NASA/TM–2017–219628

# Challenges in the Verification of Reinforcement Learning Algorithms

*Perry van Wesel*
*Eindhoven University of Technology, Eindhoven, The Netherlands*
*Alwyn E. Goodloe*
*NASA Langely Research Center, Hampton, Virginia*

June 2017

NASA/TM–2017–219628

# Challenges in the Verification of Reinforcement Learning Algorithms

*Perry van Wesel*
*Eindhoven University of Technology, Eindhoven, The Netherlands*
*Alwyn E. Goodloe*
*NASA Langely Research Center, Hampton, Virginia*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

June 2017

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA  23681-2199
Fax: 757-864-6500

## Abstract

Machine learning (ML) is increasingly being applied to a wide array of domains from search engines to autonomous vehicles. These algorithms, however, are notoriously complex and hard to verify. This work looks at the assumptions underlying machine learning algorithms as well as some of the challenges in trying to verify ML algorithms. Furthermore, we focus on the specific challenges of verifying reinforcement learning algorithms. These are highlighted using a specific example. Ultimately, we do not offer a solution to the complex problem of ML verification, but point out possible approaches for verification and interesting research opportunities.

## Contents

## List of Figures

## 1 Introduction

*Machine learning* (ML) is a general term that encompasses a broad class of algorithms that learn from data. This is useful in applications where the algorithm is required to adapt to new situations while in operation, or when it would be too complicated to write the functionality expressed by the algorithm by hand. With the rise of big data due to better hardware and the internet, machine learning algorithms that can use this data have also become more popular. They can be found in a wide range of applications from spam filters to stock trading to computer vision, and the field is still growing. In the future machine learning is predicted to guide us to a world of autonomy with ML-controlled cyber-physical systems that will be able to replace drivers, pilots and more.

Employing ML in safety-critical systems [18, 26] that possess the potential to endanger human life or the environment is quite a challenge. In highly regulated safety-critical domains such as civil aviation, engineers usually adhere to guidelines such as ARP 4761 [34] that prescribe processes that focus on ensuring the resulting system is safe. Conservative design and engineering practices are also used to ensure that the system behaves in a predictable manner thus easing the task of verification and validation. In addition, safety-critical systems are often restricted to operate in a predictable environment. Yet predictable operating environments are not always possible and unforeseen environmental influences can arise that overwhelm the capabilities of the automation. Consequently, safety-critical systems often depend on trained human operators to deal with such situations. If an algorithm is replacing the human operator, it should be able to make decisions *at least as safe* as a human would, even in unpredictable circumstances.

The core problem of software verification is to verify that a given system satisfies its specification. Conventional verification-based techniques are based on extensive testing. The weakness of solely basing verification on testing is that "program testing can be used to show the presence of bugs, but never to show their absence" [10]. Increasingly, verification techniques based on formal methods are gaining acceptance in industry. This report will highlight the challenges to formally verifying ML based systems.

Existing research into methods for the assurance of ML based systems roughly fall into two categories. The first approach is to establish rules for safe operation and to build monitors that ensure these rules are not violated. For instance, an autonomous automobile may have to obey a rule saying that it stay at least 10 meters from the vehicle in front of them. Civil aircraft must maintain a vertical separation of 1000 feet and horizontal separation of 5 nautical miles. It is not always possible to characterize such safe operational criteria and consequently, the second approach to safety requires the verification that the ML system itself satisfies some safety criteria. For instance, humans must often obey rules that are intended to ensure safety such as "do not behave erratically" or "do not take actions that are unexpected by other drivers". One difficult challenge is how to give formal expression of these safety properties for machine learning algorithms.

This report is organized as follows. Section 2 is a high-level overview of the field of machine learning and relevant definitions. Section 3 surveys the major classes of ML algorithms and techniques. Section 4 looks at the assumptions involved in the engineering of cyber-physical systems using machine learning. Verification applied to general machine learning algorithms is discussed in Section 5. Following this, Section 6 is a more in-depth look at reinforcement learning and applying verification to this more specific branch of machine learning. Finally, verification of an example system is suggested to give concrete examples of verification possibilities. Section 7 discusses related work and Section 8 concludes.

# 2 Fundamental Concepts in Machine Learning

Machine learning denotes the capability of machines to learn programs from data without being explicitly programmed by a human. Computing scientists working in ML develop general purpose algorithms that given data from a particular problem domain will create a new program to solve that specific problem. As an ML algorithm gets more data, it learns more about what it is supposed to do and refines the program accordingly. An implementation of a machine learning algorithm is referred to as a *learner* or *agent*. One can think of a learner as a higher-order function that takes in data and returns a new program based on the data it has seen. In general, all agents must establish and maintain a *representation*, which is the way it stores its knowledge. The representation chosen for a particular agent depends upon the learning algorithm employed. Among the commonly used representations are hyperplanes and neural networks.

*Training* is the process of teaching an agent by showing it instances of data. Ideally, the resulting program should be able to *generalize* to give correct results for inputs that were not part of the training data. In *supervised* learning, this training data consists of known input-output pairs where the output is referred to as *label*; in *unsupervised* learning, just input instances are considered; and in *reinforcement* learning, the training data is *feedback* received from the environment. This feedback can come in the form of *rewards*, or their negative counterparts: *penalties*. This feedback is also referred to as *reinfocement* of an earlier decision the agent made.

