On Neural Network Equivalence Checking Using SMT Solvers

Charis Eleftheriadis[1], Nikolaos Kekatos[1], Panagiotis Katsaros[1](✉), and Stavros Tripakis[1,2]

[1] School of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
{celefther,nkekatos,katsaros}@csd.auth.gr
[2] Khoury College of Computer Sciences, Northeastern University, Boston, USA
stavros@northeastern.edu

Abstract. Two pretrained neural networks are deemed (approximately) equivalent if they yield similar outputs for the same inputs. Equivalence checking of neural networks is of great importance, due to its utility in replacing learning-enabled components with (approximately) equivalent ones, when there is need to fulfill additional requirements or to address security threats, as is the case when using knowledge distillation, adversarial training, etc. In this paper, we present a method to solve various strict and approximate equivalence checking problems for neural networks, by reducing them to SMT satisfiability checking problems. This work explores the utility and limitations of the neural network equivalence checking framework, and proposes avenues for future research and improvements toward more scalable and practically applicable solutions. We present experimental results, for diverse types of neural network models (classifiers and regression networks) and equivalence criteria, towards a general and application-independent equivalence checking approach.

Keywords: Neural network · Equivalence checking · Verification · SMT

1 Introduction

Equivalence checking is the problem of checking whether two given artifacts (e.g. digital circuits, programs, etc.) are equivalent in some sense. Equivalence checking is standard practice in the domain of hardware design and the EDA industry [27,31,33,38]. There, digital circuits are subject to successive transformations for optimization and other purposes, and it is important to ensure that each successive design preserves the functionality of (i.e., is functionally equivalent to) the original.

In this paper, we are interested in the problem of equivalence checking for neural networks (NNs). For two pretrained NNs of different architectures, or of the same architecture with different parameters, the problem is to check whether they yield similar outputs for the same inputs. Contrary to equivalence checking for digital circuits, where strict functional equivalence is typically desired, *similar outputs* does not necessarily mean *identical* outputs in the case of NNs. As we shall see, the exact definition

S. Bogomolov and D. Parker (Eds.): FORMATS 2022, LNCS 13465, pp. 237–257, 2022.
https://doi.org/10.1007/978-3-031-15839-1_14

of equivalence depends on the application and NNs at hand (e.g. classifier, regression, etc.). Therefore, we consider both strict and approximate equivalences in this paper.

The equivalence checking problem for NNs is not only fundamental, intellectually interesting, and challenging. It is also motivated by a series of concerns similar to those motivating equivalence checking in the EDA industry, as well as recent developments in machine learning technology. The objective is to ensure that the smaller network is in some sense equivalent or approximately equivalent to the original one. Specifically, one application area is *neural network compression* [8]. Different compression techniques exist, e.g. *knowledge distillation*, *pruning*, *quantization*, *tensor decomposition*; see surveys in [8,26,29]. *Knowledge distillation* [15] is the process of transferring knowledge from a large neural network to a smaller one that may be appropriate for deployment on a device with limited computational resources. Another related application area includes the techniques widely known under the term *regularization* [21], which aim to lower the complexity of NNs in order to achieve better performance. In other cases, NNs used in systems with learning-enabled components [9] may have to be updated for a number of reasons [30]; for example, security concerns such as the need to withstand data perturbations (e.g. adversarial examples), or possibly incomplete coverage of the neural network's input domain. In the context of NN verification, several *abstraction* methods are often employed, for instance in order to reduce the complexity of the verification problem, e.g. see [3]. In all the aforementioned cases, the original and resulting NNs need to be functionally comparable in some way. A number of cases where equivalence checking for neural networks arise are also discussed in [19,28].

This paper presents the first, to our knowledge, systematic study of the equivalence checking problem for NNs, using a *Satisfiability Modulo Theory* (SMT) [5] approach. We define several formal notions of (approximate) equivalence for NNs, present encodings of the corresponding NN equivalence checking problems into satisfiability checking problems, and describe experimental results on examples of varying complexity.

In particular, the contributions of this paper are the following:

– We define several formal equivalence checking problems for neural networks based on various strict and approximate equivalence criteria that may each be appropriate for different neural network applications.
– We reduce the equivalence checking problem to a logical satisfiability problem using an SMT-based encoding. The approach is sound and complete in the sense that the two given NNs are equivalent iff the resulting SMT formula is unsatisfiable.
– We present a prototype implementation and experimental results including (i) sanity checks of our SMT-based encoding, and (ii) checks showing the equivalence and non-equivalence, as well as checking the scalability of SMT solvers for three diverse neural network applications covering the cases of classifiers (including the well-known MNIST dataset), as well as regression models.

The rest of this paper is organized as follows. Section 2 provides a formal definition of NN models. Section 3 presents diverse equivalence criteria for the wide range of common NN applications and formally defines the equivalence checking problem. Section 4 presents our SMT-based encoding for reducing equivalence checking to a logical satisfiability problem. Section 5 includes the experimental results. In Sect. 6, we review the related work and in Sect. 7 we provide our concluding remarks.

2 Preliminaries: Neural Networks

2.1 Notation

The set of real numbers is denoted by $\mathbb{R}$. The set of natural numbers is denoted by $\mathbb{N}$. Given some $x \in \mathbb{R}^n$ and some $i \in \{1, ..., n\}$, $x(i)$ denotes the i-th element of x.

2.2 Neural Networks

In general, a neural network (NN) can be defined as a function:

$$f : I \rightarrow O \tag{1}$$

where $I \subseteq \mathbb{R}^n$ is some input domain with n *features* and $O \subseteq \mathbb{R}^m$ an output domain.

For a NN image classifier, we typically have $I = [0, 255]^n \subseteq \mathbb{N}^n$ and a labeling function $L : \mathbb{R}^m \rightarrow \mathbb{N}$ that maps each $y \in O$ to some label $l \in \mathbb{N}$. For NNs solving regression problems, we have $I \subseteq \mathbb{R}^n$ and no labeling function.

The above definition of NNs is purely semantic. Concretely, a NN consists of layers of *nodes* (neurons), including one *hidden layer* (H) or more, beyond the layers of input (I) and output (O) nodes. Nodes denote a combination of affine value transformation with an *activation function*, which is typically piecewise linear or nonlinear. Value transformations are *weighted* based on how nodes of different layers are connected, whereas an extra term called *bias* is added per node. Weights (W) and biases (b) for all nodes are the NN's *parameters* and their values are determined via *training*.

