# THE MINIMUM VALUE STATE PROBLEM IN ACTOR-CRITIC NETWORKS

*Alvaro Velasquez*[1] *Ismail R. Alkhouri*[2] *Brett Bissey*[3] *Lior Barak*[2] *George K. Atia*[2,4]

[1] Information Directorate, Air Force Research Laboratory, Rome NY, USA
[2] Department of Electrical and Computer Engineering, University of Central Florida, Orlando FL, USA
[3] MITRE Corporation, McLean VA, USA
[4] Department of Computer Science, University of Central Florida, Orlando FL, USA

## ABSTRACT

Deep reinforcement learning (RL) methods are vulnerable to adversarially perturbed states. Though the nature of such states is difficult to characterize, popular actor-critic architectures provide a natural way to detect such pathological states by determining whether their value head is below some value. In this paper, we leverage this capacity of actor-critic architectures to generate low-value states that can be used to define a training regimen for RL algorithms, which we call the Pathological Ensemble of Actions for RL (PEARL). We pose the problem of synthesizing a low-value state for a given architecture, prove that it is **NP-hard**, and present a solution based on integer linear programming. The generated states can then be used to augment the training data with these pathological states. We demonstrate the gains obtained by PEARL when compared to standard Proximal Policy Optimization and Monte-Carlo Tree Search baselines on board games such as Go and Checkers.

***Index Terms***— Deep Reinforcement Learning, Actor-Critic, Minimum Value State, Adversarial Training

## 1. INTRODUCTION

Deep reinforcement learning has witnessed tremendous success in recent years, beginning with the development of deep Q-networks that can achieve superhuman performance in various Atari benchmarks [1] and more recently with the adoption of actor-critic architectures within planning solutions like Monte-Carlo Tree Search (MCTS) to defeat the world Go champion at the time [2]. In particular, these actor-critic architectures enabled the value prediction of an observed state to be computed within the policy network. While this facilitated their adoption within novel MCTS algorithms [2, 3, 4, 5], we argue that these value predictors have other wide-reaching implications with respect to the generation of adversarial states for measuring policy robustness and for the integration of these states within adversarial training regimes for reinforcement learning models. To that end, this paper explores the synthesis of adversarial low-value states for a given actor-critic policy network via integer linear programming. We call such states pathological in that they induce the worst behavior of the agent in terms of the expected reward output by the value head of $v_\theta$ of the policy network parameterized by $\theta$. We call this the Minimum Value State (MVS) problem and prove it belongs to the **NP-hard** complexity class. Although this means that a polynomial-time solution is not likely to exist, we present an integer linear program (ILP) that can find effective solutions to the MVS problem for networks with over 100,000 parameters in under one minute. The synthesized states are then used to train a second actor-critic policy network to improve the performance of the agent when observing low-value states. These two networks form a Pathological Ensemble of Actions for Reinforcement Learning (PEARL). We demonstrate that PEARL can yield significant gains in observed rewards in Go and Checkers board games environments. Examples of generated pathological states for a given actor-critic architecture using our approach can be seen in Figure 1. We focus our attention on discrete-valued precisely-defined state spaces such as those encountered in board games. In such settings, conventional gradient-based approaches to adversarial state generation may not generate a valid state. Indeed, these state spaces are defined by specific rules that are easily encoded by integer linear programs, but may be violated by traditional adversarial approaches. For example, a generated checkers board state can have at most 12 black or white pieces.

The contributions of this paper are as follows. First, we present the MVS problem. We include how different layers are representable as linear functions for the adoption of linear programming. Second, we present the complexity results of the MVS problem. Third, experiments are conducted on Proximal Policy Optimization (PPO) and MCTS solutions to the games of Go and Checkers with and without PEARL to illustrate the gains obtained from utilizing our method.

### 1.1. Related Work

The idea of adversarial training has been explored in the literature as means of robustifying neural network classifiers [6] [7] [8]. These methods typically utilize the Projected Gra-

This work was supported in part by NSF CAREER Award CCF-1552497 and NSF Award CCF-2106339, DOE Award DE-EE0009152, AFRL Contract Number FA8750-20-3-1004, and AFOSR Award 20RICOR012.

