Reinforcement Learning Information Sheet

Isaac Wilfong

2025-02-24

Reinforcement Learning

> Reinforcement Learning is a branch of machine learning. Reinforcement Learning involves an agent or decision maker learning by taking actions in an environment to learn the optimal strategy in that environment. Although Reinforcement learning is in a category of its own, it shares similarities from both supervised and unsupervised learning.

In every reinforcement learning model there are five basic components. These are agent, environment, states, actions, and rewards. In reinforcement learning, you have an agent that interacts with an environment. The agents goal is to find the optimized strategy to navigate its world. In a scenario where the environment is a maze, the agents goal might be to find the quickest way out of the maze. The agent might receive penalties from taking wrong moves that lead it to dead ends and receive rewards by making moves that get it closer to the end. Reinforcement learning serves a variety of different industries to solve wildly different problems. These problems can be simple like that in a maze or incredibly complex like solving the optimal price for a taxi ride in Downtown New York on a Friday night.

Agent

The agent in this case is known as the learner or decision maker. The agent can make decisions and interact within its environment. The main goal of the agent is to maximize its reward. For every action the agent takes there is a reward or punishment. The agent then learns from the reward or punishment to update its strategy. This can continue until the agent has found the optimal policy or the maximized reward strategy.

For each turn the agent takes it will:

1) Observe the current state
2) elect an action based on a policy
3) Receive a reward of punishment for said action
4) Collect this information to update its policy for the next turn

Environment

The environment is a key component of reinforcement learning. The environment is everything that the agent interacts with. For a reinforcement learning algorithm used in Finance this could be a simple ticker environment. For another situation, this can be a maze. The environment is an important part of training an agent and finding an optimized solution to a problem.

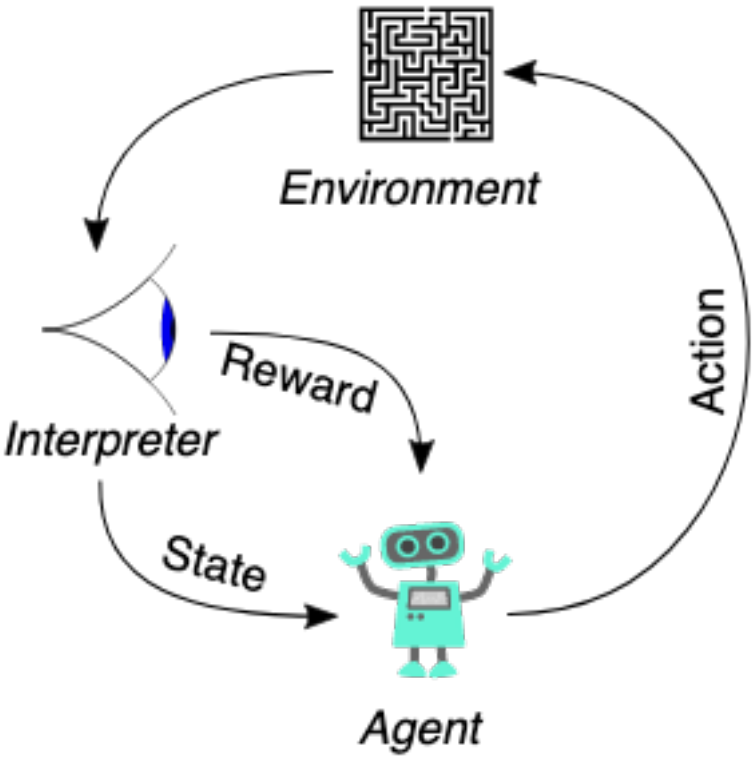


Figure 1: Example Image

State

A state is the specific situation in which the agent finds itself. The state is how the agent make decisions and it encapsulates all the relevant information that the agent needs to make informed decisions. There are different types of states. There is the *fully observable state*. In this scenario, the agent has complete information about the environment. A chessboard is a perfect example of this where the agent knows exactly where all the pieces are located on the board. The other type of state is a *partially observable state*. This is where the agent has limited information about the environment. This would be a situation of the game battleship. The agent would know where its pieces are located but not anything else.

