# A0C: Alpha Zero in Continuous Action Space

**Thomas M. Moerland**[*†], **Joost Broekens**[*], **Aske Plaat**[†] **and Catholijn M. Jonker**[*†]
[*]*Dep. of Computer Science, Delft University of Technology, The Netherlands*
[†]*Dep. of Computer Science, Leiden University, The Netherlands*

## Abstract

A core novelty of Alpha Zero is the interleaving of tree search and deep learning, which has proven very successful in board games like Chess, Shogi and Go. These games have a discrete action space. However, many real-world reinforcement learning domains have continuous action spaces, for example in robotic control, navigation and self-driving cars. This paper presents the necessary theoretical extensions of Alpha Zero to deal with continuous action space. We also provide a preliminary experiment on the Pendulum swing-up task, empirically verifying the feasibility of our approach. Thereby, this work provides a first step towards the application of iterated search and learning in domains with a continuous action space.

**Keywords:** Reinforcement Learning, Planning, Monte Carlo Tree Search, Continuous Action Space.

## 1. Introduction

Alpha Zero has achieved state-of-the-art, super-human performance in Chess, Shogi (Silver et al., 2017a) and the game of Go (Silver et al., 2016, 2017b). The key innovation of Alpha Zero compared to traditional reinforcement learning approaches is the use of a small, nested tree search as a policy evaluation.[1] Whereas traditional reinforcement learning treats each environment step or trace as an individual training target, Alpha Zero aggregates the information of multiple traces in a tree, and eventually aggregates these tree statistics into targets to train a neural network. The neural network is then used as a prior to improve new tree searches. This closes the loop between search and function approximation (Figure 1).

While Alpha Zero has been very successful in two-player games with discrete action spaces, it is not yet applicable in continuous action space (nor has Alpha Zero been tested in single-player environments). Many real-world problems, such as robotics control, navigation and self-driving cars, have a continuous action space. Compared to the Alpha Zero paradigm for discrete action spaces, we require:

1. A Monte Carlo Tree Search (MCTS) method that works in continuous action space. We built here on earlier results on *progressive widening*. Since this is known from previous work, and due to space restrictions, we detail this in Appendix A.1.

2. Incorporation of a continuous prior to steer a new MCTS iteration. While Alpha Zero uses the discrete density as a prior in a (P)UCT formula (Rosin, 2011; Kocsis and Szepesvári, 2006), we need to leverage a continuous density (which is unbounded) to direct the next MCTS iteration (Section 3)

[1] Additionally, the tree search provides an efficient exploration method, which is a key challenge in reinforcement learning (Moerland et al., 2017).

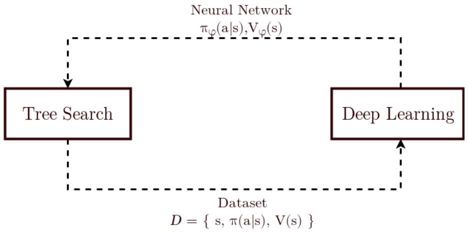

Figure 1: Iterated tree search and function approximation.

3. A training method. Alpha Zero transforms the MCTS visitation counts to a discrete probability distribution. We need to estimate a continuous density from a set of support points, and specify an appropriate training loss in continuous policy space (Section 4).

Items 2 and 3 are the core contributions of this paper, closing the loop in Figure 1 for the continuous action space setting.

The remainder of this paper is organized as follows. Section 2 presents essential preliminaries on reinforcement learning and MCTS. Section 3 discusses the required MCTS modifications for a continuous action space with a continuous prior (Fig. 1, upper part of the loop). In Section 4 we cover the generation of training targets from the tree search and specify an appropriate neural network loss (Fig. 1, lower part of the loop). Sections 5 and 7 present experiments and conclusions.

