CS230

Deep Kurve!

Taylor Howell
Department of Mechanical Engineering
Stanford, CA 94305
Stanford University
thowell@stanford.edu

Lawrence Peirson
Department of Physics
Stanford University
Stanford, CA 94305
alpv95@stanford.edu

Abstract

In this work we explore reinforcement-learning algorithms for training an agent to play the cult-classic computer game *Achtung Die Kurve!*. In this effort we modify an existing PyGame environment and experiment training an agent using open-source stochastic policy gradient, proximal policy optimization, advantage actor critic, Q-learning, and MuZero implementations. Ultimately, our experiments are unsuccessful in training an agent to have consistent and significantly improvemed performance compared to a random policy. We conclude with discussion and recommendations for achieving improved performance in future work.

CS230: Deep Learning, Fall 2020, Stanford University, CA.

1 Introduction

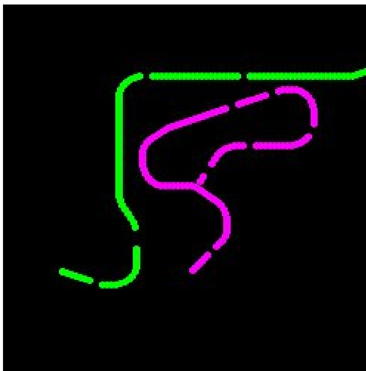

Figure 1: A screen capture from our implementation of *Achtung, Die Kurve!*. Each player controls a different colored agent and the observations is a RGB image with dimensions 320×320.

Achtung, die Kurve!—also known as Curve Fever—is a multi-player computer game released in 1995 [7]. In the game, each player commands a 2D agent which has constant forward velocity by using two keyboard inputs, left and right. Sharp turns are not possible. As the agent moves it leaves a trail, with occasional random gaps, that opponent players must avoid, otherwise they lose. Additionally, players lose if they touch the boundary of the environment. The last remaining player wins.

To date, there are no existing reinforcement learning agents we could find that perform at super-human or human-competitive levels. While this game is popular on the internet—arguably achieving cult status—and attempts have been made to train competitive agents, the game poses subtle difficulties to many classic reinforcement-learning approaches. First, the game is designed to be played human v. human and does not have bots to train against. One approach then is to train an agent using imitation learning from experience generated by humans. However, this technique would require hundreds of thousands of games, each of which can take tens of seconds to play and requires two humans. As a result, this approach is not tractable. Alternatively, self play can be utilized to train an agent's policy by playing against itself. This approach has been successfully applied to a number of games including Backgammon [6] and Go [4]. However, unlike these games, *Achtung, Die Kurve!* does not have a sequential player structure where one player makes a move, then the other player makes their move. As a result, the game's stepping dynamics must be designed such that both player act simultaneous using the same observation in order to leverage existing self-play approaches and tools.

Project. In this project, we experiment with open-source reinforcement-learning algorithms to train an agent to play *Achtung, Die Kurve!*.

First, we modify an existing PyGame implementation to follow the OpenAI Gym [1] standard. Then, we enable a sequential player structure for self play by modifying the stepping structure. Additionally, we implement a number of tweaks to the game environment to create a training environment based on suggests from the literature for agents learning to play Atari games.

Next, we experiment with training an agent on a simple scenario: one player trying to survive as long as possible. This ended up being the primary scenario we focused on in this project. The scenario has simple graphics and a small discrete action space that is similar to the many Atari games that have been successfully mastered by reinforcement-learning agents [2]. We employ open-source implementations of stochastic policy gradient, proximal policy optimization, and Q-learning to train agents for this scenario. The hope is that this trained agent will have learned to avoid walls and self collisions and that this will translate into good performance in the multi-player scenario or be amenable for transfer learning to a self-play algorithm. We verify the performance of agents trained with these approaches using a uniform random policy as a baseline.

Finally, we try an alternative approach that directly leverages self-play by using an open-source implementation of MuZero [3] to train the agent. Here deep learning is utilized to learn a policy, value-

function approximator, and game dynamics, all represented using deep neural networks. Training using this approach requires the modified sequential player stepping structure.

