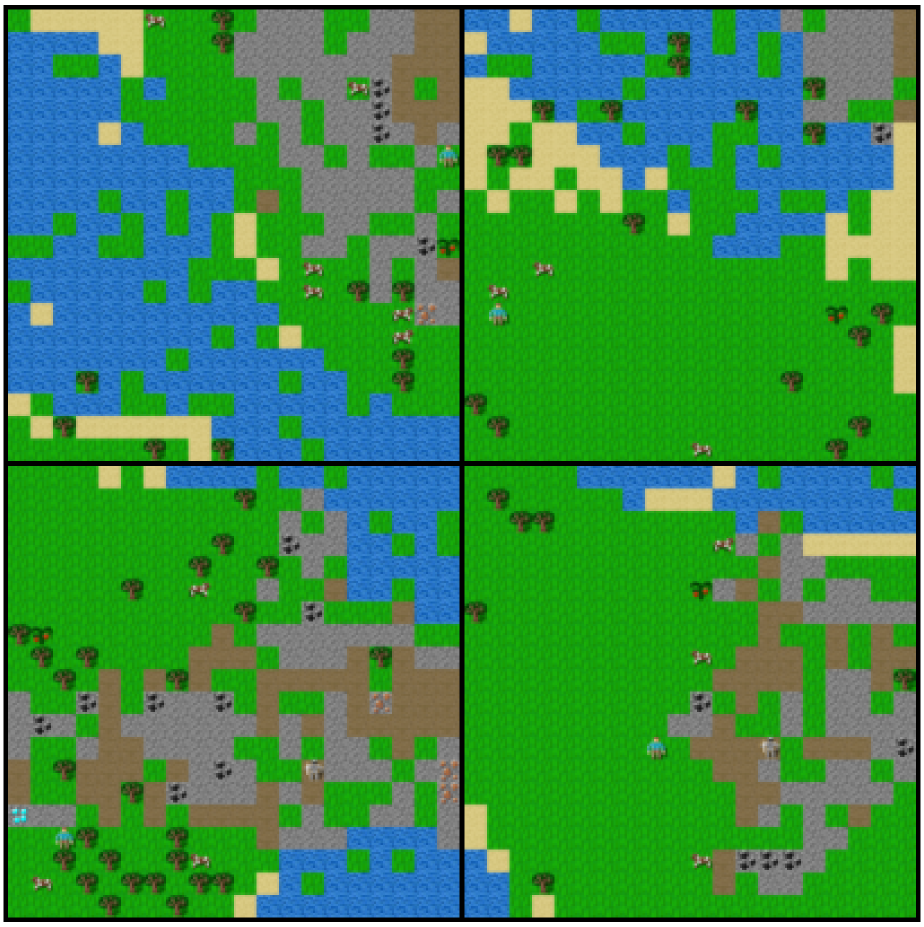


Figure 3.18: Example escape rooms generated by the Domain Randomization generator.

The domain randomization algorithm we use is a modified version of the open-simplex-based level generator from the original Crafter environment. While the structure of the levels is generally the same as those in Crafter, we also make sure that we add a single "cherry" tree goal to each level. In most cases, the cherry tree can be reached by traversing land, but occasionally there may be levels where more complex strategies are required such as chopping trees or building bridges to get to the island where a tree exists. We show examples of the DR levels in Figure 3.18

Escape Rooms

Escape Rooms Github repository `https://github.com/GriddlyAI/escape-rooms`

Table 3.7: Flattened Action Space

Action	Values
No-Op	0
Move Left	1
Move Right	2
Move Down	3
Move Up	4
Interact With Object	5
Place Stone	6
Place Table	7
Place Furnace	8
Make Wood Pickaxe	9
Make Stone Pickaxe	10
Make Iron Pickaxe	11

To make an escape room, firstly we needed a method of **escape**. To do this we repurposed the mechanic of eating a **plant** object. We added a termination condition so that if the *eat plant* goal is achieved, the episode ends and the agent receives a reward of 10.

There are also several features in Grafter that were not required in the Escape Room environment:

Plants As reaching and eating a plant (cherry tree) is now being used as the goal state, the mechanics for collecting, planting and ripening of the trees was removed. Plants are spawned in the *ripe* state and remain that way until the agent collects them and subsequently ends the episode.

Agent Survival The mechanics for surviving in the environment, such as requiring food, water, energy and maintaining health levels are unnecessary complexities for the escape rooms and limit the possible challenges that can be built. The same reasoning is applied to zombies and skeletons which are not required. Removing these mechanics also removes the need for certain actions such as sleeping. Additionally, the day/night mechanics were removed entirely.

Swords Similar to the reasoning behind survival mechanics, we decided that combat with zombies/skeletons was an unnecessary complication, therefore the mechanics for building weapons were not required. This also simplified the action space as is shown in table 3.7.

Reward Shaping In Crafter the agent is rewarded depending on their current health level, as we are not using any health or survival mechanics this reward scheme is ommitted. All achievements still give a single reward of 1 for the first time they are encountered. The exception being the *eat plant* achievement which gives a reward of 10 for completing the escape room.

3.14 Solution Trajectories

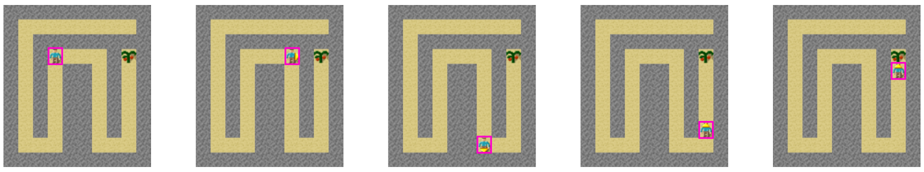


Figure 3.19: **Level 1.** In this simple level, the agent must solely navigate the maze to reach the goal cherry tree.

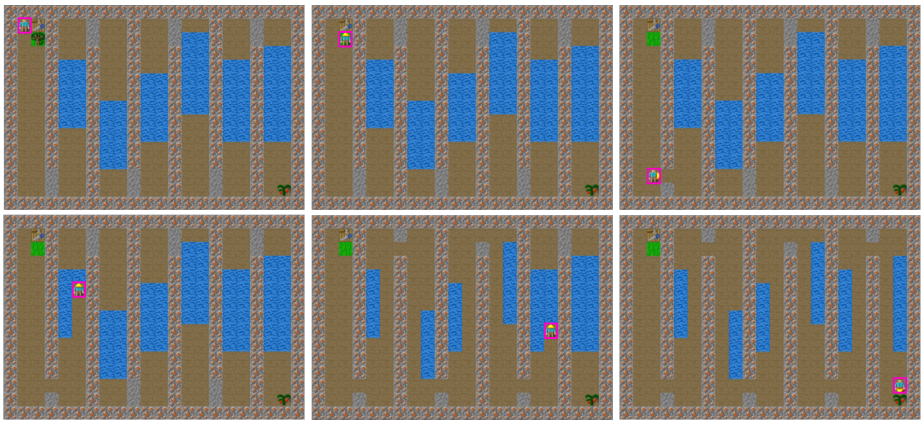


Figure 3.20: **Level 30.** The agent begins in the top-left corner, where it must first collect wood and build a wooden pickaxe using the work table. With this tool, the agent can pick up stones to clear the path, as well as successively place and pick up stones over the water to create a walkable path, making it possible to reach the goal.

To demonstrate the ease of creating custom levels and recording trajectories with GriddlyJS IDE, we create a set of 100 hand-designed levels of the Crafter-based `EscapeRoom` environment. All 100 levels are visualized in Figure 3.17. The levels are feature distinct challenges for the agent. For each level, we include a solution trajectory generated by a human player using the recording feature inside the IDE. Griddly stores trajectories as simply a list of actions taken, along with a string representation of the level or a specific seed that allows the level generator to deterministically reset to the recorded level. We visualize key

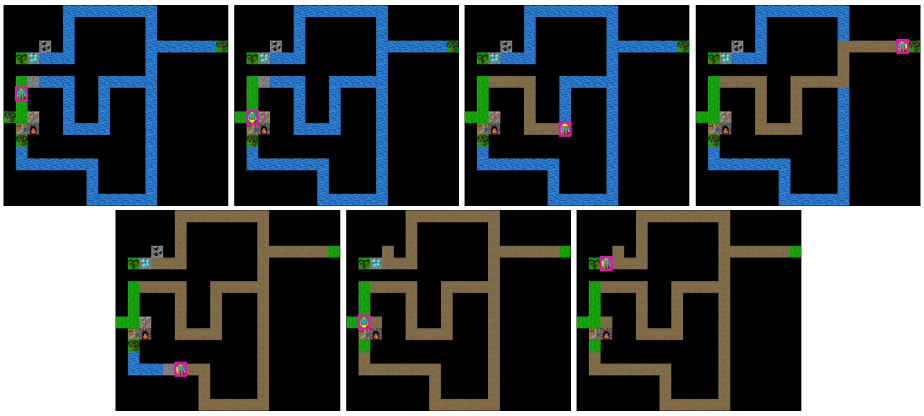


Figure 3.21: **Level 37.** The agent starts on the left and must first collect the wood and go to the work table to build a wooden pickaxe, with which it can collect the stone above. The agent must then successively place and remove this stone over the water to make the water walkable paths, making sure to collect the remaining two pieces of wood. Returning to the work table, the agent can then build a stone pickaxe, collect the coal at the top of the level, return to the work table and furnace to build an iron pickaxe, with which it can clear the diamond blocking the goal.

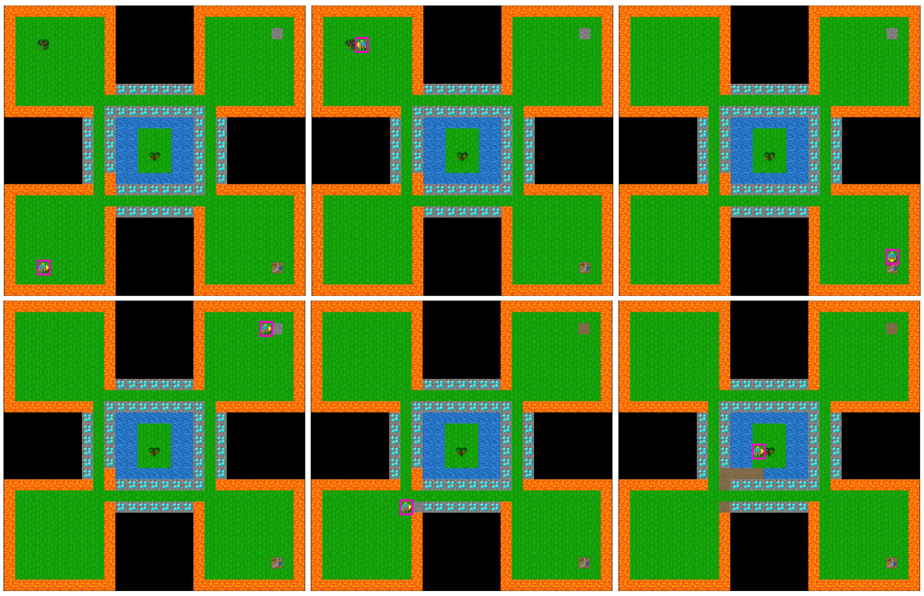


Figure 3.22: **Level 82.** The agent begins in the bottom-left corner and then must visit top-left corner to collect wood, visit the work table in the bottom-right to create a wood pickaxe, collect the stone in the top-right corner, and then successively place and remove the stone over the lava to create a walkable path through the lava corner in at the bottom-left of the central diamond-bordered square to reach the goal.

frames (left to right, top to bottom) from expert trajectories for a diverse subset of the 100 hand-designed `EscapeRoom` levels in Figures 3.19 through 3.24. In

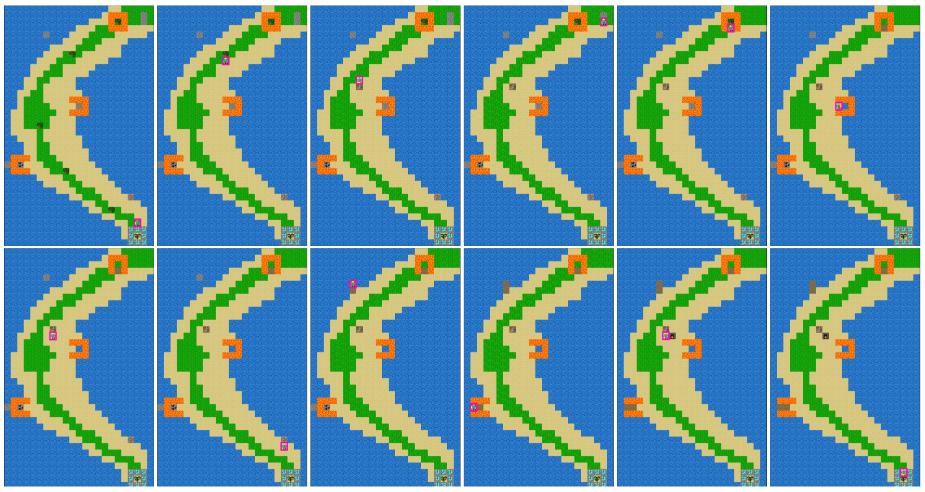


Figure 3.23: **Level 95.** The agent begins at the bottom of the island. It must collect the wood across the island to build a work table and then a wooden pickaxe, with which it can collect stone —which requires building a path to the stone in the water by placing and removing stone in the water. The agent must then return to the work table to build a stone pickaxe, with which it can collect the iron in the bottom-right. With the iron, the agent must return to the work table to create a stone pickaxe, which can be used to collect the coal, clearing the way to also place and remove stone over the lava to collect the final piece of stone. The agent must then return to the work table to build a furnace and then an iron pickaxe, with which it can use to clear the diamond blocking the goal.

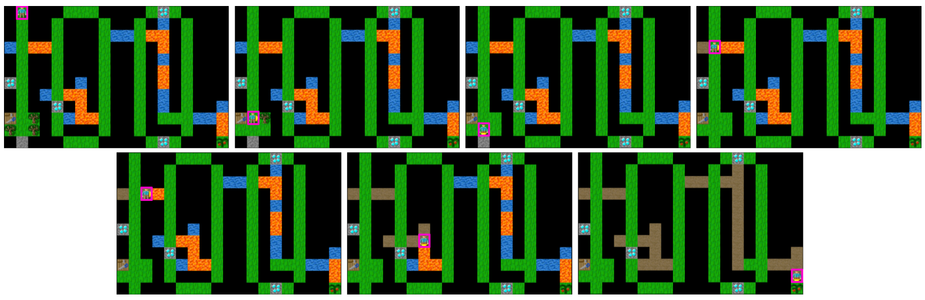


Figure 3.24: **Level 100.** The agent starts at the top-left, and must first move down to collect the wood and build a wooden pickaxe to collect the stone. With the stone, the agent must create walkable area over each of the crevices along the path in order to allow it to properly face the lava tiles, so that the agent can then place and remove the stone over the lava to create a walkable path towards the goal. This level creates difficulty by exploiting how moving towards water does not result in episode termination, while turning into lava does.

each frame, the agent is highlighted with a magenta bounding-box for clarity.

3.15 Conclusions

We introduced GriddlyJS, a fully web-based integrated development environment that streamlines the development of grid-world reinforcement learning (RL) environments. By abstracting away the implementation layers responsible for shared business logic across environments and providing a visual interface allowing researchers to quickly prototype environments and evaluate trained policies in the form of TensorFlow.js models, we believe GriddlyJS can greatly improve research productivity. GriddlyJS enables human-in-the-loop environment development, which we believe will become a major paradigm in RL research and development, allowing for the measured design of higher quality environments and therefore training data. Moreover, such approaches (and therefore GriddlyJS) can enable new training regimes for RL, such as human-in-the-loop adversarial curriculum learning.

Of course, our system is not without limitations: GriddlyJS does not currently persist user-generated data on a dedicated server, though we plan to support this functionality in the future. Additionally, although environments are rendered in the browser, pixel-based observation states are not currently supported. Moreover, training must still occur outside of GriddlyJS, a bottleneck that is mitigated by the fact that GDY files can be so easily loaded into Griddly for Python, and most model formats are easily converted to the TF.js format as highlighted in Appendix A.1.4.

As a proof-of-concept, we used GriddlyJS to rapidly develop the `EscapeRoom` environment based on the complex skill-based dynamics of Crafter, along with 100 custom hand-designed levels. We then demonstrated that an agent trained on domain-randomized levels performs poorly on human-designed ones. This result shows that PCG has difficulty generating useful structures for learning behaviors that generalize to OOD human-designed levels, thereby highlighting the value of GriddlyJS's simple interface for quickly designing custom levels. Additionally, we believe GriddlyJS's web-first approach will enable more RL researchers to share their results in the form of interactive agent-environment demos embedded in a webpage, thereby centering their reporting on rich and engaging research artifacts that directly reproduce their findings. Taken together, we believe the features of GriddlyJS have the potential to greatly improve the productivity and reproducibility of RL research.

Chapter 4

Environment Interfaces

One of the main requirements of Griddly is that it can support many possible methods of interacting with environments, for example, single-agent, multi-agent, and RTS-style games where an agent can issue commands to control multiple units in parallel.

In some environments, there may be action spaces that are conditional or hierarchical, where an action is formed of multiple components. One example of this would be an environment that has two components: the first component consists of the discrete actions "move" and "jump", the second component consists of actions "up", "down", "left", and "right". If the agent selects "move" then the other actions "up", "down", "left", and "right" are available, but if the agent selects "jump" then some options of the second component may be invalid. "jump" + "down" and "jump" + "up", might not be valid actions. Other examples such as StarCraft II[292], μRTS [138] and BotBowl [157] allow control of multiple individual units either by selecting their locations and then issuing commands to those units. Some units can perform certain actions that are not accessible to other units. Furthermore, some of those actions in turn require additional parameters. For instance, selecting a combat unit that can target several potential locations in the game requires specifying them. Moreover, the particular type of combat actions might be tied or dependent on the unit selected.

4.1 Background

Several techniques have been proposed to handle this kind of action space and in the rest of this section, we will attempt to cover the most well-known methods.

Flat Action Spaces

The most common method for dealing with conditional or hierarchical action spaces is to "flatten" the action space. Flattening consists of enumerating all possible combinations of actions and then converting this enumeration into one large list of distinct actions [163]. In our simple example above, we have an action with two components. The first component is to select "move" or "jump" and the second component is to select the respective direction: **up**, **down**, **left**, and **right**. We have already stated that there are two invalid actions, **jump down** and **jump up**. this leaves us with 6 possible combinations. The flattening procedure would therefore change the action space into the following 6 distinct actions: **move up**, **move down**, **move left**. **move right**, **jump left**, **jump right**.

This method works in environments that are relatively simple, but in environments that have large numbers of possible values in several components, the combinatorial complexity means that the eventually flattened space can be multiplicative in nature. In RTS games such as μRTS [138], the components of actions can include x and y coordinates. In small maps or levels, a flattened representation may consist of a few hundred possible actions, but as larger maps are used, the growth of the action space is multiplicative. In particular in μRTS, "attack" actions may have start and end coordinates, meaning that a flattened action space scales with respect to x^2y^2. A 30×30 game would require a policy with almost 1 million outputs. This is also ignoring the fact that there are many other actions other than "attack" that are present in the environment.

Parameterised Actions

Parameterized action spaces commonly take the form of an action a made from two components c_0, c_1 where the first component is a *type* of action and the second is a *parameter*. In [191], this action space shaping strategy was applied in the *RoboCup 2D Half-Field-Offense* environment to beat the state-of-the-art hard-coded bots. The first action component defines whether the agent will *dash, turn, tackle* or *kick*. The second component defines continuous parameters for each of these actions. Four sets of parameters are used, however, only one of them is used at each time step depending on the action type selection. This leads to many redundant outputs of any policy.

In larger environments such as RTS games, requiring parameters for every action quickly becomes infeasible as the number of action types increases. To contextualize the effect this can have for the size of the policy representation consider the example of *BotBowl*. The game contains 17 action types that require an x and y position parameter. If we proceeded to parameterize the action space, 17 sets of x,y positions would need to be predicted at each time step. From the point of view of an RL agent, the problem is exacerbated if we consider that the policy would have to specify each combination of x and y position. In a traditional *BotBowl* map (25×5) this would lead to a policy that requires $17 + 17 \times 25 \times 15 = 6,382$ logits (i.e. unnormalized scores) to parameterize these actions. The number grows quadratically with the map size, a 30×30 map, for example, would require $17 + 17 \times 30 \times 30 = 15,317$ logits.

Auto-regressive Actions

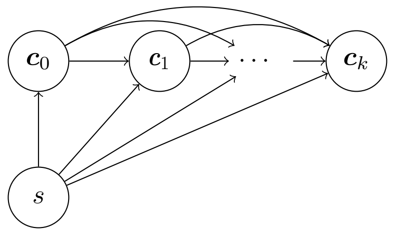


Figure 4.1: An auto-regressive policy can be graphically represented as a directed acyclic graph where we can illustrate the dependency of a component c_k on the previous components $c_{<k}$.

An alternative comes from reflecting on the structural relations that exist in complex action spaces. For example, when comparing all the potential actions that an agent could choose to enact, not all of them will belong to the same level of abstraction. Some actions will be more self-contained, whereas others may need a group of actions to be properly contextualized.

It is possible to capture this concretely by representing a policy in a more expressive manner. In [292] the authors suggest an auto-regressive model of the form:

$$\pi(a|s) = \pi(c_0, \ldots, c_k|s) = \prod_{k=0}^{K} \pi(c_k|c_{<k}, s) \tag{4.1}$$

To create the action a, the agent samples multiple sub-actions or components c_k that depend on the previous $c_{<k}$ choices (illustrated in Figure 4.1). [292] explores the usage of conditional policies within the context of StarCraft II. However, they relax the constraints imposed by the auto-regressive model opting for a policy $\pi(a|s) = \prod_k^K \pi(c_k|s)$. In [293], the approach is extended substantially as the architecture considers a conditional policy that captures the context of previous actions through different embeddings. An *invalid action masking* scheme, as described in Section 3.1.2, is also used to prevent the agent from selecting actions that are invalid or cannot be performed in the current

state.

Entity Actions

Entity actions, work in conjunction with *entity observation spaces*, and output actions per-entity type [303]. Each entity in *entity observation spaces* has a corresponding object embedding head, the output also requires an output or policy head per object type. The fact that different action heads are used per object type means that the number of redundant outputs and required masking is significantly reduced. Additionally, as objects are observed and policies are predicated on a per-entity basis, there is a reduced need for action spaces where objects have to be selected through coordinates, further reducing the complexity of the action spaces.

4.2 Conditional Action Trees

In this chapter, we propose a *Conditional Action Tree* as a paradigm to generalize several of the action space configurations introduced in section 3.1.3. *Conditional Action Trees* can be used to describe action spaces in a way that naturally reduces the required policy model output size whilst also allowing action parameterization and action reduction using invalid action masking. We show how many action spaces frequently found in single, multi-agent, and Real Time Strategy (RTS) games can be described using *Conditional Action Trees*. We also show that agents with access to *Conditional Action Trees* as part of their state observations can learn high-performing policies.

We present several experiments where we purposefully modify the action space of a game environment to include several increasingly more complex features, whilst keeping the observation space and game mechanics consistent. In these experiments, we show that agent operating with *Conditional Action Trees* maintains the performance of those operating with common action space constructions while significantly reducing the number of outputs, or *logits*, required to furnish the policy distribution.

In addition to these experiments, we also perform several ablation studies to show various possible modifications to the *Conditional Action Tree* formulation and how they can affect training.

The results suggest that the *Conditional Action Trees* could offer an alternative to generically handle complex combinatorial action spaces with multiple components. This work is made possible by using the Griddly Framework described in previous sections. All Griddly environments support conditional action trees as their default action interface.

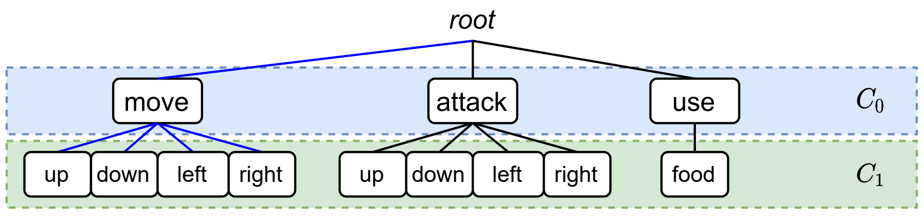


Figure 4.2: An action tree consisting of nine possible actions and two components $C_0 = 3$ and $C_1 = 4$. The possible actions are "move" or "attack" in any of the four directions, "attack" in any of the four directions, and finally "use" a food item. *Move*, *Attack* and *Use* cannot be performed at the same time.

Conditional action trees (CAT) offer a generalization of discrete action spaces to provide an interpretation of action selection as the process of traversing along a chained sequence of action components with different levels of dependency. To complete the characterization of a *Conditional Action Tree* we first need to define three main elements: *Action Trees*, *Valid Action Trees*, and finally *Conditional Masking*.

4.2.1 Compatible Action Spaces

For an action space to be compatible with by CATs, they must satisfy a few requirements:

- Action spaces must contain one or more discrete components as described in 4.2.2
- Environments must provide invalid action masks for all components as described in section 4.2.3
- It is not possible to provide invalid action masks for continuous components, so continuous actions are not compatible.

4.2.2 Action Trees

We start by formulating a single action as a list of a fixed number of components $a = \{c_0, c_1 ... c_n\}$, where $c_k \in C_k$. That is, each component takes a value from a set of possible elements. Actions in the same component level are mutually exclusive. For example, *move left* and *move right* must be options within a single component C_k. The possible values of C_k are determined first, by the specification of the environment, and second by the values of previous selections $C_k = f(c_0, c_1, ..., c_{k-1})$.

These restrictions naturally allow the components to form a tree structure, where a path from the root node to any leaf forms the action. An example of an *action tree* is shown in Fig. 4.2. Note that under this specification an environment requiring the agent to specify a single atomic action at each time step results in an *action tree* with a single component, $a = \{c_0\}$. Parameterized action spaces that contain an action type and a discrete action parameter can also be described by action trees with two components, $a = \{c_0, c_1\}$.

Previous work has touched upon the idea of using trees as a formalization of action spaces with multiple components such as in [89], where the tree structure is referred to as a *Hierarchical Action Space.* Other works have used action spaces that are similar to *action trees.* The *Global Action Space* in [137] for example can also be described as an *action tree.*

4.2.3 Valid Action Trees

We define a *valid action tree* as a sub-tree of an action tree at a particular environment state, where the sub-tree nodes correspond to *possible* actions in that state. For example, consider the tree in Fig. 4.2, an agent in a state where no enemies are surrounding it and does not have food in its inventory has a *valid action tree* only consisting of the left-most *move* branch and its children.

In the context of reinforcement learning, the environment provides a valid action tree at each time step. Valid action trees are then used to construct the *Invalid Action Masks* described in Section 3.1.2. These masks index the child nodes available in the full action tree.

4.2.4 Conditional Masking

The two-step method of masking in [139] can be generalized to an n-step masking method when the environment provides a valid action tree as described in Section 4.2.3. We refer to this generalization of action selection and masking as a *Conditional Action Tree (CAT).*

A CAT is constructed by adding a mask at each node of a valid action tree, defining which child nodes of the complete action tree are available. An action is constructed by starting at the root node of the valid action tree and selecting a child node from the masked distribution. This child node contains the mask to use for the next component. Thus, first the mask is obtained as $\mathbf{m}_{k+1} \sim p(\mathbf{m}_{k+1}|c_k)$, to produce a masked sub-policy to sample a component $c_{k+1} \sim p(c_{k+1}|\mathbf{m}_{k+1}, s)$. This process continues until all action components have been sampled. The full compound policy, as illustrated in Fig. 4.3, is factorised as:

$$\pi(a|s) = p(\mathbf{m}_0) \prod_{k}^{K} p(\mathbf{m}_{k+1}|c_k) p(c_k|\mathbf{m}_k, s) \qquad (4.2)$$

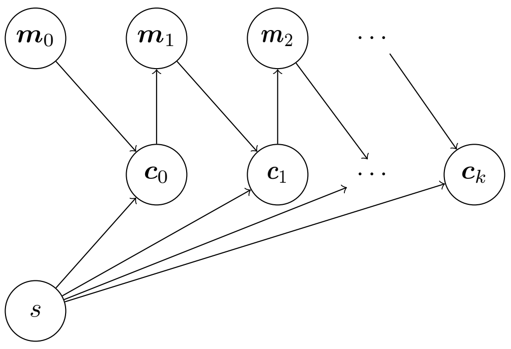


Figure 4.3: A graphical model representing the policy as a joint distribution of masks m and components c. In a CAT, a component c_0 is sampled from the options allowed by the mask $\mathbf{m}_0$. The next mask $\mathbf{m}_1$ depends on c_0, which constrains the next possible component c_1. The process is repeated until all c_k components have been sampled.

4.3 Actor-Critic with Conditional Action Trees

4.3.1 IMPALA

The description of the action spaces provided by CAT is naturally agnostic to the choice of the RL algorithm. We examine this perspective within the context of IMPALA, an actor-critic-based framework introduced in [86]. Unlike A3C [197] or other similar distributed approaches where the agents share their gradients, IMPALA considers the acting and the data collection as independent of the learning step. That is, it separates the learners who are in charge of computing the gradients and sharing the most recent parameters from the actors whose role is to execute a policy, only sharing back with the learners the observations gathered during an episode.

4.3.2 V-trace and masking

As an actor-critic, IMPALA learns $V_\theta(s)$ parameterized by θ to be used as part of the baseline, and a policy π_ϕ parameterized by ϕ. Each actor executes their own policy μ by retrieving the latest policy π from the learner. Meanwhile the learner updates continuously the parameters θ and ϕ. As the process occurs in parallel and in a decoupled manner, there will be a discrepancy between the policy μ from an actor and π. Namely, the trajectories $(s_t, a_t, r_t \dots)$ collected by an actor come from a policy μ that has become obsolete with respect to π. IMPALA proposes to address these off-policy corrections by introducing a v-trace target,

$$v_t = V(s_t) + \sum_{i=t}^{t+n-1} \gamma^{i-t} (\prod_{j=t}^{i-1} u_j) \delta_i V \tag{4.3}$$

where $\delta_i V$ corresponds to a temporal difference term,

$$\delta_i V = \rho_i (r_i + \gamma V(s_{i+1}) - V(s_i))$$

The v-trace adjusts the weight of the contributions provided by the actors through the presence of two truncated importance sampling weights $\rho_i = \min(\overline{\rho}, \frac{\pi}{\mu})$ and $u_j = \min(\overline{u}, \frac{\pi}{\mu})$. Thus the second part of the v-trace target acts as a correction term. For example assuming $\overline{p}$ and $\overline{u} \geq 1$, if $\mu > \pi$ the learner would down-weight the observations and actions followed by the actor. Intuitively, if this ratio tends towards a low number, it indicates that the policies have diverged significantly. The extent to which more recent $\delta_i V$ affect the update of a previous V is captured by the product of $u_{t:i-1}$ where $\overline{u}$ serves as a hyperparameter controlling the convergence speed towards V. In turn, ρ determines to which V we converge. A $\overline{\rho}$ close to 0 leads convergence towards a V^μ as the correction term becomes negligible in the v-trace target.

