NOVA IMS Information Management School

MAA

Mestrado em Métodos Analíticos Avançados
Master Program in Advanced Analytics

Exploring Evolution Strategies for Reinforcement Learning in the Obstacle Tower Environment

Julian Kuypers

Dissertation presented as partial requirement for obtaining the Master's degree in Data Science & Advanced Analytics

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

Exploring Evolution Strategies for Reinforcement Learning in the Obstacle Tower Environment

by

Julian Kuypers

Dissertation presented as partial requirement for obtaining the Master's degree in Data Science & Advanced Analytics

Advisor / Co Advisor: Mauro Castelli

Co Advisor*:* Illya Bakurov

July 2021

ABSTRACT

In 2017 *OpenAI* demonstrated that it was possible to train an AI agent by using *Evolution Strategies* (ES), and that the results rivaled standard *Reinforcement Learning* (RL) techniques on modern benchmarks. Their research effectively showed that *Evolution Strategies* is a viable alternative to traditional *Reinforcement Learning* techniques, and that it bypasses many of Reinforcement Learning's inconveniences, notably the use of backpropagation.

The *Obstacle Tower* environment aims to set a new *Reinforcement Learning* benchmark by challenging *Artificial Intelligence* (AI) agents to traverse 3-Dimensional procedurally generated levels using a real-time 3-Dimensional physics system. The environment tests an agent's ability to generalize by requiring it to optimize aspects that are common in many *Reinforcement Learning* environments, but rarely combined in the same environment: vision, planning, and control.

In this research, the original implementation of *OpenAI's* Evolution Strategies algorithm was applied for the first time to the *Obstacle Tower* environment to assess how well it performs in a more complex environment, where the agent's generalization ability is critical. Additionally, in the interest of exploring Evolution Strategies in this environment, common Genetic Algorithm selection and mutation techniques were developed and applied to try and improve the performance of the original *Evolution Strategies* implementation. *Crossover* techniques were not explored during this research, as they are rarely applied in *Evolution Strategies*. The results show that although the basic implementation of *Evolution Strategies* does not perform well in the complex *Obstacle Tower* environment, it is possible to improve its performance by applying different evolution methods borrowed from *Genetic Algorithm* (GA), which are algorithms belonging to the same family as Evolution Strategies.

KEYWORDS

deep reinforcement learning; evolution strategies; genetic algorithm; obstacle Tower; obstacle Tower Challenge; unity; neuroevolution; reinforcement learning benchmark.

INDEX

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

2D/3D	2-Dimensional / 3-Dimensional
AI	Artificial Intelligence
CNN	Convolutional Neural Network
DRL	Deep Reinforcement Learning
ES	Evolutionary Strategies
FC	Fully Connected
FFCNN	Feed-Forward Convolutional Neural Network
GA	Genetic Algorithms
ML	Machine Learning
NES	Natural Evolution Strategies
OT	Obstacle Tower
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RNB	Rainbow (algorithm)

1. INTRODUCTION

From Atari games to *Doom* and *Dota 2*, Deep Reinforcement Learning (DRL) is considered the go-to technique to solve end-to-end learning from high-dimensional raw pixel images [Mnih et al., 2015]. The recent successes of *Google DeepMind*'s *AlphaStar* [Vinyals et al., 2019], where an agent has reached the human level of grandmaster in the real-time strategy game of *StarCraft II* much sooner than was initially anticipated, and *AlphaGo* [Silver et al. 2016], which beat Lee Sedol[1] in the game *Go* in 2015, have garnered a great deal of interest in DRL and many advances have been made in recent years to solve more complex environments.

Feed-forward Convolutional Neural Networks (FFCNN) using basic policy gradient methods or value based gradient methods have proven to be effective at solving 2-dimensional (2D) environments such as the ones found in the Atari games [Mnih et al., 2016].

The central goal of a *Reinforcement Learning* (RL) application is to learn a policy, which is a mapping from the state of the environment to a choice of action, that yields effective performance over time. In their 2017 paper, *OpenAI* proved it was possible to train an *Artificial Intelligence* (AI) agent by using *Evolutionary Strategies* (ES) [Salimans et al., 2016], a type of *Neuroevolution* algorithm[2], is a type of black-box optimization algorithm that relies on selecting and perturbing ("mutating") individuals as an optimization strategy. ES can be understood as a simplified version *Genetic Algorithms* (GA), where crossover and eventually elitism are omitted. With this paper, they have demonstrated that it is possible to learn a policy using ES, and that this policy achieves results comparable to those of traditional RL

[1] Lee Sedol is a former South Korean professional Go player, at the time of playing agains AlphaGo, he was ranked second in the international rankings.

[2] Neuroevolution is a machine learning technique that applies evolutionary algorithms to construct artificial neural networks, taking inspiration from the evolution of biological nervous systems in nature.

techniques while overcoming many of RL's inconveniences, dispelling the common belief that ES methods are impossible to apply to high dimensional problems [Schaul et al., 2011].

In 2019, game development company Unity Technologies created the *Obstacle Tower* (OT) environment; a procedurally generated RL environment that combines platforming-style gameplay with puzzles and was designed to be a new benchmark for learning agents in the areas of computer vision, locomotion skills, high-level planning, and generalization [Juliani et al., 2019].

In this research, we applied the original implementation of the ES method used by *OpenAI* to the Obstacle Tower environment for the first time to assess how well it performs in a more complex environment, where the agent's generalization ability is critical. Additionally, we explored some GA selection and mutation techniques to see if we can improve the performance of the original ES implementation. Crossover techniques were not explored during this research as they are more difficult to implement when working in distributed systems, and because the complexity of applying crossover techniques to neural network weights falls outside of the scope of this research. Furthermore, they were not implemented in *OpenAI's* 2017 research.

The agents in this study are represented by a small *Convolutional Neural Network* (CNN) with two convolution layers, no pooling layers, and no backpropagation. Two selection methods had been implemented; *Select N Best*, and the popular *Fitness Proportionate*/*Roulette Wheel Selection*. In addition to the mutation technique implemented by *OpenAI*, where small amounts of Gaussian noise are added to the parameters of the Neural Network, two more techniques were implemented, *random mutation* and *decaying mutation*.

Multiple experiments were run to evaluate ES's performance, and to assess whether the GA selection and mutation techniques were able to improve on that performance. The

results of the experiments of this research are compared to the OT benchmark results obtained by Juliani et al. [16] in their paper introducing the environment.

2. LITERATURE REVIEW

2.1 Reinforcement Learning

Reinforcement Learning refers to the study of how an agent can interact with its environment to learn a policy which maximizes expected cumulative rewards for a task. It is one of three basic learning paradigms in Machine Learning (ML) alongside *supervised learning*, where training data is labelled, and *unsupervised learning*, where data is not labelled, and the model finds hidden patterns in the data. The history of RL is a vast and long, with artificial applications of *trial-and-error* learning dating back to the 1950s and 1960s [Minsky, 1961]. Recently, RL has undergone a renaissance due to promising results in areas such as playing Go [Silver et al. 2016], beating competitive players in video games like StarCraft II [Vinyals et al., 2019], controlling continuous robotics systems [Lillicrap et al. 2015], and autonomous driving [Stavens et al., 2012]. The abundance and accessibility of learning environments from libraries like *Gym* [Brockman et al., 2016] and the Arcade Learning Environment [Bellemare et al., 2013; Mnih et al. 2013] have further fueled this growth in attention and development. While many policies, model architectures, reward systems, and learning strategies have been developed over the course of RL's history, there are five constant essential elements to each RL problem:

> **The Agent.** The program controlling the object of concern [41] (e.g. a robot, a video game character).
>
> **The Environment.** This defines the outside world programmatically. Everything the agent(s) interacts with is part of the environment. It's built for the agent to make it seem like a real-world case. It's needed to prove the performance of an agent, meaning if it will do well once implemented in a real-world application [41].

The Action. A move made by the agent, which causes a status change in the environment.

The Reward(s). The evaluation of an action, which can be positive or negative. The reward R_t is a scalar feedback signal which indicates how well the agent is doing at step time *t*.

The Policy. The algorithm used by the agent to decide its actions. This can be seen as the strategy to accumulate the most rewards over time that the agent learns during training.

There are two broad classes for RL methods, each defined by how the model performs its optimization: *model-based* and *model-free*. *Model-based* RL uses experience to construct an internal model of the transitions and immediate outcomes in the environment. Appropriate actions are then chosen by searching or planning in this world model [41]. This is a statistically efficient way to use experience, as each morsel of information from the environment can be stored in a statistically faithful and computationally manipulable way. *Model-free* RL, on the other hand, uses experience to learn directly one or both of two simpler quantities (state/ action values or policies) which can achieve the same optimal behaviour but without estimation or use of a world model [Dayan et al., 2008]. We can think of *model-free* algorithms as *trial-and-error* methods; the agent explores the environment and learns from outcomes of the actions directly, without constructing an internal mode. Both RL classes have benefits and drawbacks, and the performance of each class on different RL problems and environments is still an active field of research in RL [Brunnbauer et al., 2021]. Appendix 1. presents *OpenAI*'s non-exhaustive, but useful taxonomy of modern RL algorithms.

2.2 Evolution Strategies

Evolution Strategies is a class of black box optimization algorithms that are heuristic search procedures inspired by natural evolution: At every iteration ("generation"), a population of parameter vectors (genotypes" or "chromosomes"), is perturbed ("mutated") and their objective function value ("fitness") is evaluated. The highest scoring parameter vectors are then recombined to form the population for the next generation, and this procedure is iterated until the objective is fully optimized [Wierstra et al., 2008].

ES is relatively easy to implement and scale; Running on a computing cluster of 80 machines and 1,440 CPU cores, *OpenAI's* implementation was able to train a 3D *MuJoCo*[3] humanoid walker in only 10 minutes (*A3C*[4] on 32 cores takes about 10 hours). Using 720 cores they could also obtain comparable performance to *A3C* on Atari while cutting down the training time from 1 day to 1 hour [Salimans et al., 2017].

Salimans et al. [28] highlights multiple advantages that ES enjoys over RL algorithms:

> **No need for backpropagation**. ES only requires the forward pass of the policy and does not require backpropagation (or value function estimation), which makes the code shorter and between 2-3 times faster in practice. On memory-constrained systems, it is also not necessary to keep a record of the episodes for a later update. There is also no need to worry about exploding gradients in RNNs. Lastly, one can explore a much larger function class of policies, including networks that are not differentiable (such as

[3] MuJoCo is a game where the goal is to make three-dimensional bipedal robot walk forward as fast as possible, without falling over. The environment is available in *OpenAI*'s *Gym* toolkit.

[4] Asynchronous Advantage Actor Critic (A3C) is a RL algorithm that uses asynchronous gradient descent for optimization of deep neural network controllers. It was presented by Mnih et al. in their 2016 paper "*Asynchronous Methods for Deep Reinforcement Learning*", where it surpassed the state-of-the-art algorithms in 2D and 3D environments. https://arxiv.org/abs/1602.01783

in binary networks), or ones that include complex modules (e.g. pathfinding, or various optimization layers).

Highly parallelizable. ES only requires workers to communicate a few scalars between each other, while in RL it is necessary to synchronize entire parameter vectors (which can be millions of numbers). As a result, they observed linear speedups in their experiments as we added on the order of thousands of CPU cores to the optimization.

Higher robustness. Several hyperparameters that are difficult to set in RL implementations are side-stepped in ES. For example, RL is not "scale-free", so one can achieve very different learning outcomes (including a complete failure) with different settings of the frame-skip hyperparameter in Atari [Braylan et al., 2005]. They showed in their work that ES works about equally well with any frame-skip.