2 Approach

In this section, we first describe the *Achtung, Die Kurve!* game and training environments. Next, we describe the open-source reinforcement-learning tools we employ. The code from our project is available at:

`https://github.com/thowell/achtung`

2.1 Environment

We implement our own version of *Achtung, Die Kurve!* in Python using PyGame. The implementation was initially based on an existing repository

`https://github.com/janowskipio/FarBy`

but has been extensively modified. The modifications are primarily removal of the existing user interface and additions include functionality required by an OpenAI gym environment.

Game environment. The dynamics of the game environment are governed by a `step` function.

$$\texttt{observation}, \texttt{reward}, \texttt{done}, \texttt{info} \leftarrow \texttt{step}(\texttt{action})$$

At each time step this function that takes as input `action` and returns `observation`, `reward`, and `done`. Additionally, `info` can return useful internal information about the environment, but we do not implement or utilize this functionality. The `step` function runs the game dynamics for one player and a given action, then increments the current player. The state of the game is only updated after all players have taken an `action`. Unlike classic board games, in this environment, both players take an action based on the previous state without knowing the other players' current actions.

The action space for this environment is discrete, having three options: do nothing, left, right. The `observation` returned after calling `step` is an 320×320 RGB image of the environment. An example is shown in Fig. 1. Each `reward` takes one of three values depending on if the player is alive and whether the episode is over. We explore various reward schemes in hopes of achieving desired behavior.

During multi-player play, if only one player remains alive the environment resets. For single-player mode, when the player experiences a collision the environment resets. Further, `done` returns true and the episode is complete.

The game environment is contained in the `Achtung` class and can be played by running,

```
$ python achtung.py num_players.
```

where the additional argument, `num_players` $\in [1, 4]$, is used to set the number of players.

Training environment. The literature for training reinforcement-learning agents to play Atari games suggests a number of modifications to game environment in order to learn better policies. First, each RGB image `observation` is downsampled by a factor of 4 and converted to a normalized grayscale image with dimension 80×80. This preprocessing step is visualized in Fig. 2. The benefit of this step is a reduction of the input size which should speed up training.

Next, we implement frame skipping and an observation history. Here, the action selected by the policy is applied to the game environment for 4 steps. The `rewards` are summed over these steps and returned. An element-wise maximum is taken over the last two processed `observations`. The result then replaces the oldest observation from an observation history of length 4. This history is returned and used as the input to the policy.

The idea behind frame skipping is that selecting the optimal action for each frame is unnecessary as the game dynamics are not evolving that quickly. Additionally, this should help the policy develop

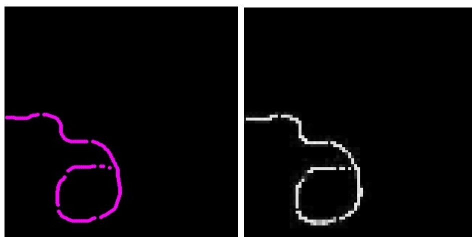

Figure 2: Observation processing. The observation (left) returned from the environment after each step is downsampled by a factor of 4 and converted to a grayscale image (right) before being used as an input to the policy. This episode lasted 137 steps.

better planning as each action has effects farther into the future. By using a history of observations, the policy can reason not just about the current state of the game, but about how it is changes. For example, it should be possible to learn the velocity of the agent. Finally, the maximum over observations frames is recommended to overcome screen flicker. It's unlikely that this is an issue for our environment.

The reward function is: $r = +1$ if player alive after step; $r = -1$ else. This reward structure should encourage the agent to stay alive as long as possible in order to achieve larger rewards and discourage collisions. For updates during training, the rewards are normalized by subtracting the mean and dividing by the standard deviation.

The training environment is implemented as the `AchtungProcess` class and is used for all of the experiments.

2.2 Reinforcement learning

In this work, we leverage existing implementations of reinforcement-learning algorithms.

