Formal Verification Methods for Solving Spatial Games

Bar-Ilan University

Authors:
Boaz Gurevich, Erel Dekel

Supervisors:
Prof. Hillel Kugler, Avraham Raviv

October 10th, 2024

Abstract

Reinforcement Learning (RL) has emerged as a powerful paradigm in artificial intelligence, demonstrating remarkable capabilities in complex decision-making tasks across various domains. However, the integration of deep neural networks with reinforcement learning (DRL) introduces additional challenges, particularly in terms of convergence and stability. These issues are especially critical in high-dimensional state spaces and continuous action domains, where DRL algorithms often struggle to maintain consistent performance and safety guarantees.

This project aims to address these challenges by investigating the application of formal verification techniques to deep reinforcement learning algorithms. Formal verification, a rigorous mathematical approach traditionally used in software and hardware validation, offers potential solutions to enhance the reliability and stability of DRL systems. Our research focuses on developing and implementing formal verification methods specifically tailored to the unique characteristics of deep neural networks within the RL framework.

Using the Sokoban puzzle game as a testbed, we explore how formal verification can be integrated into the training process of DRL agents. Sokoban, with its complex state space and requirement for long-term planning, serves as an ideal environment to examine the convergence issues inherent in DRL. We aim to develop techniques that can provide formal guarantees on the behavior of DRL policies, ensuring safety constraints are met and convergence to optimal or near-optimal solutions is achieved.

Contents

Chapter 1

Background

In this chapter, we introduce the fundamental concepts that are critical for the reader to understand. We will provide a deep understanding of Neural Networks (NNs), Reinforcement Learning (RL), Q-Learning (QL) and Formal Verification.

1.1 Neural Networks

Neural Networks (NNs) are machine learning models inspired by the brain structure which is used for approximating complex functions. They consist of layers of interconnected neurons (nodes) through which information flows to learn how to map input data to outputs.

1.1.1 Neural Network Structure

A neural network is divided into 3 types of layers:

1. **Input Layer:** Represents the input features of the network.

2. **Hidden Layers:** The layers between the input and the output layers, where most of the computations and transformation of the features happen.

3. **Output Layer:** Represents the output of the network. Usually represents a classification probability vector.

Each pair of adjacent layers is connected by a function that transforms the data. The depth of a neural network is defined by the number of hidden layers it has. A deep neural network refers to one with a large depth.

1.1.2 Forward-Propagation

Forward propagation is the process of computing the output of a neural network given an input. Each layer of the network transforms the data in a specific way. There are many types of layers in neural networks, such as the Softmax layer, Batch Normalization layer, etc. However, the most common layers are linear (fully connected) layers and activation layers. These two layer types are often combined into a single computational unit which can be expressed as the following:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$
$$a_i^{(l)} = \phi_{(l)}(z_i^{(l)})$$

Where the 1st equation is the linear layer, the 2nd is the activation layer, and:

- $a^{(l-1)} \in \mathbb{R}^{m_{l-1}}$: Activation output vector from layer $l-1$. The first layer ($l=1$) is the network input.
- $W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$: Weight matrix for layer l.
- $b^{(l)} \in \mathbb{R}^{m_l}$: Bias vector for layer l.
- $z^{(l)} \in \mathbb{R}^{m_l}$: Linear combination vector for layer l, also called pre-activation.
- $a^{(l)} \in \mathbb{R}^{m_l}$: Activation output vector for layer l.
- $\phi_{(l)} : \mathbb{R} \to \mathbb{R}$: Activation function at layer l, applied to each neuron separately and adds non-linearity to the model, which allows it to approximate more complex behaviors.

Notice, that m_i is the number of neurons in layer i. Also, there can be a variety of activation functions where the most common are:

1. **Sigmoid:** $\phi(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
2. **ReLU (Rectified Linear Unit):** $\phi(z) = max(0, z)$
3. **Tanh:** $\phi(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

1.1.3 Back-Propagation

Loss Function

In machine learning, the loss function is a way to measure the model's performance by quantifying the difference between the predicted output and the true output. The goal of an ML algorithm is the minimize the loss. Here are the most common loss functions:

- **Mean Square Error (MSE):** Used for regression tasks

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

- **Cross-Entropy Loss:** Used for classification tasks

$$\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{k} y_i \log(\hat{y_i})$$

Where y_i is the predicted output and $\hat{y_i}$ is the target output.

Back-Propagation Process and Gradient Descent

The core learning algorithm of neural networks, is used to adjust the network's weights and biases by calculating the gradient of the loss function with respect to each weight and bias, i.e.

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} \quad and \quad \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

And update the parameters using gradient descent or other variants like Adam optimizer, for gradient descent:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

Where η is called the learning rate.

Calculating the Gradients

This process involves the chain rule as the following:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

For the last layer ($l = L$), we can easily calculate

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial z^{(L)}} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z^{(L)}} = \frac{\partial \mathcal{L}}{\partial y} \cdot \phi'_{(L)}(z^{(L)})$$

As $a^{(L)} = y$, meaning the output of the last layer is the network output.

Notice that $(\cdot)$ is indicates an element-wise product.

We can find the following recursive equation to express $\delta^{(l)}$:

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} = (W^{(l+1)})^T \delta^{(l+1)} \cdot \phi'_{(l)}(z_{(l)})$$

Using $\delta^{(L)}$ and the recursive equation, we can calculate all the gradients at each layer backward from the output layer to the input layer and that is why it is called back-propagation. Now, we can define the expression for the gradients as follows:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$$

If the layer is linear, meaning the pre-activation in layer (l) is a linear transformation of the activation output of layer $(l-1)$ therefore:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

1.1.4 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized type of neural network designed for processing structured grid-like data, such as images. CNNs use convolutional layers to extract features from the input data, making them particularly effective for computer vision tasks like image classification, object detection, and facial recognition. CNNs have emerged as a cornerstone of modern deep learning, outperforming classical fully connected neural networks in tasks that require spatial data interpretation.

CNN Structure

A CNN typically consists of several types of layers, each with a specific role in processing and extracting features from the input data. These layers include:

1. **Convolutional Layers:** Responsible for detecting features such as edges, textures, and shapes by applying filters (also known as kernels) to the input data.
2. **Pooling Layers:** Reduce the dimensionality of the feature maps generated by convolutional layers while retaining the most important information.
3. **Fully Connected (FC) Layers:** Similar to standard neural networks, these layers are used at the end of the network to interpret the features and make predictions.

Convolutional Operation

The convolutional operation is the core of a CNN, where filters slide over the input data to extract local features. For a 2D input (such as an image), the operation can be defined as:

$$S_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i+a,j+b} K_{a,b}$$

Where:

- I is the input image or feature map.
- K is the filter (also known as a kernel).
- S_{ij} is the value of the output feature map at position (i, j).
- m and n are the height and width of the filter.

This operation enables the CNN to detect important patterns in small regions of the input data, which are then combined across layers to identify more complex structures. Figure 1.1 illustrates how the convolution operation works on a sample image. On the left, we see the weighted sum of the neighbors used to compute the value of a pixel (red X), and on the right, the original input image is shown.

Figure 1.1: Right: Input image. Left: Convolution operation demonstrating how the value of the red X is calculated using the neighboring pixels.

Padding and Stride

Padding: Padding involves adding extra pixels around the input, typically with zeros, to control the spatial size of the output feature map. Padding helps to preserve edge information during convolution. Common padding methods include:

- **Same Padding:** Ensures that the output feature map has the same spatial dimensions as the input.
- **Valid Padding:** No padding is applied, which reduces the size of the output feature map.

Stride: Stride determines how far the filter moves across the input during convolution. A stride of 1 moves the filter by one pixel, while larger strides result in faster traversal but a smaller output feature map. The output size O from a convolutional layer is calculated as:

$$O = \frac{N - F + 2P}{S} + 1$$

Where N is the input size (height or width), F is the filter size, P is the padding, and S is the stride.

Pooling Layers

Pooling layers reduce the spatial dimensions of the feature maps, making the network more computationally efficient and reducing the risk of overfitting. The most commonly used type of pooling is max pooling, which selects the maximum value from a set of values in a small pooling window, but other types like average pooling also exist. Pooling helps retain dominant features while discarding less important information. The output size of a pooling layer can be computed as:

$$h' = \left\lfloor \frac{h - f}{s} \right\rfloor$$

$$w' = \left\lfloor \frac{w - f}{s} \right\rfloor$$

Where h' and w' are the height and width of the output feature map, h and w are the height and width of the input feature map, f is the pooling window size and s is the stride.

Advantages of CNNs

CNNs offer several advantages over fully connected networks, especially for tasks involving image and spatial data:

- **Sparse Connectivity:** Each neuron in a convolutional layer is connected to only a small region of the input, leading to fewer parameters and lower computational complexity.
- **Weight Sharing:** The same set of weights (filter) is applied across different regions of the input, reducing the number of parameters and allowing the network to detect patterns independent of location.
- **Translation Invariance:** CNNs are highly effective in recognizing objects in different positions within an image, thanks to the localized nature of convolution and pooling.

1.2 Reinforcement Learning

Reinforcement Learning is a type of machine learning where an agent learns to make decisions (actions) based on interactions with the environment. Unlike supervised and unsupervised learning, reinforcement learning does not learn through a given training dataset but through the feedback that the agent gets when interacting with the environment which comes in the form of rewards. Therefore, reinforcement learning is often considered as a separate paradigm of machine learning that focuses on decision-making and sequential actions in an environment.

1.2.1 Markov Decision Process (MDP)

MDPs provide a mathematical description of the environment, defined as the following tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the probability transition function which defines the probability of transitioning to state s_{t+1} taking action a_t from state s_t, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function which gives the expected reward for taking action a_t at state s_t, where the actual reward can be stochastic, meaning that the received reward is drawn from a probability distribution and is not necessarily deterministic, and $\gamma \in [0, 1)$ is the discount factor.

In MDPs $\mathcal{P}(s_{t+1}|s_t, a_t) = \mathcal{P}(s_{t+1}|s_t, a_t, \ldots, s_0, a_0)$ which means that the following state depends only on the current state and action and not on the past states and actions. In reinforcement learning the environment often can be modeled as an MDP.

1.2.2 Key Components of Reinforcement Learning

- **Agent:** Interacts with the environment and decides which action to take in each state.
- **Environment:** Responds to the agent's actions and provides the new states, usually can be modeled by a probability transition function $\mathcal{P}(s'|s, a)$.
- **State $s \in \mathcal{S}$:** The current situation of the environment. The state space, $\mathcal{S}$, is the set of all possible states.

- **Action $a{\in}\mathcal{A}$:** The decision the agent made in a given state. The action space, $\mathcal{A}$, is the set of all possible actions the agent can take, potentially depending on the current state.

- **Policy $\pi(a|s)$:** A mapping from states to probabilities of selecting each possible action. A policy can be deterministic ($\pi(s) = a$) or stochastic ($\pi(a|s)$ gives a probability distribution over actions).