Structured exploration. Some RL algorithms (especially policy gradients) initialize with random policies, which often manifests as random jitter on spot for a long time. This effect is mitigated in *Q-Learning* due to *epsilon-greedy policies*[5], where the max operation can cause the agents to perform some consistent action for a while (e.g. holding down a left arrow). This is more likely to do something in a game than if the agent jitters on spot, as is the case with policy gradients. Similar to *Q-learning*, ES does not suffer from these problems because one can use deterministic policies and achieve consistent exploration.

Credit assignment over long time scales. By studying both ES and RL gradient estimators mathematically one can see that ES is an attractive choice especially when

[5] Q-Learning, short for Quality-Learning, is an off-policy (and model-free) algorithm, meaning it approximates the optimal action-value function, independent of the policy. During training, Q-learning algorithms build and populate a table ("Q-table") storing state-action pairs, the algorithm then consults this table to estimate the best possible action during inference. As a result, Q-learning due to greedy action selection, the algorithm usually selects the next action with the best estimated reward, which can prevent it from taking the longer, more rewarding path, making it a short-sighted algorithm.

the number of time steps in an episode is long, where actions have long lasting effects, or if no good value function estimates are available.

Conversely, they also found some challenges to applying ES in practice. One core problem is that for ES to work, adding noise in parameters must lead to different outcomes to obtain some gradient signal. As they elaborate on in their paper, the use of *virtual batch normalization*[6] [Salimans et al., 2016] can help alleviate this problem, but further work on effectively parameterizing neural networks to have variable behaviors as a function of noise is necessary. As an example of a related difficulty, they found that in Atari's *Montezuma's Revenge*[7], one is very unlikely to get the key in the first level with a random network, while this is occasionally possible with random actions.

2.3 The Obstacle Tower Environment

The Obstacle Tower environment is a procedurally generated RL environment, meaning the environment is generated with some randomness and is never the same twice. It was developed by game development platform Unity and uses the ML-Agents Toolkit [Juliani et al., 2019]. It can run on the Mac, Windows, and Linux platforms, and can be controlled via the

[6] Virtual Batch normalization (VBN), also known as virtual batch norm) is a method used to make artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling. It is similar to batch normalization, except that each example is normalized based on the statistics collected on a reference batch of examples that are chosen once and fixed at the start of training, and on itself, instead of the statistics of several examples belonging to the same mini-batch.

[7] Montezuma's Revenge is an Atari game available in the Arcade Learning Environment that is notoriously difficult to beat with RL due to its sparsely distributed rewards. See Annexes 1. and 2. For DeepMind and OpenAI's models' performance on the game.

OpenAI Gym interface for easy integration with existing DRL training frameworks [Brockman et al., 2016].

Figure 1. Examples of agent's perspective in the Obstacle Tower environment.

Figure 2. Examples of floor layouts from the Obstacle Tower environment.

Episode Dynamics

The Obstacle Tower environment consists of up to 100 floors, with the agent starting on floor zero. All floors of the environment are treated as a single finite episode in the RL context. Each floor contains at least a starting and ending room. Each room can contain a puzzle to solve, enemies to defeat, obstacles to evade, or a key to open a locked door. The layout of the floors and the contents of the rooms within each floor becomes more complex at higher floors in the Obstacle Tower, providing a natural curriculum for learning agents. Within an episode, it is only possible for the agent to go to higher floors of the environment, and not to return to lower floors. The episode terminates when the agent collides with a hazard such as a pit or enemy, when the timer runs out, or when the agent arrives at the top floor of the environment. The timer is set at the beginning of the episode and completing floors as well as

collecting blue time orbs increase the time left to the agent. In this way a successful agent must learn a behavior which is a tradeoff between collecting orbs and quickly completing floors of the tower in order to arrive at the higher floors before the timer ends. The reward function is a mix of sparse and dense rewards; with sparse rewards, the rewards in the environment are sparsely distributed, meaning that there are only a few states in the state space that return a reward, whereas with dense rewards, every transition in the environment is associated with a reward, which can be positive or negative. In the OT environment, the agent is given a positive reward of +1 for completing a floor of the tower (sparse), and a reward of +0.1 for opening a door, solving puzzles, or picking up a key (dense). The environment's authors describe this reward function as a mix between sparse and dense reward [Juliani et al., 2019], but in reality, they are two different sparse rewards, as the positive reward +0.1 is not associated with every change of state, but rather with a change of state involving a specific, sparsely distributed, state.

Observation Space

The observation space of the agent consists of two types of information. The first type of observation is a rendered pixel image of the environment from a third person perspective. This image is rendered in 168 × 168 RGB and can be downscaled to 84 × 84. The second type of observation is a vector of auxiliary variables which describe relevant, non-visual information about the state of the environment. The elements which make up this auxiliary vector are the number of keys agent is in possession of, as well as the time left in the episode.

Action Space

The action space of the agent is multi-discrete, meaning that it consists of a set of smaller discrete action spaces, of which the union corresponds to a single action in the environment. These subspaces are as follows: forward/backward/no-operation movement, left/right/no-operation movement, clockwise/counterclockwise rotation of the camera/no-operation, and no-operation/jump. They also provide a version of the environment with this action space flattened into a single choice between one of 54 possible actions, whose size corresponds to the product of the sizes of all the sub-spaces in the multi-discrete case.

The Obstacle Tower Environment as a Benchmark

Obstacle Tower was developed specifically to overcome the limitations of previous game-based AI benchmarks, offering a broad and deep challenge, the solving of which would imply a major advancement in reinforcement learning. In brief, the features of Obstacle Tower outline by Juliani et al. [16] are:

> **High visual fidelity.** The environment is rendered in 3D using real-time lighting and shadows, along with much more detailed textures and models than previous benchmarks. See *Fig 1*. for examples of the agent's perspective.
>
> **Procedurally generated floors and rooms**. Navigating the game requires both dexterity and planning, and the floors within the environment are procedurally generated, making generalization a requirement to perform well during evaluation. See *Fig 2*. for examples of floor layouts of various levels of the Obstacle Tower.
>
> **Physics-driven interactions**. The movement of the agent and other objects within the environment are controlled by a real-time 3D physics system.

Procedurally generated visuals. There are multiple levels of variation in the environment, including the textures, lighting conditions, and object geometry. Therefore, agents must be able to generalize their understanding of objects' appearance.

The Obstacle Tower environment is designed to provide a meaningful challenge to current and future AI agents, specifically those trained using the pixels-to-control approach. Juliani et al. [16] describes four axes of challenge that this environment provides: vision, control, planning, and generalization. While various other environments and benchmarks have been used to provide difficult challenges for AI agents, this is to the best of our knowledge the first benchmark which combines all such axes of complexity.

Vision. The primary observation available to agents within the Obstacle Tower is a rendered RGB image. Obstacle Tower contains high-fidelity real-time lighting, complex 3D shapes, and high-resolution textures. Furthermore, the floors in the environment are rendered in one of multiple different visual themes, such as *Ancient* or *Industrial*. These visual themes were chosen to provide a large amount of variation in the textures, colors, and 3D models that the agent would encounter. With the combination of high-fidelity visuals and increased visual variation, models with much greater representational capacity than those used in A3C [Mnih et al., 2016] or DQN [Mnih et al., 2015] will be needed to perform well in the environment.

Generalization & Vision. Humans can easily understand that two different doors seen under different lighting conditions are still doors; general-purpose agents should have similar abilities. However, this is not the case; in many cases agents trained under one set of visual conditions, and then tested on even a slightly different visual conditions perform much worse at the same task [Huang et al., 2017]. The procedural lighting and

visual appearance of floors within the Obstacle Tower means that agents will need to be able to generalize to new visual appearances which they may never have directly experienced before.

Control. An agent in Obstacle Tower must be able to navigate through multiple rooms and floors. Each of these rooms can contain multiple possible obstacles, enemies, and moving platforms, all of which require fine-tuned control over the agent's movement. Floors of the environment can also contain puzzle rooms, which involve the physical manipulation of objects within the room to unlock doors to other rooms on the floor. While the action space of the agent is discrete, the environment itself uses continuous metrics for the position and velocity of objects, making the state space extremely large. We expect that for agents to perform well on these sub-tasks, the ability to model and predict the results of the agents' actions within the environment will be of benefit.

Generalization & Control. The layout of the rooms on every floor are different on each instance of the Obstacle Tower; as such, it is expected that methods which are designed to exploit determinism of the training environment, such as Brute [Machado et al., 2017] and Go-Explore [Ecoffet, 2018] will perform poorly on the test set of environments. It is also the case that within a single instance of a Tower, there are elements of the environment which contain stochastic behavior, such as the movement of platforms and enemies.

Planning. Depending on the difficulty of the floor, some floors of the Obstacle Tower require reasoning over multiple dependencies to arrive at the end room. For example, some rooms cannot be accessed without a key that can only be obtained in rooms sometimes very far from the door they open. In these cases, planning is required to ensure the agent takes the most efficient path between rooms.

Generalization & Planning. Due to the procedural generation of each floor layout within the Obstacle Tower, it is not possible to re-use a single high-level plan between floors. It is likewise not possible to re-use plans between environment instances, as the layout of each floor is determined by the environment's generation seed. Because of this, planning methods which require computationally expensive state discovery phases are likely not able to generalize to unseen floor layouts.

3. METHODOLOGY

3.1 Neural Network

In DRL, the AI agents are represented by Deep Neural Networks. While there are many suitable network architectures to choose from, ranging from basic to very complex, we have opted for a feed-forward Convolutional Neural Network (FFCNN); these networks have a proven track record for automatically detecting significant features without any human supervision [Alzubaidi et al., 2021] and performing well in pixel-to-control problems [Mnih et al., 2016]. Given the agent's dependance on visual information in the Obstacle Tower environment, the same model architecture as the one used by Salimans et al. [28] and Mnih et al. [25] was adopted for this research.

A Convolutional Neural Network (CNN) is a kind of neural network that is able to extract features from data with convolution structures. Often used to solve image processing, classification, and segmentation problems, CNNs use convolution layers to preserve the relationship between pixels by learning image features using small squares of input data. The outputs of the convolution layers summarize the presence of features in the input image, also called feature maps. Each convolutional layer contains a series of filters known as convolutional kernels. The filter is a matrix of integers that are used on a subset of the input pixel values, the same size as the kernel. Each pixel is multiplied by the corresponding value in the kernel, then the result is summed up for a single value for simplicity representing a grid cell, like a pixel, in the output channel/feature map. This architecture gives CNNs advantages over more classical neural networks; neurons are only connected to a small number of neurons from the previous layer instead of all of them which effectively reduces the parameters and speeds up convergence, and a group of connections can share the same

weights, which reduced parameters further [Li et al., 2020]. A problem with the convolution layer outputs, or feature maps, is that they can be sensitive to the location of the features in the input (e.g. an object being on the right or left side of the image). One approach to mitigate this sensitivity is to down sample the feature maps in order to make them more robust to changes in the position of the feature in the image. This robustness to positional variations is referred to as *translation invariance* and means that the CNN would be able to identify an object regardless of its position in the image (Appendix 2.). *Pooling layers,* commonly placed in-between successive convolution layers, provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. In addition to progressively reducing the representation size of the input images, *Pooling layers* also reduce the number of parameters, and hence also control *overfitting*[8]. Two common pooling methods are *Average pooling* and *Max pooling* that summarize the average presence of a feature and the most activated presence of a feature respectively. As mentioned previously, each neuron only receives input from a small local region, or *kernel*, of the pixels in the input image, unlike a neural network where all the neurons are fully connected. This is the concept of *local connectivity,* which helps us understand another powerful mechanism that allow CNNs to reduce the number of parameters, and in turn increase *translation invariance* [Gu et al., 2017]: *Parameter Sharing* (or *weight sharing*). With *Parameter Sharing*, all the neurons in the *kernel* are constrained to use the same weights and biases. It stems from the assumption that if one feature is useful to compute at some position (e.g. *(x, y)*), then it should be useful to compute at a different position (e.g *(x2, y2)*).