A ML agent typically has an associated *cost* or *reward* function that quantifies the desirable output of the learner. Sophisticated mathematical *optimization* methods such as gradient descent are often used to guide the learning to yield an optimal outcome. In the case of supervised learning, algorithms often minimize a cost function metric measuring the distance between the result predicted by the training data and the result returned by the ML program when generalizing based on new data. While learning a policy for behavior, *exploitation* is the act of exploiting the best known action according to the currently learned policy to get a high reward. *Exploration* is a purposely sub-optimal choice to try to find a better, but currently unknown policy.

Training generally aims to reach *convergence* of a metric, such as accuracy or a cost function, to a stable value as the algorithm sees more data. If the algorithm converges, that indicates that showing the agent more data does not improve performance any longer.

Tasks where machine learning is commonly used include: *classification* where the goal is to learn how to classify input data as member of a certain class; *regression* where the agent learns a function mapping numeric input to numeric output; reinforcement learning where the goal is to learn a policy that can decide on actions to take in a state.

# 3 Machine Learning Algorithms

Machine learning algorithms can be split into three main categories based on how they learn:

- **Supervised learning.** The training data these algorithms are trained on consists of combinations of input instances and output labels. The algorithm is then trained to generalize a function mapping the inputs to desired outputs. These algorithms are often used for classification [27] or regression.

- **Unsupervised learning.** The algorithm is given input without labels, and its task is to find hidden structure and features in this data. Unsupervised learning is often used for clustering or anomaly detection, or combined with supervised learning into semi-supervised learning. Semi-supervised learning uses the hidden structure in data as input for supervised learning, usually used in classification tasks such as image or speech recognition [40].

- **Reinforcement learning.** A reinforcement learning algorithm usually learns a function mapping states to actions, maximizing a reward function for each state-action pair. This learning is done by trial and error. The algorithm chooses an action to take in a state because it is either the action with the highest expected reward (exploitation) or because it is trying to learn something new (exploration). The result of the chosen action, as measured in the environment, is used as reinforcement and updates the algorithm's expected reward for the action taken [23].

Machine learning distinguishes between **offline** and **online** learning. An algorithm learns offline if it is trained during development. Online learning occurs after deployment of the learner. Combinations are possible where a learner is first trained offline and then continues to learn online. This gives a good starting point while keeping the ability to adapt to the actual deployment environment. Supervised and unsupervised learning are usually trained offline because of the large amount of data needed. Reinforcement learning commonly learns online.

## 3.1 Training Data

What sets ML algorithms apart from classical algorithms is the data. Properties of classical algorithms rely on the algorithm itself: they will not change based on the algorithm's input. Machine learning algorithms, on the other hand, derive some of their properties from their training data. This is especially troublesome when trying to prove guarantees, since the properties of the algorithm can change with each instance of input and the exact future input is unknown.

## 3.2 Training Process

Machine learning is an iterative process. In the case of most ML algorithms, training data is split up into a training set and a test set. Here the iteration begins. First, part of the training set is fed to the algorithm to learn from. Next the algorithm is tested on the test set. Statistics such as accuracy, total reward or total cost of the errors made can serve as a metric for the quality of the algorithm. After testing, the algorithm is fed more data to learn from and is then again evaluated on the test set. This process generally continues until the metrics on the test set converge to a stable values. If the eventual metrics are not satisfactory, the entire process can be repeated with modified data, or a different training/evaluation method to try to optimize these metrics.

## 3.3 Generalization

The goal of training a machine learning algorithm is to have it generalize a hypothesis based on the examples it has seen. Remembering the training data is not hard, but the algorithm should be able to apply learned knowledge to previously unseen data. To do this generalization, data alone is not enough [12]. Without any assumptions outside of the training data it is impossible to train a learner that performs better than random guessing. This is known as the *"no free lunch"* theorem [44] in machine learning. Many common assumptions are discussed in a later section.

## 3.4 Overfitting

Overfitting happens when the training data and assumptions are not enough to reach a generalization. In that case, it could happen that the algorithm performs very well on the training data (because the algorithm remembers it), but poorly once presented with data that was not in the training set. Overfitting can be aggravated by the presence of noise, like the noise present on sensors of a cyber-physical system [12].

# 4 Assumptions

When designing a system or requirements for a system, different assumptions are made either explicitly or implicitly. A violation of these underlying assumptions could potentially invalidate any analysis and verification done. To prevent unexpected violations, extra care has to be taken during the development of the system to explicitly state all assumptions made on the system, so situations in which it might behave unpredictably can be easily recognized. This document breaks assumptions down into four categories.

## 4.1 Operating Environment Assumptions

Assumptions on the operating environment of a system often consider the physical environment of the system. For example the presence and properties of other objects in the system's operating boundaries, or properties or constraints on the physical environment in which the system operates.

## 4.2 Platform Assumptions

Platform assumptions are assumptions about properties and constraints of other components in the system. Engineering with imperfect components has been well-explored in the dependable computing field [26], since any cyber-physical system will have numerous components that can fail in multiple different ways. Assumptions on these components could be regarding the quality (fidelity) of their functioning, i.e. noise and accuracy on sensors, or regarding partial or total breakdown of the component (a failure). Assumptions regarding failures of components and their fidelity are discussed separately in more detail.

### 4.2.1 Failure Assumptions

Failure assumptions consider one or more system components failing. These assumptions usually state the expected time until failure of a component. Ultimately, it might be best to simply assume that system engineers implemented dependability as thoroughly as possible, allowing for simplification of the assumptions on the platform regarding the ML system. For example, if there are redundant sensors on board, it could be assumed the sensor data is reliable from the software's point of view. Dependability is a best effort solution, it is not perfect and faces limitations from budgets in available space, cost or computing power.

