TD3, Monte Carlo Tree Search

Milan Straka

December 09, 2019
Continuous Action Space

Until now, the actions were discrete. However, many environments naturally accept actions from continuous space. We now consider actions which come from range \([a, b]\) for \(a, b \in \mathbb{R}\), or more generally from a Cartesian product of several such ranges:

\[
\prod_i [a_i, b_i].
\]

A simple way how to parametrize the action distribution is to choose them from the normal distribution. Given mean \(\mu\) and variance \(\sigma^2\), probability density function of \(\mathcal{N}(\mu, \sigma^2)\) is

\[
p(x) \overset{\text{def}}{=} \frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.
\]
Continuous Action Space in Gradient Methods

Utilizing continuous action spaces in gradient-based methods is straightforward. Instead of the softmax distribution we suitably parametrize the action value, usually using the normal distribution. Considering only one real-valued action, we therefore have

\[
\pi(a|s; \theta) \overset{\text{def}}{=} P\left(a \sim \mathcal{N}(\mu(s; \theta), \sigma(s; \theta)^2)\right),
\]

where \(\mu(s; \theta)\) and \(\sigma(s; \theta)\) are function approximation of mean and standard deviation of the action distribution.

The mean and standard deviation are usually computed from the shared representation, with

- the mean being computed as a regular regression (i.e., one output neuron without activation);
- the standard variance (which must be positive) being computed again as a regression, followed most commonly by either \(\exp\) or \(\text{softplus}\), where \(\text{softplus}(x) \overset{\text{def}}{=} \log(1 + e^x)\).
Combining continuous actions and Deep Q Networks is not straightforward. In order to do so, we need a different variant of the policy gradient theorem.

Recall that in policy gradient theorem,

$$\nabla_{\theta} J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} q_\pi(s, a) \nabla_\theta \pi(a|s; \theta).$$

**Deterministic Policy Gradient Theorem**

Assume that the policy $\pi(s; \theta)$ is deterministic and computes an action $a \in \mathbb{R}$. Then under several assumptions about continuousness, the following holds:

$$\nabla_{\theta} J(\theta) \propto \mathbb{E}_{s \sim \mu(s)} \left[ \nabla_{\theta} \pi(s; \theta) \nabla_a q_\pi(s, a) \big|_{a=\pi(s; \theta)} \right].$$

The theorem was first proven in the paper Deterministic Policy Gradient Algorithms by David Silver et al.
Deep Deterministic Policy Gradients

Note that the formulation of deterministic policy gradient theorem allows an off-policy algorithm, because the loss functions no longer depends on actions (similarly to how expected Sarsa is also an off-policy algorithm).

We therefore train function approximation for both $\pi(s; \theta)$ and $q(s, a; \theta)$, training $q(s, a; \theta)$ using a deterministic variant of the Bellman equation:

$$q(S_t, A_t; \theta) = \mathbb{E}_{R_{t+1}, S_{t+1}} \left[ R_{t+1} + \gamma q(S_{t+1}, \pi(S_{t+1}; \theta)) \right]$$

and $\pi(s; \theta)$ according to the deterministic policy gradient theorem.

The algorithm was first described in the paper Continuous Control with Deep Reinforcement Learning by Timothy P. Lillicrap et al. (2015).

The authors utilize a replay buffer, a target network (updated by exponential moving average with $\tau = 0.001$), batch normalization for CNNs, and perform exploration by adding a normal-distributed noise to predicted actions. Training is performed by Adam with learning rates of $1e^{-4}$ and $1e^{-3}$ for the policy and critic network, respectively.
Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'}$, $\theta^\mu' \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $N$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^\mu')|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
    $$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}
$$
    Update the target networks:
    $\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$
  end for
end for

Algorithm 1 of the paper *Continuous Control with Deep Reinforcement Learning* by Timothy P. Lillicrap et al.
The paper Addressing Function Approximation Error in Actor-Critic Methods by Scott Fujimoto et al. from February 2018 proposes improvements to DDPG which

