Powderworld: A Platform for Understanding Generalization via Rich Task Distributions

by

Kevin Frans

B.S. Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2023)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Kevin Frans
Department of Electrical Engineering
and Computer Science
May 19, 2023

Certified by: Phillip Isola
Department of Electrical Engineering
and Computer Science
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Powderworld: A Platform for Understanding Generalization via Rich Task Distributions

by

Kevin Frans

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

One of the grand challenges of reinforcement learning is the ability to generalize to new tasks. However, general agents require a set of rich, diverse tasks to train on. Designing a 'foundation environment' for such tasks is tricky – the ideal environment would support a range of emergent phenomena, an expressive task space, and fast runtime. To take a step towards addressing this research bottleneck, this work presents Powderworld, a lightweight yet expressive simulation environment running directly on the GPU. Within Powderworld, two motivating challenges are presented, one for world-modelling and one for reinforcement learning. Each contains hand-designed test tasks to examine generalization. Experiments indicate that increasing the environment's complexity improves generalization for world models and certain reinforcement learning agents, yet may inhibit learning in high-variance environments. Powderworld aims to support the study of generalization by providing a source of diverse tasks arising from the same core rules.

Thesis Supervisor: Phillip Isola
Title: Associate Professor

Contents

List of Figures

Chapter 1

Introduction

One of the grand challenges of reinforcement learning (RL), and of decision-making in general, is the ability to generalize to new tasks. RL agents have shown incredible performance on single task settings [4, 22, 23], yet frequently stumble when presented with unseen challenges. Single-task RL agents are largely overfit on the tasks they are trained on [20], limiting their practical use. In contrast, a general agent, which can robustly perform well on a wide range of novel tasks, can then be adapted to solve downstream tasks and unseen challenges.

General agents greatly depend on a diverse set of tasks to train on. Recent progress in deep learning has shown that as the amount of data increases, so do generalization capabilities of trained models [7, 30, 6, 28]. Agents trained on environments with domain randomization or procedural generation capabilities transfer better to unseen test tasks [8, 38, 31, 19]. However, as creating training tasks is expensive and challenging, most standard environments are inherently over-specific or limited by their focus on a single task type, e.g. robotic control or gridworld movement.

As the need to study the relationships between training tasks and generalization increases, the RL community would benefit greatly from a 'foundation environment' supporting diverse tasks arising from the same core rules. The benefits of expansive task spaces have been showcased in Unsupervised Environment Design [39, 9, 16, 25], but gridworld domains fail to display how such methods scale up. Previous works have proposed specialized task distributions for multi-task training [34, 36, 10, 37], each

focusing on a specific decision-making problem. To further investigate generalization, it is beneficial to have an environment where many variations of training tasks can easily be compared.

As a step toward lightweight yet expressive environments, this paper presents Powderworld, a simulation environment geared to support procedural data generation, agent learning, and multi-task generalization. Powderworld aims to efficiently provide environment dynamics by running directly on the GPU. Elements (e.g. sand, water, fire) interact in a modular manner within local neighborhoods, allowing for efficient runtime. The free-form nature of Powderworld enables construction of tasks ranging from simple manipulation objectives to complex multi-step goals. Powderworld aims to 1) be modular and supportive of emergent interactions, 2) allow for expressive design capability, and 3) support efficient runtime and representations.

Additionally presented is an extensive framework for defining reinforcement learning tasks with Powderworld. Tasks are viewed as singular points with a larger "task space", of which three axes of variation are specified – agent space, state generation space, and reward space. Within this framework, a variety of hand-designed objectives are presented, along with wrappers to add stochastic augmentations to each task.

Experiments focus on the performance of world models and reinforcement learning agents as environment complexity is increased. World models trained on increasingly complex environments show superior transfer performance. In addition, models trained over more element types show stronger fine-tuning on novel rulesets, demonstrating that a robust representation has been learned. In the reinforcement learning case, increases in task complexity benefit generalization up to a task-specific inflection point, at which performance decreases. This point may mark when variance in the resulting reward signal becomes too high, inhibiting learning. These findings provide a starting point for future directions in studying generalization using Powderworld as a foundation.

Chapter 2

Related Work

2.1 Task Distributions for Reinforcement Learning

Video games are a popular setting for studying multi-task RL, and environments have been built off NetHack [34, 21], Minecraft [10, 17, 11], Doom [18], and Atari [3]. [37, 40, 8] describe task distributions focused on meta-learning, and [10, 36, 13, 27] detail more open-ended environments containing multiple task types. The Atari benchmark is a motivating example of a multi-task generalization benchmark; as some methods attempt to train on a subset of games, then generalize during test time to held-out games. However, there are only a fixed number of Atari games, and the games are largely disctinct from one another. Powderworld instead presents a multi-task environment where an unbounded number of tasks can be defined, thus presenting a denser training distribution. Most similar to this work may be ProcGen [8], a platform that supports infinite procedurally generated environments. However, while ProcGen games each have their own rulesets, Powderworld aims to share core rules across all tasks. Powderworld focuses specifically on runtime and expressivity, taking inspiration from online “powder games" where players build ranges of creations out of simple elements [1, 2, 5].

2.2 Generalization in Reinforcement Learning

Multi-task reinforcement learning agents are generally valued for their ability to perform on unseen training tasks [24, 20]. The sim2real problem requires agents aim to generalize to out-of-distribution real world domains [38, 33]. The platforms cited above also target generalization, often within the context of solving unseen levels within a game. This work aims to study generalization within a physics-inspired simulated setting, and creates out-of-distribution challenges by hand-designing a set of unseen test tasks.

Chapter 3

Powderworld Environment

The main contribution of this work is the Powderworld simulator, an environment specifically for examining agent generalization over customizable distributions of tasks. Towards this aim, Powderworld is built to optimize the following design principles:

- **Modularity and support for emergent phenomena.** The core of Powderworld is a set of fundamental rules defining how two neighboring elements interact. The consistent nature of these rules is key to agent generalization; e.g. fire will always burn wood, and agents can learn these inherent properties of the environment. Furthermore, local interactions can build up to form emergent wider-scale phenomena, e.g. fire spreading throughout the world. This capacity for emergence enables tasks to be diverse yet share consistent properties. Thus, fundamental Powderworld priors exist that agents can take advantage of to generalize.