There are also different types of state space. You can have *discrete state space* or *continuous state space*. In discrete space you have a finite number of distinct states. This is common with mazes, chess, and other games like that. Then you have continuous space. This is where the state space consists of an infinite number of possible states.

State transitions are another key aspect of states. State transitions describe how the environment evolves in response to the agents actions. *Deterministic transition* is described when the next state is uniquely determined by the current state and action. For example, in the game of chess, the agent would be able to move a piece in a configured fashion. There are rules that guide the movement of pieces and make the game predictable.*Stochastic Transition* is when the next state is determined through a probability rather than being predefined. This adds in an element of randomness to games. An example of this would be a card game, where cards are randomly chosen through a shuffled deck. The agents would continue to know its current state but the next state isn't clearly defined.

State in mathematical terms is s.

Action

Actions are what an agent can do in its state. In a maze environment this can be moving up, down, left, or right. In a finance environment, this can be buying or shorting a stock. The number of actions an agent has can simplify or complicate a model.

Action in mathematical terms is denoted by a

Reward

An important part of reinforcement learning are rewards and punishments. This is what directs our agent on what is good and what is bad. Rewards are simply feedback from the environment based on actions taken. Rewards can be derived using reward functions. Reward functions provide a scalar feedback signal based on state and action. An example of this would be the time it takes an agent to finish a maze or the amount of money an agent can make buy or shorting a stock. These are simple numeric values where the optimal policy gets you the largest number or the smallest number.

Reward in mathematical terms is denoted by r

Policy

Strategy used by the agents to determine next action based on current state. This is the strategy that the agent follows. It takes the states of the world and translates them to the actions that the agent will take. There are two different types of learning that come from policy, *On-policy* and *off-policy*. On-policy is when the agent develops a strategy (policy) from what it has learned. Off-policy develops a strategy (policy) from outside data. This can leverage a broader range of possible strategies than one agent can develop alone.

Policy is mathematical terms is denoted by π with optimal policy being denoted as $\pi(s(i), a(i))$ This can be futher explained into deterministic policy and stochastic policy as follows:

Deterministic Policy: $\pi(s) = \alpha$ Stochastic Policy: $\pi(\alpha|s) = Pr(\alpha|s)$

Value Function

A value function is a mathematical tool used to evaluate the quality of a state or state-action pair in terms of expected rewards.

Value Function can be denoted mathematically as $V^{\pi}(s)$

Defining the Value Function

In reinforcement learning, the overall objective is to maximize the long-term reward an agent receives and this can be demonstrated through a few key functions. G_t is the cumulative discounted reward from time t. G_t can be defined as

$$G_t = \sum(\gamma^k r_{t+k})$$

where k is iterations. γ is defined as the discount factor or at what level should we prioritize tomorrow's reward over today's reward and vice versa. γ weighs future rewards, r_t+_k. Continuing on from this, the state-value function $V^{\pi}(s)$ quantifies the expected return when starting in state s and following a policy π.

$$V^{\pi}(s) = \mathbb{E}[G_t|s_t = s]$$

This can answer the question, "How good is it to be in state s under policy π?" The action-value function $Q^\pi(s, a)$ can extend this idea to actions by measuring the expected return of taking a specific action a in state s and then following the policy. This can be defined as

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

This provides a basis to evaluate the desirability of actions given the current state.

Applications of Reinforcement Learning

Reinforcement learning has a variety of applications in the real world and in academia.

1) Robotics. Reinforcement learning can be used to teach a robot how to interact with its environment and perform certain tasks. This is popular in manufacturing.

2) Finance. Reinforcement learning is a popular tool to make trading strategies and sometimes train a bot to trade on the market without human input.