- decrease maximization bias by training two critics and choosing minimum of their predictions;
- introduce several variance-lowering optimizations:
  - delayed policy updates;
  - target policy smoothing.
Similarly to Q-learning, the DDPG algorithm suffers from maximization bias. In Q-learning, the maximization bias was caused by the explicit max operator. For DDPG methods, it can be caused by the gradient descent itself. Let $\theta_{\text{approx}}$ be the parameters maximizing the $q_\theta$ and let $\theta_{\text{true}}$ be the hypothetical parameters which maximise true $q_\pi$, and let $\pi_{\text{approx}}$ and $\pi_{\text{true}}$ denote the corresponding policies.

Because the gradient direction is a local maximizer, for sufficiently small $\alpha < \varepsilon_1$ we have

$$\mathbb{E}\left[q_\theta(s, \pi_{\text{approx}})\right] \geq \mathbb{E}\left[q_\theta(s, \pi_{\text{true}})\right].$$

However, for real $q_\pi$ and for sufficiently small $\alpha < \varepsilon_2$ it holds that

$$\mathbb{E}\left[q_\pi(s, \pi_{\text{true}})\right] \geq \mathbb{E}\left[q_\pi(s, \pi_{\text{approx}})\right].$$

Therefore, if $\mathbb{E}\left[q_\theta(s, \pi_{\text{true}})\right] \geq \mathbb{E}\left[q_\pi(s, \pi_{\text{true}})\right]$, for $\alpha < \min(\varepsilon_1, \varepsilon_2)$

$$\mathbb{E}\left[q_\theta(s, \pi_{\text{approx}})\right] \geq \mathbb{E}\left[q_\pi(s, \pi_{\text{approx}})\right].$$
Analogously to Double DQN we could compute the learning targets using the current policy and the target critic, i.e., \( r + \gamma q_{\theta'}(s', \pi_{\varphi}(s')) \) (instead of using target policy and target critic as in DDPG), obtaining DDQN-AC algorithm. However, the authors found out that the policy changes too slowly and the target and current networks are too similar.

Using the original Double Q-learning, two pairs of actors and critics could be used, with the learning targets computed by the opposite critic, i.e., \( r + \gamma q_{\theta_2}(s', \pi_{\varphi_1}(s)) \) for updating \( q_{\theta_1} \).

The resulting DQ-AC algorithm is slightly better, but still suffering from oversetimation.
The authors instead suggest to employ two critics and one actor. The actor is trained using one of the critics, and both critics are trained using the same target computed using the \textit{minimum} value of both critics as

\[ r + \gamma \min_{i=1,2} q_{\theta_i'}(s', \pi_{\varphi'}(s')). \]

Furthermore, the authors suggest two additional improvements for variance reduction.

- For obtaining higher quality target values, the authors propose to train the critics more often. Therefore, critics are updated each step, but the actor and the target networks are updated only every \textit{d}-th step (\textit{d} = 2 is used in the paper).

- To explicitly model that similar actions should lead to similar results, a small random noise is added to performed actions when computing the target value:

\[
r + \gamma \min_{i=1,2} q_{\theta_i'}(s', \pi_{\varphi'}(s') + \varepsilon) \quad \text{for} \quad \varepsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c).
\]
**Algorithm 1 TD3**

Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$
with random parameters $\theta_1$, $\theta_2$, $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $B$

for $t = 1$ to $T$ do

Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
$\epsilon \sim N(0, \sigma)$ and observe reward $r$ and new state $s'$
Store transition tuple $(s, a, r, s')$ in $B$

Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $B$
$\tilde{a} \leftarrow \pi_\phi'(s') + \epsilon$, $\epsilon \sim \text{clip}(N(0, \tilde{\sigma}), -c, c)$
y $\leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i}(s', \tilde{a})$
Update critics $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum(y - Q_{\theta_i}(s, a))^2$
if $t$ mod $d$ then