- **Expressive task design capability.** A major blocker in the study of reinforcement learning generalization is that tasks are often nonadjustable. Instead, an ideal environment should present an explorable space of tasks, capable of representing interesting challenges, goals, and constraints. Tasks should be parametrized to allows for automated design and interpretable control. Powderworld presents a task framework supporting an unbounded number of individual tasks. Each task is represented as a set of procedural generation rules, and goal

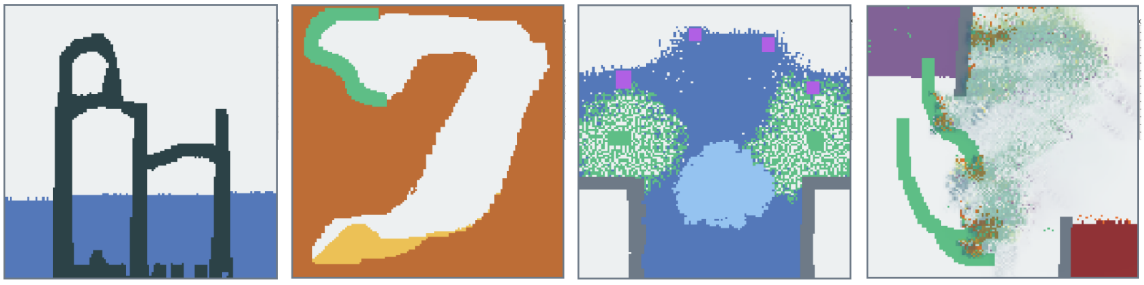

Figure 3-1: **Examples of tasks created in the Powderworld engine.** Powderworld provides a physics-inspired simulation over which many distributions of tasks can be defined. Pictured above are human-designed challenges where a player must construct unstable arches, transport sand through a tunnel, freeze water to create a bridge, and draw a path with plants. Tasks in Powderworld creates challenges from a set of core rules, allowing agents to learn generalizable knowledge.

states are simple 2D arrays. A multitude of ways exist to test a specific agent capability, e.g. “burn plants to create a gap". Additionally, all tasks in Powderworld require the agent to interact with the *dynamics* of the world, thus even simple goals like “create plants" may require complex strategies to achieve such goals through interaction.

- **Fast runtime and representation.** As multi-task learning can be computationally expensive, it is important that the underlying environment runs efficiently. Powerworld is designed to run on the GPU, enabling large batches of simulation to be run in parallel. Additionally, Powderworld employs a neural-network-friendly matrix representation for both task design and agent observations. To simplify the training of decision-making agents, the Powderworld representation is fully-observable and runs on a discrete timescale (but partial-observability is an easy modification if desired).

3.1 Core Engine

In the following section, an overview of the engine used for Powderworld simulator is provided.

World matrix. The core structure of Powderworld is a matrix of elements W representing the world. Each location $W_{x,y}$ holds a vector of information representing that location in the world. Specifically, each vector contains 9 channels:

1. The ID of the occupying element. [0,20]
2. The density of the occupying element. [0,4]
3. Flag if element is affected by gravity. [0,1]
4. X-component of Velocity Field. $[-\infty, \infty]$
5. Y-component of Velocity Field. $[-\infty, \infty]$
6. Color variable. (Unused)
7. Custom variable 1.
8. Custom variable 2.
9. Custom variable 3.

In practice, the world matrix is stored in batch form as a $[B, W, H, 9]$ Pytorch tensor. This W matrix is a Markovian state of the world, and thus past W matrices are unnecessary for state transitions. Every timestep, a new W matrix is generated via a stochastic update function, as described below.

Gravity. Certain elements are affected by gravity, as noted by the IsGravity flag in Figure 3-2. Each gravity-affected element also holds a density value, which determines the element's priority during the gravity calculation. Every timestep, each element checks with its neighbor below. If both elements are gravity-affected, and the neighbor below has a lower density, then the two elements swap positions. This interaction functions as a core rule in the Powderworld simulation and allows elements to stack, displace, and block each other. Note that it is possible that two elements will point towards occupying the same position – in this case, ties are broken by Y-value.

Element-specific reactions. The behavior of Powderworld arises from a set of modular, local element reactions. Element reactions can occur either within a single element, or as a reaction when two elements are neighbors to each other. These reactions are designed to facilitate larger-scale behaviors; e.g. the sand element falls to neighboring locations, thus areas of sand form pyramid-like structures. Elements such as water, gas, and lava are fluids, and move horizontally to occupy available space. Finally, pairwise reactions provide interactions between specific elements, e.g. fire spreads to flammable elements, and plants grow when water is nearby. See Figure 3-2 and below for a description of the Powderworld reactions.

Velocity system. Another interaction method is applying movement through the velocity system. Certain reactions, such as fire burning or dust exploding, add to the velocity field. Velocity is represented via an two-component $V_{x,y}$ vector at each world location. If the magnitude of the velocity field at a location is greater than a threshold, elements are moved in one of eight cardinal directions, depending on the velocity angle. This procedure is run twice per timestep; an element with sufficient velocity can move two positions per timestep. Velocity naturally diffuses and spreads in its own direction, thus a velocity difference will spread outwards before fading away. Walls are immune to velocity affects.

All operators are local and translation equivariant, yielding a simple implementation in terms of (nonlinear) convolutional kernels. To exploit GPU-optimized operators, Powderworld is implemented in Pytorch [26], and performance scales with GPU capacity (Figure 3-3).

3.2 Element Descriptions

In the following list, a description of the Powderworld elements is provided along with their behavior and reactions.

1. **Empty**. The default element, it has no special properties and does not directly have any behaviors. Each coordinate in Powderworld always contains an element, so "empty" is defined as an element in itself.

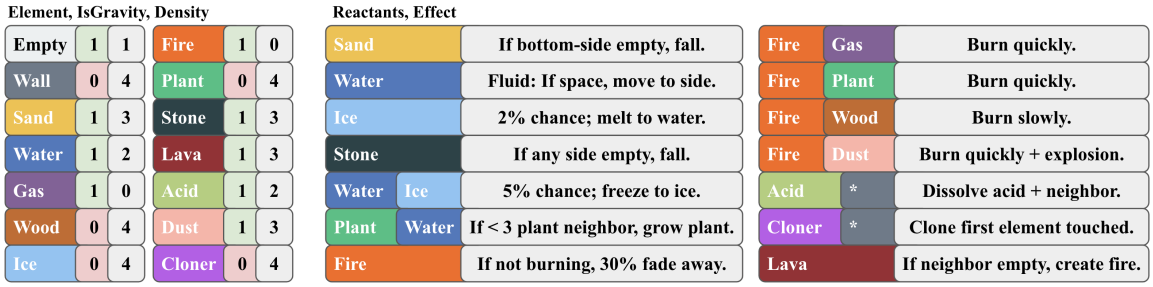

Figure 3-2: **A list of elements and reactions in the Powderworld simulation.** Elements each contain gravity and density information. A set of element-specific reactions dictates how each element behaves and reacts to neighbors. Certain reactions manipulate the world's velocity field, which can push further elements away. Together, the gravity, velocity, and reaction systems create a core set of rules by which interesting simulations arise.

