NN2EQCDT: Equivalent Transformation of Feed-Forward Neural Networks as DRL Policies into Compressed Decision Trees

Torben Logemann
Carl von Ossietzky University Oldenburg
Research Group Adversarial Resilience Learning
Oldenburg, Germany
Email: torben.logemann@uol.de

Eric MSP Veith
Carl von Ossietzky University Oldenburg
Research Group Adversarial Resilience Learning
Oldenburg, Germany
Email: eric.veith@uol.de

***Abstract*—Learning systems have achieved remarkable success. Agents trained using Deep Reinforcement Learning (RL) (DRL) methods, e.g., promise real resilience. However, no guarantees can yet be provided for the learned black-box models. For operators of Critical National Infrastructures (CNIs), this is a necessity as no responsibility can be assumed for an unknown and unvalidatable control system. Intrinsically secure learning algorithms and approximate, post-hoc interpretable models exist, but they lack either learning performance or explainability. To optimize this trade-off, this paper presents the NN2EQCDT algorithm, which equivalently transforms a Feed-Forward Deep Neural Network (DNN) (FF-DNN)-based policy into a compressed Decision Tree (DT). The compression is achieved by dynamically checking the satisfiability of the paths during construction, removing checks that are not needed further, and considering invariants. For a small policy model, NN2EQCDT was observed to drastically compress the DT, making it possible to accurately trace action regions to their observation regions in a plotted DT and visualization.**

***Keywords*—*reinforcement learning*; *explainable AI*; *equivalent transformation*; *neural network*; *decision tree*; *compression*.**

I. Introduction

Learning systems have achieved remarkable successes. DRL is at the core of many remarkable successes, beginning with its breakthrough in 2013 by end-to-end learning of Atari games [1] and Double Q-learning [2]. Further successes were made with AlphaGo (Zero), AlphaZero [3] and MuZero [4]. DRL involves learning agents with sensors and actuators to achieve specific goals through trial and error, using algorithms, such as Twin-Delayed Deep Deterministic Policy Gradient (DDPG) (TD3) [5], Proximal Policy Gradient (PPO) [6], and Soft Actor Critic (SAC) [7] have proven that they are capable of handling complex tasks. Due to its success, learning system are applied in various fields, such as the following.

- In healthcare, RL is preferred over traditional DNN methods to determine the best treatment policy [8].
- In robotics, RL agents can learn tasks, such as pouring water, reaching through a human teacher, grasping, balancing balls, and more [8].
- DRL is used in autonomous driving because of its strong interaction with the environment [9].
- In cybersecurity, DRL is used for automatic intrusion detection techniques and defense strategies [10].
- To be able to keep the power grid stable, DRL is used to train defender agents with the Adversarial Resilience Learning (ARL) framework to recover deviations from the healthy state by deploying attacker agents in an autocurriculum setting [11].

DRL agents promise true resilience by learning to counter the unknown unknowns. However, unlike intrinsically interpretable DRL models [12], no guarantees can yet be made about the behavior of DRL agents learned with black-box models. This is, however, a necessity for operators, since no responsibility can be taken for an unknown control system that cannot be validated, especially when it is used in such critical or very critical areas as CNIs.

If agents with learned black-box models are to be deployed in CNI, it is of absolute necessity to be able to provide guarantees for them, as they have the potential to significantly threaten the safety the overall system. Without guarantees, operators cannot take responsibility for such an unknown and unverifiable control system. An architecture, designed to provide such guarantees, is presented in [13], which is suitable for usage in CNIs, such as the power grid.

Agents deployed in complex environments, such as complex interconnected systems, potentially face many different situations and learn complex behaviors to cope with them according to their goals. To understand how agents achieve their goals, the effects of their strategies are studied in terms of rewards or impact on the environment. One such example is in [11], which ARL attack agents are deployed with the goal of causing voltage band violations in a power grid. They achieve this goal by exploiting a vulnerability in the deployment of voltage controllers in the used network. How the actual exploit works, is analyzed by examining the impact of the attacker actions on the victim buses. This is sufficient for commonly observed behaviors, but it is not deeply interpreted and there is no guarantee that the extracted strategy explained by the investigations is used for all situations, i.e., for all observations from the environment. This is especially important when dealing with control agents, who are expected to achieve a goal in all possible situations, e.g., defender agents, for whom the even greater problem of coping with an infinite horizon exists in the explanation.

Thus, the need arises to provide transparency to the learned strategies of agents, i.e., to approximate their behavioral model as well as possible by a more comprehensible model. This leads to the conflict of goals of wanting to construct powerful learning systems on the one hand, i.e., to rely on DRL, but on the other hand to be able to explain them afterwards with more comprehensible models, such as DTs.

