Pommerman X

Anton Potapchuk, Mikhail Papkov, Novin Shahroudi, Sofiya Demchuk

Institute of Computer Science
University of Tartu

Abstract. In this project we apply different reinforcement learning methods including imitation learning, DQN, MCST, and TPRO to the Pommerman FFA competition challenge. We could successfully perform as good as SimpleAgent which was a baseline heuristic using DQN and an architecture inspired by AlphaGo and Atari papers. Most of our agents emerged with defensive behaviors where we tried to train them further with reward shaping to observe emergence of other behaviors.

1 Introduction

This project conducted as a part of the Introduction to Computation Neuroscience course project. Our focus were on reinforcement learning practices to be applied to the Pommerman competition[1] which is a multi-agent environment for reinforcement learning researchers and practitioners which is also promoted for the NIPS 2018. With a rapid development of tasks with infinitely meaningful change, there has been a huge growth in the use of competitive multi-agent learning. This game is a good framework to test different aspects and features of the artificial intelligence. Our project is dedicated to development of a single Reinforcement Learning agent for the FFA (Free For All) scenario.

The main goal of the project is to build a reinforcement learning agent performing as good as SimpleAgent. Our approach includes two steps. The first step is dedicated to imitation learning and deep learning architectures that prove to be effective for imitation. In the second step we deploy reinforcement learning methods on the architectures from previous step. Our experiment includes evaluation of different architectures mainly inspired by AlphaGo [1], and Atari[2]. Furthermore, we evaluate different combination of architectures, data preprocessing, and reward shaping for the Dueling-DQN, TRPO, and MCTS and demonstrate how we achieved the best results for the RL agent using DQN and best imitation using AlphaGo architecture.

In section 1, we give a brief overview on the literature and comparison of the candidate methods for our experiments. In section 2.3, we explain our imitation learning process, experimentation with DQN and reward shaping, and in section 3.6, we discuss the outcomes. And in the last section discussing the conclusions and direction to explore.

[1] `https://www.pommerman.com/`

2 Background

There has been many attempts and breakthroughs in recent years in the field of reinforcement learning. Many of the ideas and methods are not that new but achieved successful results mainly due to influence of the deep learning. As more attempts in the single agent scenarios become successful such as in Atari games or game of Go, more tendency is made towards new fronts to be explored such as multiagent cases.

Although the applications of reinforcement learning are immense, but so far most of the applications were limited to research and toy problems rather than real ones. But the field has seen a significant activity in real-world applications alongside the research. It seems that there are two main line of work. Those that adopt the achievements from the single agent reinforcement learning to adapt to the circumstances of the multiagent scenario, and those that propose new approaches who are particularly suitable for non-markovian environments.

2.1 Imitation Learning and Data Aggregation

Imitation learning is a supervised learning method. The aim of it is to mimic human (or other already trained expert agent) behaviour by a mapping between observations and actions. This mapping can be called policy as in actual reinforcement learning algorithms. The measure of success of the policy can be measured is the loss terms (e.g. how much is predicted trajectory differs from the expert one). The goal of policy is to minimize the expected loss. Generic imitation learning methods could potentially reduce the problem of teaching a task to that of providing demonstrations, without the need for explicit programming or designing reward functions specific to the task.

The weak point of the supervised learning is an inability to represent all possible events in the training set. If an algorithm gets off the expert path, it can get lost. We might want to train our policy to deal with any possible situation it could encounter, but it is unrealistic. With the Dataset Aggregation algorithm (Dagger) we at first learn the initial policy, then produce a new dataset of observation with this policy and ask our expert to annotate them with his actions. Based on these data we can recover from the errors that were made by the initial policy. We can repeat this algorithm several times and aggregate all generated datasets. Among the policies we can select the last one or the best one. Dagger should be less sensitive to compounding error than supervised imitation learning, precisely because it gets trained on observations that it is likely to see at test time.

2.2 Q-Learning and Deep Q-Learning

There are several big groups of algorithms (approaches) for solving reinforcement learning problem. One of this approaches is the Value function methods. The goal of all reinforcement learning task is to maximize a future reward. The idea of the value function methods is to predict a future reward based on the current state and chosen action. As we know a reward, we can choose the action that corresponds to the maximum future reward.

