Evolutionarily-Curated Curriculum Learning for Deep Reinforcement Learning Agents

Michael Cerny Green, Benjamin Sergent, Pushyami Shandilya, Vibhor Kumar
Imbellus, Los Angeles, CA

Abstract

In this paper we propose a new training loop for deep reinforcement learning agents with an evolutionary generator. Evolutionary procedural content generation has been used in the creation of maps and levels for games before. Our system incorporates an evolutionary map generator to construct a training curriculum that is evolved to maximize loss within the state-of-the-art Double Dueling Deep Q Network architecture with prioritized replay (**?**) (**?**). We present a case-study in which we prove the efficacy of our new method on a game with a discrete, large action space we made called *Attackers and Defenders*. Our results demonstrate that training on an evolutionarily-curated curriculum (directed sampling) of maps both expedites training and improves generalization when compared to a network trained on an undirected sampling of maps.

1 Introduction

The use of games as benchmarks for AI progress has propagated to nearly the entire AI research community, including Chess, Atari Breakout, and more recently with Go as superhuman agents have been developed. Many recent papers document new AI methods being used within game environments. But the current state-of-the-art training method to ensure generalization in a neural network system of any kind remains brute force random sampling training, i.e. give a network enough unique states to train on and hope generalization naturally occurs. We propose a new method of directed sampling training called 'evolutionarily-curated curriculum learning' (ECCL), which we argue results in faster and better network generalization.

Past experiments have shown the potential in teaching simple concepts first, on which more complicated ones can then be taught, as a successful way to train networks (**?**). To go one step further, we propose a method which specifically identifies weaknesses in the network and then generates content that force the network to face these weaknesses head-on. Our system dynamically evolves a curriculum by searching for content that maximizes the network's loss, which makes the network generalize faster and perform better.

In this paper, we give a brief overview of research within reinforcement learning, the use of evolutionary algorithms in procedural content generation, and curriculum learning research for networks in Section 2. In Section 3 we discuss the theory of evolutionarily-based curriculum learning and how it could be applied to a reinforcement learning agent. We then use *Attackers and Defenders* as a case study in Section 4, with results and discussion from our experiment in Section 5, and conclude in Section 6.

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

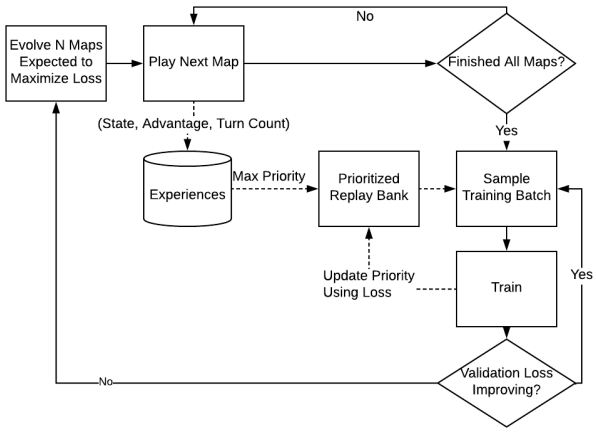

Figure 1: Evolutionarily-based curriculum learning in an agent's training loop

2 Background

This section begins with a brief overview of deep reinforcement learning research, beginning with Minksy in 1954 (**?**) and finishing with the state-of-the-art DDDQN (**?**), AlphaZero (**?**; **?**), and ExIt (**?**) agent architectures. It then discusses evolutionary algorithms and how they can be applied toward procedural content generation in games. The section concludes with the concept of curriculum learning for machines and the admittedly scant amount of research within this area.

2.1 Deep Reinforcement Learning

Reinforcement Learning (RL) concerns itself with the idea of learning through trial-and-error interactions with a dy-

namic environment and balancing the reward trade-off between long-term and short-term planning (**?**). RL has been studied since Minksy (**?**) in the 1950's. Since then, important improvements to the concept have been advanced including the temporal difference learning method (**?**; **?**), on which q-learning (**?**) and actor-critic (**?**) techniques are built. Gullapalli (**?**) and Williams (**?**) are early examples of the use of RL within artificial neural networks (ANNs). When Rumelhart et al discovered the *backpropogation* algorithm (**?**), deep learning took off in popularity. This has been bolstered recently by the rise in the capability and affordability of computer-processing units and graphical-processing units. Further readings of work in RL can be found in reviews by Schmidhuber (**?**) and Szepsvári (**?**).