Since every layer is multidimensional we use vectors and/or matrices to represent all involved operations. Let $\mathbf{x} \in \mathbb{R}^{1\times n}$ be the matrix denoting some $x \in I$. For a hidden layer with r nodes, $\mathbf{H}^{1\times r}$ represents the output of this hidden layer. Assuming that the hidden and output layers are fully connected, we denote with $\mathbf{W}^{(1)} \in \mathbb{R}^{n\times r}$ the hidden layer weights and with $\mathbf{b}^{(1)} \in \mathbb{R}^{1\times r}$ the biases associated with its nodes. Similarly, the output layer weights are denoted by $\mathbf{W}^{(2)} \in \mathbb{R}^{r\times m}$, where m refers to the number of output layer nodes, and $\mathbf{b}^{(2)} \in \mathbb{R}^{1\times m}$ denotes the corresponding biases. Then, the output $y = f(x)$ of the NN is given by $\mathbf{y} \in \mathbb{R}^m$ where $\mathbf{y}$ is computed as:

$$\mathbf{H} = \alpha(\mathbf{x}\,\mathbf{W}^{(1)} + \mathbf{b}^{(1)}) \tag{2}$$

$$\mathbf{y} = \alpha'(\mathbf{H}\,\mathbf{W}^{(2)} + \mathbf{b}^{(2)}) \tag{3}$$

where $\alpha(\cdot), \alpha'(\cdot)$ are the *activation functions* (e.g. sigmoid, hyperbolic tangent, etc.) applied to the vectors of the hidden and output layers element-wise. A common activation function is the Rectified Linear Unit (ReLU), which is defined, for $\chi \in \mathbb{R}$, as:

$$ReLU(\chi) = \begin{cases} \chi, & \text{if } \chi \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

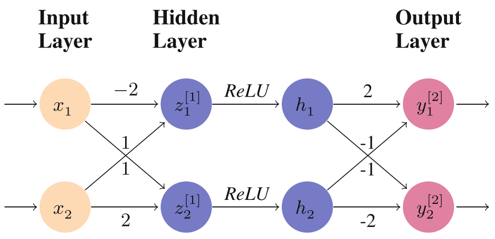

Fig. 1. Feedforward NN with an input layer of 2 inputs (x_1, x_2), an output layer with 2 outputs (y_1, y_2) and no activation function, and 1 hidden layer with 2 neurons and a ReLU activation function. The values on transitions refer to the *weights* and the superscript values to the biases.

Another example is the *hard tanh* function [10] (used in the experiments of Sect. 5) that typically serves as the output layer activation function of NNs trained for regression. For $\chi \in \mathbb{R}$, $hardtanh$ is defined as:

$$HardTanh(\chi) = \begin{cases} 1, & \text{if } \chi > 1 \\ -1, & \text{if } \chi < -1 \\ \chi, & \text{otherwise} \end{cases} \tag{5}$$

Multiple Hidden Layers: The NN definition provided above is easily generalised to multiple hidden layers $H, H', H'', \cdots$. We consider that weights and biases for all layers and nodes are fixed, since we focus on equivalence checking of NNs *after* training.

Example 1. Consider a simple feedforward NN with two inputs, two outputs, and one hidden layer with two nodes (Fig. 1). The selection of the weights and biases is done randomly, the activation function of the hidden layer is ReLU and there is no activation function for the output layer. For this example, Eq. (2) takes the form:

$$\begin{aligned}\begin{bmatrix} x_1 \ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \ b_2^{(1)} \end{bmatrix} &= \begin{bmatrix} x_1 \ x_2 \end{bmatrix} \cdot \begin{bmatrix} -2 & 1 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 1 \ 1 \end{bmatrix} \\ &= \begin{bmatrix} -2 \cdot x_1 + x_2 + 1 \ \ x_1 + 2 \cdot x_2 + 1 \end{bmatrix}\end{aligned}$$

Denote the result of the affine transformation of the NN's hidden layer by:

$$\begin{bmatrix} z_1 \ z_2 \end{bmatrix} = \begin{bmatrix} -2 \cdot x_1 + x_2 + 1 \ \ x_1 + 2 \cdot x_2 + 1 \end{bmatrix} \quad \text{and}$$
$$\mathbf{H} = \begin{bmatrix} h_1 \ h_2 \end{bmatrix} = \begin{bmatrix} ReLU(z_1) \ ReLU(z_2) \end{bmatrix}$$

The output $\mathbf{y}$ of the NN from Eq. (3) is:

$$\mathbf{y} = \begin{bmatrix} h_1 \ h_2 \end{bmatrix} \cdot \begin{bmatrix} 2 & -1 \\ -1 & -2 \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \ b_2^{(2)} \end{bmatrix} = \begin{bmatrix} 2 \cdot h_1 - h_2 + 2 \ \ -h_1 - 2 \cdot h_2 + 2 \end{bmatrix}$$

3 Strict and Approximate Equivalences for Neural Networks

In this section, we present various equivalence relations for NNs and we formulate the equivalence checking problem. Similar equivalence notions appeared recently in [19, 28].

3.1 Strict Neural Network Equivalence

Strict NN equivalence is essentially functional equivalence:

Definition 1 (Strict NN Equivalence). *For two neural networks $f : I \rightarrow O$ and $f' : I \rightarrow O$, we say that they are strictly equivalent, denoted $f \equiv f'$, if and only if:*

$$\forall x \in I, f(x) = f'(x) \tag{6}$$

Strict NN equivalence is a true equivalence relation, i.e., it is reflexive ($f \equiv f$ for any f), symmetric ($f \equiv f'$ iff $f' \equiv f$), and transitive ($f \equiv f'$ and $f' \equiv f''$ implies $f \equiv f''$).

However, strict NN equivalence can be a very restrictive requirement. For example, if we have two classifiers we may want to consider them equivalent if they *always* select the same top output class, even if the remaining output classes are not ordered in the same way. This motivates us to consider the following *approximate* notions of equivalence. These approximate "equivalences" need not be true equivalences, i.e., they may not satisfy the transitivity property (although they are always reflexive and symmetric).

3.2 Approximate Neural Network Equivalences Based on L_p Norms

As usual, we assume that $O \subseteq \mathbb{R}^m$. Let $\|y\|_p = norm_p(y)$ denoting the L_p-norm of vector $y \in O$, for $norm_p : O \rightarrow \mathbb{R}$ with $p = 1, 2, \infty$. For two vectors $y, y' \in O$, if $p = 1$ we obtain the Manhattan norm, $L_1(y, y') = \|y - y'\|_1 = \sum_{i=1}^{m} |y(i) - y'(i)|$, which measures the sum of differences between the two vectors. For $p = 2$, we refer to the Euclidean distance $L_2(y, y') = \|y - y'\|_2 = (\sum_{i=1}^{m} |y(i) - y'(i)|^2)^{\frac{1}{2}}$. Finally, for $p = \infty$, the L_∞ distance measures the maximum change to any coordinate:

$$L_\infty(y, y') = \| \, y - y' \, \|_\infty = \max(|y(1) - y'(1)|, \ldots, |y(m) - y'(m)|). \tag{7}$$

Then, we define the following notion of approximate equivalence:

Definition 2 ((p, ϵ)-approximate equivalence). *Consider two neural networks $f : I \rightarrow O$ and $f' : I \rightarrow O$, $norm_p : O \rightarrow \mathbb{R}$, and some $\epsilon > 0$. We say that f and f' are (p, ϵ)-approximately equivalent, denoted $f \sim_{p,\epsilon} f'$, if and only if:*

$$\forall x \in I, \quad \|f(x) - f'(x)\|_p < \epsilon \tag{8}$$

It can be seen that the relation $\sim_{p,\epsilon}$ is reflexive and symmetric.

3.3 Approximate Neural Network Equivalences Based on Order of Outputs

NN classifiers work essentially by computing output values and then mapping them to specific classes. Two such networks may be considered equivalent, if they always produce the same order of outputs, even though the output values might not be the same. For example, consider two classifiers f and f' over three possible output classes. Suppose that, for a given input, f produces $(0.3, 0.5, 0.2)$ and f' produces $(0.25, 0.6, 0.15)$. We may then consider that for this input the outputs of f and f' are equivalent, since they have the same order, namely, $2, 1, 3$ (assuming vector indices start at 1). If this happens for all inputs, we may want to consider f and f' (approximately) equivalent.