DTs can be trained directly, which immediately leads to an interpretable model. On the other hand, DNNs are better regularized, which increases trainability [14]. This is particularly relevant when sampling efficiency is required, as in the training of DRL models where there may be long trajectories that need to be calculated by computationally intensive simulations. Thus, the goal is to achieve high trainability with high interpretability of the resulting model.

The main contribution to this problem of explainability of efficiently learned policies is that, in terms of input-output behavior, the presented approach transforms efficient-learnable FF-DNNs into compressed DTs to improve explainability, interpretability, and verifiability. The presented algorithm NN2EQCDT relies heavily on the equivalence description of DNNs and DTs in [15], but there are still research gaps to better use it for explainability, which will be addressed with the following contributions:

- The equivalence description of DNNs and DTs from Aytekin [15] is not so easy to implement, so this paper proposes a transformation to directly use models learned with the widely used Deep Learning (DL) framework PyTorch
- Using the equivalence transformation, the DT grows exponentially with branching. This problem is addressed by lossless compression for smaller but equivalent models, which enhances human interpretability.
- The dynamic compression method reduces the computation time significantly and may reduce the inference time of the DT.
- There may be constraints inherent in the system that affect the model but are not considered in the transformation. Therefore, these are included as invariants in the satisfiability check to further compress the DT.
- Finally, we provide an implementation [16] for the transformation of FF-DNN into equivalent, compressed DTs and the generation of visualizations from DTs.

This paper fits the conference because it aims to understand hidden knowledge of machine-learned DNNs cognitively through DTs. For the essential compression, satisfiability and other constraints are used to achieve an equivalent transformation instead of an approximation with uncertainty.

The rest of this paper is organized as follows: First, related work is presented in Section II. The construction of the equivalent compressed DT from a FF-DNN is described in general terms in Section III. All necessary components and details and their meaning are explained in later sections. In Section IV, the derivation for a right-handed linear transformation is described to be able to use PyTorch models. Dynamic path checking when adding subtrees to dynamic compression is described in Section V and further compression is described in Section VI. Furthermore, the application of the NN2EQCDT algorithm to a simple model is presented in Section VII. Finally, the presented approach is discussed in Section VIII as well as a conclusion is drawn and possible future work is described in Section IX.

II. Related Work

In general, there is a tradeoff between model readability and performance [12]. Tree-based models are, e.g., more readable than DNNs, but their performance is worse. Not only performance, but also explainability is crucial for the use of a system. If a system is not trustworthy, especially in critical environments, it will not be used. In the case of DRL, there may be concerns about correctness, or at least doubts that the black box system in question does not always behave as it should to achieve a particular goal.

There are different types of interpretability in terms of the scope and timing of information extraction [12]. Interpretable models are either global or local and either intrinsic or post-hoc. Here, scope refers to the explained domain of the model in question and the timing of information acquisition. An intrinsic model is directly interpretable by itself at creation time, like a DT. Post-hoc interpretable models are models that become interpetable only after creation, e.g., by a transformation or distillation of a black-box DNN model into an interpretable model.

DTs have a simple, understandable structure and are therefore easy to interpret [17], so they are intrinsic models. But they are not suitable to be used directly as policy representation of RL agents. Only DNN-based strategies can be efficiently obtained using existing DRL methods. One approach to optimize the tradeoff between predictive accuracy and interpretability is to train DTs from DNN-based policies or to use a more direct transformation for given states [18].

In [19], it is described that DTs can be trained from samples of pre-trained DNN policies with the (Q-)DAGGER and VIPER algorithms. Such imitation learning have the problem that much larger DTs than necessary are learned and the performance can be lower compared to the original DNN.

This approach uses efficient FF-DNN policies, but approximates the DT, which can then also become large, which in turn reduces the explainability. It is therefore less suitable for the use of the explainability of learned FF-DNN policies as agent controllers in CNIs.