3) Pricing. Reinforcement learning can be used by companies like Doordash and Uber to solve issues over surge pricing. They can set up environments and determine how different people react in different situations and with different preferences.

4) Games. Reinforcement learning can be used in video games. Agents are able to be put into games and look for optimal strategies to play the game. This can be useful in training the computer to play against humans in the game.

Reinforcement Learning in Economics

Reinforcement learning is used in many sub fields of economics. Reinforcement learning is all about optimization. Thus it is incredibly efficient in applications of game theory. Reinforcement Learning can be used to solve Nash Equilibria of multi-agent games. This can be modeled with evolutionary dynamics for seeing how optimal solutions to games change overtime. These types of scenarios are scalable to handle large populations of agents. As mentioned above with the surge pricing example, that is an application of game theory that can be solved using reinforcement learning.

Exploration vs. Exploitation trade-off

Similar to that of the bias-variance trade off, in reinforcement learning we have the exploration-exploitation trade-off. *Exploration* is the agent searching for new actions to discover more about the environment it resides in. *Exploitation* is leveraging the knowledge the agent already has to try to maximize rewards from what it already knows.

If we favor exploration completely, our agent will never be able to use what it learned to as it will continually be searching for new actions. If we favor exploitation where our agent would never learn anything new and would only maximize what is already known about the environment it resides in.

Just as we try to try to optimize bias-variance trade off in other types of machine learning we need to optimize our exploration-exploitation trade off. This optimal bundle can be unique to a specific situation and there isn't one right bundle. There are different strategies for trying to find this optimal bundle.

Epsilon-Greedy Strategy

In this strategy we have probability (ϵ) which represents the probability that the agent will explore a random action. Then with probability (1 - ϵ) the agent exploits the best-known action. This can fluctuate overtime where ϵ starts high in the beginning and decreases as the agent gains a large base of knowledge.

Upper Confidence Bound (UCB)

The UCB strategy is designed to balance the exploration-exploitation trade off using a confidence interval to estimate the uncertainty of an action's reward. This works by estimating the expected reward for each action and then assigning a confidence bonus. The confidence bonus represents the uncertainty of confidence interval. The action that is selected is the action with the highest combined value between the expected reward and the confidence bonus assigned.

$$E(a) + c * \sqrt{\frac{ln(t)}{N(a)}}$$

Where: E(a) is the expected return of the action c is the parameters that controls the degree of exploration t is the total number of times any action has been taken N(a) is the number of times action a has been taken

$\sqrt{\frac{ln(t)}{N(a)}}$ is the confidence bonus term

Thompson Sampling

Thompson sampling is a Bayesian approach to the exploration-exploitation trade off. This approach models the uncertainty of the rewards using a probability distribution. This involves initializing a prior probability distribution that represents the initial belief about the reward for each action. Then at each time step there is a sample value taken from said probability distribution. The agent selects the action with the highest sampled value. After the action, the probability distribution is updated to account for the rewards of the action taken. This method helps converge to the optimal action as data is collected.

Algorithms

Q-Learning

Q-Learning is an algorithm that trains an agent to assign values to its possible actions based on its current state, without requiring a model of the environment. In this model free approach the agent improves its decision making by using a Q-table where it stores Q-values. Q-values represent the expected rewards for taking actions in a given state. Those Q-values will be saved in the Q-table. This method uses *episodes* to find an optimal policy. The episode ends when the agent reaches a terminal state. An example of this can in a maze environment the episode is one iteration of the agent solving its way through a maze. The terminal state is when the agent reaches the end point of the maze. The agent updates the Q-values using this formula.

$$Q(S, A) < -Q(S, A) + \alpha(r + \gamma * Q(S', A') - Q(S, A))$$

Where: S' is the next state the agent moves to A' is the best next action in state S' r is the reward received for taking action A in state S α is the learning rate or at what rate does the model learn.