2. **Wall.** Wall is also an element that has no direct interactions with other elements. Wall is not affected by the gravity or velocity systems, and there are no elements which can create or displace it. Thus, Wall is useful for defining barriers or constraints in a task.

3. **Sand.** Sand is a basic element that is affected by gravity. Sand also falls into lower density positions to the bottom-left or bottom-right, leading to a steady state of pyramid-style formations.

4. **Water.** Water is a fluid that is affected by gravity. As a fluid, water will move to adjacent lower density areas to the left or right. Water conserves its own momentum, and thus will continue moving right if it has moved right in the past. Upon hitting an obstacle, a Water element will change directions. Overall, Water will flow down slopes and fill up containers.

5. **Gas.** Gas is a fluid with a density lower than empty. Thus, it will naturally move upwards. Since Gas is a fluid, it additionally fills space to the left and right. Gas is flammable and will spread Fire at a moderate rate.

6. **Wood.** Wood is a solid element that is not affected by gravity. Wood is stationary and largely does not react with other elements, although it is flammable and spreads Fire at a slow rate.

7. **Ice.** Ice is a solid element not affected by gravity. If exposed to greater than three Empty neighbors, Ice will have a chance of melting to form Water. In turn, Water that has three or more Ice neighbors will have a chance of freezing into Ice. Thus large clumps of Ice can form a steady state, while small spots of Ice will quickly melt into Water.

8. **Fire.** Fire is a special element that has interactions with many flammable elements. Fire has a lower-than-empty density and thus naturally rises upwards. Fire has a chance of dissipating every timestep, unless it is neighbored by a flammable element. Thus Fire is only present temporarily when burning other elements, otherwise quickly burning out. When an element burns, it produces more Fire in empty areas around it, and creates an outwards velocity.

9. **Plant.** Plant is a solid non-gravitational element that is flammable. Plants spread through Water – if a Water element is neighbored by a Plant, it has a chance to also turn into Plant. If the Water is neighbored by more than four Plants, however, it instead may turn into Empty. This behaviors forms a natural pattern of Plants and Empty elements spreading through water. Plants also spread along the surface of Wood and Ice, forming a one-element thick border.

10. **Stone.** Stone is a solid gravity-affected element. Stone has a high density and falls through fluids. If a Stone element is surrounded by Stone on both left and right sides, it ignores gravity. Thus, Stone formations can form arches and bridges if correctly supported from both sides.

11. **Lava**. Lava is a fluid that flows similarly to Water. Lava continually produces Fire at neighboring positions. When Lava collides with Water, Stone is formed.

12. **Acid**. Acid is a fluid that dissolves other elements when coming into contact with them. Certain elements such as Wall and Cloner are immune to the acid reaction. When an element reacts with acid, both elements are destroyed and a Gas element is created.

13. **Dust.** Dust is an element that behaviors similar to Sand, forming pyramid-like structures. Dust is also highly flammable and creates large amounts of velocity when burned.

14. **Cloner.** Cloner is a special element that remembers the first non-Empty element that touches it, then continually produces that element at neighboring positions. A Cloner element therefore acts as a source for other elements, especially elements that naturally flow away.

3.3 Agent Descriptions

In addition to the natural elements described above, a set of cellular automata agents are implemented within Powderworld. Internally, these agents are implemented as elements themselves; the agents are considered elements 14-19. Agents therefore take up space within the world and can be moved via the gravity and velocity systems.

1. **Fish.** Fish is a simple agent that randomly moves in the four cardinal directions. Its density is equal to Water, so Fish in Water will naturally diffuse to fill the space.

2. **Bird.** Birds are agents which following a boiding behavior. Specifically, Birds have an internal direction $[0, 2 * \pi]$, and move by applying velocity in that direction. To enforce the boiding behavior, Birds update their internal direction as a sum of 1) the average direction of any other birds within a 15x15 radius, plus 2) directional vectors pointing away from any nearby non-empty elements. Put together, the Bird update rule results in flocks of Birds that flow with each other yet do not collide.

3. **Kangaroo.** Kangaroo is a gravity-obeying agent that moves left and right. With a small chance each timestep, if a Kangaroo is ontop of a solid element, the Kangaroo will jump upwards. The jump is implemented as a small impulse in velocity at the Kangaroo's position.

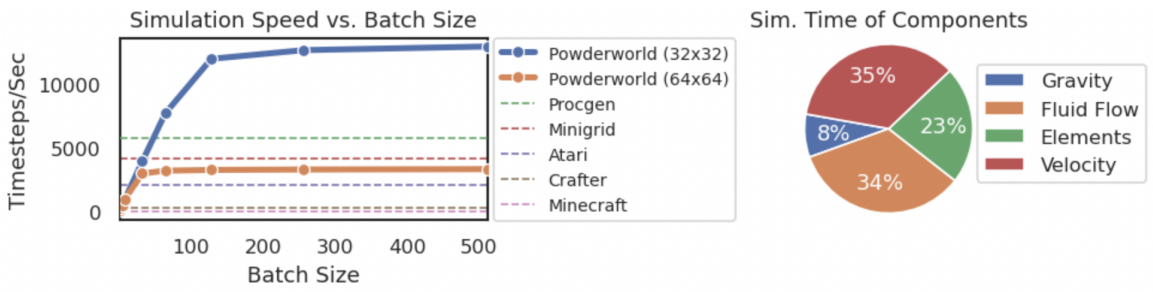

Figure 3-3: **Powderworld runs on the GPU and can simulate many worlds in parallel.** GPU simulation provides a significant speedup and allows simulation time to scale with batch size. Simulation speed is guaranteed to remain constant regardless of how many elements are present in the world.

4. **Mole.** Mole is an agent that burrows through solid elements with density $>$ 3. Moles carry an internal cardinal direction, and continue moving in that direction. With a random chance, a Mole will change direction. As long as a Mole is moving through an element, the Mole will continue and destroy that element. If such an element is not present, the Mole will stop moving.

5. **Lemming.** Lemmings are gravity-obeying agents similar to the Kangaroo. Instead of occasionally jumping, a Lemming will move left/right and can ascend one-element heights. Thus a Lemming can climb a pyramid by walking across it, but will change direction if it encounters any barrier.

6. **Snake.** Snakes move in the four cardinal directions, randomly changing direction every few timesteps. Snakes follow the direction of any Snakes ahead of themselves, thus, individual Snakes form longer multi-position Snake entities. Snakes burrow through non-Wall elements and produce Acid.

