Observation Sensitive MCTS for Elevator Transportation

Maximilian Rieger
max.rieger@tum.de

Tim Pfeifle
tim.pfeifle@tum.de

***Abstract*—We present a modification to the Monte-Carlo-Tree-Search (MCTS) approach used in AlphaZero, which incorporates observed rewards in every step. Thereby we extend the applicability of the AlphaZero algorithm to tasks with observed rewards at every step that feed directly into the final reward. We apply this algorithm to one representative of this class, the elevator transportation task, and show that our method is able to train successfully on this task. Our method reaches performance close to the collective-control heuristic.**

I. Introduction and Motivation

Many state of the art Deep Reinforcement Learning methods for robotic tasks use some form of policy gradient, e.g. trust region policy optimization [1] and proximal policy optimization [2]. AlphaZero [3] forms another family of very successful algorithms for board games based on Monte Carlo Tree Search (MCTS). MCTS allows AlphaZero to simulate many actions at every time steps, to find better actions and to estimate a better policy, on which the policy approximator can be trained. However, this method is restricted to 2 player games with rewards only at the end of an episode. MuZero [4] adapts MCTS for a wider problem class including rewards at any time step, but it uses learned approximations for future states instead of full simulations. In this work we take advantage of MCTS on simulated environments for single agent problems with continuous rewards. We adapt the MCTS in AlphaZero to be more sensitive to observed rewards during simulation for faster training, while using full simulations of the environments to gather rewards and future states.

The elevator dispatching problem, i.e. dispatching elevators to minimize the waiting times for passengers, is an example for problems with continuous rewards, as we can model the cumulative waiting time of all passengers at each time step. This problem has a huge search space and only a partially observable stochastic environment. In the past there were several attempts to solve this problem with reinforcement learning [5] [6]. We want to evaluate our method against the standard AlphaZero approach on this problem and compare the results to existing heuristics for elevator dispatching.

II. Method

A. Observation Sensitive MCTS

The core of this work[1] is our modified MCTS, which makes it sensitive to rewards observed during simulation. We achieve this by adjusting the action value approximation of AlphaZero (Eq. 2), which is used to select actions in MCTS (Eq. 1), to incorporate the observed reward $r(\pi_{s,s'})$ during roll-out paths $\pi_{s,s'}$. Observed rewards are normalized by the elapsed time of roll-outs $\delta_{time}(\pi_{s,s'})$ to avoid putting a larger weight on longer paths. We also need to normalize the result to [-1, 1] with f_{norm} to fit to the value network architecture of AlphaZero. The new parameter c_{obs} gives control over the weight of the observed rewards and the value network v.

$$a* = \arg\max_a Q(s,a) + U(s,a) \tag{1}$$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s'} v(s') \tag{2}$$

$$Q_{new}(s,a) = \frac{1}{N(s,a)} \sum_{s'} c_{obs} \cdot f_{norm}\left(\frac{r(\pi_{s,s'})}{\delta_{time}(\pi_{s,s'})}\right) + (1 - c_{obs}) \cdot v(s') \tag{3}$$

Apart from these changes to MCTS our implementation follows the AlphaZero algorithm as depicted in Figure 1.

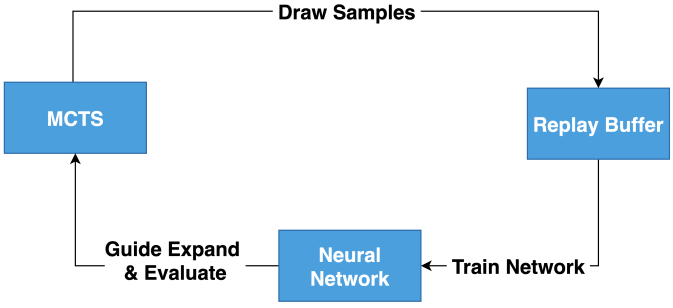

Fig. 1. Algorithm Overview:
1. Episodes are generated using the current policy resulting from the neural network augmented MCTS 2. Generated samples are stored in the replay buffer 3. Samples are randomly drawn from the replay buffer to train the policy and value Networks

B. Elevator Dispatching Environment

We have implemented an elevator environment as visualized in Figure 3, which resembles the real world scenario closely by only observing the passenger requests (up or down at a specific floor), requested floors for elevators and the total number of passengers in an elevator. Unobserved is e.g. how many people are waiting at floors with a passenger request

[1]The source code is available here: https://github.com/MaxRieger96/tum-adlr-ss20-05