RL applied to deep learning has been only recently successful due to some key advancements. Mnih et al proposed Deep Q Networks (DQNs) (**?**) using target networks and experience replay to improve the known divergence issues present in RL. van Hesselt et al built Double Deep Q Networks (**?**) which help reduce the overestimation errors that normal DQNs suffer from. Hessel et al wrote a paper surveying state-of-the-art improvements to DQN within the Atari framework, including Double Dueling Deep Q Networks, Distributional Deep Q Networks, and Noisy Deep Q Networks (**?**). Prioritized experience reply (**?**) allows DQNs to remember past experiences in a prioritized fashion, rather than at the same frequency that they were experienced.

Further DQN stability has been attained through the introduction of dueling networks (**?**), which are built on top of Double DQNs. Dueling networks use two separate estimators for the state value function and state-dependent action advantage function to generalize learning across actions without effecting the underlying RL algorithm. Double Dueling DQNs are currently considered state-of-the-art reinforcement learning algorithms.

Asynchronous Advantage Actor Critic (A3C) networks were created by Mnih et al in a successful attempt to apply neural networks to actor-critic reinforcement learning. The "asynchronous" part of A3C makes training parallelizable, allowing for massive computation speedups.

The AlphaGo algorithm combined Monte Carlo Tree Search (MCTS) with deep neural networks to play the game of Go and become the first AI to beat the human world champion of the game (**?**). The more advanced version, AlphaZero, was able to learn only by self-playing and outperformed AlphaGo (**?**). The same AlphaZero architecture was then applied to both Chess and Shogi to convincingly beat world-champion programs (**?**). At the same time, Anthony et al discovered the Expert Iteration algorithm (**?**), which also uses a neural network policy to guide tree search. Since the advent of these state-of-the-art algorithms, much research has been done to either improve or apply them to different problems with varying degrees of success.

2.2 Evolution and Procedural Content Generation

Evolutionary algorithms (EA) fall within the area of optimization search inspired by Darwinian evolutionary concepts such as reproduction, fitness, and mutation (**?**). EA has been used within games to procedurally generate levels, game elements within them, and sometimes even games themselves (**?**). Puzzle generation is a primary example of this kind of search-based generation (**?**), which can be used to create puzzles with a desired solution difficulty. *Checkpoint based fitness* allows for fitness function parameterization (**?**), affording substantial control over generated properties. Stylistic generation is made possible by using fashion-based cellular automata (**?**). An EA generator for a given game can also evolve many things at once by decomposing level generation into multiple parts. McGuinness et al did this by creating a micro evolutionary system which evolves individual tile sections of a level and an overall macro generation system which evolves placement patterns for the tiles (**?**). Evolutionary search can be used for generalized level generation in multiple domains such as General Video Game AI (**?**) and PuzzleScript (**?**). In later work by Khalifa et al. (**?**), they worked on generating levels for a specific game genre (Bullet Hell genre) using a new hybrid evolutionary search called Constrained Map-Elites. The levels were generated using automated playing agents with different parameters to mimic various human play-styles. Green et al. used EA to evolve *Super Mario Bros* (Nintendo, 1985) scenes which taught specific mechanics to the player (**?**). We recommend Khalifa's review on searched-based level generation for further reading into EA for generation in games (**?**).

2.3 Curriculum Learning in Machines

The concept of curriculum learning (CL) in machines can be traced back to Elman in 1993 (**?**). The basic idea is to keep initial training data simple and slowly ramp up in difficulty as the model learned. Krueger and Dayan (**?**) did a cognitive-based analysis with evidence that shaping data provided faster convergence. CL was further explored by Bengio et al. in 2009, in an attempt to define several machine learning guided training strategies (**?**). Their experiments suggested that incorporating CL into training a model could both speed up training and significantly increase generalization. Recently, Curriculum Learning within adversarial network training (**?**) was explored by Cai et al in an attempt to mitigate "forgetfulness" and increase generalization to reduce the effectiveness of adversarial network attacks.

2.4 Evolution within Networks

Using evolutionary strategy for neural networks is a well-researched topic. Genetic algorithms were used in the node weight balancing of a network by Ronald et al (**?**) to evolve a controller for soft-landing a toy lunar module in a simulation. Simultaneously, Gruau evolved the structures and parameters of networks using cellular encoding (**?**). Cartesian Genetic Programming was first designed by Miller et al (**?**) to design digital circuits using genetic programming. It is called 'Cartesian' because of the way it represents a program using a 2-dimensional set of nodes.