One of the examples of value function methods is the Q-learning algorithm. In this algorithm, we can use deep neural network as a value function estimator. The pros of the Q-learning algorithm are that it is an offline policy algorithm and it does not have high variance. But the disadvantage is that no convergence guarantees for non-linear function approximation. The problem is that the target value consists of the next step reward and the estimated future reward. This future reward estimated by the same estimator. So, during each gradient descent step, the target function also changes.

The solution is to use the network with frozen parameters for the reward estimation during several gradient descent steps. It helps to stabilize the target value. This is an idea of Deep Q-Network algorithm(DQN).

Lets define a maximum of the reward function.

$$max_a Q_{\theta'}(s, a) = Q_{\theta'}(s, argmax_a Q_{\theta'}(s, a))$$

where a is a current action, s is a current state and Q (s, a) is an estimated future reward. The estimator is not ideal. But we also take an action based on the result of the estimator. Thus we will overestimate the reward function. A solution is to use two separate estimators for the reward function and for the next action. If we will choose an action from different distribution, the future reward and the predicted action will be decorrelated. This is the basic idea of Double Q-learning algorithm. [3]

Another extension that recently outperformed previous DQN was the use of dueling network which evaluates state-value separate than the actions effect on the state which claimed to generalize better. [4]

2.3 Policy Gradient and Actor-Critic

As opposed to the earlier explained value-function methods such as in Q-Learning which tries to approximate the Q-value function to infer policy from the policy gradient methods try to find the optimal policy directly. Policy gradient methods are advantageous for learning stochastic policies, they have better convergence properties, and effective in high-dimensional or continuous action space. However, evaluation of policy is inefficient, prune to high variance, and has tendency to get stuck in local optimal.

There are different variations of policy gradient methods that address most of the shortcomings. The policy function is parameterized using a function approximator such as a neural networks. Parameters that are the weights of the network that can be optimized by finite-differences or stochastic gradient descent. There are also different formulations of the gradient such as score function, softmax policy, and log-likelihood that depending on the problem setting it can be picked accordingly. Monte-carlo is a vanilla policy gradient method that is unbiased, yet with high variance. Intuitively the high variance is originated from the fact that we are calculating the gradient based on the obtained reward at each time step, however this reward may not always be informative, i.e. if we are always receiving a positive reward it does not imply how good the obtained reward actually was. This leads to a new set of methods called the actor-critic,

in which the critic gives directions to actor by updating the action-value function while the actor updates the policy parameters in the suggested direction. This approach solves the so-called policy evaluation problem. The critic also employ monte-carlo and lambda temporal difference as explained earlier. To further reduce the high variance, a baseline is introduced where it can be the value function. The new term that critic needs to evaluate is now called the advantage function.

3 Methods

The Pommerman competition came with a default agent namely the SimpleAgent which uses a heuristic to operate in the environment. At each time steps it crosses out possible actions that are invalid and picks randomly among remained actions. This makes simple agent behavior to some extend unpredictable, and there are situations where no action is left to pick for execution so the simple agent gets stuck. We tried to use this opportunity to imitate the simple agent behavior and develop an RL agent on top of that. To this end, our attempts for developing an agent comprise of two parts. The imitation learning and reinforce learning.

3.1 Imitation Learning

We gathered observations and actions of simple agents within a set number of rollouts. Each rollout is a game or episode where can vary in size or number of steps. The execution of actions and environment update is discrete hence each observation-action pair can be denoted as pair in each time step. We treat the observations as input features and the actions performed by each agent as a label to perform supervised learning. Over the course of the experiment it turned out that some preprocessing are required with some architectures/methods. For an instance, the balance of labels or removing actions taken by SimpleAgent that are not desired due to some implementation flaw.

Since agent can be spawn at each 4 corners of the game board, we measured performance of our agents and the SimpleAgent at each corner. We found that the SimpleAgent heuristic performance varies at each corner of the board. According to Figure it performs best at top-left. For this aim we picked same amount of observations from all corners to compensate for this issue. Similarly, the simple agent produces no action due to its implementation and hence we have samples where the length of the consecutive actions are more than usual. Our initial results with the imitation learning resulted into an agent which produced consecutive actions very often (especially the stop). To this end, we tried to cut the observations with consecutive actions with a threshold.

We considered featurizing the input observation as a map which turned into 18 different binary matrices representing a spatial position of each item on the board. For example we had 18 different channels for our network input respectively representing the agent position, enemies, bombs, etc. In addition to that, we have considered a centric view where all items where transformed

in a relative coordinate to the agent view. Also, we preprocessed a sequence of observations with a moving window of length 4 constructing an input consisting of these 4 observations stacked together as an input for our Atari-inspired DQN architecture.

