A Deep Dive into Solving Optimization Problems using Reinforcement Learning

Dr. Prasant Misra and Risshab Srinivas

September 14, 2025

Contents

1 Introduction

Optimization problems are at the core of many challenges in science and industry, from finding the most efficient delivery routes to designing the strongest materials. Traditionally, these problems are tackled with methods like linear programming or genetic algorithms, which rely on well-defined models and exhaustive searches. However, when problems become highly complex or dynamic, Reinforcement Learning (RL) emerges as a powerful alternative.

1.1 What is Reinforcement Learning?

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by performing actions in an environment to achieve a goal. The agent learns through trial and error, guided by a system of rewards and penalties. The key elements of this branch of Machine Learning include:

- **Agent**: The learner or decision-maker.
- **Environment**: The world the agent interacts with
- **State**: The agent's current situation in the environment
- **Action**: A move the agent can make
- **Reward**: Feedback from the environment

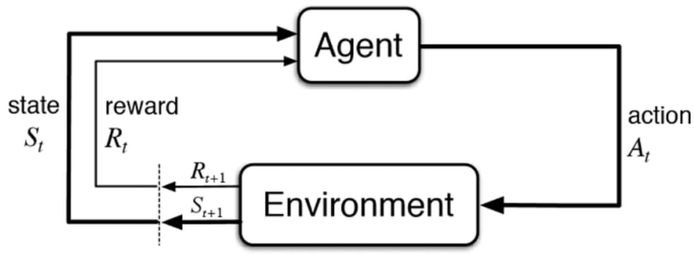

Figure 1: Reinforcement Learning - Architecture

The agent's goal is not to get the best immediate reward, but to maximize its cumulative reward over time. It starts by taking random actions (or based on some policy) to see what happens. If an action leads to a reward, the agent "reinforces" that behavior, making it more likely to take that action again in a similar situation. Over many trials, the agent builds a strategy, known as a policy, that maps states to the best actions to take to achieve its long-term goal.

1.2 RL in Optimization

Reinforcement Learning (RL) offers a powerful and flexible approach to solving complex optimization problems. By framing the problem as an environment, an RL "agent" learns to make optimal decisions through trial and error. The agent takes actions, such as adjusting a variable or choosing the next step in a sequence, and receives a "reward" based on how that action affects the objective function. For example, in a logistics problem, finding a shorter delivery route would result in a higher reward.

Learning Algorithm: Reinforcement learning is a key component in a variety of modern algorithms designed for optimization. In this tutorial, we will explore one of the most prominent of these methods: **Deep Q-Network (DQN)**.

2 Deep-Q-Networks

Deep Q-Networks (DQN) are a cornerstone of modern reinforcement learning, famously used by DeepMind to master Atari games from raw pixel data. At its core, DQN is a reinforcement learning algorithm that uses a deep neural network to learn a strategy, or "policy," for an agent to follow in order to maximize its rewards in a complex environment.

- **Q-Value (Q(s,a))**: The Q-value is a score that estimates the total future reward an agent can expect to receive if it takes a specific action (a) from a given state (s) and continues to play optimally thereafter.
- DQN uses a neural network to approximate the Q-values.
  - **Input**: The current state of the environment (e.g., the (x, y) coordinates).
  - **Output**: A Q-value for each possible action (e.g., up, down, left, right).

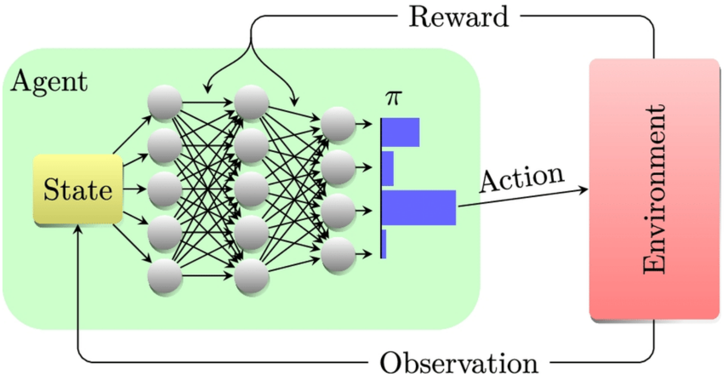

Figure 2: DQN - Architecture

3 Solving the Optimization Problem

In this tutorial, we'll tackle optimization by solving two key examples: a two-variable problem and its more challenging three-variable counterpart. The implementation can be found in the following GitHub Repository

3.1 Two Variable

Objective Function: Our primary goal is to maximize the value of z using the following equation:

$$z = 4x_1 + 3x_2$$

Subject to the following constraints:

$$x_1 + x_2 \leq 40$$

$$2x_1 + x_2 \leq 60$$

$$x_1, x_2 \geq 0$$

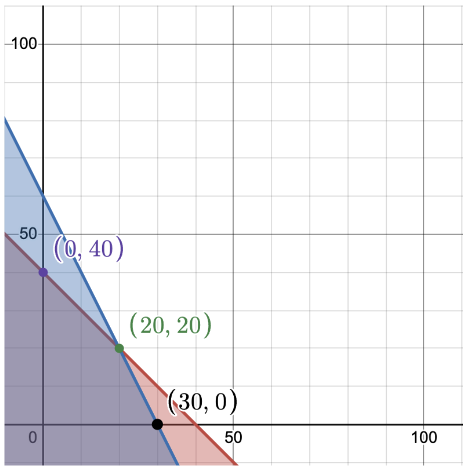

Figure 3: An overview of the problem

3.1.1 Necessary Modules

```python
# Libraries
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from collections import namedtuple, deque
import gymnasium, random
```

Listing 1: Required modules.