To capture the above notion of approximate equivalence, we introduce the function:

$$\texttt{argsort}_m : \mathbb{R}^m \to \mathcal{Z}_m, \text{ for } m \in \mathbb{N}$$

where $\mathcal{Z}_m \subseteq \{1, 2, 3, \ldots, m\}^m$ is the set of permutations of indices of the m elements. For a given $s \in \mathbb{R}^m$, $\texttt{argsort}_m(s)$ returns the permutation that sorts s in decreasing order. Thus, $\texttt{argsort}_3(0.3, 0.5, 0.2) = \texttt{argsort}_3(0.25, 0.6, 0.15) = (2, 1, 3)$. If two vector values are equal, `argsort` orders them from lower to higher index. This ensures determinism of the `argsort` function, e.g., $\texttt{argsort}_3(0.3, 0.4, 0.3) = (2, 1, 3)$.

Definition 3 (Top-k `argsort` equivalence). *Suppose $O \subseteq \mathbb{R}^m$. Consider two neural networks $f : I \to O$ and $f' : I \to O$, and some $k \in \{1, ..., m\}$. We say that f and f' are top-k `argsort` equivalent, denoted $f \approx_k f'$, if and only if*

$$\forall x \in I, \forall i \in \{1, ..., k\}, \Big(\texttt{argsort}_m\big(f(x)\big)\Big)(i) = \Big(\texttt{argsort}_m\big(f'(x)\big)\Big)(i) \qquad (9)$$

Top-k `argsort` equivalence requires the first k indices of the `argsort` of the outputs of f and f' to be equal. It is a true equivalence (reflexive, symmetric, and transitive).

A special case of top-k `argsort` equivalence is when $k = 1$. We call this `argmax` *equivalence*, with reference to the `argmax` function that returns the index of the maximum value of a vector, e.g., $\texttt{argmax}(0.3, 0.5, 0.2) = \texttt{argmax}(0.25, 0.6, 0.15) = 2$.

Definition 4 (`argmax` equivalence). *Consider the same setting as in Definition 3. We say that f and f' are* `argmax` equivalent *iff $f \approx_1 f'$.*

3.4 Hybrid L_p–`argsort` Equivalences

Approximate NN equivalences based on L_p norms may not respect the order of outputs, e.g., with $\epsilon = 1$, the output vectors $(1, 2)$ and $(2, 1)$ may be considered equivalent, even though the order is reversed. On the other hand, `argsort` and `argmax` based equivalences respect the order of outputs but may allow too large differences to be acceptable: the output vectors $(90, 7, 3)$ and $(40, 35, 25)$ both have the same order of outputs, but if the numbers are interpreted as confidence levels, we may not wish to consider them equivalent, due to the large discrepancy between the respective confidence values.

This discussion motivates the need for an equivalence, called *hybrid* $L_p -$ `argsort` equivalence, which considers *both* the order of outputs and their differences in value.

Definition 5 (Hybrid top-k `argsort` equivalence). *Suppose $O \subseteq \mathbb{R}^m$. Consider two neural networks $f : I \to O$ and $f' : I \to O$, and some $k \in \{1, ..., m\}$. Consider also $norm_p : O \to \mathbb{R}$, and some $\epsilon > 0$. We say that f and f' are* (p, ϵ)-approximately and top-k `argsort` equivalent *iff $f \sim_{p,\epsilon} f'$ and $f \approx_k f'$, i.e., they are both (p, ϵ)-approximately equivalent and top-k* `argsort` *equivalent.*

Specializing to $k = 1$ yields the following hybrid equivalence:

Definition 6 (Hybrid `argmax` equivalence). *Consider the same setting as in Definition 6. We say that f and f' are* (p, ϵ)-approximately and `argmax` equivalent *iff $f \sim_{p,\epsilon} f'$ and $f \approx_1 f'$.*

Definition 5 could be generalized further to involve different norms for each i-th element of the output vector, as well as a different bound ϵ_i for each $i \in \{1, ..., m\}$. We refrain from presenting such a generalization explicitly here, for the sake of simplicity.

3.5 The Neural Network Equivalence Checking Problem

Definition 7 (NN equivalence checking problem). *Given two (trained) neural networks f and f', and given a certain NN equivalence relation $\simeq \in \{\equiv, \sim_{p,\epsilon}, \approx_k\}$, and parameters p, ϵ, k as required, the* neural network equivalence checking problem *(NNECP) is to check whether $f \simeq f'$.*

3.6 Discussion of Application Domains for the Above Equivalence Relations

Strict equivalence is a true equivalence relation, but it might be impractical for realistic networks and numerical errors. Approximate equivalences can find several applications. For example, (p, ϵ)-equivalence can be used for multi-output learning problems [40]; specifically for i) regression problems where the goal is to simultaneously predict multiple real-valued output variables [6], and ii) classification tasks to ensure that the NN output values associated with every label are close to each other with respect to some L_p-norm. Approximate equivalences based on the order of outputs can be useful for classification. Top-1 accuracy is an established evaluation metric. For problems with hundreds of classes, e.g. ImageNet [11], it is common practice to present the top-5 accuracy along with top-1 accuracy when benchmarking. Top-k accuracy captures if any of the top k highest values of the output prediction vector is assigned the correct label. As such, it could be interesting to check if two NNs are equivalent using the top-k `argsort` equivalence. This equivalence could be important for Hierarchical Multi-label Classification [37], a classification task where the classes are hierarchically structured and each example may belong to more than one class simultaneously [7].

4 Neural Network Equivalence Checking Using SMT Solvers

Our approach to solving the NNECP is to reduce it to a logical *satisfiability* problem. The basic idea is the following. Suppose we want to check whether $f \simeq f'$, for two NNs $f : I \to O$ and $f' : I \to O$ and a given NN equivalence relation $\simeq$. We proceed as follows: (1) encode f into an SMT formula ϕ; (2) encode f' into an SMT formula ϕ'; (3) encode the equivalence relation $f \simeq f'$ into an SMT formula Φ such that $f \simeq f'$ iff Φ is unsatisfiable; (4) check, using an SMT solver, whether Φ is satisfiable: if not, then $f \simeq f'$; if Φ is satisfiable, then f and f' are not equivalent, and the SMT solver might provide a *counterexample*, i.e., an input violating the equivalence of f and f'.

This idea is based on the fact that the negation of $f \simeq f'$ can be encoded as a formula which asserts that there is $x \in I$ and $y, y' \in O$, such that $y = f(x), y' = f'(x)$,

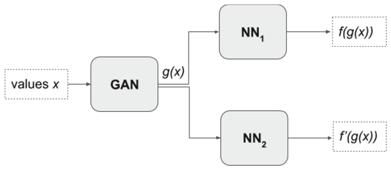

Fig. 2. Complete Neural Network Scheme for GAN generated inputs

with y and y' not satisfying the equivalence conditions imposed by $\simeq$. For example, for the case of strict NN equivalence, checking whether $f \equiv f'$ amounts to checking:

$$\neg\Big(\exists x \in I, y \in O, y' \in O, \ \ y = f(x) \wedge y' = f'(x) \wedge y \neq y'\Big)$$

This in turn amounts to checking that: $y = f(x) \wedge y' = f'(x) \wedge y \neq y'$ is unsatisfiable. In this case, we have $\phi := y = f(x)$, $\phi' := y' = f'(x)$, and $\Phi := \phi \wedge \phi' \wedge y \neq y'$.