Update $\phi$ by the deterministic policy gradient:
$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
Update target networks:
$\theta_i' \leftarrow \tau \theta_i + (1 - \tau) \theta_i'$
$\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$
end if

end for

*Algorithm 1 of the paper "Addressing Function Approximation Error in Actor-Critic Methods" by Scott Fujimoto et al.*
Table 3 of the paper "Addressing Function Approximation Error in Actor-Critic Methods" by Scott Fujimoto et al.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Ours</th>
<th>DDPG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critic Learning Rate</td>
<td>$10^{-3}$</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Critic Regularization</td>
<td>None</td>
<td>$10^{-2} \cdot</td>
</tr>
<tr>
<td>Actor Learning Rate</td>
<td>$10^{-3}$</td>
<td>$10^{-4}$</td>
</tr>
<tr>
<td>Actor Regularization</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
<td>Adam</td>
</tr>
<tr>
<td>Target Update Rate ($\tau$)</td>
<td>$5 \cdot 10^{-3}$</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Batch Size</td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>Iterations per time step</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Discount Factor</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Reward Scaling</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Normalized Observations</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Gradient Clipping</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Exploration Policy</td>
<td>$\mathcal{N}(0, 0.1)$</td>
<td>OU, $\theta = 0.15$, $\mu = 0$, $\sigma = 0.2$</td>
</tr>
</tbody>
</table>
Figure 5 of the paper “Addressing Function Approximation Error in Actor-Critic Methods” by Scott Fujimoto et al.

Table 1 of the paper “Addressing Function Approximation Error in Actor-Critic Methods” by Scott Fujimoto et al.

<table>
<thead>
<tr>
<th>Environment</th>
<th>TD3</th>
<th>DDPG</th>
<th>our DDPG</th>
<th>PPO</th>
<th>TRPO</th>
<th>ACKTR</th>
<th>SAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HalfCheetah-v1</td>
<td>9636.95 ± 859.065</td>
<td>3305.60</td>
<td>8577.29</td>
<td>1795.43</td>
<td>-15.57</td>
<td>1450.46</td>
<td>2347.19</td>
</tr>
<tr>
<td>Hopper-v1</td>
<td>3564.07 ± 114.74</td>
<td>2020.46</td>
<td>1860.02</td>
<td>2164.70</td>
<td>2471.30</td>
<td>2428.39</td>
<td>2996.66</td>
</tr>
<tr>
<td>Walker2d-v1</td>
<td>4682.82 ± 539.64</td>
<td>1843.85</td>
<td>3098.11</td>
<td>3317.69</td>
<td>2321.47</td>
<td>1216.70</td>
<td>1283.67</td>
</tr>
<tr>
<td>Ant</td>
<td>4372.44 ± 1000.33</td>
<td>1005.30</td>
<td>888.77</td>
<td>1083.20</td>
<td>-75.85</td>
<td>1821.94</td>
<td>655.35</td>
</tr>
<tr>
<td>Reacher-v1</td>
<td>-3.60 ± 0.56</td>
<td>-6.51</td>
<td>-4.01</td>
<td>-6.18</td>
<td>-111.43</td>
<td>-4.26</td>
<td>-4.44</td>
</tr>
<tr>
<td>InvertedPendulum-v1</td>
<td>1000.00</td>
<td>1000.00</td>
<td>1000.00</td>
<td>985.40</td>
<td>1000.00</td>
<td>1000.00</td>
<td>1000.00</td>
</tr>
<tr>
<td>InvertedDoublePendulum-v1</td>
<td>9337.47 ± 14.96</td>
<td>9355.52</td>
<td>8369.95</td>
<td>8977.94</td>
<td>205.85</td>
<td>9081.92</td>
<td>8487.15</td>
</tr>
</tbody>
</table>
TD3 – Ablations

Figure 7 of the paper "Addressing Function Approximation Error in Actor-Critic Methods" by Scott Fujimoto et al.