3.4 Technical Details

3.4.1 Pytorch Module

The above operators are implemented as sub-modules of a global Pytorch Powderworld module, which is referred to as the *PWSim module*. The PWSim module is

stateless and simply aims to simulate forwards an input W matrix. All the operators and behaviors are implemented as parallelizable matrix operators. Thus, computation is calculated across spatial positions and across the batch dimension simultaneously. This design grants a large speedup as the GPU can be utilized for quick computation, but it requires careful design as all elemental reactions must be designed to simultaneously update. Common failure cases involve multiple elements attempting to swap into the same position, or a single element duplicating itself into multiple positions.

One bottleneck in the GPU utilization is the bandwith between SRAM and DRAM. Because the PWSim module is comprised of many small CUDA operators, the wall-clock bottleneck becomes data transfer within the GPU memory system. To alleviate this issue, operator fusion can be used. By performing all elementwise operations in sequence, a chunk of operations can be sent as a single CUDA call, speeding up the process. PWSim utilizes this fusion through 'torch.jit'.

3.4.2 ONNX Format.

Because the PWSim module is a stateless Pytorch module, the static graph of the module can be exported into the ONNX format. This allows the Powderworld simulator to be run locally on various devices, including the browser via WebGL.

Chapter 4

Powderworld Task Distribution

A large motivation behind Powderworld is as its use as an environment for defining a large amount of reinforcement learning tasks. In comparison to multi-task domains such as Atari or ProcGen, Powderworld defines all of its tasks as objectives within the *same simulation environment.* This design choice means that even though the goals and constraints of a task may change, the underlying behavior of the environment is consistent.

Another design objective in creating the Powderworld task distribution is that tasks should form a smooth *task space*. In the environments mentioned above, tasks are distinct games and thus are largely unrelated to each other. In Powderworld, tasks are parametrized, forming a method of sampling new tasks with similar parameters. Thus it is possible to create 1000 tasks that are slightly similar to a given task, or interpolate between one task and another.

The general Powderworld task is defined as an OpenAI Gym environment with observation space $64x64x1$. Each spatial position contains an integer encoding of the element occupying that position. Internally, a policy should learn an observation layer similar to a vocabulary embedding and learn an N-dimensional embedding corresponding to each element.

Agents are viewed as disembodied policies who can interact with the world by placing elements. The action space is a set of discrete actions of size $[20, 8, 8, 8, 8,]$. The first dimension corresponds to which element should be placed into the world.

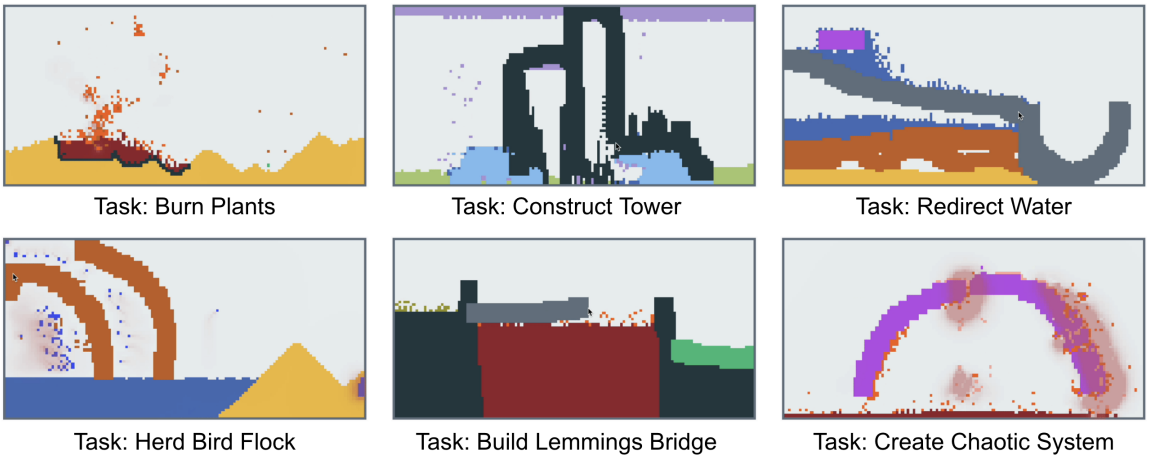

Figure 4-1: **Examples of reinforcement learning tasks in Powderworld.** A flexible framework for defining large distributions of tasks allows Powderworld to be used as a benchmark for multi-task generalization.

The next two dimensions correspond to the X and Y bins where the new element should be placed. Each bin is an 8x8 section of the world. The final two dimensions dictate the XY offset within the bin. Internally, an agent may select each action independently or autoregressively in sequence. Once an action is specified, a 3x3 square of the chosen element is placed at the chosen XY position.

Each task additionally contains a goal state, which is a $[64x64]$ matrix of element IDs, plus a $[64x64]$ element of reward weights. Agents receive reward based on which elements of the world match the goal state, multiplied element-wise by the reward weights of the location. The objective for the agent is therefore to manipulate the world state into resembling the goal state.

4.1 Task Space

Tasks within Powderworld can be viewed as points within a 3-dimensional task space. This space is largely for design purposes, as a task's position with each dimension is represented not as a continuous value but as a set of properties. The three axes are as follows:

1. **Agent Space.** This axis determines the properties of the agent. How many actions can the agent take within an episode? How many timesteps are simulated

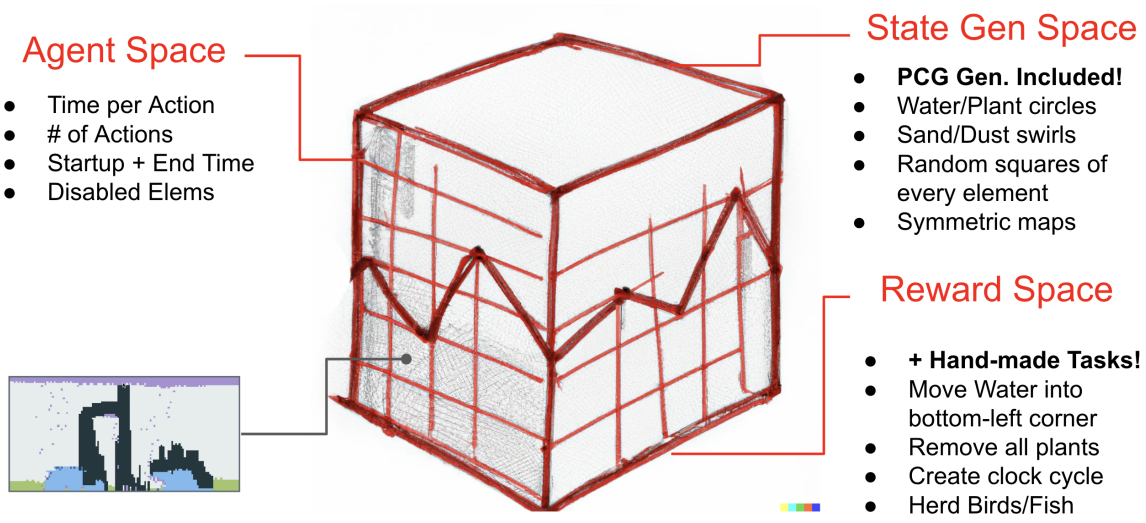

Figure 4-2: **Tasks in Powderworld can be seen as points in a 3D task space.** The dimensions of Agent Space, State Gen Space, and Reward Space categorize common axes of variation. Within the global task space, smaller sections can be used as training or test tasks.

between each action? Which elements is the agent allowed to select?