## 2. Preliminaries

**Markov Decision Process** We adopt a finite-horizon Markov Decision Process (MDP) (Sutton and Barto, 2018) given by the tuple $\{\mathcal{S}, \mathcal{A}, f, \mathcal{R}, \gamma, T\}$, where $\mathcal{S} \subseteq \mathbb{R}^{n_s}$ is a state set, $\mathcal{A} =\subseteq \mathbb{R}^{n_a}$ continuous action set, $f : \mathcal{S} \times \mathcal{A} \rightarrow P(\mathcal{S})$ denotes a transition function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ a (bounded) reward function, $\gamma \in (0, 1]$ a discount parameter and $T$ the time horizon. At every time-step $t$ we observe a state $\boldsymbol{s}^t \in \mathcal{S}$ and pick an action $\boldsymbol{a}^t \in \mathcal{A}$, after which the environment returns a reward $r^t = \mathcal{R}(\boldsymbol{s}^t, \boldsymbol{a}^t)$ and next state $\boldsymbol{s}^{t+1} = f(\boldsymbol{s}^t, \boldsymbol{a}^t)$. We act in the MDP according to a stochastic policy $\pi : \mathcal{S} \rightarrow P(\mathcal{A})$. Define the (policy-dependent) state value $V^\pi(\boldsymbol{s}^t) = \mathrm{E}_\pi[\sum_{k=0}^{T}(\gamma)^k \cdot r^{t+k}]$ and state-action value $Q^\pi(\boldsymbol{s}^t, \boldsymbol{a}^t) = \mathrm{E}_\pi[\sum_{k=0}^{T}(\gamma)^k \cdot r^{t+k}]$, respectively. Our goal is to find a policy $\pi$ that maximizes this cumulative, discounted sum of rewards.

**Monte Carlo Tree Search** We assume the reader is familiar with Monte Carlo Tree Search. We use a particular MCTS variant konwn as the PUCT algorithm (Rosin, 2011), as also used in Alpha Zero (Silver et al., 2017a,b). Due to space restrictions, we present an detailed introduction of the algorithm in Appendix A, including a section on progressive widening (A.1).

**Neural Networks** We introduce two neural networks - similar to Alpha Zero - to estimate a parametrized policy $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ and the state value $V_\phi(\boldsymbol{s})$. Both networks share the initial layers. The joint set of parameters of both networks is denoted by $\phi$. The neural networks are trained on targets generated by the MCTS procedure. These training targets, extracted from the tree search, are denoted by $\hat{\pi}(\boldsymbol{a}|\boldsymbol{s})$ and $\hat{V}(\boldsymbol{s})$.

## 3. MCTS with Continuous Prior

We first detail a way to include a continuous policy network into the MCTS search. For now assume we manage to train a policy network $\pi_\phi(\boldsymbol{s})$ from the results of the MCTS procedure. Alpha Zero can enumerate the probability for all available discrete actions, and uses this probability as a prior scaling on the upper confidence bound term in the UCT formula (Eq. 7). For the continuous policy space, we could use a similar equation, where we use $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ of the considered $\boldsymbol{a}$ as predicted by the network. However, the continuous $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ is unbounded.[2] This gives us the risk of rescaling/stretching the confidence intervals too much. Another option - which we consider in this work - is to use the policy network to sample new child actions in the tree search (when adding a new action based on progressive widening). Thereby, the policy net steers the actions that we will consider in the tree search. This has a similar effect as Eq. 7 for AlphaGo Zero does, as it effectively prunes away child actions in subtrees of which we already know that they perform poorly.

## 4. Neural network training in continuous action space

We next want to use the MCTS output to improve our neural networks. Compared to Alpha Zero, the continuous action space forces us to come up with a different policy network specification, policy target calculation and training loss. These aspects are covered in Section 4.1. Afterwards, we briefly detail the value network training procedure, including a slight variant of the value target estimation (Section 4.2).

### 4.1 Policy Network

**Policy Network Distribution** We require a neural network that outputs a continuous density. However, continuous action spaces usually have some input bounds. For example, when we learn the torques or voltages on a robot manipulator, then a too extreme torque/voltage may break the motor altogether. Therefore, continuous actions spaces are generally symmetrically bounded to some $[-c_b, c_b]$ interval, for scalar $c_b \in \mathbb{R}^+$. To ensure that our density predicts in this range, we use a transformation of a factorized Beta distribution $\pi_\phi(\boldsymbol{a}|\boldsymbol{s}) = g(\boldsymbol{u})$, with elements $u_i \sim \text{Beta}(\alpha_i(\phi), \beta_i(\phi))$ and deterministic transformation $g(\cdot)$. Details are provided in Appendix B. Note that the remainder of this section holds for any $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ network output distribution from which we know how to sample and evaluate (log) densities.