We proceed to provide the details of building ϕ and ϕ' for given NNs, as well as Φ for the NN equivalence relations defined earlier. We note that although we present the SMT encoding of a single NN, our method is general and allows the two NNs to have different internal architectures (e.g., number and type of hidden layers, etc.).

4.1 Encoding Neural Networks as SMT Formulas

Input Variables. From Eq. (2), the input of a NN f is a vector $\mathbf{x} = [x_1, ..., x_n] \in \mathbb{R}^{1\times n}$. The SMT formula ϕ encoding f will have n *input variables*, denoted as $x_1, ..., x_n$.

Encoding Input Bounds. Sometimes, the inputs are constrained to belong in a certain region. For example, we might assume that the input lies between given lower and upper bounds. In such cases, we can add input constraints as follows:

$$\bigwedge_{j=1}^{n} l_j \leq x_j \leq u_j \tag{10}$$

with $l_j, u_j \in \mathbb{R}$ denoting the lower and upper bounds for the domain of input x_j.

Encoding Other Input Constraints. Often, we may only care about inputs that are "meaningful", e.g. if we want to preserve equivalence only for "reasonable" photos of human faces, or "reasonable" pictures of handwritten digits. Encoding such input constraints can be difficult. After all, if we had a precise way to encode such "meaningfulness" as a formal mathematical constraint, we may not need NNs in the first place.

One way to address this fundamental problem is by using *generative NNs* [14]. The idea is depicted in Fig. 2. The output of the generative network g is fed into f and f' that we wish to test for equivalence. The generative network g models the input constraints, e.g. the output of g may be pictures of human faces. Let N denote the complete network consisting of the connection of all three g, f, f', plus the equivalence constraints. We encode N as an SMT constraint and check for satisfiability. An example to satisfiability is an input x of g such that $f(g(x))$ and $f'(g(x))$ differ in the sense of the given equivalence relation. Then, $g(x)$ provides a counterexample to the equivalence of f and f'. Moreover, $g(x)$ has been generated by g, therefore it belongs by definition to the set of input constraints modeled by g (e.g., $g(x)$ is a picture of a human face).

Internal Variables. For each hidden layer, we associate the *internal variables* z_i for the affine transformation, and the internal variables h_i for the activation function.

Constraints Encoding the Affine Transformations. Consider a single hidden layer of f with r nodes. Then, from the affine transformation of Eq. (2), we derive the constraints:

$$\bigwedge_{j=1}^{r} \left(z_j = \sum_{k=1}^{n} x_k W_{kj}^{(1)} + b_j^{(1)} \right) \tag{11}$$

Constraints Encoding the ReLU Activation Function. If the activation function is $ReLU$, then its effect is encoded with the following constraints:

$$\bigwedge_{j=1}^{r} (z_j \geq 0 \wedge h_j = z_j) \vee (z_j < 0 \wedge h_j = 0) \tag{12}$$

Constraints Encoding the Hard *tanh* Activation Function. If the activation function is the hard $tanh$, then its effect is encoded with the following constraints:

$$\bigwedge_{j=1}^{r} (z_j \geq 1 \wedge h_j = 1) \vee (z_j \leq -1 \wedge h_j = -1) \vee (-1 < z_j < 1 \wedge z_j = h_j) \tag{13}$$

Other Activation Functions. The constraints described so far include atoms of the linear real arithmetic theory [20] that most SMT-solvers can check for satisfiability through decision procedures of various degrees of efficiency. Encoding other activation functions, like $Tanh$, $Sigmoid$, and $Softmax$, can be problematic as they include nonlinear and exponential terms that most SAT/SMT cannot handle. A workaround is to opt for "hard" versions of these activation functions, as it is done in [1,10]. In [24], $Softmax$ is replaced by a piecewise linear function called $Sparsemax$.

Multiple Hidden Layers and Output Layer. The constraints Eq. (11–13) are generalized to multiple hidden layers, say $H, H', \ldots$ (with $r, r', \ldots$ nodes, respectively).

The NN encoding is completed with the constraints for the output layer that are derived, for $\mathbf{y} \in \mathbb{R}^m$ from Eq. (3), as previously.

Example 2 (cont. of example 1). We assume that there are input constraints, i.e. $0 \leq x_1 \leq 1$ and $0 \leq x_2 \leq 1$ and we derive the SMT constraints for the NN of Fig. 1. For the affine transformation, we obtain $z_1 = -2 \cdot x_1 + x_2 + 1$ and $z_2 = x_1 + 2 \cdot x_2 + 1$. For the activation functions, we add the constraints $\{(z_1 \geq 0 \wedge h_1 = z_1) \vee (z_1 < 0 \wedge h_1 = 0)\}$, encoding $h_1 = ReLU(z_1) = \max(0, z_1)$ and $\{(z_2 \geq 0 \wedge h_2 = z_2) \vee (z_2 < 0 \wedge h_2 = 0)\}$, for $h_2 = ReLU(z_2) = \max(0, z_2)$. Finally, we add the output constraints $y_1 = 2 \cdot h_1 - h_2 + 2$ and $y_2 = -h_1 - 2 \cdot h_2 + 2$. The resulting SMT formula is

$$\begin{aligned}\phi := \big\{0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1 \wedge z_1 = -2x_1 + x_2 + 1 \wedge \big((z_1 \geq 0 \wedge h_1 = z_1) \vee \\ (z_1 < 0 \wedge h_1 = 0)\big) \wedge z_2 = x_1 + 2x_2 + 1 \wedge \big((z_2 \geq 0 \wedge h_2 = z_2) \vee (z_2 < 0 \\ \wedge h_2 = 0)\big) \wedge y_1 = 2 \cdot h_1 - h_2 + 2 \wedge y_2 = -h_1 - 2 \cdot h_2 + 2\big\}\end{aligned}$$

4.2 Encoding of the Equivalence Relation

As mentioned at the beginning of this section, to check the equivalence of two NNs f and f', we need to generate, first, their encodings ϕ and ϕ' (Sect. 4.1), and then, the encoding of the (negation of the) equivalence relation. The latter encoding is described next. We assume that f and f' have the same number of outputs m, and we let $\mathbf{y} = (y_1, ..., y_m)$ and $\mathbf{y}' = (y'_1, ..., y'_m)$ denote their respective output variables.