3.2 Architectures

The imitation learning part also helps to find an appropriate architecture for the task before applying any reinforcement learning method. However, gaining a promising result in the supervised learning phase does not necessarily guarantee achieving a good result in reinforcement learning phase. We tried different architectures to find the best of them which has the capacity for imitation learning as well as the reinforcement learning agent. Here we report three architecture which two of them performed the best for both imitation and the reinforcement learning:

- A simple transformation of inputs to the outputs: this architecture used as a baseline to find out the best achievable performance possible with the simplest architecture so that it can be treated as a baseline for comparison with other architectures.
- Architecture inspired by AlphaGo [1]
- Architecture inspired by Atari [2]

Layer (type)	Output Shape
input_1 (InputLayer)	(None, 11, 11, 18)
flatten_1 (Flatten)	(None, 2178)
dense_1 (Dense)	(None, 6)

Fig. 1: Simple Transformation Architecture

3.3 Deep Q-Network

Deep Q-learning Networks were proved to be successful in playing classic games from Atari 2600 [2], similar approach can be applied to our task. The main difference between Pommerman and Atari is that in Pommerman environment each pixel represents a separate structure. In video games the screen image can be convolved with larger kernels without losing any information. Given that information we modified an Atari playing network as shown on Fig. 2. For the convolutional layers we used the kernel of a size 2 with a stride 1, the number of convolutional filters remained the same. We also used 3 previous observations along with the current observations. They were stacked in the third channel, so the input size was 21x21x72.

It was previously shown that DQN performs best with the imitation pre-training [5], so we pre-trained our network for 10 epochs using the previously

Layer (type)	Output Shape
conv2d_1 (Conv2D)	(None, 20, 20, 32)
activation_1 (Activation)	(None, 20, 20, 32)
conv2d_2 (Conv2D)	(None, 19, 19, 64)
activation_2 (Activation)	(None, 19, 19, 64)
conv2d_3 (Conv2D)	(None, 18, 18, 64)
activation_3 (Activation)	(None, 18, 18, 64)
flatten_1 (Flatten)	(None, 20736)
dense_1 (Dense)	(None, 512)
dense_2 (Dense)	(None, 6)

Fig. 2: Atari Inspired Architecture

collected imitation dataset. For the training we used dueling networks which learned in a competitive way with target model update after 500 steps. For the optimization we used Adam optimizer with learning rate 0.0005. For the action selection we use eps-greedy policy, which means that a random action is selected with probability eps. We anneal eps from 1.0 to 0.1 over the course of 1M steps. This is done so that the agent initially explores the environment (high eps) and then gradually sticks to what it knows (low eps). We also set a dedicated eps value that is used during testing to 0.05 so that the agent still performs some random actions. This ensures that the agent cannot get stuck.

We also tried to run DQN over the pre-trained AlphaGo model with the same hyperparameters. Note that in all of the pre-training cases the final layer activation function should be changed from softmax for the imitation to the linear for the reinforcement.

3.4 Monte Carlo Search Tree

We used AlphaGo inspired architecture with 3 residual blocks. We gathered a dataset using MCTS algorithm with pre-trained model. The number of MCTS iteration was 50. We generated a data from 150 games per each iteration. After gathering data, we trained a model in a supervised learning way. [6]

3.5 Trust Region Policy Optimization

We used AlphaGo inspired architecture with 3 residual blocks. Most of the parameters were set to default. We used a small number of rollouts. It seems increased learning speed but made the learning process more unstable.

3.6 Reward Shaping

Reward shaping can be used to better motivate and facilitate the learning process of an RL agent. In order to guide a learning process of an agent, we used

a reward shaping to compensate for the behaviors that did not emerge or was not desired in the learning process. This method incorporates domain knowledge into reinforcement learning by supplying additional rewards that helps an agent to learn a good policy faster. Based on many experiments we determined a set of rewards that could possibly be motivating for our agents to learn in a better direction. These directions can be categorized into mobility, defensive/offensive behavior and avoiding invalid actions.