III. Decision tree construction

FF-DNNs can equivalently be transformed into compressed DTs using the NN2EQCDT construction algorithm shown in Figure 1. The algorithm generates DTs by iterative computing and connecting subtrees with effective layer-wise filters from weight and bias matrices of neural networks. It shows accessing the final effective filters and computing the activation vector from the paths of the subtrees, as well as converting the final rules into expressions and compressing the whole tree. Here the algorithm is described in general and how it can be used. The individual components are explained in more detail in later sections.

```
1: $\hat{\boldsymbol{W}} = \boldsymbol{W}_0$
2: $\hat{\boldsymbol{B}} = \boldsymbol{B}_0^\top$
3: $rules$ = calc_rule_terms($\hat{\boldsymbol{W}}, \hat{\boldsymbol{B}}$)
4: $T$, $new\_SAT\_leaves$ = create_initial_subtree($rules$)
5: set_hat_on_SAT_nodes($T$, $new\_SAT\_leaves$, $\hat{\boldsymbol{W}}, \hat{\boldsymbol{B}}$)
6: for $i = 1, \ldots, n-1$ do
7:     $SAT\_paths$ = get_SAT_paths($T$)
8:     for $SAT\_path$ in $SAT\_paths$ do
9:         $\boldsymbol{a}$ = compute_a_along(SAT_path)
10:        $SAT\_leave$ = SAT_path[$-1$]
11:        $\hat{\boldsymbol{W}}, \hat{\boldsymbol{B}}$ = get_last_hat_of_leave($T$, $SAT\_leave$)
12:        $\hat{\boldsymbol{W}} = (\boldsymbol{W}_i \odot [(\boldsymbol{a}^\top)_{\times k}])\hat{\boldsymbol{W}}$
13:        $\hat{\boldsymbol{B}} = (\boldsymbol{W}_i \odot [(\boldsymbol{a}^\top)_{\times k}])\hat{\boldsymbol{B}} + \boldsymbol{B}_i^\top$
14:        $rules$ = calc_rule_terms($\hat{\boldsymbol{W}}, \hat{\boldsymbol{B}}$)
15:        $new\_SAT\_leaves$ =
           add_subtree($T$, $SAT\_leave$, $rules$, $invariants$)
16:        set_hat_on_SAT_nodes($T$, $new\_SAT\_leaves$,
           $\hat{\boldsymbol{W}}, \hat{\boldsymbol{B}}$)
17: convert_final_rule_to_expr($T$)
18: compress_tree($T$)
```

Figure 1. NN2EQCDT algorithm

The weight and bias matrices $\boldsymbol{W}_i$ and $\boldsymbol{B}_i$ from the FF-DNN model are processed layer by layer. These are used to compute rules that are used to add subtrees to the overall DT. This allows the DT to be built dynamically as the model is iterated layer by layer. From the second layer, when multiplying the weight and bias matrices, it is necessary to take into account the position of the node to which the generated subtree will be attached. This is done by applying the slope vector $\boldsymbol{a}$ to the current weight matrices. It represents the node position of the connection, since it is the vector of choices according to the ReLU activation function along the path from the root to the connection node.

When adding a node of a newly created subtree to the overall tree, each path from the root to the node in question is checked for satisfiability. If there can be no input so that its evaluation of the DT that takes this path, the node in question and thus further subtrees are not added to keep the size of the DT dynamically small. Finally, the last checks are converted into expressions, and the DT can be further compressed by removing unnecessary checks, since they are evaluated the same for all possible inputs.

IV. Derivation of the representation with right-handed linear transformation

DTs can be constructed from the effective weight matrices $\hat{\boldsymbol{W}}$ computed by spanning and connecting subtrees through them. The algorithm for this is shown in Figure 2. It is first motivated and then explained with its construction.

```
1: $\hat{\boldsymbol{W}} = \boldsymbol{W}_0$
2: $\hat{\boldsymbol{B}} = \boldsymbol{B}_0^\top$
3: for $i = 0, \ldots, n-2$ do
4:     $\boldsymbol{a} = [\,]$
5:     for $j = 0, \ldots, m_i - 1$ do
6:         if $(\hat{\boldsymbol{W}}_j \boldsymbol{x}_0^\top + \boldsymbol{B}_j^\top)^\top > 0$ then
7:             $\boldsymbol{a}$.append(1)
8:         else
9:             $\boldsymbol{a}$.append(0)
10:    $\boldsymbol{W}_{i+1} \in \mathbb{R}^{m_i \times k}, \boldsymbol{a} \in \mathbb{Z}_2^{m_i}$
11:    $\hat{\boldsymbol{W}} = (\boldsymbol{W}_{i+1} \odot [(\boldsymbol{a}^\top)_{\times k}])\hat{\boldsymbol{W}}$
12:    $\hat{\boldsymbol{B}} = (\boldsymbol{W}_{i+1} \odot [(\boldsymbol{a}^\top)_{\times k}])\hat{\boldsymbol{B}} + \boldsymbol{B}_{i+1}^\top$
13: return $(\hat{\boldsymbol{W}} \boldsymbol{x}_0^\top + \hat{\boldsymbol{B}})^\top$
```

Figure 2. Algorithm for calculation of effective weight matrices with right-handed linear transformation and bias for ReLU activation function, based on [15]

In the basis of the NN2EQCDT algorithm, the linear transformation is performed with a left multiplication of the weight matrix in [15], but there is no implementation given. For an implementation there was raised the requirement, that an algorithm must be able to use FF-DNN models in the format of the widely used DL framework PyTorch to be able to effienctly reuse existing models in a quasi standard format. But unfortunately, PyTorch uses a right instead of a left side multiplication of the weight matrices [20] as follows:

$$\boldsymbol{Y}_{\text{l}} = \boldsymbol{W}_{\text{l}}^\top \boldsymbol{X} + \boldsymbol{B} \qquad \boldsymbol{Y}_{\text{r}} = \boldsymbol{X}\boldsymbol{W}_{\text{r}}^\top + \boldsymbol{B}$$