Strict Equivalence Checking. Strict equivalence (c.f., Definition 1) requires that $\mathbf{y} = \mathbf{y}'$. To reduce this verification problem to a satisfiability problem, we encode the negation of the above constraint, more specifically:

$$\bigvee_{i=1}^{m} y_i \neq y'_i \tag{14}$$

(p, ϵ)-Approximate Equivalence Checking. (p, ϵ)-approximate equivalence (c.f., Definition 2) requires that $\|\mathbf{y} - \mathbf{y}'\|_{\mathbf{p}} < \epsilon$. Again, we encode the negation:

- for $p = 1$,

$$\sum_{i=1}^{m} |y_i - y'_i| \geq \epsilon \tag{15}$$

- for $p = 2$, it would be expected to be

$$\Big(\sum_{i=1}^{m} |y_i - y'_i|^2\Big)^{\frac{1}{2}} \geq \epsilon$$

  which involves the square-root function that yields an *undecidable* constraint. Instead, our encoding takes the form,

$$u = \sum_{i=1}^{m} |y_i - y'_i|^2 \wedge u = v \cdot v \wedge v \geq 0 \wedge v \geq \epsilon \tag{16}$$

– and for $p = \infty$,

$$\bigvee_{i=1}^{m} |y_i - y'_i| \geq \epsilon \tag{17}$$

argmax Equivalence Checking. This equivalence type (c.f., Definition 4) requires that $\texttt{argmax}(y) = \texttt{argmax}(y')$. Again, we wish to encode the negation, i.e., $\texttt{argmax}(y) \neq \texttt{argmax}(y')$. This can be done by introducing the macro $\mathsf{argmaxis}(y, i, m)$ which represents the constraint $\texttt{argmax}(y) = i$, assuming the vector y has length m. Then, $\texttt{argmax}(y) \neq \texttt{argmax}(y')$ can be encoded by adding the constraints below:

$$\bigvee_{\substack{i,i' \in \{1,\ldots,m\} \\ i \neq i'}} \mathsf{argmaxis}(y, i, m) \wedge \mathsf{argmaxis}(y', i', m) \tag{18}$$

where argmaxis is defined as follows:

$$\mathsf{argmaxis}(y, i, m) := \Big(\bigwedge_{j=1}^{i-1} y_i > y_j\Big) \wedge \Big(\bigwedge_{j=i+1}^{m} y_i \geq y_j\Big) \tag{19}$$

For example, for $m = 2$, we have:

$$\begin{aligned} \mathsf{argmaxis}(y, 1, 2) &= y_1 \geq y_2 \ \ \mathsf{argmaxis}(y', 1, 2) = y'_1 \geq y'_2 \\ \mathsf{argmaxis}(y, 2, 2) &= y_2 > y_1 \ \ \mathsf{argmaxis}(y', 2, 2) = y'_2 > y'_1 \end{aligned}$$

and the overall constraint encoding $\texttt{argmax}(y) \neq \texttt{argmax}(y')$ becomes:

$$(y_1 \geq y_2 \wedge y'_2 > y'_1) \vee (y_2 > y_1 \wedge y'_1 \geq y'_2)$$

Example 3 (cont. example 2). For the NN f, we have obtained the formula ϕ. Now assume that there is a second NN f' that has the same number of inputs and outputs as f. We define as y' the outputs of f' and the constraints are SMT encoded via ϕ'. The complete SMT formula Φ for the different equivalence relations is:

$$\begin{aligned}
\Phi_{strict} &:= \{\phi \wedge \phi' \wedge \bigvee_{i=1}^{m} y_i \neq y'_i\} \\
\Phi_{(1,\epsilon)-\text{approx}} &:= \{\phi \wedge \phi' \wedge \sum_{i=1}^{m} |y_i - y'_i| \geq \epsilon\} \\
\Phi_{(2,\epsilon)-\text{approx}} &:= \{\phi \wedge \phi' \wedge u = \sum_{i=1}^{m} |y_i - y'_i|^2 \wedge u = v \cdot v \wedge v \geq 0 \wedge v \geq \epsilon\} \\
\Phi_{(\infty,\epsilon)-\text{approx}} &:= \{\phi \wedge \phi' \wedge \bigvee_{i=1}^{m} |y_i - y'_i| \geq \epsilon\} \\
\Phi_{argmax} &:= \{\phi \wedge \phi' \wedge \bigvee_{\substack{i,i' \in \{1,\ldots,m\} \\ i \neq i'}} \mathsf{argmaxis}(y, i, m) \wedge \mathsf{argmaxis}(y', i', m)\}
\end{aligned}$$

Algorithm 1 introduces an implementation in the Z3 SMT solver[1] for `argmax`, which in our experiments showed better scalability behaviour when compared with the straightforward implementation of the aforementioned encoding for `argmax`.

Algorithm 1: Pseudo-code of `argmax` implementation in Z3

```
Require: Vector $y$ of length $m$
Ensure: argmax($y$)
  $y_{max} \leftarrow y(m)$
  $i_{max} \leftarrow m$
  for $i = m - 1$ to $i = 1$ step -1 do
    if $y(i) > y_{max}$ then
      $y_{max} \leftarrow y(i)$
    end if
    if $y(i) = y_{max}$ then
      $i_{max} \leftarrow i$
    end if
  end for
  return $i_{max}$
```

Hybrid Equivalence Checking. The encoding of the hybrid equivalence relations in Sect. 3.4 consists of combining the corresponding constraints of the (p, ϵ)-approximate equivalence and the `argsort`/`argmax` equivalence with a logical conjunction.

4.3 Optimizing the Encoding

The encoding presented so far is not optimal in the sense that it uses more SMT variables than strictly necessary. Many internal variables can be eliminated and replaced by their corresponding expressions in terms of other variables. In particular, the z_i variables encoding the affine transformations (as in Eq. 11) and the h_i variables for encoding the activation function can be easily eliminated.

5 Experimental Results

In this section, we report the results of experiments on verifying the equivalence of two NNs. We have used the SMT solver Z3 to check the satisfiability of all constraints that encode NN equivalence. The experiments were conducted on a laptop with a 4-core 2.8 GHz processor and 12 GB RAM. Our problems concern with NNs of different sizes, for checking their equivalence, with respect to the equivalence relations of Sect. 3. We focus on two main categories of supervised learning problems, i) classification (two case studies), and ii) regression (one case study). The source code of the implementation of the NNECP for the equivalence encoding of Sect. 4 is provided online[2].

[1] https://z3prover.github.io/api/html/.
[2] https://github.com/hariselef/NNequiv_Experiments.

Bit-Vec Case Study – Classification: A bit vector (Bit-Vec) is a sequence of bits. We consider that the inputs of our NN classifier are 10-bit vectors and the targets (labels) are either True (1) or False (0). The models we check for equivalence are Feed-Forward Multi-Layer Perceptrons and we focus on two architectures. In the first case, there is a single hidden layer, while in the second, there are two hidden layers; NNs have the same number of nodes per layer. We experimented with 8 different models per architecture and for each model we increased step-wise the number of nodes per layer. NNs have been trained with the objective to "learn" that for a vector with 3 or more consecutive 1s, the output label is True and the output label is False otherwise.

MNIST Case Study – Classification: The second use case is the popular *MNIST* dataset on image classification. MNIST contains 70,000 grayscale images, from which 60,000 are for training and the rest for testing the models' performance. The size of images is 28×28 (pixels), and every pixel value is a real in the range $[0, 1]$. We experimented with the same two architectures used for the Bit-Vec NNs and 5 models for each of them.

Automotive Control – Regression: For the regression case study, the goal is to use NNs that approximate the behaviour of a Model Predictive Controller, which has been designed for an automotive lane keeping assist system. The dataset contains 10,000 instances with six features representing different system characteristics obtained using sensors, along with the resulting steering angle that the car should follow (target). More details for the case study that we have reproduced can be found in Mathworks website[3].

5.1 Sanity Checks

The main goals of this set of experiments are: i) to ensure that our prototype implementation for equivalence checking does not have any bugs (sanity checks) and ii) to conduct a scalability analysis of the computational demands for NNs of increasing complexity when they are checked for equivalence, with the criteria of Sect. 3.