Figure 8 of the paper "Addressing Function Approximation Error in Actor-Critic Methods" by Scott Fujimoto et al.
## TD3 – Ablations

<table>
<thead>
<tr>
<th>Method</th>
<th>H.Cheetah</th>
<th>Hopper</th>
<th>Walker2d</th>
<th>Ant</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD3</td>
<td>9532.99</td>
<td>3304.75</td>
<td>4565.24</td>
<td>4185.06</td>
</tr>
<tr>
<td>DDPG</td>
<td>3162.50</td>
<td>1731.94</td>
<td>1520.90</td>
<td>816.35</td>
</tr>
<tr>
<td>AHE</td>
<td>8401.02</td>
<td>1061.77</td>
<td>2362.13</td>
<td>564.07</td>
</tr>
<tr>
<td>AHE + DP</td>
<td>7588.64</td>
<td>1465.11</td>
<td>2459.53</td>
<td>896.13</td>
</tr>
<tr>
<td>AHE + TPS</td>
<td>9023.40</td>
<td>907.56</td>
<td>2961.36</td>
<td>872.17</td>
</tr>
<tr>
<td>AHE + CDQ</td>
<td>6470.20</td>
<td>1134.14</td>
<td>3979.21</td>
<td>3818.71</td>
</tr>
<tr>
<td>TD3 - DP</td>
<td>9590.65</td>
<td>2407.42</td>
<td>4695.50</td>
<td>3754.26</td>
</tr>
<tr>
<td>TD3 - TPS</td>
<td>8987.69</td>
<td>2392.59</td>
<td>4033.67</td>
<td>4155.24</td>
</tr>
<tr>
<td>TD3 - CDQ</td>
<td>9792.80</td>
<td>1837.32</td>
<td>2579.39</td>
<td>849.75</td>
</tr>
<tr>
<td>DQ-AC</td>
<td>9433.87</td>
<td>1773.71</td>
<td>3100.45</td>
<td>2445.97</td>
</tr>
<tr>
<td>DDQN-AC</td>
<td>10306.90</td>
<td>2155.75</td>
<td>3116.81</td>
<td>1092.18</td>
</tr>
</tbody>
</table>

*Table 2 of the paper “Addressing Function Approximation Error in Actor-Critic Methods” by Scott Fujimoto et al.*
Deep Reinforcement Learning Overview

We can classify the approaches visited so far into several categories:

- **deep Q networks**: Applicable only for not many discrete actions, a network is used to estimate the action-value function $q_\pi(s, a)$. Can be trained using an effective off-policy algorithm without explicit importance sampling corrections (but requires replay buffer).

- **policy gradient**: REINFORCE and Actor-Critic algorithms training a policy over the actions. The policy can be generally any distribution, so apart from categorical distribution for discrete actions any continuous distribution can be used. The algorithms are inherently on-policy, so importance sampling factors must be used for off-policy training. Is often combined with a value network working as a baseline and/or TD bootstrap.

- **deterministic policy gradient**: For deterministic continuous policies only, paired with a state-action value network critic. Offers off-policy training algorithm.
On 7 December 2018, the AlphaZero paper came out in Science journal. It demonstrates learning chess, shogi and go, *tabula rasa* – without any domain-specific human knowledge or data, only using self-play. The evaluation is performed against strongest programs available.

