Exploring the Effectiveness of Klijn's Coevolutionary Evolution Strategy for Multi-Agent Reinforcement Learning in Atari's Joust

Bryce Anthony and **Jason Zheng**
{branthony,jazheng}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

Advancements in reinforcement learning have enabled agents to achieve expert-level performance in games such as Chess and Go. However, more complex games like Atari's Joust, which involve cooperative and competitive environments, require multi-agent reinforcement learning approaches. This research paper evaluates the effectiveness of Klijn's Coevolutionary Evolution Strategy, as detailed in (Klijn and Eiben 2021), for training agents to play Atari's Joust. The paper provides an overview of Atari's Joust and the strategy implemented from (Klijn and Eiben 2021), followed by a description of the experimental setup. The experiments use specified hyperparameters and neural network architectures, incorporating frame stacking techniques for preprocessing game images. The Coevolutionary Evolution Strategy is employed, involving population-wise and generation-wise mutations, to train the agents over multiple generations. The results show that, on average, the trained agents achieve higher scores than the Coevolutionary Evolution Strategy proposed in (Klijn and Eiben 2021). However, the random model performs better overall. The paper discusses the limitations and challenges faced during the training process, such as hardware constraints and limited training time. Despite the disparities in results, the Coevolutionary Evolution Strategy shows promise for multi-agent reinforcement learning in Atari's Joust and the broader field of multi-agent reinforcement learning.

1 Introduction

In recent years, advancements in reinforcement learning have led to the emergence of more sophisticated technologies. Researchers have used self-playing algorithms like Q-Learning, A-Star, and Markov Decision Processes to train agents to be capable of playing games like Chess and Go at an expert level. Although for games like Chess and Go, single-agent reinforcement learning has produced extremely capable agents. For more complex games like Atari's Joust, single-agent reinforcement learning will not suffice, as there exist two players that dwell in a cooperative/competitive environment. The presence of two players in this game means that each agent needs to be trained to work with the other agent thus leading to the use of multi-agent reinforcement learning.

Though this can be done in a variety of ways, (Klijn and Eiben 2021) uses a combination of deep learning and neuroevolution to train agents to play several Atari games including Atari's Joust. In this paper, we implement the Coevolutionary Evolution Strategy detailed in (Klijn and Eiben 2021) to train our agents to play Joust just as, if not more, effectively. In the rest of this paper, we give a brief overview of Atari's Joust game and the strategy implemented from (Klijn and Eiben 2021). We then go over our experimental setup and discuss our results as well as the broader impacts of our work in the subsequent sections of our paper.

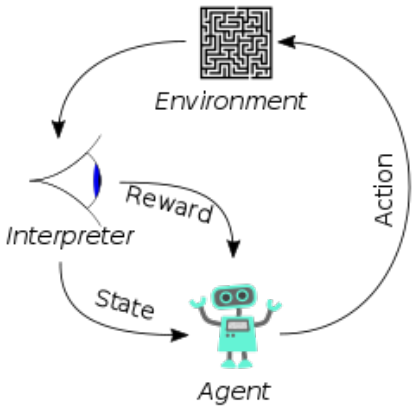

Figure 1: Reinforcement Learning Example

2 Background

Reinforcement learning is a training method based on rewarding desired behaviors and punishing undesired ones. In general, reinforcement learning involves an agent that is able to interpret its environment and interact with it via a set of predefined actions. This ability to award and punish outcomes allows for algorithms to be written that can allow an agent to learn through trial and error. As agents learn they evolve their policy, which is essentially the decision-making process behind their actions, and the goal of training an agent is to have it learn a policy that maximizes the positive rewards it can obtain.

Atari's Joust is a mixed competitive-cooperative where a player(s) must work together to defend themselves against 3 types of increasingly dangerous computer-controlled "Buzzard Riders", where points are awarded for un-seating opponents in a joust. The winner of a joust is the rider whose

Figure 2: Coevolutionary Evolution Strategy (Klijn and Eiben 2021)

1: **procedure** COES(Learning rate α, HoF size k, noise standard deviation σ, initial policy parameters θ_0)
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Sample $\epsilon_1, \ldots, \epsilon_n \sim \mathcal{N}(0, I)$
4: Compute returns $F_i = \frac{1}{k} \sum_{s=0}^{k-1} F(\theta_{t-s}, \theta_t + \sigma\epsilon_i)$
5: **for** $i = 1$ to n **do**
6: Set $\theta_{t+1} \leftarrow \theta_t + \frac{\alpha}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
7: **end for**
8: **end for**
9: **return** θ_t
10: **end procedure**

mount is highest at the moment of contact. If the mounts are of equal height, the joust is a draw. The game ends when one of the players loses all 3 lives and the winner is the player with the higher score.