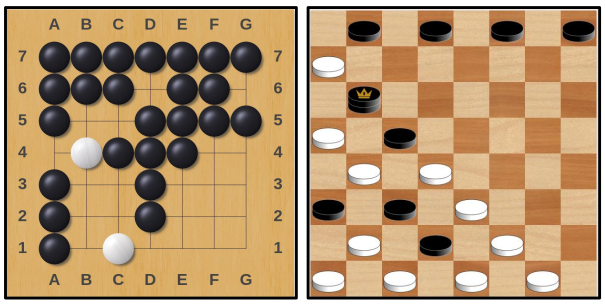


**Fig. 1**: Examples of the board games of Go (*left*) and Checkers (*right*), respectively, generated from our proposed method. In the shown Black-to-play low-value checkers-state generated with respect to Black, Black is at a clear disadvantage due to being behind in the number of pieces, as well as that White has stabilized its pieces along the main diagonal and the walls.

dient Descent (PGD) attack to generate adversarial samples to improve the training performance of the underlying neural network. Adversarial attacks on deep reinforcement learning have also leveraged similar gradient-based attacks. For example, the work in [9] presents three varieties of gradient-based adversarial attacks, such as adding perturbations to the observation space with the goal of reducing the reward signals [10]. At the intersection of these areas lies adversarial training for reinforcement learning. However, in this setting, a significant challenge arises in how to generate a valid state. Indeed, the games of precision often used as benchmarks for reinforcement learning follow very specific rules. For example, one cannot have more than 12 black pieces in a Checkers board state and no more than two knights for either player in a valid Chess board state. Indeed, these hard constraints pose a challenge to the use of popular gradient-based attack methods for adversarial training. This is a challenge we resolve via the use of integer linear programming, which provides a natural platform for defining such constraints.

It is worth noting that the verification community has also exploited the use of integer linear programming techniques to great effect in order to ensure that, for a given bounded volume of data, some property holds within feedforward neural networks. The approach proposed in [11] uses an ILP formulation to encode constraints reflecting neural network layers to derive a point-wise robustness measure. Similarly, the work in [12] studies reachability problems for feed-forward neural networks with rectified linear unit (ReLU) activation functions and dense layers by encoding these in an ILP formulation. In [13], the number of non-linearities to be handled by the ILP for verification can be significantly reduced since there is a bounded input domain for which it is observed that many of the ReLUs are always active or inactive. This, combined with other techniques based on the bounded input domain, can greatly reduce the number of ReLUs to be considered by the ILP solution. Our paper differs from these efforts in two key aspects. First is the implementation on reinforcement learning systems. Specifically, our goal is to synthesize inputs that minimize the value head representing the predicted value of a state in a given actor-critic architecture. Second is the utilization of game-related integer constraints that depend on the type of the environment.

## 2. THE COMPLEXITY OF GENERATING STATES OF MINIMUM VALUE

In order to establish the complexity of generating a state $s$ of lowest value $v_\theta(s)$ for a given actor-critic architecture, we will leverage a popular problem known as Maximum Independent Set (MIS).

**Definition 2.1** (Maximum Independent Set)**.** Given an undirected graph $G = (V, E)$, the Maximum Independent Set (MIS) problem consists of finding a subset of vertices $V' \subseteq V$ such that no two vertices in $V'$ are connected by an edge and $|V'|$ is maximized.

**Definition 2.2** (Minimum Value State (MVS))**.** Given an actor-critic architecture $f_\theta$ parameterized by $\theta$ with ReLU activation functions and value output $v_\theta : S \to \mathbb{R}$, find a state $s$ such that $v_\theta(s)$ is minimized.

**Theorem 2.3.** *MVS is* ***NP-hard****.*

*Proof.* We prove that MVS is **NP-hard** by reducing an arbitrary MIS instance to a corresponding MVS instance and demonstrating that, if an efficient solution to the latter exists, then it can be used to derive an efficient solution to the MIS instance. Given an MIS instance $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we construct an actor-critic architecture as follows. For each $v_i \in V$, we have an entry $s_i \in [0, 1]$ in the input state vector. We assume these normalized bounds of $[0, 1]$ on the input for convenience and we do not restrict the proof to only discrete values. Each of these $s_i$ is connected to a ReLU activation function $r_i$ with bias term $-1/2$ via the weight parameter $\theta_{ii} = 1$. We call these vertex ReLUs. The outputs of $r_1, \ldots, r_n$ are then connected to $v_\theta$ with weights $\theta_{iv} = -1$. For each edge $e_k = (v_i, v_j) \in E$, we have two connections from inputs $s_i$ and $s_j$ to ReLU $r_{n+k}$ in the hidden layer. We call these edge ReLUs. In particular, we have weights $\theta_{i,n+k} = 1$ and $\theta_{j,n+k} = 1$. Each of these ReLUs $r_{n+1}, \ldots, r_{n+m}$ has a bias term of $-1$. The outputs of these ReLUs are connected to $v_\theta$ with weights $\theta_{n+k,v} = n$. See Figure 3 for a visual example of this reduction. We now show that a solution $V' \subseteq V$ to the given MIS instance is of maximum cardinality if and only if the solution $s$ to the reduced MVS instance instance has minimum value $v_\theta(s)$.