- **Reward $R(s, a)$:** The immediate return value after preforming action a in state s. A function that maps pairs of action and state to a real scalar $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

- **Value Function $V^\pi(s)$:** Function $V : \mathcal{S} \to \mathbb{R}$ estimates the cumulative reward (future reward) starting from state s and following policy π.

- **Q-Function $Q^\pi(s, a)$:** Also know as Action-Value Function. Function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ estimates the cumulative reward starting from state s, taking action a, and following policy π.

1.2.3 Types of Reinforcement Learning

There are 2 primary types of reinforcement learning approaches:

Model-Free: In this approach, the agent does not have a model of the environment but learn only from interactions with the environment. There are 2 main methods of making a model-free RL:

- **Value Based:** The agent learns a value function which estimates the value of the expected reward in a given state and/or taking an action.
- **Policy Based:** The agent learn a policy that maps states to actions without learning a value function.

Model-Based: In this approach, the agent builds a model that mimics the environment and tries to predict the next state given the current state and action. Model-Based are more data efficient but more complex to build.

1.2.4 The Reinforcement Learning Problem

On each step t, the agent observes a state $s_t \in \mathcal{S}$ in the environment, selects an action $a_t \in \mathcal{A}(s_t)$ according to its policy π and receives a reward r_{t+1}, and the environment transition to a new state s_{t+1}. The environment does not have to be deterministic which means the next state can be defined by the probability function $P(s_{t+1}|s_t, a_t)$.

The goal of the agent is to maximize the future reward, also known as return. At time t looking forward T steps in the future, and considering a discount factor $\gamma{\in}[0, 1)$ (to ensure convergence as $T{\to}\infty$) one can represent the return as:

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_{t+T} = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}$$

Notice the earlier description of the Value Function and Q-Function can be expressed as the following:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t|s_t = s] \quad Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|s_t = s, a_t = a]$$

The objective is to find a policy π^* which maximize the expected return form any state s:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}[G_t|s_t = s]$$

1.2.5 Exploration vs. Exploitation

One main challenge in reinforcement learning is the trade-off between exploration and exploitation.

- **Exploration:** The agent tries new actions in order to discover better rewards and avoid getting stuck in sub-optimal policy.
- **Exploitation:** The agent select actions the based on his current knowledge yields the highest reward.

Both of them are crucial for the success of the agent. One strategy of balancing them is called ϵ-greedy which the agent explore each step at a probability of ϵ.

1.3 Q-Learning

Q-learning is a model-free value-based reinforcement learning algorithm that focuses on optimizing the policy by updating the Q-values of the action-value function $Q(s,a)$ to find the maximum total expected future rewards at any state and action. Since Q-learning is not a policy-based approach, the Q-function does not follow a specified policy, therefore instead of the notation $Q^{\pi}(s,a)$ the notation $Q(s,a)$ is in place.

1.3.1 The Bellman Equation

To find the optimal Q-function, first, we derive the Bellman equation:

$$\begin{aligned}
Q(s,a) &= \mathbb{E}[G_t|s_t = s, a_t = a] \\
&= \mathbb{E}[r_{t+1} + \gamma G_{t+1}|s_t = s, a_t = a] \\
&= \mathbb{E}[r_{t+1} + \gamma \mathbb{E}[G_{t+1}|s_{t+1} = s', a_{t+1} = a']|s_t = s, a_t = a] \\
&= \mathbb{E}[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s', a')|s_t = s, a_t = a] \\
&= \sum_{r,s'} p(r, s'|s, a)(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'))
\end{aligned}$$

The Bellman equation defines the optimal Q-value recursively, and Q-learning is a way to approximate this optimal Q-function through value iteration iteratively.

1.3.2 The Q-Learning Algorithm

Q-Learning algorithm updates the Q-function values according to the following formula derived from the Bellman equation:

$$Q(s,a) \leftarrow r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$$

There can also be added a learning rate factor $\alpha \in [0,1]$ which determines how fast the new information override the old one, so the new update rule for the Q-value is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s,a))$$

At each time step of the algorithm the agent:

1. **Observe** the current state s of the environment.
2. **Choose** an action a according to the exploration strategy used.
3. **Move** to the next state s_{t+1} of the environment and receives a reward r_{t+1}.
4. **Update** the new Q-value for the pair (s,a) according to its update rule.

One way to define the Q-function in the Q-learning algorithm is to use a Q-table to define the Q-value for each pair of (s,a), this way has advantages like time complexity, constant time for updating the Q-value, but also disadvantages like scalability, for large and complex environments the Q-table takes too much memory as there are a lot of states and actions pairs.

Convergence

The Q-learning algorithm using Q-table is guaranteed to converge to the optimal Q-function $Q^*(s,a)$ under certain conditions:

- The learning rate α decays appropriately over time.
- The agent visits every state-action pair infinitely often.

1.3.3 Deep Q-Learning

Deep Q-Learning (DQN) extends the standard Q-Learning algorithm by using a deep neural network to approximate the Q-value function $Q(s,a)$, enabling it to scale to large or continuous state spaces. This is particularly useful in environments where storing a Q-table is impractical due to the size or complexity of the state space.

Motivation for Deep Q-Learning

In Q-Learning, the Q-value of each state-action pair is stored in a Q-table and updated according to the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s,a))$$

However, for environments with large or continuous state spaces, this Q-table becomes infeasible. To address this, Deep Q-Learning uses a neural network to approximate the Q-value function, which is parameterized by weights θ. Instead of storing exact Q-values for every state-action pair, we approximate $Q(s, a)$ as $Q(s, a; \theta)$, where θ represents the parameters (weights) of the neural network.

The Q-value update now becomes:

$$Q(s, a; \theta) \leftarrow Q(s, a; \theta) + \alpha(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \theta') - Q(s, a; \theta))$$

Here, θ' represents the parameters of the target network, a stable copy of the Q-network that is periodically updated.

Training Deep Q-Network

In DQN, the neural network takes the current state s as input and outputs Q-values $Q(s, a; \theta)$ for all possible actions a. This network is trained to minimize the error between the predicted Q-values and the target Q-values based on the Bellman equation.

The MSE loss function used to train the network is:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim D}(y_t - Q(s, a; \theta))^2$$

Where $y_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta')$ is the target Q-value, D is the experience replay buffer, θ are the parameters of the current Q-network, and θ' are the parameters of the target Q-network.

This loss function is minimized using gradient descent to update the parameters θ.

Stabilizing Deep Q-Learning

Deep Q-Learning can be unstable due to the use of non-linear function approximators. Several techniques have been developed to stabilize the learning process:

- **Experience Replay Buffer:** Stores and randomly samples past experiences $(s_t, a_t, r_{t+1}, s_{t+1})$ to break correlations between consecutive samples and improve data efficiency.
- **Target Network:** A separate network with parameters θ' used to compute target Q-values, updated less frequently than the main network to reduce correlation between target and estimated Q-values.
- **Prioritized Experience Replay:** An enhancement to the standard replay buffer that prioritizes experiences with higher temporal difference errors, focusing learning on the most informative transitions.

These techniques help to mitigate instability issues and improve the convergence of the Deep Q-Learning algorithm.

Advantages and Challenges of Deep Q-Learning

Below is a table comparing traditional Q-Learning with Q-tables and Deep Q-Learning 1.1:

Aspect	Q-Learning	Deep Q-Learning (DQN)
State Space	Suitable for small, discrete state spaces	Handles large, high-dimensional, or continuous state spaces
Generalization	No generalization; requires explicit updates for each state-action pair	Neural network generalizes across similar states
Memory Usage	High memory usage for large state spaces	Memory-efficient, using fixed-size neural network parameters
Learning from Experience	Updates are immediate and local	Uses experience replay to learn from past experiences
Stability	Generally stable, but can be slow to converge in complex environments	Can be unstable, but techniques like target networks improve stability
Scalability	Limited scalability to complex problems	Scales well to complex, high-dimensional problems
Computational Complexity	Low for small state spaces, but increases rapidly with state space size	Higher computational requirements, but more efficient for large state spaces

Table 1.1: Comparison of Q-Learning and Deep Q-Learning

1.4 Formal Verification

Formal verification is the process of mathematically proving that a system adheres to a given specification. This is achieved by constructing a formal model of the system and checking whether the system satisfies a set of desired properties. This method is vital in ensuring the reliability of systems, including cryptographic protocols, hardware circuits, and increasingly, machine learning models like neural networks.

In the following sections, we will discuss the key components of formal verification: the model, the specification, and the properties, in the context of common formal verification methods.

1.4.1 Verification Models for Formal Verification

The verification of a system is done by ensuring the existence of a formal proof of a mathematical model of the system. Verification models provide an abstract but rigorous mathematical framework that allows systems to be analyzed in a precise and unambiguous way.

There are many types of verification models and we provide here a few:

1. **Transition System**

   One of the most commonly used models in formal verification is the transition system, which describes the system's behavior through states and transitions. A transition system is defined as a tuple $\mathcal{T} = (\mathcal{S}, \mathcal{A}, T, \mathcal{L})$, where:

- $\mathcal{S}$ is the set of states,
- $\mathcal{A}$ is the set of actions,
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function, which describes how the system moves from one state to another given an action,
- $\mathcal{L} : \mathcal{S} \to 2^{AP}$ is the labeling function, where AP is a set of atomic propositions. $\mathcal{L}(s)$ assigns a set of atomic propositions that hold true in a state s.

This model is particularly useful for representing finite-state systems, where a system's execution can be seen as a series of state transitions. Transition systems serve as the basis for many verification methods, including model checking.

2. Finite State Machines (FSM)

An FSM is a specialized type of transition system where the number of states is finite. FSMs are widely used to model hardware systems, communication protocols, and certain types of software systems. Each transition in an FSM is determined by inputs, and it results in outputs, making FSMs ideal for analyzing systems with clear input-output behavior.

An FSM can be described by the tuple $FSM = (\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F})$, where:

- $\mathcal{Q}$ is the finite set of states
- Σ is the set of input symbols
- $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the state transition function
- q_0 is the initial state
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states.

3. Petri Nets

Petri nets are a powerful tool for modeling concurrent and distributed systems. They consist of places, transitions, and tokens that flow between places according to transition rules. The formal definition is $PN = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0)$ where:

- $\mathcal{P}$ is the set of places
- $\mathcal{T}$ is a set of transitions
- $\mathcal{F}$ is a set of arcs (connecting places to transitions or vice versa)
- $\mathcal{M}_0$ is the initial marking, which assigns tokens to places

Petri nets are particularly effective for modeling systems with complex synchronization, resource sharing, and concurrency properties.

1.4.2 Specification and Properties

In formal verification, specifications and properties define the desired behavior of a system and guide the process of verifying its correctness. In common formal verification like model checking they are defined using LTL and CTL.