All of these methods and more may be housed under the umbrella of "neuroevolution" which is well-defined by Floreano et al (**?**). A survey of neuroevolution within games is

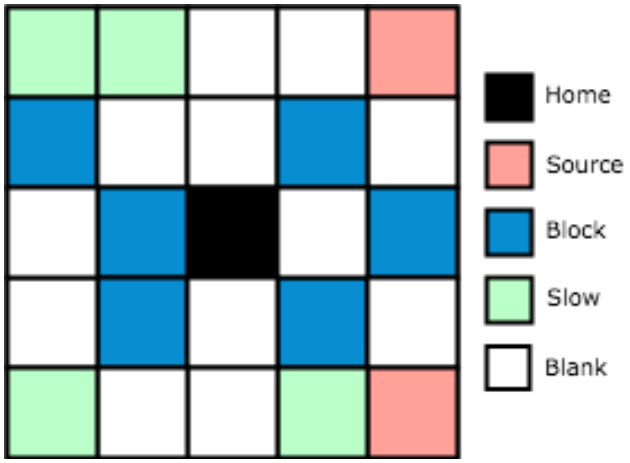

Figure 2: A visualization of a map in *Attackers and Defenders*

written by Risi and Togelius (**?**). We mention neuroevolution to highlight the major difference that whereas it is used to evolve the parameters or architecture of a network, our approach evolves training data as part of a curriculum for a constant architecture.

3 Evolved Curriculum in the Training Loop

Traditional training of a neural network involves a training schedule established by taking random batches of a fixed training set, which is assumed to be an unbiased random sample of the data space. For game-playing agents specifically, this training set is the set of levels or maps that the network is exposed to. The hope is that the sample is sufficient to train the neural network to generalize to the entire data space (i.e. all possible maps).

Instead, ECCL relies on producing a training curriculum composed of a biased sampling of the data space, specifically designed to improve generalization. ECCL involves two parts: the agent to be trained and an evolutionary generator. Unlike neuroevolution methods, the use of evolution in evolved curriculum is *not* to evolve the weights or architecture of the network. The evolutionary generator's sole purpose is to evolve *scenarios* which have the best potential to improve the agent's generalization. To do this, the goal of the evolutionary generator is to produce maps which maximize the loss of the agent network.

Figure 1 shows the training loop that uses evolutionarily-based curriculum learning. In this figure, the agent is a Double Dueling Deep Q Network (DDDQN) with Prioritized Replay (**?**; **?**) which we used in our case study, explained in Section 4. One reason for using this specific type of network architecture is the choice of our library- Tensorflow. The network used in ecperiments presents in this paper is near state-of-the-art with a few deletions that we deemed unnecessary to use.

When the network requests more maps to train on, the evolutionary generator is tasked to evolve maps which maximize the amount of loss in the network by querying the loss directly. By maximizing loss, the network is necessarily seeing a valid map in map space that it failed to generalize to properly. Concretely, this could be a edge-case map that where an uncommon move is optimal or requires another strategy altogether from maps the network has already seen. The agent plays this map, as well as any others in the batch.

After finishing a batch, the system divides the different game states caused by moves from all games into experience snapshots, and sorts the experience snapshots by priority to be stored in a prioritized replay bank. It then uses these experiences to train the weights in the network appropriately, and updates priorities in the bank using network loss. The network then asks the generator to produce more maps to repeat the process until training terminates.

4 Case Study: *Attackers and Defenders*

The following section concerns a case study in which we compare our method's impact against other state-of-the-art algorithms. We hypothesized that the evolutionary generator would create higher quality training data which would more effectively improve network performance, and our experiment attempts to prove this.

Section 4.1 explains the game of *Attackers and Defenders* a simple tower defense game which we created as a testbed. Section 4.3 explains the generator and how it produces maps. Section 4.4 describes the training/testing methods used to validate our claims.

4.1 Attackers and Defenders

To prove the concept of ECCL, we created a discrete, large action space, tower-defense game called *Attackers and Defenders* as a test-bed. Figure 2 displays a visualization of a game map. The objective of the player in *Attackers and Defenders* is to prevent enemies from reaching their *home* tile for as long as possible. This game is a model of sequential decision making that applies broadly to other game and non-game domains, which makes it an appropriate testbed for our algorithm.