Stochastic policy gradient. We try a simple approach, stochastic policy gradient [5] to train a single agent to stay alive as long as possible. First, we utilize an implementation,

`http://karpathy.github.io/2016/05/31/rl/`,

that was designed to play the Atari game Pong. Using this implementation, the Pong agent is able to consistently score points after ~ 1 day of training on a laptop computer. The policy is represented as a two-layer fully-connect neural network having a hidden layer of dimension 200 and ending with a sigmoid activation that outputs the probability that the agent should move left. We remove the implementations's preprocessing step and increase the batch size to 100.

Second, in an alternative approach we utilize a PyTorch example,

`https://github.com/pytorch/examples/blob/master/reinforcement_learning/reinforce.py`,

and replace the policy architecture. The example's default architecture is replaced with a simple convolutional neural network used in PyTorch's DQN tutorial. The architecture has three convolution layers each followed by batch normalization before a fully connect layer. The output of the network is a softmax returning the probability of taking one of three actions.

For each approach, the agents are trained overnight for $\sim 200,000$ episodes. The first approach (`fc_train.py`) uses a CPU on a laptop computer and the second approach (`cnn_train.py`) is trained using a single GPU on a workstation.

Stable Baselines3. We next explore implementations of proximal policy optimization, advantage actor critic, and Q-learning from

`https://github.com/DLR-RM/stable-baselines3`

After trying the algorithms' default training hyperparameters, we instead use the ones suggested for Atari games. The primary difference is that we do not simultaneously run multiple environments

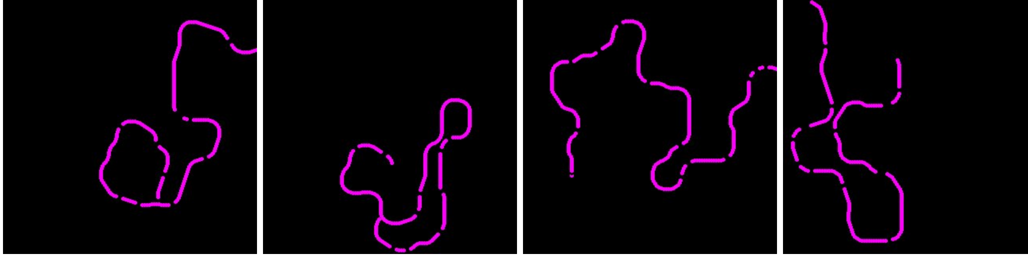

Figure 3: Collection of the best trajectories generated by agent trained with stochastic policy gradient. The accumulated rewards for each trajectory from left from left to right are: 215, 218, 244, 230.

during training. Our environment setup is the same above for stochastic policy gradient. Each of the Stable Baselines3 algorithms utilizes the default PyTorch convolution neural network to represent features used by the policy and/or value function. This network architecture is uses two convolution layers followed by ReLU activations before a fully connected layer.

We train each of these algorithms (`{ppo,a2c,dqn}.ipynb`) overnight on a laptop computer.

MuZero. Lastly, we utilize an open-source implementation of MuZero

`https://github.com/werner-duvaud/muzero-general`

to train a policy in a two-player environment. This approach trains the agent using self play and simultaneously learns a policy, value-function approximator that predicts rewards, and game dynamics. Training is made more efficient by utilizing a replay buffer than enables the current models to learn from previously collected experience. Our interface to MuZero is available in: `muzero_achtung.py`.

We modified our environment's stepping dynamics such that each player selects an action using the current state, but without knowing the other player's new action. This modification enables the environment to be utilized in a self-play manner while maintaining correct game play behavior. (This stepping scheme is analogous to playing tic-tac-toe or chess where each player simultaneously takes their action before knowing the other players').

We successfully stepped through our game with the MuZero test interface and attempted training. However, the training cost blows up and eventually returns `NaNs` so we abandoned this approach.

3 Results and Analysis

In section we present the results from our experiments compared to a random uniform policy baseline (`random_baseline.py`). All policies are evalutated over 100 episodes. The random policy achieves an average reward of ~ 78 with a standard deviation of ~ 44.