[8] *Overfitting* occurs when a machine learning model achieves a good fit (i.e good predictions) on the training data but does not generalize well on unseen data. In other words, the model has learned patterns that are specific to the training data and not relevant to the dataset as whole.

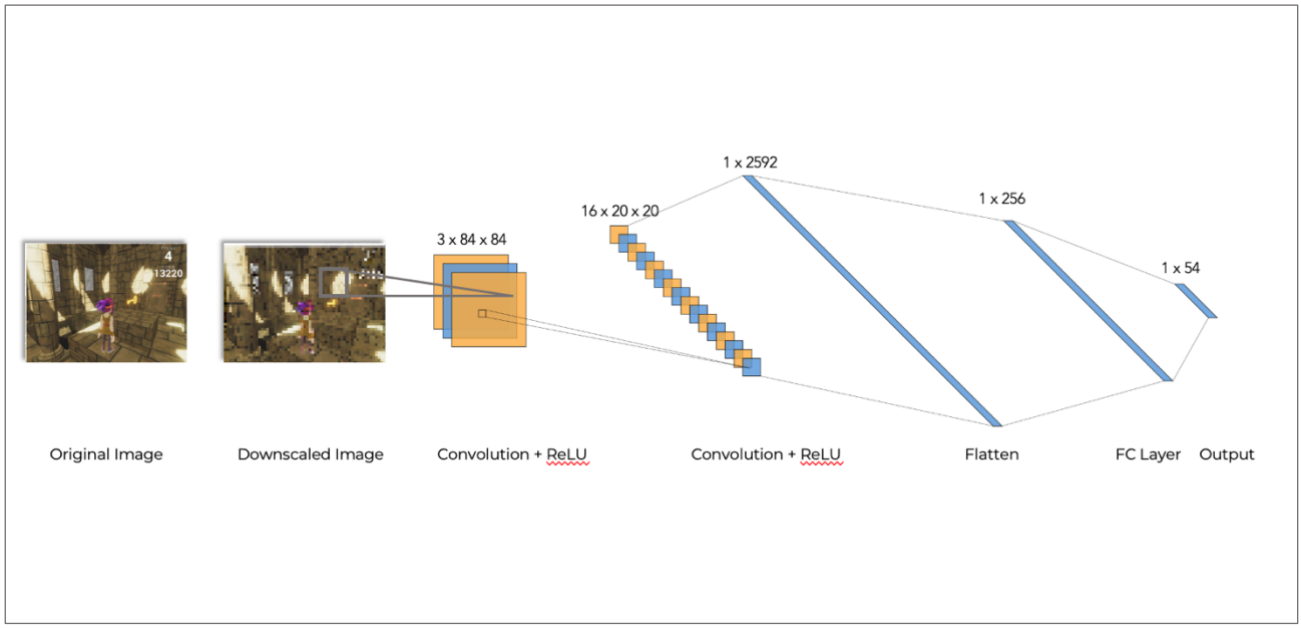


Figure 3. Convolutional Neural Network Architecture Diagram

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 20, 20]	3,088
ReLU-2	[-1, 16, 20, 20]	0
Conv2d-3	[-1, 32, 9, 9]	8,224
ReLU-4	[-1, 32, 9, 9]	0
Flatten-5	[-1, 2592]	0
Linear-6	[-1, 256]	663,808
ReLU-7	[-1, 256]	0
Linear-8	[-1, 54]	13,878

Total params: 688,998
Trainable params: 688,998
Non-trainable params: 0

Table 1. Parameters of the Convolutional Neural Network used for this research.

The CNN used for this research project was developed in PyTorch and is comprised of two convolutional layers; the first convolution layer has an input size of 3 and an output size of 16 with a kernel size of 8 and a stride of 4. It expects an input image of size 84 x 84, which is the size of the downscaled image from the Obstacle Tower. The second convolution layer has an input size of 16 and an output size of 32 with a kernel size of 4 and a stride of 2. Between

each convolutional layer and all of the subsequent fully connected layers we apply a *Rectified Linear Unit* activation function (*ReLU*) to the inputs; *ReLU* will output the input directly if it is positive, otherwise, it will output zero. The outputs of the second convolutional layer are flattened into a fully connected layer with dimensions 1 x 2592 and passed through a fully connected layer with dimensions 1 x 256. Finally, the outputs of the last activation function are sent to the output layer; a fully context layer with the number of possible outputs equal to the number of possible actions, in the case 54. Figure 3. shows the *CNN* architecture diagram for the model used in this research, and Table 1. shows the parameters of the *CNN*. An alternative diagram, following the *AlexNet* style, can be found in Appendix 3. This *CNN* implementation was used throughout all experiments, with no changes applied to it.

Pooling layers, which are often used in *CNNs* to down sample feature maps by summarizing the presence of features in patches of the feature map, or to pool the activations over the entire activation map in the case of *Global Pooling Layer*s [Christlein et al., 2019], were no included in this research's *CNN* architecture for the following reasons: the goal of this research is to explore the performance of ES in a complex environment. As such, it makes sense that the baseline performance of ES should be evaluated using a rudimentary implementation of a *CNN*, one without the addition of the *CNN* architecture components that serve to improve the model's performance. Furthermore, it has been shown that max pooling layers can be replaced by a convolutional layer with increased stride without loss in accuracy on several image recognition benchmarks [Springenberg et al., 2015], and that *CNNs* without pooling layers after the convolution layer can achieve an accuracy equivalent that that of *GoogleNet*, *VGG16*, and *AlexNet* [Howard et al., 2017].

3.2 Distributed System

Name	Description
futures = **f.remote**(*args*)	Execute function *f* remotely. **f.remote**() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
objects = **ray.get**(*futures*)	Return the values associated with one or more futures. This is blocking.
ready_futures = **ray.wait**(*futures, k, timeout*)	Return the futures whose corresponding tasks have completed as soon as either *k* have completed or the timeout expires.
actor = **Class.remote**(*args*) *futures* = *actor*.**method.remote**(*args*)	Instantiate class *Class* as a remote actor and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

Table 2. Ray project API

One of the advantages of ES over traditional RL techniques is the ability to parallelize the training of the algorithm. To take advantage of this parallelization ability, a distributed system needed to be developed; *Ray*, a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications, was used to perform the experiments of this research. *Ray* runs on top of a Kubernetes cluster, where it models an application as a graph of dependent tasks that evolves during execution. There are two important *Ray* concepts to understand before discussing the framework's benefits and the implementation of the distributed system: Tasks and Actors. A **Task** represents the execution of a remote function on a stateless[9] worker. When a remote function is invoked, a future representing the result of the task is returned immediately. Futures can be retrieved using ***ray.get()*** and passed as arguments into other remote functions without waiting for their result. This allows the user to express parallelism while capturing data dependencies. Table 2 shows how *Ray* implements functions for its tasks and actors in its API. Remote functions operate on immutable objects and are expected to be stateless and side-effect free: their outputs are determined solely by

[9] A stateless server/worker does not store data on the host. It processes requests based only on information relayed with each request and doesn't rely on information from earlier requests.

their inputs. This implies idempotence, which simplifies fault tolerance through function re-execution on failure [Moritz et al., 2018]. An **Actor** represents a stateful[10] computation. Each actor exposes methods that can be invoked remotely and are executed serially. A method execution is similar to a task, in that it executes remotely and returns a future, but differs in that it executes on a stateful worker. A handle to an actor can be passed to other actors or tasks, making it possible for them to invoke methods on that actor. Appendix 4. presents some of the trade-offs between Tasks and Actors presented by Moritz et al. [26].

For this research, the distributed system was built using a *Kubernetes* cluster and *RLlib*, a distributed-RL library that is part of the open-source *Ray* project. *RLlib* provides scalable and reusable software primitives for RL and aims to standardize the training of RL algorithms that tend to otherwise have highly irregular computation patterns [Liang et al., 2018]. It is important to note that at the time of this research, *RLlib* does not have any stable or highly configurable components for ES and that custom components were developed for this research. These components will be discussed in a later paragraph. The distributed system was deployed Google Cloud Platform (GCP), where the cluster nodes ran on GCP's High-CPU machines, specifically *n1-highcpu-96* machines, which are ideal for tasks that require a moderate increase of vCPUs relative to memory, making them well suited for ES. The *n1-highcpu-96* machine has 96 vCPUs and 86.4 Gb of memory.

While most RL environments available in *OpenAI's* Gym library can run in headless mode, meaning they do not require a computer monitor to work, the Obstacle Tower environment does not. To be able to run the Obstacle Tower environment on the cluster, additional libraries needed to be installed on the head and worker nodes of the cluster, which

[10] A stateful server requires some type of storage capacity. Requests are processed and stored, so that the server can keep the state.

were running on headless servers; *Xvfb*, a Linux package used to create display servers, was used to be able to run the Obstacle Tower environment on the head and worker nodes. The cluster configuration file can be found in Appendix 5.

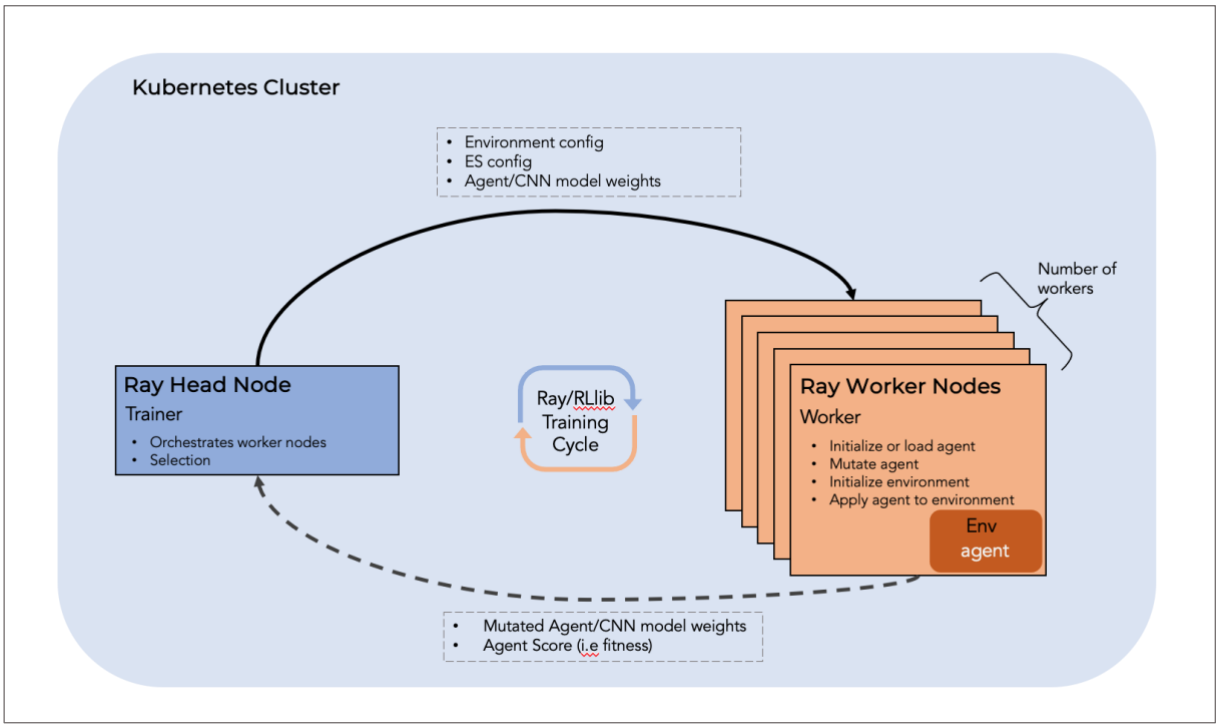


Figure 4. RLlib Training Cycle.