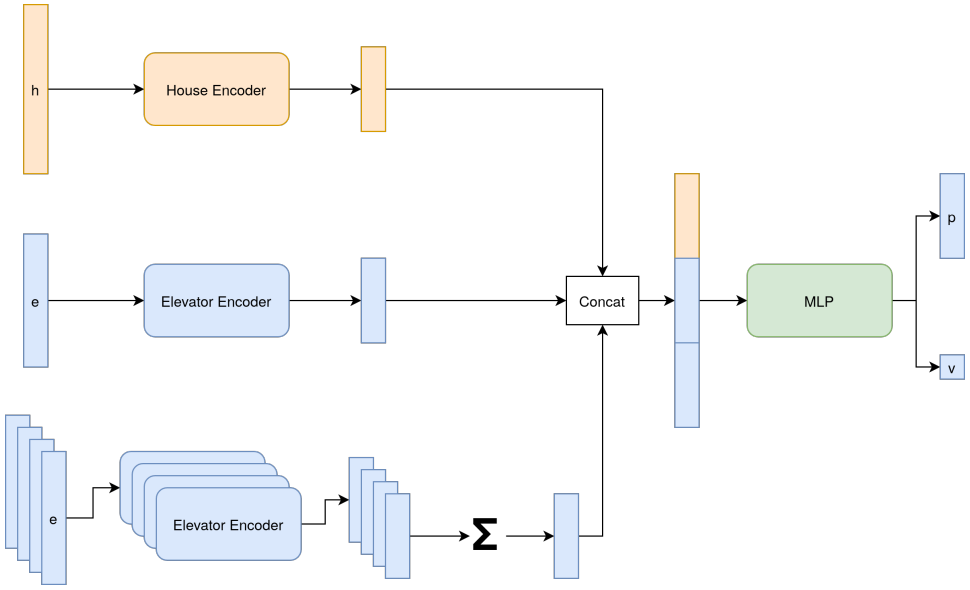

Fig. 2. Model architecture: 1. Encoding of the house and elevators, 2. Concatenation of the inputs, 3.MLP transforming the input, 4.Outputting a policy and value-estimate

and how many passengers want to leave at requested floors. To avoid those hidden states from leaking through the MCTS exploration we represent them stochastically in our state. For our reward function we use the waiting times of all passengers We compute this in every step by sampling waiting times, as passengers waiting at floors are represented stochastically and their count and arrival times are not yet determined.

We observed that the agents performance could be improved by giving an additional penalty for pending passenger requests (Eq. 4).

$$r_t = \sum_{p \in \text{passengers}} p^2_{\text{waiting\_time}} + \sum_{\text{req} \in \text{requests}} (t - t_{\text{req}})^2 \qquad (4)$$

In the observation, we represent each elevator as a vector with values for requested floors, the count of passengers in the elevator, and the current floor of the elevator. We also add a vector representation of the house with values for passenger requests and the time for which they are pending. When the agent makes a decision it controls one specific elevator, which is why we treat this elevator vector differently, from the others. Hence our observation format consists of one house vector, one elevator vector of the elevator which is to be controlled next and an unordered set of other elevators. The house vector is transformed by a MLP which we call *House Encoder*, elevator vectors are transformed by a MLP called *Elevator Encoder*. We combine the resulting set of elevator encodings to a single vector by summing them elementwise. For further processing we concatenate the resulting three vectors to a single longer vector. The concatenated vectors are then processed by an MLP outputting probabilities for possible actions (policy) and a value estimate as seen in Figure 2.

C. Problem Generator

To generate new episodes, we need to generate persons requesting elevators. As we want to avoid leaking hidden states in MCTS during simulation, we again model existing persons stochastically. This means, that when a person arrives at the elevator at a certain point in time, their arrival time is fixed. But all following persons, which want to go in the same direction are not saved explicitly. When an elevator arrives at the floor, where the person and potentially others are waiting, arrival times for all subsequent persons are sampled randomly.

When passengers enter an elevator, the floor requests need to be updated. Again, we do not save a target floor for each passenger, as this could leak. Instead we sample a target floor for each entering passenger to compute the set of newly requested floors for the elevator and the discard the mapping from passengers to target floors. Instead each passengers gets a probability distribution of possible target floors. This distribution is updated during the following time steps, so that a passenger will choose to leave at a certain floor after deciding to not leave at the same floor in a previous time step. This also ensures, that all passengers will leave at some floor eventually.

The arrival of new passengers at different floors is modelled by a Poisson distribution and can be specified to resemble the peak load at the lobby floor.

III. Results and Discussion

We compare our performance against Random Policy, AlphaZero, as well as one of the most common heuristics for elevator dispatching *Collective Control* [7], where elevators stop at the nearest request (passenger requests or floor-requests) in their running direction and switch direction if the requests in the current direction are exhausted.