**Training Target** We want to transform the result of the MCTS with progressive widening to a continuous target density $\hat{\pi}$ (to training our neural network with). Recall that MCTS returns the sets $A_0$ and $N_0$ of root actions and root counts, respectively. We can not normalize these counts like Alpha Zero does (Eq. 9) for the discrete case. The only assumption, similar

[2] For a discrete probability distribution, $\pi(\boldsymbol{a}) \leq 1 \; \forall \boldsymbol{a}$. However, although the probability density function (pdf) of continuous random variables integrates to 1, i.e. $\int \pi(\boldsymbol{a}|\boldsymbol{s}) \mathrm{d}\boldsymbol{a} = 1$, this does not bound the value of the pdf $\pi(\boldsymbol{a})$ at a particular point $\boldsymbol{a}$, i.e. $\pi(\boldsymbol{a}) \in [0, \infty)$.

to Alpha Zero, that we make here is that the density at a root action $\boldsymbol{a}_{0,i}$ is proportional to the visitation counts, i.e.[3]

$$\hat{\pi}(\boldsymbol{a}_i|\boldsymbol{s}) = \frac{n(\boldsymbol{s},\boldsymbol{a}_i)^\tau}{Z(\boldsymbol{s},\tau)} \tag{1}$$

where $\tau \in \mathbb{R}^+$ specifies some temperature parameter, and $Z(\boldsymbol{s},\tau)$ is a normalization term (that is assumed to not depend on $\boldsymbol{a}_i$, as the density at the support points is only proportional to the counts). Note that this does not define a proper density, as we never specified a density in between the support points. However, we can ignore this issue, as we will only consider the loss at the support points.

**Loss** In short, our main idea is to leave the normalization and generalization of the policy over the action space to the network loss. If we specify a network output distribution that enforces $\int_a \pi_\phi(\boldsymbol{a}|\boldsymbol{s}) = 1$, i.e., making it a proper continuous density, then we may specify a loss with respect to a target density $\hat{\pi}(\boldsymbol{a}|\boldsymbol{s})$, *even when the target density is only known on a relative scale.* More extreme counts (relative densities) will produce stronger gradients, and the restrictions of the network output density will ensure that we can not pull the density up or down over the entire support (as it needs to integrate to 1). This way, we make our network output density mimic the counts on a relative scale.

We will first give a general derivation, acting as if $\hat{\pi}(\boldsymbol{a}|\boldsymbol{s})$ is a proper density, and swap in the empirical density at the end. We minimize a policy loss $\mathcal{L}^{\text{policy}}(\phi)$ based on the Kullback-Leibler divergence between the network output $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ and the empirical density $\hat{\pi}(\boldsymbol{a}|\boldsymbol{s})$ (Eq. 1):

$$\mathcal{L}^{\text{policy}}(\phi) = \mathrm{D}_{\mathrm{KL}}\Big(\pi_\phi(\boldsymbol{a}|\boldsymbol{s})\|\hat{\pi}(\boldsymbol{a}|\boldsymbol{s})\Big) = \mathrm{E}_{\boldsymbol{a}\sim\pi_\phi(\boldsymbol{a}|\boldsymbol{s})}\Big[\log\pi_\phi(\boldsymbol{a}|\boldsymbol{s}) - \log\hat{\pi}(\boldsymbol{a}|\boldsymbol{s})\Big] \tag{2}$$

We use the REINFORCE[4] trick to get an unbiased gradient estimate of the above loss:

$$\begin{aligned}\nabla_\phi\mathcal{L}^{\text{policy}}(\phi) &= \nabla_\phi\mathrm{E}_{\boldsymbol{a}\sim\pi_\phi(\boldsymbol{a}|\boldsymbol{s})}\Big[\log\pi_\phi(\boldsymbol{a}|\boldsymbol{s}) - \tau\log n(\boldsymbol{a},\boldsymbol{s}) + \log Z(\boldsymbol{s},\tau)\Big]\\ &= \mathrm{E}_{\boldsymbol{a}\sim\pi_\phi(\boldsymbol{a}|\boldsymbol{s})}\Big[\Big(\log\pi_\phi(\boldsymbol{a}|\boldsymbol{s}) - \tau\log n(\boldsymbol{a},\boldsymbol{s}) + \log Z(\boldsymbol{s},\tau)\Big)\nabla_\phi\log\pi_\phi(\boldsymbol{a}|\boldsymbol{s})\Big]\end{aligned}$$

We now drop $Z(\boldsymbol{s},\tau)$ since it does not depend on $\phi$ (or chose an appropriate state-dependent baseline, as is common with REINFORCE estimators). Moreover, we replace the expectation over $\boldsymbol{a} \sim \pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ with the empirical support points $\boldsymbol{a}_i|\boldsymbol{s}$ in our dataset $\mathcal{D}$. Our final gradient estimator becomes

$$\nabla_\phi\mathcal{L}^{\text{policy}}(\phi) = \mathrm{E}_{\boldsymbol{a}_i|\boldsymbol{s}\sim\mathcal{D}}\Big[\Big(\log\pi_\phi(\boldsymbol{a}_i|\boldsymbol{s}) - \tau\log n(\boldsymbol{s},\boldsymbol{a}_i)\Big)\nabla_\phi\log\pi_\phi(\boldsymbol{a}_i|\boldsymbol{s})\Big] \tag{3}$$

To stabilize policy learning we additionally introduce an entropy regularization term. See Appendix C for details.