Game Entities *Attacker* entities have hit-points (HP), which may vary in number and generally increase over the course of a single play session. To facilitate this survival goal, the player is given *Defender* entities which do damage to *attacker* HP, *slow* tiles which penalize *attacker* movement when traveling through these tiles, and *block* tiles which prohibit *attacker* movement. Table 1 displays all tiles/entities within the game and how they work.

Game Loop Each turn, the player is prompted to place a *defender*, *slow*, or *block* tile on the game map. After the player places an entity, the game advances forward one turn. During this period, a *source* tile may spawn an *attacker*, which will then slowly advance toward the *home* tile. If an *attacker* moves into a space within a *defender*'s attack range (which may overlap with other *defenders*), the *attacker* will suffer damage equivalent to the sum of all in-range *defender* damage. If an an *attacker* runs out of HP, it will be destroyed. If an *attacker* manages to move onto the *home* tile, the game will end.

Game Entity	Description
Neutral	an empty tile with no penalties
Slow	a tile makings attackers 2 turns to move
Block	a tile preventing attackers from moving
Home	the tile attackers are trying to move onto
Source	the tiles from which attackers spawn
Attacker	automatous entities which are moving toward the home tile
Defender	entities which the player can place; these do damage to all attackers within range

Table 1: A table with all entities in the game

4.2 Constructive Generator

In order to appropriately measure the effect of an evolved curriculum versus the impact of increased access to additional training points, we design the constructive generator. With access to a set of underlying parameters of the data, each with a set of acceptable values, and a global set of constraints, there exists a constructive generator that produces unbiased random samples of this data by simply permuting over possible combinations of parameter values, and simply throwing away those combinations that do not satisfy the constraints. Training with such a generator (which we refer to as our "constructive" network) is analogous to the undirected sampling case where the training data is fixed.

In *Attackers and Defenders*, the constructive generator is given the available tile types and where they can be placed (i.e. parameter values and set of possible values), along with the constraints presented in Table 2, and simply selects random combinations of tile values, outputting only those combinations that satisfy the constraints and discarding the rest.

4.3 Evolutionary Generator

Our system uses the Feasible Infeasible 2-Population (FI-2Pop) genetic algorithm (**?**) to evolve boards. FI-2Pop is an evolutionary algorithm which uses two populations: a feasible population and an infeasible population. The infeasible population aims at improving infeasible solutions to "legally-playable" threshold, when they become feasible and are transfered to the feasible population. The feasible population, on the other hand, aims at improving the quality of feasible chromosomes. If one becomes infeasible, it is then relocated to the infeasible population. After evolving solutions for several generations, the system outputs the board with the highest fitness.

Chromosomal Representation, Crossover, and Mutation A board chromosome is represented as a 2-dimensional array of tile types. Crossover (Figure 3) is done using 2-d array crossover, by picking a sub-array within one parent and swapping it with the other, creating a new board as a result. Mutation is done by selecting a random tile and changing its type. Mutation may be performed multiple times on a single board after crossover is completed.

Evaluating Feasibility and Fitness Each board chromosome contains two fitness functions which determine where they fail in terms of feasibility and fitness. The *constrained* fitness dictates whether they are within the infeasible population, and the *feasible* fitness ascertains how optimal of a board it is.

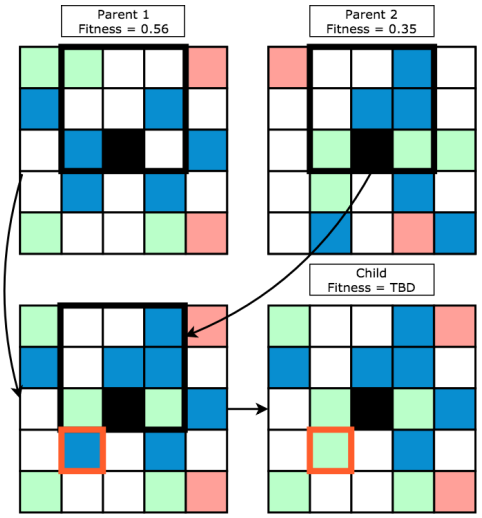

Figure 3: The 2D array representation of an *Attackers and Defenders* board. Crossover is shown, using Parent 1 as a template and Parent 2 as a replacement sub-array (black outline). A tile from resulting board is then mutated (red outline), with the fitness of the new child to be calculated later.