($\implies$) Assume that $V'$ is an independent set of maximum cardinality. Let $s_i = 1$ for every $v_i \in V'$ and $s_i = 0$ otherwise. Since the vertices in $V'$ do not share edges, the inputs into edge ReLUs $r_{n+1}, \ldots, r_{n+m}$ will be of value at most 1. Given the bias term of -1 for these ReLUs, it follows that their outputs will be 0. For the vertex ReLUs $r_1, \ldots, r_n$, we have $|V'|$ inputs of value 1. This results in a combined output of $|V'|/2$ due to the bias terms of these ReLUs and a corresponding input of $-|V'|/2$ into $v_\theta$. This is the minimum input value for $v_\theta$. Assuming that $v_\theta$ is a monotonically non-decreasing function, this yields the minimum value $v_\theta(s)$.

( $\Longleftarrow$ ) Suppose $s \in [0,1]^n$ is a state of minimum value $v_\theta(s)$. It must be the case that, for any given edge $e_k = (v_i, v_j) \in E$, we have $s_i + s_j \leq 1$. In this case, it must be that either $s_i = 1$ or $s_j = 1$ as this would yield a combined ReLU output of $1/2$, which minimizes the negative input into $v_\theta$ through the vertex ReLUs $r_i$ and $r_j$ with bias terms of $-1/2$. However, for the sake of contradiction, consider the case where $s_i + s_j > 1$. Then, there would be an output of $((s_i+s_j)-1) > 0$ from the edge ReLU $r_{n+k}$. This would result in an input of $n((s_i+s_j)-1)$ into $v_\theta$. It can be shown that the combined output of edge ReLU $r_{n+k}$ and vertex ReLUs $r_i$ and $r_j$ will be strictly greater than $-1/2$, yielding a contradiction since this would imply that a state of lower value than $s$ exists. The combined input of these three ReLUs into $v_\theta$ is $n((s_i + s_j) - 1) - (\max(0, s_i - 1/2) + \max(0, s_j - 1/2))$. This can be seen in the network of Figure 2.

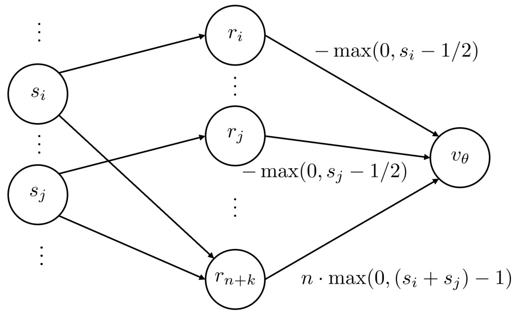


**Fig. 2**: Network used in the proof.

Let $s_i + s_j = 1 + \delta$ for $\delta \in (0, 1]$. The combined input into $v_\theta$ resulting from these three ReLU outputs then reduces to $n\delta - (\max(0, s_i - 1/2) + \max(0, s_j - 1/2))$. Note that this value is minimized as $\delta$ approaches 0. We have

$$\lim_{\delta \to 0} \left( n\delta - (\max(0, s_i - 1/2) + \max(0, s_j - 1/2) \right) \geq$$
$$n\delta - 1/2 > -1/2$$

We have thus shown that $s_i + s_j = 1$ for every edge $e_k = (v_i, v_j) \in E$, with either $s_i = 1$ or $s_j = 1$. It follows that the minimum value of $v_\theta(s)$ is obtained when the maximum number of entries in $s$ have value 1 such that their corresponding vertices in $G$ share no edges. This yields an independent set $V' = \{v_i \in V | s_i = 1\}$ of maximum cardinality. □

We have shown that generating states of minimum value is **NP-hard** and, therefore, a polynomial-time solution is not likely to exist. Nevertheless, we present an integer linear programming solution that is efficient in practice.