To this end, we verified the equivalence of two identical NNs, for two different architectures, both for the BitVec case study and the MNIST. The results are shown in Tables 1, 2 for the BitVec case study and in Tables 3, 4 for the MNIST. We report the number of nodes per hidden layer, the number of trainable parameters and the total number of variables in the formula generated for the SMT solver. For the BitVec experiments and for each equivalence relation, tables show the average time in seconds over 10 runs for the SMT solver to verify the equivalence of the identical NNs (standard deviation in all cases was less than 3%). For the MNIST experiments, we report the same results, but all sanity checks were conducted only once, since these experiments took more time to complete. In all cases, the SMT solver returned *UNSAT*, which correctly indicates that the two NNs are equivalent, as expected.

[3] https://www.mathworks.com/help/reinforcement-learning/ug/imitate-mpc-controller-for-lane-keeping-assist.html.

Table 1. Sanity check for the BitVec case study - 1st Architecture; all equivalences are true, i.e. all SMT formulas are *UNSAT*; the values in columns 4–8 show the computational time in seconds.

# nodes per layer	**# params**	**# SMT variables**	**Strict Equiv.**	L_1 **Equiv.**	L_2 **Equiv.**	L_∞ **Equiv.**	**Argmax Equiv.**
10	132	498	0.06	0.07	0.06	0.07	0.06
20	262	978	0.1	0.1	0.09	0.1	0.1
35	457	1698	0.17	0.17	0.14	0.17	0.17
50	652	2418	0.23	0.24	0.21	0.24	0.23
100	1302	4818	0.44	0.45	0.40	0.45	0.45
150	1952	7218	0.61	0.63	0.57	0.62	0.65
200	2602	9618	0.84	0.85	0.75	0.85	0.84
300	3902	14418	1.23	1.25	1.1	1.25	1.25

Table 2. Sanity check for the BitVec case study - 2nd Architecture; all equivalences are true, i.e. all SMT formulas are *UNSAT*; the values in columns 4–8 show the computational time in seconds.

# nodes per layer	**# params**	**# SMT variables**	**Strict Equiv.**	L_1 **Equiv.**	L_2 **Equiv.**	L_∞ **Equiv.**	**Argmax Equiv.**
5	97	378	0.04	0.04	0.04	0.04	0.04
10	242	938	0.1	0.09	0.08	0.09	0.1
15	437	1698	0.15	0.15	0.14	0.16	0.15
20	682	2658	0.24	0.24	0.20	0.23	0.23
30	1322	5178	0.4	0.39	0.37	0.39	0.42
40	2162	8498	0.62	0.62	0.58	0.63	0.63
50	3202	12618	0.87	0.91	0.85	0.88	0.92
60	4442	17538	1.17	1.2	1.21	1.16	1.23

Table 3. Sanity check for the MNIST case study - 1st Architecture; all equivalences are true, i.e. all SMT formulas are *UNSAT*; the values in columns 4–8 show the computational time in seconds.

# nodes per layer	**# params**	**# SMT variables**	**Strict Equiv.**	L_1 **Equiv.**	L_2 **Equiv.**	L_∞ **Equiv.**	**Argmax Equiv.**
10	7960	32424	2.5	2.53	2.2	2.63	2.67
30	23860	95624	7.52	7.73	6.3	7.63	7.5
50	39760	158824	12.2	12.4	10.4	12.8	12.4
100	79510	316824	24.5	24.3	21.9	25	24.6
200	159010	632824	48.4	55.1	44.8	48.8	48.2
300	238510	948824	74	74	74	62.4	73
500	397510	1580824	121	124	110	128	119
750	596260	2370824	182	193	167	203	182
1000	795010	3160824	241	247	220	257	256
1300	1033510	4108824	314	336	283	331	321
1700	1351510	5372824	420	434	394	435	437
2000	1590010	6320824	467	512	483	492	508

Table 4. Sanity check for the MNIST case study - 2nd Architecture; all equivalences are true, i.e. all SMT formulas are *UNSAT*; the values in columns 4–8 show the computational time in seconds.

# nodes per layer	# params	# SMT variables	Strict Equiv.	L_1 Equiv.	L_2 Equiv.	L_∞ Equiv.	Argmax Equiv.
10	8070	32864	2.7	2.7	2.3	2.54	2.57
30	24790	99344	7.55	7.86	6.7	7.5	7.5
50	42310	169024	13.2	12.9	11.0	13.7	12.9
100	89610	357224	26.5	27.5	24.9	26	27.7
200	199210	793624	58.7	62.2	50.6	61.6	64.4
300	328810	1310024	99	100	90	99	101
500	648010	2582824	194	194	167	192	192
750	1159510	4623824	334	340	298	354	349
1000	1796010	7164824	524	530	514	523	560
1300	2724810	10874024	797	779	784	836	857
1700	4243210	16939624	1225	1161	1359	1237	1223
2000	5592010	22328824	1435	1530	2837	1549	1581

5.2 Equivalence Checking

In the second set of experiments, we perform equivalence checking between NNs of different architectures. The solver will return *UNSAT* if the NNs are equivalent or *SAT* if they are not. This allows us to gain insight into the efficiency and the scalability bounds of the proposed encoding for both possible outcomes. We expect that when the two NNs are not equivalent (*SAT*) the solver is quite likely to return a counterexample shortly. A time limit of 10 min was set for the solver to respond and when this did not happen the end result was recorded as a "timeout".

Experiments with NN Classifiers. A series of experiments focused on the equivalence checking for NN classifiers, i.e. those trained for the BitVec and MNIST case studies. The pairs of NNs that were compared consist of the two NN architectures that were also used for the sanity checks (Sect. 5.1). Thus, the first pair of NNs includes the NN referred in the first line of Table 1 and the NN referred in the first line of Table 2, and so on. For the BitVec equivalence checking experiments, Table 5 summarizes the obtained results, for all equivalence relations apart from the $(2, \epsilon)$-approximate equivalence, which takes much more time than the set time limit and the results for two pairs of NNs are shown separately in Table 7. For each pair of NNs in Tables 5 and 7, the answer of the SMT solver is reported (*SAT* or *UNSAT*) along with the time that it took to respond. For the ϵ-approximate equivalences, we also note the ϵ values, for which the NN equivalence was checked. The reason for using relatively big values for ϵ is that the NNs do not have a nonlinear activation function in the output layer, e.g. sigmoid, and there was no way to guarantee that the outputs would scale in the same value ranges. The term **MME**, in some cells of the tables, stands for Maximum Memory Exceeded and is the reason for which the solver fails to respond, when not having reached the time limit of 10 min. Table 6 (and Table 7 for $(2, \epsilon)$-approximate equivalence) report the corresponding results, for the MNIST case study.