RLlib has released components for many state-of-the-art RL algorithms, including ES. However, their ES components are still a work in progress and were not suited for the OT environment, which behaves slightly differently to the standardized *OpenAI Gym* environments. Furthermore, their ES components didn't allow for different selection and mutation techniques. As such, custom components were developed for this research, notably as custom *Trainer* class to be hosted on the *Actor*, or Head, node; this class orchestrates the workers, or *Ray* Tasks, and performs the ES selection based on the results returned by the Tasks/Workers. This class also evaluates the Workers to find the elite of each step, or generation. A custom *Worker* class was also developed for this research; this class initializes

and *xvfb* server so that the OT environment can run in headless mode, starts the OT environment, loads the agent, could performs the mutation method specified for the experiment, and evaluates how well the agent performs in the environment. It then returns the results back to the Trainer class/Actor so that it can perform the selection and build the population for the next generation. The interactions between the Trainer and Worker classes for each generation, or training step, are show in Figure 4. Code snippets for the custom Trainer and Worker class can be found in Appendices 6. and 7.

3.3 Evolution Strategies

The version of ES that was used by *OpenAI* belongs to the class of *Natural Evolution Strategies* (NES) [Wierstra et al., 2008], and has been replicated for this research. Additionally, some common selection and mutation techniques borrowed from *Genetic Algorithms* (GA), another class of Evolutionary Algorithms, have been implemented to try and improve ES's performance.

Like ES, GAs are a search heuristic that are inspired by Charles Darwin's theory of natural evolution. These algorithms reflect the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring for the next generation. Though they are difficult to distinguish, the main discernable difference between GAs and ES is that GA applies crossovers between individuals in their population to share genetic information between them, while ES mainly focusses on selecting individuals and applying mutations to them, though it seems the terms can almost be used interchangeable nowadays. A basic genetic algorithm would be going through the following five phases:

Initialize a population. The process begins by creating a set of individuals, called a population, that each represent a solution to the problem you are trying to solve. The

population has a fixed size, and an individual solution is characterized by a set of parameters, called genes, which are joined to form a chromosome. In the context of GAs, an individual weight of the neural network would be considered a gene, and the ensemble of all the parameters of the network would be considered the chromosome.

Evaluate the fitness. The fitness of each solution in the population is calculated using a fitness function. Here we determine how well an individual solves the problem at hand. In the Obstacle Tower environment, the fitness of the individual is measured by the maximum number of floors the agent can reach.

Selection. Individuals are selected for variation based on their fitness scores. Generally, individuals with a higher fitness have a bigger chance of being selected for variation, although some selection methods allow for "weaker" individuals to be selected as well.

Crossover. Mimicking the natural sex reproduction mechanism, the intuition behind crossover is that by combining certain traits from two or more already fit individuals, the offspring will be even "fitter" since it will possibly inherit the best traits from each of its parents [Câmara, 2015]. The intention is to create a larger offspring pool of ideally greater fitness and to converge towards an optimum. Crossovers are rarely found in ES, where the tendency is to focus on selection and mutation mechanism but is an integral part of GAs.

Mutation. For some of the offspring, mutations can occur, where small variations such as adding noise are applied to some or all or some of their genes. The main goal of mutation is to maintain diversity in the population and to provide a mechanism for escaping a local optimum, thus avoiding early convergence.

Repeat steps 2. through 5. until the stopping criteria is met. These steps are repeated until the population has converged, or a stopping criterion has been met; for example, a desired fitness score has been reached, or a maximum number of generations has been reached.

There is a concept in GAs called *Elitism*, where the most fit individual in a population gets a guaranteed place in the next generation, generally without undergoing mutation (it can still be duplicated in the next population and undergo mutation). Although Elitism is not traditionally used in ES, but rather in GA, it was implemented in this research.

Pseudocode

```
START
Initialize a population
REPEAT:
        Fitness Evaluation + Save Elite
        Selection
        Crossover (optional for ES)
        Mutation
UNTIL specified stopping criteria has been met
STOP
```

3.4 The Obstacle Tower Environment

The open-source OT environment is available for download[11] for Linux, Mac, and Windows. The environment version used for this research is *v3.1*, and requires *Python +3.6*, the *Unity ML-Agents +0.10* and the *OpenAI Gym* Python packages. Although changes were made to the environment to be able to reduce the action space, these changes were not applied used during the experiments due to time and budget constraints. The following action space reductions were inspired by Kanervisto et al. [18]:

[11] https://github.com/Unity-Technologies/obstacle-tower-env

Full. All actions are possible (i.e. 54 possible discrete actions).

Minimal. The only allowed actions are moving forward, turning left, and turning right.

Backward. Same as *Minimal*, but with the additional option of moving backward.

Strafe. Same as *Backward*, but with the additional options of strafing (i.e moving sideways) left and right.

Always Forward. The player always moves forward, all other actions are possible except staying in place and going backward.

As a reminder, these reduced action space options were implemented in the OT environment but were not tested during the experiments. All experiments are run with the Full action space, which is the original action space provided by the Juliani et al. [16] and the one they used in their benchmarks.

No other parameters or functionalities were added to the OT environment, and the environment parameters that were used for all experiments are detailed in Table 3. under the *Experiments* section of this paper.

In order to reproduce the experiments of this research, it is important to be aware of the evaluation criteria outlined by the Juliani et al. [16]. They provide thee possible evaluation schemes that aim at making the performance on agents in the OT environment as reproducible and interpretable as possible:

No Generalization. It is possible to evaluate the performance of an agent on a single, fixed version of the OT. In this case they recommend explicitly reporting that the evaluation was performed on a fixed version of the OT, and reporting performance on five random seeds of the dynamics of the agent. These seeds can be provided on

environment reset and condition the random number generator used to generate the tower definition.

Weak Generalization. Agents should be trained on a fixed set of 100 seeds for the environment configurations. They should then be tested on a held-out set of five randomly selected tower configuration seeds not in the training set. Each should be evaluated five times using different random seeds for the dynamics of the agent (initial weights of the policy and/or value network(s)).

Strong Generalization. In addition to the requirements for weak generalization, agents should be tested on a held-out visual theme which is separate from the ones on which it was trained. In their paper, Juliani et al. [16] trained on the *Ancient* and *Moorish* themes, and test on the *Industrial* theme.

Because the OT environment is designed to test the generalization ability of agents, Juliani et al. [16] recommend evaluating agents using the latter two methods. For this research, all experiments were evaluated using the *Weak Generalization* scheme and the results will only be compared to the *Weak Generalization* benchmark results published by Juliani et al. [16].

4. EXPERIMENTS

Parameter	Program	Value
Generations – The maximum number of generations, in RL terms this would be the training step. This is the stopping criteria of the algorithm; when the maximum generation is reached, the algorithm stops training.	ES	85
Population Size - The number of individuals, or agents, in the population. A population of the specified size is initialized at the beginning of training, and the population size remains constant throughout the training process.	ES	200
Number of selected individuals – The number of individuals, or agents, that are selected to recreate the population at each generation. As the population is constant, some of the selected individuals may appear twice in the population (except during the initialization of the population at the start of training, where all individuals are unique).	ES	30
Maximum Timesteps per Episode – The maximum number of steps an agent is allowed to take during an episode, or generation. It is possible for the agent to exit the environment earlier if another exit condition is met (e.g., the agent won before using its maximum timesteps).	ES	3500
Mutation Power – The amount of perturbation applied to the parameter during mutation.	ES	0.005
Mutation Probability – The probability of mutation occurring. This is evaluated at each generation for each individual, or agent, before it enters the OT environment.	ES	0.6
Number of Workers – Number of Kubernetes nodes, or CPUs, available to Worker tasks.	ES	94
Greyscale – Turn images into greyscale (1 x 128 x 128) instead of RGB (3 x 128 x 128).	OT	False
Retro – Downscale images from 128 x 128 to 84 x 84.	OT	True
Realtime Mode – Show the agent traversing the environment in real-time speed.	OT	True

Table 3. Parameters used during experiments.

The focus of this work is on selection and mutation techniques. Crossover techniques were not explored during this research, as they are rarely applied in ES and are more of GA technique [Dianati et al., 2002], and were not explored in the original paper published by Salimans et al. [28] either. Furthermore, as crossover techniques are more difficult to implement when working in distributed systems, and because the complexity of applying crossover techniques to neural network weights falls outside of the scope of this research.

All experiments used the parameters specified in Table 3, except for the *mutation power*, which could change due to the nature of the mutation techniques that were being used.

Selection Methods

Two selection methods were implemented and tested for this research; the first is the basic ES selection used by *OpenAI*, and the second is a popular GA selection method:

Select N Best Individuals. In this selection method, individuals were ranked according to their fitness scores. The *N* highest ranking individuals were selected to re-create the population, where *N* is an exogenous parameter representing the number of selected individuals.

Fitness Proportionate Selection. Also called *Roulette Wheel Selection*, is a stochastic selection method, where the probability for selection of an individual is proportional to its fitness. This selection method gets its name from real-world roulette wheels that can be found in casinos, the only distinction is that the slots are not all the same size; instead, their size, and thus their probability of being selected, is proportional to their fitness score. This means that any individual has a chance of being selected, with fitter individuals having a higher probability of being selected.

Mutations

All mutations had the following parameters: a starting mutation power of 0.005, and a mutation probability of 0.6. All experiments were also run with a linearly decreasing mutation probability, where the mutation probability decreases with each generation.

Gaussian Mutation. Referred to as *Simple mutation* for the remainder of this research. If the individual is selected for mutation, some noise is added to the genes, or weights of that individual. The added noise is equal to a randomly selected weight value from

a normal distribution with a mean of 0 and a variance of 1, multiplied by the mutation power. This is the mutation technique that was applied in the paper published by *OpenAI*.

Random Mutation. This mutation is similar to the *Simple Mutation*, the only difference being that the mutation power is randomized to a uniform power between 0 and the defined mutation power of 0.005.

Decaying Mutation. The goal of this mutation technique is to linearly reduce the mutation power as the number of generations increases. Adding even small amounts of noise to the weights of the neural networks (i.e the individual solutions) can dramatically change the way they behave; the goal of this mutation technique is to reduce the amount of noise added to the weights as the population converges. This means earlier generations will have higher degrees of noise added, whereas later generations will have much smaller degrees of noise added.

5. RESULTS AND DISCUSSION

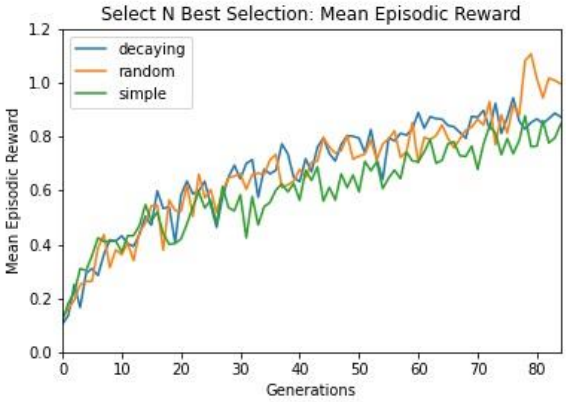


Figure 5. The mean episodic reward for all three mutation techniques with Select N Best selection.

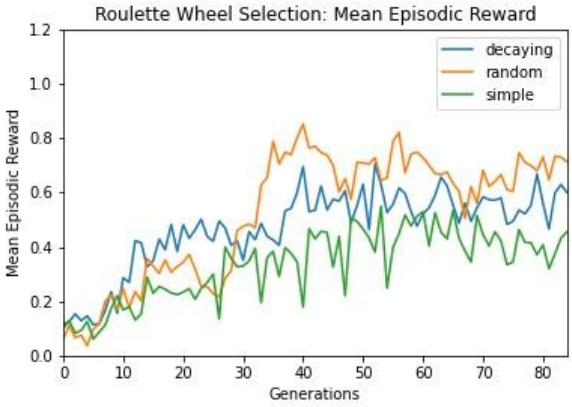


Figure 6. The mean episodic reward for all three mutation techniques with Roulette Wheel/Fitness Proportionate selection.