Table 1: Results of some recently trained configurations

No	Motivation	Behavior Category	Reward
1	Movement having at least a 1step displacement	Mobility	0.1
2	Prevent execution of consecutive actions for more than 11 steps	Invalid Actions	-0.0001
3	Planting bomb in valuable positions, sum of value based on vicinity of bomb to: wood item: 0.05 enemy: 0.1	Offensive	varying
4	Not planting a bomb	Offensive	-0.5
5	Flame Avoidance: Being on the blast direction gives a negative reward as an inverse of time left before bomb blows	Defensive	Time-variant
6	Avoid staying at the same place	Mobility	0.1
7	Increasing the distance to the bomb	Defensive	0.05
8	Picking the powers	Mobility	0.1
9	Decreasing the distance to the nearest agent	Offensive	0.001

Values for the reward shaping found by experiment. Also some of the values is used as a factor or function of time, and some is used directly. The measures for reward shaping are based on 1 previous observation and hence may not be beneficial for some of the desired behaviors. To avoid reward values blow-up for unseen scenarios the values clipped in range [-0.9, 0.9]

3.7 Experiment Results

Imitation Learning: We collected a dataset from 600 game episodes of approximately 600,000 observations that were annotated with the Simple Agent actions. This dataset was preprocessed according to the requirements of the respective methods. Figures 6, 7, and 8 depicts the average number of steps played per episode, frequency of executed actions, and length of executed actions respectively by the Simple Agent.

DQN (Atari): We trained a DQN network pre-trained with imitation learning for one million steps (35 000 episodes). After this point the agent behaviour became less stable. During the training the agent successfully learned to avoid bombs and to avoid any self-harm, its main goal apparently is to survive as

long as possible, so in most cases it tries to avoid its enemies and runs away from bombs expecting that other agents will accidentally kill each other. It does not try to bomb other agents may be considering it dangerous. In the top-left corner it performs equally well with the Simple Agents, in other corner it loses in most cases. The movement patterns of the trained agent are shown on the Figures - in the Appendix.

DQN (AlphaGo): AlphaGo architecture has shown good results for the imitation learning, so we decided to train a DQN using this weights as a starting point. With the same hyperparameters as Atari network it showed promising results, in a short training time 500 000 steps (20 000 episodes), but this direction requires more thorough exploration. The movement patterns of the trained agent are shown on the Figures - in the Appendix.

DQN with Reward Shaping: By default an agent receives a reward at the end of each episode. -1 for losing the game and +1 for winning the game, otherwise 0. As it is shown in Fig. 3 reward shaping did not help in performance of the agent but made the agent able to stay longer in the game.

3.8 Models Comparison

All of the models were evaluated for 100 episodes in 4 different corners of the board. The average rewards and episode lengths were used for the model comparison. As a benchmark we evaluated the Simple Agent as well. Evaluation results presented on Fig. 3. Most of the models tend to survive as long as possible rather than killing enemies as average episode length shows. In terms of average reward (+1 for win, -1 for lose, 0 for tie) two models performed as good as a Simple Agent: imitation learning with AlphaGo inspired architecture of 3 residual blocks and DQN with Atari inspired architecture after short imitation learning pre-training.

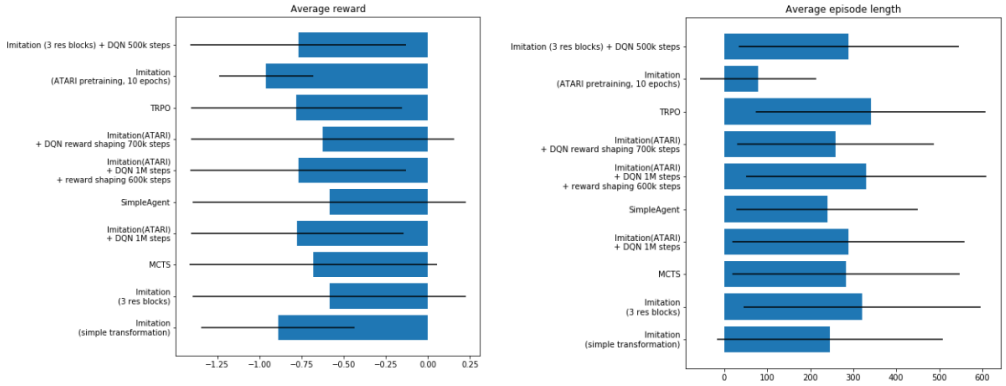

Fig. 3: Average rewards and average episode lengths for all of the evaluated models

The model performance varied in different board corners as shown on a Figure 4. For the imitation learning the data from all initial positions were collected and balanced, however all of the DQN models were trained starting in the top-left corner. It affected the evaluation results: on the example of DQN reinforcement post-training of AlphaGo residual architecture we can see that model performance improved a lot in a top-left corner and at the same time it became worse in every other corner. Similar problem happened with the Atari inspired models as well. Model reinforcement training in different corners could potentially help to overcome such problem.