### 4.2.2 Fidelity Assumptions

Traditional algorithms are usually engineered separately from their data. By this we mean that although a traditional algorithm will have declarations of data types and may include precondtions and assertions restricting the values that data can range, the algorithm does not change based on data. For machine learning algorithms this is not the case. A ML agent is defined by the data that it was trained on. If these (often very large) data sets are obtained using the sensors of the system, the fidelity of these sensors can have a large impact not only on the output of a single execution, but on the future performance of the ML algorithm. More specifically, noisy input in the training data could affect not just one output instance but all future output produced by the ML algorithm.

## 4.3 Assumptions on Data

The training data of a ML algorithm defines the algorithm's functioning. Therefore reasoning about the data becomes more important and should be done with care. There are some assumptions the developer should be aware of when working with training data [42]:

### 4.3.1 Independent, Identically Distributed Samples

The most common underlying assumption about training data that holds for most ML algorithms is the assumption that data samples are independent and identically distributed (IID). This assumption is however always invalid to some degree since online data is rarely identically distributed to the training data. The situation can be worse for multi-purpose agents trained on a general data set considering a wide variety of situations, but deployed in a very specific situation. Conversely this assumption is also broken when deploying an agent in an environment that is (partially) missing from its training data.

The IID assumption can be formalized in the language of probability theory as follows. The random variables $\{X_1, \ldots, X_n\}$ are independent if the events $A_1, \ldots, A_n$ generated by them are independent

$$P(X_i \in A_i \text{ for each } i \in \{1, \ldots, n\}) = \Pi_{i=n}^{n} P(X_i \in A_i)$$

and the $X_i$ each come from the same distribution.

### 4.3.2 Proximity

Another assumption that often appears in machine learning is the notion that samples with similar inputs also have similar outputs. That means that if the distance (according to a metric of choice) between the inputs of two samples is small, then so is the distance between the outputs. This assumption is not always true. A complete breakdown of this assumption can lead to what is known as 'Anti-Learning' where an agent can learn to systematically perform worse than random guessing on the test data set [28].

A formal definition of proximity can be given as follows. Given inputs $i$ and outputs $o$ and a learner $f$ that takes input and transforms it to output $f(i_n) = o_n$ and assume there is a Boolean function $T$ that returns *true* if its arguments are 'close' according to some definition and false otherwise, then proximity is defied as $T(i_a, i_b) \implies T(f(i_a), f(i_b))$.

### 4.3.3 Smoothness

Smoothness is closely related to the idea of proximity and assumes that the underlying model has smooth transitions from one value to another. Without this assumption interpolation or extrapolation make little sense due to the lack of knowledge on the space between two values. Like proximity this assumption is often, but not always, true. Formally, a smooth function is a function that has derivatives of all orders everywhere in its domain.

### 4.3.4 Adversarial Behavior

Adversarial behavior occurs when someone or something purposely tries to affect the algorithm such that it makes mistakes. Cyber-physical systems are predicted to become increasingly networked as they continue to develop, increasing risk with regard to malign attackers. A disturbing example of adversarial behavior is the ability to analyze image classification neural networks and use their internal structure to create a near-invisible transformation from one image to another that drastically changes the classification [19]. DARPA has also spent time researching safe cyber-physical systems in their HACMS program [8] to try to reduce the vulnerability of cyber-physical systems to outside attacks using formal methods, although not specifically considering machine learning.

## 4.4 Assumptions on the Algorithm

Designing a ML algorithm comes with some assumptions. The fact a machine learning algorithm is used implies an assumption by itself: the desired functionality can be learned from data. Other than that it is assumed that the chosen algorithm and its structure are able to accurately model the system to be learned. The developer might assume the chosen cost or goal function is sufficient to get good behavior. Another assumption might be that the algorithm does not get stuck in a local optimum but reaches an global optimum. These assumptions are usually made based on observations of the system or measured data. Some of these assumptions might not hold in the presence of sufficiently bad data.

## 4.5 Interacting Assumptions

Since all these assumptions consider the same system, they are not independent. Figure 1 illustrates the relationship among the dependencies while Table 1 shows how these assumptions depend on each other, where each row is a set of assumptions depending on assumptions from the columns.

The way the different assumptions are related implies a hierarchy where the assumptions in one layer can depend on assumptions of all of the layers below it. Theoretically this means that a violated environment assumption could cause a cascading violation leading to an assumption violation on the platform, eventually violating an assumption on the algorithm.

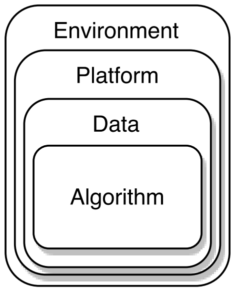

Figure 1. Assumption hierarchy

| | **Environmental assumptions** | **Platform assumptions** | **Data assumptions** | **Algorithm assumptions** |
|---|---|---|---|---|
| **Environmental assumptions** | × | | | |
| **Platform assumptions** | Platform assumptions can be based on assumptions on the environment, e.g. a sensor has a 95% accuracy at temperatures between x and y. | × | | |
| **Data assumptions** | The training data is assumed to adhere to the assumptions and constraints of the environment. | The data might be captured by the platform sensors, thus being affected by any fidelity or failure assumptions. | × | |
| **Algorithm assumptions** | The chosen algorithm is assumed to model the environment of the system well. | The algorithm used assumes the availability and accuracy of inputs. | Properties of the algorithm such as convergence depend on the data it sees. | × |

Table 1. Assumption influence

## 5 Towards Formal Verification of Machine Learning

Formal verification, mathematically proving that a program meets its specification, has been an active subject of research in computer science for over half a century [11, 16, 21]. Most early work focused on manual proofs of functional correctness of algorithms, but sophisticated automation to support program verification is now available. For instance, tools such as Frama-C [7] and SPARK-Ada [2] automatically generate verification conditions from a suitably annotated program and allow the user to complete the proof by applying fully automated proof tools such as CVC4 [3] and Z3 [32] or interactive theorem provers such as Coq [5] and PVS [36]. Model checking [6, 13] is an automated verification technique for determining if a state machine model of the software satisfies a logical specification and if not provides a counterexample. Although it has found many industrial uses, model checking is limited to checking partial specifications, such as the unreachability of bad states, via state space exploration. All conventional verification approaches assume that the algorithm being verified is fixed, but ML programs change as the system learns. When the learning can be done up front, say during development, and then frozen before deployment, conventional approaches are applicable. Also, traditional verification approaches can be applied to proofs of convergence of ML algorithms.

McIver and Morgan [31] have developed a refinement calculus for reasoning about probabilistic programs in the style of Dijkstra and has successfully applied it to reasoning about a range of programs such as cryptographic protocols and wireless protocols, but it has not been applied to reasoning about ML algorithms. Probabilistic programming is an approach to programming systems that help make decisions in the face of uncertainty. Given centrality of probabilistic reasoning to ML, probabilistic programming should be a natural medium for programming ML programs. Specialized probabilistic programming languages are being developed along with associated probabilistic semantics. There has been very little research on the verification of probabilistic program, but the work of Rand and Zdancewic [38] on a logic for probabilistic programs shows promise, but has not been applied to ML programs.

Probabilistic model checking is a recent approach to model checking targeting the modeling and analysis of systems that exhibit probabilistic behavior [14]. Probabilistic model checking has been applied to a probabilistic model learned from data using standard maximum likelihood approaches [17].

In most cases, it is either impractical or impossible to formally verify ML systems using existing tools and techniques. When it is possible to formally specify criteria that would constrain an evolving system, Runtime Verification (RV) [20, 25, 37], where monitors detect and respond to property violations at runtime, may be the only option for ensuring that ML programs do not harm people or behave erratically.

### 5.1 Specifications

The starting point for any formal verification effort is a mathematically rigorous specification precisely stating the property that the software or system is to satisfy. Getting a correct formal specification is often the most difficult aspect of applying formal methods to large systems in industry. Due to their very nature, it is almost impossible to construct a formal specification of ML programs. Liang [30] has argued that the a specification, albeit informal, for an ML system might look like the following:

Input: Training data.
Output: Weight Vector and an estimated accuracy,

but it is unclear how to formalize such a specification.

Rather than specifying an entire ML system, there are critical components of the ML program such as the optimization program that could be formally verified using existing formal verification techniques. While ensuring the correctness of such subtle algorithms is beneficial, it is not sufficient unless there is a way to formally specify properties about the dynamic behavior of the system as it learns and evolves.

## 5.2 Verification of offline learners

After an offline learning algorithm has finished training it will no longer change. From that point onward the algorithm will use the optimal policy it has learned. This means that for equal inputs, the system will give the same outputs, thus acting *deterministically*. Therefore verification in absolute terms should be possible. This was done by Goodrich and Barron Associates for their proposal to replace large lookup tables, used in aircraft fuel measurement systems, with neural network [22]. Thrun's Validity Interval Analysis [41] is an innovative approach to extracting symbolic knowledge from a neural network in a provably correct manner, but little work on this approach has been carried out.

## 5.3 Verification of online learners

Applying traditional static verification methods to online learners makes little sense as the subject system will change as it learns, invalidating all verification done previously. This means that for online learners verification has to be done at runtime.

Online verification is harder for some algorithmic approaches than it is for others. A reinforcement learning algorithm with discrete actions as output can be easily verified at runtime using traditional formal methods, since to the verification there is no difference between a learner or a classical system choosing actions. A supervised learning classifier, on the other hand, is extremely difficult to verify at runtime, since at that time there is usually no base truth available to the verification algorithm, so there is nothing to compare the output to. An alternative would be to monitor for violations of the specified assumptions to ensure the system is operating in the environment considered in its specification and training.

Another issue is that some properties are simply hard to formally define, because it is not exactly known what this property is. Consider a requirement like 'the agent should not behave erratically'. Because of the inherent vagueness this requirement is hard to formalize. Trying to make this requirement more specific might cause other (unwanted) behavior to not be covered by a requirement any longer. Machine learning algorithms are known to find solutions that humans would not consider [1], making it hard to specify non-vague requirements to preclude unexpected behaviors.

Some of the methods for offline verification or analysis seem to have the opportunity to be applied in an online setting, such as identification of input influence proposed by Datta [9], which could lead to more insight in the workings of ML algorithms. Research into increasing the explainability of ML algorithms, which could potentially be used as a form of verification, seems interesting but mostly unexplored. Dwork et al. have suggested a framework for fairness [15] that is closer to traditional runtime verification where a framework is built around the machine learning algorithm to evaluate and possibly correct its output.

Table 2. Components of reinforcement learning algorithms.

| Representation | Evaluation | Optimization technique |
|---|---|---|
| Discrete states & actions | Total/Average reward | Formally sound methods |
| Policies | Speed of convergence | Dynamic programming |
| State value-function | Regret | Gittins allocation indices |
| Q-functions and reward | Accuracy | Learning Automata |
| Generalization | Convergence | Ad-Hoc methods |
| Various function approximation techniques | | Greedy strategy |
| * Decision trees/neural networks/... | | Randomized strategy |
| | | Interval-based techniques |

## 6 Verification of reinforcement learning

For the remainder of this document we will restrict the topic to the verification of reinforcement learning (RL) algorithms [23], as the field of machine learning is too complex to consider generally.