Stochastic policy gradient. The fully-connected stochastic policy gradient approach achieves an average reward of ~ 76 with a standard deviation of ~ 45. In comparison, the convolutional neural network stochastic policy gradient approach achieves an average reward of ~ 81 with a standard deviation of ~ 52. These results are collected in Table 1 and we show a collection of the best results generated by the convolutional neural network policy in Fig. 3.

reward	**random**	**FC**	**CNN**
avg.	78	77	81
std.	44	45	52

Table 1: Comparison of rewards between random actions and policy trained with stochastic gradient descent

The variance in the rewards generated by the trained policies is very high. As a result, the average reward generated by the policies varies significantly depending on the random seeds used. *Ultimately, we find that the policies trained with stochastic policy gradient have not improved beyond the random baseline.*

Stable Baselines3. The results from the Stable Baselines3 algorithms are poor compared to the random baseline as well and are summarized in Table 2.

reward	**random**	**PPO**	**A2C**	**DQN**
avg.	78	68	51	37
std.	44	31	16	4

Table 2: Comparison of rewards between random actions and policy trained with Stable Baselines3 algorithms.

Similarly, we find that the policies trained with the Stable Baselines3 algorithms have not improved beyond the random baseline. We do note that it appears that the DQN algorithm has some bug since it always returns the same action both during evaluation before and after training. It's much worse than random because it always goes straight!

Analysis. We found it very discouraging that none of the open-source algorithms produced good policies. However, we learned a significant amount about using deep learning for reinforcement learning, recognize that it is a generally difficult problem, and now suggest a number of improvements to make this project successful in future work.

First, we should have performed extensive hyperparameter tuning. While we used reasonable values, and in the case of the Stable Baselines3 algorithms utilized values that should work for Atari games, based on the recommendations from the community it seems that good performance is highly dependent on tuning for a particular problem. Building a framework for automated testing of many combinations is key and something that we would do in the future.

Second, we originally used the game environment we designed for training. After reading more tutorials online we realized that practical performance on Atari games seems to require the modifications we make for our training environment. This point reinforced the need for building a good dataset for training. We made these improvements at a very late stage and as a result were only able to run each of the algorithms overnight once.

Next, we performed many of the experiments on CPUs. Only in the last week did I install CUDA and then try training GPUs with PyTorch. In the future, I would leverage this powerful library to train policies faster and with more flexibility from the beginning of the project.

Finally, we find that most of the open-source reinforcement-learning agents online for playing Atari are the simpler environments such as Pong or Breakout that are inherently reactionary games. In contrast, *Achtung Die Kurve!* requires both reaction (e.g., don't collide with a wall) and longer-term planning (e.g., don't get trapped within your own circle). To learn this type of planning likely requires a different type of parameterized policy. In future work, we would utilize LSTM networks in order to reason about a planning horizon.

4 Conclusion

In summary, we created a reinforcement-learning environment for the computer game *Achtung Die Kurve!* and explored a number of open-source algorithms for training an agent to play this game. Ultimately, we ran out of time to successfully train policies that can consistently outperform our random action baseline. Finally, we conclude with future improvements for this project.

My ultimate takeaway from this project is that going in I didn't understand how to structure a deep learning project in practice. With this project over–largely without success–I have a much better understanding of what I should focus on (e.g., I spend too much time tweaking the environment and not enought time testing hyperparameters), the infrastructure I should build (e.g., I should have build functions to do more live plotting and automate hyperparameter testing), and how to troubleshoot models (e.g., I would have plotted more outputs and tested algorithms on really simple environments first) in order to be successful in the future.

5 Team and Contributions

For this project, I worked with Lawrence Peirson. He is not taking this course, but is taking Decision Making Under Uncertainty (AA 228) and we are sharing this project between courses.

Essentially, I got the game environment working and then we each used it to perform separate experiments. We each wrote our own reports. Additionally, we discussed various approaches and improvements. Because of COVID and the two of us being in different time zones we didn't collaborate as much as we had initially planned.

T. Howell: wrote this entire report (and: entire proposal, entire milestone); made the video presentation, images, and animations; performed the majority of the game and training environment design (`achtung.py`); did my own experiments with stochastic policy gradient (`fc_train.py`, `cnn_train.py`), advantage actor critic, Q-learning (`{ppo,a2c,dqn}.ipynb`), and MuZero (`muzero_acthung.py`).