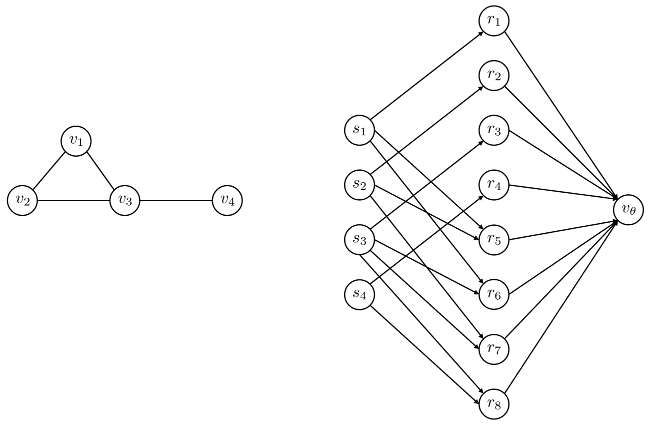


**Fig. 3**: Given the MIS instance as an undirected graph $G = (V, E)$ (*left*), and its corresponding MVS instance (*right*).

## 3. METHODOLOGY

Modern CNN architectures such as Inception, AlexNet, SqueezeNet, ResNet, and VGG, among others, consist of an input image, followed by convolutional layers, followed by batch normalization, followed by activation functions (typically ReLU activations) and average of max pooling layers in a cyclical fashion. At the end of the CNN are one or more fully connected layers which take a flattened vector representation of the tensor output by the last pooling layer and connect each of the values in this flattened vector to a softmax activation function for each of the output classes in the learning problem. In the case of actor-critic architectures, these classes correspond to the actions an agent can take and there is also a value output $v_\theta(s)$ denoting the predicted value of playing a game from a given state $s$. We propose an integer linear programming approach to generate input states for which a given actor-critic architecture yields a low-value output. We assume that this actor-critic architecture is a CNN and proceed to demonstrate how the various facets of a CNN can be encoded as linear constraints. That is, rather than optimizing $v_\theta(s)$ directly with respect to the input state $s$, we instead optimize indirectly by encoding all the intermediate layers as linear constraints.

**Handling Convolutions:** Let $s = (x^0_{ijk})$ denote the state representing the input data tensor to the CNN and let $x^l_{ijk} \in [0, 1]$ denote the normalized pixel value of the pixel in the $i^{\text{th}}$ row and $j^{\text{th}}$ column of the $k^{\text{th}}$ channel of the data volume in the $l^{\text{th}}$ layer, where $l \geq 1$. For the input, we assume integer values $x^0_{ijk} \in \mathbb{Z}$ representing, say, pixel values or indicator variables that are often used in reinforcement learning benchmarks. Assume that the $l^{\text{th}}$ convolutional layer consists of $n_f$ filters $(c^{lf}_{ijk})$ with height $h^{lf}$, width $w^{lf}$, and depth $d^{lf}$, each of which will be applied to the data volume output by the previous layer $l - 1$. The result of convolution in the $l^{\text{th}}$ layer is given by $\sum_{i'j'k'} x^{l-1}_{i'j'k'} \cdot c^{lk}_{i'j'k'} + b^{lk}$ , where $b^{lk}$ is a bias term. We have not included stride terms nor the specific indexing schemes for $i', j', k'$ for ease of presentation.

**Handling Batch Normalization:** After each convolutional layer, there is often a batch normalization layer [14] which scales and shifts its inputs $(x_{ijk})$ using the learned parameters $\gamma$ and $\beta$ as

$$\gamma_{ijk}\left(\sum_{i'j'k'} x^{l-1}_{i'j'k'} \cdot c^{lk}_{i'j'k'} + b^{lk}\right) + \beta_{ijk}\,. \tag{1}$$

The results of convolutions are often normalized to mitigate the problem of covariate shift which arises in deep networks during training.