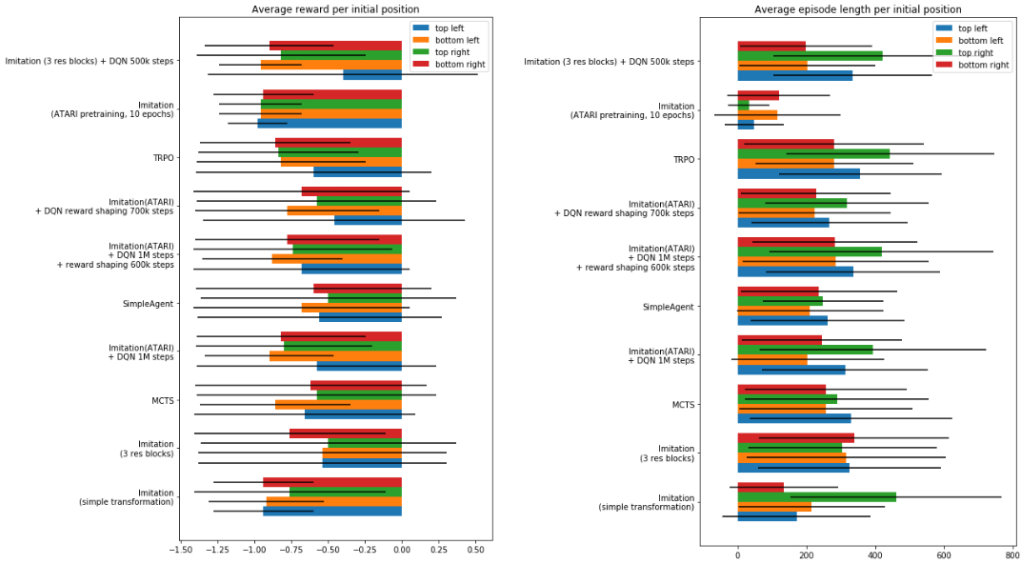

Fig. 4: Average rewards and average episode lengths for different initial positions for all of the evaluated models

Model behavior was described with the action proportion diagrams that can be found in the Appendix A.2. We can see that most of the complicated models learned to move across the board from any position and on average live longer than 100 steps..

4 Conclusion

We evaluated nine methods of agent learning in the Pommerman environment. Three of them used the pure imitation learning strategy, four used DQN pre-trained with imitation learning, one used TRPO, one used MCTS. Two of these methods performs as good as a heuristic Simple Agent algorithm on average and one of them outperformed this algorithm in the top-left corner, where it was trained with reinforcement. Most of our models produced agents with defensive behavior. The reward shaping could help somewhat to change the behavior but not significantly. It is vague how the imitation learning could best be appropriate to be used with the RL phase, however in our experience it seems that imitated models with early stopping is safest choice.

The trained models can be improved with further training in different initial positions with various reward shaping strategies. Their skills can be as well transferred to another rules in the same environment (team match two-by-two with or without communication).

Corresponding code to the reported results and methods here can be found at our code release[7].

5 Acknowledgement

We would like to thank the Daniel Marojal for suggesting the project and initial discussions, and Tambet Matiisen for his guidance and supervision throughout the project from Computation Neuroscience research group at the Institute of Computer Science.

6 Contribution of Authors

All authors contributed to the brainstorming, writing the report and discussing the results. In specific, Mikhail Papkov trained DQN models based on the pre-training and created visualizations, Novin Shahroudi refactored the data preparation and DQN training with simple transformation. Anton Potapchuk trained models in a supervised learning way, made experiments with TRPO and MCTS. Sofiya Demchuk experimented with reward shaping.

References

1. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.
2. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
3. Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.
4. Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.
5. Gabriel Victor de la Cruz, Yunshu Du, and Matthew E. Taylor. Pre-training neural networks with human demonstrations for deep reinforcement learning. *CoRR*, abs/1709.04083, 2017.
6. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
7. Code release for this project. `https://github.com/papkov/pommerman-x`, 2018.