It is important to note that for CAT we do not just consider a single set of importance sampling weights $\{\rho, u\}$ but instead we must account for multiple corrections dependent on the various sub-policies such that $\rho_{k,i} = min(\overline{\rho}, \frac{\pi(c_k|m_k,s)}{\mu(c_k|m_k,s)})$ and $u_{k,j} = min(\overline{u}, \frac{\pi(c_k|m_k,s)}{\mu(c_k|m_k,s)})$ for a sub-policy k. Moreover, we must synchronize the masks applied to c_k in both π and μ. Similarly, for updating the policy parameters ϕ we adapt the policy gradient loss function to consider the inclusion of the masks and to propagate the gradients to all sub-policies:

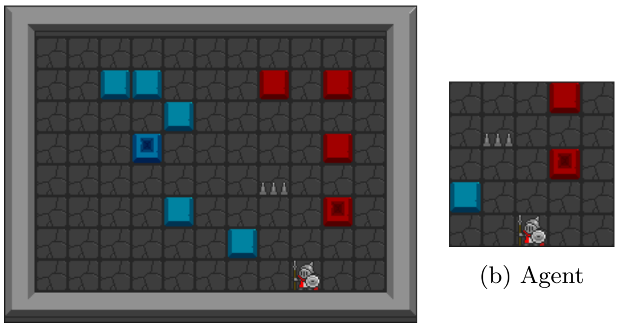
$$J = \rho_{k,i} \nabla_\phi \log \pi_\phi (c_k|m_k, s)(r_t + \gamma v_{t+1} - V_\theta(x_t))$$

4.4 Experiment Setting

4.4.1 The "Clusters" Game

We perform our experiments in the *Clusters* environment provided by Griddly [29]. *Clusters*[1] is a game in which coloured *boxes* must be clustered together in specific locations defined by the environment level. The environment contains five levels with a set of movable coloured *boxes* and a single fixed-position *block* of each colour. The agent receives a reward of $+1$ each time it pushes a coloured *box* against a fixed location *block* of the same colour. When a coloured

[1] `https://griddly.readthedocs.io/en/latest/games/Clusters/index.html`


(a) Global

(b) Agent

Figure 4.4: An example of a level in the Clusters game, showing (a) the entire game and (b) the viewpoint of the agent.

box is pushed against its respective *block*, it becomes a *block* itself. If all *boxes* are converted to *blocks* the episode is completed successfully. Some levels also contain *spikes* which give the player a negative reward (-1) and terminate the episode if the agent or any *boxes* collide with them.

The observation space of the agent consists of a 5×5 grid where the agent itself is situated at the center-bottom of the grid as shown in Fig. 4.4. Each cell of the 5×5 grid contains 10 binary values describing whether an object is present in each cell. The 10 objects are as follows: three (red, green, blue) coloured *boxes* and three associated *blocks*, *walls*, *spikes*, the agent and finally a *broken box* which only appears in the final state of an episode if a coloured *box* is pushed against *spikes*.

4.4.2 Action Space Variations

By default, the agent's movement is restricted to moving forward one position, or rotating ± 90 degrees every step. *Boxes* are "pushed" by the agent when the agent attempts to move into the cell occupied by the *box*.

In our experiments, we modify these action spaces to make it increasingly more complex whilst keeping the game mechanics, observation space and reward scheme consistent. This allows us to test the Conditional Action Tree formulation on different action spaces with minimal influencing factors. The only significant change we make to the environment across experiments is when we remove the avatar and allow the agent to move *boxes* independently by selecting their x and y coordinates. These action space variations are explained below:

Move (M)

The first action tree variation is the default action space provided by the *Clusters* environment. The action space consists of rotate left, right and move forward. As mentioned in Section 4.2.2, this is equivalent to an *Action Tree* with a single component $a = \{c_0\}$, with $c_0 \in 0, 1, 2, 3$.

Move + Push (MP)

Next we modify the action space to consider that the agent can no longer *push* boxes by simply moving into the location occupied by them. We define a separate *push* action that must be performed in order to move any of the boxes. The *push* action has no effect unless there is a box directly in front of the agent. The *move* action is left unmodified, other than the fact that it can no longer be used to push boxes. As the *move* and *push* actions are mutually exclusive they are confined to the first level in the tree $C_0 = \{0, 1\}$, whilst the second component C_1 contains either the three move parameters or the single *push* parameter.

Move + Push + Separate colours (MPS)

This action space configuration contains the same modifications as the MP variant, however it splits the *push* component into three to account for the separate colours. The agent must select the correct *push* action, depending on which colour *box* it is pushing (i.e. push green, push blue, push red). Similarly to *MP*, the action space consists of two components, but the first one now contains the three different push actions as well as the move action, that is $C_0 = \{0, 1, 2, 3\}$. The second component C_1 remains the same, defining the 4 directions for both move and push.

Move - Agent (Ma)

To make the action space significantly larger we remove the agent and the associated ego-centric partial observability. Thus the input consists of the entire 13×10 grid with the same 10 binary digits per cell. The *boxes* are now moved first by selecting their x and y coordinates and then by issuing the direction where to move it. This action space has three components: $C_0 = \{\text{valid x coordinates}\}$, $C_1 = \{\text{valid y coordinates}\}$ and $C_2 = \{0, 1, 2, 3\}$ referring to the movement directions *up*, *down left* and *right*.

Move + Separate colours - Agent (MSa)

The final and largest action space we consider starts with the same formulation as *Ma*, but separates the colour components in the same way as done in *MPS*.

This results in an action space with four components: x, y, action type and action parameters. An example of a conditional action tree for this space is shown in Fig. 4.5

4.4.3 Baselines

For each of the variations of the action space described in the previous section, we compare against two baselines which are designed to show the benefits and limitations of the CAT paradigm. The baselines modify only the way that the model interacts with the action space in terms of number of logits required. The number of actions and mechanics of the game are consistent.

No Masking

For the first comparison we use the same action components as a CAT but remove the Invalid Action Masking entirely. This means that the component selections are made independently of each other and invalid actions can be selected.

Depth-2

The second comparison also uses a conditional action tree structure, but flattens the action tree to only a depth of two. The separate x and y components (only available in MSa and Ma) are flattened into a single xy component. Additionally the action type and action parameter components are flattened into a single selection. This flattening process was also considered in [163] where multi-discrete actions are flattened into single discrete spaces. Table 4.1 shows the number of logits per-component for all experiments and the equivalent number of logits required in the depth-2 representation.

4.4.4 Masking Ablation

To show that structure of the tree and the resulting conditional masking has an effect on the learning of the policy, we perform an experiment where we relax the conditional masking restrictions and compare it against the fully conditional masking. We relax the conditional masking of the tree by collapsing the masking across the tree breadth-wise, so all masking is effectively a union of all the possible masks at each depth. This method is equivalent to applying a single mask to the entire action space with no consideration for dependencies between action selections. We refer to the relaxed *Collapsed* and full *Conditional* masking options in further sections as CAT_CL and CAT_CD respectively.

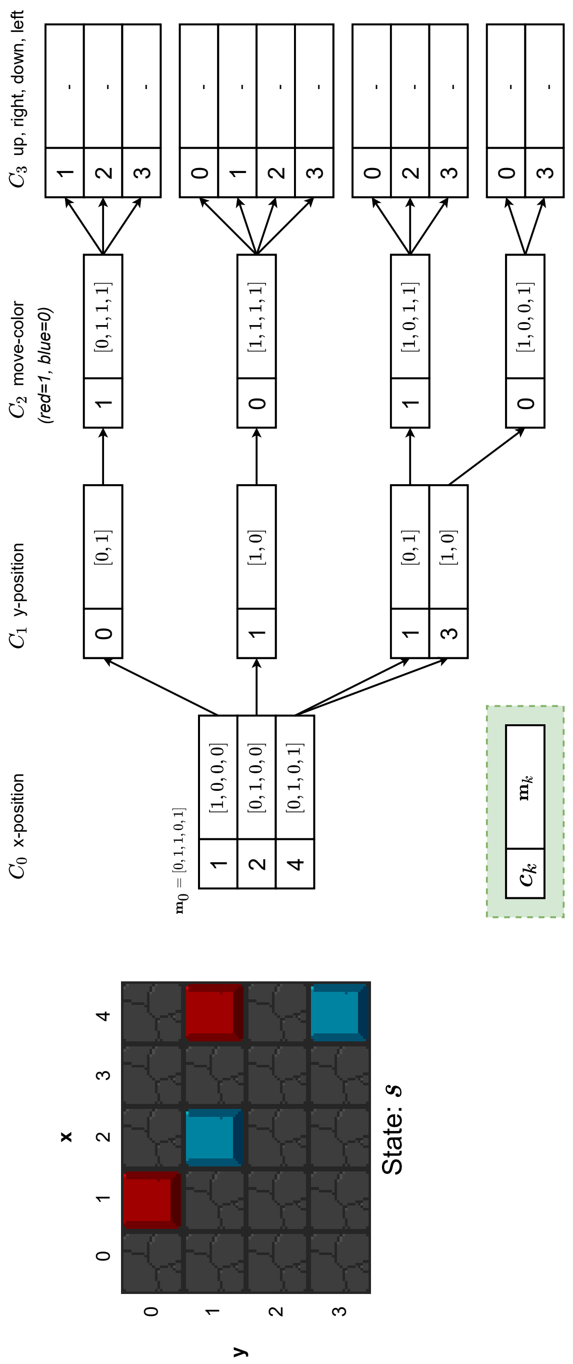

C_0 x-position
C_1 y-position
C_2 move-color (red=1, blue=0)
C_3 up, right, down, left
$\mathbf{m}_0 = [0,1,1,0,1]$
c_k
$\mathbf{m}_k$
State: s
x
y

Figure 4.5: Image of a conditional action tree from a 5x4 *Clusters* level configured with the MSa action space as described in Section 4.4.2. The agent is configured with an action space with 4 components, the agent selects which object to move by its position and the color. It then proceeds to choose which direction to move the *box*. The CAT shown contains the selected action component c_k and the mask $\mathbf{m}_k$ for each possible valid combination of components.

	$\lvert C_0 \rvert$	$\lvert C_1 \rvert$	$\lvert C_2 \rvert$	$\lvert C_3 \rvert$	Total Logits
M	3				3
MP	2	3			5
MPS	4	3			7
Ma	13	10	4		27
MSa	13	10	4	3	30
			Depth-2		
M	3				3
MP	4				4
MPS	6				6
Ma	130	4			134
MSa	130	12			142

Table 4.1: This table shows the number of action components, their sizes in term of number of logits and the total logits needed in the policy output for the action space variations described in Section 4.4.2. We also show the number of logits that are required in the *Depth-2* model.

4.4.5 Model Architecture

We keep the model architecture consistent throughout all experiments as much as possible. The size of the model input observations differs between partially observable agent-based environments (**M**, **MP**, **MPS**) and unit-selecting environments (**Ma**, **MSa**). The partially observable environments have a $5 \times 5 \times 10$ observation space, while for the unit-selecting environments, it is $13 \times 10 \times 10$. Additionally, the final layer in each experiment outputs the number of logits shown in Table 4.1. The model itself contains two convolutional layers with padding 1 and kernel size 3 that up-scales the number of values in each channel to 32 and then 64 respectively, whilst keeping the width and height the same. After these layers, the output tensor is flattened and then passed through two linear layers with 1024 and 512 neurons. We then use a separate actor and critic head. The actor head contains a further two linear layers, first to compress to 256 nodes and then a final layer to output predicted logits. The critic head contains a single layer that outputs the single predicted value.

4.5 Results

In total, we run 4 experiments on each variation of the action space of the *Clusters* game. The four experiments contain the two baselines as previously described and two versions of masking (CAT_CL and CAT_CD).

The first variation **M** provides evidence that the formulation of conditional action trees generalizes to simple action spaces. In this environment, all variations of the action space are almost identical and therefore have similar perfor-

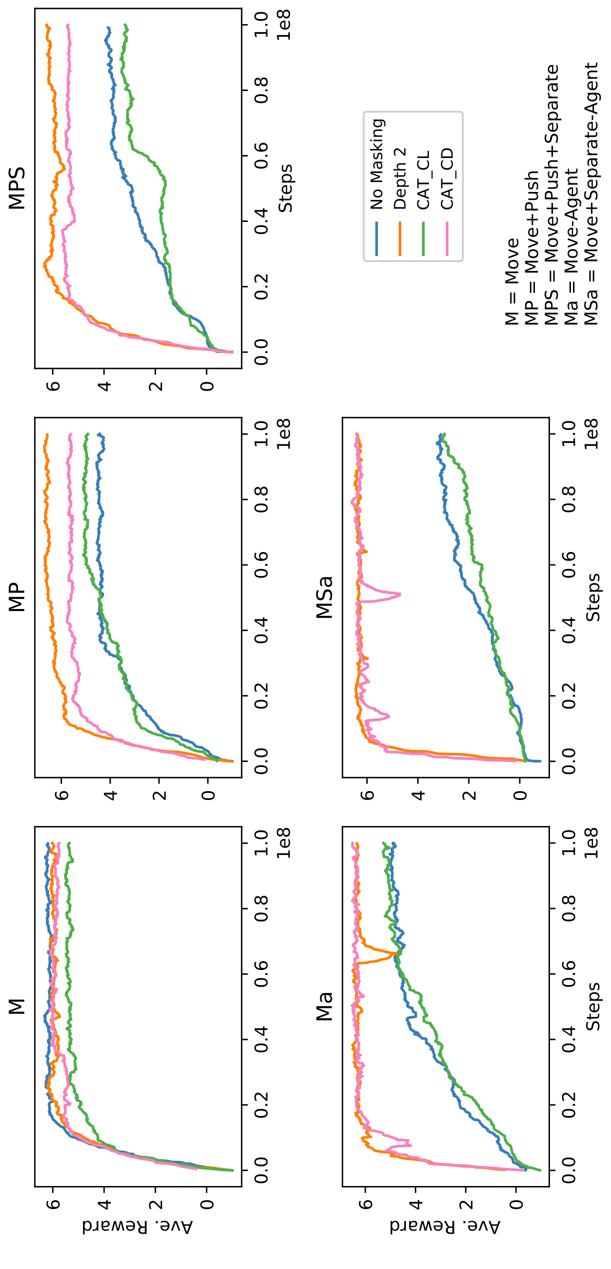


Figure 4.6: The average episode reward over three different starting seeds during training of the 5 different action space variations as described in Section 4.4.2. For each of the 5 action space variations, we compare three policies with the same action tree structure, but different masking methods: No Masking, CAT_CD (conditional) and CAT_CL (collapsed). We also provide a comparison with a model policy that uses an action tree limited to depth 2 as described in Section 4.4.3

mance. Masks in this environment have little effect because only a few actions are ever invalid. **MP** and **MPS** variations begin to show that the fully conditional tree CAT_CD and the depth-2 action tree policies learn faster and plateau at high-scoring policies. Depth-2 action policies in these variations are in fact slightly better performing than the more hierarchical formulation of the Conditional Action Tree, in addition to using one less logit in their policies. The reason for this is that in the **MP** and **MPS** the structure of the associated action tree has a degree of 1 in all of the *push* nodes, making the tree structure redundant for the push actions. In cases like these, where parent nodes have only single children, it is more efficient to flatten these nodes into a single set of children.

Conditional Action Trees excel in the variations with the highest branching factors. **Ma** and **MSa** both require the policy to select an individual unit to perform an action at each time step. As expected, the depth-2 policy and CAT_CD have similar performance as they are both CATs, but CAT_CD splits the x, y selection into separate components, which results in a greater than 4x reduction in the number of logits required by the policy, with no loss in performance.

The results for experiments on all five test environments with *Collapsed* (CAT_CL) masks are also shown in Fig. 4.6. We can see that with *Conditional Action Trees* the full *Conditional* (CAT_CD) masking strategy is important for efficient training, as the *Collapsed* masking strategy performs similarly to the *No Masking* Baseline.

4.6 Discussion

Trees are a useful data structure across many fields of computer science and can provide a natural representation for action spaces. Although the formulation and the experimental setting focused on discrete action spaces, we hypothesize that in principle the formulation can be extended to continuous spaces using Gaussian distributions for each action component [17].

It is important to note that the degree of subtrees in a CAT should be taken into consideration when deciding on parts of the tree that could be flattened, as this can lead to an unnecessary increase in policy size [204].

The current work presented the CAT formulation in five toy scenarios intended to recreate, with different levels of complexity, the conditions frequently exhibited in various single, multi-agent, and RTS games. Further work will be required to analyze the behavior of the CAT in more complex domains such as μRTS or BotBowl. Part of the current limitation resides in adapting these and other environments to provide *Valid Action Trees*. With a *Conditional Action*

Tree the parameterized part of BotBowl's action space could be reduced from 6392 logits to $25 + 15 + 17 = 57$

Other relevant research on how to handle large action spaces has applied techniques such as evolutionary algorithms [22, 156]. These proposals have also been tested in scenarios that require multiple actions per time step. A naive approach to work with CATs within this context would be to recursively append the tree to its own leaf nodes, re-sampling until a condition specifying the required number of actions is fulfilled.

In its current form, a CAT makes specific assumptions about the conditional dependencies between actions (Section 4.2.4). Following [293], a potential future research avenue is to explore the possibility of modeling more complex dependencies. Namely, contextualizing further the selection of a c_k with an encoding learned from previous components $c_{<k}$.

4.6.1 Auto regressive action spaces

Conditional action trees are similar to the tree-like structure of auto-regressive action spaces, where each action is represented as a branch in the tree. The selection of action components is similar to the traversal through a tree, where a branch is selected at each node based on the previously selected branches. The main difference in conditional action trees is that the branch selection is restricted by conditional masking. All the possible combinations of tree traversals are pre-calculated, and each tree node contains a restricted set of branches. These restrictions are encoded as invalid action masks. Additionally, conditional action trees do not predict each component autoregressively but instead rely on masking during training to learn high-performing policies.

The advantages of conditional action trees are that they only require a single pass through a neural network to produce all of the action components. Auto-regressive models have to make a conditional inference step for each component. This means that in environments with action spaces that have many components, the inference could be much slower due to the multiple steps required for each action.

Auto-regressive actions are much less strict on the requirements of the format of action space, for example, action components could be a combination of continuous and discrete variables.

It is also interesting to note that valid action trees can also be combined with auto-regressive action spaces in order to mask impossible actions. However, this is beyond the scope of our experiments.

4.7 Conclusion

In this chapter, we have proposed a formalization of a tree structure for representing discrete action spaces with any number of components. We have provided the required steps to adapt already existing action spaces to conform to a *Conditional Action Tree*. From a technical perspective, a side effect of imposing a structure to the action space is the reduction of the elements considered by a policy. The experiments showed that this modification does not reduce the sample efficiency during training and achieves comparable performance while resulting in significantly smaller models with less parameters.

As part of this work, Griddly [29] implements built-in functionality for generating *Valid Action Trees*, and we provide all reproducible examples in a GitHub repository [2]. We also provide all training parameters, statistics and videos using Weights and Biases [3]. We encourage the developers of reinforcement learning environments, especially those with large discrete action spaces, to provide *Valid Action Tree* observations in their environments.

[2]`https://github.com/Bam4d/conditional-action-trees`

[3]`https://wandb.ai/chrisbam4d/conditional_action_trees`

Chapter 5

Environment Modelling

Recent research has focused on forward models of games that utilise heuristic methods or using deep neural network architectures. These heuristic models can then be used by traditional planning algorithms, or as part of the architecture in reinforcement learning. Using neural networks in planning algorithms can be difficult, as the accuracy of state observations tends to decrease with the number of steps that are simulated. This results in diminishing efficacy of planning algorithms when larger rollout lengths are used [123]. Recent neural network models also tend to rely on a fixed dimensional observational input to predict the rewards and subsequent states and therefore struggle to generalize to games that may have different sized observational spaces.

Heuristic rule-based algorithms for learning forward models [75] [114] offer high performance when they work, but require human input regarding the form the rules will take.

Recent work on a local approach to learning forward models [76] shares some

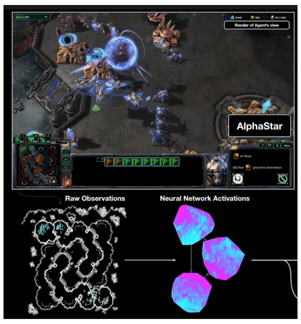


Figure 5.1: Left: A screenshot taken from the game Starcraft II, where an AI (Alphastar) plays professional player "MaNa". Right: A screenshot taken from the game Dota 2, where an AI (OpenAI Five) plays against professional team "OG"

similarities with the Neural Game Engine described in this chapter, in that both methods are able to generalize to levels of a different size than those seen during training. Compared to [76], the Neural Game Engine works directly with pixels rather than tiles, and for many games also does accurate reward prediction.

Grid-based arcade style games, although simple to understand for humans, still present highly challenging environments for artificial intelligence. In this chapter a grid-based game refers to a game that is based on a grid of discrete tiles such as walls, floors, boxes and other game-specific items. A single agent has a set of actions it can perform at each time step, such as movement or interaction with other tiles in the grid. The agent is restricted to perform a single action at each time step. Additionally, each environment may have a different grid dimensions, leading to variable observation space sizes. These game environments can be represented by a fully observable markov decision process with states s as the pixels of the environment, actions a of the agent and the rewards r given by the game score.

In section 5.2 We propose the Neural Game Engine, based on a modified *Neural GPU*[158] [93].

5.1 Background

In this section, we cover the literature that leads up to the present day's efforts at creating computational models of the state transition function $\mathcal{T}$ of the Markov-decision-process (MDP) first discussed in section 2. We first cover the use of deep neural networks to model various sub-components that are required to model environment dynamics. For example, compressing the state of the environment into a latent representation. These methods are then expanded upon with models that can predict subsequent states conditioned on an action. We then discuss a particularly popular method "state space models", which are a promising method of state representation, in which environment models can be compressed and interacted with, without having to recreate the original state until it is required. Finally, we show how these methods are being used in state-of-the-art reinforcement learning algorithms.

5.1.1 Deep Neural Network Modelling

Access to a simulated environment model for an agent to explore is, in many circumstances, impossible. In fact, in most real-world scenarios, it is simply too complex to model the environment at all. The world is filled with unknowable, unpredictable, and chaotic factors which can only be estimated to a shallow degree. Approaches such as reinforcement learning aim to create a way of estimating the value of particular actions at a point in time given the immediate observable state and, in some cases, information from the past.

These methods do not require a model of the environment in order to predict future states and they have been the subject of a large amount of recent research with many notable successes, such as beating professional players in games such as Starcraft II [293], and Dota [213], see Figure 5.1. These model-free reinforcement learning techniques are discussed in detail in chapter 2.

Agents having the ability to approximately model their environment and predict the outcomes of their actions is arguably an important aspect of general intelligence [247, 246]. Environment models have proven to be useful tools in many artificial intelligent processes, such as learning to play games [293, 259, 65, 155] and control of physical systems such as robotics [118, 120, 121, 305, 161, 178]. In both cases, the model of the environment is used to make exact predictions or approximations of future states in order to optimize the system to achieve a particular goal.

Autoencoders

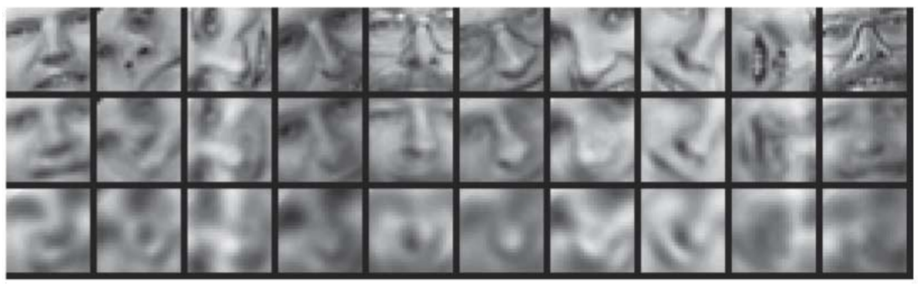


Figure 5.2: Image taken from [130] showing from top to bottom: firstly, a row of random images taken from a test data set. Secondly, a reconstruction using an autoencoder; finally a reconstruction using Principal Component Analysis (PCA)

Autoencoders [130] are arguably one of the most important inventions in deep learning. Autoencoders are typically used to compress high-dimensional and complex data into a much smaller *latent space*. For example in Figure 5.2 one of the earliest version of autoencoders is used to compress simple images into a latent space. These images are then reconstructed from the compressed representation showing a close to identical match with the original images. Since this work, There have been multitudes of improvements to neural networks making this data-compression paradigm much more accurate and usable [13]. Also, there have been many attempts to understand how and why auto-encoder-based representations are so successful from a theoretical perspective [39] [128].

The simplest construction of an autoencoder takes an input x, uses an **Encoder** function to transform this into a latent space s and then uses a **Decoder** to predict an output $\hat{x}$. The output is trained to be as close to the input as possible.

Encoder

$$s = E(x) \tag{5.1}$$

The *Encoder* of an autoencoder model typically consists of several neural network layers which decrease the dimensionality of the input until it meets that of the latent space. These encoding layers learn to compress the salient features of the input data. These neural network layers can be fully connected or convolutional layers depending on the training data. Where the dataset is images, it is most common to see encoders with convolutional layers and pooling layers [59].

Latent Space

$$s \tag{5.2}$$

The latent space refers to a vector or tensor which represents the compressed form of the input data. In simple applications only a 1D vector is used to compress the data. 2D spaces can contain data in a way that maintains spatial information about the compressed data

Decoder

$$\hat{x} = D(s) \tag{5.3}$$

The *Decoder* consists of a number of neural network layers that reconstruct the original input from the latent space. The dimensions of the output of the decoder are the same as the output.

Training The aim of autoencoders is to create a latent vector s which can compress information from a dataset X in a way that the original data can be reconstructed. Training autoencoders is unsupervised as the features used in the latent space to encode the information are unknown. *Reconstruction loss* is used to train the network via stochastic gradient descent (SGD):

$$\mathcal{L}(X) = \sum loss(\hat{x}, x) \tag{5.4}$$

The *loss* function itself in equation 5.4 depends on the format of the dataset X. For example if the data being *autoencoded* are images $x \in \mathbb{R}^{D \times W \times H}$ where

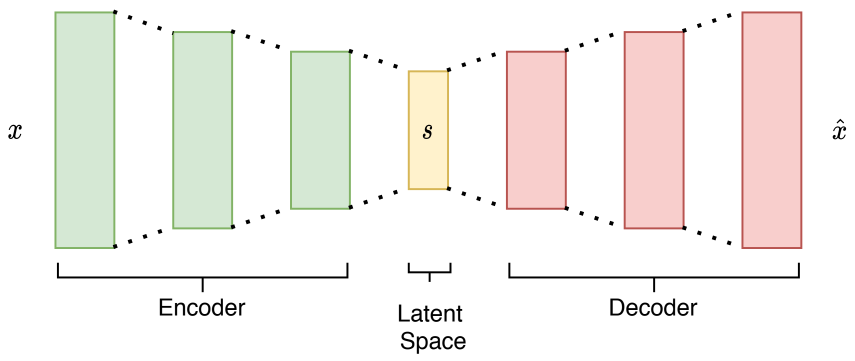


Figure 5.3: Diagram of an autoencoder showing the three main components described in 5.1.1

D is the depth of the colour palette, binary cross-entropy can be used to classify the colour at every pixel:

$$loss(\hat{x}, x) = -\sum_{i=0}^{W-1}\sum_{j=0}^{H-1}\sum_{d}^{D} x_{d,i,j} log(\hat{x}_{d,i,j}) \tag{5.5}$$

This is used mainly in images that have low colour depth D, for example grayscale or images that have low bit-depth such as frames from game emulators [37]. The issue with this method is that the memory required to store the images becomes large if the colour depth and image resolution are high. Additionally, computing the probability of every pixel class and the loss becomes computationally expensive.

Using mean squared error between the predicted $\hat{x}$ and x is a more intuitive and more scalable method of training. In grayscale images $D = 1$ and in full RGB colour images $D = 3$. The mean squared error can then be calculated as:

$$loss(\hat{x}, x) = \frac{1}{3WH}\sum_{i=0}^{W-1}\sum_{j=0}^{H-1}\sum_{d}^{3} (x_{d,i,j} - \hat{x}_{d,i,j})^2 \tag{5.6}$$

Variational Autoencoders

A well known addition to the auto-encoder is the variational autoencoder [171] [232]. Variational autoencoders allow the latent space to represent a distribution rather than a discrete symbol of representation of data. Variational autoencoders, more specifically those representing a Gaussian distribution parameterize the latent space as a tensor of means and associated variances. This latent space can then be sampled in effect to *generate* data that could be contained in the dataset it has been trained with. There are many applications for these kinds of autoencoders, for example generation of faces and denoising of

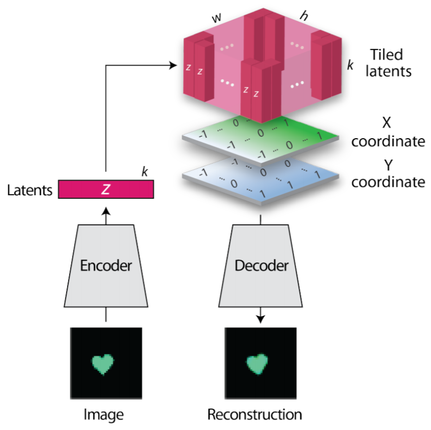


Figure 5.4: Image taken from [296] showing the broadcasting mechanism and concatenated pixel coordinates used to generate the output image

images. Variational autoencoders exist that model distributions other than the Gaussian distribution [203] [151], but they are beyond the scope of this thesis. Any variational auto-encoders described in further sections refer to those based on a Gaussian distribution form unless otherwise specified.

Variational autoencoders generally follow the same architecture as the autoencoders in section 5.1.1 but the latent space represents a distribution $p(z|x)$ where z is a sampled state $z \in \mathbb{R}^k$.

Assuming the true distribution $p(z|x)$ is a Gaussian distribution, it can be parameterized by a set of means $\mu_z \in \mathbb{R}^k$ and standard deviations $\sigma_z \in \mathbb{R}^k$, giving an approximate distribution q(z|x) The variational autoencoder can then be constructed as follows:

Encoder

$$q(z|x) = E_v(x) \tag{5.7}$$

The architecture of the variational autoencoder is similar to that of the discrete autoencoder, except instead of outputting the discrete state s. The outputs are μ_z and σ_z, which parameterize the posterior distribution $p(z|x)$.

Latent Space

$$z \sim q(z|x) \tag{5.8}$$

The latent variable z is sampled from the posterior distribution.

Decoder

$$\hat{x} = D_v(z) = p(x|z) \tag{5.9}$$

The decoder network is the same architecture as that in the autoencoder. The main difference is that the input to the network is a sampled latent variable z rather than a discrete vector. This allows the decoder of the network to be represented by $p(x|z)$.

Training There are two terms that need to be calculated for the loss of a variational autoencoder. The first is the reconstruction loss which is the same as the reconstruction loss in section 5.1.1. The second is a regularizing term to encourage the latent variable z to fit a Gaussian distribution with mean 0 and unit variance $p(z) = \mathcal{N}(0, \mathbf{I})$ [171].

$$\mathcal{L}(X) = loss(\hat{x}, x) - \mathbb{KL}(q(z)||p(z)) \tag{5.10}$$

The Second regularizing term is the Kullback-Leibler divergence which, given the conditions of the multivariate Gaussian given above can be calculated by:

$$\mathbb{KL}(q(z)||p(z)) = -\frac{1}{2}\sum_{j=0}^{L}(1 + log(\sigma_{z,j}^2) - \mu_{z,i}^2 - \sigma_{z,j}^2) \tag{5.11}$$

An important advantage that variational auto-encoders have over standard auto-encoders is that they have an inherent ability to create latent state distributions with disentangled factors [127]. This means that VAEs represent datasets in a way that means that perturbations in each of the units of the state result in perturbations of generative factors such as size, shape, position and color. [127] proposes two constraints that are important to learning disentangled representations. The first is that there should be a large amount of continuously transformed data, for example seeing the same object from many different angles and different colours. This encourages the network to learn a manifold representing similar factors rather than learning single points in the latent space. The second factor is that the network should be encouraged to reduce redundancy and learn statistically independent factors within the data. This is achieved by modifying the loss equation to contain a term β which enforces an information constraint on the KL Divergence, encouraging axis-aligned disentanglement of $\mathbf{z}$. This modification to equation 5.10 can be seen here:

$$\mathcal{L}(X) = loss(\hat{x}, x) - \boldsymbol{\beta}\mathbb{KL}(q(z)||p(z)) \tag{5.12}$$

Disentanglement with β-VAEs is explored further in [49], where additional modifications are suggested to encourage the axis-aligned disentanglement further.

Alternatively [296] Modifies the decoder of a variational autoencoder by broadcasting the latent vector to the channel dimension of a tensor of size

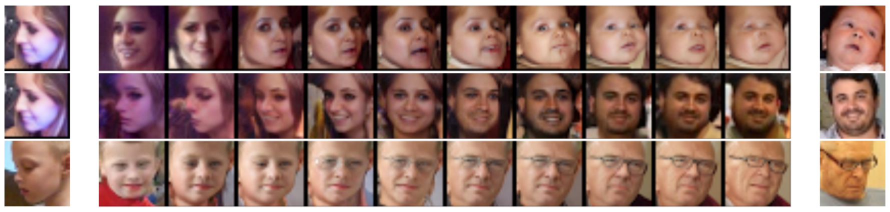


Figure 5.5: Image taken from [211] showing the output of the PixelCNN decoder when interpolating values of the latent conditioning space. The images at the far left and the far right are the start and end points of the interpolation.