Factor	Description
Separate Quads	% of *sources* in different board quadrants
Home Paths	% of *sources* that initialize with a path to *home* tile
Home Center	1 if *home* is near center of board, else 0
Home Blocks	1 if no *blocks* near *home*, else 0

Table 2: The constraint factors present in the generator

Constrained fitness is calculated by averaging the constraint factors listed in Table 2. If the constrained average is 1, then this chromosome is feasible. *Feasible fitness* is measured by calculating the loss from the agent's loss network on the given map. The larger the loss, the higher the feasible fitness for the chromosome.

4.4 Procedure

To evaluate the effectiveness of an evolutionarily-based curriculum inside a reinforcement learning training loop, we created several training schedules. A Double Dueling Deep Q Network (**?**) (DDDQN) using prioritized replay and a separate loss network was trained from initialization on each schedule. Figure 4 displays the architecture of this network, and Figure 6 displays the separate loss network.

The replay bank of the DDDQN holds $20,000$ training experiences using hyperparameters $\alpha = 0.6$, $\beta_0 = 0.4$ annealed to $\beta = 1.0$ over the first 1000 games. To create experiences for the replay bank, the network plays a map from *Attackers and Defenders*, after which it stores all encountered

Network	Curriculum
DQN 1	50 + 100% randomly constructed maps
DQN 2	50 + 100% evolutionarily-curated maps
DQN 3	50 + 50% randomly constructed maps & 50% evolutionarily-curated maps

Table 3: The networks and their training curriculum ratios

initial states, actions, next states, and rewards as experiences in the bank. The network updates its weights every 5 maps it plays. A training cycle consists of 250 batches which contain 32 samples selected according to prioritized replay. The Q-value update uses a future discount factor $\gamma = 0.99$. The loss of each individual experience is used to update the priority. Afterwards, the loss network is trained using the initial state as input and the loss as the target.

All schedules begin with 50 maps created using a constructive generator to train the loss network such that its output is usable to assess a maps loss potential. This constructive generator provides an undirected random sampling of the game space. The 50 starting maps are used are the identical for every schedule. After these initial maps, the schedules differ. The first of these schedules continues to contain only maps constructed by the constructive generator. The second of these schedules contains only evolutionarily-curated maps. The third network contained equal mix of randomly generated maps and evolutionarily-curated maps. Table 3 defines each network's training curriculum.

For each schedule, the network was tested and scored on a fixed set of 1000 randomly generated maps after every 200 training maps. The network was optimizing to slay the maximum amount of attackers over the course of play and was scored based on how many of these attackers were slain before an attacker reached the home tile. Training continued until the network failed to improve for two consecutive testing cycles.

5 Results & Discussion

Here, we present the results of the previously described case study. We compare the results of the fully-evolved-curriculum-trained network (full network), the mixed-evolved-curriculum-trained network (mixed network), and the randomly-generated-curriculum-trained (constructive network) trained on randomly sampled maps.

As Figure 7 demonstrates, the full network generalizes well within 400 maps of training with a score of 21.47, peaking after just $1,600$ maps at 22.14. In comparison, the constructive network never reaches this score. Even after training on $6,800$ maps, it only peaks at 20.83 at $1,600$ maps. The mixed network peaked at 20.22 after $2,000$ maps which is slightly below the constructive network's peak.

Figure 5 displays the loss of each network. Each tick displays the average loss of from a training cycle containing 250 batches, out to a total of $3,500$ maps. The full network starts out at 14.44, higher than the constructive network at 12.67. This suggests that the full network learning was presented with maps which high learning potential, as expected. Very quickly the two networks converge and hover between 7.5 and 9 which corresponds the full network's peak performance within the first several hundred maps. The constructive network gradually increases over time to 10 then stabilizes after it reached peak performance. Contrary to expectations, the mixed network on the other hand shows a much higher loss than either of the other two networks starting at 40.95 and remaining substantially higher ranging from 10 to 14.

This suggests that the mixed network failed to generalize when presented with an equal mix of evolved maps and generated maps. The network appears to have learned how to discriminate evolve maps from randomly generated maps in a manner that harmed performance on the test set. Specifically, the network learned two classes of strategies: evolved and constructive. As a result, the learning from evolved examples would not generalize correctly to constructive maps that were used in the testing set. By contrast, the full network only saw evolved maps after the first 50 maps and was able to generalize the strategies learned from evolved maps to randomly generated maps in the test set.

