Learning Policies from Self-Play with Policy Gradients and MCTS Value Estimates

Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne
Department of Data Science and Knowledge Engineering
Maastricht University
Maastricht, the Netherlands
{dennis.soemers,eric.piette,matthew.stephenson,cameron.browne}@maastrichtuniversity.nl

***Abstract*—In recent years, state-of-the-art game-playing agents often involve policies that are trained in self-playing processes where Monte Carlo tree search (MCTS) algorithms and trained policies iteratively improve each other. The strongest results have been obtained when policies are trained to mimic the search behaviour of MCTS by minimising a cross-entropy loss. Because MCTS, by design, includes an element of exploration, policies trained in this manner are also likely to exhibit a similar extent of exploration. In this paper, we are interested in learning policies for a project with future goals including the extraction of interpretable strategies, rather than state-of-the-art game-playing performance. For these goals, we argue that such an extent of exploration is undesirable, and we propose a novel objective function for training policies that are not exploratory. We derive a policy gradient expression for maximising this objective function, which can be estimated using MCTS value estimates, rather than MCTS visit counts. We empirically evaluate various properties of resulting policies, in a variety of board games.**

***Index Terms*—reinforcement learning, search, self-play**

I. Introduction

Monte Carlo tree search (MCTS) algorithms [1], [2], often in combination with learning algorithms, provide state-of-the-art AI in many games and other domains [3]–[6]. The most straightforward implementations of MCTS use large numbers of play-outs where actions are selected uniformly at random to estimate the value of the starting state of those play-outs. Play-outs using handcrafted heuristics, learned policies, or search to more closely resemble realistic lines of play can often significantly increase playing strength, even if the increased computational cost leads to a reduction in the number of play-outs [5], [7]–[19].

The majority of policy learning approaches use supervised learning with human expert moves as training targets, or traditional reinforcement learning (RL) update rules [20], but the most impressive results have been obtained using the Expert Iteration framework, where MCTS and a learned policy iteratively improve each other through self-play [4]–[6]. In this framework, a policy is trained to mimic the MCTS search behaviour using a cross-entropy loss, and the policy is used to bias the MCTS search. Note that play-outs are sometimes replaced altogether by trained value function estimators, leaving only the selection phase of MCTS to be biased by a trained policy [4], [6], but a learned policy may also be used to run play-outs [5].

The selection phase of MCTS provides a balance between *exploration* and *exploitation*; exploration consists of searching parts of the game tree that have not yet been thoroughly searched, and exploitation consists of searching parts of the game tree that appear the most promising based on the search process so far. Using the search behaviour of MCTS as an update target for a policy means that this policy is trained to have a similar balance between exploration and exploitation as the MCTS algorithm.

Within the context of the Digital Ludeme Project [21], we aim to learn policies based on interpretable features [22] for state-action pairs, where future goals of the project include extracting explainable strategies from learned policies, and estimating similarities or distances between different (variants of) games in terms of strategies. For the purpose of these goals, we do not expect the exploratory behaviour that is learned with the standard cross-entropy loss to be desirable.

We formulate a new training objective for policies. A policy that optimises this objective can intuitively be understood as one that selects actions such that MCTS is subsequently expected to be capable of performing well. Unlike the case where the MCTS search behaviour is used as training target, this optimisation criterion does not encourage any level of exploration. We derive an expression for the gradient of this objective with respect to a differentiable policy's parameters, which allows for training using gradient descent.

Like the standard updates used to optimise the cross-entropy loss in Expert Iteration [4]–[6], these updates are guided by "advice" generated by MCTS. This is hypothesized to be important for a stable and robust self-play learning process, with a reduced risk of overfitting to the self-play opponent. The primary difference is that this advice consists of value estimates, rather than a distribution over actions.

We empirically compare policies trained to optimise the proposed objective function, with policies trained on the standard cross-entropy loss, across a variety of deterministic, perfect-information, two-player board games. The proposed objective consistently leads to policies that are at least as strong, and in some games significantly stronger, than the cross-entropy loss. We also confirm that the resulting policies lead to significantly lower entropy in distributions over actions, which suggests that

Funded by a €2m ERC Consolidator Grant (http://ludeme.eu).

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

learned policies are less exploratory. Finally, we compare the resulting distributions of weights learned for different features, and the performance of MCTS agents biased by policies trained on the different objectives.

II. Background

This section formalises the concepts from reinforcement learning (RL) theory required in this paper. We assume a standard single-agent setting. When subsequently applying these concepts to multi-player, adversarial game settings, any states in which a learning agent is not the player to move are ignored, and moves selected by opponents are simply assumed to be a part of the "environment" and its transition dynamics.

A. Markov Decision Processes

We use the standard single-agent, fully-observable, episodic Markov decision process (MDP) setting, where $\mathcal{S}$ denotes a set of states, and $\mathcal{A}$ denotes a set of actions. At discrete time steps $t = 0, 1, \dots,$ the agent observes states $S_t \in \mathcal{S}$. Whenever S_t is not terminal, the agent selects an action $A_t \in \mathcal{A}(S_t)$ from the set of actions $\mathcal{A}(S_t)$ that are legal in S_t, which leads to an observed reward $R_{t+1} \in \mathbb{R}$. We assume that there is a fixed starting state s_0. Given a current state s and action a, the probability of observing any arbitrary successor state s' and reward r is given by $p(s', r \mid s, a) = \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$.

Let π denote some policy, such that $\pi(s, a)$ denotes the probability of selecting an action a in a state s, and $\sum_{a \in \mathcal{A}(s)} \pi(s, a) = 1$. The value $V^\pi(s)$ of a state s under policy π is given by (1):

$$V^\pi(s) \doteq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_t \sim \pi\right], \tag{1}$$

where $0 \leq \gamma \leq 1$ denotes a discount factor (in the board games applications considered in this paper, typically $\gamma = 1$). We define $R_t \doteq 0$ for $t > T$ in any episode where S_T is a terminal state. The value $Q^\pi(s, a)$ of an action a in a state s under policy π is given by (2):

$$Q^\pi(s, a) \doteq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a, A_{>0} \sim \pi\right], \tag{2}$$

where $A_{>0}$ covers all actions A_t where $t > 0$.

B. Policy Gradients

Let $J(\pi)$ denote the expected performance, in terms of returns per episode, of a policy π:

$$J(\pi) \doteq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s_0, A_t \sim \pi\right] = V^\pi(s_0). \tag{3}$$

A common goal in RL is to find a policy π such that this objective is maximised. Suppose that $\pi(\cdot, \cdot)$ is a differentiable function, parameterised by a vector $\boldsymbol{\theta}$, such that $\nabla_{\boldsymbol{\theta}} \pi(\cdot, \cdot)$ exists. Then, the Policy Gradient Theorem [23] states that:

$$\nabla_{\boldsymbol{\theta}} J(\pi) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}(s)} \nabla_{\boldsymbol{\theta}} \pi(s, a) Q^\pi(s, a), \tag{4}$$

where $d^\pi(s) \doteq \sum_{t=0}^{\infty} \gamma^t \Pr\{S_t = s \mid S_0 = s_0, A_{<t} \sim \pi\}$ gives a discounted weighting of states according to how likely they are to be reached in trajectories following π. Sample-based estimators of this gradient allow for the objective to be optimised directly, using stochastic gradient ascent to adjust the policy parameters $\boldsymbol{\theta}$ [20], [24], [25].

C. Monte Carlo Tree Search Value Estimates

Most variants of Monte Carlo tree search (MCTS) [3] can be viewed as RL approaches which, based on simulated experience, learn on-policy value estimates for the states represented by nodes in the search tree that is gradually built up [26]. Let σ denote a state from which we run an MCTS search process (meaning that σ corresponds to the root node). Then we can formally describe a policy $\mathcal{M}_\sigma$:

$$\mathcal{M}_\sigma(s, a) = \begin{cases} \frac{N(s,a)}{\sum_{a'} N(s,a')} & \text{if s in search tree,} \\ \rho(s, a) & \text{otherwise,} \end{cases} \tag{5}$$

where $N(s, a)$ denotes the number of times that the search process selected a in the node representing s, and $\rho(s, a)$ denotes the roll-out policy.

Suppose that value estimates $\hat{V}(s)$ in nodes of the search tree are computed, as is customary, as the averages of back-propagated scores, or using some other approach that can be viewed as implementing on-policy backups – such as Sarsa-UCT(λ) [26]. These value estimates are then unbiased estimators of $V^{\mathcal{M}_s}$, as defined in (1). We typically expect these value estimates to be unreliable and exhibit high variance deep in the search tree, but, given a sufficiently high MCTS iteration count, they may be more reliable close to the root node.

III. Policy Gradient with MCTS Value Estimates

Unlike the standard cross-entropy loss used in Expert Iteration, optimising the policy gradient objective of (3) does not incentivise an element of exploration in trained policies. However, this objective focuses on the long-term performance of the standalone policy π being trained. Suppose that it is infeasible to learn a good distribution $\pi(s, \cdot)$ over actions in some state s – for instance because there are no features available that allow distinguishing between any actions in s. Reaching s will then be detrimental to the long-term performance of π according to (3), and actions leading to s will therefore be disincentivized, even if they may otherwise clearly be a part of the principal variation. This is problematic when we aim to use π for purposes such as strategy extraction (even if only for some parts of the state space), rather than using it for standalone game-playing.

A. Objective Function

To address the issues illustrated above, we propose to maximise the objective function given by (6), where π is the apprentice policy to be trained, parameterised by a vector $\boldsymbol{\theta}$:

$$J_{TSPG}(\pi) \doteq \sum_{t=0}^{\infty} \mathbb{E}\left[\gamma^t R_{t+1} \mid S_0 = s_0, A_t \sim \pi, A_{\neq t} \sim \mathcal{M}\right], \tag{6}$$

where $A_{\neq t} \sim \mathcal{M}$ denotes that, for all $t' \neq t$, we run an MCTS process $\mathcal{M}_{S_{t'}}$ and sample $A_{t'}$ from $\mathcal{M}_{S_{t'}}(S_{t'}, \cdot)$. We refer to this as the *Tree-Search Policy Gradient* (TSPG) objective function. Intuitively, sampling actions $A_{t'}$ for $t' < t$ from MCTS can be understood as stating that it is only important for π to be well-trained in states that are likely to be reached when playing according to MCTS processes prior to time t. Sampling actions $A_{t'}$ for $t' > t$ from MCTS in this objective can be understood as stating that π is not required to be capable of playing well for the remainder of an episode, but only needs to be able to select actions such that MCTS would be expected to perform well in subsequent states.

Suppose that there is a small game tree, in which MCTS can easily find an optimal line of play, but where that optimal line of play leads to a subtree in which a parameterised policy π cannot play well. This may, for instance, be due to a lack of representational capacity of π itself (i.e. using a simple linear function), or due to using a restricted set of input features that is insufficient for states or actions in that subtree to be distinguished from each other. A standard RL objective function, such as the one in (3), would lead to a policy that learns to avoid that subtree altogether, because the same policy cannot guarantee long-term success in that subtree. We argue that this is detrimental for our goal of interpretable strategy extraction, because it leads to a poor strategy in the root of such a game tree. In contrast, the TSPG objective still allows for a strong strategy to be learned for states other than those in the problematic subtree.

B. Policy Gradient

Our derivation of an expression for the gradient of this objective with respect to the parameters $\boldsymbol{\theta}$ takes inspiration from the original proof for the policy gradient theorem [23]. We start by defining $V^{\pi\mathcal{M}}(s)$ as the expected value of sampling a single action from π in state s, and sampling actions from MCTS search processes for the remainder of the episode:

$$\begin{aligned} V^{\pi\mathcal{M}}(s) &\doteq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 \sim \pi, A_{>0} \sim \mathcal{M}\right] \\ &= \sum_a \pi(s,a) Q^{\mathcal{M}}(s,a), \end{aligned} \tag{7}$$

where $\mathcal{M}$ is used as a shorthand notation to indicate that a separate policy $\mathcal{M}_{S_t}$, involving a separate complete search process, is used at every time t. The gradient of this function with respect to $\boldsymbol{\theta}$ is given by:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} V^{\pi\mathcal{M}}(s) &= \nabla_{\boldsymbol{\theta}} \sum_a \pi(s,a) Q^{\mathcal{M}}(s,a) \\ &= \sum_a \Big[\nabla_{\boldsymbol{\theta}} \pi(s,a) Q^{\mathcal{M}}(s,a) \\ &\qquad + \pi(s,a) \nabla_{\boldsymbol{\theta}} Q^{\mathcal{M}}(s,a)\Big] \\ &\approx \sum_a \left[\nabla_{\boldsymbol{\theta}} \pi(s,a) Q^{\mathcal{M}}(s,a)\right], \end{aligned} \tag{8}$$

where we assume that $\nabla_{\boldsymbol{\theta}} Q^{\mathcal{M}}(s,\cdot) = 0$. Note that this assumption may be violated in practice by making use of $\boldsymbol{\theta}$ in the play-outs of MCTS processes, but it is not feasible to accurately estimate the gradient of the performance of MCTS with respect to parameters $\boldsymbol{\theta}$ used in play-outs. We can avoid violating the assumption by freezing the versions of parameters used for biasing any MCTS process, and clearing any old experience when updating parameters used by MCTS, but in practice we expect this to be detrimental to learning speed. Also note that this assumption is very similar to the omission of the $\pi(s,a)\nabla_{\mathbf{u}} Q^{\pi,\gamma}(s,a)$ term in the Off-Policy Policy-Gradient Theorem, where $\mathbf{u}$ is a parameter vector and π is a target policy [27].

Now, we rewrite the TSPG objective function to a more convenient expression, starting from (6):

$$\begin{aligned} J_{TSPG}(\pi) &\doteq \sum_{t=0}^{\infty} \mathbb{E}\left[\gamma^t R_{t+1} \mid S_0 = s_0, A_t \sim \pi, A_{\neq t} \sim \mathcal{M}\right] \\ &= \sum_{s \in \mathcal{S}} d^{\mathcal{M}}(s) V^{\pi\mathcal{M}}(s), \end{aligned} \tag{9}$$

where $d^{\mathcal{M}}(s) = \sum_{t=0}^{\infty} \gamma^t \Pr\{S_t = s \mid S_0 = s_0, A_{<t} \sim \mathcal{M}\}$. Taking the gradient with respect to $\boldsymbol{\theta}$ gives:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J_{TSPG}(\pi) &= \nabla_{\boldsymbol{\theta}} \sum_{s \in \mathcal{S}} d^{\mathcal{M}}(s) V^{\pi\mathcal{M}}(s) \\ &= \sum_{s \in \mathcal{S}} \left[\nabla_{\boldsymbol{\theta}} d^{\mathcal{M}}(s) V^{\pi\mathcal{M}}(s) + d^{\mathcal{M}}(s) \nabla_{\boldsymbol{\theta}} V^{\pi\mathcal{M}}(s)\right] \\ &\approx \sum_{s \in \mathcal{S}} d^{\mathcal{M}}(s) \sum_a \nabla_{\boldsymbol{\theta}} \pi(s,a) Q^{\mathcal{M}}(s,a), \end{aligned} \tag{10}$$

where again we assume that $\boldsymbol{\theta}$ has no effect on MCTS processes by taking $\nabla_{\boldsymbol{\theta}} d^{\mathcal{M}}(\cdot) = 0$.

The analytical expression of the gradient of the TSPG objective in (10) is exact if the involved MCTS processes are unaffected by $\boldsymbol{\theta}$, or an approximation otherwise. Note that it has a similar form to the original policy gradient expression in (4). The weighting of states and the value estimates are now both provided by $\mathcal{M}$, but the only required gradient is for $\pi(\cdot,\cdot)$ (which, by assumption, is differentiable).

C. Estimating the Gradient

In the Expert Iteration framework [4]–[6], experience is typically generated by playing self-play games where actions are selected proportional to the visit counts in root states after running MCTS processes. This corresponds precisely to the definition of policies $\mathcal{M}$ given in (5). It is customary to store states encountered in such a self-play process in a dataset $\mathcal{D}$ – keeping only one randomly-selected state per full game, to avoid excessive correlations between instances – and sample batches from $\mathcal{D}$ for stochastic gradient descent updates. Sampling batches of states $B \subseteq \mathcal{D}$ leads to unbiased estimates $\hat{g}$ of the gradient expression in (10):

$$\hat{g} = \frac{1}{|B|} \sum_{s \in B} \left[\sum_{a \in \mathcal{A}(s)} \nabla_{\boldsymbol{\theta}} \pi(s,a) Q^{\mathcal{M}}(s,a)\right]. \tag{11}$$

Optimisation of the cross-entropy loss typically used in Expert Iteration requires storing MCTS visit counts $N(s,a)$ for all $a \in \mathcal{A}(s)$ in the dataset $\mathcal{D}$, alongside the states s. Instead of storing visit counts, our approach requires storing MCTS value estimates $\hat{Q}(s,a)$ for all actions a – these are simply the state-value estimates $\hat{V}(s')$ of all successors s' of s. These values can be plugged into (11) as unbiased estimators for $Q^{\mathcal{M}}(s,a)$.

We now have an unbiased estimator of the gradient which can be readily computed from data collected as in the standard Expert Iteration self-play framework. The form of this estimator most closely resembles that of the Mean Actor-Critic [28], in the sense that we explicitly sum over all actions rather than sampling trajectories with actions selected according to π. As in the gradient estimator of the Mean Actor-Critic, it is unnecessary to subtract a state-dependent baseline from $Q^{\mathcal{M}}(s,a)$ for variance reduction, as is typically done in sample-based estimators of policy gradients [23], [25].

IV. Learning Offsets from Exploratory Policy

A differentiable policy π is typically implemented to compute logits $z(s,a) = \boldsymbol{\theta}^{\top}\boldsymbol{\phi}(s,a)$, where $\boldsymbol{\theta}$ is a trainable parameter vector and $\boldsymbol{\phi}(s,a)$ is a feature vector for a state-action pair (s,a). Probabilities $\pi(s,a)$ are subsequently computed using the softmax function; $\pi(s,a) = \frac{\exp(z(s,a))}{\sum_{a'}\exp(z(s,a'))}$. In preliminary testing, we found that there is a risk for strong features that are only discovered and added in the middle of a self-play training process [29] to remain unused. When this happens, it appears like the learning approach remains stuck in what used to be a local optimum given an older feature set, even though newly-added features should enable escaping that local optimum. First, we elaborate on why this can happen, and subsequently propose an approach to address this issue.

A. Gradients for Low-probability Actions

Suppose that π uses the softmax function, as described above. Then, the gradient of $\pi(s,a)$ with respect to the i^{th} parameter θ_i of the parameter vector $\boldsymbol{\theta}$ is given by

$$\nabla_{\theta_i}\pi(s,a) = \pi(s,a)\sum_{a'}\left(\delta_{aa'} - \pi(s,a')\right)\phi_i(s,a'), \qquad (12)$$

where the Kronecker delta $\delta_{aa'}$ is equal to 1 if $a = a'$, or 0 otherwise, and $\phi_i(s,a')$ denotes the i^{th} feature value for the state-action pair (s,a').

This is the gradient that is multiplied by $Q^{\mathcal{M}}(s,a)$ in (11) to compute the update for the parameter θ_i corresponding to the feature ϕ_i. In cases where features value $\phi_i(s,a)$ correlate strongly with state-action values $Q^{\mathcal{M}}(s,a)$, we would intuitively expect to obtain consistent, high-value gradient estimates to rapidly adapt θ_i. However, if previous learning steps – possibly taken before the feature ϕ_i was being used at all – resulted in a parameter vector $\boldsymbol{\theta}$ such that $\pi(s,a)$ is low (i.e., $\pi(s,a) \approx 0$), this gradient will also be close to zero and learning progresses very slowly.

An example in which we were consistently able to observe this problem is the game of Yavalath [30], in which players win the game by constructing lines of four pieces of their colour, but immediately lose if they first construct a line of three pieces of their colour. Fig. 1 provides a graphical representation of three features that could be used to detect winning and/or losing moves. The top feature detects winning moves that place a piece to complete a line of four, and the bottom two features detect losing moves that place pieces to complete lines of three. Note that the features that detect losing moves can be viewed as more "general" features, in the sense that they will also always be active in situations where the win-detecting feature is active.

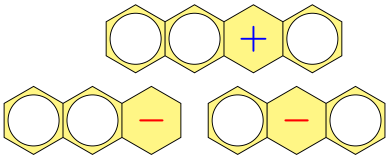

Fig. 1. Immediate win and loss features for the White player in Yavalath.

When the set of features is automatically grown over time during self-play, and more "specific" features are constructed by combining multiple more "general" features [29], the loss-detecting features are often discovered before the win-detecting features. These features are – as expected – quickly associated with negative weights, resulting in low probabilities $\pi(s,a) \approx 0$ of playing actions a in which loss-detecting features are active. When a win-detecting feature is discovered at a later point in time, the loss-detecting features result in low probabilities $\pi(s,a)$ for most situations in which the win-detecting feature also applies, leading to gradients and update steps close to 0 despite a strong correlation between feature activity and high values (winning games).

B. Exploratory Policy as Baseline

In most (sample-based) policy gradient methods [23]–[25], there is no longer a $\nabla_{\boldsymbol{\theta}}\pi(s,a)$ term in the gradient estimator. Instead of summing over all actions, updates are typically performed for actions a sampled according to $\pi(s,\cdot)$, which leads to a $\nabla_{\boldsymbol{\theta}}\log\pi(s,a)$ term in the gradient estimator. This gradient, when combined with a softmax-based policy π, no longer leads to the issue described above. However, there is a closely-related issue in that actions a with low probabilities $\pi(\cdot,a)$ are rarely sampled at all; this problem is generally viewed as a lack of exploration. This is commonly addressed by introducing an entropy regularization term in the objective function, which punishes low-entropy policies [31]. That solution is not acceptable for our goals, because it forces an element of exploration in the learned policies – this is precisely the property inherent in the standard cross-entropy-based approach of Expert Iteration that we aim to avoid. Instead, we propose to use the parameters of a more exploratory policy as a baseline, and train offsets from those parameters using our new policy gradient approach.

Consider a softmax-based policy π_{ce}, parameterised by a vector $\boldsymbol{\theta}_{ce}$, trained to minimise the standard cross-entropy loss normally used in Expert Iteration. For any given state s, this loss is given by (13), where $\boldsymbol{\mathcal{M}}_s(s)$ and $\boldsymbol{\pi}_{ce}(s)$, respectively, denote discrete probability distributions (vectors) over all actions in the state s.

$$\mathcal{L}_{ce}(s) = -\boldsymbol{\mathcal{M}}_s(s)^\top \log \boldsymbol{\pi}_{ce}(s) \tag{13}$$

Suppose that π_{ce} is defined as a softmax over linear functions of state-action features, parameterised by trainable parameters $\boldsymbol{\theta}_{ce}$, as described in the beginning of this section. Then, the gradient of this loss is given by (14):

$$\nabla_{\boldsymbol{\theta}_{ce}} \mathcal{L}_{ce}(s) = \sum_{a \in \mathcal{A}(s)} \left[\left(\pi_{ce}(s, a) - \mathcal{M}_s(s, a) \right) \times \phi(s, a) \right] \tag{14}$$

Note that, unlike the gradient in (12), this gradient does not suffer from the problem that the magnitudes of gradient-based updates are close to 0 when the trainable policy (in this case π_{ce}) has (incorrectly) converged to parameters that result in near-zero probabilities for certain state-action pairs. In the example situation described above for Yavalath, we indeed find that a policy trained to minimise this cross-entropy loss is capable of learning high weights for win-detecting features quickly after the feature itself is first introduced.

We propose to exploit this advantage of the cross-entropy loss by defining the logits $z(s, a)$ that are plugged into the softmax of a TSPG-based policy π_{tspg} (trained to maximise the TSPG objective of (6)) as follows:

$$z(s, a) = \left(\boldsymbol{\theta}_{ce} + \boldsymbol{\theta}_{tspg} \right)^\top \boldsymbol{\phi}(s, a). \tag{15}$$

Here, $\boldsymbol{\theta}_{ce}$ denotes a parameter vector of a policy π_{ce} trained to minimise the cross-entropy loss – a more "exploratory" policy which learns to mimic the exploratory behaviour of MCTS. When training the policy π_{tspg} to maximise (6), we freeze $\boldsymbol{\theta}_{ce}$ and only allow the parameters $\boldsymbol{\theta}_{tspg}$ to be adjusted. This leaves all the gradients and estimators in Section III unchanged. The parameters $\boldsymbol{\theta}_{ce}$ can be viewed as a smart "initialisation" of parameters, which is dynamic and can change over time due to its own learning process. The parameters $\boldsymbol{\theta}_{tspg}$ can be viewed as "offsets", and the sum of parameters $\boldsymbol{\theta}_{ce} + \boldsymbol{\theta}_{tspg}$ are then the parameters that actually optimise the TSPG objective.

V. Experiments

This section describes a number of experiments carried out to compare policies trained to minimise the standard cross-entropy loss of (13) with policies trained to maximise the TSPG objective of (6). All experiments are carried out using a variety of deterministic, adversarial, two-player, perfect information board games.

A. Setup

All policies are trained using self-play Expert Iteration processes [4]–[6]. The policies are all defined as linear functions of state-action features [22], transformed into probability distributions using a softmax, as described in Section IV. The sets of features grow automatically throughout self-play [29].

Experience is generated in self-play, where all players are identical MCTS agents. They use the same PUCT strategy as AlphaGo Zero [4] for the selection phase, with an exploration constant of 2.5, and a policy π_{ce} trained to minimise cross-entropy loss providing bias. All value estimates are in the range $[-1, 1]$, where -1 corresponds to losses, 0 to ties, and 1 to wins. In the selection phase, unvisited actions are not automatically prioritised; they are assigned a value estimate equal to the value estimate of the parent node. We experiment with policies trained on the cross-entropy objective, as well as policies trained on the TSPG objective, for the play-out phase. Every turn, MCTS re-uses the relevant subtree of the complete search tree generated in previous turns, and runs 1600 additional MCTS iterations (800 in Hex on the 11×11 board, due to high computation time). Actions in self-play are selected proportional to the MCTS visit counts (i.e. sampled from the $\mathcal{M}_s$ distributions in root states s).

Every training run described in this section consists of 200 sequential games of self-play. For every state s encountered in self-play, we store a tuple $\langle s, \mathcal{M}_s, \mathcal{Q}_s \rangle$ in an experience buffer, where $\mathcal{M}_s$ denotes the distribution induced by the visit counts of MCTS, and $\mathcal{Q}_s$ denotes a vector of value estimates $\hat{Q}(s, a)$ for all actions $a \in \mathcal{A}(s)$. Note that the choice to store every encountered state, rather than only one state per full game of self-play, may lead to a poor estimate of the desired distribution over states due to high correlations, but is better in terms of sample efficiency. The maximum size of the experience buffer, which operates as a FIFO queue, is 400.

After every turn in self-play, we run a single mini-batch gradient descent (or ascent) update per vector of parameters that we aim to optimise (first updating any parameters for cross-entropy losses, and then any parameters for the TSPG objective). Gradients are averaged over mini-batches of up to 30 samples, sampled uniformly at random from the experience buffer. Updates are performed using a centered variant of RMSProp [32], with a base learning rate of 0.005, a momentum of 0.9, a discounting factor of 0.9, and a constant of 10^{-8} added to the denominator for stability. After every full game of self-play, we add a new feature to the set of features [29].

All self-play games are automatically terminated after 150 moves. In the play-out phase of MCTS, play-outs are terminated and declared a tie after 200 moves have been selected according to the play-out policy.

Some of the experiments involve evaluating the playing strength of different variants of MCTS after self-play training as described above. We use *Biased MCTS* to refer to a version of MCTS that is identical to the agents used to generate self-play experience as described above, except for that it selects actions to maximise visit count, rather than selecting actions proportional to visit counts, in evaluation games. We use *UCT* to refer to a standard implementation of MCTS [1], [3], using the UCB1 strategy [33] with an exploration constant of $\sqrt{2}$ in the selection phase of MCTS, and selecting actions uniformly at random in the play-out phase. We also allow UCT to reuse search trees from previous turns.

B. Results

In the first experiment, we compare the raw playing strength of standalone policies trained to either minimise the standard

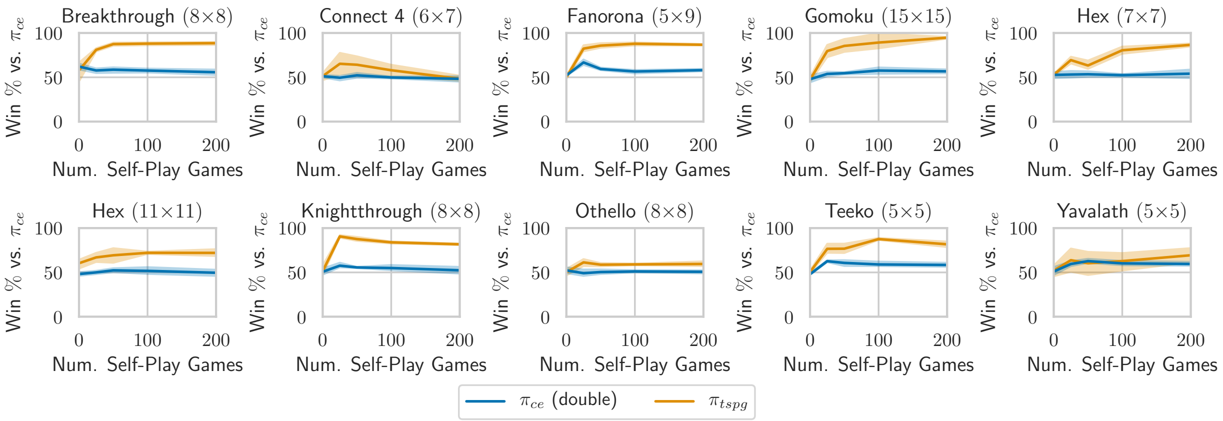

Fig. 2. Win percentages of π_{tspg} and π_{ce} (double) against π_{ce}, evaluated after 1, 25, 50, 100, and 200 games of self-play.

cross-entropy loss, or to maximise the TSPG objective. At various checkpoints during the self-play learning process (after 1, 25, 50, 100, and 200 games of self-play), we run evaluation games between softmax-based policies using the parameters learned at that checkpoint for either objective. We use π_{ce} to denote the policy trained on the cross-entropy loss. This is also the same policy that is used throughout self-play to bias the selection phase. We use π_{tspg} to denote the policy trained on the TSPG objective. Finally, we use π_{ce} (double) to denote a policy that – like π_{tspg} – uses the parameters of π_{ce} as a baseline (see Subsection IV-B), but – unlike π_{tspg} – again uses the cross-entropy loss to compute offsets from the baseline parameters.

Fig. 2 depicts learning curves, with the win percentages of π_{tspg} and π_{ce} (double) against π_{ce} measured at the different checkpoints. We repeat the complete training process from scratch five times with different random seeds, and play 200 evaluation games for each repetition. This leads to five different estimates of each win percentage, each of which is itself measured across 200 evaluation games. We use the sample bootstrap method to estimate 95% confidence intervals [34], [35] from these five estimates of win percentage per checkpoint, which are depicted as shaded areas.

It is clear from the figure that π_{tspg} consistently outperforms π_{ce}, in many games by a significant margin. We also observe that π_{ce} (double) occasionally outperforms π_{ce}, but generally by a smaller margin than π_{tspg}.

Table I shows win percentages in evaluation games of a Biased MCTS agent versus UCT. We compare two variants of the Biased MCTS; one where the cross-entropy-based π_{ce} (double) policy is used to run MCTS play-outs, and one where the TSPG-based π_{tspg} policy is used to run MCTS play-outs. In both cases, we use the final parameters learned after 200 games of self-play. Because our focus in this paper is on evaluating the quality of learned policies or strategies, we run these evaluation games with equal MCTS iteration count limits for all players. Note that this is not representative of playing strength under equal time constraints, since Biased MCTS generally takes more time to run than UCT. However, we do in most games find that Biased MCTS still outperforms UCT under equal time constraints (with most results being slightly improved since our previously-published results [29]).

TABLE I
WIN % OF BIASED MCTS VS. UCT (AFTER 200 GAMES OF SELF-PLAY).

	Win % (95% bootstrap conf. interval)	
Game (board size)	π_{ce} (double) play-outs	π_{tspg} play-outs
Breakthrough (8×8)	100.0 (100.0, 100.0)	100.0 (100.0, 100.0)
Connect 4 (6×7)	76.0 (72.0, 80.5)	72.0 (67.5, 77.3)
Fanorona (5×9)	99.5 (99.0, 100.0)	99.2 (98.5, 100.0)
Gomoku (15×15)	28.0 (22.5, 34.0)	18.0 (15.5, 20.5)
Hex (7×7)	89.5 (84.0, 95.0)	88.5 (84.0, 93.0)
Hex (11×11)	86.5 (78.5, 94.5)	71.0 (50.0, 95.0)
Knightthrough (8×8)	76.5 (73.5, 80.0)	63.0 (57.5, 69.5)
Othello (8×8)	69.0 (64.0, 73.5)	69.0 (66.0, 72.0)
Teeko (5×5)	97.0 (94.5, 100.0)	93.8 (91.8, 95.0)
Yavalath (5×5)	100.0 (100.0, 100.0)	98.5 (97.5, 99.5)

Similar to the evaluation in the previous subsection, we include all the different parameters learned from the five different repetitions of training runs in the evaluation. For each vector of parameters resulting from a different repetition, we run 40 evaluation games, for a total of 200 evaluation games across the five repetitions. The different estimates of win percentages from different repetitions are used to construct 95% bootstrap confidence intervals, which are shown in brackets in the table. In most games, we observe that both variants of Biased MCTS significantly outperform UCT, but play-outs from the cross-entrop-based π_{ce} (double) policy often appear to be slightly more informative to the MCTS agent than play-outs based on the TSPG objective.

Fig. 3 depicts how the entropy in distributions over actions as computed by a number of different policies varies throughout different stages of the different games. The entropy values are normalised to adjust for differences in the number of legal actions between different games and different stages of the same game. These entropy values were recorded in the evaluation games of Biased MCTS vs. UCT, for which win percentages are shown in Table I. In most stages of most games, we find that UCT has the highest entropy, followed

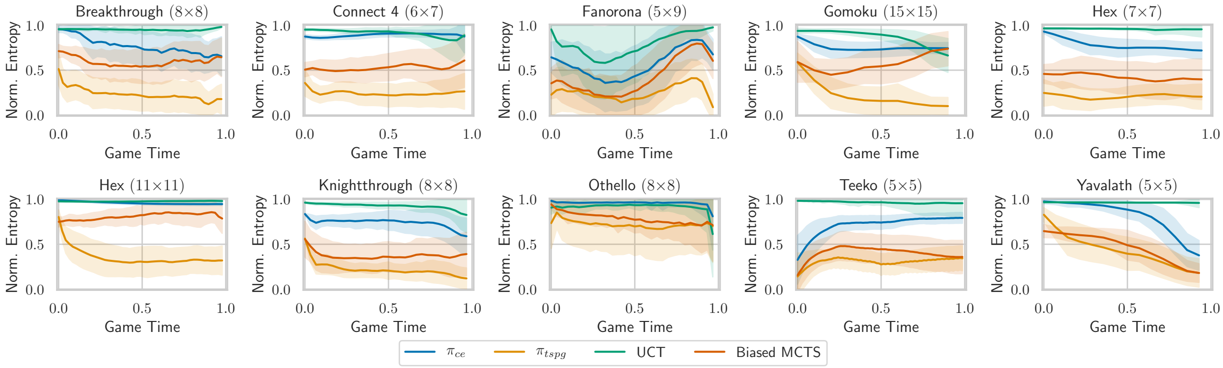

Fig. 3. Entropy in distributions over actions for different policies at different stages of a game. Entropy values on the y-axis are normalised to adjust for differences in number of legal actions. Game time (x-axis) corresponds to turn counter divided by total number of turns played in the corresponding match. For UCT and Biased MCTS, the distributions over actions are derived from the visit counts. Shaded regions depict standard deviation.

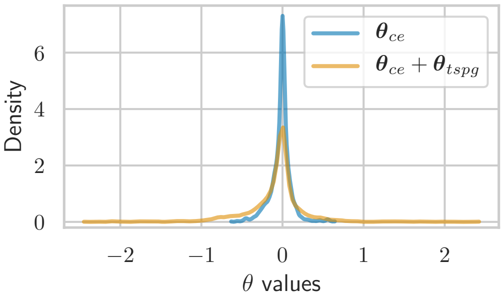

Fig. 4. Kernel density estimates for the distributions of θ values learned when optimising cross-entropy loss ($\boldsymbol{\theta}_{ce}$) or the TSPG objective ($\boldsymbol{\theta}_{ce} + \boldsymbol{\theta}_{tspg}$) in Othello.

(often closely) by π_{ce}, followed by Biased MCTS, finally followed by π_{tspg}.

Fig. 4 depicts kernel density estimates for the distributions of values in the learned parameter vectors after 200 games of self-play when optimising for the cross-entropy loss ($\boldsymbol{\theta}_{ce}$) or the TSPG objective ($\boldsymbol{\theta}_{ce} + \boldsymbol{\theta}_{tspg}$) in the game of Othello. We observe that the cross-entropy loss leads to a higher peak of parameter values close to 0, and a shorter range of more extreme parameter values far away from 0. In all other games (plots omitted to save space), we consistently observed similar differences between the two distributions.

VI. Discussion

The clear advantage in playing strength that π_{tspg} has over π_{ce} in Fig. 2 suggests that the TSPG objective is better suited for learning strong strategies, likely due to the lack of incentive to explore in the objective. The π_{ce} (double) policy slightly outperforms π_{ce} in some games, which suggests that some small gains in playing strength may simply be due to the increased number of gradient descent update steps that are taken by π_{ce} (double) in comparison to π_{ce}.

The results in Table I suggest that, despite the higher playing strength of π_{tspg}, π_{ce} (double) may be more informative when used as a play-out policy for MCTS agents. It has previously been observed [10], [12], [17] that policies optimised for “balance”, rather than standalone playing strength, may result in more informative evaluations from MCTS play-outs. Our results suggest that the cross-entropy loss may similarly lead to more balanced policies, leading to a decreased likelihood of biased evaluations.

The entropy plots in Fig. 3 show that the distributions over actions recommended by π_{tspg} tend to have the lowest entropy, which means that π_{tspg} more often approaches deterministic policies, by assigning the majority of the probability mass to only one or a few actions. We expect this to be beneficial for extraction of interpretable strategies from trained policies, because it means that there is more often a clear ranking of actions, and little ambiguity as for which action to pick in any given game state.

An interesting observation is that π_{ce} is explicitly optimised (through the cross-entropy loss) for having distributions close to those of Biased MCTS, but it still often has significantly higher entropy than Biased MCTS. In terms of entropy, the distributions resulting from π_{tspg} appear to be closer to those of Biased MCTS in many games, despite not being directly optimised for that target.

The results in Fig. 4 suggest that optimising for TSPG rather than cross-entropy loss may make it easier to obtain a clear ranking of features, due to differences between feature weights being more exaggerated, and fewer different features having highly similar weights. We again expect this to be beneficial for interpretation of learned strategies. A comparison to results published on learning balanced play-out policies in Go [12] supports the observation described above that the cross-entropy loss may lead to more “balanced” [10] policies.

VII. Conclusion

We proposed a novel objective function, referred to as the TSPG objective, for policies in Markov decision processes. Intuitively, a policy that maximises this objective function can be understood as one that selects actions such that, in expectation, an MCTS agent can perform well when playing out the remainder of the episode. We derive a policy gradient expression, which can be estimated using value estimates resulting from MCTS processes. Policies can be trained to optimise this objective using self-play, similar to cross-entropy-based policies in AlphaGo Zero and related research [4]–[6].

We argue that, due to the lack of a level of exploration in this objective's training target, it is more suitable for goals such as interpretable strategy extraction [21], [22].

Across a variety of different board games, we empirically demonstrate that the TSPG objective tends to lead to stronger standalone policies than the cross-entropy loss. Their distributions over actions tend to have significantly lower entropy, which may make it easier to extract clear, unambiguous advice or strategies from them. The TSPG objective also leads to a wider range of different values for feature weights, which can make it easier to separate features from each other based on their perceived importance.

In future work, we aim to extract interpretable strategies from learned policies, for instance by analysing the contribution [36] of individual features to the predictions made for specific game positions, or larger sets of positions. The feature representation [22] that we use is generally applicable across many different games, and allows for easy visualisation, which will be beneficial in this regard.

Acknowledgment.

This research is part of the European Research Council-funded Digital Ludeme Project (ERC Consolidator Grant #771292) run by Cameron Browne at Maastricht University's Department of Data Science and Knowledge Engineering.

References

[1] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Mach. Learn.: ECML 2006*, ser. LNCS, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Springer, Berlin, Heidelberg, 2006, vol. 4212, pp. 282–293.

[2] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games*, ser. LNCS, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer Berlin Heidelberg, 2007, pp. 72–83.

[3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–49, 2012.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[5] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *Adv. in Neural Inf. Process. Syst. 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5360–5370.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[7] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," INRIA, Paris, Tech. Rep. RR-6062, 2006.

[8] R. Coulom, "Computing "ELO ratings" of move patterns in the game of Go," *ICGA Journal*, vol. 30, no. 4, pp. 198–208, 2007.

[9] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 273–280.

[10] D. Silver and G. Tesauro, "Monte-Carlo simulation balancing," in *Proc. 26th Int. Conf. Mach. Learn.*, 2009, pp. 945–952.

[11] H. Baier and P. D. Drake, "The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 303–309, 2010.

[12] S.-C. Huang, R. Coulom, and S.-S. Lin, "Monte-Carlo simulation balancing in practice," in *Computers and Games. CG 2010.*, ser. LNCS, H. J. van den Herik, H. Iida, and A. Plaat, Eds., vol. 6515. Springer, Berlin, Heidelberg, 2011, pp. 81–92.

[13] M. H. M. Winands and Y. Björnsson, "$\alpha\beta$-based play-outs in Monte-Carlo tree search," in *Proc. 2011 IEEE Conf. Comput. Intell. Games*. IEEE, 2011, pp. 110–117.

[14] J. A. M. Nijssen and M. H. M. Winands, "Playout search for Monte-Carlo tree search in multi-player games," in *Adv. in Computer Games. ACG 2011.*, ser. LNCS, H. J. van den Herik and A. Plaat, Eds., vol. 7168. Springer, Berlin, Heidelberg, 2012.

[15] D. Silver, R. S. Sutton, and M. Müller, "Temporal-difference search in computer Go," *Mach. Learn.*, vol. 87, no. 2, pp. 183–219, 2012.

[16] T. Graf and M. Platzner, "Adaptive playouts for online learning of policies during Monte Carlo tree search," *Theoretical Comput. Sci.*, vol. 644, pp. 53–62, 2016.

[17] ——, "Monte-Carlo simulation balancing revisited," in *Proc. 2016 IEEE Conf. Comput. Intell. Games*. IEEE, 2016, pp. 186–192.

[18] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[19] T. Cazenave, "Playout policy adaptation with move features," *Theoretical Comput. Sci.*, vol. 644, pp. 43–52, 2016.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018.

[21] C. Browne, "Modern techniques for ancient games," in *Proc. 2018 IEEE Conf. Comput. Intell. Games*. IEEE, 2018, pp. 490–497.

[22] C. Browne, D. J. N. J. Soemers, and E. Piette, "Strategic features for general games," in *Proc. 2nd Workshop on Knowledge Extraction from Games (KEG)*, 2019, pp. 70–75.

[23] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Adv. in Neural Inf. Process. Syst. 12*, S. A. Solla, T. K. Leen, and K. Müller, Eds. MIT Press, 2000, pp. 1057–1063.

[24] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3-4, pp. 229–256, 1992.

[25] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *Int. Conf. Learning Representations (ICLR 2016)*, 2016.

[26] T. Vodopivec, S. Samothrakis, and B. Šter, "On Monte Carlo tree search and reinforcement learning," *J. Artificial Intell. Res.*, pp. 881–936, 2017.

[27] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," in *Proc. 29th Int. Conf. Mach. Learn.*, J. Langford and J. Pineau, Eds. Omnipress, 2012, pp. 457–464.

[28] C. Allen, K. Asadi, M. Roderick, A. Mohamed, G. Konidaris, and M. Littman, "Mean actor critic," 2018. [Online]. Available: https://arxiv.org/abs/1709.00503

[29] D. J. N. J. Soemers, É. Piette, and C. Browne, "Biasing MCTS with features for general games," in *2019 IEEE Congr. Evol. Computation*, 2019, in press. [Online]. Available: https://arxiv.org/abs/1903.08942v1

[30] C. Browne, "Automatic generation and evaluation of recombination games," Ph.D. dissertation, Queensland University of Technology, Brisbane, Australia, 2008.

[31] Z. Ahmed, N. L. Roux, M. Norouzi, and D. Schuurmans, "Understanding the impact of entropy on policy optimization," 2019. [Online]. Available: https://arxiv.org/abs/1811.11214v3

[32] A. Graves, "Generating sequences with recurrent neural networks," 2013. [Online]. Available: https://arxiv.org/abs/1308.0850v5

[33] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2–3, pp. 235–256, 2002.

[34] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC Press, 1994.

[35] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proc. 32nd AAAI Conf. Artificial Intell.* AAAI, 2018, pp. 3207–3214.

[36] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Adv. in Neural Inf. Process. Syst. 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774.