$HxWx(S+2)$ where H and W are the height and width respectively, S is the number of elements in the latent vector and the final 2 channels encode the x and y coordinates of the output image. This expanded latent space is then passed through several layers of size-preserving convolutions and finally, the channel dimension is reduced to the RGB components. Spatial broadcast decoders are shown to improve state-of-the-art disentangling techniques such as β-VAE. It is also noted that it is hard for VAEs to encode information from smaller objects, but the spatial broadcast decoder improves this issue drastically.

Variational autoencoders are typically good at learning the distributions of datasets that are continuous, but if the underlying dataset is discrete, this can lead to difficulties in learning these distributions, as traditional VAEs will not have a quantized latent state space. In [212] the Vector-Quantized Variational Auto-Encoder (VQ-VAE) is introduced. Instead of learning a continuous Gaussian latent state space like traditional variational auto-encoders, the VQ-VAE quantizes its latent space. The way this is achieved is by constructing the latent state space as a collection of categorical variables. The sampling of these variables quantizes the latent state using the following formula:

$$q(z=k|x) = \begin{cases} 1 \text{ for } k = \text{argmin}_j ||x - e_j||_2 \\ 0 \text{ otherwise} \end{cases} \tag{5.13}$$

$e \in R^(K \times D)$ represents a set of K embedding vectors where D is the size of each e_i. The quantization effectively sets the sampled latent representations $z_q(x)$ to the nearest embedding vector. VQ-VAEs are particularly useful in creating accurate models of the transition dynamics of MDPs without having to re-create pixel representations between each time step. This is covered in section 5.1.7 where a VQ-VAE is used in model-based reinforcement learning.

5.1.2 Image Generation

In contrast to many image generation algorithms that generate an image in a single step given a latent state. There are several methods that either generate images iteratively over several steps, writing to a *canvas* until the image is generated. In [104], images are generated using an iterative approach that is based on two recurrent neural networks, one which iteratively reads an image and one that iteratively writes the pixels. The locations for reading and writing pixels to and from the source and destination images are determined by the result of a transformation of the latent state itself. It is argued that this attention mechanism is more biologically plausible than methods that generate the image as a whole, as the network can shift its reading and writing focus similar to the process of foveation in nature.

Alternatively, [210] generates images by writing iteratively to a canvas, but instead of using an attention mechanism inspired by biology, it uses a rasterization-based method where pixels are written row by row. This technique also uses a recurrent LSTM to store information from previous pixels in order to generate images that have spatial coherence. In a follow-up to this work, [211] introduces a concept of images based on contextual latent variables, which is related to the tags and labels in the training dataset. Figure 5.5 shows how the images are generated by pixelCNN as the conditioning variable is iteratively interpolated between two images. Further modifications to pixelCNN are made in [239] which optimize the memory usage of the method and improve training loss values.

Another notable method of generating images is known as *Neural Style Transfer*. In general Neural Style Transfer, approaches attempt to apply the *style* of one image to the *content* of another. [97] Introduces a method of extracting features in the top-level layers in the VGG convolutional neural network [263] that represents content and style separately. To generate the images with transferred style, a joint minimization of a content loss from one image and a style loss from another image is performed on a random white noise image. An example of an image generated by this method can be seen in Figure 5.6. Neural Style Transfer has also been applied to video [235], [136] and audio data [195] [291]. Other investigations into manipulating the value of vectors in large neural networks have resulted in several research directions, for example, DeepDream [3] [277] and adversarial images [278].

The final theme of image generation that will be mentioned here is that of the *Generative Adversarial Network* (GAN) [102]. GANs consist of two neural networks, a generator network $G(z)$, where z is sampled from a random distribution $p(z)$ and a discriminator network $D(x)$. The role of the generator network

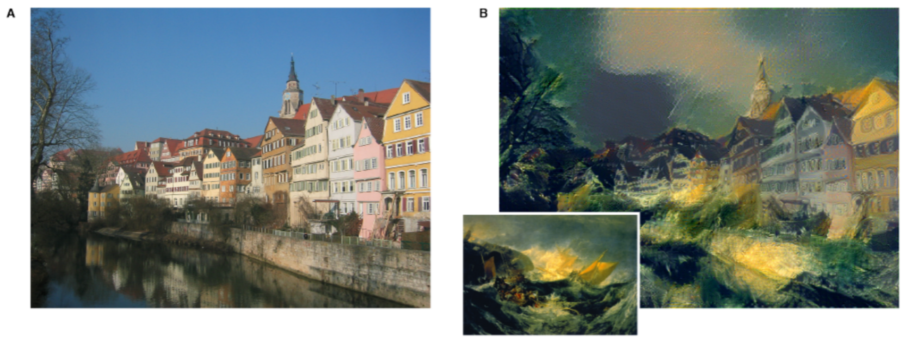


Figure 5.6: Image taken from [97] showing an image being created (right image) from the content (left image) with an applied style derived from a source (thumbnail image)

is to generate images from input noise (z) that fall within the distribution of the training dataset and the role of the discriminator network is to classify whether the generated image is part of the training set or a generated image. The end goal of the training of the network is to maximize the probability that the discriminator classifies generated images as part of the training dataset. In simple terms as the generator network gets better at creating images that are indistinguishable from those in the training data, the discriminator will no longer be able to classify whether the image is generated or not. GANs can produce sharp images that are difficult for even humans to detect they are generated [42] [223] [12]. GANs can also be conditioned in the same way as other image generation methods [211] [239] so images with particular properties can be generated.

5.1.3 Video Prediction

In the previous sections encoding images in a compressed vectorized format and then either reconstructing those images or generating similar images was described. These techniques can be combined with recurrent neural networks in order to generate a stream of images which change depending on the previous images seen. These methods can be used to predict the subsequent frames in videos [270] [228]. The format of these networks follows a similar theme to the autoencoders and variational autoencoders in the previous sections, but the latent space is also calculated based on previous latent spaces.

Recurrent Neural Networks

The most common way to encode information from previous latent spaces is to use a recurrent neural network with a memory state which evolves over time. Equation 5.14 shows how a recurrent latent vector s_t^{rnn} may be predicted in a deterministic way from the previous recurrent state s_{t-1}^{rnn} and an embedded input, which in this case is given by the encoding function $x^{\text{rnn}} = s_t = E(x_t)$ from equation 5.1.

$$s_t^{\text{rnn}} = RNN(s_{t-1}^{\text{rnn}}, E(x_t)) \tag{5.14}$$

The most frequently used form of RNN that provides prediction that takes into consideration previously seen data is the Long Short-Term Memory LSTM [132] [100].

$$\begin{aligned}
f_t &= \sigma(W_{fx}x_t^{rnn} + W_{fh}h_{t-1} + b_f) \\
i_t &= \sigma(W_{ix}x_t^{rnn} + W_{ih}h_{t-1} + b_i) \\
o_t &= \sigma(W_{ox}x_t^{rnn} + W_{oh}h_{t-1} + b_o) \\
\hat{c_t} &= \tanh(W_{gx}x_t^{rnn} + W_{gh}h_{t-1} + b_g) \\
c &= f_t \odot c_{t-1} + i_t \odot \hat{c}_t \\
h_t &= \tanh c_t \odot o_t
\end{aligned} \tag{5.15}$$

The LSTM takes an input x_t at each time step and produces several intermediate states, *forget*, *input*, *cell*, *hidden* and *output*. The *cell* state is passed between layers of LSTMs through time. The *forget* f_t and *input* state i_t are used to determine how much information is removed and added respectively in the cell state at each time step. New information to be added to the state is calculated in $\hat{c}$ and then multiplied with i_t which effectively calculates the new values of c. Conversely, the current information in the cell c_t is multiplied with f_t which effectively removes values from the cell state. The *hidden* state, like the cell state, is passed through the LSTM layers in time. It is used in combination with the current input to calculate the information to be updated based on previous information. Finally the *output* state o_t is used to decide which parts of the cell will be used in the hidden state h_t that will be passed to the next LSTM module. The hidden state h_t can then be used to predict the output.

In order to align the LSTM in equation 5.15 with the RNN in equation 5.14, s^{rnn} refers to a concatenation of both the cell c_t and hidden h_t states. Additionally the prediction of the next output $\hat{x}_t$ (image or otherwise) can be recovered from the hidden state h_t This is shown in equation 5.16.

$$\begin{aligned}
x_t^{rnn} &= E(x_t) \\
c_t, h_t &= RNN(c_{t-1}, h_{t-1}, x_t^{rnn}) \\
\hat{x}_{t+1} &= D(h_t)
\end{aligned} \tag{5.16}$$

Another commonly used recurrent neural network that uses previous states to calculate future predictions is the Gated Recurrent Unit (GRU) [56]. The

GRU is a simplified version of an LSTM that combines the *forget* and *input* gating mechanisms into a single *update* mechanism. The hidden state h_t and cell c_t state are also combined. The equivalent update functions for the GRU RNN are shown in equation 5.17. The performance of GRU units in comparison to LSTMs in many tasks have been shown to be similar [58], [110].

$$\begin{aligned}
u_t &= \sigma(W_{ux}x_t^{rnn} + W_{uh}h_{t-1} + b_u) \\
r_t &= \sigma(W_{rx}x_t^{rnn} + W_{rh}h_{t-1} + b_r) \\
\hat{h}_t &= \tanh(W_{\hat{h}x}x_t^{rnn} + W_{\hat{h}h}(r_t \odot h_{t-1}) + b_{\hat{h}}) \\
h_t &= (1-u) \odot h_{t-1} + u \odot \hat{h}_t
\end{aligned} \tag{5.17}$$

Action Conditioning

In environments such as games, or video sequences where the future frames can be affected by inputs generated by the agent, the prediction needs to take into account the actions that the agent may take. Adding the actions to the predicted model is similar to conditioning during image generation in techniques like pixelCNN [210] [239].

Action conditioning is typically implemented by replacing the input to the recurrent neural network x_t^{rnn} with a function that takes both the action at time t, a_t and the observation embedding from equation 5.1.

$$\begin{aligned}
x_t^{rnn} &= C(E(x_t), a_t) \\
s_t^{rnn} &= RNN(s_{t-1}^{rnn}, x_t^{rnn})
\end{aligned} \tag{5.18}$$

[206] shows that using action conditioning in video prediction in Atari games with a Recurrent LSTM latent space can correctly predict future frames dependent on the actions being presented.

Similarly [91] attempts to predict the movement of objects in videos of robots grasping objects. The architecture uses a recurrent convolutional LSTM (ConvLSTM) (shown in equation 5.19), which replaces the linear dot product between weights and inputs with a convolutional operation. the $*$ operator denotes a convolution operation with respect to the kernel weights W. The use of recurrent convolutional neural networks in this case is to preserve local information over time. The recurrent ConvLSTM is used to predict the distribution over the locations where a pixel will be displaced.

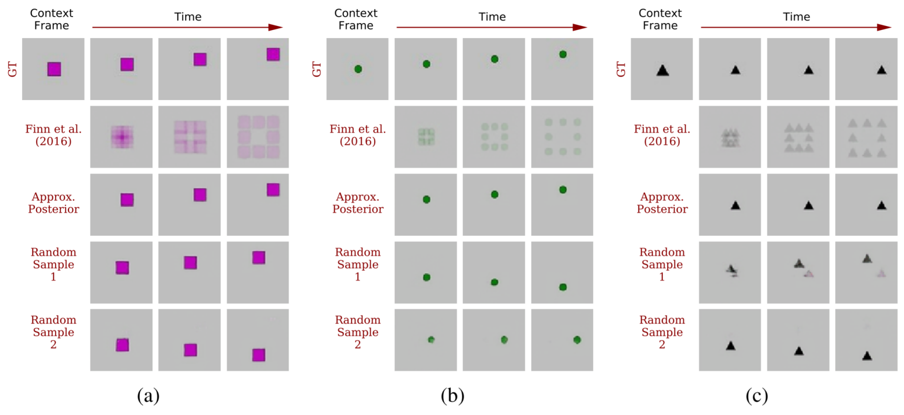


Figure 5.7: Image taken from [18] showing the issue with predicting subsequent frames using deterministic models when the movement of objects is stochastic. In each sub-image, a random shape moves in a random direction. With the deterministic model [91], all possible states are predicted and superimposed onto each other, whereas in the stochastic methods, a plausible output is generated by sampling and the underlying distribution of possible states which is also conditioned on many previous frames.

$$\begin{aligned}
f_t &= \sigma(W_{fx} * x_t^{rnn} + W_{fh} * h_{t-1} + b_f) \\
i_t &= \sigma(W_{ix} * x_t^{rnn} + W_{ih} * h_{t-1} + b_i) \\
o_t &= \sigma(W_{ox} * x_t^{rnn} + W_{oh} * h_{t-1} + b_o) \\
\hat{c}_t &= \tanh(W_{gx} * x_t^{rnn} + W_{gh} * h_{t-1} + b_g) \\
c &= f_t \odot c_{t-1} + i_t \odot \hat{c}_t \\
h_t &= \tanh c_t \odot o_t
\end{aligned} \tag{5.19}$$

Using convolutional neural networks in this fashion however, does not take into account movement stochasticity. In figure 5.7 objects in the video which move under stochasticity tend to blur over time, because the error function is the mean squared error over all examples. This in effect also causes the predictions to average out over time, which when predicting the next frames, produces blurred images. [18] solves this issue by using a variational latent space and uses sampling to predict the next image frames. Image 5.7 shows that this method produces realistic predictions in stochastic environments.

In a traditional auto-encoder model, the latent space is typically a vector of continuous values. If the auto-encoder is intended to encode a finite number of symbols in latent space, it follows that the latent space should also have a finite number of features. Discrete auto-encoders can be used to fit this purpose [160]. The discrete auto-encoder model forces the latent space to compress the sequence of symbols into a discrete representation. It does this using a semantic hashing technique first introduced in [238]. For environments that can be largely represented by discrete latent features, this technique can drastically improve prediction accuracy [159].

Other methods of predicting future states assume that the difference between consecutive states is a small constant and add constraints to the loss functions to encourage this [304], [173]. [11] builds upon this by generalizing that there may be many different constraint functions that help to shape the representation. Instead of fixed relations such as the difference between consecutive states, the introduced algorithm attempts to learn these relations.

5.1.4 State Space Models

Previous work has shown that it is possible to use neural networks to encode information into a compressed latent space in a way that the original information can be accurately retrieved. As an addition to this, those latent spaces can be conditioned either to generate images similar to those in the distribution or conditioned to steer the progression of frames in videos. This conditioning can be replaced by an action variable to predict the next state of an MDP given that action.

The literature in previous sections has included several examples of action-conditioned video prediction for games and physical systems, but these will be covered in more depth in this section. Also, it should be noted that this section concentrates on the use of state space models and their applications mainly to producing models of environments. More details of how these models interact with reinforcement learning methods are given in section 5.1.7.

The traditional way to model physical systems stems from control theory such as Kalman filters [161] and model predictive control (MPC) [178]. These methods generally require that the states of the system can be measured and only the parameters need to be learned.

Many systems can be represented as a *state space model*, which in its simplest form can be represented by the following equations:

The state transition function defines how a set of state variables x progress over time given some input u_t, the functions A and B in many cases are linear matrix multiplications.

$$x_{t+1} = A(x_t) + B(u_t) \tag{5.20}$$

The output equation defines the measurable quantities of the system such as sensor readings. Like A and B, the C and D functions are commonly represented by matrix operations.

$$y_t = C(x_t) + D(u_t) \tag{5.21}$$

To give some more context, the state variables x usually represent the dynamic characteristics of the system such as position and speed. The state space equations can therefore be used to derive the equations of motion of the system. The matrix operations defined by A,B,C, and D can in many cases, be estimated by Kalman filters [161], but this requires the states of the system to be known already.

Recent work has shown that these state space representations including the configuration of the states themselves can in fact be learned using deep neural networks. For example, [166] shows that instead of using state space models in the format of equations 5.20 and 5.21, they can be replaced by a *latent state space* model where the transitions and state variables can be learned. This latent state space model takes the form of an action-conditioned variational autoencoder. [166] shows that their method *Deep Variational Bayes Filters* DBVF can learn to infer the parameters and the states of the state space model even if the models are non-Markovian. They also show that this method is able to accurately predict future states beyond those used in the training data.

Latent state space models are the primary way in which future states are predicted in video games, and many methods have been invented to optimize certain domain-specific characteristics or impose priors that can help future prediction within certain types of environments.

In [55] *Recurrent Environment Simulators* RES take the ideas from [206] and apply it to Atari games. Instead of using a variational autoencoder, RES uses a standard deterministic autoencoder model, as the games it is predicting the future states of are also deterministic. [55] also introduces a useful concept in learning state-space models, which helps training and has inspired many future optimizations. This concept is known as *Prediction Dependent Training* PDT. PDT defines a hyperparameter T, which is the number of state predictions that are performed during training. When $T = 1$, the model is trained only to predict the next step from a previous observation; however, if T is larger, the model is trained over multiple steps, where it predicts multiple states in sequence. [55] showed that this method of training is important in predicting long-term accuracy but at the loss of short-term accuracy.

Similarly to Recurrent Environment Simulators, World Models [115] predicts the future states of several games, but instead of learning a deterministic model, the authors use a variational auto-encoder. World Models is trained on games that do not have deterministic states, for example, the VizDoom [308] and CarRacing [43] environment. A reproduction of the pixels of the VizDoom environment is shown in figure 5.8. [115] also shows that it is possible to train a reinforcement learning agent using just learned internal state representation z as the observations of the reinforcement learning agent rather than using the

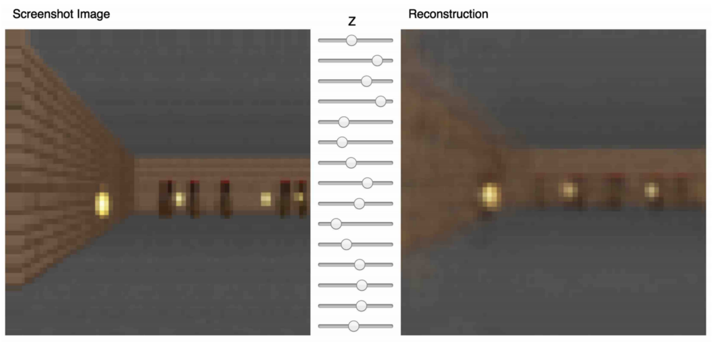


Figure 5.8: Image taken from [115] which shows the reconstruction (right) of a game image (left). The sliders in the center represent the encoding of latent state variables in z which generate the image.

pixels. The authors also show it is possible for the reinforcement learning agent to learn policies within the generated states without seeing any external data. The idea of learning reinforcement learning or MPC policies are expanded upon in [119] and [118].

Latent state spaces for both [115] and [55] use a vector latent variable and learn the compressed version of the states, however in image and video prediction experiments, it has been shown that learning a 2D latent state space representation can produce much more accurate results [91] [18]. The advantage that the 2D state representation has over the single dimension is that locality of information is preserved, as well as just having more variables to represent the state. Effectively the image is segmented and encoded into many latent state vectors that represent the information at a more local level. Convolutional neural network architectures are particularly powerful in these models as they learn complex, multi-dimensional cellular automata rules applied across the state space. Cellular automata rules such as the game of life can be effectively learned using convolutional neural networks [185], [101].

These convolutional operators are also effective when the game state and actions are discrete. For example, in [297], a "basic block" architecture encodes each grid world game tile into its own vector. Given a grid world game with grid shape $G \in \mathcal{R}^{HxW}$ where each tile in the grid is a TxT image, the observation of the game will be $O \in \mathcal{R}^{HTxHWx3}$ representing an RGB image. A convolutional neural network with kernel size T, stride T, and C output channels can be used to reduce the observation space to a *tile embedding space* that has dimensions $HxWxC$. [297] uses this embedding space as the latent state space of a model that is used to predict future states. This 2D state space using embedded tiles is used in many grid-world environment models, including those in the Neural Game Engine [26], which is described in detail in section 5. [297] is similar to [55] in that the environment model predicts the next image given an action combined with the latent state space. Details of how this is used as part of the policy in reinforcement learning are given in section 5.1.7

Instead of using blocks of convolutional layers, an improvement is to use convolutional recurrent neural networks to remember information from previous states. Two common variants are the Convolutional LSTM which is shown in equation 5.19 and the Convolutional GRU, which is shown in equation 5.22:

$$\begin{aligned} u_i &= \hat{\sigma}(U_{us} * s^{rnn}_{i-1} + b_{us}) \\ r_i &= \hat{\sigma}(U_{rs} * s^{rnn}_{i-1} + b_{rs}) \\ c_i &= \hat{tanh}(U_{cr} * r_i \odot s^{rnn}_{i-1} + b_{cr}) \\ s_i &= u_i \odot s^{rnn}_{i-1} + (1 - u_i) \odot c_i \end{aligned} \tag{5.22}$$

Two-dimensional latent state models are improved in [47], where, instead of predicting the output of the image at every step and chaining these predictions together to predict future observations, the latent state space itself is iterated without decoding to an image and re-encoding. Due to this, the iteration of states forward in time is significantly faster and takes up less memory (as no convolutions to decode and re-encode images are necessary). [47] applies this to grid-world game models and also simple systems with physics, such as a bouncing ball. An additional difference between [297] and [47] is that [47] makes use of variational inference as well as a deterministic model to create a stochastic latent state space.

The stochastic latent state space model can be modeled in the following way:

Starting from equation 5.18, we want to be able to sample the state from an underlying distribution which depends on the previous deterministic state s^{rnn}_t the action a_t and a stochastic state z^{rnn}_t. We can create the stochastic state z_t by sampling from a distribution:

$$z^{rnn}_t \; P(z^{rnn}_t | s^{rnn}_{t-1}, a_{t-1}) \tag{5.23}$$

And then the state transition function can deterministically calculate the next state s^{rnn}_t given the sampled state z^{rnn}_t, the action $a_{t-1}t$ and the previous deterministic state s^{rnn}_t:

$$\begin{aligned} s^{rnn}_0 &= C(E(x_t), a_t) \\ s^{rnn}_{t-1} &= z^{rnn}_t \\ s^{rnn}_t &= RNN(, z^{rnn}_t, a_t) \end{aligned} \tag{5.24}$$

It is important to note here that the state transition does not depend on previous observations, but the initial state s^{rnn}_0 is obtained by the action conditioning function C described by equation 5.18.

Two-dimensional latent state spaces with stochastic inference help to solve two problems with predicting the future states of game states. Firstly the issue of local interactions, which the two-dimensional state space aims to solve, and secondly stochastic elements of games such as random events can be modeled by sampling the stochastic latent state space. With these models, however, they can only predict a single time-step into the future and have calculated on a time-step-by-time-step basis what might happen in the future. It is theorized that artificially intelligent agents should be able to predict what will happen several steps in the future without having to go through every time step and calculate everything individually. [105] introduced a modified auto-encoder the "temporal-difference autoencoder", that also encodes a time variable into the latent state space. This allows the model to be arbitrarily advanced in time without progressing through many computational steps.

Predictive forward models have been applied to several tasks where parts of the environment are not directly observable, but can only be inferred by partial signals. Stochastic latent state spaces have been successful where an inherent *belief state* of the environment needs to be maintained. [6] shows that a model of a robot hand trained with a stochastic latent state space model can predict the outputs of over 100 sensors accurately with many time steps into the future and that the learned latent state space can also be used to predict the shape of objects, even if the hand has lost contact with the object after initially touching it. This shows that the latent state space memorizes past information. [6] also introduces the concept of observational overshooting during training which is used to create much more accurate belief state representations.

[106] also attempts to build a belief state in a partially observable environment. The environments that are used in this work are 3D worlds that can be modified by the agent for example by placing blocks to reach rewards. In order to improve the representation of the belief state, the variational autoencoder is paired with a Kanerva machine [306] [162], which acts as memory for the agent. In order to measure the accuracy of the learned belief state, a separate model is trained which takes the input of the belief state and predicts the map of the environment from a top-level view. As the agent explores the environment from its own egocentric view, the predictions of the environment become more accurate, showing that the belief state is understanding the structure of the environment as it explores. The agent is compared against two other reinforcement learning agents, one that is a plain latent state space model with an LSTM used to predict future states, and one that includes a slot-based memory. The agent with the Kanerva machine-based memory outperformed the two other methods.

Similarly to [6], [118] introduces PlaNet which uses a stochastic latent state space model which is used to learn accurate models of several Deepmind Lab [36] environments. Like the robot arm environment in [6], PlaNet learns the latent state space model of the underlying sensors, which in the case of Deepmind Lab are angles and positions of legs of various different creatures. PlaNet uses MPC to control the actuators with each environment but using the accurate learned model. PlaNet also introduces another important training mechanism *latent overshooting* which can be used alongside observational overshooting. *Dreamer* [119] uses a similar latent state space model to PlaNet, but instead of using the model in predictive control, it is used as an environment in which to train a deep reinforcement learning agent. An accurate latent state space model is learned which can accurately roll out many steps into the future using just the learned latent dynamics. These learned latent states are then used as the observations of a deep reinforcement learning agent. This allows the reinforcement learning algorithm to be separated entirely from the process of learning from pixels. This method also allows much more data to be generated as it can be sampled from the underlying model in parallel on GPU making the process of learning significantly more efficient in terms of time and memory.

An accurate model of the environment is also used to play atari games in [159]. The Simulated Policy Learning algorithm *SimPLe* iterates between gathering data from the environment using a reinforcement learning policy, using the gathered information to learn a world model, and then updating the policy by using the learned world model. As the model of the work improves, the policy also improves. As atari games are mainly deterministic, [159] uses a state space model that contains a discretized auto-encoder [160] described in section 5.1.1 as well as a stochastic state space. This combination of stochastic and deterministic parts is used in many other state space models [47], [118], [119].

[98] also showed that learning a latent state space representation as an auxiliary task leads to large improvements over model-free RL. This paper also provides several formal guarantees that the latent state space representation is good for accurately predicting future states and rewards.

Other methods of training belief state models have also been explored, for example, [112] uses contrastive predictive coding to learn a belief of the position and orientation of an agent based only on the agent's partially observable state. The belief state contains an encoding of the agent's uncertainty about the environment. This uncertainty can lead to more effective exploration where the agent is acting to reduce its uncertainty about the environment.

Learning a latent state space representation that can reproduce the entire observation is not necessarily required to obtain high scores in several tasks. [249] uses a recurrent model that predicts the reward, the value, and a policy at every time step. It's argued that the state representation that is learned does not include the parts of the environment which are static or are not important in predicting the value function or rewards. The model is trained in a similar way to AlphaZero [260], which uses MCTS to search through the state space given the initial observations and list of actions. During the MCTS rollouts, the reward and value functions are learned. MCTS can then be used to search the learned model itself to find accurate rewards and value functions when playing games.

[254] uses an ensemble of latent state spaces in order to measure uncertainty in future predicted states. Uncertainty is measured by taking the difference between the latent states of the ensemble. This is referred to as *latent disagreement*, which is similar to *model disagreement* proposed in [218], but the "disagreement" is calculated by the difference in latent states rather than the difference in observations of an ensemble. The uncertainty is then used as an intrinsic motivation reward in a reinforcement learning agent.

One of the main contributions to this thesis [26] builds a deterministic latent state space that is specifically aimed towards accurately reproducing the rewards and observations of 2D grid-world games.

Building accurate models of environments that are not specifically tied to policy prediction, and can encode latent spaces that are invariant to small changes in the observational state is an important challenge in reinforcement learning as small perturbations in observation states can cause reinforcement learning policies to break down [129], [120]. Learning invariant latent state space models that filter out this noise can produce more robust policies.

5.1.5 Object-Centric Models

Object-centric models follow the premise of trying to individually segment and then model each segmented object individually. Image segmentation is another field in deep learning that has been studied to a large extent [193]. Many state-of-the-art image segmentation algorithms combine attention mechanisms with locality processing [294].

Attend Infer Repeat (AIR) is introduced in [84] which uses an attention model to *attend* to specific parts of images, infer their latent representation and then continue on to the next object. Each object is associated with a specific learned latent model. Decoding networks that perform affine transformations of the learned latent models are then used to reproduce the original images. [66] improves upon (AIR) by introducing convolutional layers to the internal recurrent attention network. The introduction of convolutions allows spatial invariance and helps the algorithm to scale to much larger scenes with more objects.

Other models such as [134], [274] and [32] model the physical interactions between objects with explicit graph-like architectures.

[50] Introduces the Multi-Object Network (MONet) which attempts to decompose images into individual objects that make up the scene. MONet consists of two neural networks, a recurrent attention network and a VAE. The recurrent network is trained to produce segmentation masks of individual objects sequentially, conditioned on previous masks, and the VAE is used to store the individually extracted objects. The number of extraction steps K is set as a hyperparameter. The image is then re-composed from the learned components of the image. The thesis argues that the loss functions for masking and reproducing the image lead to masks that segment individual objects in the original image. The reasoning for this is that decomposing an image into constituent parts is the most efficient way to store information and minimizing the loss will naturally result in the image being decomposed in this way. Figure 5.9 shows the learned object decomposition and reconstructions at various points within the reconstruction process.

Similarly to [50], [103] takes on the problem of multi-object decomposition and representation. [103] introduces the Iterative Object Decomposition Inference NEtwork (IODINE) which takes a similar architecture to MONet in that it iteratively decomposes objects using a recurrent network. Instead of using an attention model however, IODINE uses an iterative amortized inference [190] to refine the segmentation of objects, and instead of a standard VAE for reconstruction, uses a spatial broadcast decoder [296].

Generative Query Networks (GQNs) [85] learn a scene representation given a query viewpoint V_q which represents a point in 3D space and a direction that generates an image of the scene given that particular location and viewing direction.

Given that the network contains no implicit rendering pipeline, for example, one that would use model-view-projection matrices in combination with fragment and vertex shaders, the thesis argues that the network must be learning a representation of the environment that contains information about the individual size, color, and positioning of the objects that can then be rendered from any new location given the query viewpoint. This hypothesis is tested in several ways for example the GQN can accurately reproduce a top-down representation of the map of a 3D environment when it is only trained using 3D images produced by a camera moving within the environment.

[128] provides a theoretical foundation for the disentanglement of information within latent states using group theory and symmetry. There are many symmetrical factors in data such as geometric symmetries in images and symmetries with physical systems such as forces pushing objects. These symmetries

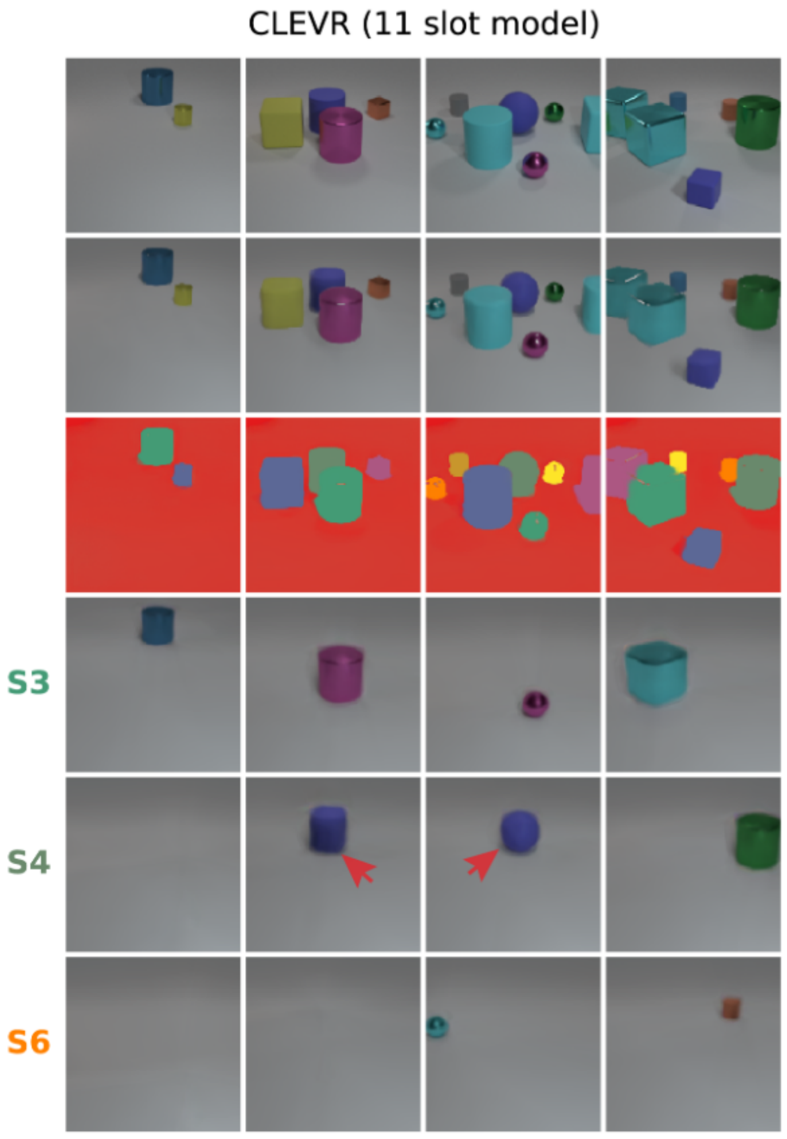


Figure 5.9: Image taken from [50] showing object decomposition and reconstruction on the CLEVR dataset, including individual reconstructions of occluded object parts (s4 red arrows). The number of steps K to segment the scene was set to 11.

in systems form groups of *disentangled representations* if there exist transformations that can map between these symmetrical states without affecting other groups of symmetries. It is also theorized that there are transformations that exist that can encompass color and lighting, meaning that neural networks can learn representations that can generalize across color and lighting, leading to stronger generalization in many areas.