Appendix A

1 Descriptive data statistics

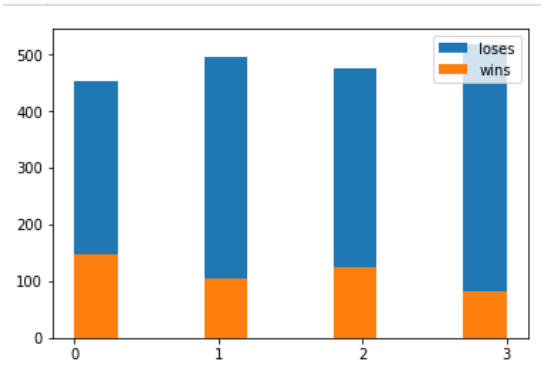

Fig. 5: Proportions of wins and loses in different corners of the board (0 - top-left, 1 - bottom-left, 2 - top-right, 3 - bottom-right)

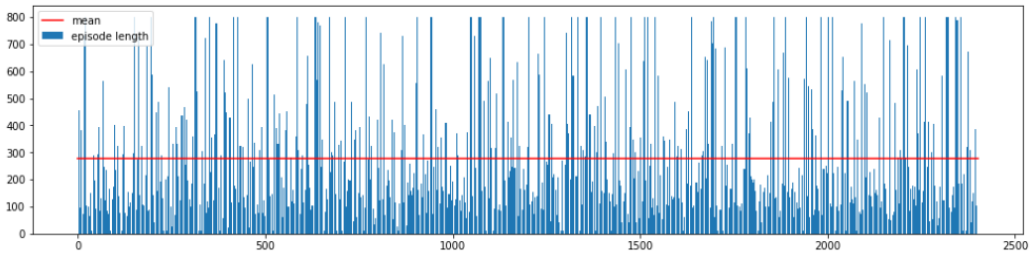

Fig. 6: Lengths of episodes where four Simple Agents played

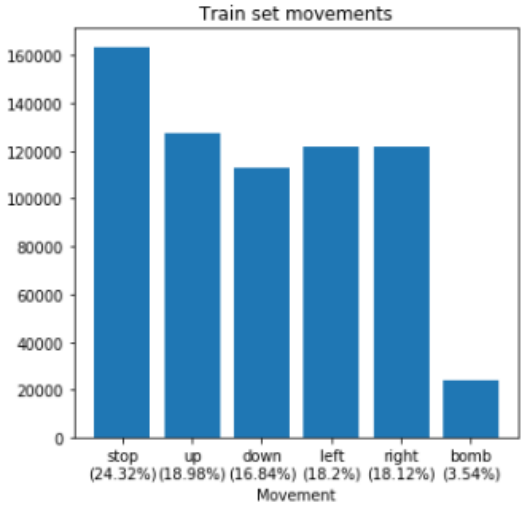

Fig. 7: Proportions of Simple Agent actions in the imitation dataset

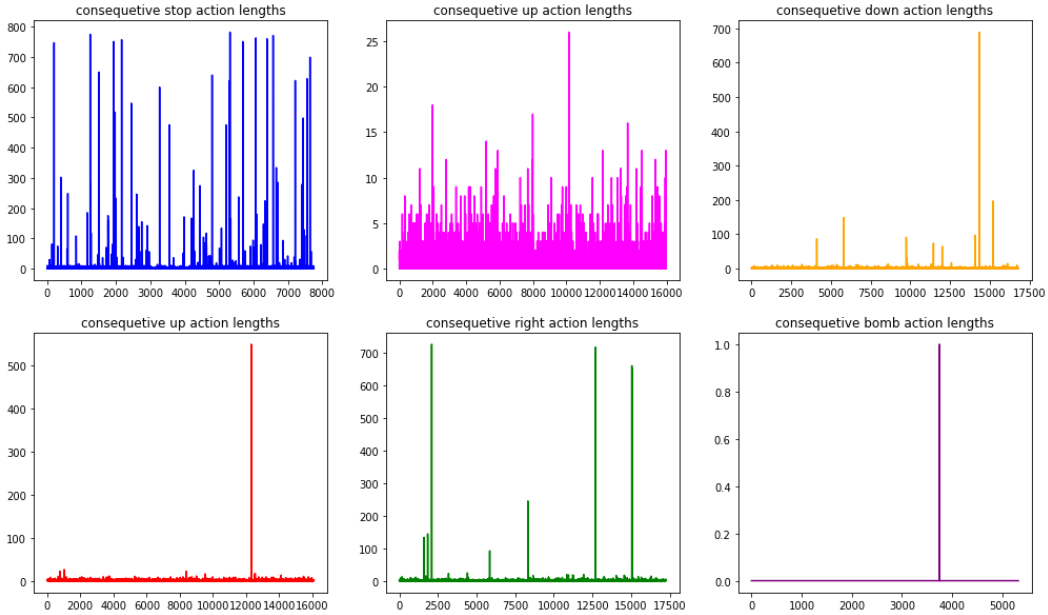

Fig. 8: Simple Agent consecutive actions in the imitation dataset

2 Models comparison

Following figures demonstrate the proportions of agent movements on each step for agents evaluated on the top-left corner.