To construct a DT from a Pytorch model consisting of linear layers with bias and applying the activation function $\sigma = \text{ReLU}$ between them, the layer-wise effective weight matrices $\hat{\boldsymbol{W}}_i$ must be computed using the right-handed linear transformation with bias as shown in Eq. (1) based on [15]. Here, the activation function is performed by multiplying the activation slopes element-wise by the weight matrices. The activation vector $\boldsymbol{a}$ must be repeated k times for the multiplication to match the size of the matrices to which it is applied. In the following equations, however, it is written simply as $\boldsymbol{a}$ when repeated to be analogous to [15].

$$\begin{aligned}
\hat{\boldsymbol{W}}_i^\top &= \sigma(\boldsymbol{x}_{i-1}\boldsymbol{W}_{i-1}^\top + \boldsymbol{B}_{i-1})\boldsymbol{W}_i^\top + \boldsymbol{B}_i \\
&= \sigma((\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top)^\top)\boldsymbol{W}_i^\top + \boldsymbol{B}_i \\
&= (\boldsymbol{a}_{i-1} \odot (\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top)^\top)\boldsymbol{W}_i^\top + \boldsymbol{B}_i \\
&= ((\boldsymbol{a}_{i-1}^\top \odot (\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top))^\top)\boldsymbol{W}_i^\top + \boldsymbol{B}_i \\
&= (\boldsymbol{W}_i(\boldsymbol{a}_{i-1}^\top \odot (\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top)))^\top + \boldsymbol{B}_i \\
&= ((\boldsymbol{W}_i^\top \odot \boldsymbol{a}_{i-1}^\top)^\top(\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top))^\top + \boldsymbol{B}_i \\
&= ((\boldsymbol{W}_i \odot \boldsymbol{a}_{i-1})(\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top))^\top + \boldsymbol{B}_i \\
&= (((\boldsymbol{W}_i \odot \boldsymbol{a}_{i-1})(\boldsymbol{W}_{i-1}\boldsymbol{x}_{i-1}^\top + \boldsymbol{B}_{i-1}^\top)) + \boldsymbol{B}_i^\top)^\top \quad (1)
\end{aligned}$$

The recursive form in Eq. (1) can be used to formulate a general closed of as shown in Eq. (2) based on [15]. It is equivalent to the right-handed linear transformation with bias and ReLU activation function.

$$\begin{aligned}\mathrm{NN}(\boldsymbol{x}_0) &= (\ldots((\boldsymbol{W}_1 \odot \boldsymbol{a}_0)(\boldsymbol{W}_0\boldsymbol{x}_0^\top + \boldsymbol{B}_0^\top) + \boldsymbol{B}_1^\top)\ldots)^\top \\ &= (\ldots(\underbrace{(\boldsymbol{W}_1 \odot \boldsymbol{a}_0)\boldsymbol{W}_0}_{\hat{\boldsymbol{W}}_{1,\boldsymbol{a}_0}}\boldsymbol{x}_0^\top + \underbrace{(\boldsymbol{W}_1 \odot \boldsymbol{a}_0)\boldsymbol{B}_0^\top + \boldsymbol{B}_1^\top}_{\hat{\boldsymbol{B}}_{1,\boldsymbol{a}_0}})\ldots)^\top\end{aligned} \tag{2}$$

The corresponding algorithm for a simple FF-DNN with linear transformation, bias, and ReLU activation function is shown in Figure 2. The subscript j of $\hat{\boldsymbol{W}}_j \in \mathbb{R}^{1\times fs}$ with $\boldsymbol{x}_0 \in \mathbb{R}^{bs\times fs}$ and a batch size of $bs \geq 1$ and a feature size of $fs \geq 1$ refers to the j-th row of the current $\hat{\boldsymbol{W}} \in \mathbb{R}^{k\times fs}$, but the subscript $i+1$ of $\boldsymbol{W}_{i+1}$ refers to the weight matrix of the $i+1$-th layer and not to a row. The same is true for the bias matrix. $[(\boldsymbol{a}^\top)_{\times k}]$ means that the transposed vector $\boldsymbol{a}^\top \in \mathbb{R}^{m_i\times 1}$ is repeated line by line k times.

To better understand the application of the algorithm in Figure 2, a simple example of converting the XOR function into a DT is given in Figure 3. The XOR function is represented by the following weight matrices of linear layers without bias as in the example for the EC-DT algorithm of [21]:

$$\boldsymbol{W}_0 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \boldsymbol{W}_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

Each coefficient of a row of $\hat{\boldsymbol{W}}_i$ is linearly expanded and used as a split point rule with ReLU activation function. After a layer, only the $\hat{\boldsymbol{W}}_{i+1,\boldsymbol{a}}$ with the respective previous activations $\boldsymbol{a}$ are used for branching.