2. **State Gen Space.** This axis determines the procedural generation algorithm used to generate starting states. A set of modular rules defines the algorithm, e.g. "Create 3 circles of Stone; Create a sine wave of Lava".

3. **Reward Space.** This axis determines the goal state of the task. It is represented as the goal matrices described above.

Powderworld tasks are represented programatically as JSON objects encoding the properties above.

4.2 Distributions of Tasks

A core design choice in Powderworld is that agents should be trained and evaluated over *distributions* of tasks. For this purpose, the provided Powderworld gym environment is characterized by a task generator function. This generator function randomly samples and returns a JSON task. Task distributions can be as small as always returning the same task, or may return a wide range of tasks.

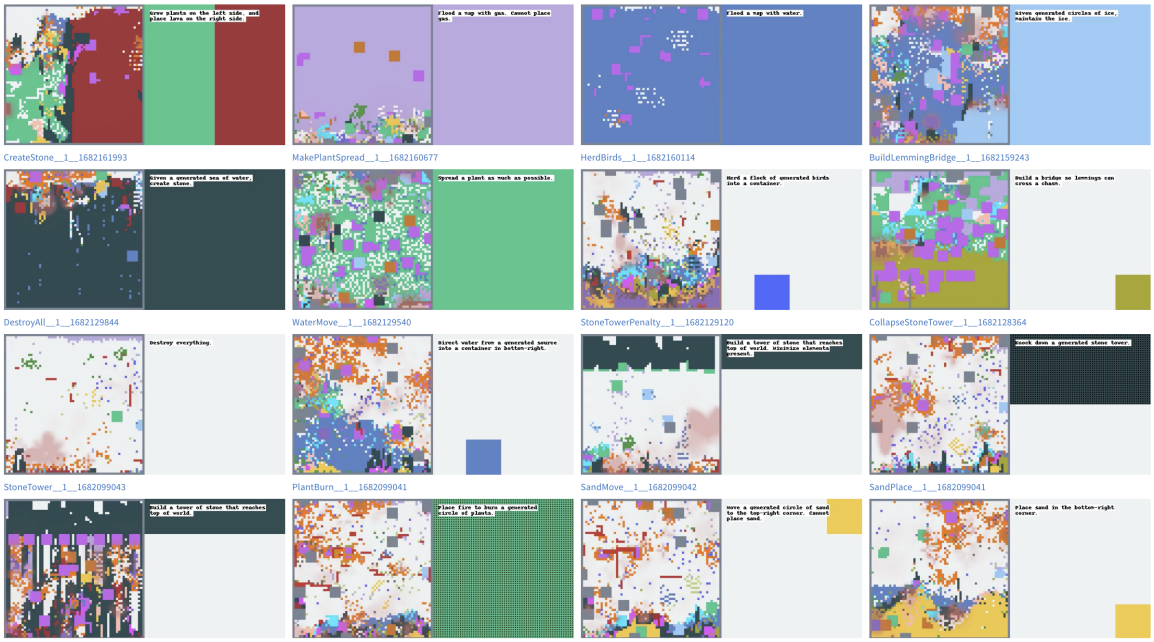

Figure 4-3: **Example observations from 16 Powderworld tasks.** On the left is the state of the world, and on the right is the goal state given to the agent. Agents learn to utilize a variety of complex strategies to manipulate the world into resembling the goal.

This work presents a few frameworks for easily defining task distributions. First, a set of 30 hand-defined training goals and 10 test goals are provided. These goals are functions that return matrices as described above. Starting from a randomly sampled goal, parameters describing agent space can also be randomly sampled – the agent may have 20 interactions with the world, or 15, or 100.

State generation space can also be easily set using random sampling. The procedural generation algorithm operates over a set of generation commands, with arguments dictating the element type as well as a shape. These shapes range from concrete objects such as circles and squares, to functional shapes such as arches, sine waves, or ramps.

Finally, augmentation functions can further add randomness to a generated state. Two augmentations are provided: one adds random perturbations to state generation, and one adds perturbations to the goal matrix. Together, these augmentations allow for any task distribution to be further expanded within the global task space.

Chapter 5

Experiments

The following section presents a series of motivating experiments showcasing task distributions within Powderworld. These tasks intend to provide two frameworks for accessing the richness of the Powderworld simulation, one through supervised learning and one through reinforcement learning. While these tasks aim to specifically highlight how Powderworld can be used to generate diverse task distributions, the presented tasks are by no means exhaustive, and future work may easily define modifications or additional task objectives as needed.

In all settings, the model is provided the $W \in [H, W, 20]$ matrix as an observation, which is a Markovian state containing element, gravity, density, and velocity information. All task distributions also include a procedural generation algorithm for generating training tasks, as well as tests used to measure transfer learning.

In all experiments below, evaluation is on out-of-distribution tests which are unseen during training.

5.1 World Modelling

This section examines a world-modelling objective in which a neural network is given an observation of the world, and must then predict a future observation. World models can be seen as learning how to encode an environment's dynamics, and have proven to hold great practical value in downstream decision making [12, 15, 14]. A model

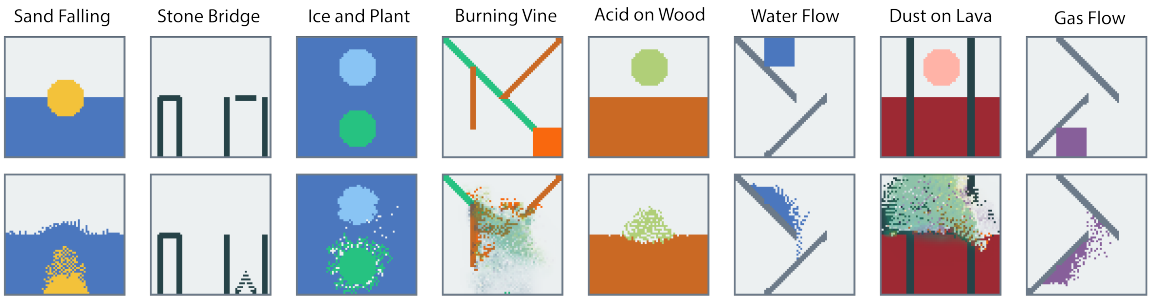

Figure 5-1: **World modelling test states are designed to showcase specific element interactions.** Test states are out-of-distribution and unseen during training. Model generalization capability is measured by how accurate its future predictions are on all eight tests.