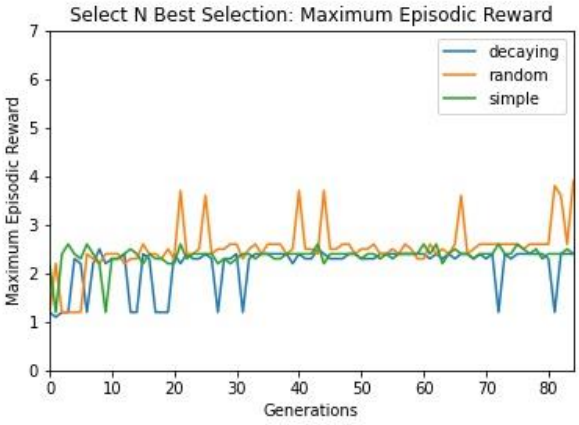


Figure 7. The maximum episodic reward for all three mutation techniques with Select N Best selection.

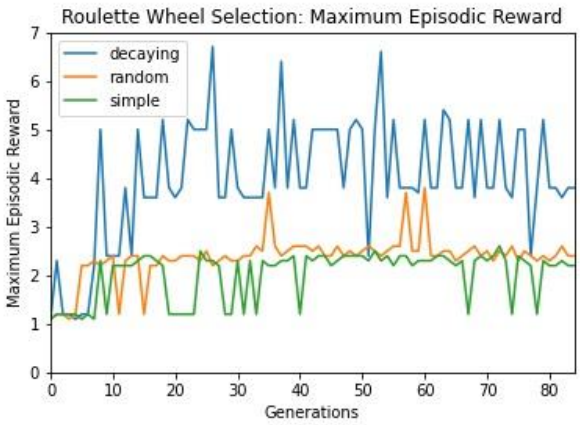


Fig 8. The maximum episodic reward for all three mutation techniques with Roulette Wheel/Fitness Proportionate selection.

We are interested in analyzing the *mean episodic rewards* and *maximum episodic rewards* of the experiments; the mean episodic rewards will give us insights into whether the population was evolving as expected and whether the population hit a plateau at some point during its evolution. The maximum episodic reward will allow us to assess the performance of the elite of each generation and determine whether a plateau was reached at a certain reward level. It is particularly important to analyze for performance plateaus in the OT environment, as they can indicate the introduction of a new game mechanic that the agents are unable to

overcome; in the baseline results published by the team behind the Obstacle Tower environment, they found that with the weak generalization evaluation criteria (i.e. where agents are trained on a fixed set of 100 seeds), the agents were plateauing at the 5th floor, which is the floor where a locked doors mechanic was introduced [Juliani et al., 2019]. Solving this mechanic required the model to have a long-term memory mechanism, which their models did not have.

The results of the experiments are organized in a way that makes it easy to compare the mean and maximum episodic rewards for both selection and all three mutation methods: Figures 5. and 6. compare the mean episodic rewards for the *Simple/Gaussian* (green), *Random* (orange), and *Decaying* (blue) mutation methods under the *Select N Best* (figure 5.) and *Roulette Wheel* (figure 6.) selection methods. Figures 7. and 8. compare the maximum episodic rewards for the same mutation and selection methods. The *y-axis* shows the rewards, while the *x-axis* shows the generation at which the reward was obtained. As a reminder, we used the OT environment's default reward configuration, where the agent receives a sparse reward of +1 for completing a floor of the tower, and a dense reward of +0.1 for opening doors, solving puzzles, or picking up keys.

Figures 5. And 6. show that the selection and mutation techniques facilitate evolution and that the average fitness scores of the populations increases with each generation. For both selection methods, the *Decaying* and *Random* mutations performed better on average than the *Simple* mutation. Figure 5. Shows that *Select N Best*, the selection method where the top performers are selected to build the next generation, has a steadier incline, and showcases higher average rewards than *the Roulette Wheel* selection (figure 6.), which seems to plateau around generation 50 for each mutation method. Looking at Table 5., which shows the results of the baseline models published by Juliani et al. [16], we see that with the *Select N Best*

selection method, all three mutation methods achieve the same mean episodic reward as their *Proximal Policy Optimization*[12] (PPO) algorithm, with the *Random* mutation achieving a higher mean episodic reward of 1.15. This is not the case for the *Roulette Wheel* selection; only the *Random* mutation method achieves a mean episodic reward of 0.8 around the 42nd generation before dropping below the PPO score. It is important to note that in order to achieve the results published in Table 5., Juliani et al. [16] trained their PPO and Rainbow[13] (RNB) models sessions spanning 20 million environment steps, which is significantly more that the number steps performed for this research (see Table 4.). As the mean episodic reward for the mutation methods under the *Select N Best* selection method do not seem to be plateauing (figure 5.), it is plausible that their mean episodic scores would have increased if they were to take up to 20 million steps. In the future, it would be interesting to increase the number of generations or the number of agents for the three mutation methods under the *Select N Best* selection method. The graph for the mean episodic rewards of PPO and RNB at different evaluation criteria published by Juliani et al. [16] is available in Appendix 8.

Mutation Method	Select N Best	Roulette Wheel
Simple	12 019 911	11 261 466
Random	12 261 356	11 824 878
Decaying	12 235 011	11 759 863

Table 4. Total number of environment steps after the maximum number of generations was reached (i.e stopping criteria)

[12] Proximal Policy Optimization (PPO) is a RL algorithm belonging to the family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment and optimizing a "surrogate" objective function using stochastic gradient ascent [Schulman et al., 2017].

[13] DeepMind's Rainbow algorithm is the combination of six independent improvements on the Deep Q-Learning (DQN) algorithm. In their 2017 paper, Hessel, M. et al. [38] demonstrated that combining the *Double Q-Learning*, *Prioritized Replay*, *Dueling Networks*, *Multi-Step Learning*, *Distributional RL*, and *Noisy Nets* techniques into one DQN algorithm outperformed the state-of-the-art RL algorithms on the Atari 2600 benchmark.

Model	Mean Episodic Reward	Number of Environment Steps
Proximal Policy Optimization (PPO)	0.8	20 000 000
Rainbow (RNB)	3.4	20 000 000
Human	15.6	-

Table 5. Baseline results from the Obstacle Tower paper comparing PPO, Rainbow (Dopamine), and human performance reported as the mean of the number of floors solved in a single episode under "weak generalization" evaluation criteria.

Figures 7. and 8. reveal that the *Roulette Wheel* selection outperforms the *Select N Best selection* in terms of achieving the highest episodic score. Figure 8. shows that the *Decaying* mutation combined with the *Roulette Wheel* selection achieves the highest scores, however it is difficult to attribute the agents' performance to the mutation function, as the figure shows that the highest score is achieved early in the evolution, at a point where the mutation power would still be very close to the one applied with the *Simple* mutation. This makes it difficult to single out any mutation technique as being more effective at producing better offspring. The success of the *Decaying* mutation function can more likely be attributed to randomness; similar to how the *OpenAI* researchers found that even a random network can get good results, the success of the decaying mutation technique can likely be attributed to a lucky initialization of the network weights.

It is clear from figures 7. and 8. that the elite agents (i.e. the agent with the highest episodic score at every generation) are plateauing very early in the evolution and that for all selection and mutation methods the elites stagnate on the second floor; only the *Decaying* mutation with *Roulette Wheel* selection reaches the 4th and even the 6th floor but plateaus 3rd floor. The few spikes in figures 7. and 8. can likely be attributed to luck and randomness, as the early floors of the OT environment are relatively straightforward; the there are no traps

or keys to collect, only a few orbs that increase the remaining time but do not reward the agent. As such, even with a random set of moves, an agent could reach the second floor.

What we learn from the graphs is that the application of ES works in the Obstacle Tower; figures 5. and 6. show that at every generation the population mean episodic reward increases. This means that on average, the individual agents in the population become better even when perturbations are applied to their parameters (i.e. mutation). However, figures 7. and 8. make it clear that the agents are unable to evolve beyond a certain performance level very early in their evolution. This can be attributed to many things, all of which are speculative at this point:

Evolution Parameters - The mutation power could be too high or too low, which would over-perturb or under-perturb the parameters of the model; over-perturbation would cause the lose the policy it had learned up until that point, while under-perturbation would not change the parameters enough for the model to learn something new. The frequency of mutations occurring during the evolution (i.e. the mutation probability) could have been too high or too low; perturbing agent parameters too often or not enough could prevent them from learning correctly. Perhaps it also does not make sense to blindly apply mutations to the entirety of the neural network parameters; it would be interesting to study the application of more targeted perturbations. The population size for these experiments was quite small (85) due to the limited number of workers/CPUs (96) available compared the 1000s of CPUs that were used by *OpenAI* in their 2017 research. Our assumption is that increasing the population size would yield better results. Finally, due to budgetary constraints, the number of generations was limited to 80. However, it is clear from figure 5. that the agents had not yet reached a plateau with regards to the mean episodic reward for all three mutation methods under the *Select N Best* selection method;

perhaps that with more generations, the mean episodic rewards would have continued to increase under those parameters.

Environment Parameters – There are a number of built-in environment parameters that could be tested, such as using the *greyscale* flag to convert the images to greyscale format (1 x 128 x 128) or turning off the retro flag so that the images would be full-size (3 x 128 x 128) instead of downscaled (3 x 84 x 84). Furthermore, customizations could have been applied to the action space and the reward functions: the actions space could have been reduced with certain moves being removed from the action space (such as jumping or turning left and right) to facilitate training [Kanervisto et al., 2020], while the reward function could have been made more complex by giving negative rewards to repeating the same actions multiple times in a row.

Model Architecture – For this research, a basic implementation of a CNN with only two convolution layers and no pooling layers was used. While such a neural network architecture could make sense for simpler 2D and 3D games, it was expected that it would not perform well in a complex environment like the OT where the agent is expected to have a long-term memory, precise motor skills, and the ability to identify objects from pixels in constantly changing backgrounds and environments. Perhaps adding components that have proven to be effective at solving these problems to the model would increase the agents' performances, increasing the number of convolution layers as well as including pooling layers between them could help the agent distinguish between details of the environment more accurately.

Finally, it would be interesting to apply more mutation and selection techniques, as well as implementing GA crossover techniques to study their effect on performance. While there were plans to develop and experiment with more selection and mutation techniques, budget

constraints for this research have made it so that we have had to limit ourselves to the ones presented in the paper.

6. CONCLUSIONS

It was not possible to achieve the baseline results published in the original Obstacle Tower paper [Juliani et al., 2019] with a basic implementation of ES. In this research we have demonstrated that a basic implementation of ES struggles to evolve in a procedurally generated environment where the agent must be able to generalize and have some form of memory. However, this does not mean that ES cannot be improved to perform better in such environments; the results of the roulette wheel selection combined with the decaying mutation show that it is possible to improve the performance of a basic implementation ES in this complex environment. However, the results of this combination can likely be attributed to a lucky random initialization, as it achieves this high score very early in its evolution, at a point where there is hardly any distinction in the application of the decaying mutation or the simple mutation.

The results obtained don't allow us to conclusively say that we were able to improve on the performance of the basic ES implementation; for both selection techniques and all mutation techniques, the agents seem to stagnate close to their initial fitness scores, leading us to believe that they are not in fact learning and only achieve slightly higher rewards from time to time entirely by chance.

To conclude, we were not able to obtain any significant results when applying *OpenAI's* ES implementation to the Obstacle Tower environment, nor were we able to definitively improve on their ES implementation by applying new selection and mutation techniques taken from GAs. However, this does not mean that it is impossible for ES to perform well in complex 3D environments; there are many techniques, model architectures, and environment changes that can have not been explored during this research and that can potentially give better

insights on how to improve ES's performance in complex 3D environments. This study can serve as a benchmark for future researchers who are interested in exploring the application of ES to complex 3D environments.

