EVOLUTION STRATEGIES AND REINFORCEMENT LEARNING FOR A MINESWEEPER AGENT

Jacob J. Hansen (s134097), Jakob D. Havtorn (s132315),
Mathias G. Johnsen (s123249), and Andreas T. Kristensen (s144026)

Technical University of Denmark

ABSTRACT

In recent years, reinforcement learning has seen a resurgence, in part thanks to the development within deep learning. These methods are poised to revolutionise the field of artificial intelligence and represent a step towards building autonomous systems with a higher level of understanding.

In this paper, we explore the use of three of these methods; policy gradient, Q-learning, and evolution strategies to solve the classic Minesweeper game. Our results show that both the policy gradient and Q-learning methods are able to solve the game in a relatively short window of time, and both methods obtain relatively high win-rates on a $\mathbf{6 \times 6}$ board with 1-12 mines. Moreover, the results show that the implementation using the evolution strategy is able to obtain increasing reward during training, however, it never learns to win the game.

Index Terms— Deep reinforcement learning, Policy gradients, Q-learning, Evolution strategies, Minesweeper

1. INTRODUCTION

The many recent successes in scaling reinforcement learning (RL) to complex sequential decision-making problems were kick-started by the Deep Q-Networks approach (DQN) [1]. This is based on classical Q-learning, where an agent learns an action-value function, which gives the expected utility of taking an action a in a given state s, and following an optimal policy thereafter. For DQN, a deep neural network (DNN) is used to estimate the Q-function.

The policy gradient method is another approach to reinforcement learning which has also seen benefits from using neural networks for estimating optimal policy functions [2]. Policy gradient is an *on*-policy method as it finds the policy directly by evaluating the current policy network in each training iteration. This is different from Q-learning methods, an *off*-policy method, as it learns the value of the optimal policy independently of the agent's actions.

Another recent success is the application of evolution strategies to the classical reinforcement learning setting [3]. This approach randomly permutes policy network parameters of a number of agents to estimate the gradient. It benefits in time complexity from being highly parallelizable.

For the evaluation of these algorithms, a popular approach is to test their ability of learning certain games, such as classical Atari 2600 games [4]. In this paper, we have chosen to focus on solving Minesweeper, a classic puzzle video game developed in the 1960s. Minesweeper is an interesting problem, since solving it is NP-complete [5]. Listing all possible configurations is equivalent to listing all possible solutions to an NP-complete problem. So the challenge faced by the artificial intelligence (AI) is to approximate the probability that a particular tile has a hidden mine and take appropriate future steps based on this.

We apply these three methods on the Minesweeper game in order to determine how effective each of the methods is at solving the problem. We evaluate in terms of average win-rate, the number of training steps for the training method and the number of steps taken in the environment.

The paper is organized into six sections: Section 2 describes work related to this paper. Section 3 details the methods used in the paper to obtain the results. Section 4 presents the experiments performed and the results obtained. Finally, we conclude the paper in Section 6.

2. RELATED WORK

Previous attempts at training a reinforcement learning agent to play minesweeper [6, 7] have used Q-learning in both the classical and neural network versions. Neither of these approaches achieve more than 5% win-rate on 6×6 boards with varying numbers of mines.

Interestingly, [7] also implement a simple constraint based approach called the *Constraint Satisfaction Problem* (CSP). Each tile is given a value of 1 if it contains a bomb and 0 if it does not. Then, each tile with a revealed number of neighbouring bombs is used to define a constraint: The sum of all eight tiles surrounding the revealed tile must equal the number of the tile. The solver then looks for solutions to a large set of these constraint equations. This often leads to very certain actions, with the uncertain actions governed by computed probabilities based on the constraint equations. The CSP solver achieves 84% win-rate on an 8×8 board with 8 mines and 90% win-rate on a 4×4 board with 3 mines.