Temporal Logic: LTL and CTL

Linear Temporal Logic (LTL) is used to describe system behavior along a single sequence of states over time. It includes operators like:

- $X\phi$ (next): ϕ holds in next state
- $F\phi$ (eventually): ϕ will hold at some time in the future
- $G\phi$ (globally): ϕ holds at all the future states
- $\phi U \psi$ (until): ϕ will holds until ψ become true

Computation Tree Logic (CTL), on the other hand, expresses system behavior across multiple possible future paths. CTL includes path quantifiers:

- $A\phi$: ϕ holds on all paths
- $E\phi$: ϕ holds on some paths

CTL formulas combine these quantifiers with temporal operators. For example:

$$AG(request \rightarrow EF(respones))$$

ensures that, on all paths, if a request is made, there must exist some path where a response will eventually happen.

Specification

A specification is a formal description of the desired behavior or requirements of a system. It defines what the system must do and outlines the conditions it must satisfy. The specification serves as a contract between the system's design and its expected operation, dictating the criteria for correctness.

The specification is generally expressed as a logical formula, denoted as ϕ. This formula serves as the target that the system model $\mathcal{S}$ must satisfy. The verification process asks the following question:

$$\mathcal{S} \models \phi$$

Here, $\mathcal{S} \models \phi$ means that the system $\mathcal{S}$ satisfies the specification ϕ. If this condition holds, the system is considered correct with respect to the specification. Otherwise, if $\mathcal{S} \not\models \phi$, the system fails to meet the expected behavior.

Properties

Properties are conditions that specify aspects of system behavior that must be met. Key categories of properties include:

1. Safety Properties: These ensure that ”nothing bad happens.” For instance, a safety property might state that two contradictory conditions never hold simultaneously. Formally:

$$G(\neg(condition_1 \wedge condiction_2))$$

2. Liveness Properties: These ensure that ”something good eventually happens.” For example, a liveness property might guarantee that after a request, a response will follow:

$$G(request \rightarrow F(response))$$

3. Fairness Properties: These ensure that all parts of the system get a chance to execute or participate under fair conditions. For example, in a multi-threaded system, fairness may guarantee that each thread eventually gains access to shared resources.

The Relationship Between Specification and Properties

Specifications are typically composed of multiple properties, which collectively describe the desired behavior of the system. These properties define both safety (preventing undesirable states) and liveness (ensuring desired outcomes). Fairness properties are often added to ensure that all parts of the system function equitably, there are of course other types of properties.

1.4.3 Formal Verification of Neural Networks

As neural networks are increasingly deployed in safety-critical domains such as autonomous vehicles, medical diagnostics, and aerospace systems, ensuring their correctness is crucial. Traditional testing methods, such as empirical validation, often fail to provide complete coverage due to the vast and continuous input spaces of neural networks. Formal verification aims to address this by mathematically proving that a neural network satisfies certain properties under all possible input conditions.

Neural Network’s Formal Verification Models

Formal verification of neural networks often revolves around constraint solving. The goal is to encode the behavior of a neural network as a system of mathematical constraints, which are then solved or checked for violations of desired properties. The general problem can be formulated as follows:

- **Neural Network Function:** Given a neural network $f_\theta : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where θ represents the weights and biases of the network and n, m are the sizes of the input and output vectors respectively, we aim to verify whether it satisfies a property ϕ, defined over its input-output behavior.

- **Verification Problem:** We want to check if for all inputs $x \in X \subseteq \mathbb{R}^n$, the output of the network $y = f_\theta(x)$ satisfies a property $\phi(x, y)$ Mathematically, this is expressed as:

$$\forall x \in X : f_\theta(x) \models \phi$$

Where X represents the input domain (e.g., bounded by physical constraints or sensor limits), and ϕ is a logical formula representing the desired property (e.g., safety, robustness, fairness).

Properties and Constraints

Verification properties ϕ are defined as constraints over the input-output behavior of the network. Examples of commonly verified properties include:

1. **Robustness:** Ensuring the network's output does not change significantly in response to small perturbations in the input. For instance, robustness to adversarial examples can be formulated as:

$$\forall x \in X, \|x' - x\| \leq \epsilon : f_\theta(x') = f_\theta(x)$$

This ensures that if the input x is perturbed by ϵ-bounded noise x', the network's output remains unchanged or within acceptable bounds.

2. **Safety:** The network should avoid unsafe regions of the output space. For instance, in an autonomous driving system, the steering angle $f_\theta(x)$ should never exceed certain thresholds for a given sensor input x. This can be expressed as:

$$\forall x \in X, y = f_\theta(x), y_{min} \leq y \leq y_{max}$$

3. **Fairness:** Ensuring that the output does not unfairly discriminate based on sensitive input features (e.g., gender, race).

Challenges in Formal Verification of Neural Networks

The formal verification of neural networks (NNs) presents significant challenges due to the complexity and scale of modern networks. The challenges include:

1. **Scalability and Complexity:** Modern neural networks, particularly deep neural networks (DNNs), have large amount of parameters, making the verification problem highly complex. The sheer size of these networks leads to a combinatorial explosion in the number of possible input-output mappings.

2. **Non-linearity of Activation Functions:** A key source of complexity arises from the non-linear activation functions used in neural networks, making it difficult to verify the output behavior analytically.

3. **Adversarial Robustness:** One of the most critical properties to verify is the robustness of neural networks against adversarial examples. Adversarial examples are slightly perturbed inputs that appear normal to humans but cause the neural network to make

incorrect predictions. Ensuring that a network is robust to these small, imperceptible changes in input is vital, especially in safety-critical domains like autonomous driving or medical diagnosis.

4. **Soundness vs. Completeness Trade-offs:** In many cases, formal verification tools must make trade-offs between soundness (ensuring the correctness of results, minimizing the false positives and negatives) and completeness (ensuring that all possible behaviors are checked).

Common Methods to Overcome Challenges

In order to overcome the challenges of incorporating formal verification into neural networks, there has been developed many verification tools and methods, here are some common methods:

1. **Abstraction and Relaxation:** One way to handle large networks is by breaking them into smaller, more manageable components that can be verified separately. Abstraction techniques can also simplify the network by approximating parts of the model while retaining key properties. Relaxation techniques loosen the constraints, trading precision for scalability.

2. **Mixed Integer Linear Programming (MILP):** For neural networks with piecewise linear activations (like ReLU), the verification problem can often be reduced to a mixed-integer linear programming (MILP) problem which is a variant of (ILP). This allows efficient solvers to handle networks with thousands of neurons.

3. **Symbolic Interval Analysis:** Another technique involves bounding the range of neuron activations symbolically, allowing verification over intervals rather than specific points. This reduces the complexity of the problem, particularly for networks with ReLU activations.

1.4.4 Marabou Verification Tool

Marabou is an open-source tool designed for the formal verification of deep neural networks (DNNs). It builds upon the foundations of the Reluplex solver, which was designed to handle the verification of networks with ReLU activation functions. Marabou aims to provide rigorous mathematical guarantees about the behavior of neural networks.

Key Features of Marabou

1. **Versatility:** Supports various neural network architectures, including feedforward and convolutional neural networks (CNNs).

2. **Multiple Activation Functions:** Handles networks with different activation functions, including ReLU, sigmoid, and tanh.

3. **Flexible Input Formats:** Accepts networks in various formats, including TensorFlow, ONNX, and native Marabou format.

4. **Property Specification:** Allows users to define and verify a wide range of properties, from input-output relationships to robustness against perturbations.

5. **Counterexample Generation:** When a property is violated, Marabou generates a counterexample which is a specific input that causes the network to fail the property.

6. **Parallelization:** Supports parallel execution to leverage multi-core processors and distributed computing.

Marabou Solution for Verification Challenges

Marabou introduces several key techniques to improve the scalability and robustness of formal verification:

1. **Piecewise Linear Constraint Handling:** Marabou specializes in verifying networks that use piecewise linear activations like ReLU. The tool formulates the verification problem as a system of piecewise linear constraints. For example, a ReLU activation is encoded as:
$$ReLU(z) = \begin{cases} z, & if \quad z \geq 0 \\ 0, & if \quad z < 0 \end{cases}$$
This behavior is translated into a set of linear constraints with case distinctions (via binary variables) that the solver can handle efficiently.

2. **Symbolic Bound Propagation (SBP):** To reduce complexity, Marabou performs symbolic bound propagation, which tracks the minimum and maximum possible values for neurons layer-by-layer. This helps prune large, irrelevant parts of the input space, making verification more scalable.

3. **Mixed Integer Linear Programming (MILP):** Marabou uses MILP encoding to represent ReLU activations as binary variables within a linear programming framework. This enables the solver to handle large neural networks efficiently by leveraging advanced MILP solvers, which are optimized for such constraints.

4. **Domain Decomposition and Parallelization:** Marabou handles large input spaces by splitting them into smaller subregions and verifying each part separately. This technique, combined with parallelization, speeds up the verification process, making it more practical for high-dimensional inputs.

Summary of Challenges and Marabou's Contributions

See the following table that summarizes Marabou's solutions to the challenges of formal verification of neural networks 1.2:

Marabou, through its combination of symbolic bound propagation, MILP, and domain decomposition, offers a robust tool for verifying neural networks. However, it is still limited by the fundamental challenges of scalability and non-linearity inherent in large and complex networks, meaning ongoing research and improvements in verification tools are needed to tackle increasingly complex neural architectures.

Challenge	**Marabou's Solution**
Scalability	Symbolic bound propagation, input domain decomposition, and MILP encoding improve the efficiency of solving large verification problems.
Non-linearity of Activations	Marabou handles ReLU activations using efficient piecewise linear constraint solving and MILP encoding, reducing the complexity of verification.
Adversarial Robustness	Marabou checks for robustness by exploring the entire input space and generating counterexamples for adversarial inputs when violations occur.
Soundness vs. Completeness	While Marabou uses over-approximation for efficiency, it ensures soundness in most cases by accurately solving the verification problem for piecewise linear functions.

Table 1.2: Formal Verification of Neural Networks Challenges and Marabou's Solutions

Chapter 2

Implementation

In this chapter, we will explain how our project containing the environment and graphics, reward systems, neural network models, deep reinforcement learning agents, and verification is implemented.

2.1 Sokoban Environment

Sokoban is an ancient Japanese video game, in which a player (the keeper) pushes boxes around a 2-dimensional grid (the warehouse) and its objective is to push them onto the targets (storage locations) 2.1. This game will be the environment the deep reinforcement learning agent will try to solve.

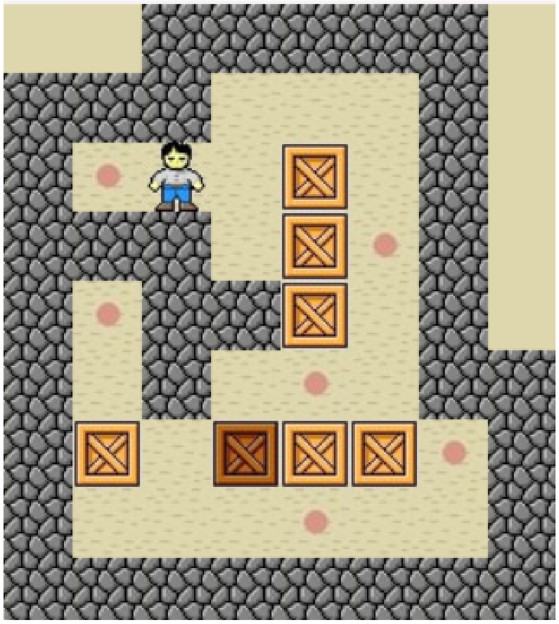


Figure 2.1: Example of a Sokoban game level. The player (blue) must push boxes (light brown) onto target locations (dark squares) while navigating obstacles (gray walls).