**Handling ReLU Activations:** After batch normalization, non-linearities are applied. This is typically accomplished with ReLU activations given by the formula $r^l_{ijk} = \max\{0, x^l_{ijk}\}$. For an input volume $(x^l_{ijk})$, note that the solution to the following linear program (2) yields $r^l_{ijk} = \max\{0, x^l_{ijk}\}$ for all height, width, depth, and layer indices $i, j, k, l$, where Equation (1) is used as input.

$$\begin{aligned}
&\min \sum_{i,j,k,l} r^l_{ijk} \text{ subject to} \\
&r^l_{ijk} \geq \gamma_{ijk}\left(\sum_{i'j'k'} x^{l-1}_{i'j'k'} \cdot c^{lk}_{i'j'k'} + b^{lk}\right) + \beta_{ijk} \quad \forall i,j,k,l \\
&r^l_{ijk} \geq 0 \quad \forall i,j,k,l
\end{aligned} \tag{2}$$

It is worth noting that each ReLU function can also be encoded using four integer linear constraints and a binary variable [13]. However, this introduction of integer linear constraints and binary variables makes solving such formulations much more complex.

**Handling Pooling:** In order to downsample image volumes to a lower-dimensional space while retaining the most meaningful features, the use of max or average pooling operations is commonly adopted. Unlike convolutional filters which are applied to a 3D image volume, pooling filters are applied to a 2D image. In particular, an $h \times w$ max (average) pooling filter simply chooses the maximum (average) value in an $h \times w$ sub-image. Given an $h \times w$ input $(x^l_{ijk})$, the max pooling filter value is given by $x'_{ijk} = \max(x^l_{i'j'k'})$. It has been shown that each max pooling function over $n = h \times w$ variables can be encoded using $n + 1$ integer linear constraints, $n$ linear constraints, and $n$ binary variables [13]. Due to the complexity incurred by these max pooling operations, we focus our attention on average pooling layers, which can represented by the simple linear transformation in $x^l_{ijk} = \frac{1}{hw}\sum_{i',j'} r^l_{i'j'k}$. For the remainder of this paper, we adopt the use of average pooling layers for simplicity and to mitigate the complexity of the synthesis procedure.

**Handling Fully-Connected Layers:** After the last pooling layer, the resulting image volume is flattened into a vector and one or more fully-connected layers (often with ReLU activations) follow before reaching the final layer which yields the policy and value $v_\theta(s)$ for a given state $s$. Let $L$ and $L_C$ denote the number of layers and convolutional layers, respectively, in the network of interest. Given a vector $(x^{l-1}_i)$ as input from layer $l-1$, a neuron $x^l_j, l > L_C$ in a fully-connected layer with incoming weights $(w^{l-1}_{ij})$ will output Equation (3).

$$x^l_j = \sum_i w^{l-1}_{ij} x^{l-1}_i + b^l_j \tag{3}$$

After the fully-connected layers comes the final layer, which typically consists of fully-connected softmax activation functions for the policy and value heads. However, we focus our attention solely on the inputs to the value head since this is the value we would like to minimize in generating low-value inputs for reinforcement learning. Since this corresponds to the outputs of the final fully-connected layer, we denote these input values within the linear program as $\sum_i x^L_i$.

We can integrate the foregoing formulations into the ILP in (4). Note that the inputs $\sum_i x^L_i$ to the value head $v_\theta$ are a part of the objective function.

$$\begin{aligned}
&\min \sum_{i,j,k,l} r^l_{ijk} + \sum_i x^L_i \text{ subject to} \\
&(i)\ r^l_{ijk} \geq \gamma^l_{ijk}\left(\sum_{i',j',k'} x^{l-1}_{i'j'k'} \cdot c^{lk}_{i'j'k'} + b^{lk}\right) + \beta^l_{ijk} \\
&\qquad\qquad \forall i,j,k,l \leq L_C \\
&(ii)\ r^l_{ijk} \geq 0 \qquad \forall i,j,k,l \leq L_C \\
&(iii)\ x^l_{ijk} = \frac{1}{hw}\sum_{i',j'} r^l_{i'j'k} \qquad \forall i,j,k,l \leq L_C \\
&(iv)\ r^l_j \geq \gamma^l_j\left(\sum_i w^{l-1}_{ij} x^{l-1}_i + b^l_j\right) + \beta^l_j \qquad \forall j, l > L_C \\
&(v)\ x^l_j = r^l_j \qquad \forall j, l > L_C \\
&\quad x^0_{ijk} \in \mathbb{Z}, x^l_{ijk} \in \mathbb{R}, r^l_{ijk} \geq 0 \qquad \forall i,j,k,l
\end{aligned} \tag{4}$$

In this program, Constraint (i) encodes the convolutional filters, batch normalization, and part of the ReLU formulation. Constraint (ii) completes the encoding of ReLU outputs. Constraint (iii) encodes the average pooling operations. Constraint (iv) encodes the outputs of fully-connected layers after the ReLU operations in these layers. Constraints (v) explicitly states that the output of a ReLU activation in the fully connected layers is itself the output of that layer. This differs from the convolutional layers, where pooling operations are performed after ReLU activations in order to determine the outputs of that layer.