Lastly, the sole requirement of the evolutionary generator is to specify the parameters of the game itself and is generalizable. The network informs the generator of its loss function which makes it not specific to any domain. It is sufficient for the network set up to have the ability to interact with the game as the architecture is not dependent on it. Since both halves of the systems (generator and network) are generalizable themselves, slight modifications tot he system can make it adapt very well to a new scenario.

6 Conclusion

In this paper, we have introduced evolutionarily-curated curriculum learning as a new methodology to train reinforcement agents. We performed a case study using a game we created called *Attackers and Defenders* to prove the validity and effectiveness of this new method. Specifically, we tested a Double Dueling Deep Q-network (DDDQN) with prioritized replay and a separate loss network using this method.

Based on our results, our initial hypothesis that evolutionarily-curated curriculum learning helps networks generalize better and faster than undirected sampling, has been proven true in this environment. Even after nearly three times the amount of training time, the constructive trained network never approaches the performance of the full evolutionarily-curated network. Therefore, it appears that this new training methodology, ECCL, can be used to both expedite training and increase generalization or max performance.

However, the mixed network does not appear to perform as well despite the fact that its loss values are much higher. This was a result we did not initially expect, allowing for any amount of evolutionarily-based curriculum learning would improve network training. Upon inspection, the mixed network spent considerable effort in differentiating between maps coming from the constructive generator vs. the evolutionary generator. This suggests that a discriminator network trained to predict whether a map was constructed or evolved could be added to the evolutionary generator's fitness functions. Another possibility would be using a similarity metric in fitness to ensure evolved maps are sufficiently different

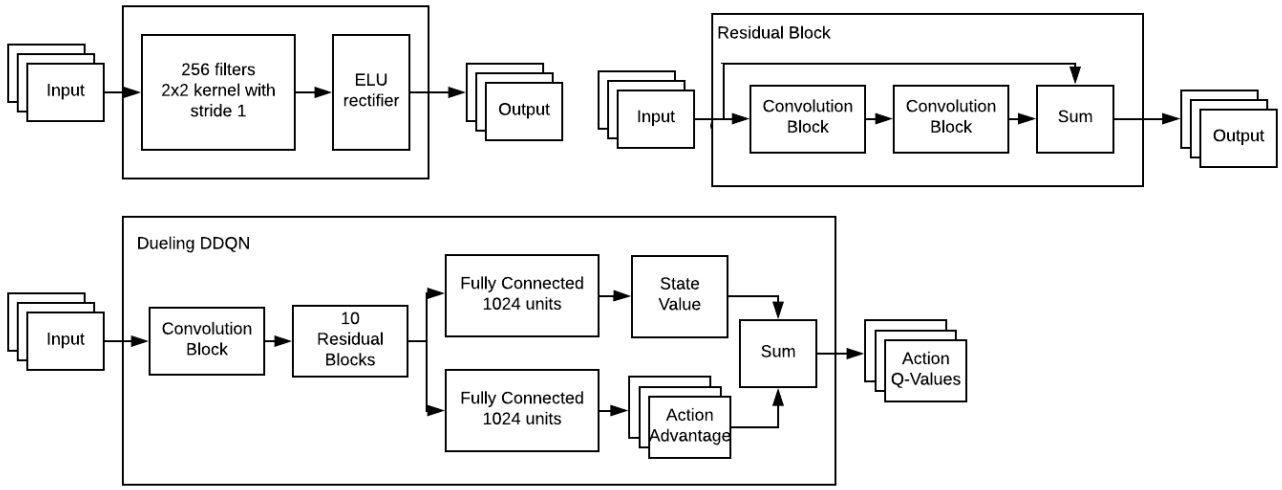

Figure 4: The architecture of our DDDQN consists of a convolutional layer followed by a tower of 10 residual blocks. The final residual block is fed to two separate fully-connected layers to produce the current states Q-value and the predicted advantage of each possible action. The streams are combined to produce predicted action Q-values.