2.1.1 Sokoban Game Class

The SokobanGame class implements the core mechanics of the Sokoban environment. It is responsible for loading levels, handling player movement, and managing the transitions of the game state. This class also provides optional features such as randomizing the keeper's position and rendering the game with graphical elements.

Class Initialization

Upon initialization, the SokobanGame class accepts several parameters: the current level to be loaded, a flag to enable or disable graphics, a randomization option for the keeper's starting position, and a seed value to ensure reproducibility when randomization is enabled.

```
def __init__(self, level, graphics_enable=False, random=False, seed=0):
```

In the initialization process, the class sets the initial coordinates of the keeper (x, y), the current level, and an internal representation of the level map stored in map_info. If graphics are enabled, a separate module is used to load a graphical tile map.

Loading Level Data

Each Sokoban level is stored in CSV format in the levels folder, where rows are separated by lines and cells are delimited by commas (,). The load_map_info function reads the level file and parses it into a 2D array representing the game grid.

```
with open(os.path.join(absolute_path, relative_path_level)) as f:
    data = csv.reader(f, delimiter=',')
    for row in data:
        row = [int(item) for item in list(row)]
        self.map_info.append(row)
```

Each cell in the map_info array corresponds to a specific game element:

- **0:** Empty space
- **1:** Wall
- **2:** Floor
- **3:** Target (storage location)
- **4:** Box
- **5:** Box on target
- **6:** Keeper
- **7:** Keeper on target

During level loading, the function also locates the keeper's initial position by scanning the map for tiles labeled as keeper (6) or keeper on target (7).

Randomized Keeper Position

If the class is initialized with the randomization option enabled, the `add_keeper_randomly` function is triggered after level loading. This function selects a random valid position within the map (either an empty floor or target tile) and moves the keeper to that position, while updating the game state to reflect the change.

```
rand_x = random.randint(1, len(self.map_info[0]) - 2)
rand_y = random.randint(1, len(self.map_info) - 2)
if self.map_info[rand_y][rand_x] in (2, 3):
    self.map_info[rand_y][rand_x] += 4
    self.map_info[self.y][self.x] -= 4
```

Randomization helps to introduce variability into the game environment, making it particularly useful for training reinforcement learning agents.

State Representation

The internal representation of the game state is crucial for interfacing with reinforcement learning algorithms. The `process_state` function extracts a 2D array of the game map, excluding the borders, and converts it into a 1D NumPy array, which can be used as input to a neural network.

This transformation ensures that the state information is presented in a format suitable for machine learning models.

Player Movement and State Transitions

Player movement is handled by the `step_action` function, which takes an action (up, right, down, or left) and moves the keeper accordingly. The movement logic ensures that Sokoban rules are enforced, meaning the keeper cannot move through walls, and boxes can only be pushed into empty spaces or targets.

```
if action == 0:  # UP
    self.move((-1, 0))
if action == 1:  # RIGHT
    self.move((0, 1))
if action == 2:  # DOWN
    self.move((1, 0))
if action == 3:  # LEFT
    self.move((0, -1))
```

The `move` function determines the outcome of each move based on the content of the destination tile. It handles interactions between the keeper, boxes, and the game environment, ensuring valid transitions between states.

Game Reset and Level Transitions

The game class supports restarting the current level, progressing to the next level, or moving back to the previous level. These transitions are handled by the `reset_level`, `next_level`, and `prev_level` methods.

```
def reset_level(self):
    self.load_map_info()

def next_level(self):
    self.level += 1
    self.reset_level()

def prev_level(self):
    self.level -= 1
    self.reset_level()
```

Each level is reset by reloading the map data from the corresponding CSV file, reinitializing the keeper's position, and resetting all game elements. This ensures that the game can be restarted under the same conditions or advanced to new levels.

End Condition Detection

The game checks for completion after each player action. The level is considered completed when all boxes have been successfully placed on target tiles. The `check_end` function iterates over the game map to determine if there are any remaining boxes not yet on targets.

```
for row in self.map_info:
    for tile in row:
        if tile in (3, 4):
            end_level = False
```

If all boxes are on targets, the function returns `True`, signaling the end of the level.

Graphical Output

The graphical output in the `SokobanGame` class is implemented using the `pygame` library in conjunction with a custom `TileMap` class. Each tile, such as walls, floors, boxes, and the keeper, is represented by individual images loaded from a spritesheet. The `Tile` class retrieves the necessary tile images, and the `TileMap` class manages their display.

The display is scalable, automatically adjusting to the screen size using `Tkinter` to retrieve screen dimensions. The game map is rendered on a `pygame.Surface`, and the `load_level` function loads a new level, drawing the appropriate tiles on the surface. The user interface is updated after each action, ensuring real-time feedback during gameplay.

2.2 Reward Generation

The reward generation system is a critical component in guiding the learning process of the reinforcement learning agent in the Sokoban game. Since we implemented the game and its graphics ourselves, we also need to create the reward system for the agent. This system comprises two primary reward generators: `Simple` and `HotCold`, both of which derive from the abstract base class `RewardGenerator`.

2.2.1 RewardGenerator Base Class

Purpose

The `RewardGenerator` class serves as an abstract base class for all reward calculation mechanisms in the Sokoban reinforcement learning environment. It provides a foundation for implementing specific reward strategies and maintains essential metrics.

Key Components

- **Loop Counter:** Counts how many times the agent revisits the same state.
- **Accumulated Reward:** Tracks the total reward accumulated over a series of actions.

Key Methods

- `calculate_reward`: An abstract method that must be implemented by subclasses to define how rewards are calculated.
- `reset`: Resets the loop counter and accumulated rewards to initial values.

Decorator: calc_accumulated

This decorator simplifies reward accumulation by automatically summing the rewards after each action:

```
def calc_accumulated(func):
    def inner(self, *args, **kwargs):
        reward = func(self, *args, **kwargs)
        self.accumulated_reward += reward
        return reward
    return inner
```

2.2.2 Simple Reward Generator

Purpose

The `Simple` reward generator provides straightforward feedback based on the agent's actions, incentivizing efficient level completion while penalizing redundant or ineffective moves.

Components

- r_waste: Penalty for redundant moves (when current state equals next state).
- r_done: Reward for completing the level.
- r_move: Standard reward for movement.

Reward Logic

1. If the level is complete (done is True), it receives a significant reward (r_done).
2. If the current state is the same as the next state (redundant move), it incurs a waste penalty (r_waste).
3. If no conditions are met, the agent receives a standard movement reward (r_move).

2.2.3 HotCold Reward Generator

Purpose

The HotCold reward generator provides a more nuanced reward structure that incorporates the spatial relationship between the agent, boxes, and targets. It promotes strategic positioning by rewarding the agent for moving boxes closer to their targets and penalizing unnecessary moves.

Components

- r_hot: Positive reward for beneficial moves (reducing distance to targets).
- r_cold: Negative reward for detrimental moves (increasing distance to targets).
- r_done: Reward for completing the level.

Reward Logic

1. If the level is completed (done is True), return r_done.
2. If the number of boxes on targets decreases, return r_cold.
3. Evaluate the state before and after the move, if evaluation improves (the metric decreases) return r_hot else return r_cold.

State Evaluation Process

1. **Find the closest box:** The algorithm identifies the box that is nearest to the agent (keeper).
2. **Calculate distance to target:** The distance between the closest box to the keeper and its closest target is computed.
3. **Evaluate distance to box:** The distance between the agent and the closest box is evaluated, according to the cells the box can be pushed from

4. **Combine distances:** The two distances are combined into a single evaluation metric, with a higher weight given to the distance between the box and its target.

Pathfinding Algorithm

The pathfinding algorithm used in the state evaluation process is a variation of Breadth-First Search (BFS). BFS is a graph traversal algorithm that systematically explores all vertices at a given depth before moving to the next depth level. In this context, the graph is represented by the game board, where vertices are grid cells and edges connect adjacent cells.

The BFS algorithm starts from the agent's position and explores neighboring cells. It keeps track of the distance from the starting point for each visited cell. If a cell of the desired type (box or target) is encountered, the algorithm returns the distance to that cell along with its coordinates.

The HotCold generator uses two BFS-based functions:

- `path_to_type(state, start_y, start_x, end_type)`: This function finds the shortest path from the given starting position (`start_y`, `start_x`) to a cell of the specified end type (either box or target). It explores the game board while avoiding obstacles and cells that are not relevant to the search.

- `path_to_pos(state, start_y, start_x, end_y, end_x)`: This function finds the shortest path from the starting position to a specific cell with coordinates (`end_y`, `end_x`). It operates similarly to `path_to_type`, but it directly targets a specific cell instead of searching for a cell of a particular type.

2.2.4 Difference Between The Rewards Generators

These reward generators create a dynamic feedback mechanism that encourages the reinforcement learning agent to explore effective strategies for solving the Sokoban puzzle. The `Simple` generator focuses on efficiency and penalization of redundancy, while the `HotCold` generator emphasizes spatial awareness and strategic decision-making. This dual approach aims to enhance the learning experience, ensuring the agent develops an understanding of both short-term actions and long-term goals in the Sokoban environment.

2.3 Neural Networks Models

In order to create a deep reinforcement learning (DRL) agent we implemented 3 basic neural network models. Each model is designed to process the game state and output a probability distribution over possible actions (up, right, down, and left). It's important to mention that since we didn't use GPU our computational resources were limited so we used simple neural networks which are not too deep.

2.3.1 Neural Network Architectures

To create those 3 neural network models we used the torch package and its library of neural networks, also the optimizers used for updating the neural network weights are Adam and RAdam optimizers.

1. **NN1**

   A simple feedforward neural network with one hidden layer. It takes the flattened game state as input, processes it through a linear layer with ReLU activation, and then outputs a probability distribution over actions.

   ```
   model = nn.Sequential(
       nn.Linear(input_size, int(input_size)),
       nn.ReLU(),
       nn.Linear(int(input_size), output_size)
   )
   ```

2. **NN2**

   A slightly more complex feedforward neural network with two hidden layers. It follows a similar structure to NN1 but has an additional hidden layer to potentially capture more complex relationships in the game state.

   ```
   model = nn.Sequential(
       nn.Linear(input_size, 2*input_size),
       nn.ReLU(),
       nn.Linear(2*input_size, input_size),
       nn.ReLU(),
       nn.Linear(input_size, output_size)
   )
   ```

3. **CNN**

   A convolutional neural network (CNN) specifically designed for processing grid-based data like the Sokoban game state. CNNs are well-suited for tasks that involve spatial patterns and relationships. The information is processed through a conv2D layer with ReLU activation followed by MaxPool2D and then a linear layer with another ReLU activation function.