[3] The remainder of this section always concerns the root state $\boldsymbol{s}_0$ and root actions $\boldsymbol{a}_{0,i}$. Therefore, we omit the depth subscript (of 0) for readability.

[4] The REINFORCE trick (Williams, 1992), also known as the likelihood ratio estimator, is an identity regarding the derivative of an expectation, when the expectation depends on the parameter towards which we differentiate: $\nabla_\phi\mathrm{E}_{\boldsymbol{a}\sim p_\phi(\boldsymbol{a})}[f(\boldsymbol{a})] = \mathrm{E}_{\boldsymbol{a}\sim p_\phi(\boldsymbol{a})}[f(\boldsymbol{a})\nabla_\phi\log p_\phi(\boldsymbol{a})]$, for some function $f(\cdot)$ of $\boldsymbol{a}$.

### 4.2 Value Network

Value network training is almost identical to the Alpha Zero specification. The only thing we modify is the estimation of $\hat{V}(\boldsymbol{s})$, the training target for the value. Alpha Zero uses the eventual return of the full episode as the training target for every state in the trace. This is an unbiased, but high-variance signal (in reinforcement learning terminology (Sutton and Barto, 2018), it uses a full Monte Carlo target). Instead, we use the MCTS procedure as a value estimator, leveraging the action value estimates $Q(s_0, a)$ at the root $s_0$. We could weigh these according to the visitation counts at the root. However, we usually built relatively small trees,[5] for which a non-negligible fraction of the traces are exploratory. Therefore, we propose an *off-policy* estimate of the value at the root:

$$\hat{V}(\boldsymbol{s}_0) = \max_{\boldsymbol{a}} Q(\boldsymbol{s}_0, \boldsymbol{a}) \tag{4}$$

The value loss $\mathcal{L}^V(\phi)$ is a standard mean-squared error loss:

$$\mathcal{L}^V(\phi) = \Big(V_\phi(\boldsymbol{s}) - \hat{V}(\boldsymbol{s})\Big)^2. \tag{5}$$

## 5. Experiments

Figure 2 shows the results of our algorithm on the Pendulum-v0 task from the OpenAI Gym (Brockman et al., 2016). The curves show learning performance for different computational budgets per MCTS at each timestep. Note that the x-axis displays true environment steps, which includes the MCTS simulations. For example, if we use 10 traces per MCTS, then every real environment step counts as 10 on this scale.

First, we observe that our continuous Alpha Zero version does indeed learn on the Pendulum task. Interestingly, we observe different learning performance for different tree sizes, where the 'sweet spot' appears to be at an intermediate tree size (of 10). For larger trees, we complete less episodes (a single episode takes longer) and therefore train our neural network less frequently. Therefore, although each individual trace gets more budget, it takes longer before the tree search starts to profit from improved network estimates (generalization).

We train our neural network after every completed episode. However, the runs with smaller tree sizes complete much more episodes compared to the runs with a larger tree size. Moreover, the data generated from larger tree searches could be deemed 'more trustworthy', as we spend more computational effort in generating them. We try to compensate for this effect by making the number of training epochs over the database after each episode proportional to the size of the nested tree search. Specifically, after each episode we train for

$$n_{\text{epochs}} = \left\lceil \frac{N_{\text{traces}}}{c_e} \right\rceil \tag{6}$$

for constant $c_e \in \mathbb{R}^+$ and $\lceil \cdot \rceil$ denoting the ceiling function. In our experiments we set $c_e = 20$. This may explain why the run with $N_{\text{traces}} = 25$ performs suboptimal compared

[5] AlphaGo Zero uses 1600 traces per timestep. We evaluate on smaller domains, and have less computational resources.