Depending on device and GPU availability, one can set the NN to run on either "cpu", "mps" (macOS) or "cuda" (NVIDIA support)

```python
device = torch.device("mps")
print(device)
```

Listing 2: mps used in the above example

3.1.2 Defining the Neural Network

This is the neural network architecture that will act as our agent's "brain." It's a simple feed-forward network that takes the current state (our x and y coordinates) as input and outputs the estimated Q-value for each possible action. The agent will use these Q-values to decide which action to take.

```python
class DQN(nn.Module):
    def __init__(self, state_size, n_actions):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_size, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )
    def forward(self, x):
        return self.net(x)
```

Listing 3: Neural Network Definition

3.1.3 Creating the custom environment

- **State**: Let the state be defined by the variables $s = (x_1, x_2)$. The state is defined by the coordinates (x_1, x_2). In the code, this is represented as a NumPy array. Before being fed into the neural network, this state is normalized by dividing by 100. This scaling ensures that the input values are between 0 and 1, which helps the neural network train more effectively.

```python
def _get_state(self):
    # The state is the current (x, y) coordinate pair.
    return np.array([self.x, self.y], dtype=np.float32)

def reset(self, ...):
    # Returns the initial state, normalized.
    return self._get_state() / 100.0, {}
```

Listing 4: States

- **Reward**: The reward function $R(s)$ is defined as:

$$R(s) = \begin{cases} 4x_1 + 3x_2 & \text{if } x_1 + x_2 \leq 40 \text{ and } 2x_1 + x_2 \leq 60 \\ -P & \text{otherwise} \end{cases} \tag{1}$$

  Where:

  - $4x_1 + 3x_2$ is the objective function, z.
  - P is the penalty for violating a constraint (in our code, $P = 100$).

- **Action**: Actions are the discrete moves the agent can make to navigate the environment. For our 2D problem, we define four possible actions that correspond to moving along the grid's axes.

  Each action is mapped to an integer:

  - 0: Move Up (increase x_2)
  - 1: Move Down (decrease x_2)
  - 2: Move Left (decrease x_1)
  - 3: Move Right (increase x_1)

3.1.4 Training Algorithm

Algorithm 1 Deep Q-Network (DQN) Training Algorithm

Initialize replay memory buffer $\mathcal{D}$ with capacity N
Initialize policy network Q with random weights θ
Initialize target network $\hat{Q}$ with weights $\hat{\theta} \leftarrow \theta$
for all episode = 1 to M **do**
  Reset environment and get initial state s_1
  for t = 1 to T **do**
    Select action a_t using ϵ-greedy policy:
    $$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a; \theta) & \text{otherwise} \end{cases}$$
    Execute action a_t in the environment and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
    Set $s_t \leftarrow s_{t+1}$
    if size of $\mathcal{D}$ > BATCH_SIZE **then**
      Sample a random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
      Set target $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \hat{\theta}) & \text{otherwise} \end{cases}$
      Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ
    end if
    if t mod C = 0 **then**
      Update the target network: $\hat{\theta} \leftarrow \tau\theta + (1 - \tau)\hat{\theta}$
    end if
    if episode terminates **then**
      break
    end if
  end for
end for

3.1.5 Results

After training our DQN agent, we can analyze its performance and behavior using the following visualizations. These graphs confirm that the agent successfully learned to solve the optimization problem

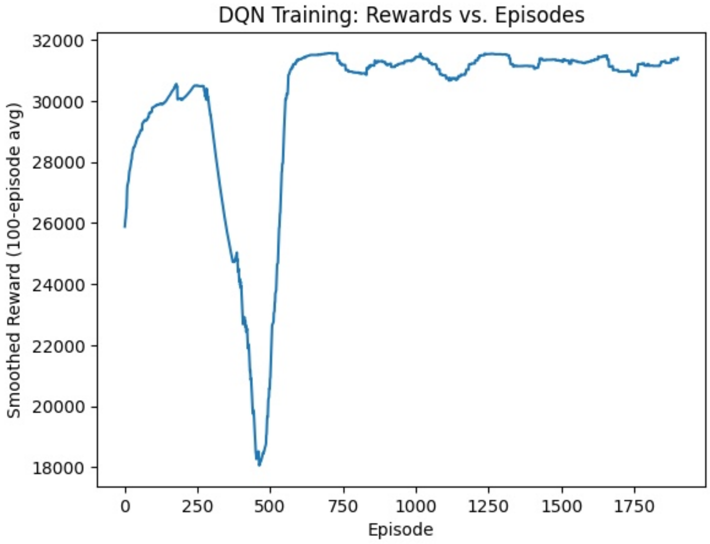

Figure 4: Average reward vs. Episodes

- **Smoothed Rewards**: Figure 4 shows the 100-episode smoothed average of the total reward collected by the agent in each episode. Since our reward function is based on maximizing the objective z while avoiding penalties, this curve's shape closely mirrors the objective value graph. The high and stable reward in the later episodes confirms that the agent is consistently and efficiently achieving its goal: finding high-value states without violating constraints.