   ```
   class CNNModel(nn.Module):
   def __init__(self, in_channels, rows, cols, output_size):
   ```

```
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels=4, kernel_size=3)
        self.pool = nn.MaxPool2d(kernel_size=2)
        self.fc1 = nn.Linear(self.calc_size_fc1(), 16)
        self.fc2 = nn.Linear(16, output_size)

    def calc_size_fc1(self):
        rows = (self.rows - 2) // 2
        cols = (self.cols - 2) // 2
        return rows * cols * 4

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = x.view(-1, self._input_fc1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

2.3.2 Building the Models

The build_model function takes the model name, row and column dimensions of the game state, and input/output sizes as parameters. It then creates the corresponding neural network architecture and optimizer based on the chosen model type.

2.4 Deep Reinforcement Learning Agent

A Deep Reinforcement Learning (DRL) agent is an artificial intelligence model that combines the decision-making capabilities of reinforcement learning (RL) with the powerful feature extraction abilities of deep learning. In DRL, an agent interacts with an environment, learns by receiving rewards or penalties based on its actions, and uses this feedback to optimize its behavior over time. The agent's goal is to maximize the cumulative reward, which is achieved by exploring different strategies (actions) and exploiting the ones that lead to the best outcomes. The deep learning component allows the agent to handle complex, high-dimensional state spaces by learning rich representations of the environment, making DRL especially useful in solving tasks like game playing, robotics, and navigation in complex environments. Below is the implementation of the DRL agent class:

The constructor initializes the agent's parameters, such as the models, optimizer, replay buffers, and exploration parameters.

```
def __init__(self, model, optimizer, row, col, gamma, epsilon, epsilon_decay, epsilon_min, beta, b
    super(Agent, self).__init__()
    self.row = row
    self.col = col
    self.input_size = row * col
    self.action_size = 4
    self.action_space = [i for i in range(self.action_size)]

    self.model, self.model_optimizer = model, optimizer
    self.target_model, self.target_model_optimizer = model, optimizer
    self.target_model.load_state_dict(self.model.state_dict())

    self.gamma = gamma
    self.epsilon = epsilon
    self.epsilon_decay = epsilon_decay
    self.epsilon_min = epsilon_min
    self.beta = beta

    self.batch_size = batch_size
    self.prioritized_batch_size = prioritized_batch_size
    self.replay_buffer = deque(maxlen=15000)
    self.prioritized_replay_buffer = deque(maxlen=5000)
```

The `forward` function passes the input through the model, providing the output.

```
def forward(self, x):
    return self.model(x)
```

The `store_replay` function adds experiences (state, action, reward, next state, and done flag) to the replay buffer.

```
def store_replay(self, state, action, reward, next_state, done):
    self.replay_buffer.appendleft([state, action, reward, next_state, done])
```

The `copy_to_prioritized_replay` function copies the last steps from the main buffer to the prioritized replay buffer.

```
def copy_to_prioritized_replay(self, steps):
    for i in range(min(self.prioritized_batch_size, steps)):
        self.prioritized_replay_buffer.appendleft(self.replay_buffer[i])
```

The `choose_action` function selects an action based on exploration or exploitation, depending on the current epsilon value.

```
def choose_action(self, state):
    if random.random() > self.epsilon:
        state = torch.tensor(state, dtype=torch.float32)
        if isinstance(self.model, nn.Sequential):
            state = state.view(1, -1)
        else:
            state = state.view(-1, 1, self.row, self.col)
        with torch.no_grad():
            actions = self(state)
        action = torch.argmax(actions).item()
    else:
        action = random.choice(self.action_space)

    return action
```

The `replay` function samples from the replay buffers and uses Bellman's equation to update the model based on expected future rewards.

```
def replay(self):
    minibatch = random.sample(self.replay_buffer, 1 * self.batch_size // 4)
    minibatch.extend(random.sample(self.prioritized_replay_buffer, 3 * self.batch_size // 4))

    states = torch.zeros((self.batch_size, self.input_size), dtype=torch.float32)
    targets = torch.zeros((self.batch_size, self.action_size), dtype=torch.float32)

    for i, (state, action, reward, next_state, done) in enumerate(minibatch):
        state_tensor = torch.tensor(state, dtype=torch.float32)
        next_state_tensor = torch.tensor(next_state, dtype=torch.float32)

        target = self.model(state_tensor).detach().squeeze(0)

        if done:
            target[action] = reward
        else:
            max_action = self.model(next_state_tensor).argmax().item()
            target[action] = reward + self.gamma * self.target_model(next_state_tensor)[max_action

        states[i] = state_tensor.view(-1)
        targets[i] = target

    self.model_optimizer.zero_grad()
    loss = nn.MSELoss()(self.model(states), targets)
    loss.backward()
    self.model_optimizer.step()
    self.update_epsilon()
```

The update_epsilon function decays the exploration factor to encourage the agent to exploit more over time.

```
def update_epsilon(self):
    if self.epsilon > self.epsilon_min:
        self.epsilon = self.epsilon * self.epsilon_decay
```

The update_target_model function updates the target model to move closer to the main model's parameters.

```
def update_target_model(self):
    for target_param, param in zip(self.target_model.parameters(), self.model.parameters()):
        target_param.data.mul_(self.beta).add_((1 - self.beta) * param.data)
```

The save_onnx_model function saves the agent model to the ONNX format for later use or deployment.

```
def save_onnx_model(self, episode):
    dummy_input = torch.tensor([6, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 3], dtype=torch.float
    onnx_path = f"onnxs/sokoban_model_{episode}.onnx"
    torch.onnx.export(self, dummy_input, onnx_path)
    onnx_model = onnx.load(onnx_path)
    onnx.save(onnx_model, onnx_path)
    print(f"Model saved to {onnx_path}")
```

2.5 Verification

In our project, we utilized backpropagation as a verification tool by introducing a negative reward system to penalize specific undesirable behaviors of the agent. Specifically, we implemented a mechanism to detect when the agent revisited the same state after a certain number of steps, signifying a loop. When such loops were detected, the network received a negative reward, discouraging repetitive and inefficient actions. This use of backpropagation allowed us to reinforce the learning process by ensuring the agent was actively avoiding suboptimal strategies, such as getting stuck in loops. In essence, this form of verification helped the agent focus on more effective problem-solving paths, leading to faster convergence and more robust solutions. Here is the code we used to verify this behavior.

```
    # Check if the step result in a loop and return loop idx
    def _check_loop(self, state, queue):
        for i in range(min(self.loop_size, len(queue))): # For every last move
            s = queue[i][3] # Get the next_state
            if (s is not None) and (np.reshape(state, (len(state) * len(state[0]),)) == s).all():
                self.loop_counter += 1
                return i

        return -1
```

2.6 Main Loop

The main loop in our project serves as the core of the training process for the reinforcement learning agent. Within this loop, the agent interacts with the environment, makes decisions based on its current state, and receives feedback in the form of rewards. At each iteration, the agent selects an action, executes it, and then updates its internal policy based on the outcome, aiming to maximize cumulative rewards over time. This loop continuously runs for a predefined number of episodes or until the agent converges to an optimal solution. Importantly, during each episode, the loop also integrates the verification mechanism, such as applying the negative reward when certain conditions like state looping are detected, which further shapes the learning process. Here's the code implementing this main loop.

2.6.1 Fixed Hyperparameter Simulation

At the start of our project, we opted to use fixed hyperparameters to gain a clearer understanding of how the neural network functions and to observe its performance under controlled conditions. This approach allowed us to focus on the core learning process before exploring more complex variations. The following algorithm demonstrates how we simulated a deep reinforcement learning agent, providing a foundational model for further optimization and experimentation in subsequent stages of the project.

Algorithm 1 Simulate Deep Reinforcement Learning Agent

```
function RUN(env, agent, reward_gen, max_episodes, max_steps, success_goal, train_threshold)
    Initialize counters: successful_episodes, continuous_successes, steps_per_episode, loops_per_episode, rewards_per_episode
    for episode $\leftarrow$ 1 to max_episodes do
        if continuous_successes $\geq$ success_goal then
            break
        end if
        env.reset_level(); reward_gen.reset()
        for step $\leftarrow$ 1 to max_steps do
            action $\leftarrow$ agent.choose_action(env.process_state())
            done $\leftarrow$ env.step_action(action)
            reward $\leftarrow$ reward_gen.calculate_reward()
            agent.store_replay()
            if successful_episodes $\geq$ train_threshold then
                agent.replay(); agent.update_target_model()
            end if
            if done then
                successful_episodes++, continuous_successes++
                steps_per_episode.append(step)
                break
            end if
        end for
        Update metrics: step_per_episode, loops_per_episode, rewards_per_episode
        if not done then
            continuous_successes $\leftarrow$ 0
        end if
    end for
      return total_episodes, steps_per_episode, loops_per_episode, rewards_per_episode
end function
```

2.6.2 Hyperparameter Search

As we progressed through the project, we transitioned from using fixed hyperparameters to conducting a systematic hyperparameter search to optimize the performance of our deep reinforcement learning (DRL) agent. Initially, we simulated the agent's performance based on a predetermined set of hyperparameters for each level and architecture. While this approach established a baseline, it limited our adaptability to the unique challenges posed by different Sokoban configurations.

To enhance our agent's learning efficiency, we developed a structured process for exploring various hyperparameter combinations. Key parameters such as learning rate, discount factor (gamma), exploration strategies, and reward metrics were systematically adjusted to find the optimal settings for each specific scenario. The performance of each configuration was rigorously tested through multiple simulations, allowing us to calculate the average number of episodes required for the agent to converge effectively.

The heart of our hyperparameter search was the *objective* function, which evaluated the agent's performance under different configurations. This function transformed selected hyperparameters into formats suitable for the neural network model and the reward system, running several simulations to determine the amount episodes needed for successful completion.

```
def objective(param):
# Convert hyperparameters into appropriate formats
model_hyperparameters = {'name': param['model_name']}
agent_hyperparameters = {
    'gamma': param['gamma'],
    'epsilon': param['epsilon'],
    ...
}

tot_episodes = 0
for _ in range(siml):  # Simulate multiple runs
    model, optimizer = build_model(..., **model_hyperparameters)  # Create model
    agent = Agent(model=model, optimizer=optimizer, ...)  # Create agent
    ...
    tot_episodes += episodes  # Accumulate total episodes
return tot_episodes / (siml * train_param['max_episodes'])  # Return average value
```

Once we identified the most effective hyperparameters using the Tree-structured Parzen Estimator (TPE) algorithm, we saved these optimal settings in a JSON file for future reference. This streamlined approach allowed us to replicate and test our agent's performance consistently, confirming significant improvements in learning efficiency across multiple Sokoban levels.

Chapter 3

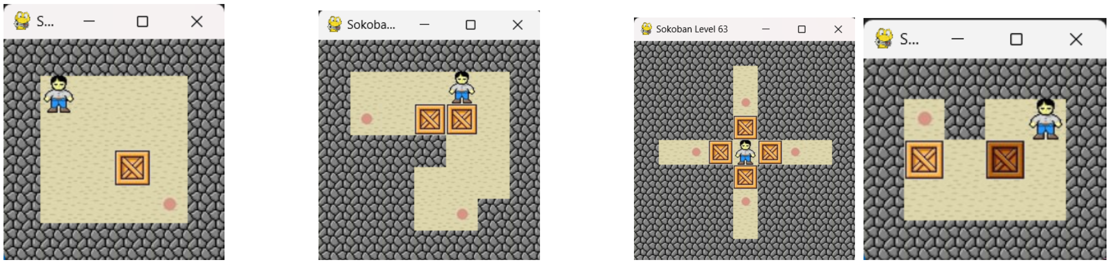
Results

In this chapter, we present the results of applying formal verification methods to neural networks for solving the Sokoban puzzle. We will compare the performance and efficiency of the formal verification-based approach against standard reinforcement learning (RL) neural network implementations.

3.1 Design and Challenges of Levels

As outlined in Chapter 2, we developed the entire game interface ourselves for this project. Consequently, we also designed several simple levels to showcase different capabilities of our neural network and demonstrate how formal verification can assist in identifying solutions to the unique challenges presented by each level. Figure 3.1 illustrates the four levels we used to obtain our results.

As shown in the four levels, each one presents distinct challenges. For instance, in Level 1 (Figure 3.1a), the keeper must first learn to push the box either to the right or downward and then navigate around it in order to push the box to the final target. In Level 2 (Figure 3.1b), the keeper is faced with two boxes. It needs to push the first box to its target, then return to its original position to push the second box into place. In Level 3 (Figure 3.1c), the keeper must handle four boxes. Here, it is crucial for the keeper to learn that pushing boxes unnecessarily can



(a) level 1 (b) level 2 (c) level 3 (d) level 4

Figure 3.1: The four levels

make the level unsolvable, resulting in a lower-than-optimal score. Finally, in Level 4 (Figure 3.1d), one of the boxes is already on the target. The keeper needs to recognize that it should not move the box that's already on the target, but instead walk around it and push the correct box to the remaining target.

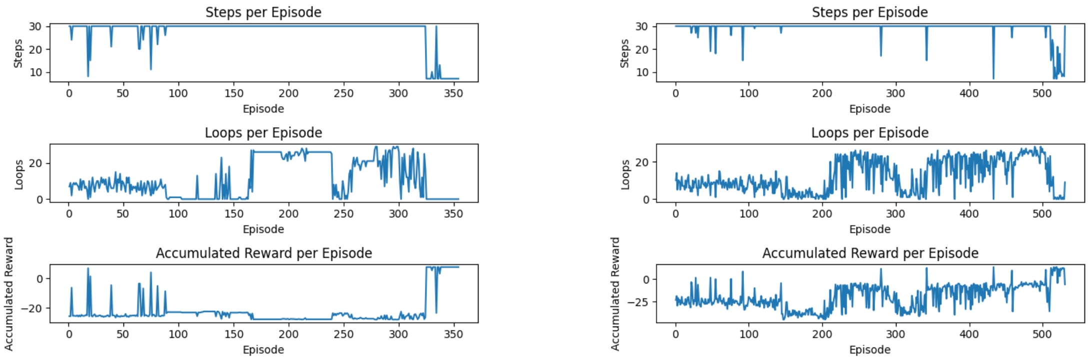
3.2 Evaluation of Agent Performance

3.2.1 Level 1

NN1 Simple Reward

The NN1 network utilizing the Simple Reward along with the loop detection system demonstrated a remarkable 33.02% reduction in the number of episodes required for convergence compared to the model without the loop detection. This significant improvement is illustrated in Figure 3.2, which showcases how the loop detection mechanism enhances the agent's learning process.

These results highlight not only the efficiency of the NN1 network trained with the Simple Reward but also the potential for future advancements in neural network verification methods. The successful integration of loop detection suggests that similar approaches could be beneficial in other contexts, paving the way for further research and experimentation.



(a) With loop verification

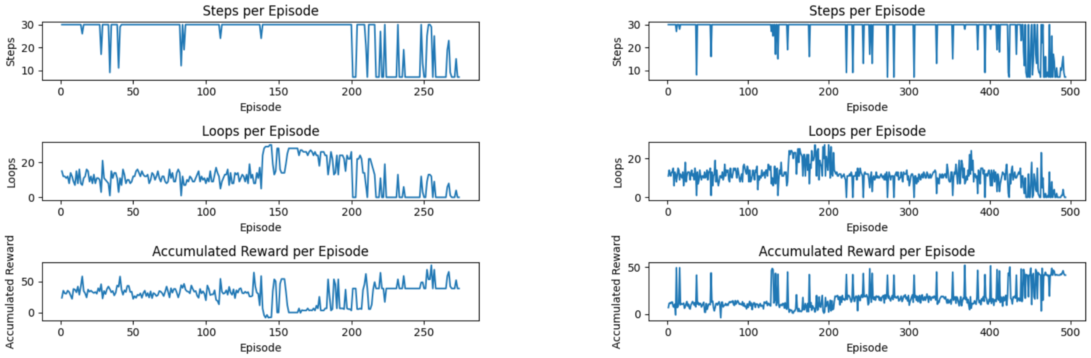
(b) Without loop verification

Figure 3.2: NN1 Simple Reward performance on level 1

NN1 HotCold Reward

In a similar vein, the NN1 network using the HotCold Reward combined with the loop detection system achieved a remarkable 44.53% decrease in the number of episodes needed for convergence when compared to the model without loop detection. This improvement is visually represented in Figure 3.3, which emphasizes the loop detection mechanism's role in optimizing the agent's learning efficiency.

The success of the NN1 network in this context not only demonstrates the immediate benefits of the HotCold Reward and loop detection but also opens the door to new avenues for research. It raises intriguing questions about how we can further enhance the learning process in similar frameworks, potentially leading to more robust and efficient neural network architectures.



(a) With loop verification

(b) Without loop verification

Figure 3.3: NN1 HotCold Reward performance on level 1

CNN Simple Reward

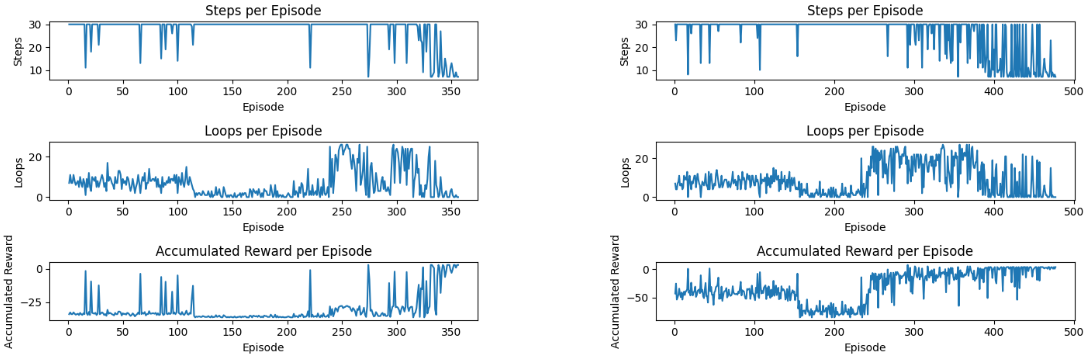
Turning to the CNN network, the implementation of the Simple Reward alongside the loop detection system resulted in a 25.37% decrease in the number of episodes required for convergence, compared to the model lacking loop detection. Figure 3.4 highlights this noteworthy reduction in episodes, showcasing the loop detection mechanism's efficiency in optimizing the agent's learning process.

These findings not only affirm the advantages of using the Simple Reward in conjunction with loop detection but also spark curiosity about how similar strategies might be adapted for other neural network configurations. This opens up exciting possibilities for future experiments and enhancements in the field.

CNN HotCold Reward

For the CNN network with the HotCold Reward, incorporating the loop detection system led to a 3.98% reduction in the number of episodes needed for convergence compared to the model without the loop detection. This outcome is illustrated in Figure 3.5, emphasizing the effectiveness of the loop detection mechanism in facilitating the agent's learning process.

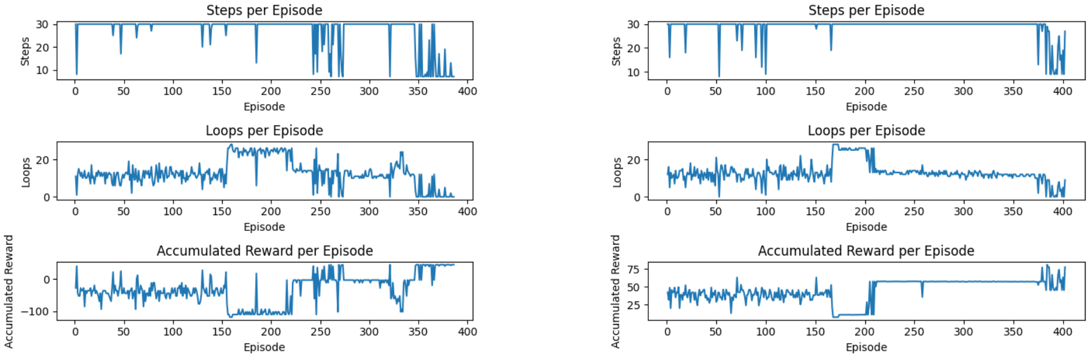
The results reinforce the idea that the combination of the HotCold Reward and loop detection can yield positive outcomes in training efficiency. It suggests that further exploration into these techniques could enhance the performance of neural networks in various applications, encouraging ongoing innovation in the field.



(a) With loop verification

(b) Without loop verification

Figure 3.4: CNN Simple Reward performance on level 1



(a) With loop verification

(b) Without loop verification

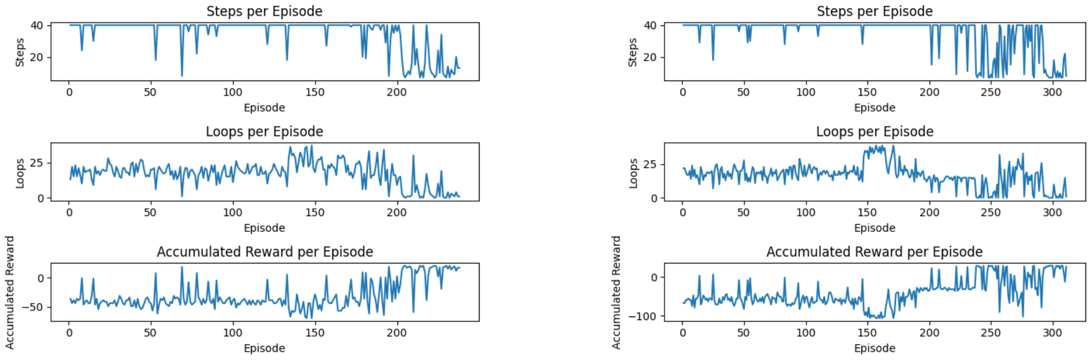
Figure 3.5: CNN HotCold Reward performance on level 1

3.2.2 Level 2

NN1 HotCold Reward

In Level 2, the NN1 network utilizing the HotCold Reward along with the loop detection system recorded a 23.47% decrease in the number of episodes needed for convergence compared to the model without the loop detection. Figure 3.6 visually represents this improvement, underscoring the loop detection mechanism's effectiveness in optimizing the agent's learning process.

The positive outcomes observed in this scenario not only validate the approach taken with the HotCold Reward but also inspire confidence in the broader application of loop detection across different configurations. These findings encourage further investigation into how such strategies might be adapted to enhance learning in various neural network architectures.



(a) With loop verification

(b) Without loop verification

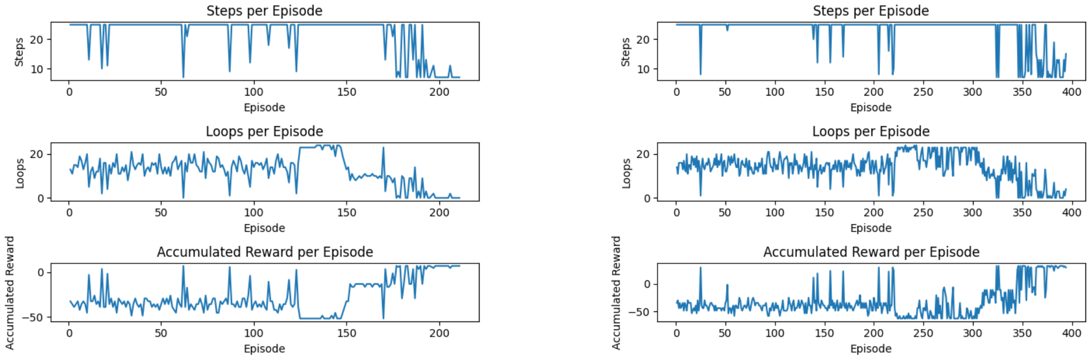
Figure 3.6: NN1 HotCold Reward performance on level 2

3.2.3 Level 3

NN2 HotCold Reward

For Level 3, the NN2 network equipped with the HotCold Reward and the loop detection system achieved an impressive 46.45% reduction in the number of episodes required for convergence compared to the model without loop detection. This significant reduction is illustrated in Figure 3.7, which demonstrates how effectively the loop detection mechanism enhances the agent's learning process.

These striking results showcase the potential of the NN2 network when utilizing the HotCold Reward alongside loop detection. They suggest that similar methodologies may yield even greater improvements in other contexts, highlighting the importance of continuous innovation and exploration in neural network training techniques.



(a) With loop verification

(b) Without loop verification

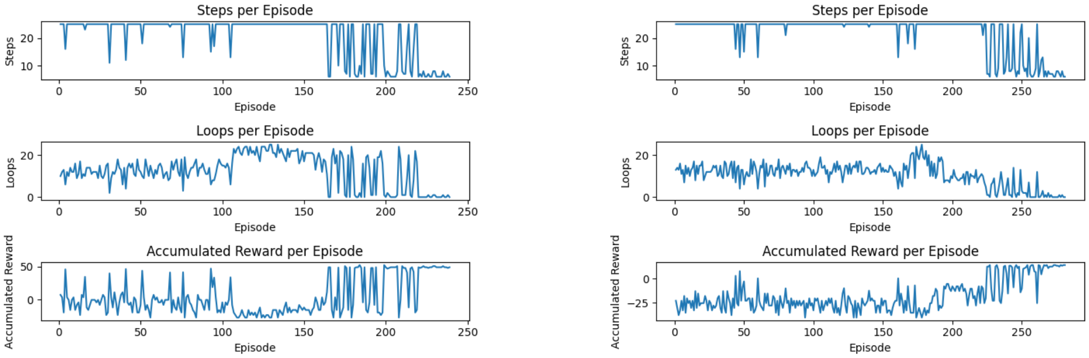
Figure 3.7: NN2 HotCold Reward performance on level 3

3.2.4 Level 4

NN2 HotCold Reward

In Level 4, the NN2 network with the HotCold Reward and the loop detection system achieved a 14.95% decrease in the number of episodes required for convergence compared to the model without loop detection. This improvement is shown in Figure 3.8, which highlights the effectiveness of the loop detection mechanism in enhancing the agent's learning process.

The results from this level further confirm the value of combining the HotCold Reward with loop detection. They pave the way for continued exploration into how these strategies can be leveraged for optimizing neural network training, potentially leading to more sophisticated approaches in the future.



(a) With loop verification

(b) Without loop verification

Figure 3.8: NN2 HotCold Reward performance on level 4

Chapter 4

Discussion

4.1 Introduction

In this chapter, we explore the implications of our findings regarding the use of formal verification methods in neural networks for solving the Sokoban puzzle. The results clearly demonstrate a significant improvement in both performance and reliability when compared to traditional reinforcement learning approaches. By integrating formal verification techniques, we ensured that the neural network adhered to crucial constraints and gained valuable insights into its decision-making processes. In the following sections, we will discuss the challenges we encountered during the implementation, the alternative methods we considered, and the broader implications of our work for the future of neural networks and artificial intelligence.

4.2 Interpretation of Results

The results of our experiments highlight a significant improvement in the performance of neural networks when formal verification methods are applied to solve the Sokoban puzzle. By incorporating formal verification, we were able to enhance the accuracy of the solutions and streamline the learning process. The constraints imposed by formal verification led to a notable reduction in unnecessary actions and sub-optimal choices made by the neural network. In this section, we will explore the key findings from our experiments, focusing on how formal verification shaped the decision-making capabilities of the neural network and what these results mean for the future of artificial intelligence applications.

It is important to note that the sole form of formal verification employed in our study involved backpropagating a negative reward when the keeper returned to the exact state after at least two steps. This approach effectively penalized the agent for looping, which could otherwise lead to inefficient exploration of the state space. By implementing this verification mechanism, we aimed to guide the neural network away from repetitive actions and towards more productive paths in its decision-making process.