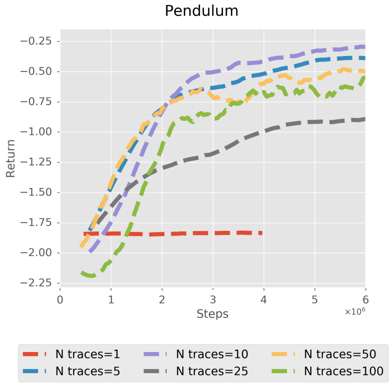

Figure 2: Learning curves for Pendulum domain. Compared to the OpenAI Gym implementation we rescale every reward by a factor 1/1000 (which leaves the task and optimal solution unchanged). Results averaged over 10 repetitions. Implementation details can be found in Appendix D.

to the others, as the non-linearity in Eq. 6 (due to the ceiling function) may accidentally turn out bad for this number of tree traces. Moreover, note that the learning curve of training with a tree size of 1 is shorter than the other curves. This happens because we gave each run an equal amount of wall-clock time. The run with tree size 1 finishes much more episodes, and because $c_e > 1$ it still trains more frequently than the other runs, which makes it eventually perform less total steps in the domain.

## 6. Discussion

The results in Fig. 2 reveal an interesting trade-off in the iterated tree search and function approximation paradigm. We hypothesize that the strength of tree search is the in the locality of information. Each edge stores its own statistics, and this makes it easy to locally separate the effect of actions. Moreover, the forward search gives a more stable value estimate, smoothing out local errors in the value network. In contrast, the strength of the neural network is generalization. Frequently, we re-encounter the (almost) same state in a different subtree during a next episode. Supervised learning is a natural way to generalize the already learned knowledge from a previous episode.

One of the key observations of the present paper is that we actually need both. If we *only* perform tree search, then we eventually fail at solving the domain because all information is kept locally. In contrast, if we only build trees of size 1, then we are continuously generalizing without ever locally separating decisions and improving our training targets. Our results suggest that there is actually a sweet spot halfway, where we build trees of moderate size, after which we perform a few epochs of training.

Future work will test the A0C algorithm in more complicated, continuous action space tasks (Brockman et al., 2016; Todorov et al., 2012). Moreover, our algorithm could profit from recent improvements in the MCTS algorithm (Moerland et al., 2018) and other network architectures (Szegedy et al., 2015), as also leveraged in Alpha Zero.

## 7. Conclusion

This paper introduced Alpha Zero for Continuous action space (A0C). Our method learns a continuous policy network - based on transformed Beta distributions - by minimizing a KL-divergence between the network distribution and an unnormalized density at the support points from the MCTS search. Moreover, the policy network also directs new MCTS searches by proposing new candidate child actions in the search tree. Preliminary results on the Pendulum task show that our approach does indeed learn. Future work will further explore the empirical performance of A0C. In short, A0C may be a first step in transferring the success of iterated search and learning, as observed in two-player board games with discrete action spaces (Silver et al., 2017a,b), to the single-player, continuous action space domains, like encountered in robotics, navigation and self-driving cars.

## References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Guillaume M JB Chaslot, Mark HM Winands, H Jaap Van Den Herik, Jos WHM Uiterwijk, Bruno Bouzy, et al. Progressive Strategies For Monte-Carlo Tree Search. *New Mathematics and Natural Computation (NMNC)*, 4(03):343–357, 2008.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*, pages 433–445. Springer, 2011.

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

Rémi Coulom. Computing elo ratings of move patterns in the game of go. In *Computer games workshop*, 2007.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv preprint arXiv:1801.01290*, 2018.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, volume 6, pages 282–293. Springer, 2006.

Christopher R Mansley, Ari Weinstein, and Michael L Littman. Sample-Based Planning for Continuous Action Markov Decision Processes. In *ICAPS*, 2011.

Joseph Victor Michalowicz, Jonathan M Nichols, and Frank Bucholtz. *Handbook of differential entropy*. Crc Press, 2013.

Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Efficient exploration with Double Uncertain Value Networks. *Deep Reinforcement Learning Symposium @ NIPS 2017*, 2017. arXiv preprint arXiv:1711.10789.

Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. Monte Carlo Tree Search for Asymmetric Trees. *arXiv preprint arXiv:1805.09218*, 2018.

Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529 (7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*, 2017a.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017b.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An Introduction*. MIT press Cambridge, second edition, 2018.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

Timothy Yee, Viliam Lisỳ, and Michael H Bowling. Monte Carlo Tree Search in Continuous Action Spaces with Execution Uncertainty. In *IJCAI*, pages 690–697, 2016.