The initial goal of this research, which was to test ES' generalization ability by evaluating its performance in a complex 3D environment, and to assess the performance of a number of selection and mutation methods borrowed from GA, was achieved. While ES did not outperform the benchmarks on the OT environment presented by Juliani et al. [16], it achieved mean episodic rewards that were on par with their PPO benchmarks, as did the suggested mutation methods under *Select N Best* selection. This was the first time that ES was applied to the Obstacle Tower environment, and this research can serve as a benchmark for the performance of evolutionary algorithms as alternatives to RL in the Obstacle Tower environment.

7. LIMITATIONS AND RECOMMENDATIONS FOR FUTURE WORKS

In the future, it would be interesting to study the impact that crossover techniques could have on ES's performance, as sharing genetic information between two fit parents is a key mechanism to ensure convergence in GAs. Some consideration will need to be given as to how these crossovers would affect the individuals, as the individuals are comprised of model weights and the smallest changes to these weight values can completely change the agent's performance. Staying in the realm of GAs, it would also be interesting to further explore traditional as well as more cutting-edge selection and mutation techniques, as the ones that were implemented for this research were very basic, and not what could be considered state-of-the-art [Abdoun et al., 2012]. Again, for the mutation techniques it will be important to consider how the mutation affects the weights of the model; perhaps it would make sense to only mutate targeted parameters of the model instead of blindly applying the mutation to all parameters like it was done in this study.

Using more complex Neural Network architectures that have some form of memory and that are better suited for vision tasks would perhaps perform better than the basic *CNN* that was used for this research. When completing the *Obstacle Tower Challenge*, which was a challenge issued by Unity when they first release the environment, the top performer used a stack of *Deep Neural Networks* (DNN), where the different models had different roles, such as classifying objects in the game and controlling movement. Regarding the *CNN* used for this research, it was expected that this basic implementation with only two convolution layers and no pooling layers would yield mediocre results; adding more convolution layers with pooling layers between them would likely improve the model's capacity to distinguish important items (keys, doors, orbs) more accurately and perform better in a 3D environment like the OT.

Finally, there are changes to the environment, or the way the agent interacts with the environment, that can be applied in order to increase performance. For this research, changes were made to the environment to minimize the action space. Unfortunately, these methods were not tested due to budgetary constraints. Using greyscale instead of RGB images could also help the agent identify with its surroundings better and changing to reward function to include more densely distributed rewards can potentially help the agent learn to navigate the environment more effectively. Most of the suggested improvements in this section were not tested due to time and budget constraints, as mentioned previously. However, given the work that has already been delivered for this paper, most of the suggestions above could be easily implemented in future research.

8. BIBLIOGRAPHY

[1] Abdoun, O., et al. "*Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem*." *ArXiv:1203.3099 [Cs]*, (2012). *arXiv.org*, http://arxiv.org/abs/1203.3099.

[2] Alzubaidi, L., Zhang, J., Humaidi, A.J. *et al.* "*Review of deep learning: concepts, CNN architectures, challenges, applications, future directions*". *J Big Data* 8**,** 53 (2021). https://doi.org/10.1186/s40537-021-00444-8

[3] Bellemare, M. G., et al. "*The Arcade Learning Environment: An Evaluation Platform for General Agents.*" *Journal of Artificial Intelligence Research*, vol. 47, (2013), pp. 253–79. *arXiv.org*, doi:10.1613/jair.3912.

[4] Booth, J. "*PPO Dash: Improving Generalization in Deep Reinforcement Learning*." ArXiv:1907.06704 [Cs], (2019). arXiv.org, http://arxiv.org/abs/1907.06704.

[5] Braylan A., Hollenbeck M., Meyerson E., Miikkulainen R., et al. "*Frame skip is a powerful parameter for learning to play Atari*". AAAI workshop, (2015). UTexas, http://nn.cs.utexas.edu/downloads/papers/braylan.aaai15.pdf

[6] Brockman, G., et al. "*OpenAI Gym*." ArXiv:1606.01540 [Cs], (2016). arXiv.org, http://arxiv.org/abs/1606.01540.

[7] Brunnbauer, A., et al. "*Model-Based versus Model-Free Deep Reinforcement Learning for Autonomous Racing Cars*." *ArXiv:2103.04909 [Cs]*, (2021). *arXiv.org*, http://arxiv.org/abs/2103.04909.

[8] Câmara, D. "*Evolution and Evolutionary Algorithms*." *Bio-Inspired Networking*, Elsevier, (2015), pp. 1–30. *DOI.org (Crossref)*, doi:10.1016/B978-1-78548-021-8.50001-6

[9] Christlein, V., et al. "*Deep Generalized Max Pooling*." *ArXiv:1908.05040 [Cs]*, (2019). *arXiv.org*, http://arxiv.org/abs/1908.05040.

[10] Dayan, P., Niv, Y. "*Reinforcement Learning: The Good, The Bad and The Ugly*." *Current Opinion in Neurobiology*, vol. 18, no. 2, (2008), pp. 185–96. *DOI.org (Crossref)*, doi:10.1016/j.conb.2008.08.003.

[11] Dianati, M., Song, I., Treiber, M. *"An Introduction to Genetic Algorithms and Evolution."* ceas3.uc.edu, (2002). University of Cincinnati https://static.aminer.org/pdf/PDF/000/304/974/multi_population_evolution_strategies_for_structural_image_analysis.pdf

[12] Ecoffet, A. "*Montezuma's Revenge Solved by Go-Explore, a New Algorithm for Hard-Exploration Problems (Sets Records on Pitfall, Too)*." Uber Engineering Blog, (2018), https://eng.uber.com/go-explore/.

[13] Gu, J., et al. "*Recent Advances in Convolutional Neural Networks." ArXiv:1512.07108 [Cs]*, (2017). *arXiv.org*, http://arxiv.org/abs/1512.07108.

[14] Howard, A. G., et al. "*MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.*" *ArXiv:1704.04861 [Cs]*, (2017). *arXiv.org*, http://arxiv.org/abs/1704.04861.

[15] Huang, S., et al. "*Adversarial Attacks on Neural Network Policies*." ArXiv:1702.02284 [Cs, Stat], (2017). arXiv.org, http://arxiv.org/abs/1702.02284.

[16] Juliani, A., et al. "*Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning*." ArXiv:1902.01378 [Cs], (2019). arXiv.org, http://arxiv.org/abs/1902.01378.

[17] Juliani, A., et al. "*Unity: A General Platform for Intelligent Agents*." ArXiv:1809.02627 [Cs, Stat], (2020). arXiv.org, http://arxiv.org/abs/1809.02627.

[18] Kanervisto, A., et al. "*Action Space Shaping in Deep Reinforcement Learning*." *ArXiv:2004.00980 [Cs]*, (2020). *arXiv.org*, http://arxiv.org/abs/2004.00980.

[19] Li, Z., et al. "*A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects.*" *ArXiv:2004.02806 [Cs, Eess]*, (2020). *arXiv.org*, http://arxiv.org/abs/2004.02806.

[20] Liang, E., et al. "*RLlib: Abstractions for Distributed Reinforcement Learning*." *ArXiv:1712.09381 [Cs]*, (2018). *arXiv.org*, http://arxiv.org/abs/1712.09381.

[21] Lillicrap, T. P., et al. "Continuous Control with Deep Reinforcement Learning." *ArXiv:1509.02971 [Cs, Stat]*, (2019). *arXiv.org*, http://arxiv.org/abs/1509.02971.

[22] Machado, M. C., et al. "*Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents*." ArXiv:1709.06009 [Cs], (2017). arXiv.org, http://arxiv.org/abs/1709.06009.

[23] Minsky, M. "*Steps toward Artificial Intelligence*," in *Proceedings of the IRE*, vol. 49, no. 1, pp. (1961), doi: 10.1109/JRPROC.1961.287775.

[24] Mnih, V., Kavukcuoglu, K., Silver, D. et al. "*Human-level control through deep reinforcement learning.*" Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[25] Mnih, V., et al. "*Asynchronous Methods for Deep Reinforcement Learning.*" ArXiv:1602.01783 [Cs], (2016). arXiv.org, http://arxiv.org/abs/1602.01783.

[26] Moritz, P., et al. "*Ray: A Distributed Framework for Emerging AI Applications.*" ArXiv:1712.05889 [Cs, Stat], (2018). arXiv.org, http://arxiv.org/abs/1712.05889.

[27] Pleines, M., et al. "*Obstacle Tower Without Human Demonstrations: How Far a Deep Feed-Forward Network Goes with Reinforcement Learning*." ArXiv:2004.00567 [Cs, Stat], (2020). arXiv.org, http://arxiv.org/abs/2004.00567.

[28] Salimans, T., et al. "*Evolution Strategies as a Scalable Alternative to Reinforcement Learning.*" ArXiv:1703.03864 [Cs, Stat], (2017). arXiv.org, http://arxiv.org/abs/1703.03864.

[29] Salimans, T., et al. "*Improved Techniques for Training GANs*." *ArXiv:1606.03498 [Cs]*, June 2016. *arXiv.org*, http://arxiv.org/abs/1606.03498.

[30] Schaul, T., Glasmachers, T., Schmidhuber, J., et al. "*High dimensions and heavy tails for natural evolution strategies*". (2011). In Proceedings of the 13th annual conference on Genetic and evolutionary computation, pages 845–852. ACM, (2011).

[31] Silver, D., Huang, A., Maddison, C. et al. "*Mastering the game of Go with deep neural networks and tree search*". Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

[32] Springenberg, J. T., et al. "*Striving for Simplicity: The All Convolutional Net.*" *ArXiv:1412.6806 [Cs]*, 2015. *arXiv.org*, http://arxiv.org/abs/1412.6806.

[33] Stavens, D., Sebastian, T. "*A Self-Supervised Terrain Roughness Estimator for Off-Road Autonomous Driving.*" ArXiv:1206.6872 [Cs], (2012). arXiv.org, http://arxiv.org/abs/1206.6872.

[34] Vinyals, O., Babuschkin, I., Czarnecki, W.M. et al. "*Grandmaster level in StarCraft II using multi-agent reinforcement learning*". Nature 575, 350–354 (2019). https://doi.org/10.1038/s41586-019-1724-z

[35] Wierstra, D., Schaul, T., Peters, J., Schmidhuber, J. et al. "*Natural evolution strategies. In Evolutionary Computation*". (2008). CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on, pages 3381–3387. IEEE, 2008.

[36] *Part 2: Kinds of RL Algorithms — Spinning Up Documentation*. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. Accessed 8 Nov. 2021.

[37] Cheoi KJ., Choi H., Ko J." *Empirical Remarks on the Translational Equivariance of Convolutional Layers*". (2020). Applied Sciences. 2020; 10(9):3161. https://doi.org/10.3390/app10093161

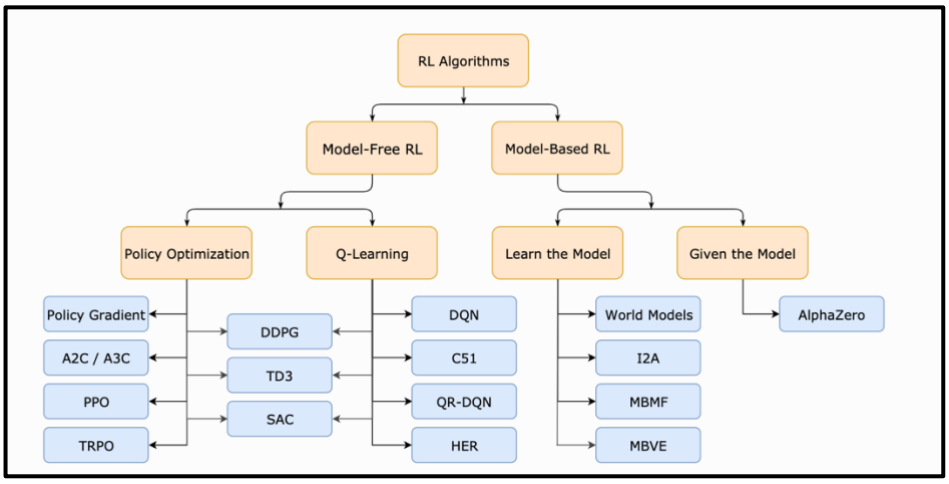
[38] Hessel, Matteo, et al. "*Rainbow: Combining Improvements in Deep Reinforcement Learning.*" ArXiv:1710.02298 [Cs], Oct. 2017. arXiv.org, http://arxiv.org/abs/1710.02298.