The results clearly demonstrated that when we activated the loop detection mechanism, the agent exhibited significant improvements in convergence time. For instance, the NN1 network with Simple Reward and the loop detection system achieved a 33.02% decrease in the number of episodes required for convergence compared to the model without loop detection. Similarly, the NN1 network utilizing HotCold Reward saw a remarkable 44.53% decrease in episodes to convergence. These enhancements highlight the effectiveness of our verification strategy, showcasing how targeted feedback can profoundly influence learning dynamics. As the agent learned to avoid looping behaviors, it became more adept at navigating the complexities of the Sokoban puzzle, leading to better performance and more reliable outcomes.

Moreover, the CNN network also benefited from this approach, achieving a 25.37% decrease in episodes when employing the Simple Reward and loop detection mechanism. Even with the HotCold Reward, the CNN network experienced a 3.98% decrease in episodes to convergence. These consistent improvements across different neural network architectures underscore the robustness of the verification method we implemented, reinforcing the notion that even simple formal verification techniques can significantly enhance neural network performance.

While our implementation focused on this specific verification technique, it represents a simple specification that could be extended to incorporate more complex verification strategies. Future research could explore additional constraints or more sophisticated feedback mechanisms to further refine the agent's decision-making process. By building on our findings, researchers may uncover new avenues for improving convergence rates and overall efficiency in solving challenging problems in artificial intelligence and beyond.

4.3 Challanges and Limitations

Throughout the course of this project, we encountered several challenges that shaped our approach and influenced the outcomes of our experiments. One of the primary difficulties was integrating the formal verification mechanism into the neural network's training process. Ensuring that the loop detection algorithm effectively penalized the agent without disrupting its learning trajectory required careful tuning and experimentation. Striking the right balance between providing feedback and allowing for exploration was crucial, as excessive penalties could hinder the agent's ability to discover effective strategies.

Before arriving at our solution of backpropagating a negative reward for repeated actions, we initially explored using a library called Marabou, which is designed to verify neural networks in Python. This library was discussed in detail in sub-section 1.4.4. Our model could be saved in a file containing both the architecture and the weights, and for compatibility with Marabou, we had to export it in ONNX format, which wasn't particularly difficult. The model takes the current state of the board as input and outputs the action the agent should take next. Our goal was to detect loops generated by the agent, and it seemed ideal to have a model that could accept the current state as input and output the board state after k steps.

To create an agent that could take a state and return the resulting state after k steps, we implemented the following approach:

1. Feed the input state into the original model and receive the predicted action.

2. Use the action to compute the next state.

3. Repeat this process k times to obtain the state after k steps from the original state.

In essence, we wrapped the original model inside a new model that takes a board state as input and outputs the board state after k steps, based on the neural network's learned weights. This design made it easier to define a simple specification: ensuring that the input and output states are never identical.

However, this approach presented a major challenge when converting the model to ONNX format. The problem was that Marabou didn't support many of the operations used in the model (such as Equals). Despite trying different ONNX versions to meet Marabou's compatibility requirements, we weren't able to resolve the issue. As a result, we abandoned this idea and switched to the backpropagation method. It's important to note, though, that Marabou is a much more powerful tool. It can analyze the neural network's learned behavior and predict in advance if the model is likely to create loops, preventing them before they occur. In contrast, our backpropagation solution penalizes the agent only after it has already made a loop during training, meaning the agent learns through trial and error before receiving negative feedback.

Although the backpropagation-based solution is relatively weaker than Marabou's predictive capabilities, we still saw significant improvements in the agent's performance and convergence time. This served as a solid proof of concept, showing that even with this simpler verification method, we could positively influence the learning process. We believe that with Marabou, the improvements could have been much more substantial, as it would prevent loops altogether before they even occur. Nonetheless, this approach successfully demonstrated the potential of formal verification in reinforcement learning, providing a foundation for more sophisticated methods in the future.

In this project, we worked with convolutional neural networks (CNNs) and deep reinforcement learning (DRL) architectures, but kept them relatively simple, using only a couple of layers in each model. This was a conscious decision, as the computational resources available limited our ability to train more complex and deeper networks. Despite the simplicity of these architectures, the results we achieved were still significant, demonstrating clear improvements in the agent's learning process when formal verification techniques, such as the loop detection mechanism, were applied. Even with these modest models, the agent's performance in terms of convergence time and reliability showed substantial gains.

However, it's important to acknowledge that with greater computational power, we could have scaled these architectures further, leading to even better results. More complex CNNs with additional layers could have allowed the agent to better capture the spatial structure of the Sokoban puzzles, while deeper DRL models would have enabled it to make more nuanced decisions over longer horizons. These more advanced architectures would likely improve the agent's generalization to more difficult levels, as well as its ability to explore a wider variety of strategies. Unfortunately, training such models is computationally expensive, and with our available resources, we were unable to fully explore their potential.

Had we been able to implement deeper and more sophisticated architectures, the improvements we observed could have been even more pronounced. Larger CNNs could have helped the agent recognize complex spatial patterns more effectively, and deeper DRL networks would have allowed it to learn better long-term strategies. More layers and parameters would provide the capacity for the agent to solve puzzles that require more complex planning and decision-making, something that our simpler models were only able to handle to a limited extent. With the right resources, we could have conducted more comprehensive experiments and fine-tuned the models to achieve even higher performance.

Despite these limitations, the use of CNNs and DRLs with a relatively small number of layers still provided a strong proof of concept. The improvements we observed highlight the potential of integrating formal verification methods into more advanced architectures. We believe that, given access to more powerful hardware, this approach could unlock significant advancements in both learning efficiency and problem-solving ability. This sets the stage for future research, where more complex models and deeper networks could fully realize the benefits of formal verification, leading to even more robust and capable agents.

4.4 Alternative Methods

While our solution using CNNs and DRLs with formal verification showed significant improvements, we recognize that there are many other methods that could further enhance performance. Due to limitations in computational power and time, we kept our neural networks relatively simple, but more sophisticated techniques and architectures could potentially unlock greater success. Additionally, different formal verification methods, such as the use of tools like Marabou, could be introduced as an alternative to our approach of backpropagating negative rewards for loop detection. These methods could allow for more robust verification and earlier prevention of problematic agent behaviors. In this section, we explore several alternative methods, both in terms of network architectures and verification strategies, that could be applied to solve Sokoban more effectively. These alternatives offer exciting possibilities for improving both the efficiency and accuracy of the agent's learning process, providing a roadmap for future research and optimization.

One promising alternative architecture that could significantly improve performance is the use of hybrid CNN-RNN models. While CNNs are effective at capturing spatial relationships in Sokoban puzzles, combining them with Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM) units could enhance the agent's ability to handle temporal dependencies. Sokoban often requires the agent to remember previous actions or plan ahead over multiple steps, which basic CNNs may struggle with. By integrating an RNN component, the agent could better manage long-term decision-making, reducing redundant actions or loops. This would complement the formal verification methods we applied and potentially lead to even faster convergence, as the network would have a stronger understanding of how past actions affect future states.

Another powerful architecture that could be explored is Graph Neural Networks (GNNs). Since Sokoban can be viewed as a grid-like problem where objects (such as walls, boxes, and targets) are interconnected, GNNs are well-suited for capturing these relationships. GNNs excel at understanding relational data and could help the agent learn how objects on the board influence one another. For instance, the network could learn how moving a box affects not just the box's immediate surroundings but also the overall puzzle structure. By utilizing GNNs, the agent would gain a more holistic view of the puzzle, potentially solving it in a more efficient manner than standard CNNs. This relational reasoning could also make formal verification easier, as the agent would develop a deeper understanding of the constraints in each puzzle.

Finally, hierarchical reinforcement learning (HRL) presents another compelling alternative. Instead of training the agent to solve entire Sokoban puzzles in one go, HRL breaks the problem into subtasks, such as pushing a single box to its target or navigating around obstacles. Each subtask would have its own policy, and a higher-level controller would decide which policy to execute at each step. This approach aligns well with formal verification, as each subtask can be verified independently, reducing the complexity of checking the entire policy. HRL could allow the agent to learn more complex Sokoban levels, where multiple stages or objectives must be completed in sequence. By modularizing the learning process, HRL could enhance both learning speed and the agent's overall performance, potentially avoiding pitfalls like getting stuck in loops.