[8] Introduces a method of learning a method of representing the Atari environment based on DeepInfomax [131] which aims to be able to encode features of the environment that are also reflected in the internal state of the Atari environment's Read-Only Memory. Many features of games that are important for learning good policies are not available directly by pixels at every frame but have to be inferred over several steps or remembered from previous frames. Spatio-Temporal DeepInfomax (ST-DIM) is introduced to be able to learn these features in a way that can be extracted using linear classifiers, referred to as sensors. Several features are hand-picked from various games to test how accurately the ST-DIM method can be used to predict internal states. ST-DIM is compared against several other methods such as VAEs video prediction methods [206] and outperforms them in predicting the internal features of several Atari games.

[176] uses the unsupervised object landmark discovery algorithm from [318] to identify parts of objects in videos of games (key points). These key points can be used to generate latent models of the environment that represent objects, relations, and geometries rather than learning a latent model from pixels, for example with an autoencoder. The network is trained to predict key points more accurately by learning a method of mapping the key points between two images where the key points are spatially translated. This "key point bottleneck" enforces a more accurate key point representation to be learned and encourages key points to be detected on image locations that are not static. As the key point representation learned by the *Transporter* model contains mostly non-static and controllable parts of the environment, control policies in Reinforcement Learning have a simplified action space which leads to faster learning.

Additionally, the key point representation can be used to define intrinsic rewards to aid exploration. The intrinsic rewards are designed to be proportional to the distance that the key points move, meaning that actions that affect the key points the most are more favored. In this way, the key point method improves the empowerment of the agent, as it is less distracted by parts of the environment that are inconsequential [109].

An alternative method of keypoint detection is introduced in [144] where key points are used as the basis of minimizing the information between different images of human poses. This information bottleneck technique is used to detect

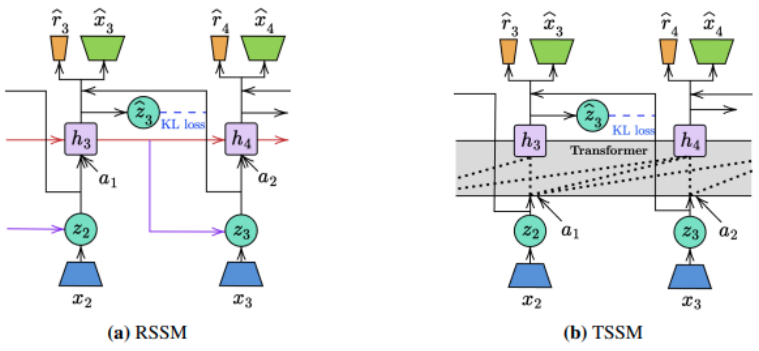


Figure 5.10: Image taken from [51] showing the architecture differences between the Recurrent state-space model and the Transformer state-space model.

particular features in the images without any manual supervision or labeling data.

[50] [85] [176] require prior knowledge of the environments they are tested in such as the number of objects, key points or changing factors, which in unseen data or new environments is unknown, this leads of difficulty in generalization. [172] however proposes SlowVAE which attempts to find disentangled representations with no prior knowledge of the components of video data. SlowVAE assumes *temporal sparsity* which assumes that latent representations between time steps of video frames tend to change slowly but have sudden occasional jumps. This is modeled as a Gaussian prior over state variables in subsequent time steps. It is noted in the thesis, however, that this theoretical assumption does not necessarily fit the datasets it is trained on, even though they produce state-of-the-art results.

The methods described previously in this section concentrate mainly on decomposing static scenes into disentangled object representations, but they do not describe how these object-oriented systems can be used in model-based reinforcement learning. [320] introduces a method that combines the decomposition of objects in scenes and action-conditioning using a spatial transformer network [142] to translate the decomposed images of objects for the prediction of the next observation.

5.1.6 Transformer Models

With the recent successes of transformers [290] in many areas of deep learning, several attempts have been made to model the latent state space of environment models using them. Transformer architectures are a common replacement for sequential models that allows information to be shared and processed across a sequence. At the heart of the transformer architecture is the self-attention mechanism shown below:

$$\text{attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{5.25}$$

The inputs of the attention mechanism are the **keys**, **queries**, and **values** K, Q, and V respectively. The values of K, Q, and V are encoded vectors given by linear layers that act upon embedded text tokens. The attention vector in equation 5.25 is a weighted sum of the embedded inputs, where higher weighting is given to more important text tokens. The attention mechanism effectively allows a single vector to be produced by combining and comparing a set of inputs. Inputs that are more important give higher weight to the subsequent values.

Transformers make use of multi-headed attention layers, which contain several parallel self-attention mechanisms. This allows different self-attention mechanisms to encode multiple different relationships, the generated values for each self-attention model are then concatenated and reduced to a single output.

Architectures that use transformers commonly beat the state-of-the-art title in NLP and image processing benchmarks. For example, the transformer architecture in ANNA [154] has state-of-the-art performance in the Stanford Question-Answering Dataset (SQuAD) [227, 226]. The Vision Transformer (ViT) architecture [77] uses almost the same architecture as the transformer proposed in [290], but instead of its inputs being text tokens, it uses image patches, with added positional encodings. When trained on large numbers of images, the ViT architecture outperforms many convolutional architectures.

As transformers are very successful at sequence prediction tasks, it's natural to extend their usage to state prediction in model-based reinforcement learning. In *Transdreamer* [51], the recurrent state-space model used in DreamerV2 is replaced with a transformer that uses the hidden state history to predict the next hidden state. This architecture is shown in Figure 5.10.

Similarly, in [192], a transformer architecture named *IRIS* is used to predict subsequent states of a world model, but the states are encoded using a VQ-VAE instead of the categorical latent states used in Transdreamer/DreamerV2. IRIS is trained on the Atari 100k benchmark and performs impressively, beating methods such as MuZero and efficient Zero.

5.1.7 Model-Based Reinforcement Learning

Many of the methods of learning forward models using deep neural networks are covered in section 5.1.4. This section focuses on how these models can be used to generate new results and improvements when used as part of the architecture of agents in deep reinforcement learning models.

The previous section has covered several advances that have been made in the pursuit of model-free reinforcement learning. However contrary to this, some of the most interesting advances, such as [21] and [20] have used inverse-dynamics models of the environment as part of their exploration policy.

The inverse dynamics model [217] used in these papers does not predict the next of the game given an action and previous state but predicts the action a based on two consecutive states s_t and s_{t+1}. The model used to produce this prediction learns an embedding of the state of environment $\phi(s_t)$ by minimizing the loss between the predicted action and the actual action:

$$\hat{a} = g(\phi(s_t), \phi(s_{t+1}), \theta_I) \tag{5.26}$$

The embedding function ϕ learned by this model (although not used in [21] [20]) can then be used as an input to a separate forward model $\hat{\phi}(s_{t+1}) = f(\phi(s_t), a_t, \theta_F)$ and the prediction error between the embedded state and the predicted state $\hat{\phi}(s_{t+1})$ can be used as an intrinsic exploration reward:

$$r_i = ||\hat{\phi}(s_{t+1}), \phi(s_{t+1})||_2^2 \tag{5.27}$$

[218] expands upon [217] by using an ensemble of models which predict the next state and measure the difference between the models to use as an intrinsic reward. This allows the agent to look ahead and plan which action will lead to the next state, which it cannot predict. In the previous work, the exploration reward was measured after the agent had already reached a state, meaning that the agent would be inclined to follow that path if it reached that same state again, however in the ensemble method, the agent can proactively make decisions about which action to take next having seen the predictions of the next states. Figure 5.11 shows an architecture diagram of this method.

Arming reinforcement learning agents with intrinsic rewards based on prediction error encourages agents to take actions to reach states with high prediction errors. As more of these states are encountered, the prediction error reduces for the commonly seen states, which means the previously learned Q-values now have different targets. Careful balancing of the intrinsic rewards and the environment rewards must be undertaken to avoid the models being biased too much by high intrinsic rewards at the start of the training.

Another issue with generating intrinsic curiosity rewards by measuring prediction error is known as the *noisy TV problem.* The noisy TV problem occurs when stochastic parts of the environment have an inherent baseline error as they cannot be predicted. In some cases, this prediction noise can render the

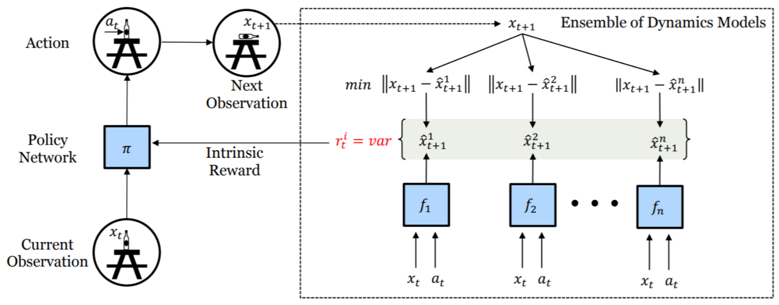


Figure 5.11: Image taken from [218] showing a diagram of the ensemble of models used to measure disagreement as an intrinsic exploration reward.

intrinsic reward useless as the agent will learn to seek out states that have high noise and will never learn the real extrinsic rewards of the environment.

Random Network Distillation (RND) [48] is proposed to attempt to alleviate the issue of noise in the environment causing large intrinsic rewards. Instead of learning to predict the next state from the previous state and action, RND proposed that a network with random weights is used to generate a random state, and then another network is trained to try and predict that same random state. The prediction reward is then given by the measurement of the error of that prediction. Since the prediction is entirely deterministic and there is no next-state prediction, the stochasticity generated by randomness in the environment itself is no longer a problem.

Other methods that avoid the noisy TV problem have also been proposed. For example, [16] improves upon Variational Information Maximization Exploration VIME [135], which derives its exploration bonus by finding actions that maximize the information gain between subsequent states. Information gain refers to quantifying the reduction in entropy between two consecutive states, indicating the amount of new information obtained. [16] introduces Neural Differential Information Gain Optimization (NDIGO), which predicts a baseline of state information and a prediction of the next state from the current observation. This baseline prediction can reduce the noise generated from stochastic environments and therefore avoid the intrinsic rewards created by stochastic noise in observations.

Other measures are also used to generate prediction errors. For example, Reward Impact-Driven Exploration (RIDE) [224] avoids the problem of curiosity rewards reducing over time as the forward model is learned by using an impact measure. The impact I is measured by learning a latent-state space representation in the same way as [217] and then taking the difference between the learned consecutive states:

$$I = ||\phi(s_{t+1}) - \phi(s_t)||_2^2 \tag{5.28}$$

The impact measure is then divided by the square root of the number of times that the state s_{t+1} is measured $N(s_{t+1})$ to stop the agent from learning to change between two different states constantly. This gives the intrinsic reward:

$$r_{RIDE} = \frac{||\phi(s_{t+1}) - \phi(s_t)||_2^2}{\sqrt{N(s_{t+1})}} \tag{5.29}$$

Models of the environment are not just used for calculating intrinsic rewards for curiosity-based agents. Models of the environment can also be used to improve the prediction accuracy of value functions and increase sample efficiency [90] [98]. [90] directly uses a model of the environment to predict the value function at any given state, whereas [98] uses learning of a model as an auxiliary loss. Both techniques typically use model-free algorithms but use models to augment them. [60] and [7] use a similar approach to [90] but learn the model of the environment through backpropagating the value prediction error as well as just the state observation error.

In certain games, for example, Sokoban, the agent can perform actions that put the game into a state in which the agent cannot complete the level. In Sokoban, avoiding these un-recoverable states have to be avoided using a certain amount of planning. Reinforcement learning does not directly handle these circumstances because reaching these un-reversible states does not necessarily result in a negative reward. In this case, the reinforcement learning agent does not learn to avoid these states. Additionally, if the levels are procedurally generated, these un-reversible states are very difficult to detect.

In environments with these irreversible states that require careful planning to solve, models of environments can be searched using algorithms like MCTS to avoid these states.

Imagination Augmented Agents (I2A) [297] uses an environment model to predict both the value function of a network and a policy. At each time step, the environment model is rolled out several steps into the future, and the result of these rollouts is combined with a prediction network for both the value and policy of the agent. [297] argues that these rollouts into the future at each time step allow the agent to view potential future states before they happen and adjust the agent's policy based on these predictions. [47] improves upon the model used by I2A by allowing stochastic future predictions and removing the need to transform to pixel observations between actions. These stochastic state-space models are explained in more detail in section 5.1.4. Using the stochastic state space models improved the maximum score that the agents could obtain with the deterministic I2A models.

Using models for planning has seen several advances in both accuracies of the model itself and how to use the model to enhance the policy. In [150] a similar method to [192] is used to encode the latent state space and predict subsequent states, however instead of using the model to train an RL agent, the model is used to plan trajectories using beam search.

It has also been suggested that planning can be an inherent property of some RL architectures, [110] argues that in some cases, it is possible that a model-free approach to learning in these kinds of difficult environments can lead certain network architectures to inherently plan. [110] uses a stack of ConvLSTM units that share a very similar architecture to the latent state-space models, which predict future game states. However, it uses these to predict the policy and value function for an agent. The Stack of ConvLSTMs is not trained to predict future states of the environment but is trained directly as an actor-critic network using IMPALA [86]. Different rollout and stack sizes are experimented with, and the results show that larger rollouts, which the authors associated with "more planning time," allowed the agent to perform better.

[115] Builds a recurrent stochastic state space model of several 2D and 3D environments such as CarRacing from [44] and Doom [308]. It then uses these learned models to learn policies without interacting with the environment. Learning a dynamics model of the environment or underlying state space is often faster than learning a reinforcement learning agent to solve the environment itself. In section 5.1.4, we mentioned both PlaNet [118] and Dreamer [119], which use stochastic state space models in which control algorithms are applied. As both of these methods learn the environment's dynamics in a supervised manner, the models require fewer samples to train to a reasonable accuracy than to train a reinforcement learning agent. This agent can then be trained in the "imagined" model itself.

The Dreamer model used in [119] is built upon in Dreamer V2 [121]. DreamerV2 modifies the original Dreamer architecture by using a latent space consisting of a set of discrete categorical variables; this effectively quantizes the state space making it less prone to cumulative errors when predicting several states in the future. Dreamer V2 is used to learn models of many of the Atari 2600 games. These models are then fixed and then used exclusively to train an actor-critic algorithm. Using these accurate latent state space models, the actor-critic algorithm is able to reach human-level performance on many of the Atari 2600 games.

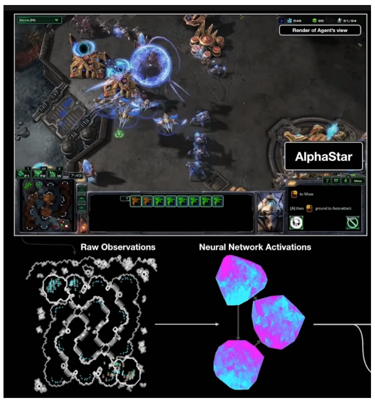
In [117], Dreamer V2 is also proposed to learn the model of a complex grid-world environment, "Crafter," which has dynamics such as an inventory and NPCs that can chase the player. In addition to this, the environment is partially observable, meaning that the underlying model must also generate the world as the player moves around. Despite the complex modeling requirements, the DreamerV2 agent outperforms other simple model-based baselines in the paper that are trained with the same state budget. It is interesting to note, however, that by performing a more-in-depth hyperparameter search, a model-free PPO agent was able to out-perform the DreamerV2 agent in the same number of steps it takes to learn the model itself [271].

In many cases, when learning models for reinforcement learning, emphasis is put on trying to reproduce a state of the environment that can be converted back into an observational state that is as close as possible to the original observation. However, learning the exact representation of the environment is not necessary in many cases, as there are many parts of the observational state that contain very little information about solving the task at hand. [249] introduces μZero, in which modifications to the AlphaGo are made that use a learned environment model. In this case, the environment model is only required to learn when a reward is given by the environment, and learning the exact state transition function is not required. The authors argue that learning the reward in this way without needing to learn the exact state transitions is much more efficient and filters out only the information relevant to solving the tasks. Several improvements have since been made to the μZero algorithm since its publication [248, 312]

In many of the model-based planning algorithms described above, the "planning" algorithms are also used during inference in order to choose the next best action. however [123] shows that policies trained using models for planning rollouts actually do not require planning during inference. Performance in many environments is similar with or without the planning module. This suggests that policies trained in this way will learn to avoid parts of the state space that lead to unrecoverable states. This suggests that the planning algorithm in a model can effectively act as a teacher to provide high-quality training trajectories.

Learning accurate models of environments does not exist without issues. One of the most prevalent issues in learning world models is the fact that the model learning is dependent on the trajectories of the agents collecting the experience data. [231] demonstrates that in certain environments such as those with stochasticity, policies that rely on models to plan do not learn the causality of specific events and therefore make mistakes in predictions that lead to bad performance of models. To fix this, [231] models this issue under the context of causal reasoning and proposes a causal partial model to learn adequately to predict under any arbitrary policy.

[24] introduces *Ready Policy One* RPO, which jointly optimizes policies for both reward and model uncertainty reduction. RPO specifically avoids stochas-



(a) A screenshot taken from the game Starcraft II, where an AI (Alphastar) plays professional player "MaNa"

(b) A screenshot taken from the game Dota 2, where an AI (OpenAI Five) plays against professional team "OG"

tic areas of the state space so the model of the environment is as deterministic as possible. Policies in a world model are directed with the objective of acquiring data that most likely leads to subsequent improvement in the model. This reduces the amount of data required to learn the world model itself and leads to more accurate rollouts.

Another method of avoiding incomplete models of the environment is to make sure all the states are visited an equal amount of times or use a large amount of data and build a model architecture that is designed specifically to model the dynamics of the particular environment. [79] Achieves this by training a model which concentrates on learning the physical dynamics of the environment rather than action-conditioned states. It uses this model as an input to the policy network so as to provide information on the dynamics of the environment without directly predicting the next states given the actions.

In section 5.1.7 we introduced the concept of model-based RL. In model-based RL the agent specifically learns a model of the dynamics of an environment in which it is situated. There are several approaches to utilising the learned model such as planning subsequent actions, using prediction error to provide intrinsic motivation, or using experience in the model instead of the original environment. In the case of planning or learning within the model, having an accurate model of the environment is vital, as the model need to closely cover the distribution of states.

As an example, if a chess playing agent has access to a perfect model of the game of chess, it can test out many moves in advance and then pick out the best move from the ones it has tried. If the model is imperfect, then the planning algorithm will assign incorrect scoring to states and thus hinder the performance.

5.1.8 Neural GPU

The Neural GPU (NGPU) architecture introduced in [158] and improved upon in [93] can learn several unbounded binary operations. For example, multiplication, addition, reversal, and duplication. This is achieved by effectively learning 1D cellular automata rules which are then applied over a number of steps until the result is achieved. The number of steps I is typically proportional to the size of the binary digits being processed. The Neural GPU applies the cellular automata rules to an embedded representation of the binary digits using a convolutional gated recurrent unit (CGRU) with hard non-linearities. The CGRU itself is described by the following set of update rules:

$$\begin{aligned} u_i &= \hat{\sigma}(U' * s_{i-1} + B') \\ r_i &= \hat{\sigma}(U'' * s_{i-1} + B'') \\ c_i &= \hat{tanh}(U * r_i \odot s_{i-1} + B) \\ s_i &= u_i \odot s_{i-1} + (1 - u_i) \odot c_i \end{aligned} \tag{5.30}$$

In the above equations U, U'', U''' are convolutional kernel banks and B, B', B'' are learnable biases. The $*$ operator is used to describe a convolution operation of the left parameter over the right. For example, $U' * s$ denotes the kernels in U' convolved over the values in s. $\hat{tanh}$ and $\hat{\sigma}$ represent the hard non-linearity versions of the *tanh* and *sigmoid* functions respectively and $\odot$ represents the Hadamard (or element-wise) product between two tensors. Details of the hard non-linearities are given in [93]. When dealing with binary operations, the Neural GPU takes an input of arbitrary length, containing the binary encoded digits and the operation to perform. The binary digits and operation symbol are embedded into the initial state s_0. This state is then iterated through the CGRU for n steps and final state s_n is read out using a softmax layer which predicts the binary result.

As the Neural GPU can be seen as a recurrent application of learnable cellular automata rules, this leaves it well suited to being able to learn the local rules of grid-world-based games. This architecture is comparable to other state-space architectures that use size-preserving layers [297], [47] [110], with the exception that parameters are shared between layers, no latent state information is shared between frames and different gating mechanisms are explored.

5.2 Neural Game Engine

The Neural Game Engine is a neural network architecture based on a modified Neural GPU. The main modifications to the Neural GPU are outlined in this

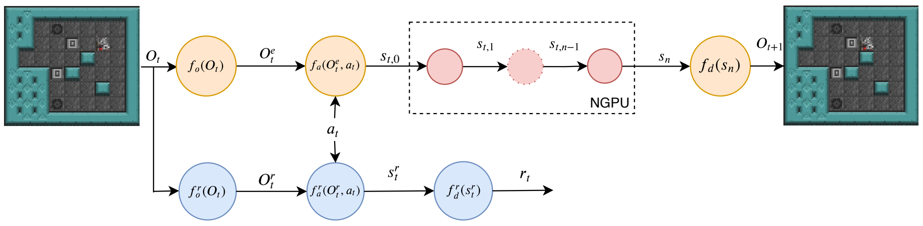


Figure 5.13: Architecture of the Neural Game Engine

section. The Neural Game Engine state differs from the Neural GPU in that the state s is three dimensional (W_s, H_s, C_s) where the width W_s and height H_s reflect the width and height in tiles of the game being trained and C_s is the number of channels. Each vector stored at (w_s, h_s) represents a single tile in the grid environment. The convolutional kernel banks U, U', and U'' are also modified to be two-dimensional with a $(3, 3)$ shape. The stride and zero-padding are the same as the original paper at 1. As no diagonal movements are allowed in any GVGAI environment, the kernels are also masked to ignore the non-adjacent cells. Similarly to the NGPU, an iteration of the CGRU unit with input s_i produces a new state s_{i+1}. The number of iterations of CGRU cell per frame of the game state is tuned as a hyperparameter n.

The width and height of the games in the GVGAI environment can be any positive integer value. Since changing the values of W_s and H_s does not change the number of parameters in the underlying Neural GPU, the Neural Game Engine can generalize to any W_s and H_s. This unbounded computation of game state is discussed further in section 5.4.4.

In many reinforcement learning techniques, the rewards that the game provides to the player are augmented in order to aid exploration, modify the agent's goals, or provide auxiliary losses to reduce training time [256], [10]. In some cases, the original rewards supplied by the environments are modified from their original values with a technique known as reward shaping [108].

Reward prediction in the Neural Game Engine aims to reproduce the original game rewards as accurately as possible but decouples reward prediction learning from learning the game mechanics.

At every time step the Neural GPU is applied to an encoded observation image O_t, iterated n times and then decoded to give the next observation state O_{t+1} and reward r_t. Figure 5.13 shows the architecture for a single time-step calculation.

5.2.1 Observation Encoder - $f_o(O_t)$

In the GVGAI environment, tiles in the trained games are set to have the same width and height dimensions D. This consistency allows the tiles to be embedded into a tensor O_t^e with the same dimensions of the NGPU initial state s_0. This tile embedding is achieved by using a convolutional neural network with kernel width, kernel height and stride set to D, input channels set to 3 to reflect the RGB components of the image and finally output channels set to C_s, the number of channels in the NGPU state.

5.2.2 Observation Decoder - $f_d(s_n)$

To render the game pixels, a mapping from the underlying embedded tile representations to the pixel representations of the tile is learned. This mapping takes the form of a convolutional transpose with kernel size D and stride D. The number of input channels is set to 3 to reproduce the RGB components. This mapping recovers a tensor of shape $(D.W_s, D.H_s, 3)$ which can be rendered.

5.2.3 Action Conditioning - $f_a(O_t^e, a_t)$

As the action needs to be considered as part of the local rule calculations in the NGPU, information about the actions must be available in the s_0 state, along with the observations. To achieve this, the action a_t is one-hot encoded and then embedded with a linear layer of output size C_s. This is then added to each cell of the initial state s_0. In practice this can be achieved by tiling the one-hot representation of the action into a tensor of size (W_s, H_s, A_s), where W_s and H_s are the width and height of the NGPU state, and A_s is the cardinality of the set of actions for the game. This state can then by passed to a 1x1 convolutional neural network with C_s output channels. The resulting tensor can then be added to the O_t^e, which results in the initial s_0 state of the Neural GPU.

5.2.4 Reward Observation Encoder - $f_o^r(O_t)$

The reward observation encoder consists of a tile embedding layer similar to the observation encoder encoding each tile into a vector with C_r channels, giving an embedded observation state O_t^r of size (W_s, H_s, C_r).

5.2.5 Reward Action Conditioning - $f_a^r(O_t^r, a_t)$

A separate action conditioning network encodes the action at each step a_t to a one-hot vector which is then embedded into a linear layer of size C_r and finally added to each of the embedded tile vectors giving the reward state s_t^r. This process is identical to the NGPU action conditioning. The only difference is that the number of channels may differ depending on hyperparameter choices.

5.2.6 Reward Decoder - $f_d^r(s_t^r)$

In order to decode the rewards from the reward state s_t^r, a convolutional network with a kernel size of 3 and padding 1 is used, followed by two convolutional layers with a kernel size of 1, 0 padding, and the number of channels decreasing in each layer. A final convolutional layer with kernel size 3 is used to decrease the number of channels to 16 and an arbitrary height and width. Global max pooling is applied across the remaining arbitrary height and width dimensions leaving 16 outputs. These 16 outputs are then trained with categorical cross-entropy loss to predict an 8 bit binary number corresponding to the reward. We assume here that the environments in which we are training can only have integer rewards between 0 and 255. Predicting binary rewards in this way instead of predicting continuous values means that the reward can be predicted more accurately as we effectively quantize the reward to exact values.

5.3 Neural GPU enhancements

5.3.1 2D Diagonal Gating

Diagonal gating, introduced in [93], is a technique used in the NGPU architecture to pass state cell values directly to neighboring state cells. In the context of a grid-world game, information, such as tile type, could be transferred in this manner. The state of the original NGPU allows the copying of state information from the left and right cells via the diagonal gating mechanism. The state of the underlying NGPU in the Neural Game Engine adds an additional dimension, which means that the diagonal gating mechanism can be used to copy from above and below, as well as left and right. To achieve this, the state is now split into 5 parts $s_i = (s_i^1, s_i^2, s_i^3, s_i^4, s_i^5)$ and a 2D convolution operator with fixed kernels as shown in equation 5.31 is used.

$$
\begin{aligned}
s_i &= u_i \odot \tilde{s_i} + (1-u_i) \odot c_t \\
\tilde{s_i} &= (\tilde{s_i^1}, \tilde{s_i^2}, \tilde{s_i^3}, \tilde{s_i^4}, \tilde{s_i^5}) \\
\tilde{s_i^1} &= s_{i-1}^1 * \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
\tilde{s_i^2} &= s_{i-1}^2 * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \\
\tilde{s_i^3} &= s_{i-1}^3 * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\
\tilde{s_i^4} &= s_{i-1}^4 * \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
\tilde{s_i^5} &= s_{i-1}^5 * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned}
\tag{5.31}
$$

5.3.2 Selective Gating

One of the issues with diagonal gating is that the copying of the state information is uni-directional for the state values in each cell (w_s, h_s, c_s). To illustrate this issue consider the values in any sub-state s_i^x. The values in each sub-state are only shifted in a single direction. This means that sub-states that are shifted in one direction are not the same states that can be shifted in the other directions. This uni-directional flow does not allow consistent copying of state information across all directions. Intuitively this means that if a tile moves upwards, the state information it can bring to the cell above cannot be moved to the left, right or even back to the cell that it started in.

To alleviate this issue, a *selective gating* mechanism is proposed which allows the gating mechanism to copy values in any direction for any value in any cell (w_s, h_s, c_s).

The selection mechanism works by learning a classifier that, given the state tensor s_i, outputs a selection tensor $\hat{S}$ of dimensions $(W_s, H_s, C_s, 5)$ where the selection of the gating directions (up, down, left, right, center) are one-hot encoded into the last dimension. The selection tensor is created by applying a convolution operation to the state s_i with a kernel size of 3x3, stride of 1, padding of 1, and $5C_s$ output channels. The $5C_s$ channels are then reshaped into a tensor of size $(5, C_s)$, and a softmax is applied across the first dimension to give a *selection* for each of the C_s values. The selection tensor $\hat{S}$ is then multiplied by a tensor $\hat{K}$ of shape $(5, C_s, 1, W_s, H_s)$ containing 5 directionally shifted versions of the original state. This gives the new state $\tilde{s_i}$.

$$\begin{aligned} s_i &= u_i \odot \tilde{s_i} + (1 - u_i) \odot c_t \\ \tilde{s_i} &= \hat{S}\hat{K} \\ \hat{K} &= [M_u(s_i), M_d(s_i), M_l(s_i), M_r(s_i), s_i] \end{aligned} \tag{5.32}$$

The shifting operation can be achieved by convolving a fixed kernel that copies states from adjacent cells. Zero padding of 1 is applied to the state to retain its original shape. For example:

$$\begin{aligned} M_u(s_i) &= s_i \odot \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ M_d(s_i) &= s_i \odot \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\ M_l(s_i) &= s_i \odot \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ M_r(s_i) &= s_i \odot \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{5.33}$$

5.3.3 Evaluation Methodology

The experiments aim to reach pixel-perfect reproduction of original GVGAI environment games over arbitrarily long time frames for levels with any dimensions. To achieve this, the network must learn the game mechanics on a symbolic level and then be able to apply these to larger game states.

The results presented in this chapter are performed on the game *Sokoban* as it is a good example of a GVGAI game with local rules.

Two related measures are used to measure the accuracy of the game reproduction. First is the mean-squared error E_{mse} of the raw pixel outputs at each step and second is a *closest tile f1* F_t measure. The *closest tile measure* is created by firstly taking a tile map of the original observation T_m, which has dimensions (W_s, H_s) where each element in the map corresponds to an index of the set of possible tiles $\mathcal{T}$. A second tile map $\hat{T}_m$ is then created by finding the closest matching tile in the set of tiles $\mathcal{T}$ for each DxD tile in the predicted observation. The *closest tile f1* measure is calculated from the mean of the f1 scores for each of the tiles in T_m and $\hat{T}_m$. The f1 scores are generated by measuring the precision and recall of the tile predictions.

Both measures reflect each other. A lower F_t would correspond to a high E_{mse}. However, F_t is calculated as *closest* tile to a pre-generated set of tiles $\mathcal{T}$, which does not measure how close the tile predictions are to the original pixels. The measurement of E_{mse} achieves this more direct comparison.

Alongside learning pixel accuracy, the rewards given by the environment are learned. Reward error is measured by converting the real reward values to a binary representation and then calculating the cross-entropy loss. Reward accuracy is measured using the binary classifications' precision, recall, and f1 F_r score.