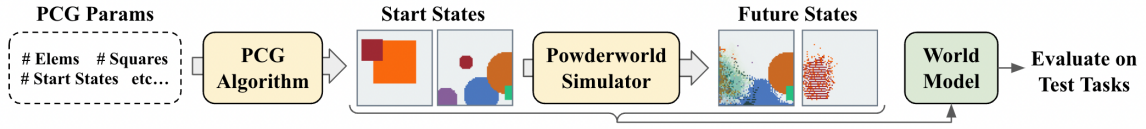

Figure 5-2: **Training states are generated via a procedural content generation (PCG) algorithm followed by Powderworld simulation.** Experiments examine the affect of increasing complexity in PCG parameters.

which can correctly predict the future of any observation can be seen as thoroughly understanding the core rules of the environment. The world-modelling task does not require reinforcement learning, and is instead solved via a supervised objective with the future state as the target.

Specifically, given an observation $W^0 \in R^{H \times W \times N}$ of the world, the model is tasked with generating a $W' \in R^{H \times W \times N}$ matrix of the world 8 timesteps in the future. W' values corresponds to logit probabilities of the N different elements, and loss is computed via cross-entropy between the true and predicted world. Tasks are represented by a tuple of starting and ending observations.

Training examples for the world-modelling task are created via an parametrized procedural content generation (PCG) algorithm. The algorithm synthesizes starting states by randomly selecting elements and drawing a series of lines, circles, and squares. Thus, the training distribution can be modified by specifying how many of each shape to draw, out of which elements, and how many total starting states should be generated. A set of hand-designed tests are provided as shown in Figure 5-1 which each measures a distinct property of Powderworld, e.g. simulate sand falling through

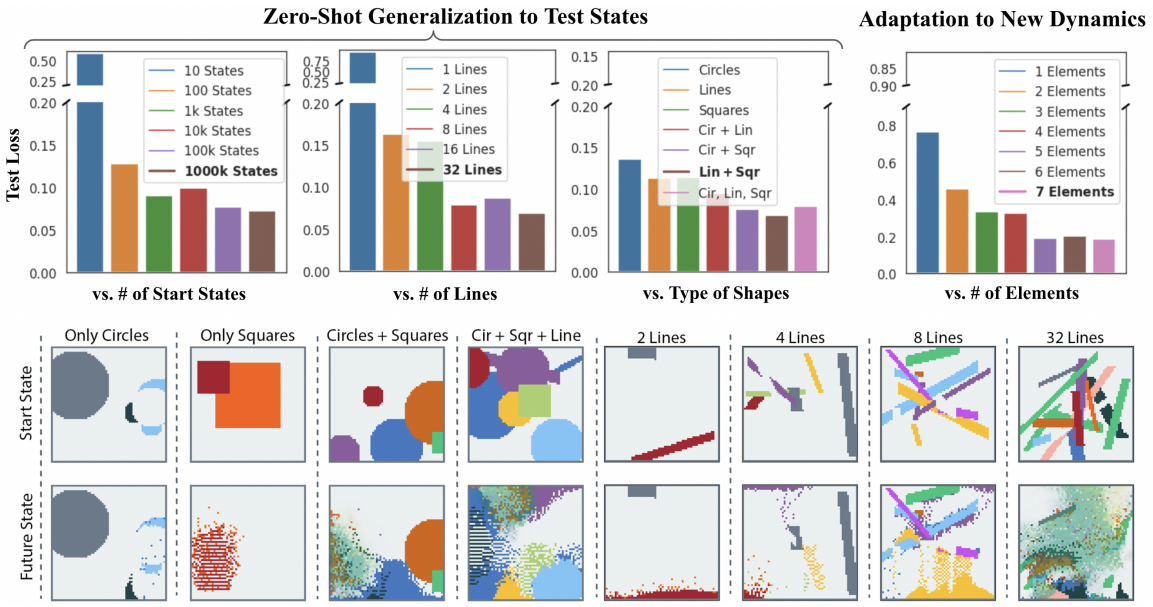

Figure 5-3: **World model generalization improves as training distribution complexity is increased.** Shown are the test performances of world models trained with data from varying numbers of start states, number of lines, and types of shapes. By learning from diverse data, world models can better generalize to unseen test states. **Top-Right: World models trained on more elements can better fine-tune to novel elements.** These results show that Powderworld provides a rich enough simulation that world models learn robust representations capable of adaptation to new dynamics. **Bottom:** Examples of states generated with various PCG parameters.

water, fire burning a vine, or gas flowing upwards. To generate the targets, each starting state is simulated forwards for 8 timesteps, as shown in Figure 5-2.

The model is a convolutional U-net network [32], operating over a world size of 64x64 and 14 distinct elements. The agent network consists of three U-net blocks with 32, 64, and 128 features respectively. Each U-net block contains two convolutional kernels with a kernel size of three and ReLU activation, along with a MaxPool layer in the encoder blocks. The model is trained with Adam for 5000 iterations with a batch size of 256 and learning rate of 0.005. During training, a replay buffer of 1024*256 data points is randomly sampled to form the training batch, and the oldest data points are rotated out for fresh examples generated via the Powderworld simulator.

Can world models generalize to unseen test states?

A starting experiment examines whether world models trained purely on simulated data can correctly generalize on hand-designed test states. The set of tests, as shown in Figure 5-1, are out-of-distribution hand-designed worlds that do not appear in the training set. A world model must discover the core ruleset of environmental dynamics in order to successfully generalize.

Scaling laws for training large neural networks have shown that more data consistently improves performance kaplan2020scaling, zhai2022scaling. Figure 5-3 shows this observation to be true in Powderworld as well; world models trained on increasing amounts of start states display higher performance on test states. Each world model is trained on the same number of training examples and timesteps, the only difference is how this data is generated. The average test loss over three training runs are displayed.

Results show that the 10-state world model overfits and does not generalize to the test states. In contrast, the 100-state model achieves much higher test accuracy, and the trend continues as the number of training tasks improves. These results show that the Powderworld world-modelling task demonstrates similar scaling laws as real-world data.

How do increasingly complex training tasks affect generalization?

As training data expands to include more varieties of starting states, does world model performance over a set of test states improve? More complex training data may allow world models to learn more robust representations, but may also introduce variance which harms learning or create degenerate training examples when many elements overlap.