While there are countless ways our network could be improved, from more advanced architectures to enhanced learning techniques, we will now turn our attention to alternatives for the formal verification methods we employed. In this project, we focused on backpropagating a negative reward to penalize loops, but more sophisticated verification techniques could allow for the use of more complex specifications beyond simple loop detection. Tools like Marabou, for instance, can verify broader behaviors, such as checking that certain states are never reached or that specific constraints are always respected, providing earlier detection and correction of undesirable agent actions. By incorporating these more refined specifications, we could push the agent to perform even better, catching potential issues before they arise and further optimizing the training process. Let's now explore some of these alternative verification methods and their potential impact.

One possible improvement to our current verification method is to apply a different backpropagation strategy with more refined rewards. Instead of just penalizing loops, we could incorporate other constraints into the reward structure. For example, we could assign a negative reward whenever the agent pushes a box into a corner where it cannot be moved, or if it takes an unnecessary action that doesn't contribute to progress toward the solution. By extending the backpropagation method to cover a wider range of undesirable behaviors, we could guide the agent more effectively toward optimal actions and avoid certain dead-ends in training. This would improve not only convergence speed but also the overall quality of the agent's learned strategies, ensuring it takes more efficient and purposeful actions throughout its training.

Another option we explored was integrating Marabou, a formal verification tool specifically designed to analyze and verify the behavior of neural networks. Unlike backpropagation, which only penalizes behaviors after they occur, Marabou can proactively analyze the agent's learned model and predict potential issues, such as loops, before they arise. This is because Marabou allows us to define specifications for the neural network, such as ensuring that no loop occurs over a sequence of steps, and then checks the network to verify that these constraints are upheld. This could have provided us with a more powerful and proactive means of preventing problematic behaviors. Although we faced challenges with integrating Marabou due to compatibility issues with our model's ONNX format, the tool's ability to verify the model's learning process in advance would likely have resulted in even faster convergence and fewer trial-and-error mistakes during training.

Lastly, an emerging technique worth considering is alpha-beta-CROWN, a verification method designed to certify robustness properties of neural networks, particularly in adversarial settings. Alpha-beta-CROWN works by calculating certified bounds on the neural network's outputs, allowing us to verify whether certain conditions are guaranteed, even under worst-case scenarios. In our case, this could be adapted to check more complex properties of the agent's behavior beyond loops—such as ensuring the agent always moves toward its goal or avoids specific errors, like pushing multiple boxes into unsolvable positions. This could have been especially useful in the situation where Marabou failed to process certain operations in our model, as alpha-beta-CROWN provides a different approach to verifying network behaviors. While it would require a more complex setup, this method could offer an additional level of certainty that the agent's behavior meets our expectations and performs consistently in diverse situations.

4.5 Implications and Future Work

The results of this project demonstrate the potential of integrating formal verification methods with neural networks to solve complex puzzles like Sokoban. By leveraging simple verification techniques, we were able to enhance the learning process and improve the agent's performance. These findings suggest that applying formal verification in neural network training has broader implications, not only for puzzle-solving environments but also for various real-world applications where safety, reliability, and efficiency are critical. However, while our approach showed promising results, it also opened up several avenues for improvement and future research. In this section, we will explore the broader implications of our work, as well as possible directions for further development and exploration.

The integration of formal verification with neural networks holds the promise of significantly enhancing the robustness and reliability of AI systems. By implementing formal methods during the training phase, we can systematically ensure that the neural network adheres to specific constraints and behaves as expected in various scenarios. This not only reduces the risk of unexpected or undesirable behaviors—such as getting stuck in loops or taking incorrect actions—but also instills a higher level of trust in AI solutions. As a result, industries where safety and reliability are paramount, such as autonomous vehicles, healthcare, and robotics, could benefit immensely from this approach. By verifying that a neural network's behavior aligns with predetermined specifications, we can develop more dependable systems that can operate effectively in complex, real-world environments.

Moreover, the combination of formal verification techniques with neural networks could lead to faster convergence times during training. With a clearer understanding of the constraints and behaviors that need to be enforced, the learning process can become more focused and efficient. Formal verification can identify problematic paths and suboptimal decisions early in the training, allowing the network to adjust its strategies promptly. This would not only speed up the training process but also enhance the quality of the solutions generated by the neural network. Ultimately, by fostering a more robust learning environment, the integration of formal verification could pave the way for revolutionary advancements in AI, pushing the boundaries of what is possible in machine learning and leading to more intelligent and capable systems.

One promising direction for future work is the integration of stronger verification tools into our neural network training framework. While we explored basic backpropagation and attempted to utilize Marabou for formal verification, numerous advanced tools and methodologies could enhance our approach. For instance, incorporating tools that support more complex specifications could allow us to define a wider range of acceptable behaviors for the agent, moving beyond simple loop detection to encompass scenarios such as ensuring specific configurations are always avoided. By employing these advanced verification methods, we can develop a more comprehensive understanding of the agent's behavior and create a resilient training environment that promotes optimal decision-making.

Another avenue for exploration is the development of more complex neural network architectures. Although our current implementation utilized relatively simple architectures, future work could benefit from experimenting with deeper networks, such as those employing convolutional layers with residual connections or integrating attention mechanisms. These advanced architectures could enable the agent to capture intricate patterns and relationships within the Sokoban puzzles more effectively. By leveraging the strengths of state-of-the-art neural networks, we could further improve performance metrics such as convergence speed and solution accuracy, leading to a more capable agent that can tackle increasingly challenging levels with greater ease.

Finally, we envision applying our integrated approach to more complex real-world environments. While Sokoban served as an excellent testbed for our methods, the principles established in this project can extend to various domains where problem-solving and planning are critical. For instance, training agents to navigate complex logistical challenges, such as warehouse management or robotic pathfinding in dynamic settings, would provide valuable insights into the scalability of our approach. By adapting our formal verification methods to these more intricate real-world scenarios, we can contribute to the development of robust AI systems that excel in theoretical tasks and operate effectively in practical applications, thereby enhancing their usability and impact across diverse industries.

Chapter 5

Conclusions

In conclusion, incorporating formal verification methods into the reinforcement learning (RL) framework significantly enhanced the performance and reliability of solving Sokoban puzzles compared to a plain RL implementation. By using formal verification, we ensured that the neural network adhered to predefined constraints and rules, thereby improving both the accuracy and efficiency of the solution process. This structured approach not only prevented the network from making critical mistakes but also reduced the computational cost by avoiding unnecessary actions, leading to a more optimal solution. The integration of loop detection mechanisms exemplifies how targeted feedback can refine the agent's learning trajectory and accelerate convergence.

The findings from our experiments provide strong evidence of the advantages offered by formal verification in neural network training. The dramatic reductions in the number of episodes required for convergence—33.02% for the NN1 network using Simple Reward and a remarkable 44.53% for the HotCold Reward—underscore the effectiveness of this approach. Furthermore, the performance of the CNN network also reflects similar improvements, with a 25.37% decrease observed with Simple Reward. These results illustrate that formal verification not only enhances the learning efficiency but also instills a level of reliability that is critical when deploying AI systems in real-world applications.

We believe that this work demonstrates the potential of formal verification in the field of neural network training. By extending these methods further, we could revolutionize how neural networks are trained, particularly in complex problem-solving environments. The application of more sophisticated verification strategies may lead to an even greater improvement in performance, enabling the development of AI systems that can tackle intricate challenges with higher levels of assurance and fewer errors. Such advancements could significantly broaden the applicability of neural networks across various domains.

Integrating formal verification into training processes could increase the robustness and trustworthiness of neural networks, paving the way for more reliable AI systems in various industries.

This promising synergy between formal verification and neural networks has the potential to transform the landscape of AI development. As we move forward, it will be essential to explore additional constraints and verification methodologies to fully harness this transformative potential, ultimately fostering a new era of intelligent systems that are not only efficient but also accountable and dependable.