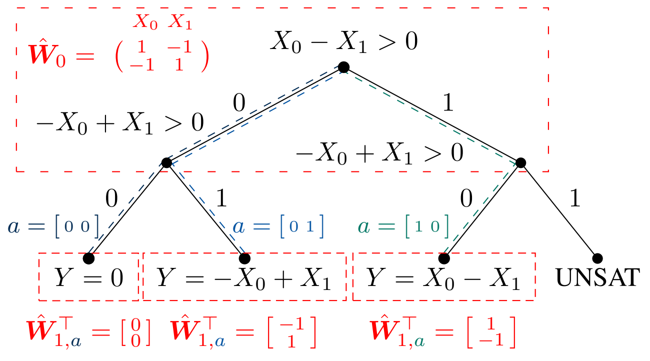
$\begin{array}{cc} X_0 & X_1 \end{array}$
$\hat{\boldsymbol{W}}_0 = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$
$X_0 - X_1 > 0$
0
1
$-X_0 + X_1 > 0$
$-X_0 + X_1 > 0$
0
1
0
1
$a = [\,0\ 0\,]$
$a = [\,0\ 1\,]$
$a = [\,1\ 0\,]$
$Y = 0$
$Y = -X_0 + X_1$
$Y = X_0 - X_1$
UNSAT
$\hat{\boldsymbol{W}}_{1,a}^\top = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$\hat{\boldsymbol{W}}_{1,a}^\top = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$
$\hat{\boldsymbol{W}}_{1,a}^\top = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

Figure 3. Simple example of an DT representing an XOR function constructed using the algorithm of Figure 2 without bias based on an example of the EC-DT algorithm [21]

In the algorithm in Figure 2, the calculation of $\hat{\boldsymbol{W}}$ requires the next weight or bias matrix. The iteration index can be equivalently converted from $i+1$ to i by iterating with **for** $i = 1, \ldots, n-1$ **do**. Thus, the last effective weight or bias matrix is required for the calculation of the current weight or bias matrix and of the next subtree to be attached. This is used in the NN2EQCDT algorithm in Figure 1 to be able to use the last instead of the next effective weight and bias matrices. To access them in the current iteration, they are stored in the iteration before by appending them to all SAT nodes of the subtrees they span.

In addition to the last effective and current weight or bias matrix, the activation vector $\boldsymbol{a}$ is also needed for the calculation of the new effective weight or bias matrix. In the concept algorithm in Figure 2, it is computed by using the input to branch to different activations. When converting the concept into an implementation of a dynamic design in Figure 1, the activation vector is required for each temporary SAT node for which the next effective weight and bias matrices are computed and attached as a subtree. Since the activation vector corresponds to the branches of a path, it is computed along it in the DT as seen in Figure 3.

V. Dynamic path checking when adding subtrees

In a DT spanned by the algorithm in Figure 2, there may be regions that are invalid due to conflicting categorizations. For example, the split point rule $x > 0$ and its inverse $x \leq 0$ contradict each other, so they are not jointly satisfiable. If such jointly unsatisfiable split point rules occur along a path in a tree, the path is invalid from that point on.

When a subtree is added to the entire DT, the joint satisfiability of all path rules is checked to avoid unnecessary calculation of additional paths that cannot be satisfied. If a path is not satisfiable from a certain node, there is no input where the evaluation of the DT follows the path from the node in question. The path is then terminated with an UNSAT node, as seen in Figure 3. Since the path cannot be followed any further, further checks and associated nodes and subtrees are not required. As a result, node concatenation and subtree computation can be stopped from this node. In this way, the DT is dynamically compressed in the design phase, but is still equivalent to the input FF-DNN, since only unreachable checks are omitted.

In addition to path rules, other constraints, such as input ranges or output checks for input ranges can also be used as invariants by expressing them as assertions. This allows the DT to be further compressed while maintaining equivalence, since further potentially unnecessary nodes can be omitted due to the invariants. All related assertions, such as path assertions and general input domain assertions can be written in the Satisfiability Modulo Theories (SMT) format and are checked for satisfiability together with the SMT solver Z3 [22].

Since path generation can be dynamically stopped at certain nodes, entire subtrees may not be computed. This can compress an DT and increase the overall computation time while maintaining an equivalent representation.

VI. Further tree compression

In addition to pruning the DT when it is created, it can be further compressed by deleting checks in it that are evaluated the same for their entire direct input space and are therefore not needed to distinguish inputs from each other. If an DT is created while its paths are dynamically checked for satisfiability, it can have UNSAT nodes as leaves, as seen in Figure 3. This example can be compressed, as seen in Figure 4, by removing the right-hand check $-X_0 + X_1 > 0$, which evaluates to false for all inputs, since the root check $X_0 - X_1 > 0$ evaluates to true in this branch.