## Appendix A. Monte Carlo Tree Search

We present a brief introduction of the well-known MCTS algorithm (Coulom, 2006; Browne et al., 2012). In particular, we discuss a variant of the PUCT algorithm (Rosin, 2011), as also used in Alpha Zero (Silver et al., 2017a,b). Every action node in the tree stores statistics $\{n(\boldsymbol{s},\boldsymbol{a}), W(\boldsymbol{s},\boldsymbol{a}), Q(\boldsymbol{s},\boldsymbol{a})\}$, where $n(\boldsymbol{s},\boldsymbol{a})$ is the visitation count, $W(\boldsymbol{s},\boldsymbol{a})$ the cumulative return over all roll-outs through $(\boldsymbol{s},\boldsymbol{a})$, and $Q(\boldsymbol{s},\boldsymbol{a}) = W(\boldsymbol{s},\boldsymbol{a})/n(\boldsymbol{s},\boldsymbol{a})$ is the mean action value estimate. PUCT alternates four phases:

1. **Select** In the first stage, we descent the tree from the root node according to:

$$\pi_{tree}(\boldsymbol{a}|\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} \left[ Q(\boldsymbol{s},\boldsymbol{a}) + c_{\text{puct}} \cdot \pi_\phi(\boldsymbol{a}|\boldsymbol{s}) \cdot \frac{\sqrt{n(\boldsymbol{s})}}{n(\boldsymbol{s},\boldsymbol{a})+1} \right] \tag{7}$$

   where $n(\boldsymbol{s}) = \sum_{\boldsymbol{a}} n(\boldsymbol{s},\boldsymbol{a})$ is the total number of visits to state $s$ in the tree, $c_{\text{puct}} \in \mathbb{R}^+$ is a constant that scales the amount the exploration/optimism, and $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ is the probability assigned to action $\boldsymbol{a}$ by the network.[6] The tree policy is followed until we either reach a terminal state or select an action we have not tried before.

2. **Expand** We next expand the tree with a new leaf state $\boldsymbol{s}_L$[7] obtained from simulating the environment with the new action from the last state in the current tree.

3. **Roll-out** We then require an estimate of the value $V(\boldsymbol{s}_L)$ of the new leaf node, for which MCTS uses the sum of reward of a (random) roll-out $\mathrm{R}(\boldsymbol{s}_L)$. In Alpha Zero, this gets replaced by the prediction of a value network $\mathrm{R}(\boldsymbol{s}_L) := V_\phi(\boldsymbol{s}_L)$.

4. **Back-up** Finally, we recursively back-up the results in the tree nodes. Denote the current forward trace in the tree as $\{\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{s}_1, ..\boldsymbol{s}_{L-1}, \boldsymbol{a}_{L-1}, \boldsymbol{s}_L\}$. Then, for each state-action edge $(\boldsymbol{s}_i, \boldsymbol{a}_i)$, $L > i \geq 0$, we recursively estimate the state-action value as

$$\mathrm{R}(\boldsymbol{s}_i, \boldsymbol{a}_i) = r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \gamma \mathrm{R}(\boldsymbol{s}_{i+1}, \boldsymbol{a}_{i+1}). \tag{8}$$

   where $\mathrm{R}(\boldsymbol{s}_L, \boldsymbol{a}_L) := \mathrm{R}(\boldsymbol{s}_L)$. We then increment $W(\boldsymbol{s}_i, \boldsymbol{a}_i)$ with the new estimate $\mathrm{R}(\boldsymbol{s}_i, \boldsymbol{a}_i)$, increment the visitation count $n(\boldsymbol{s}_i, \boldsymbol{a}_i)$ with 1, and set the mean estimate to $Q(\boldsymbol{s}_i, \boldsymbol{a}_i) = W(\boldsymbol{s}_i, \boldsymbol{a}_i)/n(\boldsymbol{s}_i, \boldsymbol{a}_i)$. We repeatedly apply this back-up one step higher in the tree until we reach the root node $s_0$.

[6] This equation differs from the standard UCT-like formulas in two ways. The $\pi_\phi(a|s)$ term scales the confidence interval based on prior knowledge, as stored in the the policy network. The $+1$ term in the denominator ensures that the policy prior already affects the decision when there are unvisited actions. Otherwise, every untried action would be tried at least once, since without the +1 term Eq. 9 becomes $\infty$ for untried actions. This is undesirable for large action spaces and small trees, where we directly want to prune the actions that we already know are inferior from prior experience.