As seen in the training curves in Figure 4 the unmodified AlphaZero algorithm does not significantly improve at all over

Method	Avg. Waiting Time	Avg. Percent Transported	Avg. Quadr. Waiting Time
1Elev-3Floor			
Random Policy	45.8 ±0.9	90.5% ±5.0%	125,634.6 ±61,885.9
Pure MCTS	33.0 ±5.8	93.7% ±4.2%	72,391.6 ±30,712.9
AlphaZero	45.8 ±15.9	91.6% ±3.7%	137,572.5 ±113,445.6
Observation sensitive MCTS (ours)	12.5 ±0.4	98.3% ±1.6%	7,348.2 ±817.9
Collective Control	**10.5 ±0.3**	**98.7% ±1.5%**	**4,707.3 ±971.4**
2Elev-3Floor			
Random Policy	33.6 ±3.4	94.2% ±3.7%	70,085.7 ±20,346.4
Pure MCTS	27.5 ±3.9	95.4% ±5.2%	44,757.2 ±18,650.1
AlphaZero	30.5 ±2.8	95.3% ±2.7%	56,528.8 ±16,056.5
Observation sensitive MCTS (ours)	11.8 ±0.6	97.7% ±2.2%	6,898.5 ±1,122.0
Collective Control	**8.9 ±0.3**	**99.0% ±2.0%**	**3,254.8 ±779.5**

TABLE I
METHOD COMPARISON

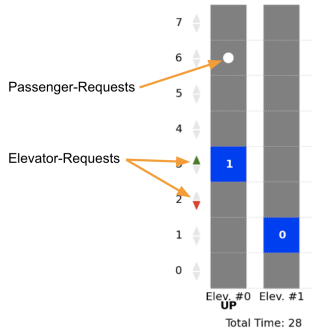

Fig. 3. Visualization of the implemented, not fully-observable elevator simulation environment

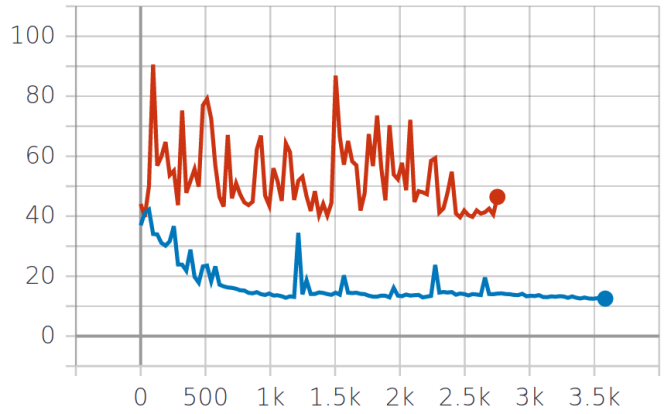

Fig. 4. Average Waiting Time per Person over Training: AlphaZero (Red), Observation Sensitive MCTS (Blue)

time. In contrast to this, our method improves over time by guiding the learning through the early observation rewards. This indicates that our modification of AlphaZero makes it suitable to continuous reward tasks. In Table I we see that pure MCTS, without the guidance of our neural network, yields a worse policy for this task, even though it outperforms Random Policy.

But why does AlphaZero, while being a very powerful algorithm for Go and other problems, not work for our problem? Our intuition for this problem is, that AlphaZero's use of a value network for evaluating environment states is less suitable for this problem. While the probable winner of a game of go can be predicted by looking at the board, this does not work very well for a house with elevators. Looking at a single state within an episode does not allow conclusions of how long passengers have waited before that time step. Also future waiting times cannot be predicted well, as in contrast to board games, the current state might only affect a limited number of future states. In contrast to board games, where the one task of an episode is to defeat the opponent, an episode of elevator dispatching might be considered as a series of mini-tasks, i.e. transporting a single person to their goal. While AlphaZero would try to evaluate the whole episode of transportation, our observation sensitive MCTS is able to attend to the smaller time-frames with the observations and rewards collected during MCTS. By learning to perform well in this short term, our agent learns to perform well for whole episodes, while AlphaZero is trapped by trying to figure out how good a whole episode might turn out.

Our RL approach still comes only close to the (in our example cases) near optimal collective control heuristic (see Table I). For multi-elevator examples it is likely that the RL outperforms this heuristic, as the heuristic does not take the state and action of the other elevators into account, resulting in a phenomenon called *bunching*, where elevators perform the same action instead of working together.

As in the case of AlphaZero the majority of the computation time is spent on simulating the environment. AlphaZero is also known to require a lot of computation, the resources used in [3] exceed our available hardware by a factor of about 1000. We therefore limited our comparison to simple examples (3 floors) of the environment, to be able to perform the hyper-parameter search. To improve the training performance we drew samples using multiple processes in parallel. Interesting future work would improve the sampling performance to enable the comparison with e.g. collective control on more complex examples.

With an increasing number of floors, new problems arise:

- Bias towards the lobby floor, especially during peak-times

- Shifting passenger requests over the span of the day

A RL algorithm is potentially better equipped to handle those, as its policy is not based on general rules, but instead learned from the flow of passengers.

REFERENCES

[1] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[4] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *arXiv preprint arXiv:1911.08265* (2019).

[5] R. H. Crites and A. G. Barto. "Improving elevator performance using reinforcement learning". In: *Advances in neural information processing systems*. 1996, pp. 1017–1023.

[6] X. Yuan, L. Buşoniu, and R. Babuška. "Reinforcement learning for elevator control". In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 2212–2217.

[7] M.-L. Siikonen. "Elevator traffic simulation". In: *Simulation* 61.4 (1993), pp. 257–267.