Figure 5-3 displays how as additional shapes are included within the training distribution, zero-shot test performance successfully increases. World models are trained on distributions of training states characterized by which shapes are present between lines, circles, and square. Lines are assigned a random (X^1,Y^1), (X^2,Y^2),

and thickness. Circles and Squares are assigned a random (X^1,Y^1) along with a radius. Each shape is filled in with a randomly selected element. Between 0 and 5 of each shape are drawn. Interestingly, training tasks with less shape variation also display higher instability, as shown in the test loss spikes for Line-only, Circle-only, and Square-only runs. Additionally, world models operating over training states with a greater number of lines display higher test performance. This behavior may indicate that models trained over more diverse training data learn representations which are more resistant to perturbations.

Results showcase how in Powderworld, as more diverse data is created from the same set of core rules, world models increase in generalization capability.

Does environment richness influence transfer to novel interactions?

While a perfect world model will always make correct predictions, there are no guarantees such models can learn new dynamics. This experiment tests the *adaptability* of world models, by examining if they can quickly fine-tune on new elemental reactions.

Powderworld's ruleset is also of importance, as models will only transfer to new elements if all elements share fundamental similarities. Powderworld elements naturally share a set of behaviors, e.g. gravity, reactions-on-contact, and velocity. Thus, this experiment measures whether Powderworld presents a rich enough simulation that models can generalize to new *rules* within the environment.

To run the experiment, distinct world models are trained on distributions containing a limited set of elements. The 1-element model sees only sand, the 2-element sees only sand and water, the 3-element sees sand, water, and wall, and so on. Worlds are generated via the same procedural generation algorithm, specifically up to 5 lines are drawn. After training for the standard 5000 iterations, each world model is then fine-tuned for 100 iterations on a training distribution containing three held-out elements: gas, stone, and acid. The world model loss is then measured on a new environment containing only these three elements.

Figure 5-3 (top-right) highlights how world models trained on increasing numbers of elements show greater performance when fine-tuned on a set of unseen elements.

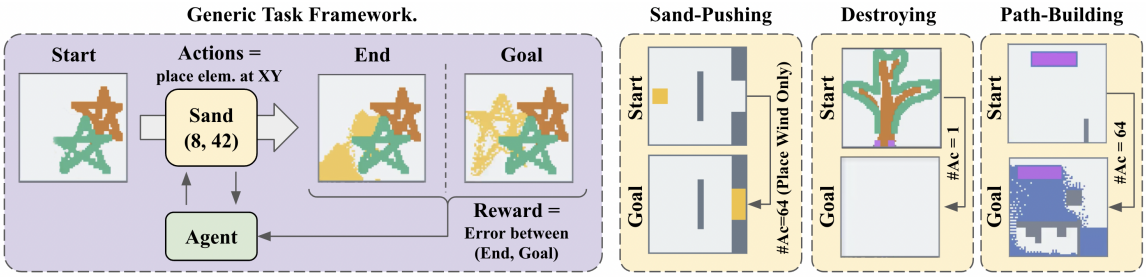

Figure 5-4: **In Powderworld RL tasks, agents must iteratively place elements (including directional wind) to transform a starting state into a goal state.** Within this framework, we present three RL tasks as shown above. Each task contains many challenges, as starting states are randomly generated for each episode. Agents are evaluated on test states that are unseen during training.

These results indicate that world models trained on richer simulations also develop more robust representations, as these representations can more easily be trained on additional information. Powderworld world models learn not only the core rules of the world, but also general features describing those rules, that can then be used to learn new rules.

5.2 Reinforcement Learning Tasks

Note: This section was written before the RL Task Space described in Chapter 4 was developed. It is included here for completeness, but the following RL experiments use an older version of the Powderworld RL setup.

Reinforcement learning tasks can be defined within Powderworld via a simple framework, as shown in Figure 5-4. Agents are allowed to iteratively place elements, and must transform a starting state into a goal state. The observation space contains the Powderworld world state $W \in R^{64\times64\times20}$, and the action space is a multi-discrete combination of $X, Y, Element, V_x, V_y$. V_x and V_y are only utilized if the agent is placing wind.

Tasks are defined by a function that generates a starting state, a goal state, and any restrictions on element placement. Note that Powderworld tasks are specifically designed to be stochastically diverse and contain randomly generated starting states. Within this framework, many task varieties can be defined. This work considers:

- **Sand-Pushing.** The Sand-Pushing environment is an RL environment where an agent must move sand particles into a goal slot. The agent is restricted to only placing wind, at a controllable velocity and position. By producing wind, agents interact with the velocity field, allowing them to push and move elements around. Wind affects the velocity field in a 10x10 area around the specified position. Reward equals the number of sand elements within the goal slot, and episodes are run for 64 timesteps. The Sand-Pushing task presents a sparse-reward sequential decision-making problem.

- **Destroying.** In the Destroying task, agents are tasked with placing a limited number of elements to efficiently destroy the starting state. Agents are allowed to place elements for five timesteps, after which the world is simulated forwards another 64 timesteps, and reward is calculated as the number of empty elements. A general strategy is to place fire on flammable structures, and place acid on other elements to dissolve them away. The Destroying task presents a task where correctly parsing the given observation is crucial.

- **Path-Building.** The Path-Building task presents a construction challenge in which agents must place or remove wall elements to route water into a goal container. An episode lasts 64 timesteps, and reward is calculated as the number of water elements in the goal. Water is continuously produced from a source formation of Cloner+Water elements. In the Path-Building challenge, agents must correctly place blocks such that water flows efficiently in the correct direction. Additionally, any obstacles present must be cleared or built around.

To learn to control in this environment, a Stable Baselines 3 PPO agent [29, 35] is trained over 1,000,000 environment interactions. The agent model is comprised of two convolutional layers with feature size 32 and 64 and kernel size of three, followed by two fully-connected layers. A learning rate of 0.0003 is used, along with a batchsize of 256. An off-the-shelf RL algorithm is intentionally chosen, so experiments can focus on the impact of training tasks.

Figure 5-5: **Increasing the complexity of RL training tasks helps generalization, up to a task-specific inflection point.** Shown are the test rewards of RL agents trained on tasks with increasing numbers of shapes (shown in log-scale). In Sand-Pushing, too much complexity will decrease test performance, as agents become unable to extract a sufficient reward signal. In Destroying, complexity consistently increases test performance. While increased complexity generally increases the difficulty of training tasks and reduces reward, in Path-Building certain obstacles can be used to complete the goal, improving training reward.

Figure 5-6 highlights agents solving the various RL tasks. Training tasks are generated using the same procedural generation algorithm as the world-modelling experiments. Task-specific structures are also placed, such as the goal slots in Sand-Pushing and Path-Building, and initial sand/water elements.

To test generalization, agents are evaluated on test tasks that are out of distribution from training. Specifically, test tasks are generated using a procedural generation algorithm that only places squares (5 for Destroying and Sand-Pushing, 10 for Path-Building). In contrast, the training tasks are generated using only lines and circles.