An addition to Q-Learning is Deep Q-Networks. This method uses a neural network to approximate the Q-values.

Policy Gradient Methods

The Policy Gradient Theorem: The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ

This method aims to directly optimize the policy by gradient ascent. Instead of learning value functions, they learn a parameterized policy that maps states to actions.

This works by: 1) Start with an initial guess for the parameters 2) Use multi-variables calculus to compute the gradient of the function with respect to the parameters 3) update the parameters in the direction of the gradient

$$\nabla_\theta J(\theta) = \mathbb{E}\pi\Big[\nabla\theta \log \pi_\theta(a_t|s_t) G_t\Big]$$

- $J(\theta)$: Objective Function - $\pi_\theta(a_t|s_t)$: Policy (probability of taking action a_t in s_t, parameterized by θ - G_t : Return (cumulative discounted reward from time t) This acts as a weighting factor, meaning actions that lead to higher rewards are given more importance. θ is the parameter vector - $\log \pi_\theta(a_t|s_t)$ - scales probabilities - $\nabla\theta \log \pi_\theta(a_t|s_t)$ Tells us how sensitive the probability of an action is to change to policy parameter

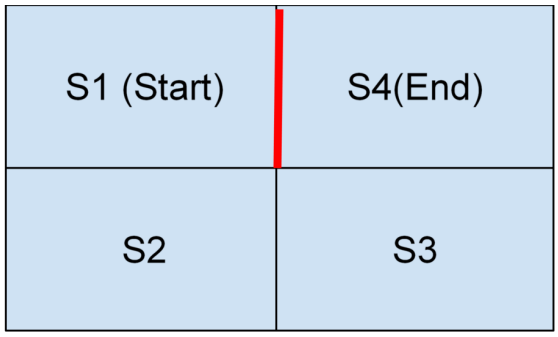
Methods that use the policy gradient method are *Actor-Critic*. This is an approach where the actor updates the policy parameters and the critic estimates the value function. *Trust Region Policy Optimization* which limits the policy from being updated too far from the old policy.

History and Development

Reinforcement learning comes from late 19th century psychology. This was demonstrated through humans and animals showing that both groups could learn through rewards and punishments. One famous example is Pavlov's bell ringing experiment. This is where he noticed that dogs could associate the ringing of a bell with food. This laid the ground work for classical conditioning.

Later on mathematicians used the framework of Markov Decision Processes (MDP) to develop an algorithm for agents in reinforcement learning. MDP is a framework to model decision making in situations where outcomes are unknown. This lead us to modern day reinforcement learning in which we have states, actions, transition functions, reward functions, optimal policies and parameters in which to tune our algorithms.

Gridworld Example



This is an example of a deterministic reinforcement learning game. This is a 2x2 Gridworld. In this game, the agent must make it from S1 to S4. There is a wall between S1 and S4 so the only way to get there is to move down, right, then up. This game is deterministic because whatever action the agent takes will have a set outcome. There is no randomness in this game.

```
library(MDPtoolbox)
#up action
up = matrix(c(1,0,0,0,
        1, 0, 0, 0,
        0, 0, 0, 1,
        0, 0, 0, 1),
       nrow=4,ncol=4,byrow=TRUE)

#down action
down=matrix(c(0, 1, 0, 0,
           0, 1, 0, 0,
           0, 0, 1, 0,
           0, 0, 1, 0),
         nrow=4,ncol=4,byrow=TRUE)

#Left Action
left=matrix(c( 1, 0, 0, 0,
         0, 1, 0, 0,
         0, 1, 0, 0,
         0, 0, 0, 1),
       nrow=4,ncol=4,byrow=TRUE)

#Right Action
right=matrix(c( 1, 0, 0, 0,
           0, 0, 1, 0,
           0, 0, 1, 0,
           0, 0, 0, 1),
         nrow=4,ncol=4,byrow=TRUE)

#Combined Actions matrix
Actions=list(up=up, down=down, left=left, right=right)

#2. Defining the rewards and penalties
Rewards=matrix(c( -1, -1, -1, -1,
              -1, -1, -1, -1,
              -1, -1, -1, -1,
              10, 10, 10, 10),
            nrow=4,ncol=4,byrow=TRUE)

#3. Solving the navigation
solver=mdp_policy_iteration(P=Actions, R=Rewards, discount = 0.1)

#4. Getting the policy

names(Actions)[solver$policy]
```

```
## [1] "down"  "right" "up"    "up"
```