The presence of a cooperative and competitive environment means that many traditional single-agent reinforcement learning algorithms aren't properly suited for the task as they fail to take into account the rewards that result from playing both cooperatively and/or competitively. However, the coevolutionary evolution strategy from (Klijn and Eiben 2021) consists of an altered reinforcement learning algorithm that learns by playing against older versions of itself and other learning agents concurrently, as shown in Figure 2.

3 Experiments

For our experiments, we begin by using the hyperparameters specified in (Klijn and Eiben 2021); we used a learning rate of 0.1, a HoF size of 1, a noise standard deviation of 0.05, and two neural networks (Model 1 and Model 2). Although the hyperparameters above were used to create policies and mutate them the more general structure of our evolutionary strategy is outlined below. To obtain our results, we ran our algorithm for 3 generations of a population of 3 capped at 10,000 game frames.

Mutations

We have two different mutations happening for each generation, one for the mutation of each individual of the population, and one for the overarching policy for each generation

- **Mutation 1: Population-Wise Mutation** – Each individual policy in the population was a randomly generated Gaussian distribution matrix with a mean of 0 and a standard deviation of 1. Each randomly generated policy was combined with the policy in Model 1 using the noise standard deviation to generate an "explorative" policy based on the policy in Model 1.
- **Mutation 2: Generation-Wise Mutation** – To obtain the new policy to be used by Model 1 in the next generation, we followed line 6 in 3. This was done by normalizing the fitness-weighted sum of the policies in the population using the learning rate, the population size, and the noise standard deviation and combining it with the current policy in Model 1.

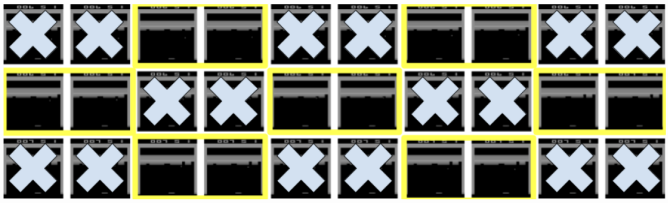

Figure 3: Frame Stacking Visual Demonstration (Seita 2016)

Neural Network Architecture

The architecture of our neural networks was implemented following the specifications outlined in (Klijn and Eiben 2021) and originally from (Mnih et al. 2013). The Neural Networks used stacked frames from the game as inputs and use this to return a move for a specified player.

Each model consists of 3 convolutional layers, and a 512-node densely connected layer followed by a 9-node output layer. The first convolutional layer had 32 channels with a filter size of 8x8 taking strides of 4 pixels. The second convolutional layer had 64 channels with a filter size of 4x4 taking strides of 2 pixels. The third convolutional layer was 64 channels with a filter size of 3x3 taking strides of 1 pixels. Each of these layers used rectifier linear units as their activation function and the final layer was only 9 nodes to represent the 9 unique moves available in the game of joust.

Frame Stacking

While the authors weren't very clear about their implementation of frame stacking in (Klijn and Eiben 2021) we implemented frame stacking based on the article from (Seita 2016) by creating tensors of 4 layers where each layer contains the pixel-wise maximum of every 3rd and 4th frame/image of the current Joust game, shown in Figure 3. Then based on the implementation from (Klijn and Eiben 2021) we added two additional layers to represent which player was being controlled, these functioned as one hot encoding where the player being controlled was a layer of all one's while the other player was a layer of all zero's. After enough frames/images to make a layer have been returned from the game we would pass in a tensor of (86x86x6) for each player and subsequent moves were made based on the output of our models rather than being randomly generated.

Before stacking frames, we preprocessed the images following the methods of (Klijn and Eiben 2021), which are based on the methods from (Mnih et al. 2013). We used grayscaled images from the game and then reshaped the images from $(210, 160, 1)$pixels to $(84, 84, 1)$ pixels and standardized the pixel values of each image. These were done to reduce the noise of the images allowing them to be processed more easily by the models.