![Figure 2 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.](image)

<table>
<thead>
<tr>
<th>Chess</th>
<th>Shogi</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaZero vs. Stockfish</td>
<td>AlphaZero vs. Elmo</td>
<td>AlphaZero vs. AG0</td>
</tr>
<tr>
<td>W: 29.0% D: 70.6% L: 0.4%</td>
<td>W: 84.2% D: 2.2% L: 13.6%</td>
<td>W: 68.9%</td>
</tr>
<tr>
<td>○</td>
<td>W: 98.2% D: 0.0% L: 1.8%</td>
<td>L: 31.1%</td>
</tr>
<tr>
<td>●</td>
<td>W: 53.7%</td>
<td>L: 46.3%</td>
</tr>
</tbody>
</table>
AlphaZero – Overview

AlphaZero uses a neural network which using the current state $s$ predicts $(p, v) = f(s; \theta)$, where:

- $p$ is a vector of move probabilities, and
- $v$ is expected outcome of the game in range $[-1, 1]$.

Instead of usual alpha-beta search used by classical game playing programs, AlphaZero uses Monte Carlo Tree Search (MCTS). By a sequence of simulated self-play games, the search can improve the estimate of $p$ and $v$, and can be considered a powerful policy evaluation operator – given a network $f$ predicting policy $p$ and value estimate $v$, MCTS produces a more accurate policy $\pi$ and better value estimate $w$ for a given state $s$:

$$(\pi, w) \leftarrow \text{MCTS}(p, v, f) \text{ for } (p, v) = f(s; \theta).$$
The network is trained from self-play games. The game is played by repeatedly running MCTS from the state $s_t$ and choosing a move $a_t \sim \pi_t$, until a terminal position $s_T$ is encountered, which is scored according to game rules as $z \in \{-1, 0, 1\}$. Finally, the network parameters are trained to minimize the error between the predicted outcome $v$ and simulated outcome $z$, and maximize the similarity of the policy vector $p_t$ and the search probabilities $\pi_t$:

$$L \overset{\text{def}}{=} (z - v)^2 + \pi^T \log p + c||\theta||^2.$$

The loss is a combination of:

- a mean squared error for the value functions;
- a crossentropy/KL divergence for the action distribution;
- L2 regularization
MCTS keeps a tree of currently explored states from a fixed root state. Each node corresponds to a game state. Each state-action pair \((s, a)\) stores the following set of statistics:

- visit count \(N(s, a)\),
- total action-value \(W(s, a)\),
- mean action value \(Q(s, a) \triangleq W(s, a)/N(s, a)\),
- prior probability \(P(s, a)\) of selecting action \(a\) in state \(s\).

Each simulation starts in the root node and finishes in a leaf node \(s_L\). In a state \(s_t\), an action is selected using a variant of PUCT algorithm as \(a_t = \arg \max_a (Q(s_t, a) + U(s_t, a))\), where

\[
U(s, a) \triangleq C(s)P(s, a) \frac{\sqrt{N(s)}}{1 + N(s, a)},
\]

with \(C(s) = \log((1 + N(s) + c_{base})/c_{base}) + c_{init}\) being slightly time-increasing exploration rate. Additionally, exploration in \(s_{root}\) is supported by \(P(s_{root}, a) = (1 - \varepsilon)p_a + \varepsilon \text{Dir}(\alpha)\),

with \(\varepsilon = 0.25\) and \(\alpha = 0.3, 0.15, 0.03\) for chess, shogi and go, respectively.
When reaching a leaf node, it is evaluated by the network producing \((p, v)\) and all its children are initialized to \(N = W = Q = 0\), \(P = p\), and in the backward pass for all \(t \leq L\) the statistics are updated using \(N(s_t, a_t) \leftarrow N(s_t, a_t) + 1\) and \(W(s_t, a_t) \leftarrow W(s_t, a_t) + v\).