As you can see from this example, the RL Agent solved the puzzle with the optimal policy of down, right, up, up. You did not need a reinforcement learning algorithm to solve this game as the solution is fairly intuitive. This game, however, is a great introduction to how reinforcement learning works.

Financial Trading Reinforcement Learning Example

For this example we are going to need three libaries. R6, dplyr, and quantmod. R6 will help us set up our environment, quantmod will bring in our financial data, and dplyr will help us sort the data easier.

```
library(quantmod)
library(dplyr)
library(R6)
```

First thing we need to do is pick the stock we want to trade. I picked a company called Weyhauser. This is a timber company that has been publically traded for decades. I chose this stock for two reasons. During the financial crisis the stock took a hit and has not recovered from it and that will make it harder for my RL bot to trade. The second reason is the timber industry suffers heavily from seasonality and I wanted to see if my bot could learn to exploit those seasonalities to make money.

Another part of this code is the dollar cap. I wanted to give my bot only $1000 to work with. Without this cap the bot overly buys to make money.

```
set.seed(12345)

# Download historical stock prices
getSymbols("WY", src = "yahoo", from = "2001-01-01", to = "2024-01-01")
stock_data <- Cl(WY) %>% as.data.frame()
stock_data$date <- row.names(stock_data)
colnames(stock_data) <- c("close", "date")
dollar_cap_value <- 10000
```

The next thing we need to do is to create an environment for our agent to live in. For this we created an R6class object that will take in public stock data. We need to have the data, the state, whether or not the bot is holding money, we need to record the actions taken, the total reward, the dollar cap amount defined above, and the available funds the bot has to buy or sell with.

There is a reset function below that. That will help us reset the environment in between episodes. For our actions, we had to force the bot through penalties to not buy when it didn't have money and not sell when it did not have stock to sell. The agent gets a reward if it buys and the next state the stock rises and sells if the next state is lower. The agent also has an opportunity to hold on to the stock which it is then granted the reward of the stock appreciating.

```
set.seed(12345)

StockEnv <- R6Class("StockEnv",
                    public = list(
                      data = NULL,
                      state = NULL,
                      holding = FALSE,
                      actions_taken = NULL,  # Track actions taken
                      total_reward = 0,  # Track total reward
                      dollar_cap = NULL,  # Dollar amount cap
                      available_funds = NULL,  # Available funds
                      initialize = function(stock_data, dollar_cap) {
                        if (is.null(stock_data) || nrow(stock_data) == 0) {
                          stop("Invalid stock data")
                        }
                        self$data <- stock_data
```

```
    self$state <- 1
    self$holding <- FALSE
    self$actions_taken <- data.frame(state = integer(), action = integer())  # Init
    self$total_reward <- 0  # Initialize total_reward
    self$dollar_cap <- dollar_cap  # Set dollar cap
    self$available_funds <- dollar_cap  # Initialize available funds
  },
  reset = function() {
    self$state <- 1
    self$holding <- FALSE
    self$actions_taken <- data.frame(state = integer(), action = integer())  # Rese
    self$total_reward <- 0  # Reset total_reward
    self$available_funds <- self$dollar_cap  # Reset available funds
    return(self$state)
  },
  step = function(action) {
    if (is.null(self$data) || nrow(self$data) == 0) {
      stop("Stock data not initialized")
    }
    next_state <- min(self$state + 1, nrow(self$data))
    reward <- 0

    if (action == 1) {  # Buy
      if (self$available_funds >= self$data$close[self$state]) {
        reward <- self$data$close[next_state] - self$data$close[self$state]
        self$holding <- TRUE
        self$available_funds <- self$available_funds - self$data$close[self$state]
      } else {
        reward <- -1  # Penalty for trying to buy without enough funds
      }
    } else if (action == 2) {  # Sell
      if (self$holding) {
        reward <- self$data$close[self$state] - self$data$close[next_state]
        self$holding <- FALSE
        self$available_funds <- self$available_funds + self$data$close[next_state]
      } else {
        reward <- -1  # Penalty for trying to sell when not holding
      }
    } else if (action == 3) {  # Hold
      reward <- self$data$close[next_state] - self$data$close[self$state]
    }

    self$total_reward <- self$total_reward + reward

    # Apply dollar cap
    if (self$total_reward > self$dollar_cap) {
      reward <- reward - (self$total_reward - self$dollar_cap)
      self$total_reward <- self$dollar_cap
    }

    self$state <- next_state
    done <- self$state == nrow(self$data)
    self$actions_taken <- rbind(self$actions_taken, data.frame(state = self$state, a
```

```
                        return(list(state = self$state, reward = reward, done = done))
                      }
                    )
)


# Create the environment instance
env <- StockEnv$new(stock_data, dollar_cap = dollar_cap_value)
```