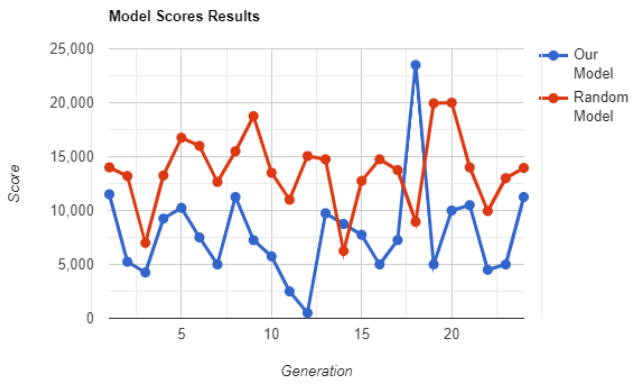

Figure 4: Combined Model Results

Evolution Strategy

To start, our evolutionary strategy starts by generating an initial population of policies to be used in Model 2 and a single randomly generated policy in Model 1. Each individual policy has a mutated policy based on Mutation 1. Since each policy should be able to control the actions of both player 1 and player 2 in the game, the collaborative fitness of the two policies was obtained from playing two games; one where model 1's policy controlled player 1 and model 2's policy controlled player 2 and vice versa. The mean of the scores from both games was used as fitness for the policy in model 2. After this was done for all individuals in the population, the policy in Model 1 was updated using Mutation 2. This process was repeated for the specified number of generations, where for each individual in the population, Model 1 uses the updated policy, and Model 2 uses a combination of Model 1's policy with a randomly generated policy.

4 Results

To further explore the effectiveness of the evolutionary algorithm from (Klijn and Eiben 2021) we ran the algorithm 24 times to evaluate the performance of the final model after training. We also ran the random model 24 times to see how it would perform and found that the average score of the random model was significantly higher than the average score of our model by nearly roughly $6,000$ points. Although the worst score from our model was 500 nearly 10x worse than the random model's 6250, the highest score from our model was 23500, which is 3500 points higher than that from the random model.

Overall, the results suggest that the random model outperformed our model in terms of generating higher results. However, based on the results shown in Figure 4, our model seems to shadow that of the random model, meaning that with increased training of our model, our model can be on par if not better than the random model. With the current results, we can conclude that our model, on average, is able to get a higher score than that of (Klijn and Eiben 2021)'s Co-ES model. 5

	Our Algo	Co-ES (Klijn and Eiben 2021)	Random
Game Frames	10000	$200M$	10000
Test Frames	$2,500$	$50M$	-
Generations	4	500	-
Score	7854.2	5362.2	13695.8

Figure 5: Our results

5 Conclusions

We noticed that our model had a higher average score compared to (Klijn and Eiben 2021)'s Co-ES algorithm despite them doing 250 runs. This was surprising because we did not use a VBN, which ensures that the evolved agents are diverse enough (Klijn and Eiben 2021), which would improve the results. In both cases, the random model did better, so that is a constant result across the two experiments.

One of the biggest obstacles faced in the training process was the singular GPU available for our experiments as well as the limited amount of memory we had access to. The combination of these problems with memory and Singular GPU prevented us from using the same hyperparameters specified in (Klijn and Eiben 2021), which is likely a source of the disparities between our results and theirs. We also did fewer game runs on our model because of the time restraints, as it would have taken too long for us to get the results.

The volume of training done to obtain the final model in (Klijn and Eiben 2021) compared to our model, along with the variance in points our model obtained from playing each game, leads us to believe that with more training our model would have had a higher best score and a higher average score than our current results displayed in Figure 5. The performance of our model could also be tied to the evolutionary strategy used as it produced drastically different results using the same hyperparameters for each run of the algorithm. Overall the co-evolutionary evolution strategy outlined by (Klijn and Eiben 2021) shows promise for being an effective solution for multi-agent reinforcement learning in the Atari game of joust and in the field of multi-agent reinforcement learning in general.

6 Contributions

Bryce and Jason both theorized and wrote code for the algorithm used to train the model that plays Joust. We both did research to better understand and explore different multi-agent algorithms by reading up on research papers and trying to test out different algorithms proposed. We wrote the paper together, revising and proofreading each other's suggestions.

7 Acknowledgements

Thank you to Dr. Raghuram Ramanujann for helping us diagnose our code and giving us insight into certain concepts like frame stacking, using libraries like TensorFlow and numpy, and helping us better understand the coevolutionary evolutionary strategy in (Klijn and Eiben 2021).

References

Klijn, D., and Eiben, A. E. 2021. A coevolutionary approach to deep multi-agent reinforcement learning.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning.

Seita, D. 2016. Frame skipping and pre-processing for deep q-networks on atari 2600 games.