5.3.4 Training

To obtain accurate rollouts over long time periods for any size network, the training data is generated in a way that does not bias toward game sizes, numbers of tiles (such as walls, boxes, and holes in Sokoban), or particular RL or planning policies.

Level generation for GVGAI games has been explored in [168], [78], and [156]. However, these generators are aimed at either producing levels that help RL agents to learn or are pleasing to human players.

To generate levels for learning the environment dynamics, the probability for an agent to interact with different types of tiles must be evenly distributed. To achieve this, levels are randomly generated with height H and width W between certain values H_{min}, H_{max}, W_{min}, W_{max}. GVGAI environments typically contain 5 pre-built levels. These pre-built levels are used to generate the probabilities of each tile being placed in the environment. Tiles are positioned with these calculated probabilities with the caveat that wall tiles are always placed on the edges of the game state if this is consistent with the 5 pre-built levels. Additionally, tiles that only appear once in each level are placed only once in generated levels.

A random agent is used to generate experience data in the environment. To improve the training data distribution, each step is augmented by creating an 8-way tile-symmetrical observation and actions. Each learning step uses mini-batch gradient descent, where the batch contains symmetrical experiences. Batch sizes are fixed at 32 state transitions, giving 256 frame transitions per batch.

As the observation predictions at each time step become the inputs for the next prediction, errors can build up over time and cause the rollout accuracy to decrease rapidly. During experiments, the same Prediction Dependent Training (PDT) technique introduced in [55] coupled with a curriculum schedule was employed which increased accuracy and training stability. Observation noise is also added to training data, this was integral to achieving high accuracy.

In order to evaluate the training progress of the environment, rollouts are performed every 200 epochs using real game levels from the GVGAI environment. 3 repeats of rollouts of length 100 are performed, and F_t and E_{mse} are calculated. [1]

5.4 Experiments and Results

5.4.1 Comparison of gating mechanisms

In figure 5.14, the NGE architecture using the different NGPU gating mechanisms described in section 5.2 is shown. Even with no diagonal or selective gating, the NGE can learn accurate models of game environments. In the experiments, *Selective* gating had a small advantage in stability over long time horizons, this is also reflected in table 5.1.

5.4.2 Comparison with other methods

The best performing Neural Game Engine (NGE) model from section 5.4 is compared against several common networks from recent literature with the game Sokoban. Rewards prediction is not analysed as it is a separate network. The network architectures that are compared are the following:

FeedForward (FF) This model replaces the NGPU module with two feed-forward convolutional layers with kernel size of 3, stride 1 and padding 1. This is the equivalent of the *basic block* used in [297] when training Sokoban. The model compared does not use pool-and-inject layers as Sokoban has no long-distance dependencies that require global state changes. This model is commonly used as the deterministic component of generative state-space architectures and is well suited to deterministic grid environments.

Recurrent Environment Simulators (RES) The state of the game is encoded into a latent state using an auto-encoder. This latent state then forms the input to an LSTM unit which can store past state information in its hidden

[1]All training and testing is performed on a single Ubuntu 18.04 machine with an NVIDIA 2080ti GPU, Intel® Core™ i7-6800K CPU and openBLAS (0.2.20) libraries installed.

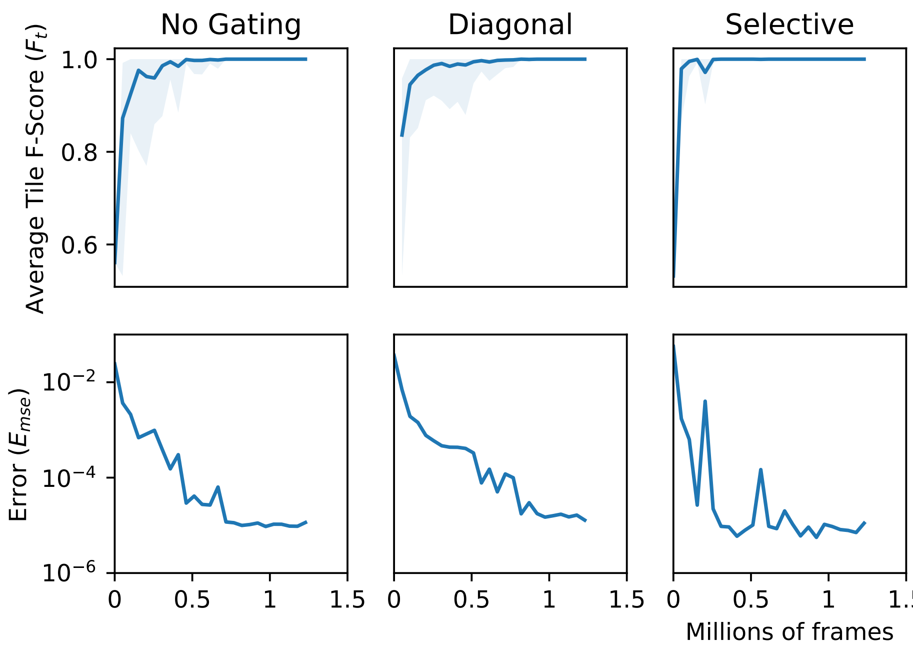


Figure 5.14: This figure shows the average F1 score ($F1_t$) and mean squared error (E_{mse}) at intervals of 200 epochs over training. Each score is calculated over 100 evaluation steps and averaged over three runs. Selective gating trains the fastest and is the most accurate out of the 4 tested methods.

state. This model is equivalent to the Recurrent Environment Simulator (RES) [55] and models that use an auto-encoder to create a latent state.

Stochastic State Space (sSS) The most complex model, like NGE, heavily uses cellular automata-like layers which encode pixel information into a compressed grid. The model differs from NGE in that it works with continuous and stochastic environments, and therefore uses sampling in order to produce the output observations.

Figure 5.15 shows the comparison of these 4 methods with the same input data and the number of epochs. The training in this experiment is limited to random grids of fixed size (10x10). This is due to the fact that RES and sSS models contain architectural components that cannot generalize to different sized grids. Each method trains to high accuracy quickly, followed by a plateau in decreasing error, leading to maximum accuracy. In the case of Sokoban, FF, sSS and NGPU methods have a slight advantage as Sokoban is naturally suited to local modeling. However, the sSS model is disadvantaged by the fact that it contains stochastic components that are trying to model completely deterministic state transitions.

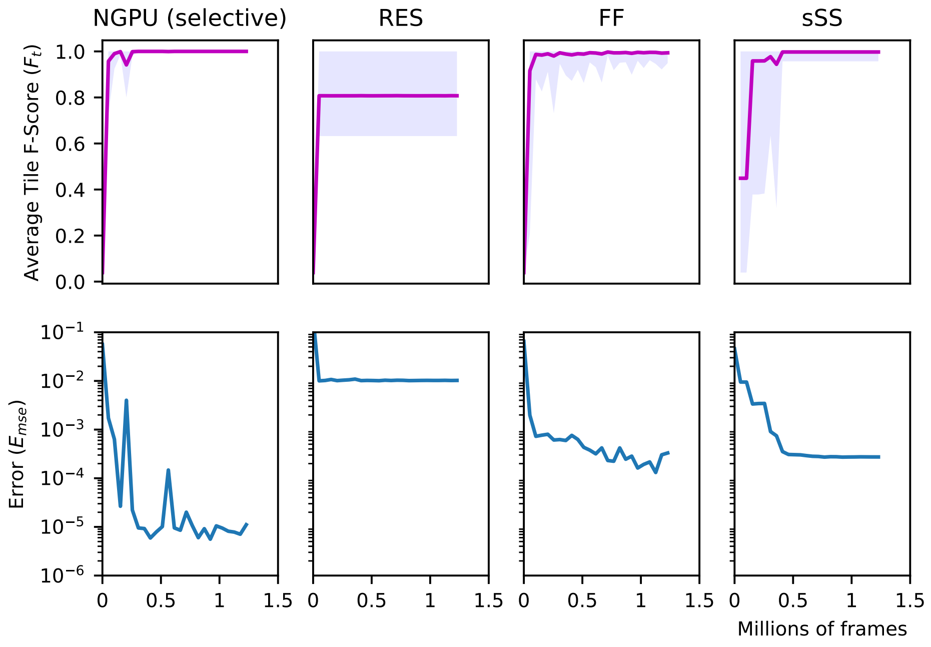


Figure 5.15: This figure shows the average F1 score ($F1_t$) and mean squared error (E_{mse}) at intervals of 200 epochs over training. Each score is calculated over 100 evaluation steps and averaged over three runs. NGPU achieves the lowest E_{mse} and highest average F_t score.

5.4.3 Ablation Testing

An important feature of many games is that many interactions between adjacent cells can be dependent on other surrounding cells. For example, in Sokoban, when pushing a movable block against a wall, a 3x3 grid around the location of the agent will not take into account the wall when calculating the next state of the cell currently occupied by the agent. However, the NGPU accounts for non-local interactions when it iterates during a single time step. This effectively lets cells share information during the processing of a single state. With $n = 2$, the NGPU can share information from more adjacent cells, encompassing the wall that the block cannot be moved past. Other models such as those used in [47] [297] use similar techniques, but use fixed networks with different convolutional network sizes and apply residual layers. Using a NGPU with multiple iterations removes the requirement for multiple layers of convolutions and residual connections, making the network much simpler and smaller.

In [93], diagonal gating is used to share state information between adjacent cells. As described in section 5.32, this only allows single-direction information flow, which reflects in the higher error rate of NGE models using diagonal gating.

To test that the iteration of NGPU is vital for information flow in local

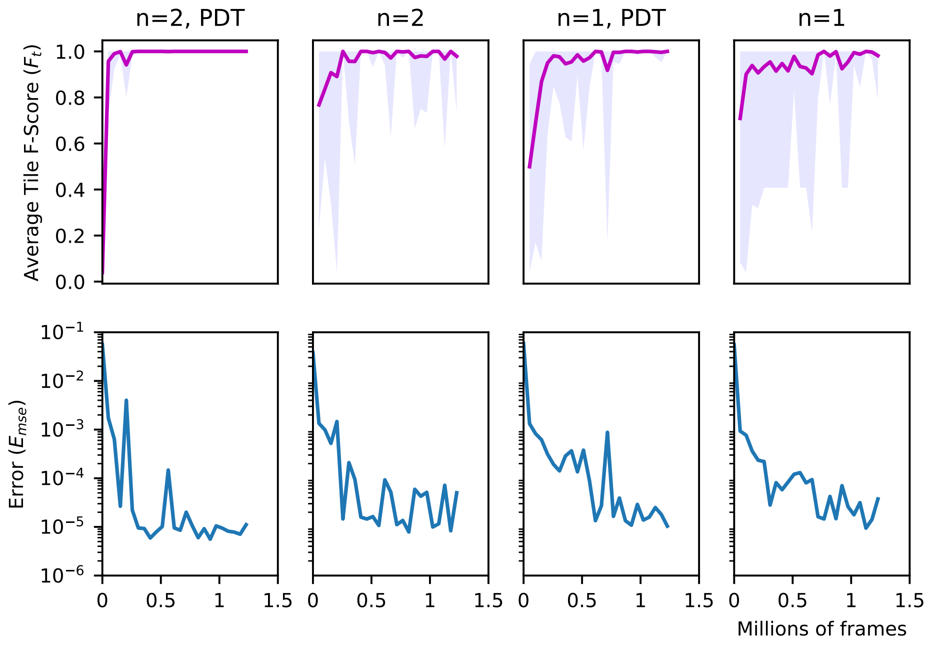


Figure 5.16: Learning an accurate model of the Sokoban environment is dependent on having multiple iterations of the NGPU units and also training over multiple states. When the network is restricted to a single iteration $n = 1$ or is trained without PDT, the accuracy suffers. The results shown here are the rollout accuracy measurements against the 5 original hand-built GVGAI environments, taken every 200 epochs during training.

interactions, two experiments are performed under all the same conditions as the high-performing models. One with the modification that only a single NGPU step and no PDT are configured during training. The other is with a single NGPU step, but using PDT described in section 5.3.4. The second experiment aimed to rule out that local information could be transported through pixels. The results of this are shown in figure 5.16.

In both the 2-step and 2-step+PDT experiments, the accuracy achieved is high, but with the single step options, the accuracy achieved plateaus at a much lower value and the prediction error remains high. This result shows that multiple steps of the NGPU are vital to achieving high accuracy. It's also important to note that the 2-layer FF model in figure 5.15 also could not achieve this high accuracy.

5.4.4 Generalising to different size grids

To test the generalization ability of the trained NGE, the models trained in section 5.3.4 are used to play several levels with much larger dimensions than those during training. These larger models are then compared against the orig-

Grid Size	$max(E_{mse})$	F_1
30x30	7.5e-6	**1.0**
50x50	8.3e-6	**1.0**
70x70	7.9e-6	**1.0**
100x100	7.9e-6	**1.0**

Table 5.1: The maximum mean squared error and closest tile error for 500 steps averaged over 10 repeats. NGPU with *Selective* gating obtains high tile accuracy in all of the generalization tests.

inal GVGAI environment with an identical starting state and action list. The two methods (E_{mse} and E_t) of measuring the accuracy of these models are used as described in section 5.3.4. For each model, the two measures are calculated for each step up to 500, and an average of the measures are taken over 10 repeats. These results are shown in table 5.1

5.4.5 Results on GVGAI games

The results of training Neural Game Engine on several GVGAI games are shown in table 5.2. The rollouts follow the same setup described in section 5.3.3 Games that result in F_t scores of **1.0** show that the underlying game rules are learned accurately and the NGE does not make any mistakes when tested. Reward F1 scores F_r can be interpreted in the same way. Most of the tested games achieve high accuracy, however, there are some game mechanics that cannot be supported by the NGE without modifications. As an example, *clusters* completely fails to learn the reward function. Rewards are fairly common in the game and the forward model itself learns accurately, so the reason for this is unclear. The game *aliens* is included as an example stochastic and partially observable game (the enemies randomly shoot at the player and the enemies spawn from a location that has no visible markers). The reward function $min(F_r)$ of aliens is partially learned by NGE, however, the $min(F_t)$ score is 0.73 meaning that just over a quarter of the tiles are predicted incorrectly.

5.5 Discussion

There are several interesting applications for games trained with the NGE architecture, for example the fact that games can be learned with high accuracy over long time horizons, these can be used in planning algorithms. Additionally, because these games also run entirely on the GPU, the sample rate and parallelization ability mean that they can be used as efficient environments for reinforcement learning experimentation.

Game	$max(E_{mse})$	$min(F_t)$	$min(F_r)$
sokoban	7.5e-6	**1.0**	**1.0**
cookmepasta	9e-4	0.98	0.83
bait	5.2e-4	0.97	0.99
brainman	3.6e-4	0.97	**1.0**
labyrinth	1.6e-5	0.97	**1.0**
realsokoban	1.8e-3	0.86	**1.0**
painter	4.6e-6	**1.0**	**1.0**
clusters	1.3e-5	**1.0**	0.0
zenpuzzle	8.2e-6	**1.0**	**1.0**
aliens	5.1e-3	0.73	0.85

Table 5.2: The maximum mean squared error $max(E_{mse})$, minimum closest tile f1 $min(F_1)$ and minimum reward f1 $min(F_r)$ for 100 steps over 3 repeats.

There are two main limitations that the NGE architecture suffers from: its lack of ability to model stochastic game elements, and global state-changes. Further experimentation and research is required to achieve these goals. One approach could be to use NGPU modules in place of the deterministic size preserving layers in sSS models.

One large area for improvement for the Neural Game Engine is that the statistical method of level generation and random agent movement does not produce enough examples for some local patterns and can produce unsolvable or unplayable levels. In many cases, tweaking the random level generation parameters is enough to give the NGPU a distribution which greatly improves the accuracy of training. Improving the data distribution of local states to train the NGPU is an area which could greatly be improved. Using curiosity driven agents, or planning agents may provide much better data distributions for learning rewards, but may avoid areas of low rewards and therefore not learn the full game dynamics.

Another area of improvement would be that the Neural Game Engine only predicts a single time step in the future, therefore events that do not specifically change the observational state are completely lost. For example, in some games the agent picks up a key and then the agent tile changes to show the agent holding a key. Once the agent has a key, the agent can open a door. NGE learns these dynamics well and learns that if the agent lands on a tile with a key, it changes to an agent with a key and can then interact with a door. However if the fact that the agent is holding a key does not change the agent tile, NGE has no knowledge of this at the next step and therefore the information is lost. This could be fixed by following the latent state space model training techniques used in [47], [6] and [16] where future observations are predicted several steps in the future without decoding the visual information between steps.

5.6 Conclusion

In this chapter, the Neural Game Engine architecture is proposed as a method of learning accurate forward models for grid-world games. The Neural Game Engine architecture, which is built upon the Neural GPU, learns a set of underlying local rules that can be applied over several iterations rather than stacking layers with different parameters. Improvements to the Neural GPU architecture such as selective gating are introduced which enable it to be applied to predicting the forward dynamics of games. This chapter shows that this method has many advantages: fast learning time; high accuracy over long time-horizons and fast and easily parallelized execution. The Neural Game Engine shows higher accuracy at predicting the state transitions in the game Sokoban when compared to similar state space models that are used in several model-based reinforcement learning applications. Additionally the Neural Game Engine is shown to generalize well to different game environment dimensions not seen during training.

Chapter 6

Equivariant Data Augmentation

Geometric Deep Learning is a promising direction in understanding how the structure of neural networks and training of neural networks can be analyzed using the mathematics of geometry. In image recognition tasks, affine transformations of input data, such as scaling, translation, and rotation, should not change the output of image classification, i.e. a small cat and a large cat picture should both classify as a "cat". However, when the image classifier is a neural network, it must be trained with significant data for high classification accuracy. This accuracy can be improved by the augmentation of the input dataset, for example, by adding rotated and scaled versions of the original dataset images. Alternatively, the neural networks can be designed so that their classification accuracy is *invariant* to these transformations. In this thesis, we are particularly concerned with finding geometric priors in reinforcement learning environments, which we can use to enhance training or understand issues with particular algorithms. More specifically, we concentrate on the geometric priors found in the perception of environmental agents, such as rotational and dihedral symmetry of observations. In previous works it has been shown that while the value function is *invariant* to transformations, the policy outputs are *equivariant*, meaning that the as the input is transformed, we must apply an equivalent transformation to the output for consistency. However, in several algorithms, it is unclear how to apply these transformations. We show two potential solutions to the off-policy correction algorithm V-Trace [86] which we call *naive behaviour augmentation* and *behaviour replay augmentation.* We provide several ablation experiments, showing the benefit of these additional techniques.

6.1 Background

Reinforcement learning requires a large amount of varied experience data in order to learn generic policies. One method of artificially increasing the amount of data, without requiring more environment trajectories is to augment the data. Data augmentation techniques have a wide range of implementations and can be as simple as just adding noise to inputs, or as complex as applying linear transformations to the input data. In this section, we start by exploring commonly used data augmentation techniques in supervised learning. We then move on to how these methods can be applied to reinforcement learning and show that different formulations of data augmentation are required under certain types of transformation that are not required in supervised learning. These differences arise in the policy network when certain augmentation transformations are applied to the inputs.

6.1.1 Data augmentation in Supervised Learning

Data augmentation is a well-known technique used to improve the performance and stability of training deep neural networks [52]. Data augmentation in supervised learning uses a set of stochastic or deterministic transformation functions to produce additional samples of data to be included in the training. Augmentations are typically designed to be invariant [262], meaning that they should result in the same output as the unmodified inputs. An example of this would be adding translation, scaling, rotation, or noise to images [175], or replacing words in text with synonyms in textual data [298].

Due to the stochastic nature of classifier predictions, naively using data augmentation to improve model performance can still result in misclassification due to unseen perturbations or adversarial examples [278]. To alleviate this issues, regularization methods that try to preserve semantic information in latent variables [317, 53, 141], and methods for automating augmentations [67, 68, 313], have been proposed. These methods attempt to apply data augmentation in a more principled manner.

Consistency Regularization Consistency regularization is a popular method for regularizing data augmentation [19, 237]. It is particularly useful when the data augmentation methods are stochastic, such as adding noise or using dropout during training. Consistency regularization typically adds a loss function to the output of classifiers that encourages the output to stay robust under data augmentation. This loss function can take the form of a L^2 norm or, in the case of stochastic classifiers, the KL-divergence is commonly used [309].

Consistency regularization can also be performed under geometric augmentations of inputs such as scaling and rotation. For example, in [264], consistency regularization is used to constrain the latent representation of a Variational Autoencoder [171] so that latent representations of transformed images are similar, regardless of their scale or orientation. Another example use-case is in Generative Adversarial Networks [102], where consistency regularization is used to stabilize training [316]. This is achieved by adding a loss function to encourage the discriminator to produce the same output for augmented images.

6.1.2 Data Augmentation in Reinforcement Learning

Similar success using data augmentation has been shown in Reinforcement Learning. In many reinforcement learning domains, the state space can be augmented similarly to the inputs in supervised learning. Techniques such as adding noise, cropping, scaling, and manipulating the color of pixels are common [181, 174, 310, 225]. Data augmentation in Reinforcement learning is particularly useful in improving generalization and sample efficiency in generative environments, for example, procedurally generated environments [61] and environments designed specifically to contain *distractions* [272].

Domain randomization [145, 282, 4], a method similar to data augmentation, has been used to improve robustness in robotic domains. Simulating robotic environments for training is notoriously difficult as there are many unknown sources of noise and dynamics that are hard to reproduce accurately. In addition to this, every physical device has imperfections that can dramatically change the performance when controlled with the same model. Domain randomization adds noise by changing various simulation parameters, including friction, lighting, colors, and even gravity, to train models that are robust to different physical characteristics when deployed.

Data augmentation has also been used to encourage efficient data representations for reinforcement learning. For example, CURL [269] uses contrastive unsupervised learning [141, 53] to generate an accurate latent state model from pixels, which can then be used as input to a reinforcement learning algorithm.

Additionally, in model-based reinforcement learning, data augmentation has been used to create accurate recurrent models that can predict future states from several augmented initial states [253, 285].

6.1.3 Equivariant Networks in Supervised Learning

One desired property of many neural networks, such as image classifiers, is that of **invariance** and **equivariance**. An invariant neural network is one where the output does not change when the input is modified by a particular class of transformations, more formally $f(i) = f(F_{i_p}(i))$, where and i is the input, f is a neural network and F_{i_p} is a class of bijective, linear transformations indexed by p. Similarly, equivariant neural networks hold a property that a particular class of input transformations corresponds to another class of output transformations: $F_{o_p}(f(i)) = f(F_{i_p}(i))$ [64, 63, 299, 46]. The transformations used are typically geometric in nature, such as functions of symmetry groups.

Recent work has also shown that equivariant neural networks can significantly outperform networks trained with data augmentation under the same geometric transformation classes. [299]

6.1.4 Equivariant Networks in Reinforcement Learning

Equivariance and invariance are particularly useful priors in RL, as environment observations commonly involve symmetries that can be translated to equivalent symmetries in resulting policies [199]. As an example, in actor-critic reinforcement learning, combinations of equivariant and invariant layers can be used in the actor and critic. In grid-based environments with rotational symmetry and actions that control the four directions of movement, any rotation of the input state $F_s(s)$ can directly result in a permutation of the logits in the actor $F_l(l)$, giving an equivariant policy $F_l(f_{actor}(s)) = f_{actor}(F_s(s))$. In this case, the value function at any given state is also invariant to the rotation $f_{critic}(s) = f_{critic}(F_s(s))$.

When an environment has geometric symmetries that can be represented as group transformations, such as rotations and flips, the underlying Markov Decision Process (MDP) can be modeled as an MDP homomorphism [229, 230, 286]. Symmetries in behavior in multi-agent settings can also be exploited using the same frameworks [287, 107]

6.2 Invariant and Equivariant Augmentation

We define a **trajectory** $\mathcal{T}$ as a sequence of tuples (a, s, r) consisting of action a, state s, and return r. These trajectories (also referred to as rollouts) form episodes or batches and are used during the training process. Data augmentation is applied to trajectories by performing transformations on the states $F_s : S \to S$ and actions $F_a : A \to A$. Where the state and action transformation functions are bijective.

Using this notation, we can interpret MDPs under data augmentations as MDP Homomorphisms [229, 230]. This allows us to link the transformation functions F_a and F_s under the following rules:

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{A} : R(a, s) = R(F_a(a), F_s(s)) \tag{6.1}$$

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{A} : T(s', a, s) = T(F_s(s'), F_a(a), F_s(s)) \tag{6.2}$$

Additionally, we can specify that if the states in a trajectory are transformed by F_s, the probability distribution of the actions in those states must be transformed by the associated action transformation function F_a. This gives a constraint for policies that are trained under augmented trajectories.

$$\pi(F_a(a)|F_s(s)) = \pi(a|s) \tag{6.3}$$

In policy gradient methods, we can describe the policy as being parameterized by logits for each action $l_s = \{l_{s,0}, l_{s,1}, \ldots, l_{s,k}\}$ where $k = |A| - 1$ and s is the current state. Typically the policy $\pi(a, s)$ can be written as a function of these logits, commonly referred to as the *Softmax* function:

$$\pi(a|s) = \frac{\exp(-l_{s,a})}{\sum_{i=0}^{k} \exp(-l_{s,i})} \tag{6.4}$$

It then follows that the augmented policy can be recovered by calculating the policy using augmented inputs:

$$\pi(F_a(a)|F_s(s)) = \frac{\exp(-l_{F_s(s),F_a(a)})}{\sum_{i=0}^{k} \exp(-l_{F_s(s),i})} \tag{6.5}$$

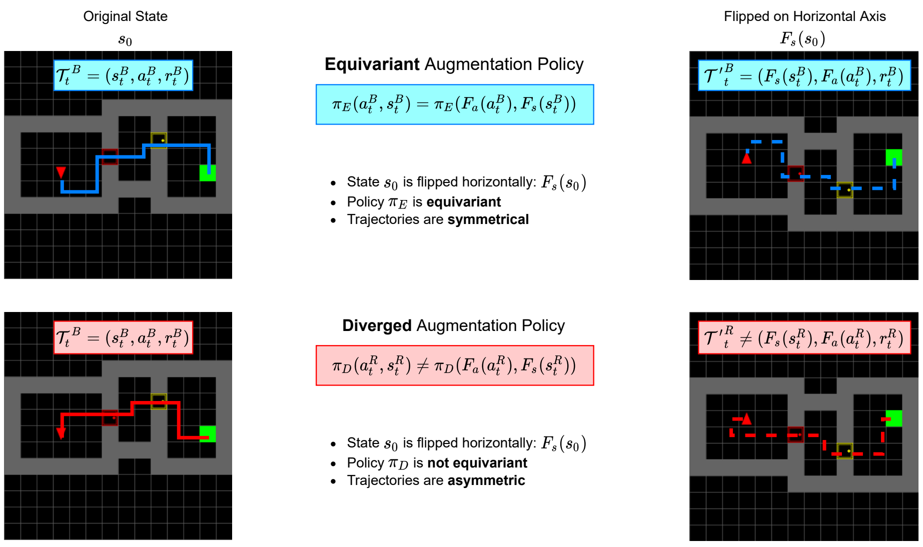
or by permutation of the logits $F_l(l_s) = P_l \cdot l_s$ where P_l is a permutation matrix:

$$\pi(F_a(a)|F_s(s)) = \frac{\exp(-F_l(l_s)_{F_a(a)})}{\sum_{i=0}^{k} \exp(-F_l(l_s)_i)} \tag{6.6}$$

Invariant Augmentation We describe augmentations as *Invariant* under transformations if they follow the following condition: An augmentation is *Invariant* if the augmentation applied to the input should not result in a change of the output policy. This is the most studied form of augmentation in both supervised and reinforcement learning [52]. In this case, F_a is an identity transformation and has no effect on the action distribution of the policy. Substituting Identity functions for F_a into equation 6.3 we have:

$$\pi(a|F_s(s)) = \pi(a|s)$$

Examples of this include colour jitter and greyscale [181] as they only perform transformations on the state itself.


Original State
s_0
Flipped on Horizontal Axis
$F_s(s_0)$
$\mathcal{T}_t^B = (s_t^B, a_t^B, r_t^B)$
Equivariant Augmentation Policy
$\pi_E(a_t^B, s_t^B) = \pi_E(F_a(a_t^B), F_s(s_t^B))$
$\mathcal{T}'^B_t = (F_s(s_t^B), F_a(a_t^B), r_t^B)$
State s_0 is flipped horizontally: $F_s(s_0)$
Policy π_E is **equivariant**
Trajectories are **symmetrical**
$\mathcal{T}_t^B = (s_t^B, a_t^B, r_t^B)$
Diverged Augmentation Policy
$\pi_D(a_t^R, s_t^R) \neq \pi_D(F_a(a_t^R), F_s(s_t^R))$
$\mathcal{T}'^R_t \neq (F_s(s_t^R), F_a(a_t^R), r_t^R)$
State s_0 is flipped horizontally: $F_s(s_0)$
Policy π_D is **not equivariant**
Trajectories are **asymmetric**
Figure 6.1: It is assumed that the policy predicted from an augmented state $F_s(s_0)$ should follow a trajectory $\mathcal{T}'$ that is a transformation of the original trajectory $\mathcal{T}$ predicted from the original s_0. We refer to this as **equivariant** augmentation, shown in blue. However, this is not always the case. When policies are naively trained with data augmentation, they can produce trajectories that break this assumption. We refer to these policies as **diverged** augmentation policies, shown in red.

Equivariant Augmentation In the case where the transformation of the state results in a transformation of the actions, 6.3 cannot be simplified. This is referred to as an equivariant augmentation (See Figure 6.1 for an example). For example, equivariant augmentations include those in which F_s is a rotation or flip transformation. Equivariant augmentation functions can also exist as function groups, such as rotations of 90 deg of the state s which result in equivalent transformations of the actions a [286].

Value Function Invariance Finally, under data augmentation, the value function $V(s)$ is a scalar and therefore should be invariant to any transformation of the state:

$$V(F_s(s)) = V(s) \tag{6.7}$$

One of the most sought-after goals of deep reinforcement learning (RL) research is generalization. RL has successfully solved problems where the set of possible scenarios (i.e levels, or states) is relatively small. Unfortunately, successes in these smaller domains can be prone to over-fitting, and changing small features of the environment, such as introducing new levels, may lead to unexpectedly low performance in the modified environment. Even if the modified environment is extremely similar to the original environment, this can cause issues for the trained agent. The solution to this is to train the agent with a large amount of data, which can encompass the distribution of the unknown levels.

Challenging procedurally generated domains such as ProgGen [61], MiniGrid [54] and Nethack [177, 241] have been proposed as test beds for the generalization performance of reinforcement learning agents, as these environments consistently generate new levels, add background noise, or introduce distracting factors such as background images. The goal of training RL agents in these environments is for the agents to be able to generalize to unseen levels and ignore parts of the environment that are not useful for high-performing agents.

6.2.1 Augmentation Groups

Data augmentations commonly form group functions. For example, a group augmentation for rotating the input by 90 degrees is a collection of 4 functions for the actions, states, and logits: $F_a = \{\mathbf{I}, F_{a_{90}}, F_{a_{180}}, F_{a_{270}}\}$, $F_s = \{\mathbf{I}, F_{s_{90}}, F_{s_{180}}, F_{s_{270}}\}$ and $F_l = \{\mathbf{I}, F_{l_{90}}, F_{l_{180}}, F_{l_{270}}\}$. Calculating the policy and value function regularization, in this case, is a sum of the contributions from each augmentation function.

Even in simple invariant data augmentation settings, the augmentation functions can be structured in this way. For example, just changing the colour of the pixels, such as grayscaling: $F_a = \{\mathbf{I}, \mathbf{I}\}$, $F_s = \{\mathbf{I}, F_{s_{greyscale}}\}$ and $F_l = \{\mathbf{I}, \mathbf{I}\}$.

6.3 Data Augmentation using IMPALA

In this section, we propose two possible data augmentation methods when using IMPALA and show that assumptions required for data augmentation to be applicable do not hold in many cases. Subsequently, we propose regularization methods which constrain these assumptions.