Figure 5-5 showcases how training task complexity affects generalization to test tasks. Displayed rewards are averaged from five independent training runs each. Agents are trained on tasks generated with increasing numbers of lines and circles (0, 1, 2, 4 ... 32, 64). These structures serve as obstacles, and training reward generally decreases as complexity increases. One exception is in Path-Building, as certain element structures can be useful in routing water to the goal.

Different RL tasks display a different response to training task complexity. In Sand-Pushing, it is helpful to increase complexity up to 8 shapes, but further complexity harms performance. This inflection point may correspond to the point where learning signal becomes too high-variance. RL is highly dependent on early reward

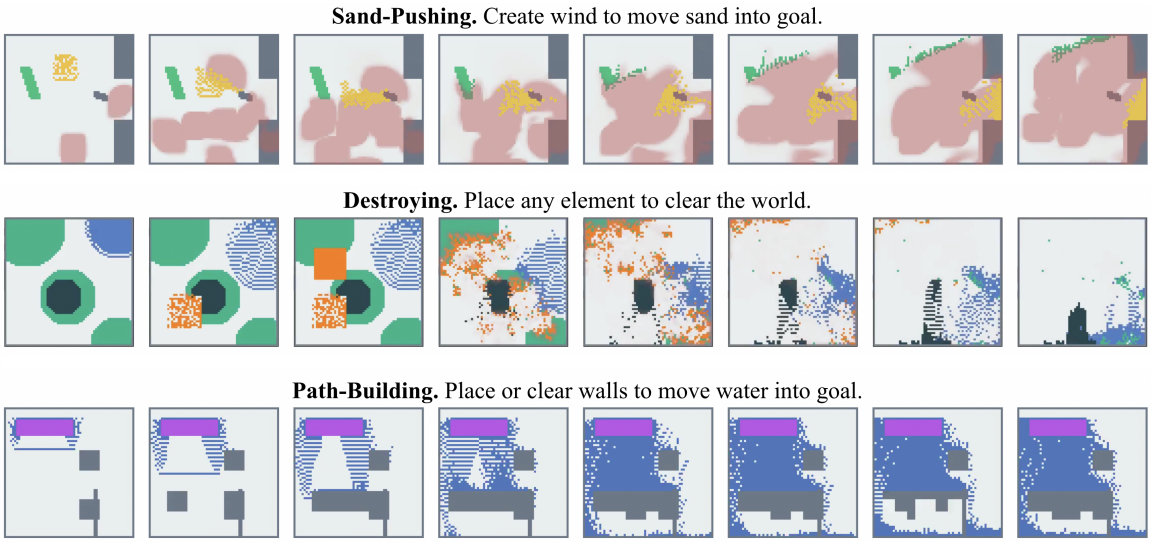

Figure 5-6: **Agents solving the Sand-Pushing, Destroying, and Path-Building tasks.** In the Sand-Pushing task, wind is used to push a block of sand elements between obstacles to reach the goal slot on the right. In Destroying, agents must place a limited number of elements to efficiently destroy the world. In Path-Building, agents must construct a path for water to flow from a source to a goal container. Tasks are randomly generated via a procedural algorithm.

signal to explore and continue to improve, and training tasks that are too complex can cause agent performance to suffer.

In contrast, agents on the Destroying and Path-Building task reliably gain a benefit from increased training task complexity. On the Destroying task, increased diversity during training may help agents recognize where to place fire/acid in test states. For Path-Building, training tasks with more shapes may present more possible strategies for reaching the goal.

The difference in how complexity affects training in Powderworld world-modelling and reinforcement learning tasks highlights a motivating platform for further investigation. While baseline RL methods may fail to scale with additional complexity and instead suffer due to variance, alternative learning techniques may better handle the learning problem and show higher generalization.

Chapter 6

Conclusion

Generalizing to novel unseen tasks is one of the grand challenges of reinforcement learning. Consistent lessons in deep learning show that *training data* is of crucial importance, which in the case of RL is training tasks. To study how and when agents generalize, the research community will benefit from more expressive foundation environments supporting many tasks arising from the same core rules.

This work introduced Powderworld, an expressive simulation environment that can generate both supervised and reinforcement learning task distributions. Powderworld's ruleset encourages modular interactions and emergent phenomena, resulting in world models which can accurately predict unseen states and even adapt to novel elemental behaviors. Experimental results show that increased task complexity helps in the supervised world-modelling setting and in certain RL scenarios. At times, complexity hampers the performance of a standard RL agent.

Powderworld is built to encourage future research endeavors, providing a rich yet computationally efficient backbone for defining tasks and challenges. The provided experiments hope to showcase how Powderworld can be used as a platform for examining task complexity and agent generalization. Future work may use Powderworld as an environment for studying open-ended agent learning, unsupervised environment design techniques, or other directions. As such, all code for Powderworld is released online in support of extensions.

Bibliography

[1] Physics simulation game: Powder game.

[2] The powder toy.

[3] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[5] Max Bittker. Making sandspiel.

[6] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, pages 2048–2056. PMLR, 2020.

[9] 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. *Advances in neural information processing systems*, 33:13049–13061, 2020.

[10] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *arXiv preprint arXiv:2206.08853*, 2022.

[11] William H Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noburu Kuno, Stephanie Milani, Sharada Prasanna Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. The minerl competition on sample efficient reinforcement learning using human priors. 2019.

[12] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. *Advances in neural information processing systems*, 31, 2018.

[13] Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

[14] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[15] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

[16] Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pages 4940–4950. PMLR, 2021.

[17] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *Ijcai*, pages 4246–4247. Citeseer, 2016.

[18] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *2016 IEEE conference on computational intelligence and games (CIG)*, pages 1–8. IEEE, 2016.

[19] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 95–101, 2020.

[20] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *arXiv preprint arXiv:2111.09794*, 2021.

[21] Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

[22] 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 preprint arXiv:1509.02971*, 2015.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[24] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

[25] Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. *arXiv preprint arXiv:2203.01302*, 2022.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[27] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[28] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021.

[29] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 2021.

[30] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021.

[31] Sebastian Risi and Julian Togelius. Increasing generality in machine learning through procedural content generation. *Nature Machine Intelligence*, 2(8):428–436, 2020.

[32] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on*

Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015.

[33] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[34] 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 preprint arXiv:2109.13202*, 2021.

[35] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[36] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents. *arXiv preprint arXiv:1903.00784*, 2019.

[37] Open Ended Learning Team, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, et al. Open-ended learning leads to generally capable agents. *arXiv preprint arXiv:2107.12808*, 2021.

[38] 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 *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[39] 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. *arXiv preprint arXiv:1901.01753*, 2019.

[40] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on robot learning*, pages 1094–1100. PMLR, 2020.