3. METHODS

3.1. Q-Learning

Optimally, an agent performs actions which maximizes the discounted future reward $R_t = r_t + \gamma R_{t+1}$ [8]. We define the optimal action-value function $Q^*(a, s)$ as the maximum expected reward achievable by following a policy π after seeing some sequence s and then taking action a,

$$Q^*(s, a) \equiv \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi] \,.$$

The optimal policy is then defined as

$$\pi^*(s_t) \equiv \arg\max_{a_t} Q^*(s_t, a_t) \,.$$

To actually learn the Q-function, we exploit the assumed Markov property of the environment and use the definition where the optimal Q-function follows the Bellman equation

$$Q^*(s_t, a_t) \equiv \mathbb{E}_{s_{t+1}}\big[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s_t, a_t\big] \,.$$

In practice, we use a DNN with parameters θ as a function approximator of the action-value function, $Q_\theta(s, a) \approx Q^*(s, a)$ [4].

A Q-network can be trained by adjusting the parameters θ_i at iteration i to reduce the mean-squared error in the Bellman equation. The loss is then given as

$$L_i(\theta_i) = \big[\underbrace{r_t + \gamma \max_{a_{t+1}} Q_{\overline{\theta}_i}(s_{t+1}, a_{t+1})}_{Q_{\text{target}}} - \underbrace{Q_{\theta_i}(s_t, a_t)}_{Q_{\text{predict}}}\big]^2 \,.$$

The gradients of the loss are back-propagated only into the *training* network with parameters θ (which is also used to select actions during training). The term $\overline{\theta}$ represents the parameters of a *target network*; a periodic copy of the training network which is not directly optimized. Using separate networks to compute the loss stabilizes the training [4].

During training, the networks ability to predict the utility of performing actions in a state is evaluated. Q-learning is thus an *off*-policy method since it learns the value of the optimal policy independently of the agent's actions.

However, the classical deep Q-learning strategy introduces an overestimation bias which can harm learning. The `max` operator uses the same values to both select and evaluate an action. This can therefore lead to an overoptimistic value estimate. Using *Double Q-learning*, we mitigate this problem by decoupling the selection from the evaluation [9]. The target Q-value is then

$$Q_{\text{target}} = r_t + \gamma Q_{\overline{\theta}_i}\big(s_{t+1}, \arg\max_{a_{t+1}} Q_{\theta_i}(s_{t+1}, a_{t+1})\big) \,.$$

Another architectural choice is the use of duelling networks. Here, the Q-function is decomposed into two meaningful functions, the value function V and the advantage function A,

$$Q(s_t, a_t) = V(s_t) + A(s_t, a_t) \,,$$

where $V(s_t)$ denotes the value of being in a given state and $A(s_t, a_t)$ the advantage of an action in this state. To accommodate this, the network is split into two streams in its latter part, one estimating the value and the other estimating the advantage, finally, the streams are combined to produce a single output Q-function.

3.2. Policy Gradients

With policy gradients, instead of learning the utility of each state-action pair, we directly learn a parameterized decision policy [2]. This policy is similarly encoded using a DNN and trained by optimizing the weights with respect to the expected return of applying the current policy to the environment. As such, it is an *on*-policy method which constantly executes the current policy on the environment to get information about the states, actions and rewards. From these, it finds the gradient of the weights and updates these using gradient descent. The approach used in this work is a vanilla policy gradients implementation with the gradient estimator

$$\nabla_\theta \mathbb{E}[R|\theta] \approx \frac{1}{V} \sum_{v=1}^{V} \nabla_\theta \log p_\theta(\mathbf{a}^{(v)}|\mathbf{s}^{(v)}) A(\mathbf{a}^{(v)}) \,.$$

where V is the number of roll-outs performed and A is the advantage. The advantage is computed using the discounted rewards and the baseline,

$$\begin{aligned} A_t &= R_t - b_t \,, \\ b_t &= \frac{1}{V} \sum_{v=1}^{V} R_t^{(v)} \,, \\ R_t &= r_t + \gamma R_{t+1} \,, \end{aligned}$$

using γ as the discount factor, determining how far into the future we look.