IMPALA separates the collection of data from the process of training by using multiple *behaviour* policies, which collect trajectories in the environment. These trajectories are then sent to a central location where a *target* policy is updated. Additionally in IMPALA, trajectories described in section 6.2 contain the logits l_s of the policy for the state s. We provide a more in-depth explanation of IMPALA in section 4.3.1

The parameters of the *behaviour* policies are updated by periodically copying the parameters from the *target* network. This update procedure happens asynchronously. This can lead to some *behaviour* policies diverging from the *target* policy, effectively becoming "out-of-date". IMPALA uses the v-trace algorithm to down-weight the training effects from trajectories generated by diverged *behaviour* policies. At the core of the v-trace algorithm, truncated importance sampling weights are calculated using a ratio of the *behaviour* and *target* policies $\rho = \min(\bar{\rho}, \frac{\pi(a|s)}{\mu(a|s)})$ and $c = \min(\bar{c}, \frac{\pi(a|s)}{\mu(a|s)})$, where $\bar{\rho}$ and $\bar{c}$ are hyperparameters typically set to 1. During training, it is assumed that when the *behaviour* policy has not diverged, $\pi(a, s) = \mu(a, s)$ and therefore $\rho = 1$. However if the policies have diverged, $\pi(a, s) \neq \mu(a, s)$ and $\rho \leq 1$, $c \leq 1$. The ρ and c values then weight the contribution of *behaviour* training samples.

Naive Behaviour Augmentation We define naive augmentation as an augmentation that is applied to behaviour trajectories just before the point of training, producing an augmented trajectory $\mathcal{T}_n$ where each tuple has augmented state, action and policy logits $(F_a(a), F_s(s), F_l(l_s), r)$. The augmented behaviour policy $\pi_n(F_a(a), F_s(s))$ is given by equation 6.6.

Behaviour Replay Augmentation In behaviour policy replay augmentation, instead of producing the augmented behaviour policy by permuting the logits using equation 6.6, the logits for augmented trajectories are re-calculated by passing the augmented trajectories through the behaviour policy, giving an augmented trajectory $\mathcal{T}_r$ as $(F_a(a), F_s(s), l_{F_s(s)}, r)$. In this case the augmented behaviour policy $\pi_r(F_a(a), F_s(s))$ is calculated using equation 6.5.

In both trajectories $\mathcal{T}_n$ and $\mathcal{T}_r$, the augmented actions $F_a(a)$ and states $F_s(s)$ are available. This means that the augmented target policy can be calculated using $\mu(F_a(a), F_s(s))$. We can now calculate the v-trace importance sampling weights for **naive behavior augmentation** as $\frac{\pi_n(F_a(a), F_s(s))}{\mu(F_a(a), F_s(s))}$ and **behaviour replay augmentation** as $\frac{\pi_r(F_a(a), F_s(s))}{\mu(F_a(a), F_s(s))}$. Figure 6.2 shows the various constructions of possible policies used in the proposed augmentation methods in IMPALA.

6.3.1 Augmentation Constraint Assumptions

In equation 6.3, we state a constraint that is implied when training with data augmentation, however in practice, the assumption of this required constraint is likely to be violated.

In a randomly initialized policy, $\pi_\phi(a|s)$ and $\pi_\phi(F_a(a)|F_s(s))$ will be roughly similar as the probabilities of all actions are equal, however during training, $\pi_\phi(a|s)$ and $\pi_\phi(F_a(a)|F_s(s))$ can diverge. This divergence can happen if there are multiple policies that can lead to high rewards, leading to trajectories that are not symmetric, even if the state is symmetric. When these policies diverge,

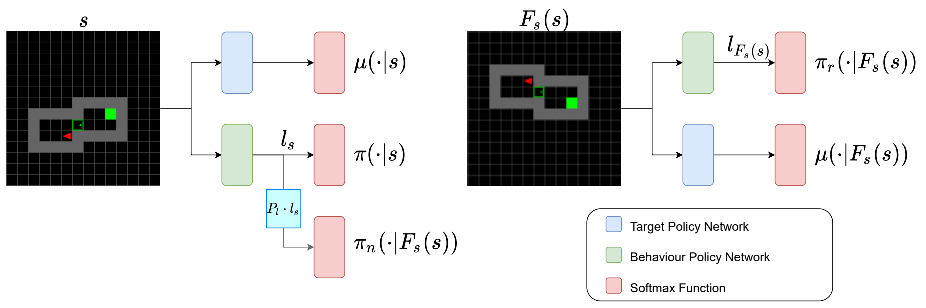


Figure 6.2: How augmented policies are produced for naive π_n and behaviour replay π_r methods in the minigrid Multi-Room environment [54]. Additionally, the target policies μ for the original and augmented states s and $F_s(s)$ are shown. These policy outputs are then used to generate the clipped ρ and c values. The augmentation function $F_s(s)$ in this particular example is a horizontal axis flip.

the assumptions that are required for the augmentation methods in equations 6.6 and 6.5 no longer hold. Figure 6.1 shows a simple example of this issue. This leads to worse policy performance when being trained with augmented data.

In section 6.5, we show empirical evidence for this policy divergence under data augmentation.

Policy Regularization

In order to encourage the augmentation constraint to hold, we expand upon the regularization terms introduced in [225].

For each augmentation transformation, we introduce a regularization term which encourages policies under augmentation to be *equivariant*. We pose the policy constraint in 6.3 as the KL divergence loss $\mathcal{L}_{\pi_F}$ between the policy π and the policy derived from augmented data $\hat{\pi}$. We replace $\hat{pi}$ depending on the method used: $\hat{\pi} = \pi_n$ for **naive behaviour augmentation** and $\hat{\pi} = \pi_r$ for **behaviour replay augmentation**

Additionally, the contribution of this loss function is scaled using the normalized entropy of the policy: $H_\pi(s) = (1 - H(\pi(.|s)))$. This scaling increases the contribution of policies that have lower entropy, leading to only more meaningful samples to be used in calculating $\mathcal{L}_{\pi_F}$.

$$\mathcal{L}_{\pi_F} = \beta H_\pi(s) D_{\text{KL}}(\pi(a|s))||\hat{\pi}(F_a(a)|F_s(s))) \tag{6.8}$$

Value Regularization

A similar assumption to equation 6.3 also applies to the value function under data augmentation. As the calculation of the value function, V only requires the state, we can frame equation 6.7 as a regularization term similar to the one proposed for the policy:

$$\mathcal{L}_{V_F} = \alpha \sum_{k=1}^{K} MSE(R(s), V(F_s(s))) \tag{6.9}$$

6.4 Equivariant Networks

Equivariant neural networks solve the issues raised in previous sections when the data augmentations can be organized into group transformations. Group equivariant neural networks, such as those proposed in [286, 299, 63, 64], allow neural networks to encompass natural groups in the data such as symmetries, scaling, and translations in the structure of the neural network itself, rather than try to learn it from large amounts of data, including augmentations.

In reinforcement learning, group equivariant neural networks can be constructed to specifically enforce the rules that we are trying to learn when using data augmentation. Given we know the transformations $F_s(s)$, $F_a(a)$, and $F_l(l)$ that are required to perform data augmentation, we can create an equivariant neural network that encodes these transformations in its structure.

Policies built using equivariant neural networks enforce the conditions in equation 6.3 and 6.7, which means that value and policy regularization are not required.

6.5 Experiments

Environment To compare the baseline, augmentation, and equivariant methods, we use the Minigrid Multi-Room environment. Multi-Room provides a procedurally generated, grid-world maze in which an agent must traverse several rooms, opening doors between each room and then finally reaching a goal state. The agent can see a 14x14 fully observable top-down view of the set of rooms. Each grid cell contains a one-hot representation of the state at that cell, including the color of doors and the orientation of the agent itself. A Reward of 1 is given to the agent once the goal has been reached, minus a small penalty proportional to the number of steps taken to reach the goal.

Augmentations The two data augmentations used are referred to as $\boldsymbol{AugC4}$ and $\boldsymbol{AugS2}$. These augmentations are the group of 90-degree rotations $C4$ and the symmetry group formed by a horizontal axis flip $S2$. It is important to note that these augmentations are chosen as the Multi-Room environment has natural $D4$ (horizontal, vertical, and rotational) symmetry. We can also construct equivariant neural networks that encompass these geometric symmetries as a comparison point.

Networks The network trained with data augmentation in all cases is the same architecture as used in [149]. The equivariant neural network has the same structure, but the number of channels in convolutions and layer sizes are adjusted to result in a similar number of trainable parameters. The Equivariant networks are created using the *symmetrizer* method introduced in [286] for both $C4$ and $S2$ equivariance groups. These networks are referred to as $\boldsymbol{EqC4}$ and $\boldsymbol{EqS2}$ respectively. Finally, we train a baseline neural network with no augmentation; we refer to this baseline using B.

In all experiments, we train for 25M time steps of the environment; we additionally run every experiment 10 times, each with different seeds, and then average the results across all runs. Similarly to [149] we train and use 4000 seeds for level generation and evaluate on an unlimited set of randomly generated levels. Our evaluations are performed every 10 training epochs and averaged across 8 levels. Unless otherwise stated, the average episode reward means (referred to as V in our experiments) has been calculated from the evaluation levels.

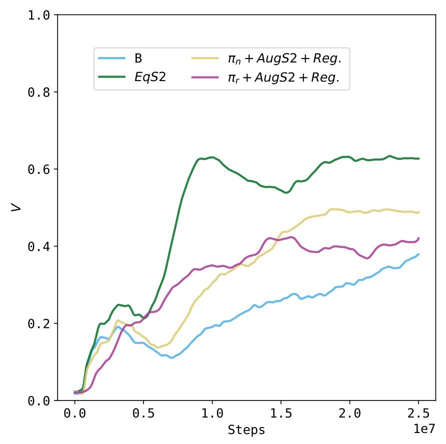
6.6 Results and Discussion

6.6.1 Augmentation vs Equivariance

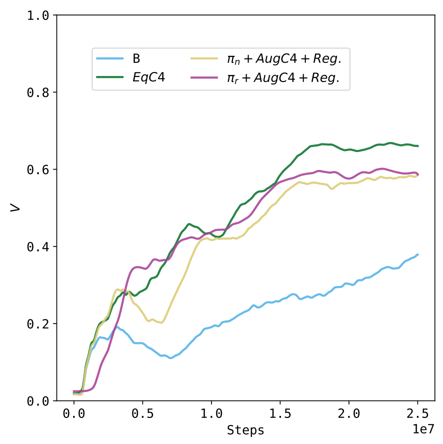
In all of our experiments, both of our data augmentation methods outperform the baseline. We also notice that the larger augmentation group $C4$ outperforms the group $S2$; this is likely due to the simple fact that C4 contains more data due to twice the number of augmented trajectories that are processed. On top of this, equivariant neural networks further outperform the data augmentation methods. It's important to note that equivariant neural networks, in this case, did not use any augmented data or regularization at all, so their high performance is significantly more sample efficient as well as more general. Figure 6.3 shows the training curves of these experiments. We did not include the results of the augmentation methods that do not use regularization, as we compare them separately with the regularized versions as part of an ablation study in the next section.

6.6.2 Ablation Study

In Figure 6.4 It can be seen that the non-regularized version of the augmented policy methods π_n and π_r have significantly worse performance than those with



(a) Horiz. flip $S2$ group transformations.



(b) 90° rotations $C4$ group transformations.

Figure 6.3: These results show the evaluation performance during training of the two proposed augmentation methods in IMPALA and the equivariant neural network. We group the experiments by the augmentation transformation groups $S2$ (6.3a) and $C4$ (6.3b) so they can be fairly compared. We also include a baseline comparison B which uses a standard neural network architecture but with no augmentations to the data.

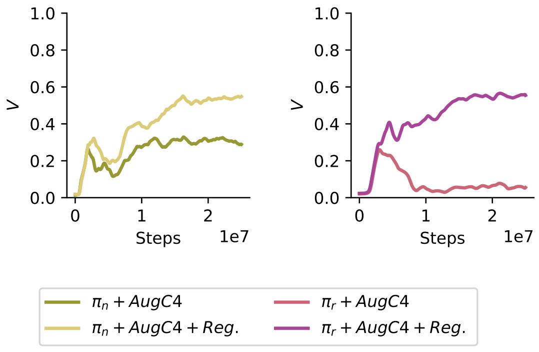


Figure 6.4: Here we show the effect of using regularization during training on policy performance in both naive π_n and behavior replay π_r methods. In these experiments, we use the augmentation group $C4$ and measure the mean evaluation episode reward V. The regularized results are the same as those in figure 6.3b Surprisingly without regularization, the behavior replay π_r method performs poorly.

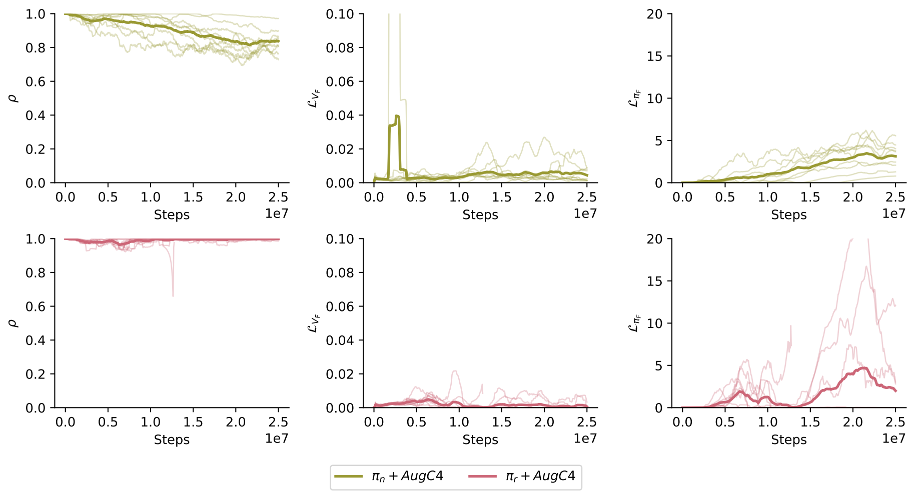


Figure 6.5: When augmentation methods are applied without the regularization methods proposed in section 6.3.1, the predicted policies can diverge, which in turn causes the degradation in the ρ values in IMPALA and training performance. The losses $\mathcal{L}_{V_F}$ and $\mathcal{L}_{\pi_F}$ are calculated and shown above in the middle and right plots, but are not applied to training. Policy divergence is particularly bad when using behaviour replay augmentation. We show the results from 10 different seeds for each measure, and highlight the average across the seeds.

the regularization methods from section 6.3.1 applied. When data augmentation is performed in a naive way by assuming that the properties 6.3 and 6.7 will be learned, the performance of the learned policy can be poor.

We can further analyse these assumptions by plotting the losses in equations 6.8 and 6.9, without applying these as regularization terms. In addition we can measure the clipped ρ values to view the effect on training in IMPALA. This can be seen in Figure 6.5.

We then analyse the same measures, but with the policy and value regularization applied, the results of these are in Figure 6.6.

6.6.3 No Regularization

With no regularization terms, there is nothing that is specifically constraining the policy or value function to hold the required equivariance and invariance assumptions. The ρ values diverging for the **Naive** policy π_n in figure 6.5 can be explained by the fact that this policy is generated by permuting the logits of the policy for a state that is not transformed (equation 6.6). When the ratio of π_n and μ is calculated, initially these policies are the same as they are essentially random. These initially random policies result in a ratio close to 1. In addition to this, the KL divergence measure $\mathcal{L}_{\pi_r}$ between $\pi_n(F_a(a), F_s(s))$

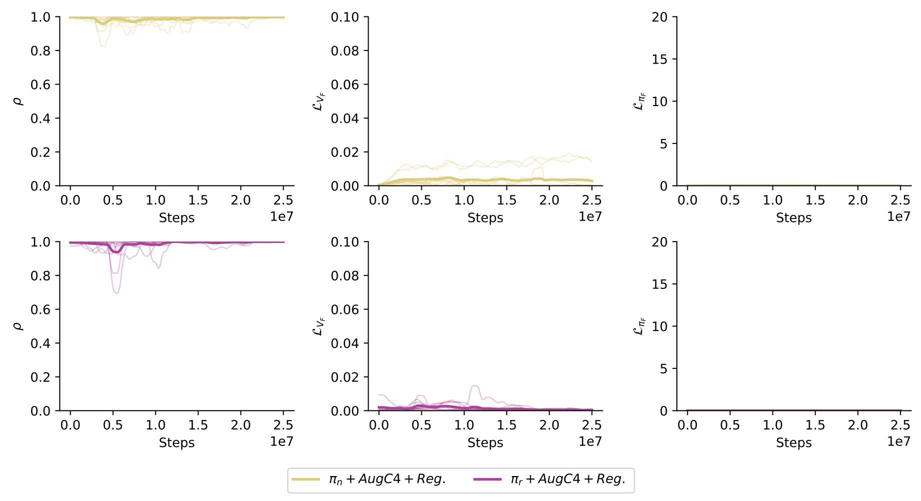


Figure 6.6: When the regularization methods from section 6.3.1 are applied as well as data augmentation, the policies no longer diverge, and the ρ values stay close to 1. This results in significantly higher-performing policies. Similarly to the results in figure 6.5, the results of the 10 different seeds are shown.

and $\pi(a, s)$ is initially low, but becomes much larger as training progresses. Due to this divergence in $\mathcal{L}_{\pi_r}$, the assumption in IMPALA that the behaviour and target policies should be approximately equal deteriorates, leading to the ρ values falling. Note also that ρ is clipped in IMPALA, if the ratio is calculated without clipping then arbitrarily large values would also be present.

In the **Behavior Replay** the IMPALA assumption is maintained by the fact that the behaviour replay policy π_r and the target policy are calculated from the same augmented trajectory. This is reflected in figure 6.5 where the ρ value for behaviour replay π_r is close to 1 and does not diverge during training like the naive policy π_n. This however, does not stop the assumption for augmented policies in equation 6.3 from deteriorating. In fact, in **behaviour replay** augmentation policies, this deterioration is significantly worse than in **naive** augmentation.

6.6.4 With Regularization

When the regularization methods from section 6.3.1 are applied to both **naive** and **behavior replay** methods, the performance of these policies dramatically increases. The difference in performance for these methods is shown in figure 6.4. We also show, in figure 6.6, that the constraints for ρ, $\mathcal{L}_{\pi_r}$ and $\mathcal{L}_{V_F}$ are controlled.

6.7 Conclusions

Data augmentation in reinforcement learning is known to be problematic and, in many cases, is avoided [174, 181].

In this chapter, we have provided a theoretical framework for equivariant data augmentation in deep reinforcement learning. Using this framework, we have provided a description of assumptions that have to be fulfilled by the policy and training regime to avoid poor performance. In addition, when these assumptions are broken, this can lead to algorithmic failures in assumptions in training algorithms such as IMPALA and PPO. More specifically, we conduct experiments using IMPALA and show that naively applying data augmentations to off-policy trajectories results in poor performance. We introduce two methods of data augmentation that can be used in IMPALA. The first **Naive** method: trajectories are augmented at training, which re-uses importance sampling weights from non-augmented data. The second **Behaviour Replay** method: trajectories are augmented at the point of data collection, and importance sampling weights are re-calculated using the augmented trajectories. We show that these methods apply to the V-Trace algorithm, which uses importance sampling to filter off-policy corrections. These regularization methods are inspired by consistency regularization and contrastive learning methods and are applicable in many cases where training is performed in environments with natural symmetries.

In both of these cases, the performance of these methods is dependent on the assumptions holding. However, we measure these assumptions during training and see that it is common for these assumptions to break down and diverge.

To solve this issue, we introduce two regularization terms that can be used in both equivariant and invariant data augmentation methods. These augmentation terms have been previously used in [225] for PPO and are related closely to contrastive losses [269] and consistency regularization [19, 237, 264], however, we now provide theoretical justification and empirical evidence to show why these regularization terms are required.

Finally, we compare the data augmentation and regularization methods to equivariant neural networks, in which the assumptions required for equivariant data augmentations are enforced by the structure of the network. We show that equivariant neural networks outperform data augmentation with no regularization terms and no additional augmented data.

Although we took inspiration for the regularization terms in this work from [225], we did not show empirical evidence from the importance sampling ratio $\frac{\pi_\theta(a|F_s(s))}{\pi_{\theta_{old}}(a|s)}$ that is deemed to be the source of problems for augmentation using PPO. However we believe it is likely that the theoretical assumptions in equation

6.3 and 6.7 are the source of error in PPO also.

Chapter 7

Conclusions and Future Work

In this thesis, we have explored several different areas around how reinforcement learning agents can interact with environments. We have explored the many different observation spaces that can be used and have created baseline experiments and data to allow future results to be tested.

In chapter 6 we investigate several forms of geometric data augmentation that can be applied to environment observations in order to increase learning efficiency. During this investigation of data augmentations, we discovered violations of assumptions in several algorithms that use off-policy corrections. We provide regularization functions that can correct these violations. These regularization functions allow data augmentation to be used in several off-policy correction algorithms.

We have also investigated the many ways in which agents can interact with environments in the form of action spaces. In Section 4.2, We show that many different formulations of action spaces can be converted to a canonical form which also contains additional information about which actions are possible at any given time and requires smaller numbers of logits output layers of policy networks. We show that this action space formulation, "Conditional Action Trees," has no degradation of performance to underlying policies in a simple grid-world setting.

In chapter 5, we introduce the Neural Game Engine, a neural network-based architecture for grid-world games. We show that these kinds of architectures can accurately reproduce game states for unlimited time steps. These architectures can be trained with relatively small numbers of environment transitions and then can be used to train RL environments.

We present Griddly and GriddlyJS in chapter 3 as a tool suite for researchers to investigate many areas of reinforcement learning research in principled and controlled environments.

Griddly is not just a set of grid worlds but a platform to build complex environments without the need to optimize rendering and game logic. Unlike other game engines, Griddly is designed bottom-up with an AI-first philosophy. This places Griddly in a unique position for many areas of research, but currently focused on reinforcement learning.

Griddly's configurability allows many components of environments to be flexible to focus on different areas of the reinforcement learning research space.

This flexibility allows observation spaces and action spaces to be tailored to a large array of problem domains that we have covered in this thesis. For example, in chapter 4, we show that principled thinking of how discrete action spaces can be constructed leads to a compact and general representation of neural network interfaces. Also, in 6, we show the benefit of having pixel-based representations for learning models of environments.

In section 3.11, we show that having a simple description language (GDY) allows expansions of features that can be useful to future research. With GDY, we can create an array of different environments separately from the rendering and optimization of the environment mechanics. This allows researchers to concentrate on building scientific challenges for research rather than on engineering a fast simulation platform.

Additionally, GDY enables tooling such as GriddlyJS to be integrated, showing the flexibility of the underlying Griddly engine. GriddlyJS adds several important features to the Griddly ecosystem:

GriddlyJS allows complex environment mechanics to be designed in the browser without having to install or configure any additional software. These mechanics can be tested and debugged in place, lowering the barrier to building sample-efficient environments. It is also simple to gather large datasets of human play-throughs of the designed challenges. These play-throughs can be used in many areas of research, such as generating human-level baselines, AI alignment, and behavioral cloning. Trained RL policies can also be tested against unseen environment layouts or mechanics in order to identify generalization issues.

We believe that the components of GriddlyJS and Griddly can be used in many future projects and lay a strong foundation for future reinforcement learning research, and opens the door to several new avenues of exploration. We will cover these in the next section.

7.1 Future Work

7.1.1 Griddly

Griddly has become quite a large project with several components that can be enhanced to improve flexibility and allow different areas of research to be experimented with. For example, adding more features to GDY to allow more mechanics to be possible, or adding more continuous actions and observations. Aside from these, Griddly's underlying engine can be significantly improved in several ways.

Even though Griddly is already heavily optimized, new approaches could be integrated, such as vectorization of environments and parallelized GPU rendering with Jax [301].

Environment vectorization is already a common pattern used to spread many environment interfaces across many processes. However, this method is usually implemented in Python and uses many more software processes than there are cores in the CPU. This can result in large overheads in thread context switching and remote procedure calls which involve serialization and deserialization of environment data. The method introduced in [301] avoids these pitfalls by moving the parallelization to the environment engine level (i.e in the underlying engine code) and then interfacing each environment with Python directly rather than through multiple processes.

Additionally, as Griddly is rendered using the Vulkan framework [2], native parallelization of GPU rendering across environments such as that used in Brax [92] and ISAAC [189] could be implemented. This would increase the speed at which pixel observations could be rendered and remove the latency of copying environment states from the GPU to the CPU and back for gradient calculations. Again allowing experimentation turnaround with Griddly environments to be much faster.

Griddly has been used in projects which require forward models to be cloned, for example, in [200]. However, it is noted in this project that this process can be inefficient. Several improvements have been made recently using caching where appropriate to avoid the calculation of unused properties, however, there are more improvements that could be made within the engine itself. For example, the allocation of state memory could be consolidated into a single location, instead of being fragmented as it is at the moment. Once the memory is consolidated, the copying of environment states would be achieved by simply allocating and copying a single block of memory rather than doing this for every variable required.

Other possible improvements and bugs can be raised by users and are tracked on Griddly's GitHub page `https://github.com/Bam4d/Griddly/issues`

7.1.2 GriddlyJS

Currently, GriddlyJS uses browser *local storage* to keep track of environments that are created by users. This means that the environments that the users have created are only stored on the machine they are using. This means that to share environments or saved trajectories, the user must copy the GDY or trajectory JSON text in order to share it via a zip file or similar.

In order to make sharing of Griddly environments significantly simpler, GriddlyJS would require server-side persistence which allows users to log in anywhere to view the environments, models, and trajectories they have created. With a server-side persistence model and an updated frontend, GriddlyJS could also be a service that is used to share and compare agent performance across multiple different environments that have been created by users. GriddlyJS could store benchmarking information for each environment for uploaded models, and become a user-curated competition platform. For example, users could submit models that then could be automatically run against the environments included in the curated benchmark environments. The performance of the models in these environments could then be stored and presented online.

Another feature that would be useful for users is being able to download a set of generated training and evaluation scripts that are tailored to the environments that have been created in the online tool. These scripts and the environment description could also be exported to platforms such as Google Colab [40] or Huggingface [1] to train online. This would allow simple baselines for further experimentation to be generated with just a few clicks.

7.1.3 Conditional Action Trees

There are several comparisons of the Conditional Action Trees paper that were not undertaken but could shed significantly more light on the limitations of CATs.

Firstly the environments in which CATs were tested only allowed single actions at each time-step and arguably do not have MDPs where the action components were interdependent. We could design a set of experiments where the optimal policy of the environment would require high correlation in the components of the action space. In this case, we would be able to show whether the CAT action space can still create a policy that can capture these output dependencies. Intuitively, adding an auto-regressive component to these action spaces would allow this dependency on previous action selections to be modelled by the neural network, but this would require empirical evidence to analyse how important this is, or whether the CAT formulation can inherently model these dependencies in lower layers by learning a similar masking function to those imposed in [99].

Secondly, there is the question of comparing the performance and training of a policy using CATs against a flattened policy in an environment that has multiple controllable objects at each time step. For example, comparing a CAT policy against a state-of-the-art policy in a game such as BotBowl [157], μRTS [139] or [177]. However, these environments would need to be modified to produce the valid action trees and their masks, and then handle the actions as several components.

7.1.4 Entity Neural Networks

Entity Neural Networks (ENNs) are mentioned briefly in several sections of this thesis, but we believe they are a very promising direction that requires several experiments and baseline environments in order to show that they are competitive with current methods.

ENNs provide a method of controlling multiple objects in the same policy at the same time as using attention methods to understand the relationships between objects in the environment as a whole. This method would be particularly interesting when comparing it to state-of-the-art methods in games such as μRTS or other RTS-style games. There are few benchmarks dedicated to efficient reinforcement learning in RTS games, or games that control multiple objects. Griddly provides a framework for this to be realised and also supports an efficient Entity Observation Space. Producing a complex and principled environment for this research is made possible using this tooling.

7.1.5 Environment Modelling

Since the publication of the Neural Game Engines [26] paper, many advances in the modelling of environments have been made. The most promising of these improvements either use stochastic state space models with quantized state spaces, such as vector-quantized auto-encoders, or transformer models. Any advance in this area still requires principled benchmarking, for example analysing which environment mechanics can be modelled, and how successful this modelling is.

To the author's knowledge, there are no environments with a set of principled mechanics which can be used as a benchmarking platform. Again in this case, as Griddly uses the GDY language, in which mechanics can be defined with simple instructions, generating a benchmark set of mechanics would be fairly simple.

These mechanics would currently be limited to grid worlds, but these mechanics can be difficult to learn for neural networks. For example, Griddly can create mechanics that have time delays, stochasticity, non-local effects and partial observability. Experiments could be designed to specifically enumerate and analyse the effectiveness of different forward models when learning these mechanics. This would allow further focus to design models that can handle many types of environments and find particular mechanics that are difficult for models to learn.

7.1.6 Geometric Deep Reinforcement Learning

With the regularization functions that this thesis introduces, geometric data augmentations can be extended to algorithms that use off-policy corrections such as those that use importance sampling [86]. In this work, only discrete geometric data augmentations were explored. In many environments, continuous action and state spaces can also be augmented with geometric transformations. In the continuous case, it would not be possible to augment all possible transformations as the number of transformations is an infinite set.

It would be interesting to see if there is a certain finite subset of transformations that can be used to produce the requirements explained in Chapter 6. However, there's a question about the required size of this subset. It might be the case that the subset of transformations required may be too large and thus make training infeasible. In this case, using a neural network with built-in geometric priors [286] would be significantly faster to train.

In addition to the geometric augmentations, similar linear transformations could also be incorporated into training. For example, in pixel-based observation spaces, the RGB values of objects could be rotated to augment the representation of objects in environments, making the training learn to be invariant to the transposition of the colours. It would be an interesting question to see if just rotating the colours of the pixel can lead to significant gains in performance under other, different transformations, such as lighting changes, or noise.

Bibliography

[1] Hugging Face – The AI community building the future. URL `https://huggingface.co/`.

[2] Vulkan. `https://www.khronos.org/vulkan/`, 2020.

[3] Google AI. Google AI blog: Inceptionism: Going deeper into neural networks, 2020. URL `https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html`.

[4] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. *arXiv*, oct 2019. URL `https://arxiv.org/abs/1910.07113`.

[5] Prithviraj Ammanabrolu and Matthew Hausknecht. Graph Constrained Reinforcement Learning for Natural Language Action Spaces. 2020. URL `https://openreview.net/forum?id=B1x6w0EtwH`.

[6] Brandon Amos, Laurent Dinh, Serkan Cabi, Thomas Rothörl, Sergio Gómez Colmenarejo, Alistair Muldal, Tom Erez, Yuval Tassa, Nando de Freitas, and Misha Denil. Learning awareness models. *arXiv*, apr 2018. URL `https://arxiv.org/abs/1804.06318`.

[7] Brandon Amos, Samuel Stanton, Denis Yarats, and Andrew Gordon Wilson. On the model-based stochastic value gradient for continuous reinforcement learning. *arXiv*, aug 2020. URL `https://arxiv.org/abs/2008.12775`.

[8] Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R. Devon Hjelm. Unsupervised State Representation Learning in Atari. URL `http://arxiv.org/abs/1906.08226`.

[9] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 166–175. PMLR, 06–11 Aug 2017. URL `https://proceedings.mlr.press/v70/andreas17a.html`.

[10] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv*, jul 2017. URL `https://arxiv.org/abs/1707.01495`.

[11] Rika Antonova, Maksim Maydanskiy, Danica Kragic, Sam Devlin, and Katja Hofmann. Analytic manifold learning: Unifying and evaluating representations for continuous control. *arXiv*, jun 2020. URL `https://arxiv.org/abs/2006.08718`.

[12] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. *arXiv*, jan 2017. URL `https://arxiv.org/abs/1701.07875`.

[13] Matan Atzmon, Amos Gropp, and Yaron Lipman. Isometric autoencoders. *arXiv*, jun 2020. URL `https://arxiv.org/abs/2006.09289`.

[14] Yusuf Aytar, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas. Playing hard exploration games by watching YouTube. *arXiv*, may 2018. URL `https://arxiv.org/abs/1805.11592`.

[15] Abdus Salam Azad, Edward Kim, Qiancheng Wu, Kimin Lee, Ion Stoica, Pieter Abbeel, and Sanjit A. Seshia. Scenic4RL: Programmatic Modeling and Generation of Reinforcement Learning Environments. URL `http://arxiv.org/abs/2106.10365`.

[16] Mohammad Gheshlaghi Azar, Bilal Piot, Bernardo Avila Pires, Jean-Bastien Grill, Florent Altché, and Rémi Munos. World discovery models. *arXiv*, feb 2019. URL `https://arxiv.org/abs/1902.07685`.