[39] Schulman, John, et al. "*Proximal Policy Optimization Algorithms*." ArXiv:1707.06347 [Cs], Aug. 2017. arXiv.org, http://arxiv.org/abs/1707.06347.

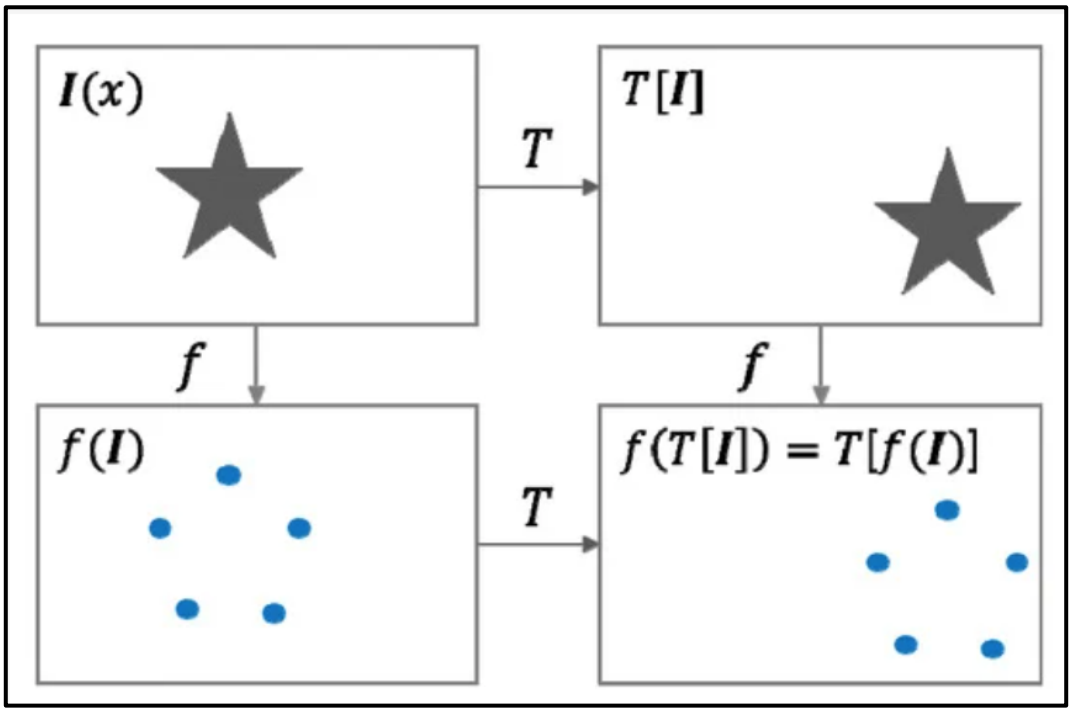
[40] Hausknecht, Matthew, et al. "*HyperNEAT-GGP: A HyperNEAT-Based Atari General Game Player*." Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, 2012, pp. 217–24. ACM Digital Library, https://doi.org/10.1145/2330163.2330195.

[41] "*Model-Based and Model-Free Reinforcement Learning - Pytennis Case Study*." Neptune.Ai, 27 Nov. 2020, https://neptune.ai/blog/model-based-and-model-free-reinforcement-learning-pytennis-case-study.

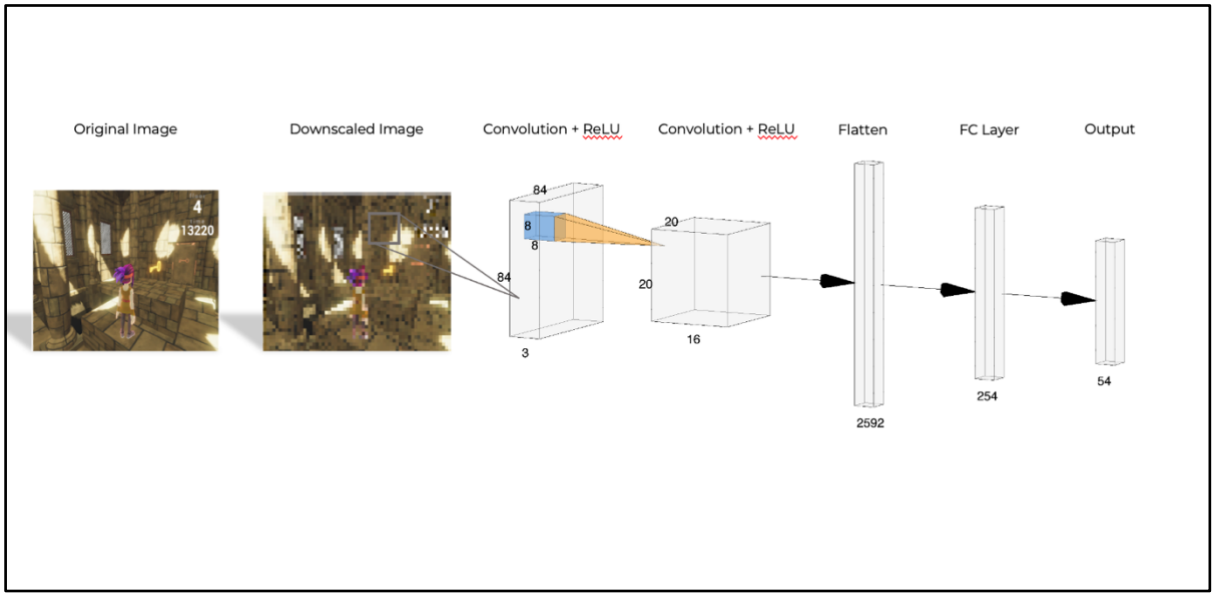
9. APPENDIX



Appendix 1. OpenAI's non-exhaustive taxonomy of modern RL algorithms [36].



Appendix 2. Showing the Translation Invariance property; when the inputs are shifted towards the right, the representations are also shifted [37].



Appendix 3. Alternative Convolutional Neural Network Architecture Diagram

Tasks (stateless)	**Actors (stateful)**
Fine-grained load balancing	Coarse-grained load balancing
Support for object locality	Poor locality support
High overhead for small updates	Low overhead for small updates
Efficient failure handling	Overhead from checkpointing

Appendix 4. Trade-offs between Tasks and Actors presented Moritz et al. [26].

```yaml
cluster_name: obstacle-tower-cluster-rllib-v0

min_workers: 0
max_workers: 10
upscaling_speed: 1.0
provider:
    type: gcp
    region: europe-west4
    availability_zone: europe-west4-c
    project_id: obstacle-tower-gcp

head_node:
  machineType: n1-highcpu-96 #n1-standard-16
  disks:
    - boot: true
      autoDelete: true
      type: PERSISTENT
      initializeParams:
        diskSizeGb: 50
        sourceImage: projects/deeplearning-platform-release/global/images/family/tf-1-13-cpu


worker_nodes:
  machineType: n1-highcpu-96
  disks:
    - boot: true
      autoDelete: true
      type: PERSISTENT
      initializeParams:
        diskSizeGb: 50
        sourceImage: projects/deeplearning-platform-release/global/images/family/tf-1-13-cpu


setup_commands:
    - export DEBIAN_FRONTEND=noninteractive
    - sudo apt-get update
    - sudo apt install -y libgconf-2-4
    - sudo apt install -y libsoup2.4
    - sudo apt install -y xorg
    - sudo apt-get install -y libxfont-dev
    - sudo apt-get install -y xvfb
    - wget https://repo.continuum.io/archive/Anaconda3-5.0.1-Linux-x86_64.sh || true
    - bash Anaconda3-5.0.1-Linux-x86_64.sh -b -p $HOME/anaconda3 || true
    - echo 'export PATH="$HOME/anaconda3/bin:$PATH"' >> ~/.bashrc
    - pip install --upgrade pip
    - pip install -U https://s3-us-west-2.amazonaws.com/ray-wheels/latest/ray-0.9.0.dev0-cp36-cp36m-manylinux1_x86_64.whl
    - pip install -U ray[dashboard]==1.1.0
```

Appendix 5. Configuration file to deploy the Kubernetes cluster with Ray on Google Cloud Platform to run the experiments for this research (.YAML).

```
from worker import Worker
from selections import roulette_wheel_selection, select_n_best



DEFAULT_CONFIG = with_common_config({
    "population_size": 85, #1000
    "max_timesteps_per_episode": 1000, #2000
    "max_evaluation_steps": 3, #2000
    "number_elites": 20,
    "mutation_power": 0.005,
    "mutation_probability": 0.6,
    "num_workers": 7,
    "create_display": False,
    "generations": 10,
    "mutation_func": "simple",
    "decreasing_mutation": False,
    "action_space": 54,
    "local_dir": "models/"
})

class GATrainer(Trainer):
    _name = "GA"
    _default_config = DEFAULT_CONFIG

    @override(Trainer)
    def _init(self, config: TrainerConfigDict, env_creator: Callable[[EnvContext], EnvType]):

        self._default_config = DEFAULT_CONFIG
        self.config = config
```