Figure 2 of the paper "Mastering the game of Go without human knowledge" by David Silver et al.
The Monte Carlo Tree Search runs usually several hundreds simulations in a single tree. The result is the vector of search probabilities recommending moves to play. This final policy is either

- proportional to visit counts $N(s_{\text{root}}, \cdot)$:
  \[
  \pi_{\text{root}}(a) \propto N(s_{\text{root}}, a)
  \]
- or a deterministic policy choosing the most visited action
  \[
  \pi_{\text{root}} = \arg \max_a (N(s_{\text{root}}, a)).
  \]

When simulating a full game, the stochastic policy is used for the first 30 moves of the game, while the deterministic is used for the rest of the moves. (This does not affect the internal MCTS search, there we always sample according to PUCT rule.)
Visualization of the 10 most visited states in a MCTS with a given number of simulations. The displayed numbers are predicted value functions from the white's perspective, scaled to $[0, 100]$ range. The border thickness is proportional to a node visit count.

*Figure 4 of the paper “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play” by David Silver et al.*
The network processes game-specific input, which consists of a history of 8 board positions encoded by several $N \times N$ planes, and some number of constant-valued inputs.

Output is considered to be a categorical distribution of possible moves. For chess and shogi, for each piece we consider all possible moves (56 queen moves, 8 knight moves and 9 underpromotions for chess).

The input is processed by:

- initial convolution block with CNN with 256 $3 \times 3$ kernels with stride 1, batch normalization and ReLU activation,
- 19 residual blocks, each consisting of two CNN with 256 $3 \times 3$ kernels with stride 1, batch normalization and ReLU activation, and a residual connection around them,
- policy head, which applies another CNN with batch normalization, followed by a convolution with 73/139 filters for chess/shogi, or a linear layer of size 362 for go,
- value head, which applies another CNN with 1 $1 \times 1$ kernel with stride 1, followed by a ReLU layer of size 256 and final tanh layer of size 1.
### AlphaZero – Network Inputs

<table>
<thead>
<tr>
<th>Feature</th>
<th>Go Planes</th>
<th>Chess Feature</th>
<th>Chess Planes</th>
<th>Shogi Feature</th>
<th>Shogi Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 stone</td>
<td>1</td>
<td>P1 piece</td>
<td>6</td>
<td>P1 piece</td>
<td>14</td>
</tr>
<tr>
<td>P2 stone</td>
<td>1</td>
<td>P2 piece</td>
<td>6</td>
<td>P2 piece</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Repetitions</td>
<td>2</td>
<td>Repetitions</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P1 prisoner count</td>
<td>7</td>
<td>P1 prisoner count</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P2 prisoner count</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colour</td>
<td>1</td>
<td>Colour</td>
<td>1</td>
<td>Colour</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total move count</td>
<td>1</td>
<td>Total move count</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P1 castling</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P2 castling</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No-progress count</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Total         | 17        | Total         | 119          | Total         | 362          |

*Table S1 of the paper “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play” by David Silver et al.*
### AlphaZero – Network Outputs

<table>
<thead>
<tr>
<th>Feature</th>
<th>Chess Planes</th>
<th>Shogi Feature</th>
<th>Shogi Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queen moves</td>
<td>56</td>
<td>Queen moves</td>
<td>64</td>
</tr>
<tr>
<td>Knight moves</td>
<td>8</td>
<td>Knight moves</td>
<td>2</td>
</tr>
<tr>
<td>Underpromotions</td>
<td>9</td>
<td>Promoting queen moves</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promoting knight moves</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drop</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>73</strong></td>
<td><strong>Total</strong></td>
<td><strong>139</strong></td>
</tr>
</tbody>
</table>

*Table S2 of the paper “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play” by David Silver et al.*
AlphaZero – Training

Training is performed by running self-play games of the network with itself. Each MCTS uses 800 simulations. A replay buffer of one million most recent games is kept.

During training, 5000 first-generation TPUs are used to generate self-play games. Simultaneously, network is trained using SGD with momentum of 0.9 on batches of size 4096, utilizing 16 second-generation TPUs. Training takes approximately 9 hours for chess, 12 hours for shogi and 13 days for go.
Figure 1 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.

Table 53 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.