[7] We use superscript $\boldsymbol{s}^t$ to index real environment states and actions, subscripts $\boldsymbol{s}_d$ to index states and actions at depth $d$ in the search tree, and double subscripts $\boldsymbol{a}_{d,j}$ to index a specific child action $j$ at depth $d$. For example, $\boldsymbol{a}_{0,0}$ is the first child action at the root $\boldsymbol{s}_0$. At every timestep $t$, the tree root $\boldsymbol{s}_0 := \boldsymbol{s}^t$, i.e. the current environment state becomes the tree root.

This procedure is repeated until the overall MCTS trace budget $N_{\text{trace}}$ is reached. MCTS returns a set of root actions $A_0 = \{\boldsymbol{a}_{0,0}, \boldsymbol{a}_{0,1}, .., \boldsymbol{a}_{0,m}\}$ with associated counts $N_0 = \{n(\boldsymbol{s}_0, \boldsymbol{a}_{0,0}), n(\boldsymbol{s}_0, \boldsymbol{a}_{0,1}), .., n(\boldsymbol{s}_0, \boldsymbol{a}_{0,m})\}$. Here $m$ denotes the number of child actions, which for Alpha Zero is always fixed to the cardinality of the discrete action space $m = |\mathcal{A}|$. We select the real action $\boldsymbol{a}^t$ to play in the environment by sampling from the probability distribution obtained from normalizing the action counts at the root $\boldsymbol{s}_0 (= \boldsymbol{s}^t)$:

$$\boldsymbol{a}^t \sim \hat{\pi}(\boldsymbol{a}|\boldsymbol{s}_0), \quad \text{where} \quad \hat{\pi}(\boldsymbol{a}|\boldsymbol{s}_0) = \frac{n(\boldsymbol{s}_0, \boldsymbol{a})}{n(\boldsymbol{s}_0)} \tag{9}$$

and $n(\boldsymbol{s}_0) = \sum_{\boldsymbol{b} \in A_0} n(\boldsymbol{s}_0, \boldsymbol{b})$. Note that $n(\boldsymbol{s}_0) \geq N_{\text{trace}}$, since we store the subtree that belongs to the picked action $\boldsymbol{a}^t$ for the MCTS at the next timestep.

### A.1 MCTS with Progressive Widening for Continuous Action Space

During MCTS with a discrete action space we evaluate the PUCT formula for *all* actions. However, in continuous action space we can not enumerate all actions, i.e., there are actually infinitely many actions in a continuous set. A practical solution to this problem is *progressive widening* (Coulom, 2007; Chaslot et al., 2008), where we make the number of child actions of state $\boldsymbol{s}$ in the tree $m(\boldsymbol{s})$ a function of the total number of visits to that state $n(\boldsymbol{s})$. This implies that actions with good returns, which will get more visits, will also gradually get more child actions for consideration. In particular, Couëtoux et al. (2011) uses

$$m(\boldsymbol{s}) = c_{pw} \cdot n(\boldsymbol{s})^{\kappa} \tag{10}$$

for constants $c_{pw} \in \mathbb{R}^+$ and $\kappa \in (0, 1)$, making $m(\boldsymbol{s})$ a polynomial (root) function of $n(\boldsymbol{s})$. The idea of progressive widening was introduced by Coulom (2007), who made $m(\boldsymbol{s})$ a logarithmic function of $n(\boldsymbol{s})$. Although originally conceived for discrete domains, this technique turns out to be an effective solution for continuous action space as well (Couëtoux et al., 2011).

Finally, we note that other methods for MCTS in continuous action space have been proposed as well. Yee et al. (2016) extends the progressive widening idea with kernel regression, to share statistical strength if the child states of two actions are similar. Mansley et al. (2011) introduced Hierarchical Optimistic Optimization applied to Trees (HOOT), which recursively partitions the continuous action space in piecewise segments. This approach has the benefit of generalization over actions and removes any discretization hyperparameters, although it does come at additional computational cost during action selection (linear in the number of samples for UCT, quadratic for HOOT). Moreover, the progressive widening approach is conceptually closer to the count-based policy derivation in AlphaZero Silver et al. (2017b), and is therefore the method of choice for continuous MCTS in this paper.