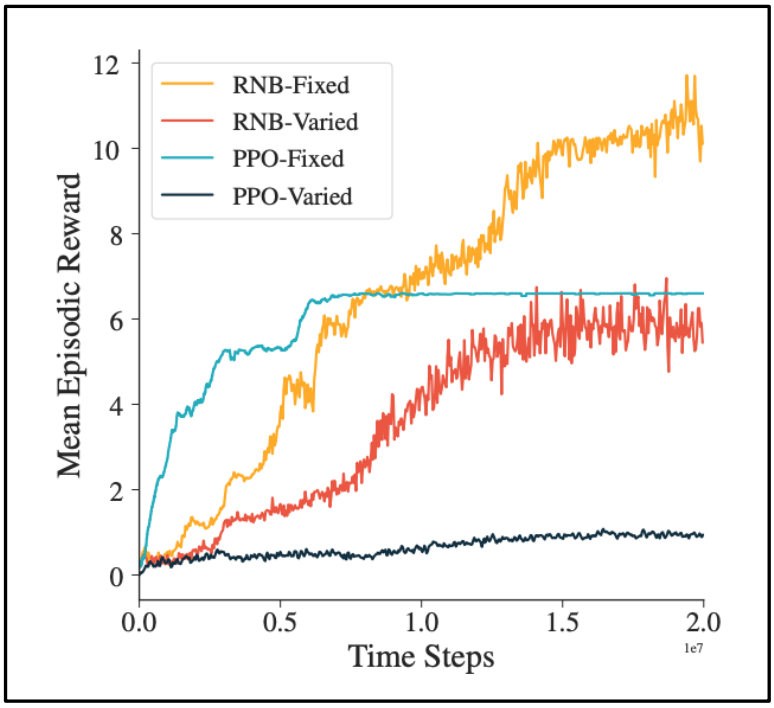
Appendix 6. Source code for the custom Ray/RLlib Trainer class (GATrainer).

```
        self.generation = 0
        self._workers = [
            Worker.remote(config, env_creator, idx + 1)
            for idx in range(config["num_workers"])
        ]
        self.episodes_total = 0
        self.timesteps_total = 0
        self.elites = []
        self.elite = None
        self.highest_reward = 0
        if self.config['decreasing_mutation']:
            self.mutation_probs = np.round(np.linspace(self.config['mutation_probability'],
                                                       0.01, self.config['generations']), 3)

    @override(Trainer)
    def step(self):
        logging.basicConfig(level=logging.INFO)
        worker_jobs = []
        for i in range(self.config['population_size']):
            elite_id = i % self.config['number_elites']
            worker_id = i % self.config['num_workers']
            weights = self.elites[elite_id] if self.elites else None
            worker_jobs += [self._workers[worker_id].evaluate.remote(weights=weights, mutate=True, record=False,
                                                                     generation=self.generation,
                                                                     mutation_probability=self.mutation_probs[self.generation])]

        results = ray.get(worker_jobs)
        elites = select_n_best(results, self.config['number_elites'])

        self.elites = []
        for result_id in elites:
            self.elites.append(results[result_id]['state_dict'])

        rewards = [result['total_reward'] for result in results]
        elite_index = np.argmax(rewards)

        if results[elite_index]['total_reward'] >= self.highest_reward:
            self.highest_reward = results[elite_index]['total_reward']
            self.elite = results[elite_index]['state_dict']


        self.timesteps_total += sum([result['timesteps_total'] for result in results])
        self.episodes_total += len(results)
        self.generation += 1

        return dict(
            timesteps_total=self.timesteps_total,
            episodes_total=self.episodes_total,
            generation=self.generation,
            train_reward_min=np.min(rewards),
            train_reward_mean=np.mean(rewards),
            train_reward_med=np.median(rewards),
            train_reward_max=np.max(rewards),
        )
```

```
    def save_checkpoint(self, checkpoint_dir=None):
        checkpoint_path = os.path.join(checkpoint_dir, "elite_model.pth")
        torch.save(self.elite, checkpoint_path)
        return checkpoint_dir



from mutations import simple_mutation, decaying_mutation, random_mutation
from model import ConvModel

@ray.remote
class Worker:
    def __init__(self,
                 config,
                 env_creator,
                 worker_index):
        self.config = config

        if config['create_display']:
            self.display = Display(visible=0, size=(800, 600), backend="xvfb")
            self.display.start()
        env_context = EnvContext(config["env_config"] or {}, worker_index)
        self.env = env_creator(env_context)
        self.model = ConvModel((3, 84, 84), action_space=config['action_space'])

        for param in self.model.parameters():
            param.requires_grad = False


    def evaluate(self, weights, mutate, record, generation, mutation_probability):
        if weights:
            self.model.load_state_dict(weights)
        else:
            self.model.init_weights()
        if mutate:
            rand_val = np.random.rand()
            if rand_val <= mutation_probability:
                if self.config['mutation_func'] == 'simple':
                    self.model = simple_mutation(model=self.model, mutation_power=self.config['mutation_power'])
                elif self.config['mutation_func'] == 'decaying':
                    self.model = decaying_mutation(model=self.model, mutation_power=self.config['mutation_power'],
                                                   generation=generation, max_generations=self.config['generations'])
                elif self.config['mutation_func'] == 'random':
                    self.model = random_mutation(model=self.model, max_mutation_power=self.config['mutation_power'])
                else:
                    sys.exit()

        obs = self.env.reset()

        rewards = []
        for ts in range(self.config['max_timesteps_per_episode']):
            obs = np.reshape(obs, (1, 3, 84, 84))
            torch_obs = torch.tensor(obs, dtype=torch.float)
            action = self.model.determine_action(torch_obs)

            obs, reward, done, info = self.env.step(action)
```

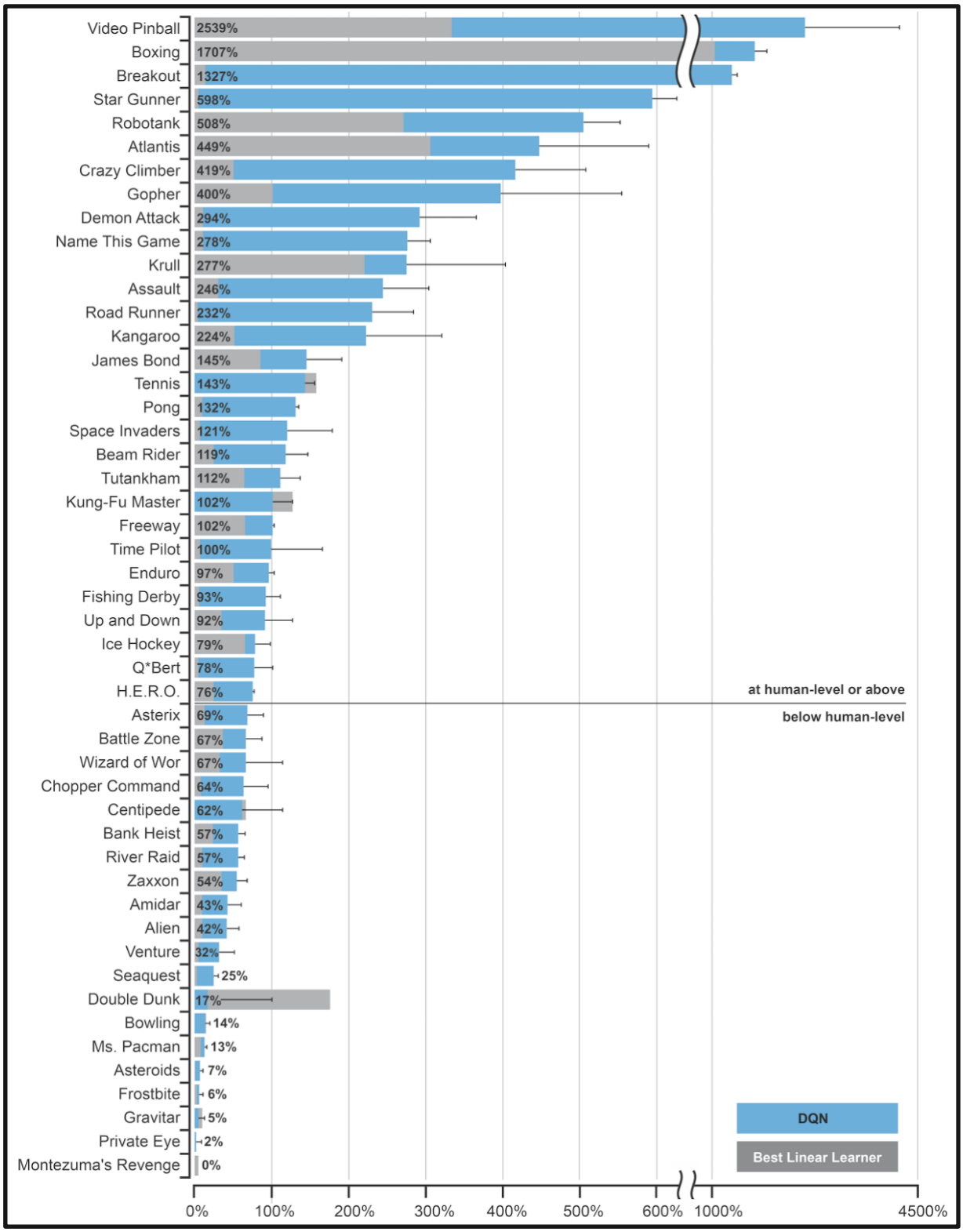
```
        rewards += [reward]



        if (done):
            break

    weights = self.model.get_weights()
    return {
        'total_reward': sum(rewards),
        'timesteps_total': ts,
        'weights': weights,
        'state_dict': self.model.state_dict(),
    }
```

Appendix 7. Source code for the customer Ray/RLlib Worker class (Worker).



Appendix 8. Proximal Policy Optimization (PPO) and Rainbow (RNB) mean episodic rewards published as benchmark performances published by Juliani et al. [16].

10. ANNEXES



Annex 1. DeepMind's Deep Q Network's performance on Atari games vs. human performance [Mnih et al., 2015].

Game	DQN	A3C FF, 1day	HyperNEAT	ES FF, 1hour	A2C FF
Amidar	133.4	283.9	184.4	112.0	548.2
Assault	3332.3	3746.1	912.6	1673.9	2026.6
Asterix	124.5	6723.0	2340.0	1440.0	3779.7
Asteroids	697.1	3009.4	1694.0	1562.0	1733.4
Atlantis	76108.0	772392.0	61260.0	1267410.0	2872644.8
Bank Heist	176.3	946.0	214.0	225.0	724.1
Battle Zone	17560.0	11340.0	36200.0	16600.0	8406.2
Beam Rider	8672.4	13235.9	1412.8	744.0	4438.9
Berzerk	-	1433.4	1394.0	686.0	720.6
Bowling	41.2	36.2	135.8	30.0	28.9
Boxing	25.8	33.7	16.4	49.8	95.8
Breakout	303.9	551.6	2.8	9.5	368.5
Centipede	3773.1	3306.5	25275.2	7783.9	2773.3
Chopper Command	3046.0	4669.0	3960.0	3710.0	1700.0
Crazy Climber	50992.0	101624.0	0.0	26430.0	100034.4
Demon Attack	12835.2	84997.5	14620.0	1166.5	23657.7
Double Dunk	21.6	0.1	2.0	0.2	3.2
Enduro	475.6	82.2	93.6	95.0	0.0
Fishing Derby	2.3	13.6	49.8	49.0	33.9
Freeway	25.8	0.1	29.0	31.0	0.0
Frostbite	157.4	180.1	2260.0	370.0	266.6
Gopher	2731.8	8442.8	364.0	582.0	6266.2
Gravitar	216.5	269.5	370.0	805.0	256.2
Ice Hockey	3.8	4.7	10.6	4.1	4.9
Kangaroo	2696.0	106.0	800.0	11200.0	1357.6
Krull	3864.0	8066.6	12601.4	8647.2	6411.5
Montezuma's Revenge	50.0	53.0	0.0	0.0	0.0
Name This Game	5439.9	5614.0	6742.0	4503.0	5532.8
Phoenix	-	28181.8	1762.0	4041.0	14104.7
Pit Fall	-	123.0	0.0	0.0	8.2
Pong	16.2	11.4	17.4	21.0	20.8
Private Eye	298.2	194.4	10747.4	100.0	100.0
Q*Bert	4589.8	13752.3	695.0	147.5	15758.6
River Raid	4065.3	10001.2	2616.0	5009.0	9856.9
Road Runner	9264.0	31769.0	3220.0	16590.0	33846.9
Robotank	58.5	2.3	43.8	11.9	2.2
Seaquest	2793.9	2300.2	716.0	1390.0	1763.7
Skiing	-	13700.0	7983.6	15442.5	15245.8
Solaris	-	1884.8	160.0	2090.0	2265.0
Space Invaders	1449.7	2214.7	1251.0	678.5	951.9
Star Gunner	34081.0	64393.0	2720.0	1470.0	40065.6
Tennis	2.3	10.2	0.0	4.5	11.2
Time Pilot	5640.0	5825.0	7340.0	4970.0	4637.5
Tutankham	32.4	26.1	23.6	130.3	194.3
Up and Down	3311.3	54525.4	43734.0	67974.0	75785.9
Venture	54.0	19.0	0.0	760.0	0.0
Video Pinball	20228.1	185852.6	0.0	22834.8	46470.1
Wizard of Wor	246.0	5278.0	3360.0	3480.0	1587.5
Yars Revenge	-	7270.8	24096.4	16401.7	8963.5
Zaxxon	831.0	2659.0	3000.0	6380.0	5.6

Annex 2. From Salimans et al.[28]: Final results obtained using Evolution Strategies on Atari 2600 games (feedforward CNN policy, deterministic policy evaluation, averaged over 10 re-runs with up to 30 random initial no-ops), and compared to results for DQN and A3C from Mnih et al. [24] and HyperNEAT from Hausknecht et al. [40]. A2C is our synchronous variant of A3C, and its reported scores are obtained with 320M training frames with the same evaluation setup as for the ES results. All methods were trained on raw pixel input.