It is worth noting that the program (4) is an ILP since we are restricting the input state $s = (x^0_{ijk})$ to be composed of integers. Therefore, the solution to this program may require an exponential number of operations in the worst case. The justification for adopting an ILP formulation follows from the fact that we have proved MVS to be **NP-hard** and, therefore, no polynomial-time solution is likely to exist. In order to increase efficiency, we have modeled ReLU operations using continuous variables and added these to the objective function along with the original objective of minimizing the inputs into $v_\theta$. In cases where the two objectives in the objective function of the ILP may compete, there is no guarantee that a generated state will be of minimum value. We adopt this formulation as a point of practicality since an exact formulation would require the addition of a binary variable for each ReLU activation function [13] in order to remove the ReLUs from the objective function. This would make the problem intractable for all but the smallest of networks. Indeed, for our experiments, we define a network of modest size that includes over 100,000 ReLU activations. An exact ILP solution to the MVS problem would require the addition of over 100,000 binary variables. In the worst case, this leads to an exponential runtime proportional to $2^n$, where $n$ denotes the number of binary variables [15], [16]. We therefore make this trade-off between precision and tractability and demonstrate that our proposed ILP can yield low-value states in under one minute. These states are used to train a second actor-critic architecture for a given agent, enabling the agent to reason over a PEARL by arbitrating between using its conventional actor-critic architecture or its secondary pathologically trained architecture.

## 4. EXPERIMENTAL RESULTS

We extend the ILP in (4) with domain-specific constraints in order to generate a valid input state under which an actor-critic network experiences a low-value output. The input tensor is $(x^0_{ijk}) \in \{0,1\}^{I \times J \times K}$, where $I$ and $J$ represent the board dimensions, and $K$ gives the number of channels/slices that represent the pieces location and/or player turns. For Go and Checkers, we use two common sets of constrains for $(x^0_{ijk})$. The purpose of the first constraint is to limit the number of each piece in the board. For an instance in the game of Go, let the black team piece be represented by channel $k = 1$, and maximum acceptable number at a given board setup be $N_B$, then the constraint $\sum_{i,j} x^0_{i,j,1} \leq N_B$ is added. The objective of the second set of constraints is to prevent the case of having two pieces at the same board location. As the instance of Checkers, let $k$ equals to $1, 2, 3, 4$ represent player 1 men, player 2 men, player 1 king, and player 2 king, respectively, then the constraints $\sum_k x^0_{i,j,k} \leq 1, \forall i \in [I], j \in [J]$ are included in the ILP.

Two sets of additional constraints are used in Checkers. The purpose of the first is to enforce that all white entries of all the pieces location slices are enforced to zero. The second is for the rule for which pawns can not be placed at the last row of the opposite team. This is not an exhaustive list and other game-specific constraints are added. Our code along with the NN structures are available online[1].

In order to evaluate the efficacy of using ILP-generated states throughout training, we will compare Actor-Critic (A2C) models learning zero-sum board game baselines such as $7 \times 7$ Go and Checkers. Specifically, we will train some models using a PPO loss function and others using an MCTS loss function and DeepMCTS architecture akin to AlphaZero [17]. The environments are forked from the PettingZoo API [18], with functions added to load board-states from the ILP-generated state tensors and start games from these board-states. We create unique ILP state generation modules for each game based on each game's unique integer constraints. We then start playing from the generated low-value input states as part of the training process in order to learn more effective policies.