Reinforcement learning seems to be the most transparent and best understood machine learning category and has historically been used most in cyber-physical systems.

### 6.1 Reinforcement learning

As mentioned before, reinforcement learning (RL) algorithms learn through trial and error by interacting with their environment. There is a large number of different reinforcement learning algorithms, but at their core they can be broken down into three main building blocks [23], as shown in Table 2. The representation is the way the algorithm stores its knowledge. This can be in the form of full policies, a value-function for states or Q-functions (the combination of states and actions) and associated rewards. These representations work well for discrete state and action spaces, leading to a small discrete state space. For very large or continuous state spaces these methods would require intractable amounts of training effort, in this case the representation needs to be capable of generalization. This can mean generalizing on input, actions, or both and can usually be done by adapting a reinforcement learning algorithm to include function approximation. The evaluation is used to compare different RL agents or different versions of the same agent during the learning process. Optimization techniques determine how the algorithm learns and how to optimize its representation based on some evaluation metric. The process of optimizing a RL agent is also referred to as exploration. In an online setting pure exploration might not be desired, so a trade-off has to be made between improving the algorithm or exploiting its current knowledge.

### 6.2 Model-based vs. Model-Free Learning

An important distinction to make between RL algorithms is the distinction of model-based learning versus model-free learning. In model-based learning, a model of the environment and system response is available. This means the possible resulting states are known when taking one of the possible actions from any state. This model will often be in the form of a Markov Decision Process (MDP). Such a model consists of transitions from one state to another through actions. Actions can have more than one resulting state associated with it, each resulting state has a probability so that the sum of probabilities of states reachable from an action equals 1.
In Model-free learning, no model of the environment and system response is known. Therefore any action is possible in any state, and any action can lead to any other state. While

training a RL agent, a model of the environment can be learned at the same time. This would mean the model of the environment and system response depends on the training data instead of the other way around, but it does gives the opportunity to construct a reasonably accurate model (given enough training) to use in verification.

## 6.3 Predictability/Exploration Conflict

An issue inherent to reinforcement learning is trading off safety versus exploration. To find an optimal policy, an agent will often use exploration methods that sometimes choose random actions: the 'trials' from the trial-and-error approach. From a safety standpoint a system should act in a more or less predictable, non-erratic way. Limiting exploration to prevent this unwanted behavior would make the system safer, but could limit the overall learning performance severely.

## 6.4 What can be Verified

Looking at the verification of reinforcement learning it is useful to catalog the aspects of these algorithms that could be verified. A distinction is made between aspects of the system that can be verified offline and those that have to be verified at runtime.

### 6.4.1 Offline Verification

- **State to action mappings.**
  In the case where the algorithm will do no further online learning and with a deterministic exploitation strategy, state-action mappings can be extracted from the learner, allowing for verification of requirements in the form "From all states in this set, never do this action".

- **Action sequences.**
  Since the algorithm creates a mapping of states to actions, the only way to verify action sequences is to know the underlying model of the system. Without an environment model any state-action pair could lead to any other state, making all sequences possible. The model is already available in model-based algorithms. In model-free algorithms the model could be learned next to the value function. Assuming the environment model is in the form of a Markov decision process, requirements can be verified in some probabilistic temporal logic specification. An example of a tool that could do this verification is PRISM [29].

- **Algorithm properties independent of data.**
  Proving properties on algorithms is not software verification in the traditional sense of the word, but proofs are required to ensure the algorithm behaves as it should, at least theoretically. An example of this could be a proof of convergence that guarantees that the algorithm will converge over time, like Q-learning has. These proofs consider arbitrary data so no concrete data is needed for the proof.

- **Validating assumptions on training data.**
  Validating assumptions on training data is easier to do offline than it is online because the entire data set is available. Regardless, some properties and assumptions are simply difficult to verify. Independence, identical distribution and proximity are all difficult to verify statistically. Other properties like skewness and variable ranges are easier to verify.

### 6.4.2 Runtime Verification

Many properties that can not be verified offline can be verified at runtime, although this might not always be feasible with regards to computation time or resource efficiency. Instead of verifying the entire specification, only the affected parts can be verified at runtime, assuming that if the specification was verified at the start of learning, and each change is deemed valid, then the specification is still valid after an arbitrary number of changes.

- **State to action mappings.**
  Assuming this is verified often, only one or a small amount of states will have changed, which can be verified the same way as for the state-action mappings verified offline.

- **Action sequences.**
  This can be considered as the runtime verification of a temporal logic specification without any knowledge on the underlying system, just observing the output actions of the agent.

- **Monitoring (violations of) assumptions.**
  Assumptions made on the environment or the platform can easily be checked at a distinct moment in time, like verifying assumptions on one single instance of training data. Other assumptions might be harder or even impossible to verify.

- **Input instance in training data.**
  If an input instance can be shown not to be present in the training data, that is a good indication that the algorithm is encountering unknown situations, which generally should be avoided. The question how to check if an instance of data was present in the input data is tricky however. This problem is related to anomaly detection in data mining. This method is ill-suited for online learners since training data is not well defined.