[17] Amin Babadi, Yi Zhao, Juho Kannala, Alexander Ilin, and Joni Pajarinen. Continuous Monte Carlo Graph Search. URL `http://arxiv.org/abs/2210.01426`.

[18] Mohammad Babaeizadeh, Chelsea Finn, Dumitru Erhan, Roy H. Campbell, and Sergey Levine. Stochastic variational video prediction. *arXiv*, oct 2017. URL `https://arxiv.org/abs/1710.11252`.

[19] Philip Bachman, Ouais Alsharif, and Doina Precup. Learning with pseudo-ensembles. *arXiv*, dec 2014. URL `https://arxiv.org/abs/1412.4864`.

[20] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. *arXiv*, mar 2020. URL `https://arxiv.org/abs/2003.13350`.

[21] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, and Charles Blundell. Never give up: Learning directed exploration strategies. *arXiv*, feb 2020. URL `https://arxiv.org/abs/2002.06038`.

[22] Hendrik Baier and Peter I. Cowling. Evolutionary MCTS for multi-action adversarial games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, aug 2018. ISBN 978-1-5386-4359-4. doi: 10.1109/{CIG}.2018.8490403.

[23] Bram Bakker. Reinforcement learning with long short-term memory. *Advances in neural information processing systems*, jan 2002. URL `https://www.researchgate.net/publication/{2395590_Reinforcement_Learning_with_Long_Short}-{Term_Memory}`.

[24] Philip Ball, Jack Parker-Holder, Aldo Pacchiano, Krzysztof Choromanski, and Stephen Roberts. Ready policy one: World building through active learning. URL `http://proceedings.mlr.press/v119/ball20a.html`.

[25] Martin Balla, Simon M. Lucas, and Diego Perez-Liebana. Evaluating generalisation in general video game playing. *arXiv*, May 2020. URL `https://arxiv.org/abs/2005.11247`.

[26] Chris Bamford and Simon M. Lucas. Neural Game Engine: Accurate learning of generalizable forward models from pixels. In *2020 IEEE Conference on Games (CoG)*, pages 81–88, 2020. doi: 10.1109/CoG47356.2020.9231688.

[27] Christopher Bamford. Griddly Docs — Griddly 1.4.3 documentation. URL `https://griddly.readthedocs.io/en/latest/index.html`.

[28] Christopher Bamford and Alvaro Ovalle. Generalising Discrete Action Spaces with Conditional Action Trees. In *2021 IEEE Conference on Games (CoG)*, pages 1–8, Copenhagen, Denmark, August 2021. IEEE.

ISBN 978-1-66543-886-5. doi: 10.1109/CoG52621.2021.9619093. URL `https://ieeexplore.ieee.org/document/9619093/`.

[29] Christopher Bamford, Shengyi Huang, and Simon Lucas. Griddly: A platform for AI research in games. In *Workshop on Reinforcement Learning in Games*, . URL `http://aaai-rlg.mlanctot.info/2021/papers/{AAAI21}-{RLG_paper_34}.pdf`.

[30] Christopher Bamford, Minqi Jiang, Mikayel Samvelyan, and Tim Rocktäschel. GriddlyJS: A Web IDE for Reinforcement Learning. . URL `https://openreview.net/forum?id=YmacJv0i_UR&referrer=%5BAuthor%20Console%5D(%2Fgroup%3Fid%3DNeurIPS.cc%2F2022%2FTrack%2FDatasets_and_Benchmarks%2FAuthors%23your-submissions)`.

[31] André Barreto, Shaobo Hou, Diana Borsa, David Silver, and Doina Precup. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences of the United States of America*, aug 2020. doi: 10.1073/pnas.1907370117. URL `http://dx.doi.org/10.1073/pnas.1907370117`.

[32] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *arXiv*, dec 2016. URL `https://arxiv.org/abs/1612.00222`.

[33] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, jun 2018. URL `https://arxiv.org/abs/1806.01261`.

[34] Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. SE(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *arXiv*, jan 2021. URL `https://arxiv.org/abs/2101.03164`.

[35] Charles Beattie, Thomas Köppe, Edgar A. Duéñez-Guzmán, and Joel Z. Leibo. DeepMind Lab2D. URL `http://arxiv.org/abs/2011.07027`.

[36] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. DeepMind lab. *arXiv*, dec 2016. URL `https://arxiv.org/abs/1612.03801`.

[37] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. 47: 253–279. ISSN 1076-9757. doi: 10.1613/jair.3912. URL `https://www.jair.org/index.php/jair/article/view/10819`.

[38] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *arXiv*, jun 2016. URL `https://arxiv.org/abs/1606.01868`.

[39] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: a review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, aug 2013. doi: 10.1109/{TPAMI}.2013.50. URL `http://dx.doi.org/10.1109/{TPAMI}.2013.50`.

[40] Ekaba Bisong. Google Colaboratory. In Ekaba Bisong, editor, *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, pages 59–64. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4470-8. doi: 10.1007/978-1-4842-4470-8_7. URL `https://doi.org/10.1007/978-1-4842-4470-8_7`.

[41] Diana Borsa, André Barreto, John Quan, Daniel Mankowitz, Rémi Munos, Hado van Hasselt, David Silver, and Tom Schaul. Universal successor features approximators. *arXiv*, dec 2018. URL `https://arxiv.org/abs/1812.07626`.

[42] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. *arXiv*, sep 2018. URL `https://arxiv.org/abs/1809.11096`.

[43] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *arXiv*, jun 2016. URL `https://arxiv.org/abs/1606.01540`.

[44] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[45] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[46] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv*, apr 2021. URL `https://arxiv.org/abs/2104.13478`.

[47] Lars Buesing, Theophane Weber, Sebastien Racaniere, S. M. Ali Eslami, Danilo Rezende, David P. Reichert, Fabio Viola, Frederic Besse, Karol Gregor, Demis Hassabis, and Daan Wierstra. Learning and querying fast generative models for reinforcement learning. *arXiv*, feb 2018. URL `https://arxiv.org/abs/1802.03006`.

[48] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv*, oct 2018. URL `https://arxiv.org/abs/1810.12894`.

[49] Christopher P. Burgess, Irina Higgins, Arka Pal, Loic Matthey, Nick Watters, Guillaume Desjardins, and Alexander Lerchner. Understanding disentangling in β-VAE. *arXiv*, apr 2018. URL `https://arxiv.org/abs/1804.03599`.

[50] Christopher P. Burgess, Loic Matthey, Nicholas Watters, Rishabh Kabra, Irina Higgins, Matt Botvinick, and Alexander Lerchner. MONet: Unsupervised scene decomposition and representation. *arXiv*, jan 2019. URL `https://arxiv.org/abs/1901.11390`.

[51] Chang Chen, Yi-Fu Wu, Jaesik Yoon, and Sungjin Ahn. TransDreamer: Reinforcement Learning with Transformer World Models. URL `http://arxiv.org/abs/2202.09481`.

[52] Shuxiao Chen, Edgar Dobriban, and Jane H Lee. A group-theoretic framework for data augmentation. *arXiv*, jul 2019. URL `https://arxiv.org/abs/1907.10905`.

[53] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv*, feb 2020. URL `https://arxiv.org/abs/2002.05709`.

[54] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018.

[55] Silvia Chiappa, Sébastien Racaniere, Daan Wierstra, and Shakir Mohamed. Recurrent environment simulators. *arXiv*, apr 2017. URL `https://arxiv.org/abs/1704.02254`.

[56] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Stroudsburg, PA, USA, 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-4012. URL `http://aclweb.org/anthology/W14-4012`.

[57] Sungho Choi, Seungyul Han, Woojun Kim, and Youngchul Sung. Cross-domain imitation learning with a dual structure. *arXiv*, jun 2020. URL `https://arxiv.org/abs/2006.01494`.

[58] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv*, dec 2014. URL `https://arxiv.org/abs/1412.3555`.

[59] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649. IEEE, jun 2012. doi: 10.1109/{CVPR}.2012.6248110. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6248110`.

[60] Ignasi Clavera, Violet Fu, and Pieter Abbeel. Model-augmented actor-critic: Backpropagating through paths. *arXiv*, may 2020. URL `https://arxiv.org/abs/2005.08068`.

[61] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv*, dec 2019. URL `https://arxiv.org/abs/1912.01588`.

[62] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv*, December 2019.

[63] Taco S. Cohen and Max Welling. Steerable CNNs. *arXiv*, dec 2016. URL `https://arxiv.org/abs/1612.08498`.

[64] Taco S. Cohen and Max Welling. Group equivariant convolutional networks. *arXiv*, feb 2016. URL `https://arxiv.org/abs/1602.07576`.

[65] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, H. H. L. M. Donkers, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum, editors,

Computers and Games, volume 4630 of *Lecture notes in computer science*, pages 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-75537-1. doi: 10.1007/978-3-540-75538-8_7. URL `http://link.springer.com/10.1007/978-3-540-75538-8_7`.

[66] Eric Crawford and Joelle Pineau. Spatially invariant unsupervised object detection with convolutional neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:3412–3420, jul 2019. ISSN 2374-3468. doi: 10.1609/aaai.v33i01.33013412. URL `https://aaai.org/ojs/index.php/{AAAI}/article/view/4216`.

[67] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. AutoAugment: Learning augmentation policies from data. *arXiv*, may 2018. URL `https://arxiv.org/abs/1805.09501`.

[68] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. RandAugment: Practical automated data augmentation with a reduced search space. *Advances in Neural Information Processing Systems*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/d85b63ef0ccb114d0a3bb7b7d808028f-Abstract.html`.

[69] Wojciech Marian Czarnecki, Razvan Pascanu, Simon Osindero, Siddhant M. Jayakumar, Grzegorz Swirszcz, and Max Jaderberg. Distilling policy distillation. *arXiv*, feb 2019. URL `https://arxiv.org/abs/1902.02186`.

[70] Marc-Alexandre Côté, Ákos Kádár, Xingdi (Eric) Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. TextWorld: A Learning Environment for Text-based Games. pages 1–29, June 2018. URL `https://www.microsoft.com/en-us/research/publication/textworld-a-learning-environment-for-text-based-games/`.

[71] Will Dabney, Georg Ostrovski, and André Barreto. Temporally-extended e-greedy exploration. *arXiv*, jun 2020. URL `https://arxiv.org/abs/2006.01782`.

[72] Steven Dalton, Iuri Frosio, and Michael Garland. Accelerating Reinforcement Learning through GPU Atari Emulation, October 2020. URL `http://arxiv.org/abs/1907.08467`. arXiv:1907.08467 [cs, stat].

[73] T. Degris, P. M. Pilarski, and R. S. Sutton. Model-free reinforcement learning with continuous action in practice. In *2012 American Control*

Conference (ACC), pages 2177–2182. IEEE, jun 2012. ISBN 978-1-4577-1096-4. doi: 10.1109/{ACC}.2012.6315022. URL `http://ieeexplore.ieee.org/document/6315022/`.

[74] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design, 2021. URL `http://arxiv.org/abs/2012.02096`.

[75] Alexander Dockhorn and Daan Apeldoorn. Forward model approximation for general video game learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, aug 2018. ISBN 978-1-5386-4359-4. doi: 10.1109/{CIG}.2018.8490411. URL `https://ieeexplore.ieee.org/document/8490411/`.

[76] Alexander Dockhorn, Simon M. Lucas, Vanessa Volz, Ivan Bravi, Raluca D. Gaina, and Diego Perez-Liebana. Learning local forward models on unforgiving games. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, aug 2019. ISBN 978-1-7281-1884-0. doi: 10.1109/{CIG}.2019.8848044. URL `https://ieeexplore.ieee.org/document/8848044/`.

[77] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. URL `http://arxiv.org/abs/2010.11929`.

[78] Olve Drageset, Mark H. M. Winands, Raluca D. Gaina, and Diego Perez-Liebana. Optimising level generators for general video game AI. In *2019 IEEE conference on games (CoG)*, pages 1–8. IEEE, August 2019. ISBN 978-1-72811-884-0. doi: 10.1109/{CIG}.2019.8847961. URL `https://ieeexplore.ieee.org/document/8847961/`.

[79] Yilun Du and Karthik Narasimhan. Task-agnostic dynamics priors for deep reinforcement learning. *arXiv*, may 2019. URL `https://arxiv.org/abs/1905.04819`.

[80] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. URL `http://arxiv.org/abs/1604.06778`.

[81] Sam Earle, Justin Snider, Matthew C. Fontaine, Stefanos Nikolaidis, and Julian Togelius. Illuminating Diverse Neural Cellular Automata for Level Generation. URL `http://arxiv.org/abs/2109.05489`.

[82] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv*, jan 2019. URL `https://arxiv.org/abs/1901.10995`.

[83] Epic Games. Unreal. `https://www.unrealengine.com/en-US`. [Accessed: June 20, 2023].

[84] S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, infer, repeat: Fast scene understanding with generative models. *arXiv*, mar 2016. URL `https://arxiv.org/abs/1603.08575`.

[85] S M Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S Morcos, Marta Garnelo, Avraham Ruderman, Andrei A Rusu, Ivo Danihelka, Karol Gregor, David P Reichert, Lars Buesing, Theophane Weber, Oriol Vinyals, Dan Rosenbaum, Neil Rabinowitz, Helen King, Chloe Hillier, Matt Botvinick, Daan Wierstra, Koray Kavukcuoglu, and Demis Hassabis. Neural scene representation and rendering. *Science*, 360(6394): 1204–1210, jun 2018. ISSN 0036-8075. doi: 10.1126/science.aar6170. URL `http://www.sciencemag.org/lookup/doi/10.1126/science.aar6170`.

[86] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. *arXiv*, feb 2018. URL `https://arxiv.org/abs/1802.01561`.

[87] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: Scalable and efficient deep-RL with accelerated central inference. *arXiv*, oct 2019. URL `https://arxiv.org/abs/1910.06591`.

[88] Pietro Falco, Abdallah Attawia, Matteo Saveriano, and Dongheui Lee. On policy learning robust to irreversible events: An application to robotic in-hand manipulation. *IEEE robotics and automation letters*, 3(3):1482–1489, jul 2018. ISSN 2377-3766. doi: 10.1109/{LRA}.2018.2800110. URL `http://ieeexplore.ieee.org/document/8276248/`.

[89] Zhou Fan, Rui Su, Weinan Zhang, and Yong Yu. Hybrid actor-critic reinforcement learning in parameterized action space. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 2279–2285, California, aug 2019. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1. doi: 10.24963/ijcai.2019/316.

[90] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv*, feb 2018. URL `https://arxiv.org/abs/1803.00101`.

[91] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *arXiv*, may 2016. URL `https://arxiv.org/abs/1605.07157`.

[92] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax – A Differentiable Physics Engine for Large Scale Rigid Body Simulation, June 2021. URL `http://arxiv.org/abs/2106.13281`. arXiv:2106.13281 [cs].

[93] Karlis Freivalds and Renars Liepins. Improving the neural GPU architecture for algorithm learning. *arXiv*, feb 2017. URL `https://arxiv.org/abs/1702.08727`.

[94] Raluca D. Gaina, Jialin Liu, Simon M. Lucas, and Diego Pérez-Liébana. Analysis of vanilla rolling horizon evolution parameters in general video game playing. In Giovanni Squillero and Kevin Sim, editors, *Applications of evolutionary computation*, volume 10199 of *Lecture notes in computer science*, pages 418–434. Springer International Publishing, Cham, 2017. ISBN 978-3-319-55848-6. doi: 10.1007/978-3-319-55849-3_28. URL `https://link.springer.com/book/10.1007/978-3-319-55849-3`.

[95] Raluca D. Gaina, Simon M. Lucas, and Diego Perez-Liebana. Rolling horizon evolution enhancements in general video game playing. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 88–95. IEEE, aug 2017. ISBN 978-1-5386-3233-8. doi: 10.1109/{CIG}.2017.8080420. URL `http://ieeexplore.ieee.org/document/8080420/`.

[96] Xiang Gao. Deep reinforcement learning for time series: playing idealized trading games. *arXiv*, mar 2018. URL `https://arxiv.org/abs/1803.03916`.

[97] Leon Gatys, Alexander Ecker, and Matthias Bethge. A neural algorithm of artistic style. *Journal of Vision*, 16(12):326, sep 2016. ISSN 1534-7362. doi: 10.1167/16.12.326. URL `http://jov.arvojournals.org/article.aspx?doi=10.1167/16.12.326`.

[98] Carles Gelada, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G. Bellemare. DeepMDP: Learning continuous latent space models

for representation learning. *arXiv*, jun 2019. URL `https://arxiv.org/abs/1906.02736`.

[99] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked Autoencoder for Distribution Estimation. URL `http://arxiv.org/abs/1502.03509`.

[100] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Continual prediction using LSTM with forget gates. In Maria Marinaro, Roberto Tagliaferri, and J. G. Taylor, editors, *Neural Nets WIRN Vietri-99*, Perspectives in neural computing, pages 133–138. Springer London, London, 1999. ISBN 978-1-4471-0877-1. doi: 10.1007/978-1-4471-0877-1_10. URL `http://link.springer.com/10.1007/978-1-4471-0877-1_10`.

[101] William Gilpin. Cellular automata as convolutional neural networks. *arXiv*, sep 2018. URL `https://arxiv.org/abs/1809.02942`.

[102] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv*, jun 2014. URL `https://arxiv.org/abs/1406.2661`.

[103] Klaus Greff, Raphaël Lopez Kaufman, Rishabh Kabra, Nick Watters, Chris Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. *arXiv*, mar 2019. URL `https://arxiv.org/abs/1903.00450`.

[104] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. DRAW: A recurrent neural network for image generation. *arXiv*, feb 2015. URL `https://arxiv.org/abs/1502.04623`.

[105] Karol Gregor, George Papamakarios, Frederic Besse, Lars Buesing, and Theophane Weber. Temporal difference variational auto-encoder. *arXiv*, jun 2018. URL `https://arxiv.org/abs/1806.03107`.

[106] Karol Gregor, Danilo Jimenez Rezende, Frederic Besse, Yan Wu, Hamza Merzic, and Aaron van den Oord. Shaping belief states with generative environment models for RL. *arXiv*, jun 2019. URL `https://arxiv.org/abs/1906.09237v2`.

[107] Niko A. Grupen, Bart Selman, and Daniel D. Lee. Cooperative multi-agent fairness and equivariant policies. *arXiv*, jun 2021. URL `https://arxiv.org/abs/2106.05727`.

[108] Marek Grzes. Reward shaping in episodic reinforcement learning. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, page 565–573, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[109] Christian Guckelsberger and Christoph Salge. Does empowerment maximisation allow for enactive artificial agents? In *Proceedings of the Artificial Life Conference 2016*, pages 704–711, Cambridge, MA, jul 2016. MIT Press. ISBN 978-0-262-33936-0. doi: 10.7551/978-0-262-33936-0-ch112. URL `https://mitpress.mit.edu/sites/default/files/titles/content/alife16/978-0-262-33936-0-ch112.pdf`.

[110] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy Lillicrap. An investigation of model-free planning. *arXiv*, jan 2019. URL `https://arxiv.org/abs/1901.03559`.

[111] Maxim Gumin. Wave Function Collapse Algorithm. URL `https://github.com/mxgmn/WaveFunctionCollapse`.

[112] Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Bernardo A. Pires, and Rémi Munos. Neural predictive belief representations. *arXiv*, nov 2018. URL `https://arxiv.org/abs/1811.06407`.

[113] William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. MineRL: A Large-Scale Dataset of Minecraft Demonstrations, July 2019. URL `http://arxiv.org/abs/1907.13440`. arXiv:1907.13440 [cs, stat].

[114] Matthew Guzdial, Boyang Li, and Mark O. Riedl. Game engine learning from video. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3707–3713, California, aug 2017. International Joint Conferences on Artificial Intelligence Organization. ISBN 9780999241103. doi: 10.24963/ijcai.2017/518. URL `https://www.ijcai.org/proceedings/2017/518`.

[115] David Ha and Jürgen Schmidhuber. World models. *arXiv*, mar 2018. URL `https://arxiv.org/abs/1803.10122`.

[116] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv*, jan 2018. URL `https://arxiv.org/abs/1801.01290`.

[117] Danijar Hafner. Benchmarking the Spectrum of Agent Capabilities. URL `https://openreview.net/forum?id=1W0z96MFEoH`.

[118] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *arXiv*, nov 2018. URL `https://arxiv.org/abs/1811.04551`.

[119] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv*, dec 2019. URL `https://arxiv.org/abs/1912.01603`.

[120] Danijar Hafner, Pedro A. Ortega, Jimmy Ba, Thomas Parr, Karl Friston, and Nicolas Heess. Action and perception as divergence minimization. *arXiv*, sep 2020. URL `https://arxiv.org/abs/2009.01791`.

[121] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering Atari with Discrete World Models, February 2022. URL `http://arxiv.org/abs/2010.02193`. arXiv:2010.02193 [cs, stat].

[122] Eric Hambro, Sharada Mohanty, Dmitrii Babaev, Minwoo Byeon, Dipam Chakraborty, Edward Grefenstette, Minqi Jiang, Daejin Jo, Anssi Kanervisto, Jongmin Kim, et al. Insights from the neurips 2021 nethack challenge. *arXiv preprint arXiv:2203.11889*, 2022.

[123] Jessica B. Hamrick, Abram L. Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Buesing, Petar Veličković, and Théophane Weber. On the role of planning in model-based deep reinforcement learning, March 2021. URL `http://arxiv.org/abs/2011.04021`. arXiv:2011.04021 [cs].

[124] Hado Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL `https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html`.

[125] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable MDPs. *arXiv*, jul 2015. URL `https://arxiv.org/abs/1507.06527`.

[126] Jean-Baptiste Hervé and Christoph Salge. Automated Isovist Computation for Minecraft. URL `http://arxiv.org/abs/2204.03752`.

[127] I. Higgins, Loïc Matthey, A. Pal, C. Burgess, Xavier Glorot, M. Botvinick, S. Mohamed, and Alexander Lerchner. beta-VAE:

Learning basic visual concepts with a constrained variational framework. 2017. URL https://www.semanticscholar.org/paper/beta-VAE-Learning-Basic-Visual-Concepts-with-a-Higgins-Matthey/a90226c41b79f8b06007609f39f82757073641e2.

[128] Irina Higgins, David Amos, David Pfau, Sebastien Racaniere, Loic Matthey, Danilo Rezende, and Alexander Lerchner. Towards a definition of disentangled representations. *arXiv*, dec 2018. URL https://arxiv.org/abs/1812.02230.

[129] Felix Hill, Andrew Lampinen, Rosalia Schneider, Stephen Clark, Matthew Botvinick, James L. McClelland, and Adam Santoro. Environmental drivers of systematicity and generalization in a situated agent. *arXiv*, oct 2019. URL https://arxiv.org/abs/1910.00571.

[130] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, jul 2006. doi: 10.1126/science.1127647. URL http://dx.doi.org/10.1126/science.1127647.

[131] R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler, and Yoshua Bengio. Learning deep representations by mutual information estimation and maximization. *arXiv*, aug 2018. URL https://arxiv.org/abs/1808.06670.

[132] S Hochreiter and J Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[133] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv*, mar 2018. URL https://arxiv.org/abs/1803.00933.

[134] Yedid Hoshen. Vain: Attentional multi-agent predictive modeling. *arXiv*, jun 2017. URL https://arxiv.org/abs/1706.06122.

[135] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. VIME: Variational information maximizing exploration. *arXiv*, may 2016. URL https://arxiv.org/abs/1605.09674.

[136] Haozhi Huang, Hao Wang, Wenhan Luo, Lin Ma, Wenhao Jiang, Xiaolong Zhu, Zhifeng Li, and Wei Liu. Real-time neural style transfer for videos. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7044–7052. IEEE, jul 2017. ISBN 978-1-5386-0457-

1. doi: 10.1109/{CVPR}.2017.745. URL `http://ieeexplore.ieee.org/document/8100228/`.

[137] Shengyi Huang and Santiago Ontañón. Comparing observation and action representations for deep reinforcement learning in urts. *arXiv*, oct 2019. URL `https://arxiv.org/abs/1910.12134`.

[138] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *arXiv*, jun 2020.

[139] Shengyi Huang, Santiago Ontañón, Chris Bamford, and Lukasz Grela. Gym-μRTS: Toward Affordable Full Game Real-time Strategy Games Research with Deep Reinforcement Learning. URL `http://arxiv.org/abs/2105.13807`.

[140] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, and Jeff Braga. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. 2021.

[141] Olivier J. Hénaff, Aravind Srinivas, Jeffrey De Fauw, Ali Razavi, Carl Doersch, S. M. Ali Eslami, and Aaron van den Oord. Data-efficient image recognition with contrastive predictive coding. *arXiv*, may 2019. URL `https://arxiv.org/abs/1905.09272`.

[142] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *arXiv*, jun 2015. URL `https://arxiv.org/abs/1506.02025`.

[143] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv*, nov 2016. URL `https://arxiv.org/abs/1611.05397`.

[144] Tomas Jakab, Ankush Gupta, Hakan Bilen, and Andrea Vedaldi. Unsupervised learning of object landmarks through conditional image generation. 2018. URL `https://papers.nips.cc/paper/7657-unsupervised-learning-of-object-landmarks-through-conditional-image-generation`.

[145] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. *arXiv*, dec 2018. URL `https://arxiv.org/abs/1812.07252`.

[146] Peter A. Jansen. A Systematic Survey of Text Worlds as Embodied Natural Language Environments, July 2021. URL `http://arxiv.org/abs/2107.04132`. arXiv:2107.04132 [cs].

[147] Minqi Jiang, Michael D. Dennis, Jack Parker-Holder, Jakob Nicolaus Foerster, Edward Grefenstette, and Tim Rocktäschel. Replay-Guided Adversarial Environment Design. . URL `https://openreview.net/forum?id=5UZ-AcwFDKJ`.

[148] Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized Level Replay, . URL `http://arxiv.org/abs/2010.03934`.

[149] Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. *arXiv*, oct 2020. URL `https://arxiv.org/abs/2010.03934`.

[150] Zhengyao Jiang, Tianjun Zhang, Michael Janner, Yueying Li, Tim Rocktäschel, Edward Grefenstette, and Yuandong Tian. Efficient Planning in a Compact Latent Action Space, . URL `http://arxiv.org/abs/2208.10291`.

[151] Yuan Jin, Lan Du, Longxiang Gao, Yong Xiang, Yunfeng Li, and Ruohua Xu. Variational auto-encoder based bayesian poisson tensor factorization for sparse and imbalanced count data. *arXiv*, oct 2019. URL `https://arxiv.org/abs/1910.05570`.

[152] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. *arXiv preprint arXiv:1902.01378*, 2019.

[153] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents, May 2020. URL `http://arxiv.org/abs/1809.02627`. arXiv:1809.02627 [cs, stat].

[154] Changwook Jun, Hansol Jang, Myoseop Sim, Hyun Kim, Jooyoung Choi, Kyungkoo Min, and Kyunghoon Bae. ANNA: Enhanced Language Representation for Question Answering. URL `http://arxiv.org/abs/2203.14507`.

[155] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *arXiv*, aug 2017. URL `https://arxiv.org/abs/1708.07902`.

[156] Niels Justesen, Tobias Mahlmann, Sebastian Risi, and Julian Togelius. Playing multiaction adversarial games: online evolutionary planning versus tree search. *IEEE Transactions on Games*, 10(3):281–291, sep 2018. ISSN 2475-1502. doi: 10.1109/{TCIAIG}.2017.2738156.

[157] Niels Justesen, Peter David Moore, Lasse Moller Uth, Julian Togelius, Christopher Jakobsen, and Sebastian Risi. Blood bowl: A new board game challenge and competition for ai. In *2019 IEEE Conference on Games (COG)*. IEEE, 2019.

[158] Lukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. *arXiv*, nov 2015. URL `https://arxiv.org/abs/1511.08228`.

[159] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari. *arXiv*, mar 2019. URL `https://arxiv.org/abs/1903.00374`.

[160] Łukasz Kaiser and Samy Bengio. Discrete autoencoders for sequence models. *arXiv*, jan 2018. URL `https://arxiv.org/abs/1801.09797`.

[161] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1): 35, 1960. ISSN 00219223. doi: 10.1115/1.3662552. URL `http://{FluidsEngineering}.asmedigitalcollection.asme.org/article.aspx?articleid=1430402`.

[162] Pentti Kanerva. *Sparse Distributed Memory*. MIT Press, jan 1988. ISBN 0262111322. URL `https://dl.acm.org/citation.cfm?id=534853`.

[163] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. *arXiv*, apr 2020.

[164] Anssi Kanervisto, Stephanie Milani, Karolis Ramanauskas, Nicholay Topin, Zichuan Lin, Junyou Li, Jianing Shi, Deheng Ye, Qiang Fu, Wei Yang, Weijun Hong, Zhongyue Huang, Haicheng Chen, Guangjun Zeng, Yue Lin, Vincent Micheli, Eloi Alonso, François Fleuret, Alexander Nikulin, Yury Belousov, Oleg Svidchenko, and Aleksei Shpilman. MineRL Diamond 2021 Competition: Overview, Results, and Lessons Learned, February 2022. URL `http://arxiv.org/abs/2202.10583`. arXiv:2202.10583 [cs].

[165] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. may 2019. URL https://www.researchgate.net/publication/{340285012_Recurrent_Experience_Replay_in_Distributed_Reinforcement_Learning}.

[166] Maximilian Karl, Maximilian Soelch, Justin Bayer, and Patrick van der Smagt. Deep variational bayes filters: Unsupervised learning of state space models from raw data. *arXiv*, may 2016. URL https://arxiv.org/abs/1605.06432.

[167] T. Anderson Keller and Max Welling. Topographic VAEs learn equivariant capsules. *arXiv*, sep 2021. URL https://arxiv.org/abs/2109.01394.

[168] Ahmed Khalifa, Diego Perez-Liebana, Simon M. Lucas, and Julian Togelius. General video game level generation. In Tobias Friedrich, editor, *Proceedings of the 2016 on genetic and evolutionary computation conference - GECCO '16*, pages 253–259, New York, New York, USA, July 2016. ACM Press. ISBN 978-1-4503-4206-3. doi: 10.1145/2908812.2908920. URL http://dl.acm.org/citation.cfm?doid=2908812.2908920.

[169] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. PCGRL: Procedural Content Generation via Reinforcement Learning, August 2020. URL http://arxiv.org/abs/2001.09212. arXiv:2001.09212 [cs, stat].

[170] Kuno Kim, Yihong Gu, Jiaming Song, Shengjia Zhao, and Stefano Ermon. Domain adaptive imitation learning. *arXiv*, sep 2019. URL https://arxiv.org/abs/1910.00105.

[171] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv*, dec 2013. URL https://arxiv.org/abs/1312.6114.

[172] David Klindt, Lukas Schott, Yash Sharma, Ivan Ustyuzhaninov, Wieland Brendel, Matthias Bethge, and Dylan Paiton. Towards nonlinear disentanglement in natural data with temporal sparse coding. *arXiv*, jul 2020. URL https://arxiv.org/abs/2007.10930.

[173] Varun Raj Kompella, Matthew Luciw, and Juergen Schmidhuber. Incremental slow feature analysis: Adaptive and episodic learning from high-dimensional input streams. *arXiv*, dec 2011. URL https://arxiv.org/abs/1112.2113.

[174] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv*, apr 2020. URL `https://arxiv.org/abs/2004.13649`.

[175] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[176] Tejas Kulkarni, Ankush Gupta, Catalin Ionescu, Sebastian Borgeaud, Malcolm Reynolds, Andrew Zisserman, and Volodymyr Mnih. Unsupervised learning of object keypoints for perception and control. *arXiv*, jun 2019. URL `https://arxiv.org/abs/1906.11883`.

[177] Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack Learning Environment. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[178] W. H. Kwon, A. M. Bruckstein, and T. Kailath. Stabilizing state-feedback design via the moving horizon method. *International journal of control*, 37(3):631–643, mar 1983. ISSN 0020-7179. doi: 10.1080/00207178308932998. URL `https://www.tandfonline.com/doi/full/10.1080/00207178308932998`.

[179] Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack learning environment. *arXiv*, June 2020. URL `https://arxiv.org/abs/2006.13760`.

[180] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. *arXiv*, sep 2016. URL `https://arxiv.org/abs/1609.05521`.

[181] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. In *Advances in Neural Information Processing Systems 33*. 2020.

[182] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of*

the 35th International Conference on Machine Learning, pages 3053–3062. PMLR. URL https://proceedings.mlr.press/v80/liang18b.html.