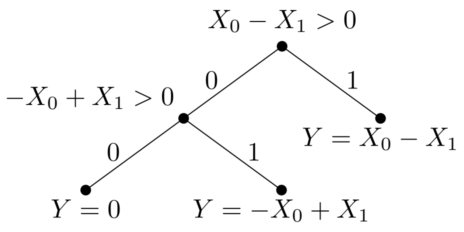

Figure 4. Simple compression example using DT from Figure 3

In any case, the rule of a parent node to an UNSAT node is evaluated on the further path of the non-UNSAT node, since the rules and their evaluations of the nodes preceding it in the path cuts the input space to this evaluation region. Since there is otherwise no input to evaluate, the rule check of a parent node to an UNSAT node can be omitted. Therefore, the entire parent node of an UNSAT node can be replaced by the non-UNSAT child node and its associated subtree, without the DT losing accuracy compared to the neural network model. This operation is therefore consistent with the goal of equivalent transformation of a neural network into a DT.

VII. Application to simple model

A simple controller model was trained using the DDPG algorithm for MountainCarContinuous-v0 environment (MCC) [23], [24]. Originally an actor model shown in Figure 5 was trained with larger hidden size of *hid* $= 64$. Since it is not necessary and more difficult to further analyse a model of that size a student model with *hid* $= 8$ and MSE-loss was destilled from the larger one only in the relevant region of $x \in [-1.2, 0.6]$ and $y \in [-0.7, 0.07]$ and stepsize of 0.1. It was visually found to perform about the same as the larger model [16].

```
nn.Sequential(
    nn.Linear(2,   hid, bias=True), nn.ReLU(),
    nn.Linear(hid, hid, bias=True), nn.ReLU(),
    nn.Linear(hid,   1, bias=True)
)
```

Figure 5. Actor model in PyTorch with variable hidden size

The smaller student model was then converted to an equivalent compressed DT using the algorithm from Figure 1. DTs are represented by networkx graphs that can be plotted with pyvis, as shown for the simple control example in Figure 6. The rules and expressions are node labels that are not visible at this zoom factor. Both models have the same output ($\delta = 1\text{e}{-4}$) for a sampled grid, strongly confirming the correctness of the implementation. The relevant input range was specified as an invariant for further compression.

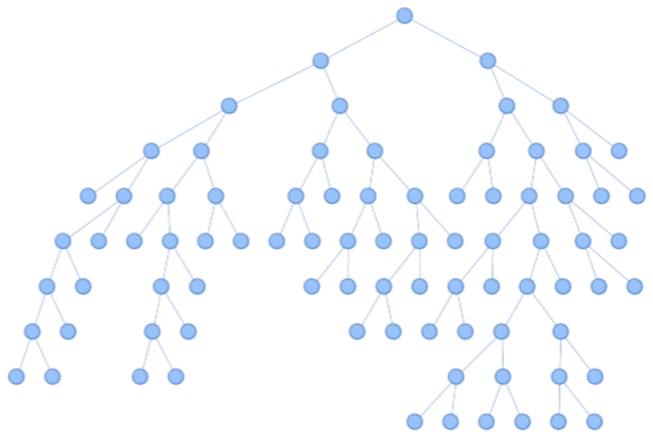

Figure 6. Compressed DT equivalent to the FF-DNN of a MCC controller

The different regions of a DT can be easily separated for 2D inputs. If the expressions for each input are to be visualized, they can be evaluated for the corresponding decision region and plotted as a third dimension, as shown in Figure 7. The points for the 2D regions (x and y) are obtained by implicitly plotting with sympy. The values for the z dimension are evaluated for each x and y point by the final expression and then drawn as a scatter plot using plotly. The gaps between the planes are due to a plotting problem, the input space is actually completely covered.