L. Peirson: wrote entirely separate reports for AA228; performed a separate stochastic policy gradient experiments with the game instead of training environment (see older commits for `train.py`) using a ResNet and trained using GPU's on Stanford's Sherlock cluster.

6 Appendix

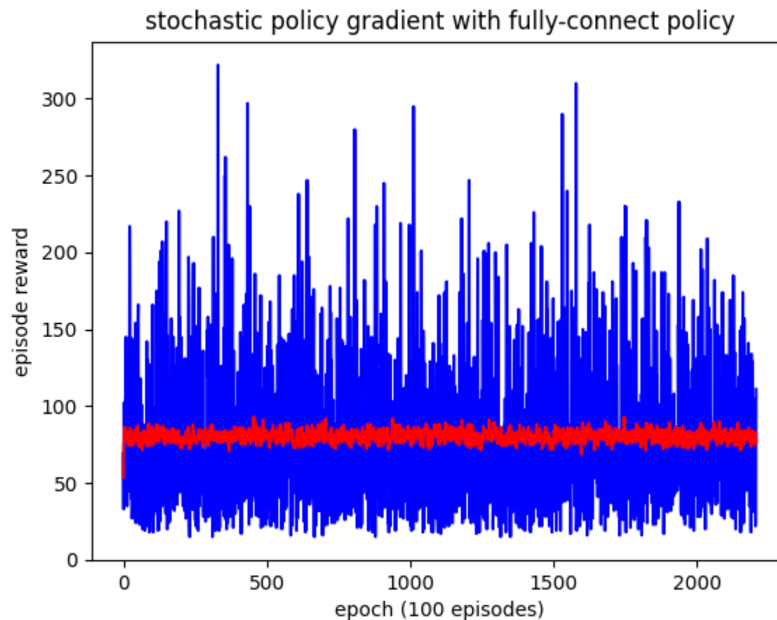

Figure 4: Rewards during training for the episode reward (blue) and running average (red).

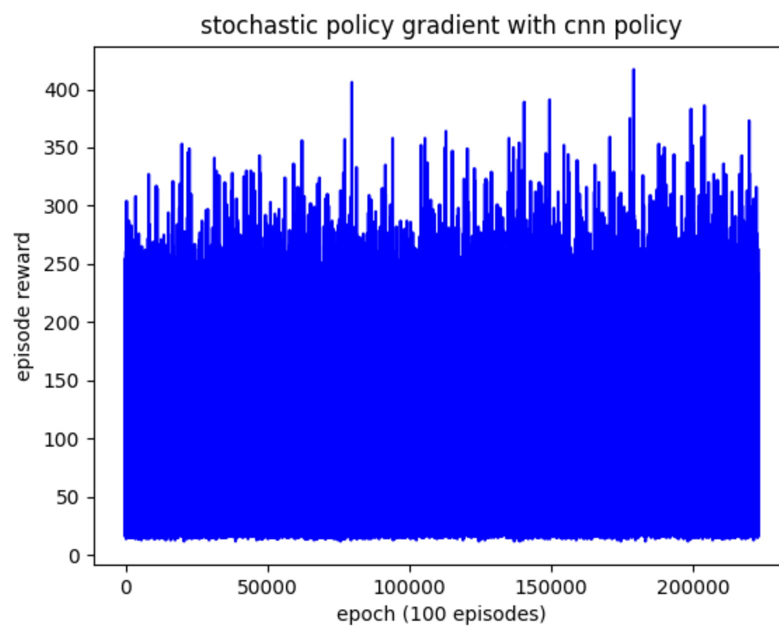

Figure 5: Rewards during training for the episode reward (blue). Note: I forgot to save the running average (red).

References

[1] Greg Brockman et al. “OpenAI Gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[2] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[3] Julian Schrittwieser et al. “Mastering atari, go, chess and shogi by planning with a learned model”. In: *arXiv preprint arXiv:1911.08265* (2019).

[4] David Silver et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[6] Gerald Tesauro. “Temporal difference learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[7] Wikipedia contributors. *Achtung, die Kurve! — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-September-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Achtung,_die_Kurve!&oldid=976491301`.