[183] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv*, sep 2015. URL https://arxiv.org/abs/1509.02971.

[184] Evan Zheran Liu, Aditi Raghunathan, Percy Liang, and Chelsea Finn. Explore then execute: Adapting without rewards via factorized meta-reinforcement learning. *arXiv*, aug 2020. URL https://arxiv.org/abs/2008.02790.

[185] Simon M. Lucas, Alexander Dockhorn, Vanessa Volz, Chris Bamford, Raluca D. Gaina, Ivan Bravi, Diego Perez-Liebana, Sanaz Mostaghim, and Rudolf Kruse. A local approach to forward model learning: Results on the game of life game. *arXiv*, mar 2019. URL https://arxiv.org/abs/1903.12508.

[186] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *arXiv*, nov 2019.

[187] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, mar 2018. ISSN 1076-9757. doi: 10.1613/jair.5699. URL https://jair.org/index.php/jair/article/view/11182.

[188] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning, August 2021. URL http://arxiv.org/abs/2108.10470. arXiv:2108.10470 [cs].

[189] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning, August 2021. URL http://arxiv.org/abs/2108.10470. arXiv:2108.10470 [cs].

[190] Joseph Marino, Yisong Yue, and Stephan Mandt. Iterative amortized inference. *arXiv*, jul 2018. URL https://arxiv.org/abs/1807.09356.

[191] Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 1934–1940. AAAI Press, 2016.

[192] Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are Sample Efficient World Models. URL `http://arxiv.org/abs/2209.00588`.

[193] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *arXiv*, jan 2020. URL `https://arxiv.org/abs/2001.05566`.

[194] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[195] Parag K. Mital. Time domain neural audio style transfer. *arXiv*, nov 2017. URL `https://arxiv.org/abs/1711.11160`.

[196] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, feb 2015. doi: 10.1038/nature14236. URL `http://dx.doi.org/10.1038/nature14236`.

[197] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[198] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv*, feb 2016. URL `https://arxiv.org/abs/1602.01783`.

[199] Arnab Kumar Mondal, Pratheeksha Nair, and Kaleem Siddiqi. Group equivariant deep reinforcement learning. *arXiv*, jul 2020. URL `https://arxiv.org/abs/2007.03437`.

[200] Antti Mäkipää. A Quantitative Study of Interactions Between Coupled Empowerment Maximising and General Game Playing Agents.

July 2022. URL https://aaltodoc.aalto.fi:443/handle/123456789/116219. Accepted: 2022-08-28T17:00:10Z.

[201] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *arXiv*, jul 2015. URL https://arxiv.org/abs/1507.04296.

[202] Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. *arXiv*, jul 2020. URL https://arxiv.org/abs/2007.02686.

[203] Eric Nalisnick and Padhraic Smyth. Stick-breaking variational autoencoders. *arXiv*, may 2016. URL https://arxiv.org/abs/1605.06197.

[204] Mark J. Nelson. Estimates for the Branching Factors of Atari Games. In *2021 IEEE Conference on Games (CoG)*, pages 1–5. doi: 10.1109/CoG52621.2021.9619137.

[205] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in RL. *arXiv*, apr 2018. URL https://arxiv.org/abs/1804.03720.

[206] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. *arXiv*, jul 2015. URL https://arxiv.org/abs/1507.08750.

[207] Onnx. ONNX | Home, 2022. URL https://onnx.ai/.

[208] Santiago Ontanon. The combinatorial multi-armed bandit problem and its application to real-time strategy games. nov 2013. URL https://www.aaai.org/ocs/index.php/{AIIDE}/{AIIDE13}/paper/{viewPaper}/7377.

[209] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'13, page 58–64. AAAI Press, 2013. ISBN 1577356071.

[210] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv*, jan 2016. URL https://arxiv.org/abs/1601.06759.

[211] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with PixelCNN decoders. *arXiv*, jun 2016. URL https://arxiv.org/abs/1606.05328.

[212] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural Discrete Representation Learning, May 2018. URL http://arxiv.org/abs/1711.00937. arXiv:1711.00937 [cs] version: 2.

[213] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[214] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning Dexterous In-Hand Manipulation. URL http://arxiv.org/abs/1808.00177.

[215] Georg Ostrovski, Marc G. Bellemare, Aaron van den Oord, and Remi Munos. Count-based exploration with neural density models. *arXiv*, mar 2017. URL https://arxiv.org/abs/1703.01310.

[216] Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving Curricula with Regret-Based Environment Design. URL http://arxiv.org/abs/2203.01302.

[217] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 488–489. IEEE, jul 2017. ISBN 978-1-5386-0733-6. doi: 10.1109/{CVPRW}.2017.70. URL http://ieeexplore.ieee.org/document/8014804/.

[218] Deepak Pathak, Dhiraj Gandhi, and Abhinav Gupta. Self-supervised exploration via disagreement. *arXiv*, jun 2019. URL https://arxiv.org/abs/1906.04161.

[219] Vihang P. Patil, Markus Hofmarcher, Marius-Constantin Dinu, Matthias Dorfer, Patrick M. Blies, Johannes Brandstetter, Jose A. Arjona-Medina, and Sepp Hochreiter. Align-rudder: Learning from few demonstrations by reward redistribution, 2020. URL https://arxiv.org/abs/2009.14108.

[220] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic. *ACM transactions on graphics*, 37(4):1–14, jul 2018. ISSN

07300301. doi: 10.1145/3197517.3201311. URL http://dl.acm.org/citation.cfm?doid=3197517.3201311.

[221] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D. Gaina, Julian Togelius, and Simon M. Lucas. General video game AI: A multi-track framework for evaluating agents, games and content generation algorithms. URL https://arxiv.org/abs/1802.10363.

[222] Diego Perez Liebana, Jens Dieskau, Martin Hunermund, Sanaz Mostaghim, and Simon Lucas. Open loop search for general video game playing. In Sara Silva, editor, *Proceedings of the 2015 on genetic and evolutionary computation conference - GECCO '15*, pages 337–344, New York, New York, USA, July 2015. ACM Press. ISBN 978-1-4503-3472-3. doi: 10.1145/2739480.2754811. URL http://dl.acm.org/citation.cfm?doid=2739480.2754811.

[223] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv*, nov 2015. URL https://arxiv.org/abs/1511.06434.

[224] Roberta Raileanu and Tim Rocktäschel. RIDE: Rewarding impact-driven exploration for procedurally-generated environments. *arXiv*, feb 2020. URL https://arxiv.org/abs/2002.12292.

[225] Roberta Raileanu, Max Goldstein, Denis Yarats, Ilya Kostrikov, and Rob Fergus. Automatic data augmentation for generalization in deep reinforcement learning. *CoRR*, abs/2006.12862, 2020.

[226] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know What You Don't Know: Unanswerable Questions for SQuAD, . URL http://arxiv.org/abs/1806.03822.

[227] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text, . URL http://arxiv.org/abs/1606.05250.

[228] MarcAurelio Ranzato, Arthur Szlam, Joan Bruna, Michael Mathieu, Ronan Collobert, and Sumit Chopra. Video (language) modeling: a baseline for generative models of natural videos. *arXiv*, dec 2014. URL https://arxiv.org/abs/1412.6604.

[229] B. Ravindran and A. G. Barto. Symmetries and model minimization in markov decision processes. Technical report, USA, 2001.

[230] Balaraman Ravindran and Andrew G Barto. Approximate homomorphisms: A framework for non-exact minimization in markov decision processes. *International Conference on Knowledge Based Computer Systems*, 2004.

[231] Danilo J. Rezende, Ivo Danihelka, George Papamakarios, Nan Rosemary Ke, Ray Jiang, Theophane Weber, Karol Gregor, Hamza Merzic, Fabio Viola, Jane Wang, Jovana Mitrovic, Frederic Besse, Ioannis Antonoglou, and Lars Buesing. Causally correct partial models for reinforcement learning. *arXiv*, feb 2020. URL `https://arxiv.org/abs/2002.02836`.

[232] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *arXiv*, jan 2014. URL `https://arxiv.org/abs/1401.4082`.

[233] Martin Riedmiller. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, Lecture Notes in Computer Science, pages 317–328, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31692-3. doi: 10.1007/11564096_32.

[234] Sebastian Risi and Julian Togelius. Increasing generality in machine learning through procedural content generation. *Nature Machine Intelligence*, 2(8):428–436, 2020.

[235] Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. Artistic style transfer for videos. In Bodo Rosenhahn and Bjoern Andres, editors, *Pattern Recognition*, volume 9796 of *Lecture notes in computer science*, pages 26–36. Springer International Publishing, Cham, 2016. ISBN 978-3-319-45885-4. doi: 10.1007/978-3-319-45886-1_3. URL `http://link.springer.com/10.1007/978-3-319-45886-1_3`.

[236] Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv*, nov 2015. URL `https://arxiv.org/abs/1511.06295`.

[237] Mehdi Sajjadi, Mehran Javanmardi, and Tolga Tasdizen. Regularization with stochastic transformations and perturbations for deep semi-supervised learning. *arXiv*, jun 2016. URL `https://arxiv.org/abs/1606.04586`.

[238] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, jul 2009. ISSN 0888613X. doi: 10.1016/j.ijar.2008.11.006. URL `http://linkinghub.elsevier.com/retrieve/pii/{S0888613X08001813}`.

[239] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. PixelCNN++: Improving the PixelCNN with discretized logistic mixture likelihood and other modifications. *arXiv*, jan 2017. URL `https://arxiv.org/abs/1701.05517`.

[240] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft multi-agent challenge. *arXiv*, feb 2019.

[241] Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL `https://openreview.net/forum?id=skFwlyefkWJ`.

[242] Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel. MiniHack the Planet: A Sandbox for Open-Ended Reinforcement Learning Research. *arXiv:2109.13202 [cs, stat]*, November 2021. URL `http://arxiv.org/abs/2109.13202`. arXiv: 2109.13202.

[243] Tom Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, aug 2013. ISBN 978-1-4673-5311-3. doi: 10.1109/{CIG}.2013.6633610. URL `http://ieeexplore.ieee.org/document/6633610/`.

[244] Tom; Schaul, Dan; Gregor, Karol; Silver, and David. Universal value function approximators. jun 2015. URL `https://dl.acm.org/citation.cfm?id=3045258`.

[245] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv*, nov 2015. URL `https://arxiv.org/abs/1511.05952`.

[246] Juergen Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *arXiv*, nov 2015. URL `https://arxiv.org/abs/1511.09249`.

[247] Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE transactions on autonomous mental development*, 2(3):230–247, sep 2010. ISSN 1943-0604. doi: 10.1109/{TAMD}.2010.2056368. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5508364`.

[248] Julian Schrittwieser, Thomas Hubert, Amol Mandhane, Mohammadamin Barekatain, Ioannis Antonoglou, and David Silver. Online and offline reinforcement learning by planning with a learned model. URL `https://arxiv.org/abs/2104.06294`.

[249] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv*, nov 2019. URL `https://arxiv.org/abs/1911.08265`.

[250] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *arXiv*, feb 2015. URL `https://arxiv.org/abs/1502.05477`.

[251] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv*, jun 2015. URL `https://arxiv.org/abs/1506.02438`.

[252] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, jul 2017. URL `https://arxiv.org/abs/1707.06347`.

[253] Max Schwarzer, Ankesh Anand, Rishab Goel, R Devon Hjelm, Aaron Courville, and Philip Bachman. Data-efficient reinforcement learning with self-predictive representations. *arXiv*, jul 2020. URL `https://arxiv.org/abs/2007.05929`.

[254] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. *arXiv*, may 2020. URL `https://arxiv.org/abs/2005.05960`.

[255] Pierre Sermanet, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, Sergey Levine, and Google Brain. Time-contrastive networks: Self-supervised learning from video. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1134–1141. IEEE, may 2018. ISBN 978-1-5386-3081-5. doi: 10.1109/ICRA.2018.8462891. URL `https://ieeexplore.ieee.org/document/8462891/`.

[256] Evan Shelhamer, Parsa Mahmoudieh, Max Argus, and Trevor Darrell. Loss is its own reward: Self-supervision for reinforcement learning. *arXiv*, dec 2016. URL `https://arxiv.org/abs/1612.07307`.

[257] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. 6(1):60, 2019. ISSN 2196-1115. doi: 10.1186/s40537-019-0197-0. URL `https://doi.org/10.1186/s40537-019-0197-0`.

[258] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. jan 2014. URL `http://proceedings.mlr.press/v32/silver14.html`.

[259] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. doi: 10.1038/nature16961. URL `http://dx.doi.org/10.1038/nature16961`.

[260] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*, dec 2017. URL `https://arxiv.org/abs/1712.01815`.

[261] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, October 2021. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103535. URL `https://www.sciencedirect.com/science/article/pii/S0004370221000862`.

[262] Patrice Y. Simard, Yann A. LeCun, John S. Denker, and Bernard Victorri. Transformation invariance in pattern recognition – tangent distance and tangent propagation. In Grégoire Montavon, Geneviève B.

Orr, and Klaus-Robert Müller, editors, *Neural networks: tricks of the trade: second edition*, volume 7700 of *Lecture notes in computer science*, pages 235–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35288-1. doi: 10.1007/978-3-642-35289-8_17. URL `http://link.springer.com/10.1007/978-3-642-35289-8_17`.

[263] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, sep 2014. URL `https://arxiv.org/abs/1409.1556`.

[264] Samarth Sinha and Adji B. Dieng. Consistency regularization for variational auto-encoders. *arXiv*, may 2021. URL `https://arxiv.org/abs/2105.14859`.

[265] Matthew Siper, Ahmed Khalifa, and Julian Togelius. Path of Destruction: Learning an Iterative Level Generator Using a Small Dataset, February 2022. URL `http://arxiv.org/abs/2202.10184`. arXiv:2202.10184 [cs].

[266] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Charles Nicholson, Nick Kreeger, Ping Yu, Shanqing Cai, Eric Nielsen, David Soegel, Stan Bileschi, et al. Tensorflow. js: Machine learning for the web and beyond. *Proceedings of Machine Learning and Systems*, 1:309–321, 2019.

[267] Laura Smith, Ilya Kostrikov, and Sergey Levine. A Walk in the Park: Learning to Walk in 20 Minutes With Model-Free Reinforcement Learning. URL `http://arxiv.org/abs/2208.07860`.

[268] Sam Snodgrass. Controllable Procedural Content Generation via Constrained Multi-Dimensional Markov Chain Sampling. page 7.

[269] Aravind Srinivas, Michael Laskin, and Pieter Abbeel. CURL: Contrastive unsupervised representations for reinforcement learning. *arXiv*, apr 2020. URL `https://arxiv.org/abs/2004.04136`.

[270] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using LSTMs. *arXiv*, feb 2015. URL `https://arxiv.org/abs/1502.04681`.

[271] Aleksandar Stanić, Yujin Tang, David Ha, and Jürgen Schmidhuber. Learning to Generalize with Object-centric Agents in the Open World Survival Game Crafter, August 2022. URL `http://arxiv.org/abs/2208.03374`. arXiv:2208.03374 [cs].

[272] Austin Stone, Oscar Ramirez, Kurt Konolige, and Rico Jonschkowski. The distracting control suite – a challenging benchmark for reinforcement

learning from pixels. *arXiv*, jan 2021. URL https://arxiv.org/abs/2101.02722.

[273] Joseph Suarez, Yilun Du, Clare Zhu, Igor Mordatch, and Phillip Isola. The Neural MMO Platform for Massively Multiagent Research, October 2021. URL http://arxiv.org/abs/2110.07594. arXiv:2110.07594 [cs].

[274] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. *arXiv*, may 2016. URL https://arxiv.org/abs/1605.07736.

[275] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural Content Generation via Machine Learning (PCGML), May 2018. URL http://arxiv.org/abs/1702.00539. arXiv:1702.00539 [cs].

[276] Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, feb 2000. URL https://www.researchgate.net/publication/{2503757_Policy_Gradient_Methods_for_Reinforcement_Learning_with_Function_Approximation}.

[277] Keisuke Suzuki, Warrick Roseboom, David J Schwartzman, and Anil K Seth. A deep-dream virtual reality platform for studying altered perceptual phenomenology. *Scientific Reports*, 7(1):15982, nov 2017. doi: 10.1038/s41598-017-16316-2. URL http://dx.doi.org/10.1038/s41598-017-16316-2.

[278] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv*, dec 2013. URL https://arxiv.org/abs/1312.6199.

[279] Yunhao Tang, Michal Valko, and Rémi Munos. Taylor expansion policy optimization. *arXiv*, mar 2020. URL https://arxiv.org/abs/2003.06259.

[280] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[281] Open Ended Learning Team, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard, and Wojciech Marian Czarnecki. Open-Ended Learning Leads to Generally Capable Agents. URL `http://arxiv.org/abs/2107.12808`.

[282] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, sep 2017. ISBN 978-1-5386-2682-5. doi: 10.1109/{IROS}.2017.8202133. URL `http://ieeexplore.ieee.org/document/8202133/`.

[283] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[284] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.

[285] Elise van der Pol, Thomas Kipf, Frans A. Oliehoek, and Max Welling. Plannable approximations to MDP homomorphisms: Equivariance under actions. *arXiv*, feb 2020. URL `https://arxiv.org/abs/2002.11963`.

[286] Elise van der Pol, Daniel E. Worrall, Herke van Hoof, Frans A. Oliehoek, and Max Welling. MDP homomorphic networks: Group symmetries in reinforcement learning. *arXiv*, jun 2020. URL `https://arxiv.org/abs/2006.16908`.

[287] Elise van der Pol, Herke van Hoof, Frans A. Oliehoek, and Max Welling. Multi-agent MDP homomorphic networks. *arXiv*, oct 2021. URL `https://arxiv.org/abs/2110.04495`.

[288] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv*, sep 2015. URL `https://arxiv.org/abs/1509.06461`.

[289] Herke van Hoof, Tucker Hermans, Gerhard Neumann, and Jan Peters. Learning robot in-hand manipulation with tactile features. In

2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids), pages 121–127. IEEE, nov 2015. ISBN 978-1-4799-6885-5. doi: 10.1109/{HUMANOIDS}.2015.7363524. URL `http://ieeexplore.ieee.org/document/7363524/`.

[290] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv*, jun 2017. URL `https://arxiv.org/abs/1706.03762`.

[291] Prateek Verma and Julius O. Smith. Neural style transfer for audio spectograms. *arXiv*, jan 2018. URL `https://arxiv.org/abs/1801.01589`.

[292] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A new challenge for reinforcement learning. *arXiv*, aug 2017. URL `https://arxiv.org/abs/1708.04782`.

[293] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P Agapiou, Max Jaderberg, Alexander S Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, oct 2019. ISSN 0028-0836. doi: 10.1038/s41586-019-1724-z. URL `http://www.nature.com/articles/s41586-019-1724-z`.

[294] Huiyu Wang, Yukun Zhu, Bradley Green, Hartwig Adam, Alan Yuille, and Liang-Chieh Chen. Axial-deeplab: Stand-alone axial-attention for panoptic segmentation. *arXiv*, mar 2020. URL `https://arxiv.org/abs/2003.07853`.

[295] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly

Complex and Diverse Learning Environments and Their Solutions, February 2019. URL http://arxiv.org/abs/1901.01753. arXiv:1901.01753 [cs].

[296] Nicholas Watters, Loic Matthey, Christopher P. Burgess, and Alexander Lerchner. Spatial broadcast decoder: A simple architecture for learning disentangled representations in VAEs. *arXiv*, jan 2019. URL https://arxiv.org/abs/1901.07017.

[297] Théophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. *arXiv*, jul 2017. URL https://arxiv.org/abs/1707.06203.

[298] Jason Wei and Kai Zou. EDA: easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6381–6387, Stroudsburg, PA, USA, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1670. URL https://www.aclweb.org/anthology/D19-1670.

[299] Maurice Weiler and Gabriele Cesa. General $e(2)$-equivariant steerable CNNs. *arXiv*, nov 2019. URL https://arxiv.org/abs/1911.08251.

[300] Nathaniel Weir, Xingdi (Eric) Yuan, Marc-Alexandre Côté, Matthew Hausknecht, Romain Laroche, Ida Momennejad, Harm van Seijen, and Ben Van Durme. One-Shot Learning from a Demonstration with Hierarchical Latent Language. March 2022. URL https://www.microsoft.com/en-us/research/publication/one-shot-learning-from-a-demonstration-with-hierarchical-latent-language/.

[301] Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine, June 2022. URL http://arxiv.org/abs/2206.10558. arXiv:2206.10558 [cs].

[302] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/{BF00992696}. URL http://link.springer.com/10.1007/{BF00992696}.

[303] Clemens Winter, Christopher Bamford, and Shengyi Huang. Entity-neural-network, 2022. URL `https://github.com/entity-neural-network`.

[304] Laurenz Wiskott and Terrence J Sejnowski. Slow feature analysis: unsupervised learning of invariances. *Neural Computation*, 14(4):715–770, apr 2002. doi: 10.1162/089976602317318938. URL `http://dx.doi.org/10.1162/089976602317318938`.

[305] Philipp Wu, Alejandro Escontrela, Danijar Hafner, Ken Goldberg, and Pieter Abbeel. DayDreamer: World Models for Physical Robot Learning, . URL `http://arxiv.org/abs/2206.14176`.

[306] Yan Wu, Greg Wayne, Alex Graves, and Timothy Lillicrap. The kanerva machine: A generative distributed memory. *arXiv*, apr 2018. URL `https://arxiv.org/abs/1804.01756`.

[307] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation, . URL `http://arxiv.org/abs/1708.05144`.

[308] Marek Wydmuch, Michal Kempka, and Wojciech Jaskowski. ViZDoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, pages 1–1, 2018. ISSN 2475-1502. doi: 10.1109/{TG}.2018.2877047. URL `https://ieeexplore.ieee.org/document/8500330/`.

[309] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. Unsupervised data augmentation for consistency training. *arXiv*, apr 2019. URL `https://arxiv.org/abs/1904.12848`.

[310] Denis Yarats, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto. Mastering visual continuous control: Improved data-augmented reinforcement learning. *arXiv*, jul 2021. URL `https://arxiv.org/abs/2107.09645`.

[311] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. Rotation, translation, and cropping for zero-shot generalization. *arXiv*, January 2020. URL `https://arxiv.org/abs/2001.09908`.

[312] Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data. URL `https://arxiv.org/abs/2111.00210`.

[313] Sangdoo Yun, Dongyoon Han, Sanghyuk Chun, Seong Joon Oh, Youngjoon Yoo, and Junsuk Choe. Cutmix: regularization strategy to

train strong classifiers with localizable features. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6022–6031. IEEE, oct 2019. ISBN 978-1-7281-4803-8. doi: 10.1109/{ICCV}.2019.00612. URL `https://ieeexplore.ieee.org/document/9008296/`.

[314] Yahia Zakaria, Magda Fayek, and Mayada Hadhoud. Procedural Level Generation for Sokoban via Deep Learning: An Experimental Study. *IEEE Transactions on Games*, pages 1–1, 2022. ISSN 2475-1510. doi: 10.1109/TG.2022.3175795. Conference Name: IEEE Transactions on Games.

[315] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia. Relational deep reinforcement learning. *arXiv*, jun 2018. URL `https://arxiv.org/abs/1806.01830`.

[316] Han Zhang, Zizhao Zhang, Augustus Odena, and Honglak Lee. Consistency regularization for generative adversarial networks. *arXiv*, oct 2019. URL `https://arxiv.org/abs/1910.12027`.

[317] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv*, oct 2017. URL `https://arxiv.org/abs/1710.09412`.

[318] Yuting Zhang, Yijie Guo, Yixin Jin, Yijun Luo, Zhiyuan He, and Honglak Lee. Unsupervised discovery of object landmarks as structural representations. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2694–2703. IEEE, jun 2018. ISBN 978-1-5386-6420-9. doi: 10.1109/{CVPR}.2018.00285. URL `https://ieeexplore.ieee.org/document/8578383/`.

[319] Victor Zhong, Tim Rocktäschel, and Edward Grefenstette. Rtfm: Generalising to new environment dynamics via reading. In *ICLR*, pages 1–17. ICLR, 2020.

[320] Guangxiang Zhu, Zhiao Huang, and Chongjie Zhang. Object-oriented dynamics predictor. *arXiv*, may 2018. URL `https://arxiv.org/abs/1806.07371`.

[321] Haosheng Zou, Tongzheng Ren, Dong Yan, Hang Su, and Jun Zhu. Reward shaping via meta-learning. *arXiv*, January 2019. URL `https://arxiv.org/abs/1901.09330`.

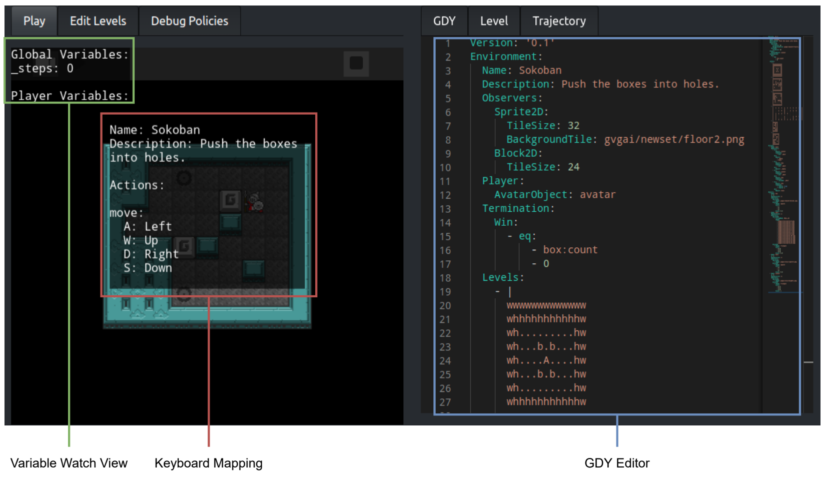
Appendix A

GriddlyJS

A.1 GriddlyJS UI Walkthrough

In this section we show various screenshots of the GriddlyJS user interface and highlight various useful features.

A.1.1 Building And Testing Environment Mechanics



```
Global Variables:
_steps: 0

Player Variables:
```

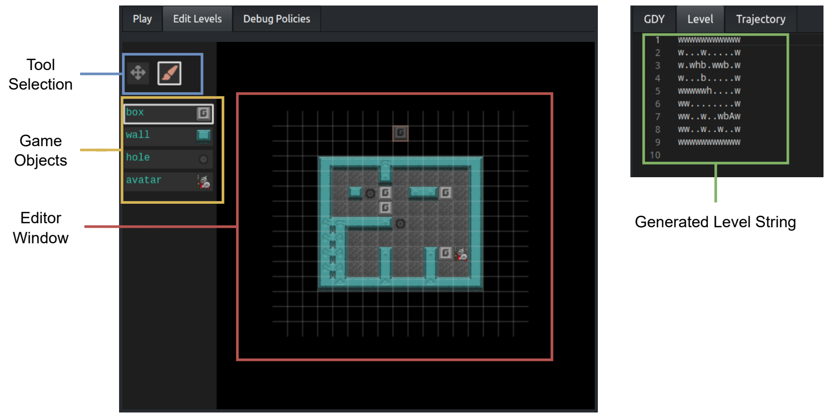
```
Name: Sokoban
Description: Push the boxes
into holes.

Actions:

move:
  A: Left
  W: Up
  D: Right
  S: Down
```

```
Version: '0.1'
Environment:
  Name: Sokoban
  Description: Push the boxes into holes.
  Observers:
    Sprite2D:
      TileSize: 32
      BackgroundTile: gvgai/newset/floor2.png
    Block2D:
      TileSize: 24
  Player:
    AvatarObject: avatar
  Termination:
    Win:
      - eq:
          - box:count
          - 0
  Levels:
    - |
      wwwwwwwwwwwww
      whhhhhhhhhhhw
      wh.........hw
      wh...b.b...hw
      wh....A....hw
      wh...b.b...hw
      wh.........hw
      whhhhhhhhhhhw
```

Figure A.1: Environment Designing and Debugging interfaces..

GriddlyJS provides many tools for building and debugging environments. Figure A.1 shows several of these. Firstly, as soon as GDY file is loaded in the editor and a level selected, it will be playable in the editor window. Actions in the environment are automatically mapped to the keyboard, and an explanation of this mapping can be toggled by pressing **P**. Additionally all global and playerwise variables can be toggled by pressing **I**. These variables are updated live while the game is being played.

A.1.2 Level Design



```
wwwwwwwwwww
w...w.....w
w.whb.wwb.w
w...b.....w
wwwwh.....w
ww........w
ww..w..wbAw
ww..w..w..w
wwwwwwwwwww
```

Figure A.2: Level Editor view and level string view..

The Level Editor view shown in Figure A.2 allows the user to choose objects to place on the game grid in order to create levels. The user can selects an object from the menu and then can *paint* it into the editing grid. The editing grid automatically grows if objects are placed near the boundaries, so levels of any shape or size can be created. Additionally as objects are painted into the editor grid, a level string is automatically generated and displayed. This level string can also be manually edited and its changes reflected in the editor window.

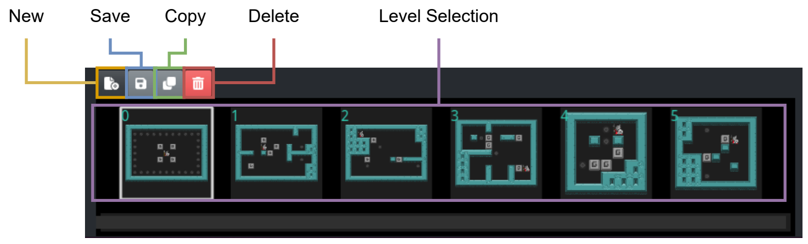
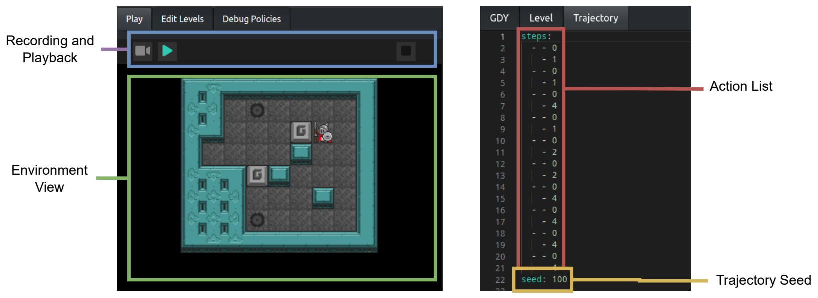


Figure A.3: The level selector view showing thumbnails of the the levels in the GDY..

Managing the set of levels in the environments GDY file is handled by the Level Selection interface shown in Figure A.3. Users can create, update and delete levels in the GDY file for quickly generating large datasets of levels.

A.1.3 Recording Trajectories



```yaml
steps:
  - - 0
    - 1
  - - 0
    - 1
  - - 0
    - 4
  - - 0
    - 1
  - - 0
    - 2
  - - 0
    - 2
  - - 0
    - 4
  - - 0
    - 4
  - - 0
    - 4
  - - 0
    - 4
seed: 100
```

Figure A.4: Recording and playback menus for generating and viewing trajectories.

Recording and playback options in the GriddlyJS interface. On clicking the **Record** icon, the user's actions are recorded as they play the game in the environment view. When the environment terminates, it is reset to its original state and a **play** button is shown next to the record button. If pressed, the play button re-plays the recorded trajectory. These options are shown in Figure A.4. Additionally, the actions and seed are displayed as YAML in the trajectory view in the editor. This trajectory can then be copied and stored for later use, for example in behavioural cloning algorithms. Trajectories can also be copied into the text-area from external sources and played within the editor.

A.1.4 Evaluating Models

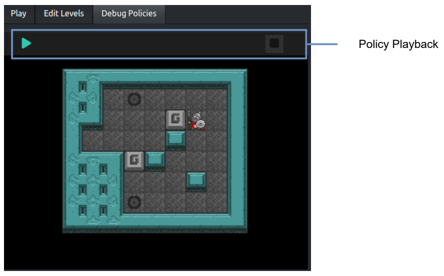


Figure A.5: Policy debugger view, similar to the "play" view but only visible if a policy is loaded using TensorflowJS.

Trained policies can be loaded into GriddlyJS and replayed using the Debug Policies view, shown in Figure A.5. If a model is loaded, a **play** button will be visible which will sample actions from the policy to view its performance. once the episode is finished, the level is reset.