The overall idea is that the performed action on a state will return a reward. Based on the size of this reward and the size and direction of the gradient, the individual parameters in the network are updated such that large rewards lead to large changes in the direction determined by the gradient. Over time, this strategy increases the probability of actions expected to lead to large rewards and decreases that of actions expected to lead to small rewards. Exploration is implemented by sampling from the action space probabilities produced by the policy. As the training converges, these probabilities will concentrate around the optimal action, reducing the probability of sampling unlikely actions and thus also reducing the amount of exploration of the algorithm. This of course introduces the risk of converging on a local optimum.

3.3. Evolution Strategies

Evolutionary optimisation strategies (ES) are a class of black box optimization algorithms that numerically estimate the

gradient of the objective function using stochastic techniques. Some variants of the methods are highly parallelizable and have been shown to be applicable to and competitive within deep reinforcement learning (DRL) [3].

The strategy used in [3] relies on the score function estimator used in natural evolution strategies

$$\nabla_\psi \mathbb{E}_{\theta\sim p_\psi}[F(\theta)] = \mathbb{E}_{\theta\sim p_\psi}[F(\theta)\nabla_\psi \log p_\psi(\theta)]\,,$$

where $p_\psi(\theta)$ is a distribution over parameters, θ, parameterized by ψ while $F(\theta)$ is the objective function parameterized by θ.

By taking $p_\psi(\theta)$ to be an isotropic multivariate Gaussian and performing a change of variables the score function estimator becomes

$$\nabla_\theta \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}[F(\theta)] = \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}[F(\theta+\sigma\epsilon)\epsilon]\,,$$

where only the optimization over the parameters θ, corresponding to the mean of the Gaussian, is considered and optimization over the variance is ignored.

The resulting algorithm instantiates a number of agents with certain permutations of the 'parent' policy network given by $\theta + \sigma\epsilon$. The noise is drawn from a zero mean isotropic Gaussian, $\epsilon \sim \mathcal{N}(0, \sigma^2)$ with σ fixed before training. The fitness is then evaluated for each of the agents, $F_i = F(\theta + \sigma\epsilon_i)$ and from Equation 3.3 the gradient estimate becomes

$$\theta_{t+1} = \theta_t + \frac{\alpha}{n\sigma}\sum_{i=1}^{n} F_i\epsilon_i\,.$$

The method can be made more robust by a few extensions. Fitness shaping, replaces rewards by a fitness ranking based only on relative values of the rewards. This removes dependency on the numerical size of rewards and improves learning in early stages of training. The specific utility function to compute the fitnesses is the one used in [10]. Weight decay regularization was implemented as a penalty on the rewards and tends to rank agents with small weight norms higher. Antithetic sampling adds the policy permutation $\theta - \sigma\epsilon$ to the set and serves to reduce the variance of the gradient estimate.

Contrary to [3], our strategy did not implement virtual batch normalization and did not use an adaptive optimizer, such as Adam [11], although it may benefit from such.

4. EXPERIMENTS

4.1. Minesweeper

The practical aspect of this project has been focused on solving the classical game of Minesweeper. In Minesweeper, the agent must clear an $n \times m$ grid with k mines by activating each coordinate once. The bombs are randomly scattered after the agent clears the first field. The agent wins if it clears all fields except the bombs and loses by striking a bomb.

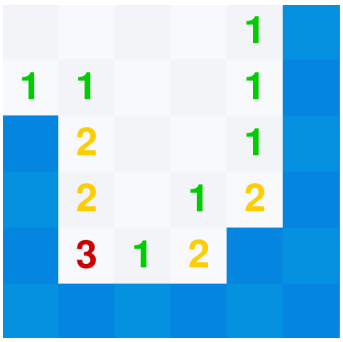

Fig. 1. Graphical rendering of the Minesweeper environment.

The environment was implemented to allow control over the reward structure and behaviour. A graphical rendering of the environment is shown in Figure 1 with a 6×6 board with 6 hidden mines.

All unknown fields which are not bombs must be revealed to win the game. Thus, revealing an unknown field can be seen as making progress towards winning the game. Since all fields *must* be revealed, every move that makes progress are equally good since they all have to be made to win the game. Since actions are completely decoupled, it makes sense to get a reward after every interaction with the environment.

The following reward structure was used in this project: {"*win*": 1, "*loss*": -1, "*progress*": 0.9, "*noprogress*": -0.3, "*YOLO*": -0.3}. *Progress* is clearing an unknown field that is not a bomb, *noprogress* is clicking on an already revealed field (does not make progress towards winning the game) and *YOLO* is for clicking a field which has 0 revealed neighbours. The reason for the YOLO-reward is, that as the density of bombs drops, the agent will figure out that it can get the progress reward by clicking a random unknown field with little chance of hitting a bomb. Clicking a completely random field is a gamble that often pays off in regards to making progress, but is very unlikely to result in winning the game - therefore the negative reward.

Our implementation is able to return the current game state in 3 different ways:

- FULL: 1-hot encoded $N \times M \times 10$ matrix. The first 8 channels contain the different integers, one channel for each, the 9th channel has 1 if the field is empty and the 10th channel has 1 if the field is unknown and 0 otherwise.
- CONDENSED: An $N \times M \times 2$ matrix. The first channel is the integers, second channel has 1 if the field is unknown, 0 otherwise.
- IMAGE: $N \times M \times 1$ matrix. Only a single channel holding an integer or "colour" of a field. For our experiments, the two other representations where much better.

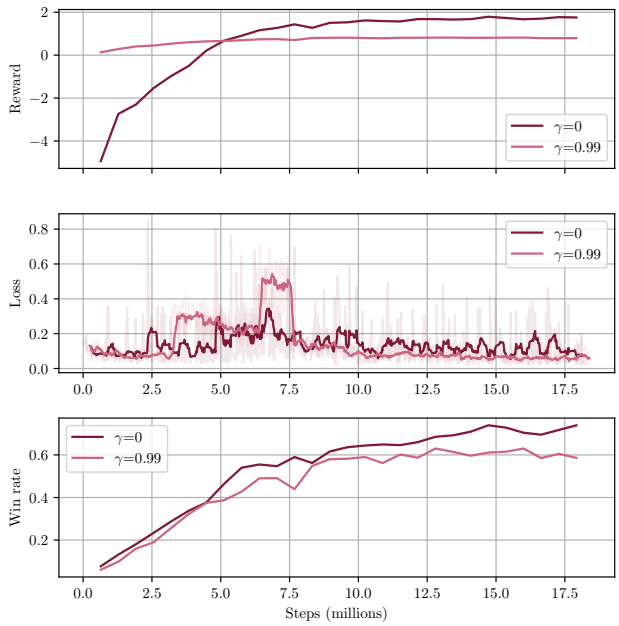

Fig. 2. Average reward and win-rate for model evaluation during training and the training loss for the Q-learning agent using $\gamma = 0$ and $\gamma = 0.99$.

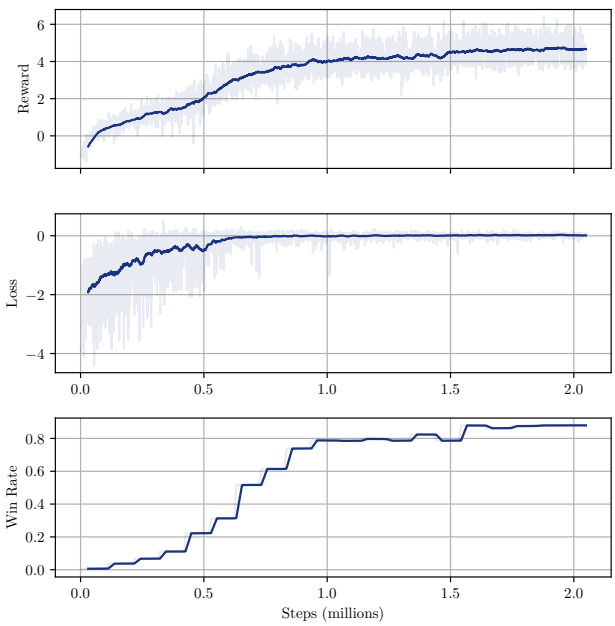

Fig. 3. Average reward and win-rate for model evaluation during training and the training loss for the policy-gradient agent.

4.2. Hyperparameters

The values of all hyperparameters and optimization parameters were selected using a heuristic approach. For Q-learning, most of the parameters are based on those given in [12] with some modifications following the same heuristic approach.

4.3. Q-Learning

During training, the agent selects an action ϵ-greedily, with ϵ being annealed linearly from 1 to 0.1 over 250,000 iterations during the training to initially explore the environment. For each interaction with the environment, we add a transition (s_t, a_t, r_t, s_{t+1}) based on the current state to a replay memory buffer that holds the last million transitions [13]. With experience replay, it becomes possible to break any temporal correlations by mixing more and less recent experiences for the updates.

In Figure 2, we show the results for Q-learning with $\gamma = 0$ and $\gamma = 0.99$. This allows us to get some insight into the agents ability to use a future reward in the minesweeper game. Both of the models described use duelling.

As shown, the models initially have the same win-rate, on 1000 games at each evaluation. However, the agent with $\gamma = 0$ achieves higher win-rates after more training. We also note that the average reward is quite high for $\gamma = 0$ for the first evaluations, however, this does not mean that it can actually win any games, as seen from the win-rates.

Testing with other network architectures and other parameters, it was determined that it was best for Q-learning to set $\gamma = 0$. That the discount-rate can be set to zero without impairing the learning process resonates well with the very low temporal correlations between the states of Minesweeper.

Moreover, it did not seem like duelling was a significant benefit for the Q-learning agent. So, for the optimal results presented in Section 4.6, the network architecture used is the same as that used for the policy gradient agent, with no duelling and $\gamma = 0$.

4.4. Policy Gradients

The policy of the agent is parametrized by a neural network using the condensed representation as input followed by two convolutional layers (5x5, 3x3 kernels with 18 and 36 filters respectively and no max-pooling) and three dense layers with 288, 220 and 220 hidden units with the last connected to the output layer of size $N \times M$. Since the representation contains perfect state information (i.e we are looking directly at the state and not just an image of the game) no max-pooling is used to preserve all information. The algorithm has been tested using both the FULL and the CONDENSED representation. Both representations were found to converge to the same win-rate, however, the CONDENSED representation resulted in significantly faster learn convergence. In Figure 3 the mean reward, loss and win-rate are shown as the train-

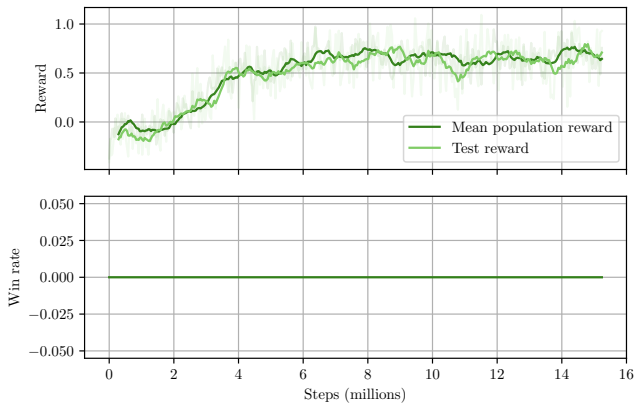

Fig. 4. Example of learning progression of ES on Minesweeper. The policy improves at first and sees the test and average reward of the permuted population follow each other closely. It, however, stalls after about 6 million steps.

ing progresses. After around 1.6 million steps the win rate converges to 89%, which is much faster than the Q-learning approach. The Adam optimizer had a hard time adjusting the learning rate. It is therefore initially set to 0.001 and dropped two times during training, first to 0.0001 and then to 0.00002, whenever the win-rate would start to converge. It took around an hour to train on a modern laptop with a graphics card. As was found for Q-learning, setting the discount factor to 0 gave the best results.

4.5. Evolution Strategies

It proved difficult to master Minesweeper using ES. An example of the learning process on Minesweeper is shown in Figure 4. The mean reward of the population is computed as the mean reward of all agents of a generation evaluated on a single episode. This statistic is accumulated during training. The mean test reward is computed periodically by testing the current policy on 50 episodes.

Different network architectures were tried out including fully dense networks and networks with convolution layers. Generally, the convolutional networks showed marginally stronger learning early on but reach a learning plateau as the dense networks. Both Minesweeper representations were tried yielding similar results.

All networks and representation combinations were tried with different weight decay rates including no weight decay, different learning rates and different numbers of agents.

Generally, the learning progression exhibited by ES on Minesweeper was to initially improve the test reward but fairly quickly stall. This behaviour was attempted alleviated by implementing antithetic sampling and weight decay. Unfortunately, these additions did not improve learning on Minesweeper, although stabilization was observed for CartPole. Virtual batch normalization was not implemented but could potentially improve performance further.

ES did however successfully learn other environments such as CartPole. An example of the learning progression for this is seen in figure 6 in the appendix.

Table 1. Win-rate on Minesweeper for the different strategies on a 6×6 board with 6 mines. Q-learning and policy gradients best the previous Q-learning results by a large margin, perform slightly better than human level and match the performance of the CSP solver.

Authors	Model				Human
	ES	Q-Learning	Policy	CSP	
This paper	0%	$\mathbf{90.20\% \pm 0.30\%}$	$89.6\% \pm 0.31\%$	–	$87\% \pm 4.5\%$
Honnungar [6]	–	$< 5\%$	–	84 to 90%	–

4.6. Comparison

In Table 1, the win rates for the best models of each applied methods are listed and compared to the results of [7].

Our Q-learning results best the previous Q-learning results by a large margin achieving a win rate of 90.2% on 10000 games for the final evaluation.

The policy gradients approach achieves a similar level of performance with a win rate of 89.6%. Both of the reinforcement learning approaches perform match the performance of the CSP solver. Furthermore, both methods outdo the performance of an author of this paper, averaged over 100 games.

To obtain the best results for Q-learning, we train for 440000 batch updates. Each update uses a batch size of 400 state-action transitions in the environment sampled from the replay memory (initialized with 80,000 transitions). Before each new batch-update, a single new transition is taken in the environment and pushed into the memory. This amounts to about 520000 steps in the environment. Similarly, policy gradients trains for 12000 batch updates with batches of 200 environment steps amounting to 2.8 million environment steps.

So, using a replay memory, Q-learning is more data efficient than policy gradient and this in fact the case here. However, Q-learning may not benefit as much from the replay memory when playing Minesweeper in terms of removing temporal dependencies as it does in other environments.

An evaluation of the performance for the Q-learning and policy gradient methods for different numbers of mines on a 6×6 board is shown in Figure 5. For Q-learning, we have two trained agents, one trained on 6 mines, just as the policy gradient agent, and one trained on a random number of mines from 1-12. The results show that the agent is actually able to overfit to the board configuration with 6 mines. Therefore, another Q-learning agent was trained with a random number of mines. The results show that the agent can in fact perform

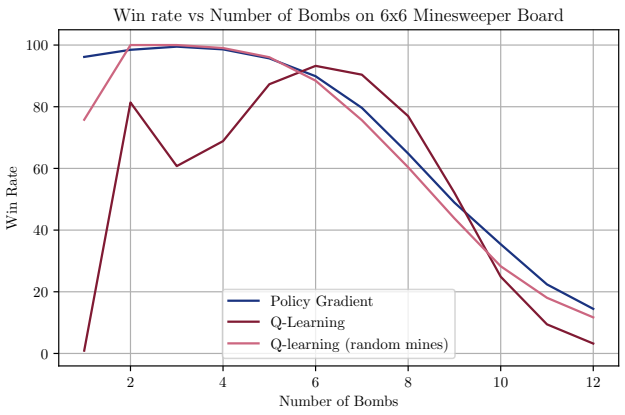

Fig. 5. Comparison of the performance of the trained agents with different number of mines. For Q-learning, two models are plotted. One model which was trained on 6 mines and another which was trained with a random number of mines in the range $[1, 12]$. The network trained with an alternating number of mines performs better.

better with fewer mines on the board. Compared to the DQN agent, the policy gradients agent was only trained on boards with 6 mines. It seems to generalize better to boards with different numbers of mines without having actually been trained specifically on these.

5. FUTURE WORK

This section focuses on potential future work on evolutionary computing within deep reinforcement learning as well as the solution in general.

5.1. Evolution strategies

ES is in its tentative beginnings when it comes to reinforcement learning applications. As such, there are many possibilities for methodological developments that can improve their performance.

A primary source of inspiration should be the vast literature on evolutionary computing for conventional gradient free optimization. This includes natural evolution strategies [10] and the well-known CMA-ES algorithm [14]. Additionally, recent developments of CMA-ES [15].

Another interesting approach is that of variational optimization [16], where a differentiable bound on the objective function replaces the non-differentiable objective. In fact, the simple evolution strategy implemented here can be derived as a special case in the variational setting, see e.g. [17].

Finally, a recent set of publications [18, 19, 20, 21, 22] demonstrates the applicability of genetic algorithms (GA) to train DNNs providing details for how to achieve better exploration using novelty search [18], performing more efficient mutation by safe mutation through gradients (SM-G) [19]. Additionally, in [21] it is shown that solutions found by ES are robust to parameter pertubations compared to TRPO and GA solutions.

5.2. Scaling to larger boards

Generally, the principles learned by a human Minesweeper player apply to arbitrary board sizes. For the agents considered here however, increasing the board size will change the input and output dimensions so a new agent would have to be trained with a different network architecture.

To combat this scaling problem we can observe, that a human does not solve large boards by looking at all the fields at once, rather he or she will tend to solve a subsection of the board. This approach could be used for RL agents, i.e train them on solving a subboard of e.g 6x6 and then use this agent to stride over a larger board and take actions for the different sub boards encountered to solve the puzzle. Alternatively, transfer learning or weight sharing could be applied in different ways to the layers shared between different networks for board sizes.

5.3. Hyperparameters

Hyperparameters were determined heuristically. Rather large networks are used and generally the loss is noisy. Thus, optimizers such as Adam have a hard time accurately predicting the moments of the gradients. Given larger computational resources it would be possible to run many training sessions in parallel giving the possibility of performing a hyperparameter grid search in a structured way. Changing hyperparameters this way, could possibly yield faster convergence in training.

6. CONCLUSION

In this paper, three reinforcement learning methods have been implemented and trained on the game of Minesweeper. Deep Q-learning and policy gradients both learn to win the game, achieving win-rates around 90% on 6×6 boards with 6 mines. Both methods demonstrate the ability to generalize to different numbers of mines with Q-learning performing as well as policy gradients only when trained on an alternating number of mines. Additionally, both methods match the performance of the CSP solver. Evolution strategies showed promise but requires further development to be competitive on Minesweeper.

7. SOURCE CODE ACCESS

The source code for the project can be accessed and downloaded from our GitHub repository at `https://github.`

com/jakejhansen/minesweeper_solver. The Jupyter notebook Minesweeper_results.ipynb recreates the learning progress plots in this article by loading saved training statistics. The notebook further contains code for executing training of the three proposed agents.

8. REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[2] Andrej Karpathy, "Deep reinforcement learning: Pong from pixels," 2016.