When the outcome of equivalence checking is *SAT*, the solver returns a counterexample (NN input) that violates the equivalence relation. If the input space is finite, it may be possible to obtain all counterexamples, as in the BitVec case study, for which we found two counterexamples of `argmax` equivalence for the NNs of the pair *model*_1_4 vs *model*_2_4. The NN predictions for all possible inputs were then tested and it was confirmed that the two counterexamples are the only ones that violate `argmax` equivalence. Beyond seeing them as an extra sanity check, the counterexamples of equivalence checking may be useful, toward improving the NN robustness. However, we have not yet developed a systematic way to utilize them for this purpose. For most applications, as is the case in MNIST, it may not be feasible to obtain all possible counterexamples, due to the size of their input space. Moreover, often there is no easy way to limit the counterexample search to only "meaningful" inputs. This is the case in the MNIST case study, where the most common counterexample in the equivalence checking

Table 5. Equivalence checking for the BitVec case study

Model Pairs	**# SMT variables**			**Strict Equiv.**	$L_1 > 5$	$L_\infty > 10$	**Argmax Equiv.**
	Input	***Internal***	***Output***				
model_1_1 vs model_2_1	10	424	4	SAT/0.042 s	UNSAT/35 s	UNSAT/48 s	SAT/0.23 s
model_1_2 vs model_2_2	10	944	4	SAT/0.075 s	SAT/0.20 s	UNSAT/157 s	SAT/0.4 s
model_1_3 vs model_2_3	10	1630	4	SAT/0.124 s	UNSAT/385 s	UNSAT/531 s	UNSAT/182 s
model_1_4 vs model_2_4	10	2524	4	SAT/0.19 s	SAT/245 s	Timeout	SAT/86 s
model_1_5 vs model_2_5	10	4984	4	SAT/0.35 s	Timeout	MME/509 s	SAT/240 s
model_1_6 vs model_2_6	10	7844	4	SAT/0.5 s	Timeout	MME/568 s	SAT/450 s
model_1_7 vs model_2_7	10	11104	4	SAT/0.75 s	Timeout	MME/588 s	Timeout
model_1_8 vs model_2_8	10	15964	4	SAT/1.13 s	Timeout	Timeout	Timeout

Table 6. Equivalence checking for the MNIST case study

Model Pairs	**# SMT variables**			**Strict Equiv.**	$L_1 > 5$	$L_\infty > 10$	**Argmax Equiv.**
	Input	***Internal***	***Output***				
mnist_1_1 vs mnist_2_1	784	31840	20	SAT/2.1 s	SAT/44 s	SAT/42 s	SAT/41 s
mnist_1_2 vs mnist_2_2	784	96680	20	SAT/6.2 s	SAT/16 s	SAT/17 s	SAT/17 s
mnist_1_3 vs mnist_2_3	784	163120	20	SAT/10.3 s	SAT/28 s	SAT/28 s	MME/230 s
mnist_1_4 vs mnist_2_4	784	336220	20	SAT/21 s	SAT/56 s	SAT/ 57 s	SAT/54 s
mnist_1_5 vs mnist_2_5	784	712420	20	SAT/45 s	SAT/120 s	SAT/120 s	SAT/118 s

Table 7. Equivalence checking on BitVec and MNIST under L_2 ϵ-approximate equivalence

Model Pairs	**# SMT variables**			$L_2 > 1$	$L_2 > 10$
	Input	***Internal***	***Output***		
model_1_1 vs model_2_1	10	424	4	SAT/35 s	UNSAT/1494 s
model_1_2 vs model_2_2	10	944	4	SAT/105 s	UNSAT/2793 s
mnist_1_1 vs mnist_2_1	784	31840	20	MME/15436 s	-

experiments was an input vector filled with the value 0.5, which corresponds to a grayscale image whose pixels have all the same color, i.e. it is not a digit.

Experiments with Regression NNs. In another case study, we focused on NNs that serve as controllers in lane keeping assistant systems. Table 8 presents the details for the features and the outputs of the regression NNs for this case, as well as the valid value ranges. The pairs of NNs that were checked for equivalence consist of different versions of the same NN, produced through varying the number of epochs, for which the model was trained before being verified. We experimented with three versions of the NN controller trained for $30, 35$ and 40 epochs. The `argmax` equivalence is meaningless, since there is only one output variable. Additionally, for the same reason, the results in Table 9 refer only to the $(1, \epsilon)$-approximate equivalence, since the various L_p norms are indistinguishable, when they are applied to scalar values.

Experiments with Weight Perturbations. Table 10 reports the results of a set of experiments with a NN for MNIST, in which we have randomly altered the values of some weights before checking the equivalence with the original NN. In this way, since it is very likely to have a pair of equivalent NNs, the solver is forced to search for almost all possible inputs before reaching the UNSAT result. We observe that when having altered two weights (out of 7960) the solver reaches the 10 min timeout.

Table 8. Regression Problem Input Characteristics – Constraints

Type/Parameter	Answer/Value	Remarks
output/target	$[-1.04, 1.04]$	steering angle $[-60, 60]$
input range x_1	[-2,2]	v_x (m/s)
input range x_2	[-1.04, 1.04]	rad/s
input range x_3	[-1,1]	m
input range x_4	[-0.8, 0.8]	rad
input range x_5	[-1.04, 1.04]	u_0 (steering angle)
input range x_6	[-0.01,0.01]	ρ

Table 9. Equivalence checking for the Regression NNs

Model Pairs	# SMT variables			Strict Equivalence	$L_1 > 0.5$
	Input	*Internal*	*Output*		
MPC_30 vs MPC_35	6	17912	2	SAT/1.95 s	Timeout
MPC_30 vs MPC_40	6	17912	2	SAT/1.97 s	Timeout
MPC_35 vs MPC_40	6	17912	2	SAT/1.98 s	Timeout

Table 10. Equivalence checking with weight perturbations on the MNIST case study

Model	**# SMT variables**			**# weight changes**	**Value range**	$L_1 > 5$	$L_\infty > 10$	**Argmax Equiv.**
	Input	***Internal***	***Output***					
mnist_1_1	784	31840	20	1	1e−1 - 1e−6	UNSAT/30 s	UNSAT/30 s	Timeout
mnist_1_1	784	31840	20	2	1e−1 - 1e−6	Timeout	Timeout	Timeout

6 Related Work

Verification of NNs for various kinds of properties, such as safety, reachability, robustness, is of fast-growing interest, due to the many, often critical applications, in which NNs are employed. The authors of [23] present several algorithms for verifying NNs and classify them into three categories: reachability-based, optimization-based, and search-based. Typically, SMT-based approaches belong to the latter category. Two comprehensive surveys that include the verification of NNs using SMT solvers are given in [22] and [16]. An extensive survey of methods for the verification and validation of systems with learning enabled-components, not only NNs, is given in [39]. The equivalence checking problem for NNs is different from other verification problems for NNs: (a) in robustness verification, the goal is to check whether the output of a NN remains stable despite perturbations of its input; (b) for input-output verification of a NN, the goal is to check whether for a given range of input values, the output of the NN belongs in a given range of output values.

Other works on the equivalence of NNs are [28] and [19,35]. [28] only considers the case of strict equivalence where the outputs of the two networks must be identical for all inputs. We also define notions of approximate equivalence, as we believe strict equivalence is often too strong a requirement. [28] is also based on a SAT/SMT based encoding of the equivalence checking problem, but the overall approach is applicable only to a specific category of NNs, the so-called binarized NNs [2,17] that are not widely used in many different real-life applications. Approximate NN equivalence checking is also studied in [19,35], but their approach is based on mixed-integer linear programming (MILP) instead of SMT encoding. [19] applies the solution only to restricted regions of the input space, within a radius around the training data, whereas [35] introduces an abstraction-based solution. None of these works allows checking of hybrid equivalence.

The work in [30] focuses on the relationship between two NNs, e.g. whether a modified version of a NN produces outputs within some bounds relative to the original network. A “differential verification” technique is proposed consisting of a forward interval analysis through the network’s layers, followed by a backward pass, which iteratively refines the approximation, until having verified the property of interest. Differential verification is related to equivalence checking, but it is actually a different problem.