## Appendix B. Enforcing Action Space Bounds with Transformed Beta Distributions

Continuous action spaces are generally bounded, i.e., we want to sample $\boldsymbol{a} \in [-c_b, c_b]^{n_a}$ for some constant $c_b \in \mathbb{R}^+$ and action space dimensionality $n_a$. There are various probability distributions with support on a continuous bounded interval. A well-known and flexible option is the Beta distribution, which has support in $[0,1]$. We will therefore make our network predict the parameters of a factorized Beta distribution $\boldsymbol{u} \sim q(\boldsymbol{u})$, where each element $u_i \sim \text{Beta}(\alpha_i(\phi), \beta_i(\phi))$. Our goal is to transform the random variable $\boldsymbol{u}$ to a random variable $\boldsymbol{a}$ with support $\boldsymbol{a} \in [-c_b, c_b]^{n_a}$. A simple transformation $g$ that achieves this goal is

$$\boldsymbol{a} = g(\boldsymbol{u}) = c_b \cdot (2\boldsymbol{u} - 1) \tag{11}$$

For the loss specification in the paper, we require the (log)-density $\pi(\boldsymbol{a})$ of the transformed variable. We know from the change of variables rule that:

$$\pi(\boldsymbol{a}) = q(\boldsymbol{u}) \Big| \det(\frac{\mathrm{d}\boldsymbol{a}}{\mathrm{d}\boldsymbol{u}}) \Big|^{-1} \tag{12}$$

For the transformation $\boldsymbol{a} = g(\boldsymbol{u})$, the Jacobian $\frac{\mathrm{d}\boldsymbol{a}}{\mathrm{d}\boldsymbol{u}} = \text{diag}(2c_b)$ is a diagonal matrix. Therefore, we can derive a simple expression for the (log-)likelihood of $\boldsymbol{a}$:

$$\pi(\boldsymbol{a}) = q(\boldsymbol{u}) \cdot (2c_b)^{-n_a}, \quad \text{and} \quad \log \pi(\boldsymbol{a}) = \log q(\boldsymbol{u}). - n_a \cdot \log(2c_b). \tag{13}$$

### B.1 Entropy of Transformed Beta Distribution

We know the entropy of a linear transformation of some variable from differential entropy (Michalowicz et al., 2013). For a linear transformation $\boldsymbol{M}\boldsymbol{u} + \boldsymbol{l}$, with matrix $\boldsymbol{M}$ and vector $\boldsymbol{l}$, we have

$$H(\boldsymbol{M}\boldsymbol{u} + \boldsymbol{l}) = H(\boldsymbol{u}) + \log |\det(\boldsymbol{M})| \tag{14}$$

For our transformation $g(\boldsymbol{u})$ (Eq. 11), the second term of this equation equals $n_a \log(2c_b)$. Since this term does not depend on $\phi$, and therefore does not contribute any gradients, we will simply ignore it. The entropy of the Beta distribution $q(\boldsymbol{u})$ can be computed analytically (Michalowicz et al. (2013), p.63).

## Appendix C. Policy Entropy Regularization

Continuous policies have a risk to collapse (Haarnoja et al., 2018). If all sampled actions are close to each other, then the distribution may narrow too much, loosing any exploration. In the worst case, the distribution may completely collapse, which will produce NaNs and break the training process. As we empirically observed this problem, we augment the training objective with an entropy maximization term. This prevents the policy from collapsing, and additionally ensures a minimum level of exploration. We define the entropy loss as

$$\mathcal{L}^H(\phi) = H(\pi_\phi(\boldsymbol{a}|\boldsymbol{s})) = -\int \pi_\phi(\boldsymbol{a}|\boldsymbol{s}) \log \pi_\phi(\boldsymbol{a}|\boldsymbol{s}) \mathrm{d}a. \tag{15}$$

Details on the computation of the entropy for the case where $\pi_\phi(\boldsymbol{a}|\boldsymbol{s})$ is a transformed Beta distribution are provided in Appendix B.1. The full policy loss thereby becomes

$$\mathcal{L}^\pi(\phi) = \mathcal{L}^{\text{policy}}(\phi) - \lambda \mathcal{L}^H(\phi), \tag{16}$$

where $\lambda$ is a hyperparameter that scales the contribution of the entropy term to the overall loss.

## Appendix D. Implementation details

We use a three layer neural network with 128 units in each hidden layer and ELu activation functions. For the MCTS we set $c_{\text{puct}} = 0.05$, $c_{\text{pw}} = 1$ and $\kappa = 0.5$, and for the policy loss $\lambda = 0.1$ and $\tau = 0.1$. We train the networks in Tensorflow (Abadi et al., 2016), using RMSProp optimizer on mini-batches of size 32 with a learning rate of 0.0001. Episodes last at maximum 300 steps.