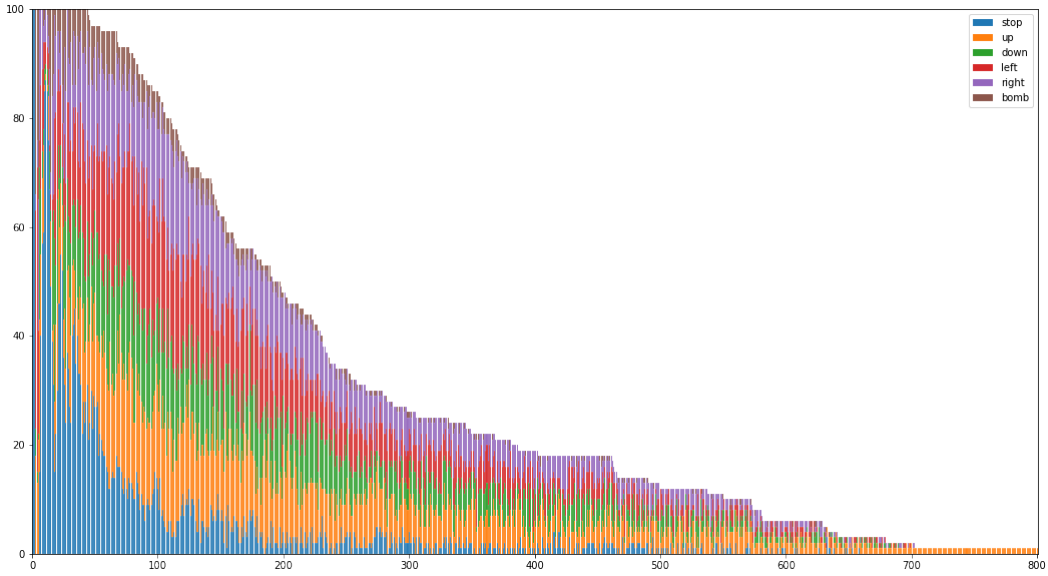

Fig. 9: SimpleAgent

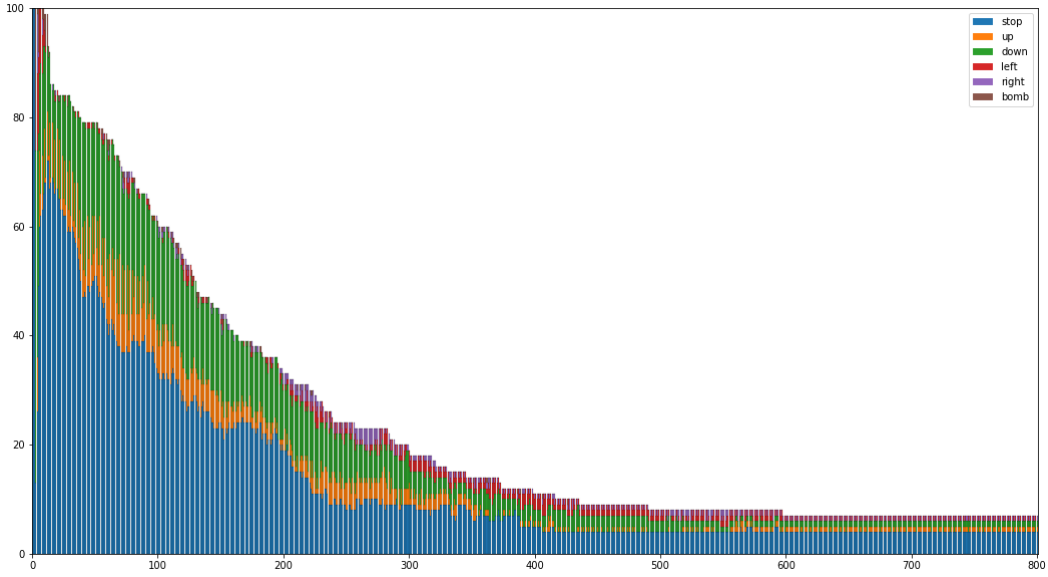

Fig. 10: Imitation learning (simple transformation)