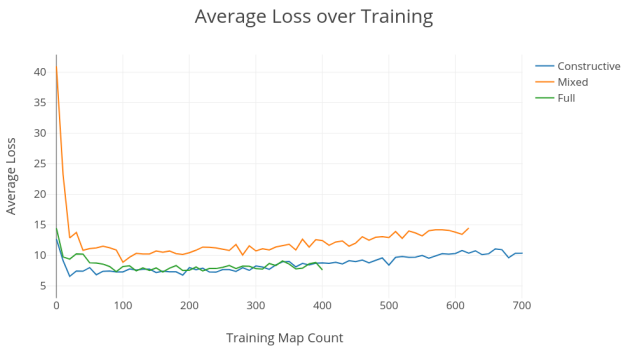

Figure 5: The loss over time for the training of each network. Loss was collected every 5 maps of training by averaging the loss across the 5 maps.

from previously evolved maps to prevent the network from learning to recognize evolved maps.

Given the generalizability of ECCL as in the discussion from the previous section, as it only requires a data generator and a game-playing agent architecture, we also expect it to work well with AlphaGo Zero-based agents as well, and leave that open for future work.

Quis iste a laudantium corrupti, rerum optio odio adipisci praesentium recusandae ex quidem atque in quos, quae vitae voluptate, dolore hic eos impedit molestiae inventore labore optio quisquam distinctio quaerat accusantium?Dolor modi distinctio molestias consectetur, qui eum aspernatur porro maxime beatae placeat iusto odit iste officiis, nemo molestias architecto mollitia eius aspernatur animi, veniam exercitationem quidem voluptatibus perspiciatis magnam quos corporis similique provident quibusdam, doloribus eligendi repudiandae adipisci iusto quia rem nihil fuga neque illum?Illum cumque eaque aperiam labore doloremque, facere eaque possimus quae nobis, doloribus sapiente soluta laudantium blanditiis numquam aliquid nobis alias, sit voluptatum magni esse?Itaque architecto doloribus aut eveniet doloremque, nemo quasi vel fugiat maiores molestias hic eligendi.Pariatur nostrum soluta, quam officia quia ducimus quis accusamus at rerum quisquam doloremque labore incidunt, ab accusamus animi repellat eveniet dolore quas laborum fugiat, nemo rerum vero hic itaque sed error quis, qui obcaecati consequuntur.Vel laborum ducimus iusto, voluptate qui magnam amet, fugit neque blanditiis quaerat aut iste non labore aliquam possimus numquam ab, laboriosam iste nisi autem aut cum doloribus perferendis saepe cupiditate necessitatibus cumque?Ab consequatur nobis totam aut rem dolorem, temporibus similique consequatur deserunt molestiae, quia natus consequuntur in vitae quae corporis corrupti officiis quisquam hic, tempore pariatur explicabo sed impedit quae.Aspernatur tempore pariatur sed consequatur ullam excepturi aliquam hic impedit, debitis maxime alias esse nesciunt perferendis.Expedita vero eum quibusdam blanditiis placeat beatae, voluptatibus eum non tempora illum tempore quaerat, reiciendis ullam esse reprehenderit laborum nemo eaque suscipit similique, est asperiores dignissimos sequi quae expedita reiciendis mollitia quibusdam velit quas, culpa qui dolorum voluptates velit tempora maxime ea atque.Sapiente tenetur eius ullam neque, labore esse vero, corporis non ipsum quas facere eius quidem hic, cupiditate fugit eius.Quaerat saepe maxime quasi deleniti perferendis quia consectetur excepturi reiciendis eveniet corrupti, molestias adipisci corporis omnis odio illum quidem qui,

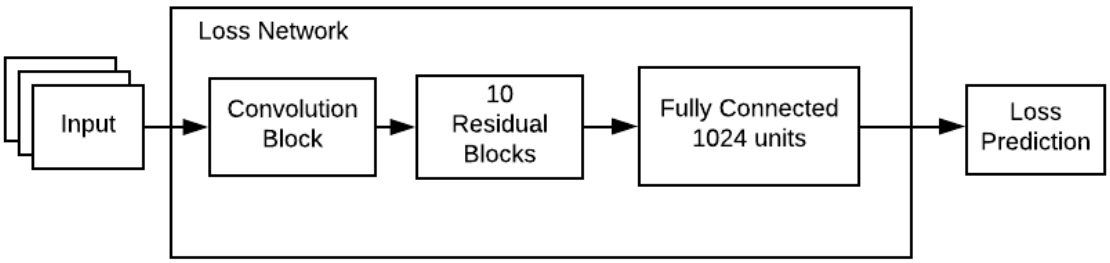

Figure 6: The architecture of our loss network consists of a convolutional layer followed by a tower of 10 residual blocks which feeds a fully connected layer that outputs the loss prediction.

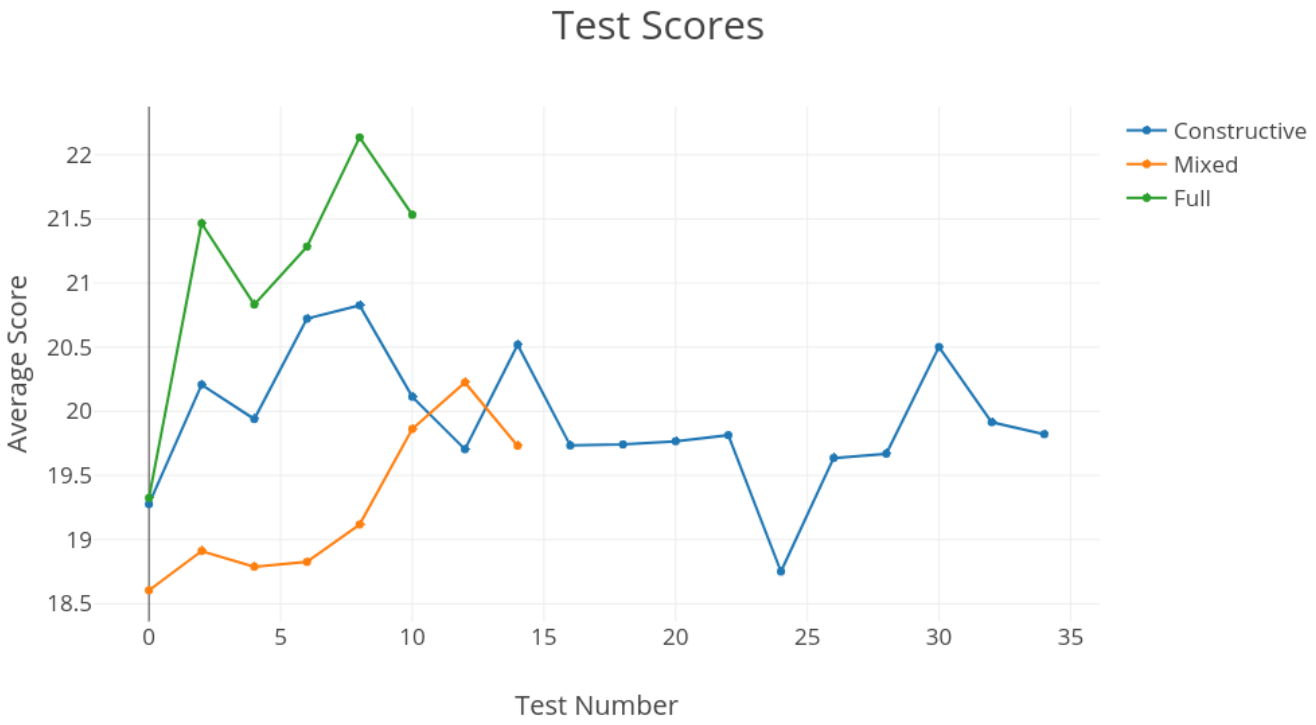

Figure 7: The testing score results averaged in $1,000$ map batches over the course of training. The y-intercept marks network performance at network initialization with randomized weights.

voluptas reiciendis deserunt dolore excepturi odio molestiae eius quam aliquid corrupti corporis, fugiat aperiam obcaecati perferendis quis quibusdam, quisquam voluptate voluptatibus accusantium sapiente quae doloremque enim.Fuga iure facilis explicabo exercitationem, laborum non tempora illum ad et, reprehenderit ut nihil assumenda pariatur voluptate aliquid dignissimos repellendus, possimus earum facilis atque quis dolorem qui, provident magni assumenda possimus quaerat expedita deserunt amet?Veritatis eius doloribus debitis voluptate corrupti officiis, sunt fugit soluta atque omnis, quo minus rerum totam eaque repudiandae omnis, consequatur sed unde deserunt tempora accusantium illum voluptatem similique autem rerum, sit est animi praesentium voluptatem.Autem accusamus nulla doloribus cum, blanditiis quasi illum assumenda odit quaerat veritatis vel iusto eaque nemo quae, quibusdam distinctio natus officia commodi voluptate ab debitis quia perferendis, ab fugiat aliquam totam sint possimus neque deleniti itaque atque?Unde aperiam consectetur ipsam nemo consequuntur laudantium earum velit amet ullam, iusto quas