Formal Verification Methods for Solving Spatial Games

Authors:

Boaz Gurevich & Erel Dekel

Supervisors:

Hillel Kogler & Avraham Raviv

Date:

Thursday, October 10th 2024

Abstract

Reinforcement Learning (RL) has emerged as a cornerstone of modern technology, driving advancements in areas such as autonomous systems, robotics and adaptive control.
However, a critical challenge in deploying RL algorithms lies in their tendency to suffer from convergence issues, particularly in complex, high-dimensional environments where instability can lead to sub-optimal or unsafe policies.
This project aims to explore the impact of formal verification techniques on RL, specifically focusing on how these methods can address the convergence challenges that plague RL systems.
We will empirically investigate this by applying formal verification to a model-free RL algorithm designed to solve the Sokoban puzzle game, a notoriously difficult problem requiring complex planning and decision-making.

Contents

1. Background

In this chapter, we introduce the fundamental concepts that 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 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 devided into 3 types of layers:

1. **Input Layer:** Represents the input features of the network.
2. **Hidden Layers:** Layers between the input and output layers, where most of the computations and transformation of the features happens.
3. **Output Layer:** Represents the output of the network. Usually represent 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, e.i.

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

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 role 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 Network (CNNs)

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, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the probability transition function which defines the probability of transitioning to state s_{t+1} taking action a_t from state s_t, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ 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 which 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 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 iteratively approximate this optimal Q-function through value iteration.

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. **Moves** to the next state s_{t+1} of the environment and receives a reward r_{t+1}.
4. **Updates** 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

1.4 Formal Verification

TBA