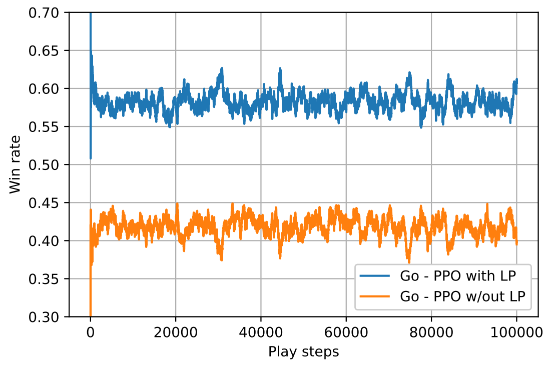
For each learning architecture, we train one learner which starts games from an ILP-generated state every three games, and another traditional learner which starts all games from the standard initial board-state, i.e., an empty Go board or initial Checkers configuration. The post-training evaluation stage evaluates a PEARL PPO agent against a conventional PPO agent, and a PEARL Deep MCTS agent against a conventional Deep MCTS agent.

Each learning architecture is trained using self-play for one million play-steps. The Deep MCTS architecture simulates tree-steps between play-steps, but these tree-steps are not counted towards the one million play-steps. At the end of each self-play training game, the winning (losing) self-play team is awarded a unit reward of 1 (-1), which is paired with the winning (losing) state and discounted back through the prior states in the winning (losing) trace. Each game yields symmetric, zero-sum winning and losing traces, both of which are stored in a replay buffer which holds the most recent 100,000 states. Following each game, we add the two traces to the replay buffer, and then the CNN is trained on 10 randomly selected traces from this replay buffer using the given architecture's loss function (either PPO or MCTS).

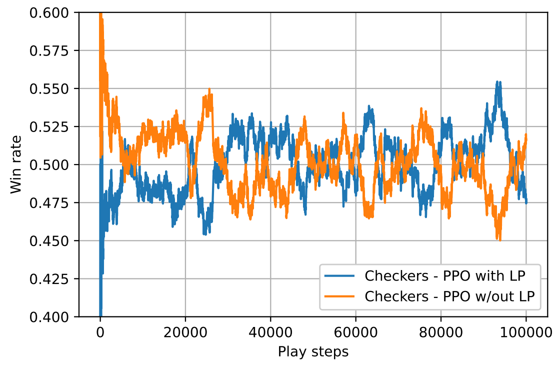
For the PEARL training architectures, each game's unique ILP generation module periodically fills a queue with 5 ILP-generated low-value states at a time. The PEARL architectures pop states from this queue to use as their initial board-state every 3 games during training, using standard initial board-states every other training game. Once an ILP-generated state has been used, it is removed from the queue. When the queue is empty, the ILP generates a new pool of 5 low-value states and adds it to the queue. Each time a new pool of states is generated, the ILP generation is run with updated CNN weights. By generating a pool of only 5 states at a time throughout training, we ensure the ILP-generated low-value states are dynamically reflective of the CNN model at that point in training.

We choose to use LP states every 3 games rather than every game so that the model is still able to train on board-

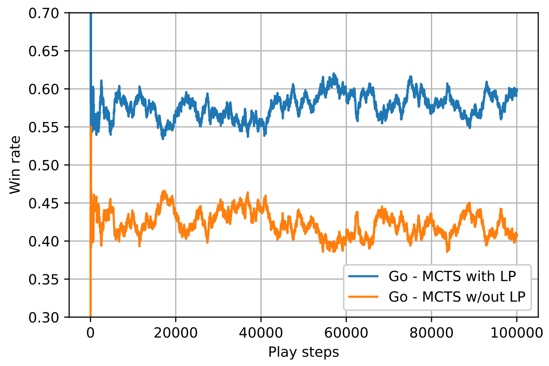
[1]https://anonymous.4open.science/r/TheMVSinA2C-C12B/



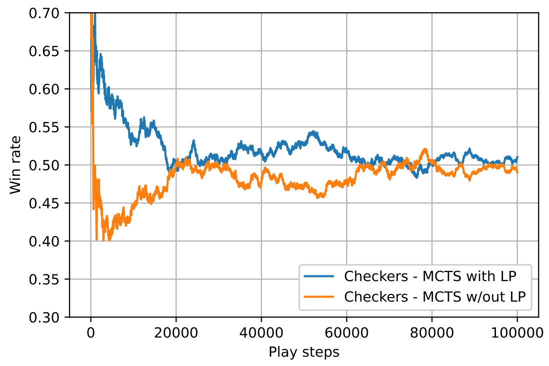
(a) Agents trained using PPO for the game of Go.