## 6.5 Example System

As an example of a more exact and robust specification of assumptions and requirements regarding machine learning, a toy example is considered. The example agent is the software of an automatic gearbox trained with model-free reinforcement learning. The controller uses the vector $[\lfloor speed \rfloor, \lfloor \frac{RPM}{500} \rfloor, gear]$ as its state. The actions available to the controller are shifting up, shifting down or staying in the current gear.

The algorithm used to implement this learner is basic Q-learning [43] with an $\epsilon$-greedy exploration strategy. It chooses the best known action with probability 1-$\epsilon$, and a random action with probability $\epsilon$. The algorithm receives a big reward when it reaches a speed of 30 $m/s$. This reward is divided by the time it took to get there, teaching the algorithm to accelerate to 30 as fast as possible. Small rewards are given based on the current speed during the training to guide the learner in the right direction. If after 40 seconds the car is still not going 30 a big penalty is given. If the gearbox performs an action that would break the gearbox, like shifting down at high rpm, a high penalty is given too.

### 6.5.1 Assumptions

The assumptions placed on the system are explicitly listed. Each assumption is categorized to be AE (Environment Assumption), AP (Platform Assumption), AA (Algorithm Assumption) or AD (Data Assumption).
**AE1:** The gearbox is not used on slopes with a gradient of more than 20%.

**AE2:** The controller is only used when the car moves forward, so speed $\geqslant 0$.

---

**AP1:** The engine and gearbox are working as expected.
**AP2:** The engine can handle RPM between 1000 and 6000.
**AP3:** The sensor readings have a $\pm 5\%$ accuracy.
**AP4:** The sensors are assumed not to fail (from a software perspective).

---

**AA1:** Since the algorithm has a formal convergence guarantee, it is assumed to converge.

---

**AD1:** The data is subject to the proximity assumption.
**AD2:** The data is subject to the smoothness assumption.
**AD3:** No adversarial forces are working against the algorithm.

### 6.5.2 Safety Requirements

**R1:** The gearbox controller does not shift down in first gear.
**R2:** The gearbox controller does not shift up in fifth gear.
**R3:** The gearbox controller does not cause the engine to go over 6000 RPM.
**R4:** The gearbox controller does not cause the engine to go under 1000 RPM in any gear but first.
**R5:** The controller should not shift up or down immediately twice in a row.

**R6:** The gearbox controller should not shift too quickly.

**R7:** Convergence should be guaranteed.
**R8:** The rate of convergence should be bounded.

## 6.6 Proposed Verification Approach

Verification methods are proposed for the example system with the goal to verify all safety requirements. These verification methods are again split up into offline and online verification.

### 6.6.1 Offline Verification

#### R7 & R8 - Algorithm Properties

Properties like convergence rely on the algorithm used. General proofs of convergence are possible on these algorithms, even though these guarantees might not work out in practice [12] (which is why there is an assumption stating that it is assumed they will). These proofs do not consider any data so they can be done offline.

### 6.6.2 Online Verification

#### R1 & R2 - State-Action Mappings

Inspect all possible actions from the current state or all states changed because of an update. Considering deterministic action selection, verify the agent will not perform an illegal action in the updated state. In the example, if the gear in the state is 1, verify no *ShiftDown* actions will be performed. If the gear in the considered state is 5, verify no *ShiftUp* action will be chosen.

The drawback of this method as described is that it only works with deterministic action selection. Eliminating random action choices limits the exploration of the learner. Alternatively, actions chosen by the controller can be intercepted by the verification system. Verifying if the action is illegal in the current state and forcing the algorithm to force a new action allows the use of any action selection method, but greatly increases the coupling between the subject system and the verification. This method results in a safer system since it allows for active prevention (and not just detection) of illegal actions while minimizing the restrictions on the exploration strategy.

### R3 & R4 - Probabilistic Action Sequences

To verify an action based on its expected result, that result should be predictable before the action is executed. This is not necessarily possible, as the results of actions on the environment could be uncertain or even unknown depending on the presence of a (probabilistic) model of the world. In the complete absence of a world model, one can be learned while the algorithm is trained. Using this learned world model for verification relies heavily on the assumption that the training environment is representative of the deployment environment. Once a world model is found it can be used to verify slightly modified, probabilistic versions of these requirements, i.e. the probability the gearbox causes the engine to go over 6000 RPM is $\leqslant 10^{-5}$.

### R5 & R6 - Illegal Sequences

Requirements *R5* and *R6* can be rephrased more formally as illegal action sequences. This would cause *R5* to be become 'The action sequences $Up \rightarrow Up$, $Down \rightarrow Down$, $Up \rightarrow Down$ and $Down \rightarrow Up$ are illegal.'
Requirement *R6* can be rewritten as 'Action sequences $\{Up, Down\} \rightarrow None^k \rightarrow \{Up, Down\}$ are illegal for any $0 \leqslant k < \eta$ where $\eta$ is the minimum number of $None$-actions between two shift actions.
These requirements can be specified in temporal logic and verified at runtime. Similar to the verification of state-action mappings, the system can be made safer if the verification algorithm is allowed to actively prevent certain actions from happening.

### Validity of Environmental Assumptions

Environmental assumptions will often be easy to verify since usually they can easily be measured. The considered assumptions *AE1* and *AE2* can simply be measured with an accelerometer and speed sensor respectively. Since environmental assumptions form the first layer of the assumption hierarchy, violation here can be important to catch as they can affect a large part of the system. Violations of these assumptions can be an early indication of unpredictable or unknown situations.