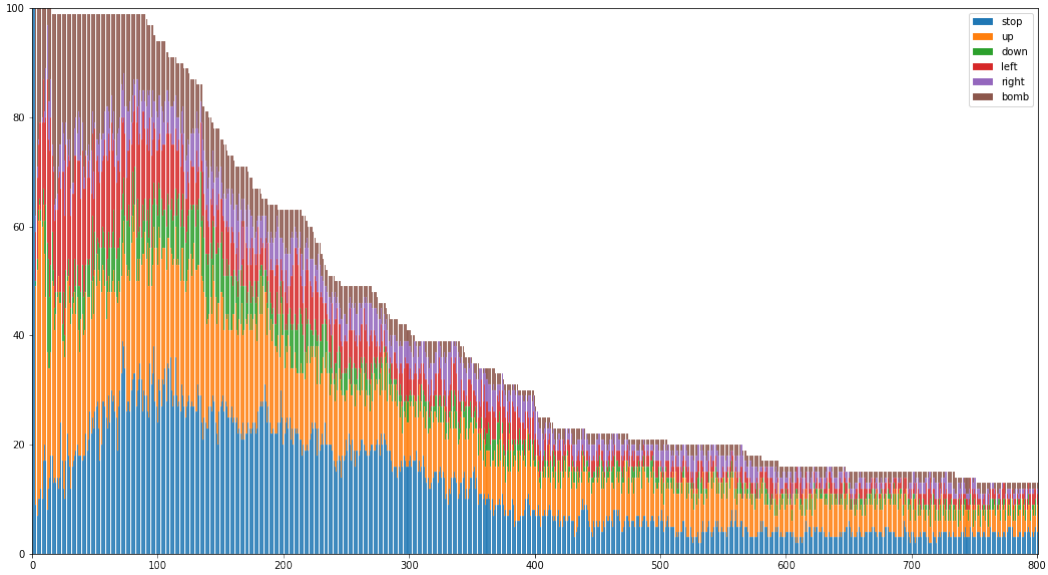

Fig. 11: DQN Atari, short pre-training after imitation learning

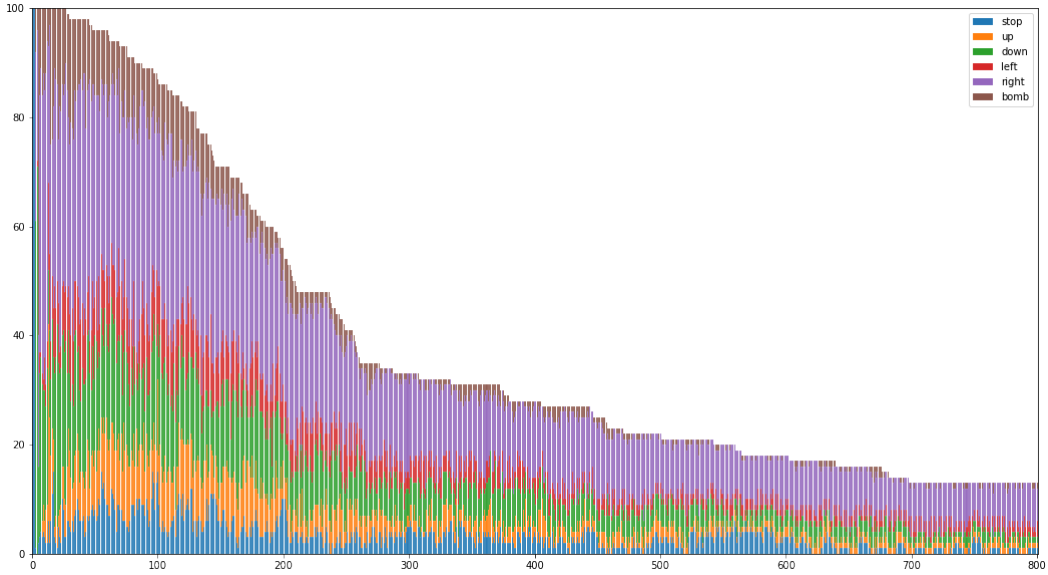

Fig. 12: DQN AlphaGo, long pretraining after imitation learning