<table>
<thead>
<tr>
<th></th>
<th>Chess</th>
<th>Shogi</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini-batches</td>
<td>700k</td>
<td>700k</td>
<td>700k</td>
</tr>
<tr>
<td>Training Time</td>
<td>9h</td>
<td>12h</td>
<td>13d</td>
</tr>
<tr>
<td>Training Games</td>
<td>44 million</td>
<td>24 million</td>
<td>140 million</td>
</tr>
<tr>
<td>Thinking Time</td>
<td>800 sims</td>
<td>800 sims</td>
<td>800 sims</td>
</tr>
<tr>
<td></td>
<td>~ 40 ms</td>
<td>~ 80 ms</td>
<td>~ 200 ms</td>
</tr>
</tbody>
</table>
According to the authors, training is highly repeatable.

Figure S3 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.
In the original AlphaGo Zero, symmetries were explicitly utilized, by

- randomly sampling a symmetry during training,
- randomly sampling a symmetry during evaluation.

However, AlphaZero does not utilize symmetries in any way (because chess and shogi do not have them).

Figure S1 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.
During inference, AlphaZero utilizes much less evaluations than classical game playing programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Chess</th>
<th>Shogi</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaZero</td>
<td>63k (13k)</td>
<td>58k (12k)</td>
<td>16k (0.6k)</td>
</tr>
<tr>
<td>Stockfish</td>
<td>58,100k (24,000k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elmo</td>
<td></td>
<td>25,100k (4,600k)</td>
<td></td>
</tr>
<tr>
<td>AlphaZero</td>
<td>1.5 GFlop</td>
<td>1.9 GFlop</td>
<td>8.5 GFlop</td>
</tr>
</tbody>
</table>

Table S4 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.
### AlphaZero – Ablations

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Match</th>
<th>Start Position</th>
<th>AlphaZero</th>
<th>Opponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Book</td>
<td>Main</td>
</tr>
<tr>
<td>2A</td>
<td>Main</td>
<td>Initial Board</td>
<td>No 3h 15s</td>
<td>No</td>
</tr>
<tr>
<td>2B</td>
<td>1/100 time</td>
<td>Initial Board</td>
<td>No 108s 0.15s</td>
<td>No</td>
</tr>
<tr>
<td>2B</td>
<td>1/30 time</td>
<td>Initial Board</td>
<td>No 6min 0.5s</td>
<td>No</td>
</tr>
<tr>
<td>2B</td>
<td>1/10 time</td>
<td>Initial Board</td>
<td>No 18min 1.5s</td>
<td>No</td>
</tr>
<tr>
<td>2B</td>
<td>1/3 time</td>
<td>Initial Board</td>
<td>No 1h 5s</td>
<td>No</td>
</tr>
<tr>
<td>2C</td>
<td>latest Stockfish</td>
<td>Initial Board</td>
<td>No 3h 15s</td>
<td>No</td>
</tr>
<tr>
<td>2C</td>
<td>Opening Book</td>
<td>Initial Board</td>
<td>No 3h 15s</td>
<td>Yes</td>
</tr>
<tr>
<td>2D</td>
<td>Human Openings</td>
<td>Figure 3A</td>
<td>No 3h 15s</td>
<td>No</td>
</tr>
<tr>
<td>2D</td>
<td>TCEC Openings</td>
<td>Figure S4</td>
<td>No 3h 15s</td>
<td>No</td>
</tr>
</tbody>
</table>

Table S8 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Match</th>
<th>Start Position</th>
<th>AlphaZero</th>
<th>Opponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Book</td>
<td>Main</td>
</tr>
<tr>
<td>2A</td>
<td>Main</td>
<td>Initial Board</td>
<td>No 3h 15s</td>
<td>Yes</td>
</tr>
<tr>
<td>2B</td>
<td>1/100 time</td>
<td>Initial Board</td>
<td>No 108s 0.15s</td>
<td>Yes</td>
</tr>
<tr>
<td>2B</td>
<td>1/30 time</td>
<td>Initial Board</td>
<td>No 6min 0.5s</td>
<td>Yes</td>
</tr>
<tr>
<td>2B</td>
<td>1/10 time</td>
<td>Initial Board</td>
<td>No 18min 1.5s</td>
<td>Yes</td>
</tr>
<tr>
<td>2B</td>
<td>1/3 time</td>
<td>Initial Board</td>
<td>No 1h 5s</td>
<td>Yes</td>
</tr>
<tr>
<td>2C</td>
<td>Aperyqhapaq</td>
<td>Initial Board</td>
<td>No 3h 15s</td>
<td>No</td>
</tr>
<tr>
<td>2C</td>
<td>CSA time control</td>
<td>Initial Board</td>
<td>No 10min 10s</td>
<td>Yes</td>
</tr>
<tr>
<td>2D</td>
<td>Human Openings</td>
<td>Figure 3B</td>
<td>No 3h 15s</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table S9 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.
# AlphaZero – Ablations

### Figure 2

**Chess**
- **1/100 time**
  - AlphaZero wins 76%, AlphaZero draws 22%, AlphaZero loses 2%
  - AlphaZero white wins 72%, AlphaZero black loses 2%
- **1/30 time**
  - AlphaZero wins 96%, AlphaZero draws 4%, AlphaZero loses 0%
  - AlphaZero white wins 94%, AlphaZero black loses 0%
- **1/10 time**
  - AlphaZero wins 100%, AlphaZero draws 0%, AlphaZero loses 0%
  - AlphaZero white wins 100%, AlphaZero black loses 0%
- **1/3 time**
  - AlphaZero wins 100%, AlphaZero draws 0%, AlphaZero loses 0%
  - AlphaZero white wins 100%, AlphaZero black loses 0%
- **same time**
  - AlphaZero wins 100%, AlphaZero draws 0%, AlphaZero loses 0%
  - AlphaZero white wins 100%, AlphaZero black loses 0%

**Shogi**
- **1/100 time**
  - AlphaZero wins 76%, AlphaZero draws 22%, AlphaZero loses 2%
  - AlphaZero white wins 72%, AlphaZero black loses 2%
- **1/30 time**
  - AlphaZero wins 96%, AlphaZero draws 4%, AlphaZero loses 0%
  - AlphaZero white wins 94%, AlphaZero black loses 0%
- **1/10 time**
  - AlphaZero wins 100%, AlphaZero draws 0%, AlphaZero loses 0%
  - AlphaZero white wins 100%, AlphaZero black loses 0%
- **1/3 time**
  - AlphaZero wins 100%, AlphaZero draws 0%, AlphaZero loses 0%
  - AlphaZero white wins 100%, AlphaZero black loses 0%
- **same time**
  - AlphaZero wins 100%, AlphaZero draws 0%, AlphaZero loses 0%
  - AlphaZero white wins 100%, AlphaZero black loses 0%

### C
- **Latest Stockfish**
  - AlphaZero wins 69%, AlphaZero draws 31%, AlphaZero loses 0%
  - AlphaZero white wins 66%, AlphaZero black loses 0%
- **Opening Book**
  - AlphaZero wins 69%, AlphaZero draws 31%, AlphaZero loses 0%
  - AlphaZero white wins 66%, AlphaZero black loses 0%

### D
- **Human openings**
  - AlphaZero wins 69%, AlphaZero draws 31%, AlphaZero loses 0%
  - AlphaZero white wins 66%, AlphaZero black loses 0%
- **TCEC openings**
  - AlphaZero wins 69%, AlphaZero draws 31%, AlphaZero loses 0%
  - AlphaZero white wins 66%, AlphaZero black loses 0%

---

*Figure 2 of the paper “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play” by David Silver et al.*
AlphaZero – Ablations

Figure 4 of the paper “Mastering the game of Go without human knowledge” by David Silver et al.
AlphaZero – Preferred Chess Openings

Figure S2 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.