### Validity of Platform Assumptions

Platform assumptions are generally hard to verify due to the difficulty in measuring them. However, most of these assumptions are well-explored in the field of dependable and fault-tolerant systems engineering. This makes it tempting to replace platform assumptions with the assumption the system was engineered to be fault-tolerant. Due to limitations regarding for example cost, space, or power drain, fault-tolerance might not be an option.

### Validity of Algorithm Assumptions

Assumptions on the learning algorithm could be verifiable, although this is a difficult subject. For example, there is no straightforward way to measure convergence, and methods such as measuring the total reward or total penalty over the last $n$ runs can only give an estimate. The downside to this estimate is that it might classify a change in reward related to a change in the environment as diverging behavior, while in actuality it is just the algorithm adapting to a new situation. Verification methods for algorithm assumptions depend on the individual assumption, and for many assumptions it is not clear how to verify them.

### Validity of Data Assumptions

Verifying some of the assumed statistical properties online is not trivial (such as verifying independence), and the results will be running estimates as future data can not be predicted and included. These properties are affected by sensor noise and could even change over time. Although data assumptions are fundamental to machine learning, verifying them might only be feasible in an offline setting.

Properties on individual input instances are easier to verify. Input instances can be verified to adhere to certain assumed bounds on their variables. They can also be checked for anomalies against a predefined data set defining the variable space in which the algorithm is well-defined. Failure to pass these tests could indicate situations unknown to the agent, causing possibly unwanted or unpredictable behavior.

## 7 Related work

Recently there has been more interest in the topic of AI safety with events such as the Workshop on Safety and Control for Artificial Intelligence (SafArtInt 2016) [35] organized by the Office of Science and Technology Policy (OSTP) and Carnegie Mellon University. This event shows there is a real concern towards safety, but it also considered a fairly flexible definition of safety. The talk "On the Elusiveness of a Specification for AI" [30] from the "Algorithms Among Us" symposium early 2016 is an interesting presentation that considers a more formal approach to safety and in particular specification of machine learning algorithms. This talk also displays the tendency of the AI community to solve problems using AI. This method can improve safety in AI systems, but it will not be able to give hard guarantees. If AI verifies AI, at no point in this process is a specification verified although one could argue that confidence in the safety is increased.

There is little known work on actual formal methods for machine learning algorithms. An interesting example of a method to verify some specification on neural networks is from 1993 by Thrun [41]. This method propagates bounds on input variables through the network to produce a bound on the output variable(s). This highlights an interesting way of looking at neural networks, and also shows how verification methods seem limited to one or just a few ML algorithms. A more recent paper [17] presents a method to include Probabilistic Computation Tree Logic (PCTL) constraints into the learning process for ML algorithms that can be represented as Markov Chains. The method proposed in this paper looks promising, although there is still some lack of clarity with regard to the relation between their modified data sets and real world data, and how that impacts performance and safety. Very recent work by Katz et al. [24] suggests a method to extend SMT solvers, allowing for the verification of constraints on deep neural networks. These constraints are a combination of bounds on input variables in combination with bounds on output variables. This is a huge step in the right direction, introducing a scalable method to verify constraints on large neural networks.

Another approach to safety is increasing the transparency of machine learning algorithms: their ability to explain decisions. If some of these black-box-like algorithms can produce an explanation of their decisions, that could lead to interesting properties to monitor on these algorithms. Examples of this is the work done by Datta et al. [9] and Dwork et al. [15] on the fairness of classifiers, as well as similar work done by Ribeiro et al. [39] on a method of explaining the decisions by showing the parts of the input that contributed most to the final decision. Possible runtime verification could involve looking at these explanations.

Lastly the connection between the problem considered in this paper and the field of cyber-physical systems should be mentioned. Since most safety-critical machine learning applications will be cyber-physical systems, it is interesting to see how the CPS field specifies requirements in the face of non-deterministic environments. Recent work suggests using models to provide a more formal specification method for requirements [33], as well as suggesting frameworks defining strict verifiable contracts on individual components to make overall verification of a composition of components easier [4].

## 8 Conclusion

There is no one-size-fits-all solution to the verification of machine learning algorithms. Different domains and different algorithms allow for different verification methods. Although not complete, the methods mentioned in this document are a first step towards verification and eventually adoption of machine learning in autonomous cyber-physical systems.

The applications where machine learning is used most in practice are not safety-critical. This seems to have led to a neglect of requirements engineering in the AI community. For machine learning components to be accepted by regulatory agencies this will have to change, and the AI community will have to engage the safety community in research to enable the creation safety cases for ML based safety-critical systems.

This paper has shown a path – at least for reinforcement learning – to specify and verify safety requirements on parts of machine learning algorithms. We believe it is possible to combine knowledge from the domains of formal methods, dependable systems engineering, and artificial intelligence to develop new formal methods that can create more complete specifications and verification of machine learning algorithms. For now this research area seems mostly unexplored and full of opportunities.