In [32], the authors adopt abstractions of the input domain using zonotopes and polyhedra, along with an MILP solver for verifying properties of NNs. An SMT-based verification method for a single NN is also presented in [18], whose applicability is limited only to NNs with ReLU activation functions. Finally, an interesting symbolic representation targeting only piecewise linear NNs is the one presented in [34].

7 Conclusions

In this work, we examined a series of formal equivalence checking problems for NNs, with respect to equivalence relations that can be suitable for various applications and verification requirements. We provided an SMT-based verification approach, as well as a prototype implementation and experimental results.

In our future research plans, we aim to explore whether the equivalence checking problem can be encoded in existing verification tools for NNs (e.g. Reluplex [18], ERAN [32], $\alpha\!-\!\beta$ Crown [36,41,42], VNN competition [4]) through the parallel composition of the two networks that are to be compared. An interesting prospect is to extend our approach towards finding the smallest ϵ, for which two NNs become equivalent.

As additional research priorities, we also intend to explore the scalability margins of alternative solution encodings, including the optimized version of the current encoding (as described in Sect. 4.3) and to compare them with the MILP encoding in [19]. Lastly, it may be also worth exploring the effectiveness of techniques applied to similar problems from other fields, like for example the equivalence checking of digital circuits [12,13,25]. In this context, we may need to rely on novel ideas toward the layer-by-layer checking of equivalence between two NNs.

Acknowledgment. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956123.

References

1. Albarghouthi, A.: Introduction to neural network verification. Found. Trends® Program. Lang. **7**(1–2), 1–157 (2021)
2. Amir, G., Wu, H., Barrett, C., Katz, G.: An SMT-based approach for verifying binarized neural networks. In: TACAS 2021. LNCS, vol. 12652, pp. 203–222. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_11
3. Ashok, P., Hashemi, V., Křetínský, J., Mohr, S.: DeepAbstract: neural network abstraction for accelerating verification. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 92–107. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_5
4. Bak, S., Liu, C., Johnson, T.: The second international verification of neural networks competition (VNN-COMP 2021): summary and results (2021)
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
6. Borchani, H., Varando, G., Bielza, C., Larrañaga, P.: A survey on multi-output regression. Wiley Int. Rev. Data Min. Knowl. Disc. **5**(5), 216–233 (2015)
7. Cerri, R., Barros, R.C., de Carvalho, A.C.P.L.F.: Hierarchical multi-label classification using local neural networks. J. Comput. Syst. Sci. **80**(1), 39–56 (2014)
8. Cheng, Y., Wang, D., Zhou, P., Zhang, T.: A survey of model compression and acceleration for deep neural networks. arXiv preprint arXiv:1710.09282 (2017)
9. Christakis, M., et al.: Automated safety verification of programs invoking neural networks. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 201–224. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_9

10. Collobert, R.: Large scale machine learning. PhD thesis, Université Paris VI (2004)
11. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L.: Imagenet: a large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 248–255 (2009)
12. Disch, S., Scholl, C.: Combinational equivalence checking using incremental sat solving, output ordering, and resets. In: 2007 Asia and South Pacific Design Automation Conference, pp. 938–943 (2007)
13. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001, pp. 114–121 (2001)
14. Goodfellow, I., et al.: Generative adversarial nets. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 27. Curran Associates Inc., Red Hook (2014)
15. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. In: NIPS Deep Learning and Representation Learning Workshop (2015)
16. Huang, X., et al.: A survey of safety and trustworthiness of deep neural networks: verification, testing, adversarial attack and defence, and interpretability. Comput. Sci. Rev. **37**, 100270 (2020)
17. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 29. Curran Associates Inc., Red Hook (2016)
18. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
19. Kleine Büning, M., Kern, P., Sinz, C.: Verifying equivalence properties of neural networks with ReLU activation functions. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 868–884. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_50
20. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View, 1st edn. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-74105-3
21. Kukacka, J., Golkov, V., Cremers, D.: Regularization for deep learning: a taxonomy. arXiv, abs/1710.10686 (2017)
22. Leofante, F., Narodytska, N., Pulina, L., Tacchella, A.: Automated verification of neural networks: advances, challenges and perspectives (2018)
23. Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. Found. Trends® Optim. **4**(3–4), 244–404 (2021)
24. Martins, A., Astudillo, R.: From softmax to sparsemax: a sparse model of attention and multi-label classification. In: International Conference on Machine Learning, pp. 1614–1623. PMLR (2016)
25. Mishchenko, A., Chatterjee, S., Brayton, R., Een, N.: Improvements to combinational equivalence checking. In: Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2006, pp. 836–843. Association for Computing Machinery, New York (2006)
26. Mishra, R., Gupta, H.P., Dutta, T.: A survey on deep neural network compression: challenges, overview, and solutions. arXiv preprint arXiv:2010.03954 (2020)
27. Molitor, P., Mohnke, J., Becker, B., Scholl, C.: Equivalence Checking of Digital Circuits. Springer, New York (2004). https://doi.org/10.1007/b105298
28. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence (2018)

29. Neill, J.O.: An overview of neural network compression. arXiv preprint arXiv:2006.03669 (2020)
30. Paulsen, B., Wang, J., Wang, C.: Reludiff: differential verification of deep neural networks. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 714–726 (2020)
31. Rülling, W.: Formal verification. In: Jansen, D. (ed.) The Electronic Design Automation Handbook, pp. 329–338. Springer, Boston (2003). https://doi.org/10.1007/978-0-387-73543-6_14
32. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 1–30 (2019)
33. Somenzi, F., Kuehlmann, A.: Equivalence checking. In: Lavagno, L., Martin, G.E., Scheffer, L.K., Markov, I.L. (eds.) Electronic Design Automation For Integrated Circuits Handbook, vol. 2. CRC Press, Boca Raton (2016)
34. Sotoudeh, M., Thakur, A.V.: A symbolic neural network representation and its application to understanding, verifying, and patching networks. CoRR, abs/1908.06223 (2019)
35. Teuber, S., Büning, M.K., Kern, P., Sinz, C.: Geometric path enumeration for equivalence verification of neural networks. In: 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI), pp. 200–208. IEEE (2021)
36. Wang, S., et al.: Beta-CROWN: efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. In: Advances in Neural Information Processing Systems, vol. 34 (2021)
37. Wehrmann, J., Cerri, R., Barros, R.: Hierarchical multi-label classification networks. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 5075–5084. PMLR (2018)
38. Wen, H.P.C., Wang, L.C., Cheng, K.T.T.: Functional verification. In: Wang, L.-T., Chang, Y.-W., Cheng, K.-T.T. (eds.) Electronic Design Automation, pp. 513–573. Morgan Kaufmann, Boston (2009)
39. Xiang, W., et al.: Verification for machine learning, autonomy, and neural networks survey. arXiv preprint arXiv:1810.01989 (2018)
40. Xu, D., Shi, Y., Tsang, I.W., Ong, Y.-S., Gong, C., Shen, X.: Survey on multi-output learning. IEEE Trans. Neural Netw. Learn. Syst. **31**(7), 2409–2429 (2019)
41. Xu, K., et al.: Fast and complete: enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In: International Conference on Learning Representations (2021)
42. Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems, vol. 31 (2018)