[3] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] R Kaye, "Minesweeper is np-complete," *Mathematical Intelligencer*, vol. 22, no. 2, pp. 9–15, 2000.

[6] Sagar Honnungar, Sanyam Mehra, and Mohana Prasad, "Aim: Ai agent for minesweeper," 2016.

[7] Luis Gardea, Griffin Koontz, and Ryan Silva, "Training a minesweeper solver," 2015.

[8] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath, "A brief survey of deep reinforcement learning," *CoRR*, vol. abs/1708.05866, 2017.

[9] Hado Van Hasselt, Arthur Guez, and David Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, 2016, pp. 2094–2100.

[10] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber, "Natural evolution strategies.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 949–980, 2014.

[11] Diederik P. Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," 2017.

[12] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver, "Rainbow: Combining improvements in deep reinforcement learning," *arXiv preprint arXiv:1710.02298*, 2017.

[13] Long-H Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," 1992.

[14] Nikolaus Hansen, "The CMA Evolution Strategy: A Tutorial," 2009.

[15] Abbas Abdolmaleki, Bob Price, Nuno Lau, Luis Paulo Reis, and Gerhard Neumann, "Deriving and improving cma-es with information geometric trust regions," *Gecco 2017 - Proceedings of the 2017 Genetic and Evolutionary Computation Conference*, pp. 657–664, 2017.

[16] Joe Staines and David Barber, "Variational optimization," *arXiv:1212.4507*, 2012.

[17] David Barber, "Evolutionary optimization as a variational method," *Blog post at https://davidbarber.github.io/blog/2017/04/03/variational-optimisation/*, 2017.

[18] Felipe Petroski Such Joel Lehman Kenneth O. Stanley Jeff Clune Edoardo Conti, Vashisht Madhavan, "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents," *arXiv:1712.06560*, 2017.

[19] Jeff Clune Joel Lehman, Jay Chen and Kenneth O. Stanley, "Es is more than just a traditional finite-difference approximator," *arXiv:1712.06563*, 2017.

[20] Kenneth O. Stanley Xingwen Zhang, Jeff Clune, "On the relationship between the openai evolution strategy and stochastic gradient descent," *arXiv:1712.06564*, 2017.

[21] Jeff Clune Joel Lehman, Jay Chen and Kenneth O. Stanley, "Safe mutations for deep and recurrent neural networks through output gradients," *arXiv:1712.06567*, 2017.

[22] Edoardo Conti Joel Lehman Kenneth O. Stanley Jeff Clune Felipe Petroski Such, Vashisht Madhavan, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv:1712.06568*, 2017.

9. APPENDIX

9.1. Evolution strategies for inverted pendulum balancing

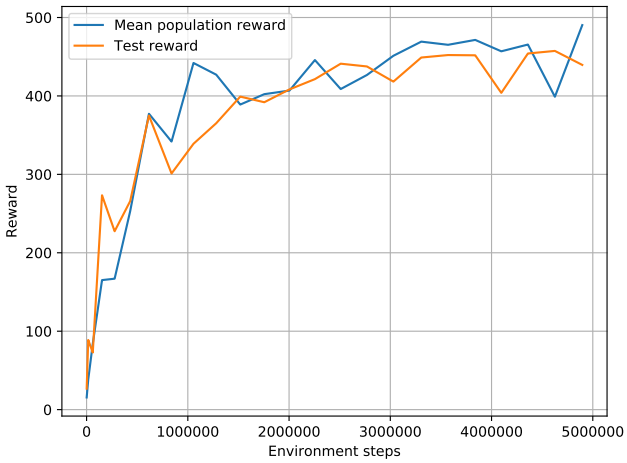

Fig. 6. Learning progression of ES on the OpenAI Gym CartPole-v1 environment. After 3-4 million steps in the environment (about 100 generations), the agent has learned the task. The comparatively large amount of steps is due to the lower utilization of information by ES due largely to its derivative free algorithm.