Next we have to set in our parameters and how our agent is going to learn. Alpha is our learning rate. This means that through every episode that 10% of the old information will be updated by new information. The gamme or discount factor determines the importance of future rewards. Epsilon is our exploration rate. This comes from our exploration-exploitation trade off.

In this code below we also set up our Q-table. This will keep track of all our rewards for the agent to be able to look at when deciding what to do next.

```
set.seed(12345)

# Q-learning parameters
alpha <- 0.1  # Learning rate
gamma <- 0.9  # Discount factor
epsilon <- 0.1  # Exploration rate

# Initialize Q-table
n.states <- nrow(stock_data)
n.actions <- 3  # Buy, Sell, Hold
Q <- matrix(0, nrow = n.states, ncol = n.actions)
n.episodes <- 10
cumulative_rewards <- numeric(n.episodes)  # Store cumulative rewards for each episode
```

This is our training loop. This is going to let our agent run through the 20 years of stock data that was pulled from Yahoo finance and learn on it through 10 iterations. With each iteration the bot will learn from the last and update its strategy of buying and holding.

```
set.seed(12345)


# Training loop
n.episodes <- 10
for (episode in 1:n.episodes) {
  state <- env$reset()
  done <- FALSE
  total_reward <- 0  # Initialize total reward for the episode
  while (!done) {
    if (runif(1) < epsilon) {
      action <- sample(1:n.actions, 1)  # Explore
    } else {
      action <- which.max(Q[state, ])  # Exploit
    }
    result <- env$step(action)
    next_state <- result$state
    reward <- result$reward
```