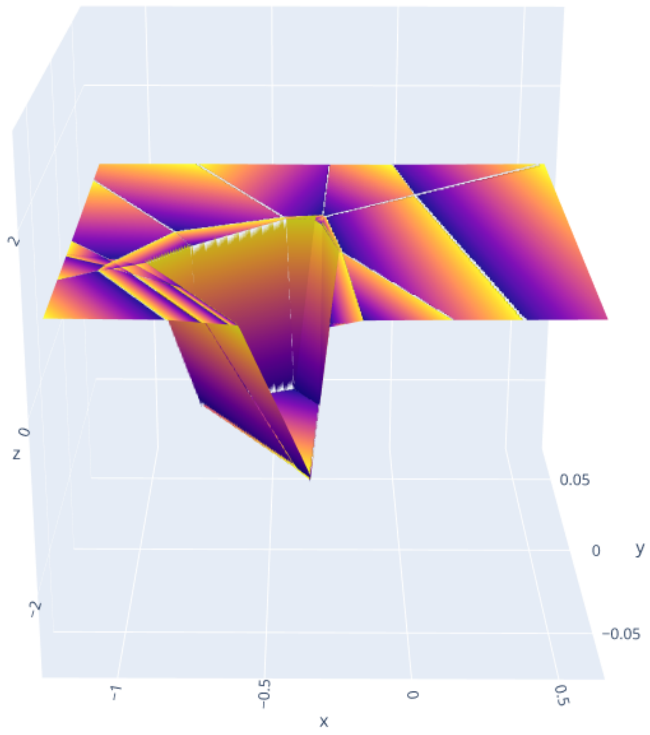

Figure 7. 3D visualization of DT regions for the MCC example

The compressed DT from the example in Figure 6 contains 83 nodes. It was computed with a median computation time of 9.75s as seen in Figure 8.

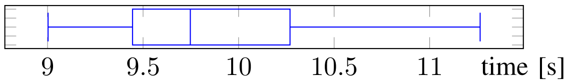

Figure 8. Boxplot ($n = 30$) for the computation time of the NN2EQCDT algorithm for the simple model

The amount of nodes of a DT according to the equivalence description of [15] without compression can be calculated with the following formula. It depends on the depth of each layer $d = \sum_{i=0}^{n-2} m_i$ with the number of filters in each layer m_i.

$$\#_{\text{nodes}} = \sum_{i=0}^{d-1} 2^i$$

This formula was tested by computing the DT with the equivalence description but without compression and summing the number of nodes for different Linear-ReLU FF-DNN architectures and hidden sizes. For an architecture as in Figure 5, it can be calculated as $d = 2 + 2hid$. Thus, for *hid* $= 8$, such a DT already consists of $\sum_{i=0}^{18-1} 2^i = 262143$ nodes, which corresponds to a compression ratio of 99.97% with respect to the number of nodes. At this size, the computation was aborted after a computation time of 1.5h. However, for other small sizes of *hid*, it was also observed that the computation time without compression starts to explode compared to the computation with compression.

VIII. Discussion

The principle of equivalence description of Aytekin [15] could be verified by implementation, testing and application to a simple model. The presented compression method seems to be a useful tool in transformation to increase the explainability of FF-DNN-based DRL policies, since the transformed, relatively small DT model and visualization can be used to trace actions back to observations. But in this form it is not meaningful enough to intuitively recognize a learned strategy. Probably, for this, the environment with its dynamics must be included in order to explain the agent's reactions and their effects on the next observations.

The transformation was successfully tested on a learned DRL model in a benchmarking environment. The results are summarized in Table I. However, the calculated compression ratio of 99.97% cannot be assumed to be representative without further evaluations. Also, a general statement about the performance of this approach for more complex environments and larger models cannot be made yet.

The NN2EQCDT algorithm can in principle transform arbitrary Linear-ReLU FF-DNN with any size for the input and output dimensions. The number of coefficients and variables in the transformed DT would then correspond to the size of the input dimension and the number of output values would then correspond to the size of the output dimension, but this has not yet been implemented due to implementation difficulties. Also, only three dimensions can be easily visualized together, more dimensions require more work and possibly information splitting or reduction.

Table I. Comparison of results or calculations for the construction of a DT from the simple model without and with compression of the NN2EQCDT algorithm

Compression	$\#_{\text{nodes}}$	Computation time
☐	262143	> 1.5h
☑	83	9.75s

IX. Conclusion and Future Work

In this paper, an algorithm capable of equivalently transforming a FF-DNN into a compressed DT was presented. Using a simple model, it was shown that a compressed DT may be significantly smaller than one without compression.

This approach can be used to trace the output regions exactly to the input regions. It can furthermore be a useful tool to accurately analyze the behavior of black-box models of FF-DNN. Furthermore, if a FF-DNN was learned as a DRL policy for an agent in a CNI, this approach has the potential to fundamentally strengthen the explainability, operator confidence, and hopefully the safety of the system.

For future work, better benchmarking of the algorithm in terms of computation time and compression ratio could be interesting. To better counteract unkown-unkowns in explaining FF-DNN models as DRL policies, the learned policies should be better analyzed with such an equivalently transformation approach. Therefore, we will attempt to combine the agent model with a learned world model and identify useful metrics based solely on the agent model and its traceability of output to input DT regions to indicate specific behaviors.

In particular, we will try to explain the learned strategy of ARL attack agents without unknown-unkowns using this approach. In addition, for visualization of more than three dimensions together, multiple combinations of three dimensions or other reduction methods, such as Principle Component Analysis (PCA) may be of interest. The implementation of the transformation could further be generalized to arbitrary sizes of input and output dimensions. And furthermore, for other use cases, it could also be interesting to use other layers for exact transformations.

Acknowledgements

This work was funded by the German Federal Ministry for Education and Research (BMBF) under Grant No. 01IS22071.

References

[1] V. Mnih *et al.*, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, pp. 1–9, 2013, [retrieved: 05, 2023]. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[2] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, vol. 30, AAAI Press, 2016, pp. 2094–2100.

[3] D. Silver *et al.*, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,” *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[4] J. Schrittwieser *et al.*, “Mastering atari, go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[5] S. Fujimoto, H. Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, PMLR, vol. 80, 2018, pp. 1587–1596.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, Jul. 19, 2017, [retrieved: 05, 2023]. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” *CoRR*, vol. abs/1801.01290, 2018, [retrieved: 05, 2023]. arXiv: 1801.01290. [Online]. Available: http://arxiv.org/abs/1801.01290.

[8] M. Naeem, S. T. H. Rizvi, and A. Coronato, “A gentle introduction to reinforcement learning and its application in different fields,” *IEEE access*, vol. 8, pp. 209 320–209 344, 2020.

[9] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, “Deep reinforcement learning framework for autonomous driving,” *Electronic Imaging*, vol. 29, no. 19, pp. 70–76, Jan. 2017, [retrieved: 05, 2023]. DOI: 10.2352/issn.2470-1173.2017.19.avm-023. [Online]. Available: https://library.imaging.org/ei/articles/29/19/art00012.

[10] T. T. Nguyen and V. J. Reddi, “Deep reinforcement learning for cyber security,” *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–17, 2021. DOI: 10.1109/TNNLS.2021.3121870.

[11] E. M. S. P. Veith, A. Wellßow, and M. Uslar, “Learning new attack vectors from misuse cases with deep reinforcement learning,” *Frontiers in Energy Research*, vol. 11, pp. 01–23, 2023, [retrieved: 05, 2023], ISSN: 2296-598X. DOI: 10.3389/fenrg.2023.1138446. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fenrg.2023.1138446.

[12] E. Puiutta and E. M. S. P. Veith, “Explainable reinforcement learning: A survey,” in *Machine Learning and Knowledge Extraction. CD-MAKE 2020*, vol. 12279, Dublin, Ireland: Springer, Cham, 2020, pp. 77–95. DOI: 10.1007/978-3-030-57321-8_5.

[13] E. M. Veith, “An architecture for reliable learning agents in power grids,” *ENERGY 2023 : The Thirteenth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pp. 13–16, 2023, [retrieved: 05, 2023], ISSN: 2308-412X. [Online]. Available: https://www.thinkmind.org/articles/energy_2023_1_30_30028.pdf.

[14] J. Ba and R. Caruana, “Do deep nets really need to be deep?” *Advances in Neural Information Processing Systems*, vol. 27, pp. 2654–2662, 2014.

[15] Ç. Aytekin, “Neural networks are decision trees,” *CoRR*, vol. abs/2210.05189, pp. 1–8, 2022, [retrieved: 05, 2023]. arXiv: 2210.05189. [Online]. Available: https://arxiv.org/abs/2210.05189.

[16] T. Logemann, *Nn2eqcdt implementation*, [retrieved: 05, 2023], 2023. [Online]. Available: https://gitlab.com/arl-experiments/nn2eqcdt.

[17] M. Du, N. Liu, and X. Hu, “Techniques for interpretable machine learning,” *Communications of the ACM*, vol. 63, no. 1, pp. 68–77, 2019.

[18] Y. Qing, S. Liu, J. Song, and M. Song, “A survey on explainable reinforcement learning: Concepts, algorithms, challenges,” *CoRR*, vol. abs/2211.06665, pp. 1–25, 2022, [retrieved: 05, 2023]. arXiv: 2211.06665. [Online]. Available: https://arxiv.org/abs/2211.06665.

[19] O. Bastani, Y. Pu, and A. Solar-Lezama, “Verifiable reinforcement learning via policy extraction,” *Advances in neural information processing systems*, vol. 31, pp. 2499–2509, 2018.

[20] PyTorch Foundation, *Pytorch linear*, [retrieved: 05, 2023], 2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html.

[21] D. T. Nguyen, K. E. Kasmarik, and H. A. Abbass, “Towards interpretable deep neural networks: An exact transformation to multi-class multivariate decision trees,” *CoRR*, vol. abs/2003.04675, pp. 1–57, 2020, [retrieved: 05, 2023]. arXiv: 2003.04675. [Online]. Available: https://arxiv.org/abs/2003.04675.

[22] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary. Proceedings 14*, Springer, vol. 4963, 2008, pp. 337–340.

[23] A. W. Moore, “Efficient memory-based learning for robot control,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-209, 1990, [retrieved: 05, 2023], pp. 1–248. DOI: 10.48456/tr-209. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-209.pdf.

[24] Farama Foundation, *Mountain car continuous*, Gymnasium Documentation, [retrived: 05, 2023], 2023. [Online]. Available: https://gymnasium.farama.org/environments/classic_control/mountain_car_continuous/.