## References

1. Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016. 13

2. John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, 2012. 12

3. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 171–177. Springer-Verlag, 2011. 12

4. Peter Battram, Bernhard Kaiser, and Raphael Weber. A modular safety assurance method considering multi-aspect contracts during cyber physical system design. In *REFSQ Workshops*, pages 185–197, 2015. 20

5. Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer-Verlag, 2010. 12

6. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. 12

7. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - a software analysis perspective. In *SEFM*, pages 233–247, 2012. 12

8. DARPA. High Assurance Cyber Military Systems (HACMS). `http://www.darpa.mil/program/high-assurance-cyber-military-systems`, 2012. 9

9. Anupam Datta, Shayak Sen, and Yair Zick. Algorithmic transparency via quantitative input influence. In *Proceedings of 37th IEEE Symposium on Security and Privacy*, 2016. 13, 20

10. Edsger W. Dijkstra. Notes of Structured Programming. Technical Report EWD 249, Technical University of Eindhoven, 1970. 4

11. Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976. 12

12. Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012. 7, 17

13. Vijay D'Silvay, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008. 12

14. David Duvenaud and James Lloyd. Introduction to probabilistic programming, 2013. Available at `http://jamesrobertlloyd.com/talks/prob-prog-intro.pdf`. 12

15. Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. Fairness through awareness. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 214–226. ACM, 2012. 13, 20

16. Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967. 12

17. Shalini Ghosh, Patrick Lincoln, Ashish Tiwari Shalini, Sri, Com, and Xiaojin Zhu. Trusted machine learning for probabilistic models. 2016. 12, 20

18. Nancy G.Leveson. *Engineering a Safer World.* MIT Press, 2011. 4

19. Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014. 9

20. Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010. 12

21. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. 12

22. Jason Hull, David Ward, and Radoslaw R Zakrzewski. Verification and validation of neural networks for safety-critical applications. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No. CH37301)*, volume 6, pages 4789–4794. IEEE, 2002. 13

23. Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. 6, 14

24. Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. *ArXiv e-prints*, 2017. 20

25. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems*, pages 114–122, 1999. 12

26. John Knight. *Fundamentals of Dependable Computing for Software Engineers.* CRC Press, 2012. 4, 8

27. Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. In *Frontiers in Artificial Intelligence and Applications*, volume 160, pages 3–24. IOS Press, 2007. 6

28. Adam Kowalczyk, D Greenawalt, Justin Bedo, Cuong Duong, Garvesh Raskutti, R Thomas, and W Phillips. Validation of anti-learnable signature in classifcation of response to chemoradiotherapy in esophageal adenocarcinoma patients. In *Proc. Intern. Symp. on Optimization and Systems Biology, OSB*, 2007. 9

29. Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002. 15

30. Percy Liang. On the Elusiveness of a Specification for AI. https://www.microsoft.com/en-us/research/video/symposium-algorithms-among-us-percy-liang/, 2016. 12, 20

31. Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004. 12

32. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer-Verlag, 2008. 12

33. Anitha Murugesan, Sanjai Rayadurgam, and Mats Heimdahl. Using models to address challenges in specifying requirements for medical cyber-physical systems. In *Fourth workshop on Medical Cyber-Physical Systems*. Citeseer, 2013. 20

34. The Society of Automotive Engineers. ARP 4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, 1996. 4

35. OSTP and Carnegie Mellon University. The Public Workshop on Safety and Control for Artificial Intelligence. https://www.cmu.edu/safartint/, 2016. 20

36. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *cade92*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, 1992. 12

37. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4), 2013. 12

38. Robert Rand and Steve Zdancewic. Vphl: A verified partial-correctness logic for probabilistic programs. *Electronic Notes in Theoretical Computer Science*, 319:351 – 367, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). 12

39. Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. *arXiv preprint arXiv:1602.04938*, 2016. 20

40. Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. 6

41. Sebastian B Thrun. Extracting provably correct rules from artificial neural networks. In *University of Bonn*, 1993. 13, 20

42. Larry Wasserman. The role of assumptions in machine learning and statistics: Dont drink the koolaid! Technical report, Carnegie Mellon University, 2015. 8

43. Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. 16

44. David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996. 7

| REPORT DOCUMENTATION PAGE | | | | *Form Approved* *OMB No. 0704–0188* |
|---|---|---|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* 01-06-2017 | **2. REPORT TYPE** Technical Memorandum | **3. DATES COVERED** *(From - To)* |
| **4. TITLE AND SUBTITLE** Challenges in the Verification of Reinforcement Learning Algorithms | **5a. CONTRACT NUMBER** | |
| | **5b. GRANT NUMBER** | |
| | **5c. PROGRAM ELEMENT NUMBER** | |
| **6. AUTHOR(S)** Perry van Wesel and Alwyn E. Goodloe | **5d. PROJECT NUMBER** | |
| | **5e. TASK NUMBER** | |
| | **5f. WORK UNIT NUMBER** 154692.02.30.07.01 | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** NASA Langley Research Center Hampton, Virginia 23681-2199 | **8. PERFORMING ORGANIZATION REPORT NUMBER** L–20806 | |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** National Aeronautics and Space Administration Washington, DC 20546-0001 | **10. SPONSOR/MONITOR'S ACRONYM(S)** NASA | |
| | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** NASA/TM–2017–219628 | |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified-Unlimited
Subject Category 64
Availability: NASA CASI (443) 757-5802

**13. SUPPLEMENTARY NOTES**
An electronic version can be found at http://ntrs.nasa.gov.

**14. ABSTRACT**
Machine learning (ML) is increasingly being applied to a wide array of domains from search engines to autonomous vehicles. These algorithms, however, are notoriously complex and hard to verify. This work looks at the assumptions underlying machine learning algorithms as well as some of the challenges in trying to verify ML algorithms. Furthermore, we focus on the specific challenges of verifying reinforcement learning algorithms. These are highlighted using a specific example. Ultimately, we do not offer a solution to the complex problem of ML verification, but point out possible approaches for verification and interesting research opportunities.

**15. SUBJECT TERMS**
Formal Verificaiton, Machine Learning, Safety Critical Systems, Reinforcement Learning

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | STI Help Desk (email: help@sti.nasa.gov) |
| U | U | U | UU | 30 | **19b. TELEPHONE NUMBER** ***(Include area code)*** (443) 757-5802 |