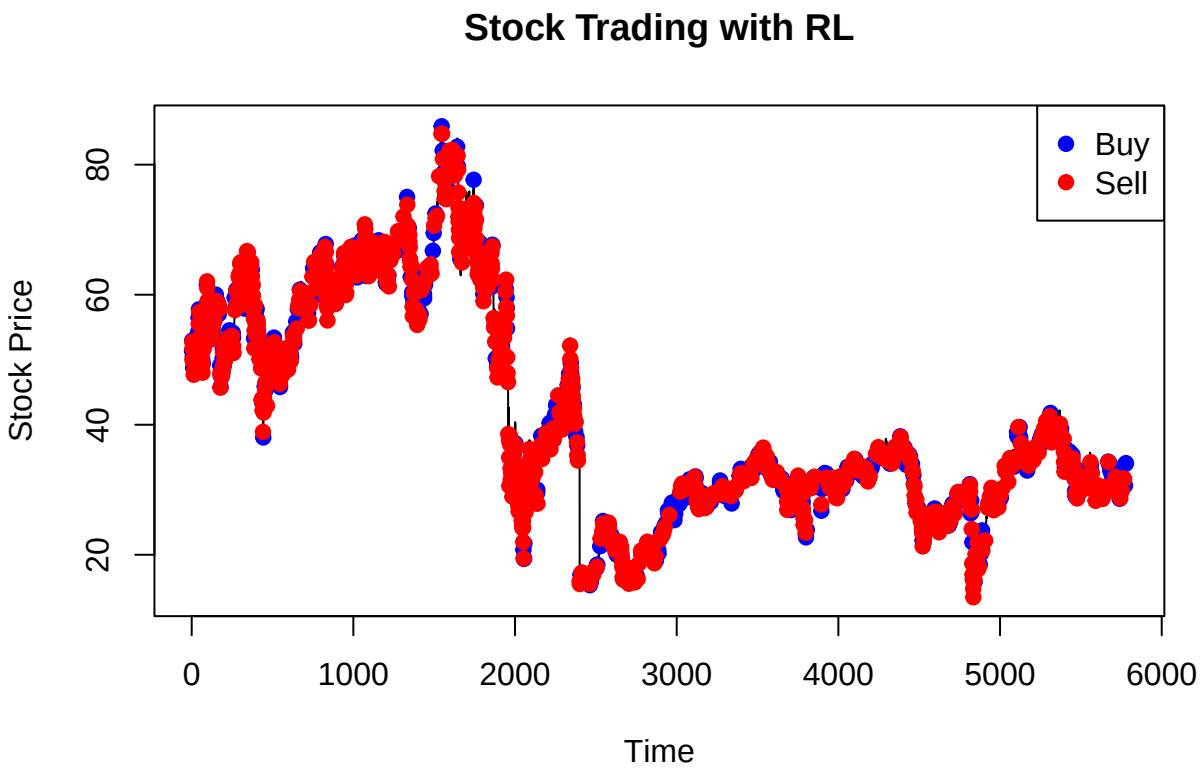
```
    done <- result$done
    # Update Q-value
    Q[state, action] <- Q[state, action] + alpha * (reward + gamma * max(Q[next_state, ]) - Q[state, act
    state <- next_state
    total_reward <- total_reward + reward  # Update total reward
  }
  cumulative_rewards[episode] <- total_reward  # Store cumulative reward for the episode
}
```

The final code chunk is just extracting all the information from the episodes to plot and figure out the best our agent did and the worst our agent did.

```
set.seed(12345)


# Extract buy and sell points
buy_points <- env$actions_taken %>% filter(action == 1)
sell_points <- env$actions_taken %>% filter(action == 2)
hold_points <- env$actions_taken %>% filter(action==3)


# Plot stock prices
plot(stock_data$close, type = "l", col = "black", xlab = "Time", ylab = "Stock Price", main = "Stock Tra
points(buy_points$state, stock_data$close[buy_points$state], col = "blue", pch = 19)  # Buy points
points(sell_points$state, stock_data$close[sell_points$state], col = "red", pch = 19)  # Sell points
legend("topright", legend = c("Buy", "Sell"), col = c("blue", "red"), pch = 19)
```




```
print(max(cumulative_rewards))
```

```
## [1] 474.2675
```

```
print(min(cumulative_rewards))
```

```
## [1] -5029.535
```

In this example you can see from the plot when our agent decided to buy and when it decided to sell. This is just one plot from one iteration of the 10 iterations we put the agent through. You can also see from the outputs above that in the best of the 10 iterations our RL agent made us roughly 470 dollars and in the worst iteration our agent lost us roughly 5000 dollars.