(b) Agents trained using PPO for the game of Checkers.



(c) Agents trained using MCTS for the game of Go.



(d) Agents trained using MCTS for the game of Checkers.

**Fig. 4**: Evaluation curves of the A2C agents for the games of Go (*left*) and Checkers (*right*).

states within traces starting from the standard initial state, given that evaluation runs will start from this standard initial state. Hyper-parameters such as MCTS expansion leaf-count (Go: 60, Checkers: 40) and learning rate (Go/Checkers-MCTS: .003, Checkers-PPO: .03) are chosen manually based on perceived game difficulty and observed loss curves. We hold hyper-parameters constant when comparing PEARL agents to their conventionally trained counterparts. We evaluate trained checkpoints through win-rate comparison when running 100,000 play steps worth of games with traditional architectures (PPO, MCTS) versus the same architecture supported by ILP-generated states during training (MCTS vs. MCTS-LP, PPO vs. PPO-LP). The architectures using ILP-generated input states train a second LP-net to specifically learn from low-value states. During evaluation, agents use the network policy of whichever network, LP-trained or conventionally-trained, returns the highest value head for the given evaluation state. This varies from training, where we only use the ILP-trained network for games starting from a low-value, ILP-generated state. The 100,000 evaluation steps all consist of games starting from the initial board configuration, and agents alternate taking the first turn.

Figure 4 presents the evaluation of the games of Go (Figures 4a and 4c) and Checkers (Figures 4b and 4d). The PEARL agent used ILP-generated low-value board configurations every 3 episodes during self-play training. We observe that the PEARL agent wins 14.9% (15.9%) more games during evaluation than the conventionally trained PPO (MCTS) counterparts when averaged over 16 (9) evaluation rounds. The area under curve (AUC) metric for Checkers is also significantly greater for the PEARL agent using MCTS.

## 5. CONCLUSION

In this paper, we have presented an integer linear programming formulation for the synthesis of low-value states given an actor-critic policy network. We furthermore demonstrated that the use of these pathological states within the training procedure of reinforcement learning agents can lead to significant gains in performance, though the settings under which this is possible are not well-understood and provide an interesting direction for future work in adversarial training of reinforcement learning architecture. In particular, we leverage an ensemble of two actor-critic architectures, with one trained conventionally and the other exposed to the generated pathological states. The agent can then decide which policy to follow based on the value outputs of this ensemble. This Pathological Ensemble of Actions for Reinforcement Learning (PEARL) can be integrated within off-the-shelf reinforcement learning solutions.

## 6. REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] 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*, vol. 529, no. 7587, pp. 484, 2016.

[3] 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*, vol. 550, no. 7676, pp. 354, 2017.

[4] 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*, 2017.

[5] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al., "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[6] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[7] F Tramèr, D Boneh, A Kurakin, I Goodfellow, N Papernot, and P McDaniel, "Ensemble adversarial training: Attacks and defenses," in *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, 2018.

[8] Uri Shaham, Yutaro Yamada, and Sahand Negahban, "Understanding adversarial training: Increasing local stability of supervised models through robust optimization," *Neurocomputing*, vol. 307, pp. 195–204, 2018.

[9] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary, "Robust deep reinforcement learning with adversarial attacks," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, 2018, pp. 2040–2042.

[10] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[11] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi, "Measuring neural net robustness with constraints," in *Advances in neural information processing systems*, 2016, pp. 2613–2621.

[12] Alessio Lomuscio and Lalit Maganti, "An approach to reachability analysis for feed-forward relu neural networks," *arXiv preprint arXiv:1706.07351*, 2017.

[13] Vincent Tjeng, Kai Y Xiao, and Russ Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *International Conference on Learning Representations*, 2018.

[14] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[15] Schrage Linus, "Optimization modeling with lingo," *LINDO Systems Inc*, 2015.

[16] Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al., *Integer programming*, vol. 271, Springer, 2014.

[17] 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*, vol. 529, no. 7587, pp. 484–489, 2016.

[18] Justin K Terry, Benjamin Black, Mario Jayakumar, Ananth Hari, Luis Santos, Clemens Dieffendahl, Niall L Williams, Yashas Lokesh, Ryan Sullivan, Caroline Horsch, and Praveen Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning," *arXiv preprint arXiv:2009.14471*, 2020.