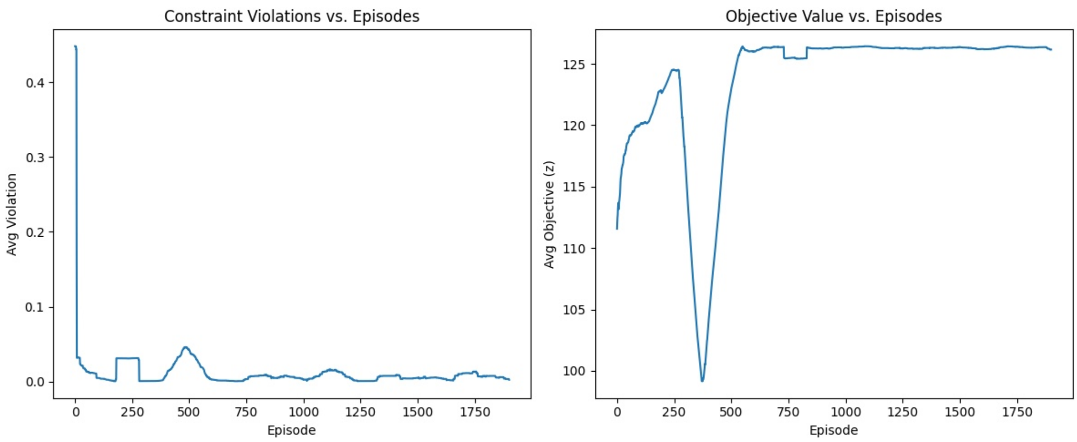

Figure 5: Left: Constraint Violation; Right: Objective value

- **Constraint Violation and Optimization Function**:

  Constraint Violations (Figure 5, Left): This graph shows that the agent very quickly learned the rules of the environment. The average violation starts high during the initial random exploration but drops sharply to near zero within the first couple hundred episodes. This indicates that the agent effectively learned to stay within the feasible region to avoid the large penalties.

Objective Value (Figure 5, Right): This plot tracks the average objective value (z) the agent achieved. The clear upward trend shows the agent successfully learned not just to stay in the feasible region, but to actively seek out states that maximize z. The large dip around episode 450 represents a period of exploration where the agent temporarily tried less optimal strategies but quickly recovered. The final stable, high value shows it has converged on a strong policy.

- Representation: The following graph (Figure 6) provides a visual summary of the agent's behavior.

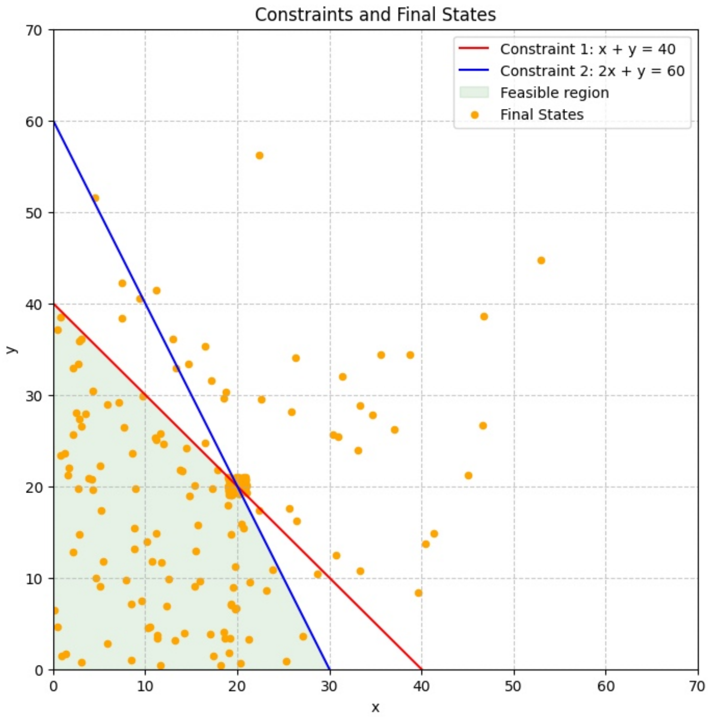

Figure 6: Average reward vs. Episodes

3.2 Three Variable

Objective Function: Our primary goal is to maximize the value of z using the following equation: