AI PROJECT
Dino Game Playing Using Reinforcement Learning

Presented by:-

- Arzoo Jangra - 6
- Dhirender Kumar - 10
- Vidhi Khare - 54

Submitted to:-
Dr. Punam Bedi

Dino Run or **T-Rex** run is an endless runner game in Chrome browser which is available to play when you’re offline aka ‘**the game you don’t usually like to see**’.

Technologies Used

Python

- Python Selenium Package
- Numpy, Matplotlib, Pandas, Seaborn

Reinforcement learning

- Q-Learning

Git/Github

- **NumPy** can be used to perform a wide variety of mathematical operations on arrays.
- **Matplotlib** is a cross-platform, data visualization and graphical plotting library for Python.
- **Pandas** is a Python library for data analysis.
- **Seaborn** is a library in Python predominantly used for making statistical graphics.
- **Selenium**, a popular browser automation tool, was used to send actions to the browser and get different game parameters like current score.

Supervised, Unsupervised, and Reinforcement Learning: What are the Differences?

Difference #1: Static Vs. Dynamic

The goal of supervised and unsupervised learning is to search for and learn about patterns in training data, which is quite static. RL, on the other hand, is about developing a policy that tells an agent which action to choose at each step — making it more dynamic.

Difference #2: No Explicit Right Answer

In supervised learning, the right answer is given by the training data. In Reinforcement Learning, the right answer is not explicitly given: instead, the agent needs to learn by trial and error. The only reference is the reward it gets after taking an action, which tells the agent when it is making progress or when it has failed.

Difference #3: RL Requires Exploration

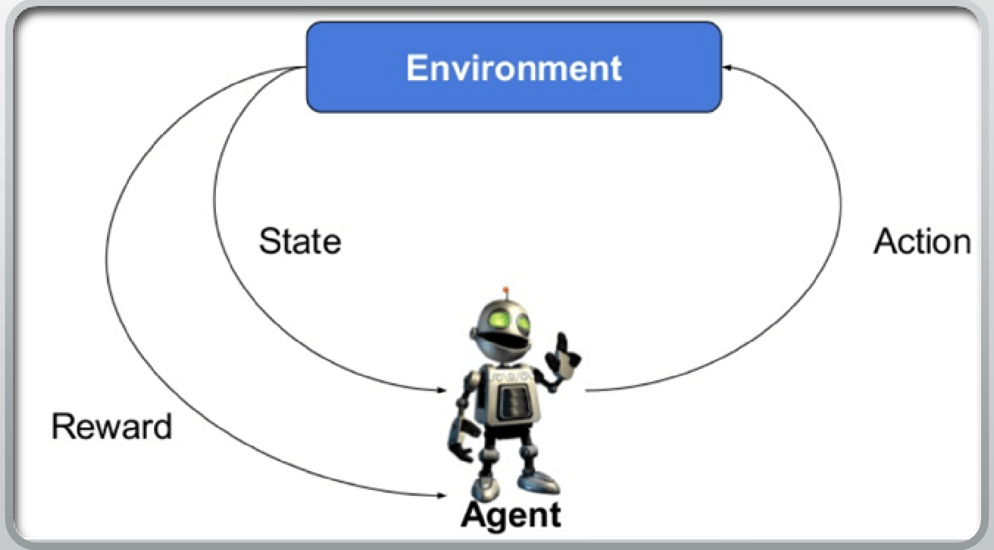
A Reinforcement Learning agent needs to find the right balance between exploring the environment, looking for new ways to get rewards, and exploiting the reward sources it has already discovered. In contrast, supervised and unsupervised learning systems take the answer directly from training data without having to explore other answers.

Difference #4: RL is a Multiple-Decision Process

Reinforcement Learning is a multiple-decision process: it forms a decision-making chain through the time required to finish a specific job. Conversely, supervised learning is a single-decision process: one instance, one prediction.

Reinforcement Learning

- **Reinforcement Learning** is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions. For each good action, the agent gets positive feedback, and for each bad action, the agent gets negative feedback or penalty.



Elements of Reinforcement Learning

Reinforcement Learning has **four** essential elements:

- **Agent:** The program you train, with the aim of doing a job you specify.
- **Environment:** The world, real or virtual, in which the agent performs actions.
- **Action:** A move made by the agent, which causes a status change in the environment.
- **Rewards:** The evaluation of an action, which can be positive or negative.

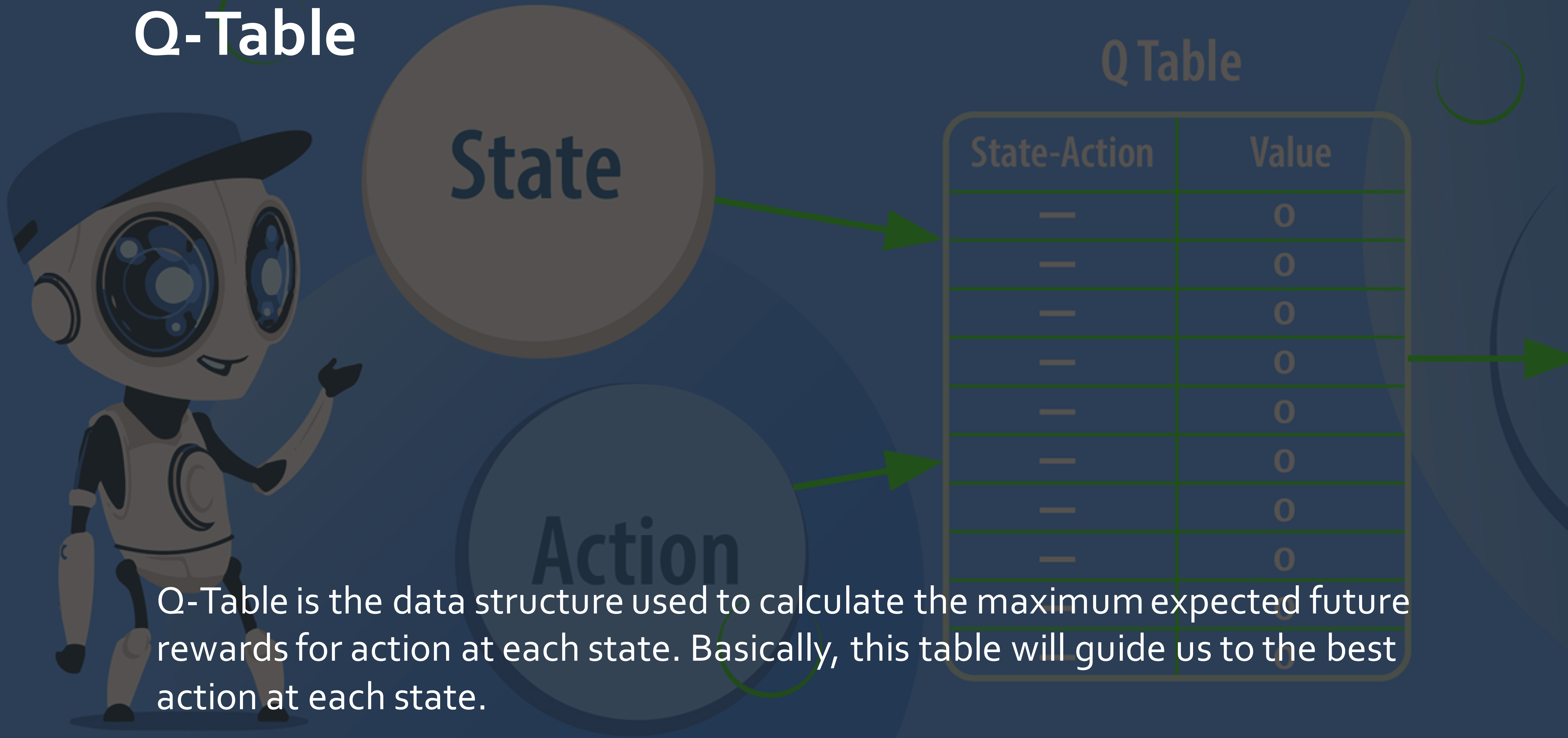
Q-Learning

- Q-learning is a popular **values-based** reinforcement learning algorithm based on the **Bellman equation**.
- **The main objective of Q-learning is to learn the policy which can inform the agent that what actions should be taken for maximizing the reward under what circumstances.**
- It is an **off-policy RL** that attempts to find the best action to take at a current state.
- The ‘Q’ in Q-learning stands for quality. Quality here represents how useful a given action is in gaining some future reward.

Elements of Reinforcement Learning According to our project

- **Agent:** Dino
- **Environment:** Web Browser
- **Action:** Jump & Duck
- **Rewards:** 15 for not getting out
- **Penalty:** -1000 for getting out

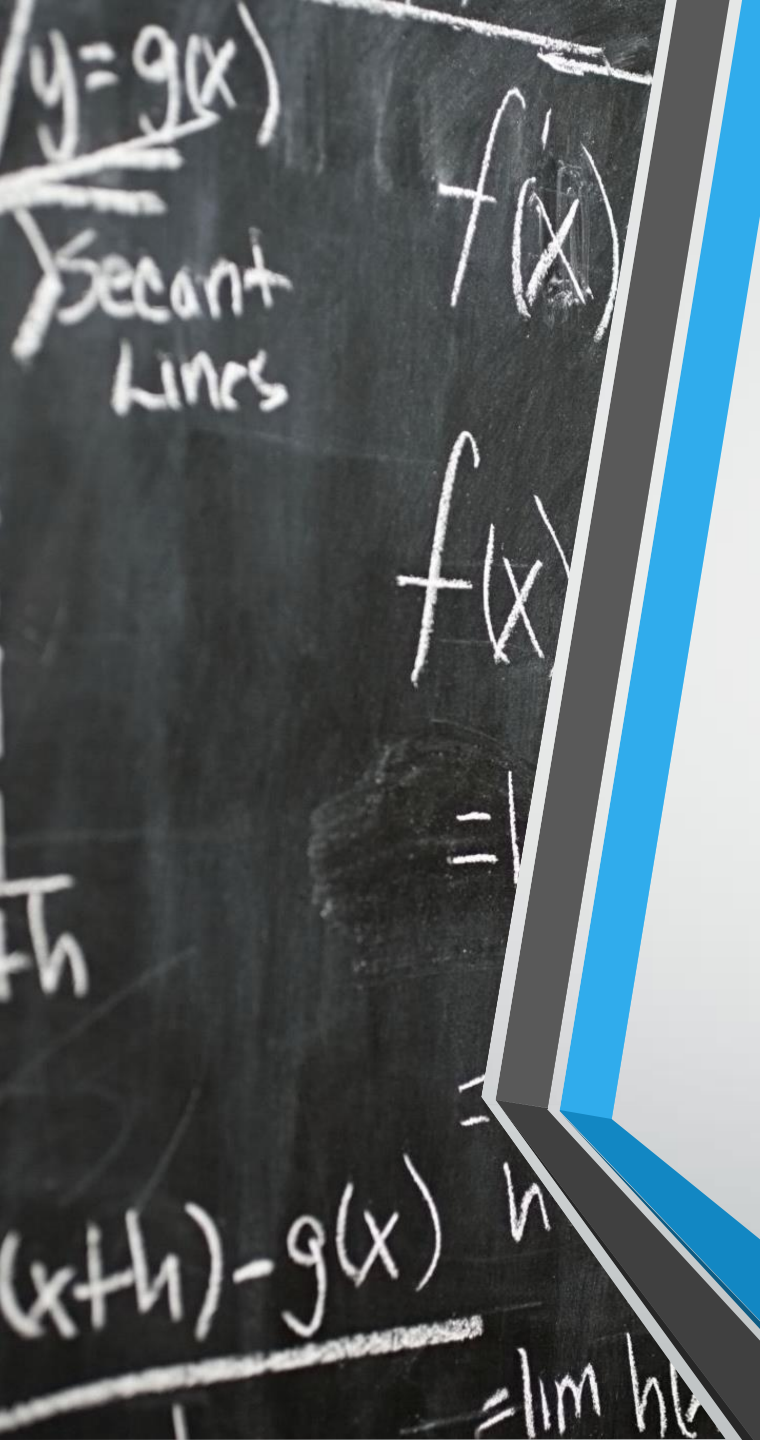
Q-Table



Q-Table is the data structure used to calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state.

State						Action		
Dino Information		Obstacle Information			Current speed of game	Do Nothing	Jump	Duck
X_Coordinate	Y_Coordinate	X_Coordinate	Y_Coordinate	Width				
45	36	80	33	50	8	3	2.5	-1
50	34	56	45	65	8	1	3	-4
55	39	67	34	70	8	2	5.1	-2
---	---	---	---	---	---	---	---	---

Q- Table According to our project



Maths behind Q-Learning

The goal of the agent in Q-learning is to maximize the value of Q-function [Q(s, a)], which uses the Bellman equation and takes two inputs state(s) and action(a).

Notations:

s = A particular state

a = Action

s' = State to which user goes from s

ϒ = Discount factor

α = Learning rate

R(s, a) = Reward function which takes a state s and action a and outputs a reward value.

Q'(s', a') = Expected future reward.

Q-function

$$\underbrace{\boxed{\text{New } Q(s,a)}}_{\substack{\text{New Q value}\\ \text{for the}\\ \text{state and action}}} = \underbrace{\boxed{Q(s,a)}}_{\text{Current Q values}} + \overbrace{\boxed{\alpha}}^{\text{Learning Rate}} \left[\underbrace{\boxed{R(s,a)}}_{\substack{\text{Reward for taking an}\\ \text{action in a state}}} + \overbrace{\boxed{\gamma}}^{\text{Discount Rate}} \underbrace{\boxed{\max Q'(s',a')}}_{\substack{\text{Maximum expected}\\ \text{future reward}}} - \underbrace{\boxed{Q(s,a)}}_{\text{Current Q values}}\right]$$

Exploration vs Exploitation

Exploration vs Exploitation problem arises when our model tends to stick to same actions while learning, in our case the model might learn that jumping gives better reward rather than doing nothing and in turn apply an always jump policy. However, we would like our model to try out random actions while learning which can give better reward. We introduce ε, which decides the randomness of actions. We gradually decay its value to reduce the randomness as we progress and then exploit rewarding actions.

Credit Assignment Problem

Credit Assignment problem can confuse the model to judge which past action was responsible for current reward. Dino cannot jump again while mid-air and might crash into a cactus, however, our model might have predicted a jump. So the negative reward was in fact a result of previously taken wrong jump and not the current action. We introduce Discount Factor γ, which decides how far into the future our model looks while taking an action. Thus, γ solves the credit assignment problem indirectly. In our case the model learned that stray jumps will inhibit it's ability to jump in the future when we set $\gamma=0.99$

Procedure

- Game Initialization
- Initialize the Variables and Q-Table
- Start game playing
- Get current state of dino
- Check for the current state in the Q-Table
  - If current state is in Q-Table get the Q-Values for the given state
  - Else add this new state to the Q-Table and randomly generate its Q-Values
- Based on the Q-Values take the action with highest Q-Value
- Based on the result of the action update the Q-Values for the current (state, action).

```python
def startGame():
    driver.find_element(By.TAG_NAME, "body").send_keys(Keys.SPACE)


def initialize():
    s = Service('/home/the-techie/Desktop/chromedriver')

    options = webdriver.ChromeOptions()
    options.add_argument("--disable-web-security")
    driver = webdriver.Chrome(service=s, options=options)
    driver.maximize_window()

    try:
        driver.get("chrome://dino/")
    except:
        pass

    return driver


def addNewState(state):
    q_table[state] = np.round(np.random.rand(3), decimals = 2)
```

Game Initialization

Initialize the Variables and Q-Table

```
import numpy as np
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait

scores = []
q_table = dict()
reward = 15
out = -1000
learning_rate = 0.1
discount = 0.99
episodes = 20000
driver = None

```

Start game playing

```python
def restart():
    driver.execute_script("Runner.instance_.restart()")


def startGame():
    driver.find_element(By.TAG_NAME, "body").send_keys(Keys.SPACE)
```

Get current state of Dino

```
def getState():
    x_pos = 0
    y_pos = 0
    obs_pos = []
    curr_speed = float(driver.execute_script("return Runner.instance_.currentSpeed"))

    if round(curr_speed % 1, 1) >= 0.5:
        curr_speed = int(curr_speed) + 0.5
    else:
        curr_speed = float(int(curr_speed))

    x_pos = driver.execute_script("return Runner.instance_.tRex.xPos")
    y_pos = driver.execute_script("return Runner.instance_.tRex.yPos")
    isJumping = driver.execute_script("return Runner.instance_.tRex.jumping")

    num_obstacles = int(driver.execute_script("return Runner.instance_.horizon.obstacles.length"))
    if num_obstacles > 0:
        a = min(150, driver.execute_script("return Runner.instance_.horizon.obstacles[0].xPos")//3)
        b = driver.execute_script("return Runner.instance_.horizon.obstacles[0].yPos")//2
        width = driver.execute_script("return Runner.instance_.horizon.obstacles[0].width")

        obs_pos = [a, b, width]

    res = tuple([x_pos, y_pos, isJumping, round(curr_speed, 1)] + obs_pos)

    return res

```

```python

def addNewState(state):
    q_table[state] = np.round(np.random.rand(3), decimals = 2)

def getQvalues(curr_state):
    try:
        p = q_table[curr_state]

    except:
        addNewState(curr_state)
        p = q_table[curr_state]

    return p

```

Get the Q-Value for the current state

Based on the Q-Values take the action with highest Q-Value

```python
# 0: Nothing
# 1: JUMP
# 2: DUCK

def performAction(choice):
    if choice == 1:
        driver.find_element(By.TAG_NAME, "body").send_keys(Keys.ARROW_UP)
    elif choice == 2:
        driver.find_element(By.TAG_NAME, "body").send_keys(Keys.ARROW_DOWN)

```

```python
p = getQvalues(curr_state)

action = np.argmax(p)

performAction(action)

```

Based on the result of the action update the Q-Values for the current (state, action)

```python
        # updating ******************************
        isPlaying = driver.execute_script("return Runner.instance_.playing")

        if isPlaying:
            reward = 15

        else:
            reward = -1000

        current_qsa = p[action]
        next_state = getState()
        next_state_qs = None

        try:
            next_state_qs = q_table[next_state]

        except:
            addNewState(next_state)
            next_state_qs = q_table[next_state]

        max_future_q = np.max(next_state_qs)

        new_q = (1 - learning_rate ) * current_qsa + learning_rate * (reward + discount * max_future_q)

        p[action] = new_q

        q_table[curr_state] = p

        curr_state = next_state
```

Week 2 updates

(17-01-2022 to 23-01-2022)

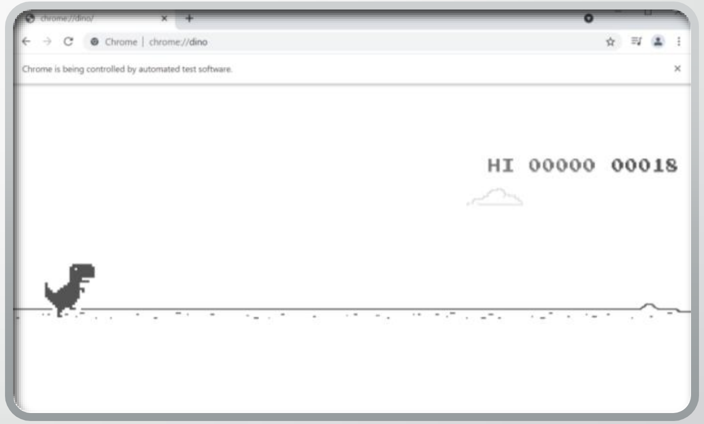
State							Action		
Dino Information			Current speed of game	Obstacle Information			Do Nothing	Jump	Duck
X-Coordinate	Y-Coordinate	Is Jumping (T/F)		X-Coordinate	Y-Coordinate	Width			
0	82	F	6				1.96	0.36	0.39
23	93	F	6.5	150	52	51	0.62	2.36	0.87
23	93	F	8	78	52	51	2.19	0.21	0.27
23	70	T	6	14	52	51	0.51	0.1	0.2
----	----	----	----	----	----	----	----	----	----

Explanation

- We are performing a certain number of episodes in one run of the program.
- At start of the game, we get the initial state of the Dino and the Obstacles(if present). Since in the image it can be seen that there is no obstacle present so our program only returns the state of the Dino (i.e. it's x-coordinate, y-coordinate, is it jumping or not, current speed of the game).

0	82	T	6			

- After getting the state, we retrieve the current score from the browser and while the score is less than 35, our Dino will do nothing.



- If score>35 and our Dino is still playing then we try to get the Q-Value for the current state using **getQvalues()** method.
- If current state of our Dino is not present in the Q-Table then we will add that state into the Q-Table by generating random Q-value for that state and generated Q-values would be returned by our method.
  - In our case it returns 0.45(for Nothing), 0.36(for Jump), 0.39(for Duck)
  - Current State ->

0	82	T	6				0.45	0.36	0.39

- If the current state is already present in the Q-Table then the **getQvalues()** would simply retrieve the Q-values for the current state from the Q-Table and return it.
- Further the Dino would perform action based on the action with highest Q-values.
  - In our case the max Q-value is 0.45 (Do Nothing - Index 0)

- After taking the action we check if our Dino is still playing or not.
  - If alive we give Dino reward of 15
  - Else a penalty of –1000

<u>If alive (Reward = 15)</u>

- We update the Q-value of the particular action taken for the current state to incur the effect of reward.
- To calculate the updated Q-value we retrieve the state in which Dino is after performing the action.
- We retrieve the next state, Q-values for the next state(next_state_qs) and also the maximum of the Q-values of this next state(max_future_q).
  - Next_state

23	93	F	6 .5	150	52	51	0.62	0.95	0.87

  - Next_state_qs = [0.62, 0.95, 0.87]
  - Max_future_q = 0.95 for Jump

Calculation to Update Q-Value, If Dino is Still playing

- Now we calculate the updated Q-value using the Q-Function
- new_q = [(1 - learning_rate) * current_qsa] +

  [learning_rate * (reward + discount * max_future_q)]

  - new_q = [(1 - 0.1) * 0.45] + [0.1 * (15 + 0.99* 0.95)]
  - new_q = [0.99*1.27] + [0.1*15.9405]
  - new_q = 0.4455 + 1.59405
  - new_q = 2.03
  - Now we update this calculated new_q in the q_table for the current state and for the action that was taken (I.e. Do Nothing)

Current State Initial

0	82	T	6				0.45	0.36	0.39

Current State Updated

0	82	T	6				2.03	0.36	0.39

Else (Reward = -1000)

- Here the reward represents the penalty given to the Dino.
- Penalty is given if the Dino gets out in the game.
- We update the Q-value of the particular action taken for the current state to incur the effect of penalty.
- To calculate the updated Q-value we retrieve the state in which Dino is after performing the action.
- We retrieve the next state, Q-values for the next state(next_state_qs) and also the maximum of the Q-values of this next state(max_future_q).
  - Next_state

23	93	F	6 .5	150	52	51	0.62	0.95	0.87

  - Next_state_qs = [0.62, 0.95, 0.87]
  - Max_future_q = 0.95 for Jump

Calculation to Update Q-Value, If Dino is Still playing

- Now we calculate the updated Q-value using the Q-Function
- new_q = [(1 - learning_rate) * current_qsa] +

  [learning_rate * (reward + discount * max_future_q)]

  - new_q = [(1 - 0.1) * 0.45] + [0.1 * (-1000 + 0.99* 0.95)]
  - new_q = [0.99*1.27] + [0.1*(-940.5)]
  - new_q = 0.4455 - 94.05
  - new_q = -93.6045
  - Now we update this calculate new_q in the q_table for the current state and for the action that was taken (I.e. Do Nothing)

Current State Initial

0	82	T	6				0.45	0.36	0.39

Current State Updated

0	82	T	6				-93.60	0.36	0.39

- Further if Dino is alive we would make next state as our current state to work over the next state that we are in after taking the action.
- We would go on performing the method listed in previous slides for all the upcoming states and fill up our Q-Table simultaneously to train our Dino.
- If Dino gets out, a different episodes starts with the same process listed but with a filled Q-Table which it uses to train Dino.
- After each episode we are storing the score that is generated and printing the same.

- As we can see in the image that there is no obstacle on the screen in the very beginning of the game that's why the obstacle info part of the first row of our Q-table is empty.

0	82	T	6				1.96	0.36	0.39

- Now in the picture below we can see the obstacle arriving towards our Dino and hence the second row of our Q-table provides the information about the obstacle as well which helps it to perform action accordingly.

23	93	F	6.5	150	52	51	0.62	2.36	0.87

Training in batches

- Previously, we had to train the model in one go.
- Now, with the use of json file, we are able to train our model in batches and on multiple systems.

Json File

This file contains our Q-table for all the sessions which helps us to start a next session with this filled Q-table instead of starting with an empty Q-table which also helps us to train our model in different batches on different systems.


```
{"0 82 True 6 ": [1.96, 0.36, 0.39], "23 93 False 6.5 150 52 51": [0.62, 2.36, 0.87], "23 93 False 6.5 78 52 51": [2.19, 0.21, 0.27],
 "23 93 False 6.5 74 52 51": [2.14, 0.46, 0.39], "23 93 False 6.5 70 52 51": [0.69, 0.54, 2.27], "23 93 False 6.5 63 52 51":
[0.04, -99.33, 0.3], "23 64 True 6.5 -2 52 51": [0.58, 0.71, 0.64], "23 93 False 6 ": [13.75, 0.32, 0.53], "23 93 False 6 92 45 25":
[0.18, -99.27, 1.84], "23 80 True 6 16 45 25": [0.97, 0.89, 0.29], "23 93 False 6 90 45 25": [0.71, 0.33, 2.41], "23 93 False 6 70 45 25":
[0.36, 3.52, 0.06], "23 93 False 6 99 52 51": [0.52, -99.04, 0.88], "23 93 False 6 18 52 51": [-99.16, 0.11, -99.18], "23 93 False 6 89 52
[0.05, -99.12, 0.47], "23 70 True 6 14 52 51": [0.51, 0.1, 0.2], "23 93 False 6 90 52 17": [3.77, 0.85, 0.2], "23 93 False 6 78 52 17":
[0.6, 2.44, 0.66], "23 93 False 6 80 45 25": [0.69, -99.04, 0.74], "23 53 True 6 8 45 25": [0.66, 0.21, 0.44], "23 93 False 6 92 52 51":
[0.21, -99.66, 0.12], "23 78 True 6 15 52 51": [0.41, 0.57, 0.52], "23 93 False 6 91 45 25": [0.39, 0.65, 2.23], "23 93 False 6 62 45 25":
[2.03, 0.11, 0.14], "23 93 False 6 48 45 25": [2.19, 0.42, 0.5], "23 93 False 6 34 45 25": [0.41, 0.26, -99.36], "23 93 False 6 20 45 25":
```


Json file snippet

Week 3 updates

(14-02-2022 to 20-02-2022)

Training Phase

- What are the parameter changes done for training to get the best results?
  - We are changing the learning rate in our program for training in different phases. (0.1 to 0.6)

Why we choose different values of learning rate for training ?

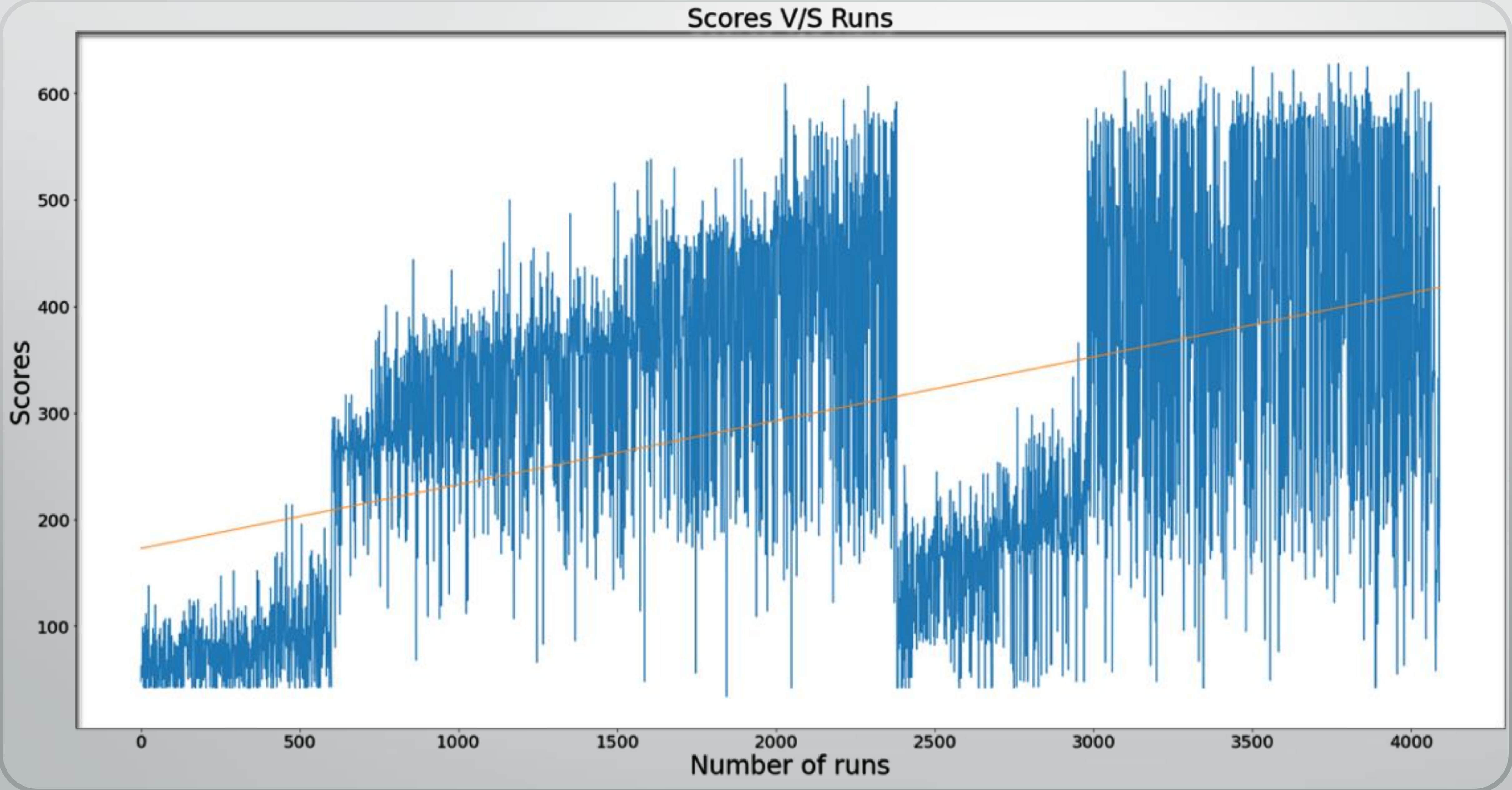
new_q = [(1 - learning_rate) * current_qsa] +

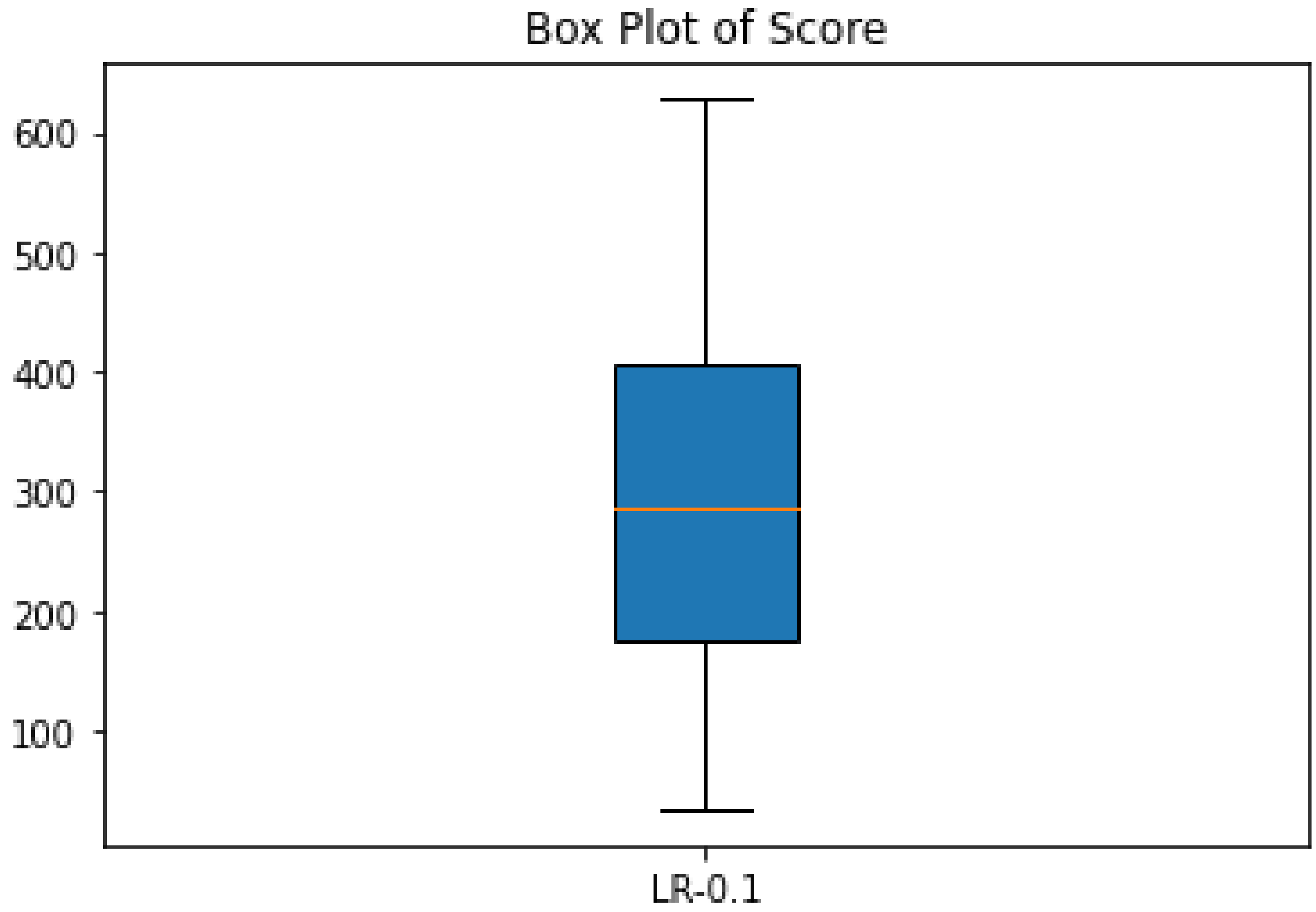
[learning_rate * (reward + discount * max_future_q)]

Since it can be seen that learning rate is making a tradeoff between our current q-value and future q-value, thereby to get the effect of learning rate on both the values simultaneously, we chose learning rate to vary between 0.1 and 0.6 for our training.

Visualizations

1st Parameter (Learning Rate - 0.1)

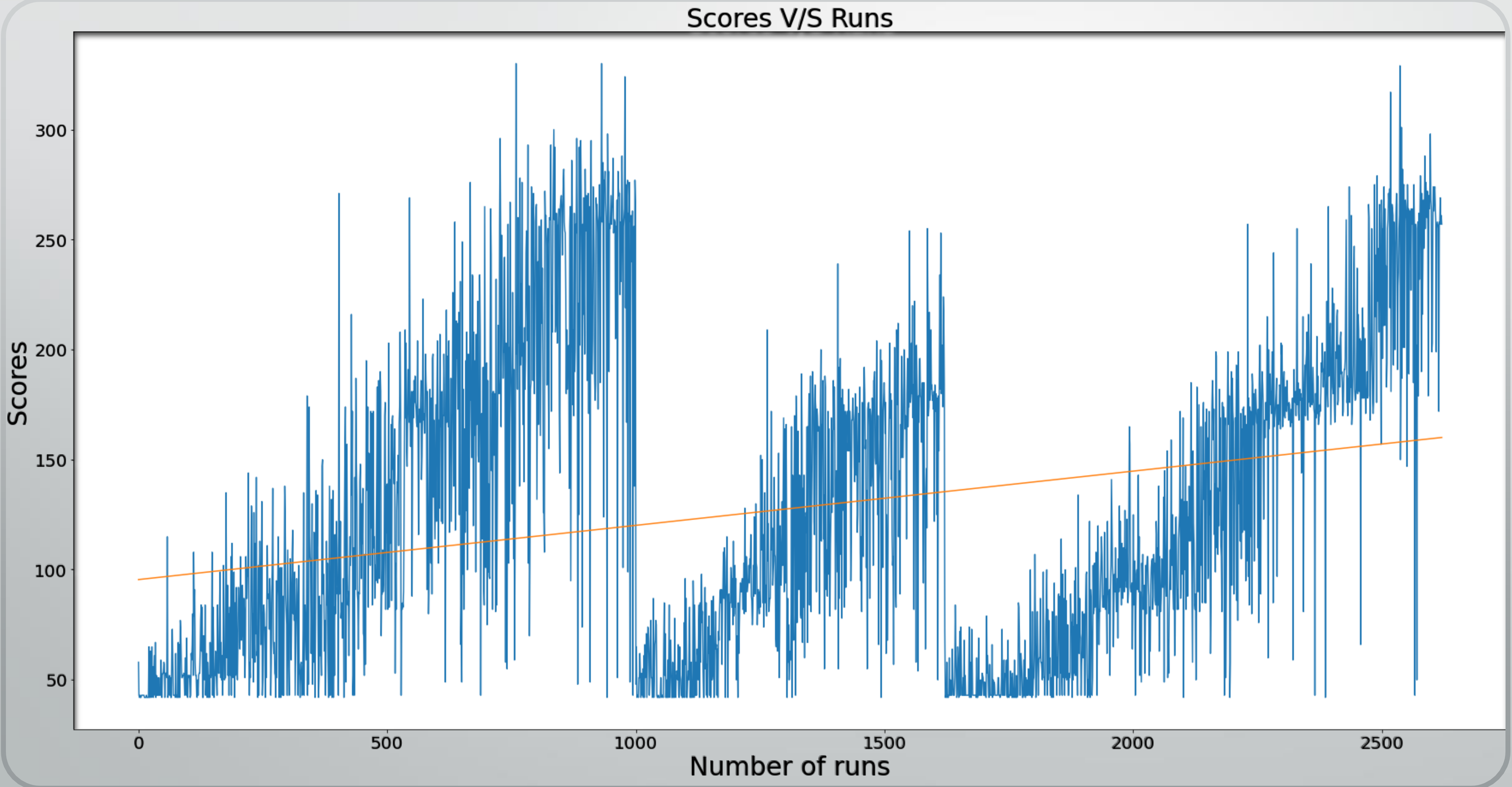


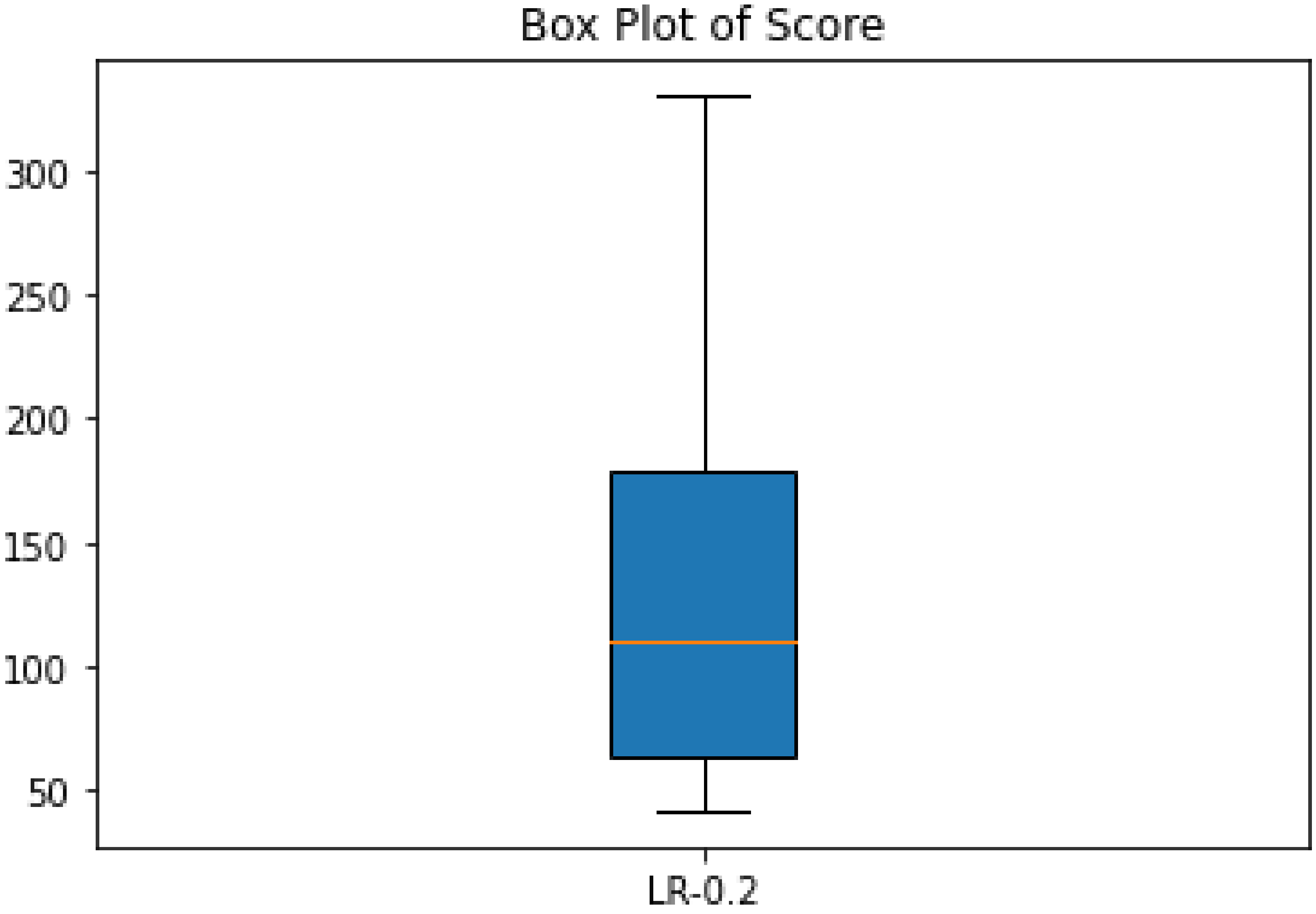


Five-number summary

- First_quartile: 174.0
- Max: 628
- Median: 286.0
- Min: 34
- Third_quartile: 405.0

2nd Parameter (Learning Rate - 0.2)

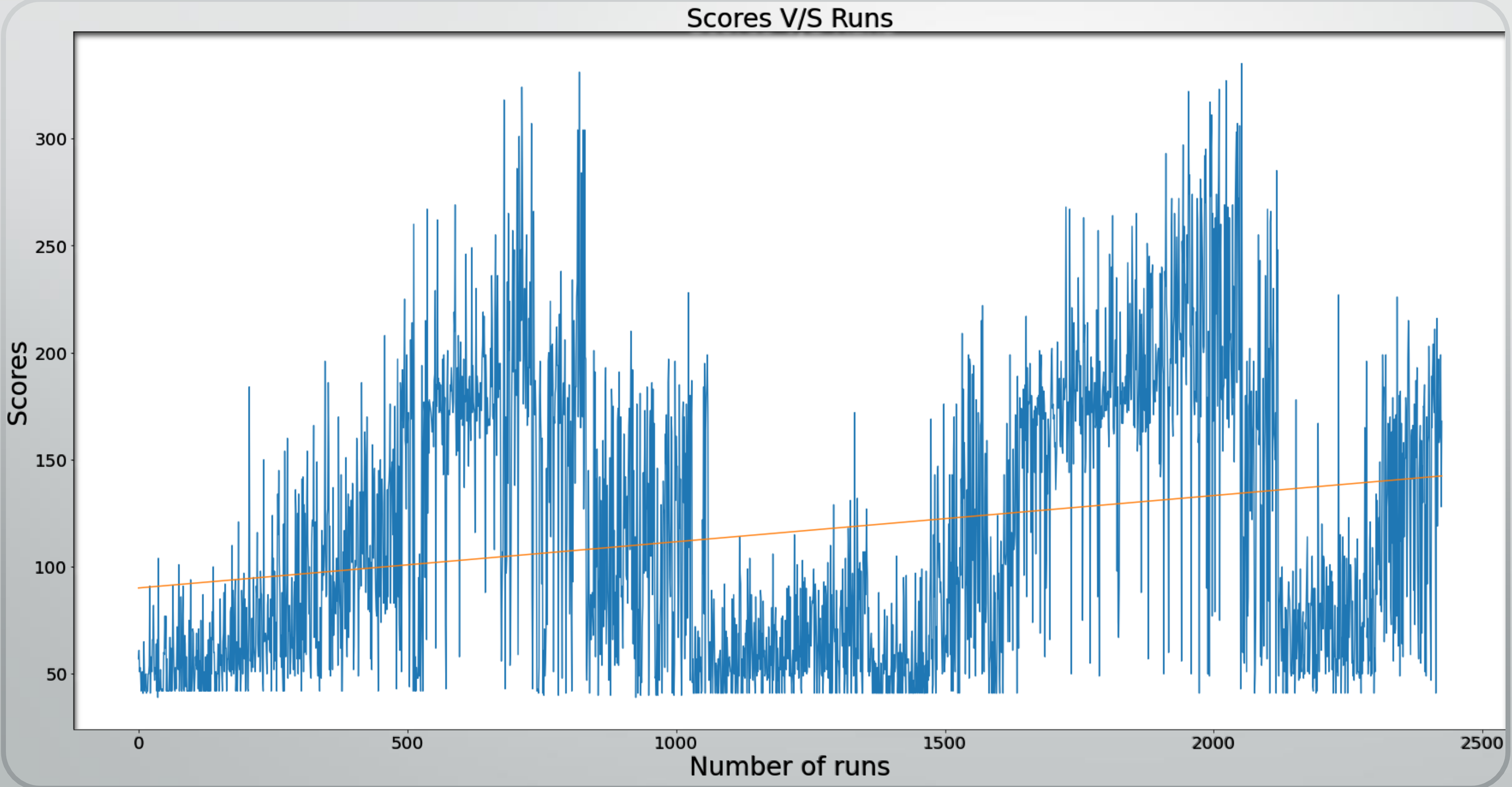


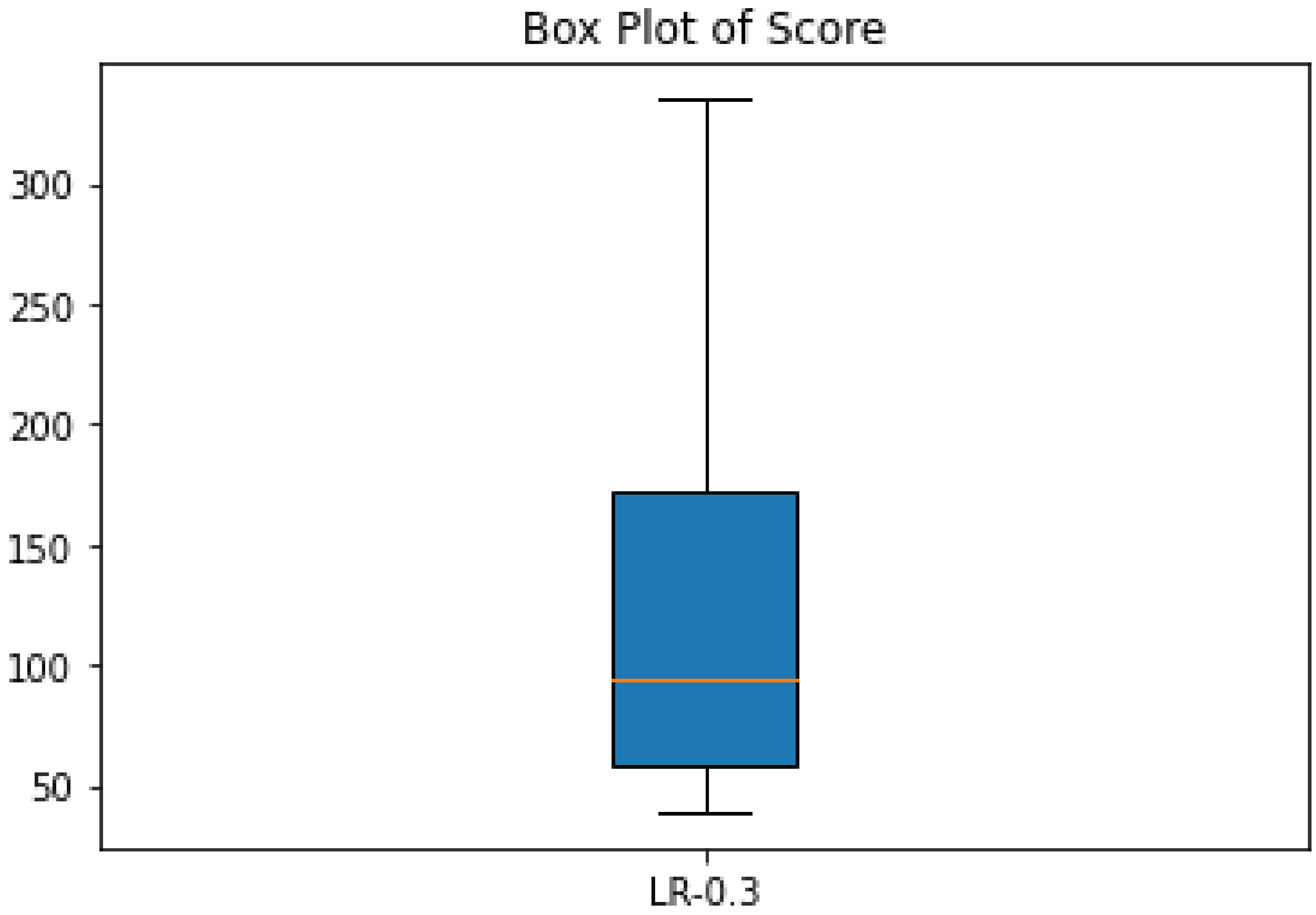


Five-number summary

- First_quartile: 64.0
- Max: 330
- Median: 110.0
- Min: 42
- Third_quartile: 179.0

3rd Parameter (Learning Rate - 0.3)

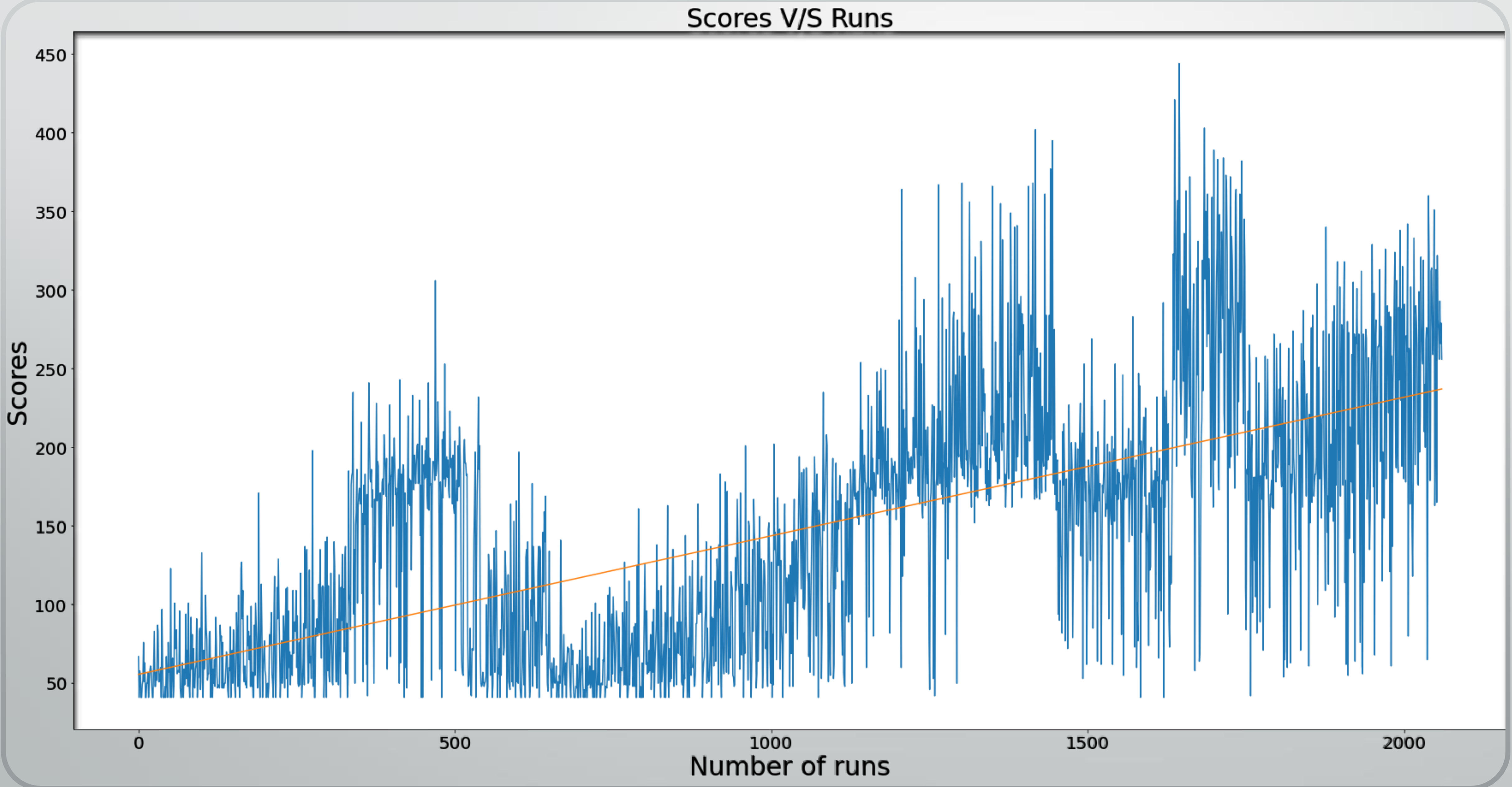


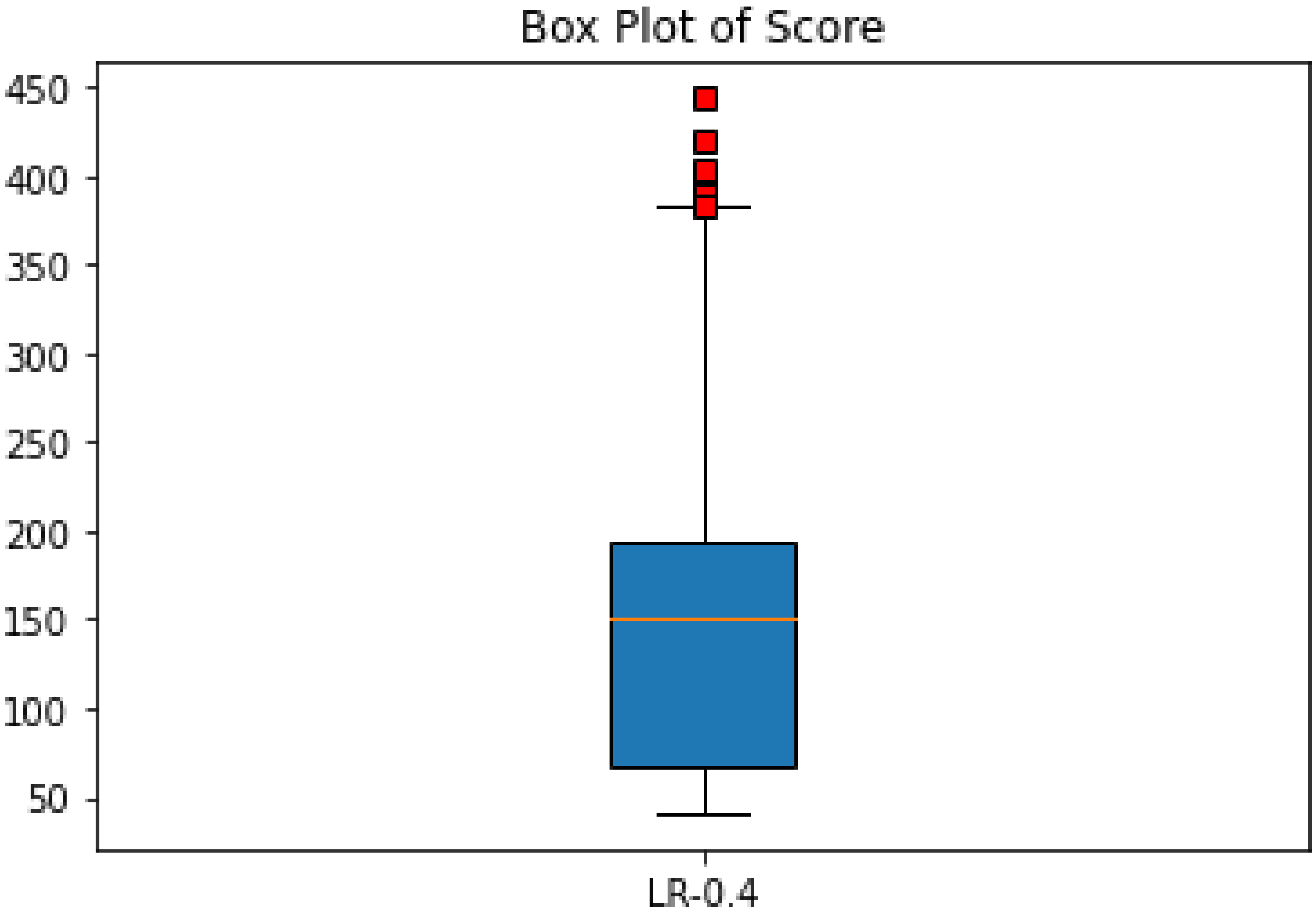


Five-number summary

- First_quartile: 58.0
- Max: 335
- Median: 95.0
- Min: 39
- Third_quartile: 172.0

4th Parameter (Learning Rate - 0.4)

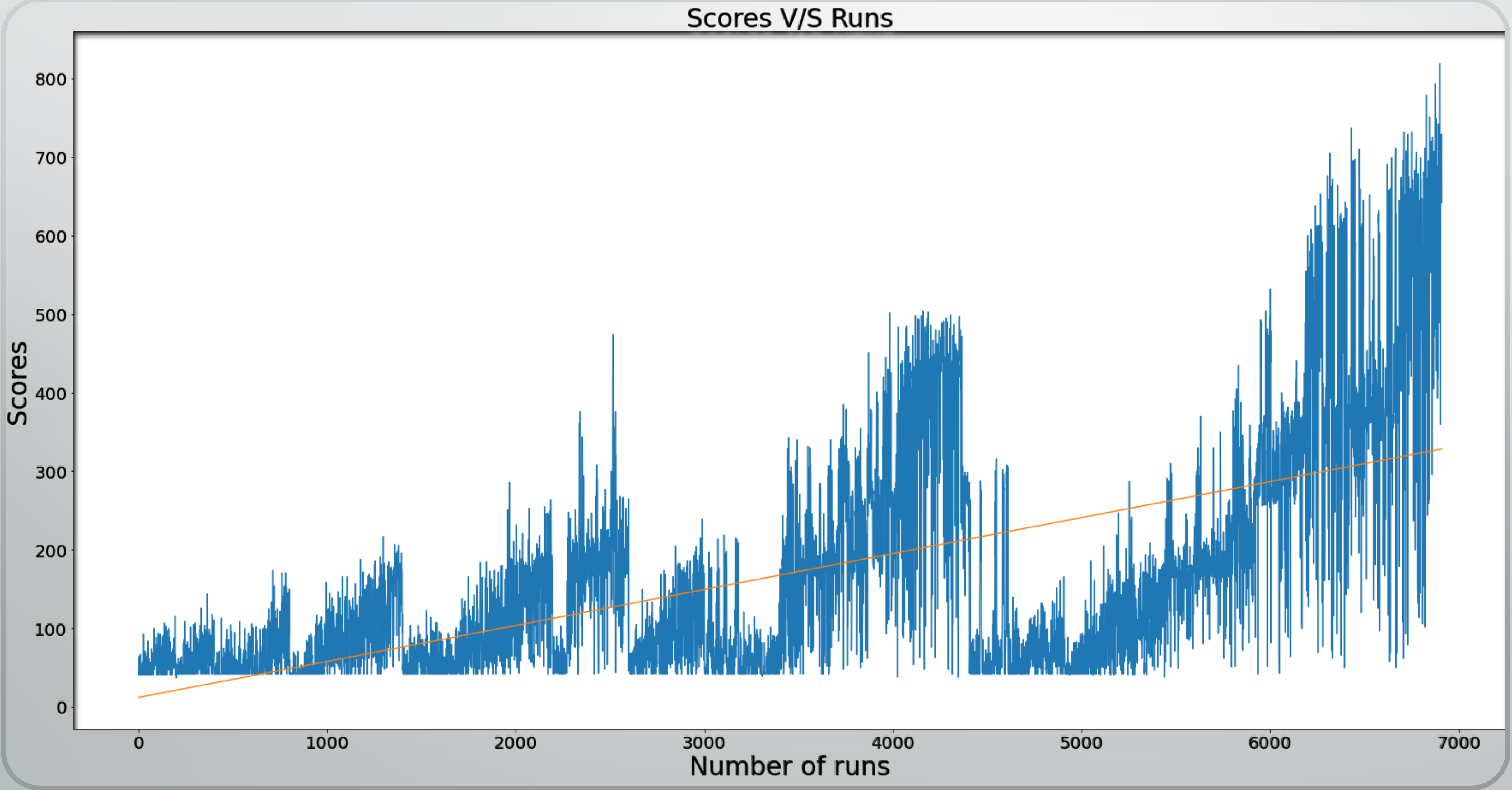


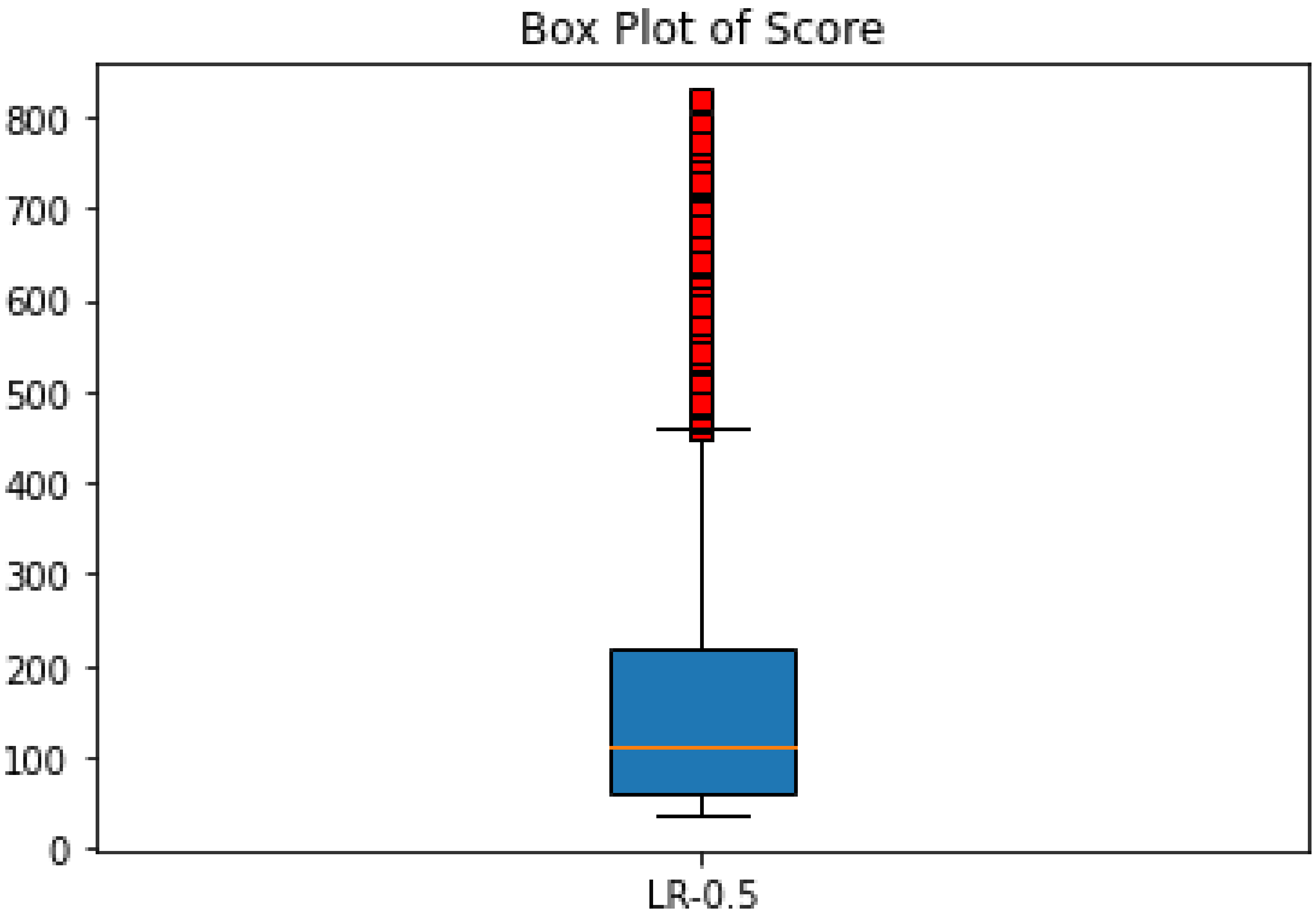


Five-number summary

- First_quartile: 68.0
- Max: 444
- Median: 150.0
- Min: 41
- Third_quartile: 194.0

5th Parameter (Learning Rate - 0.5)

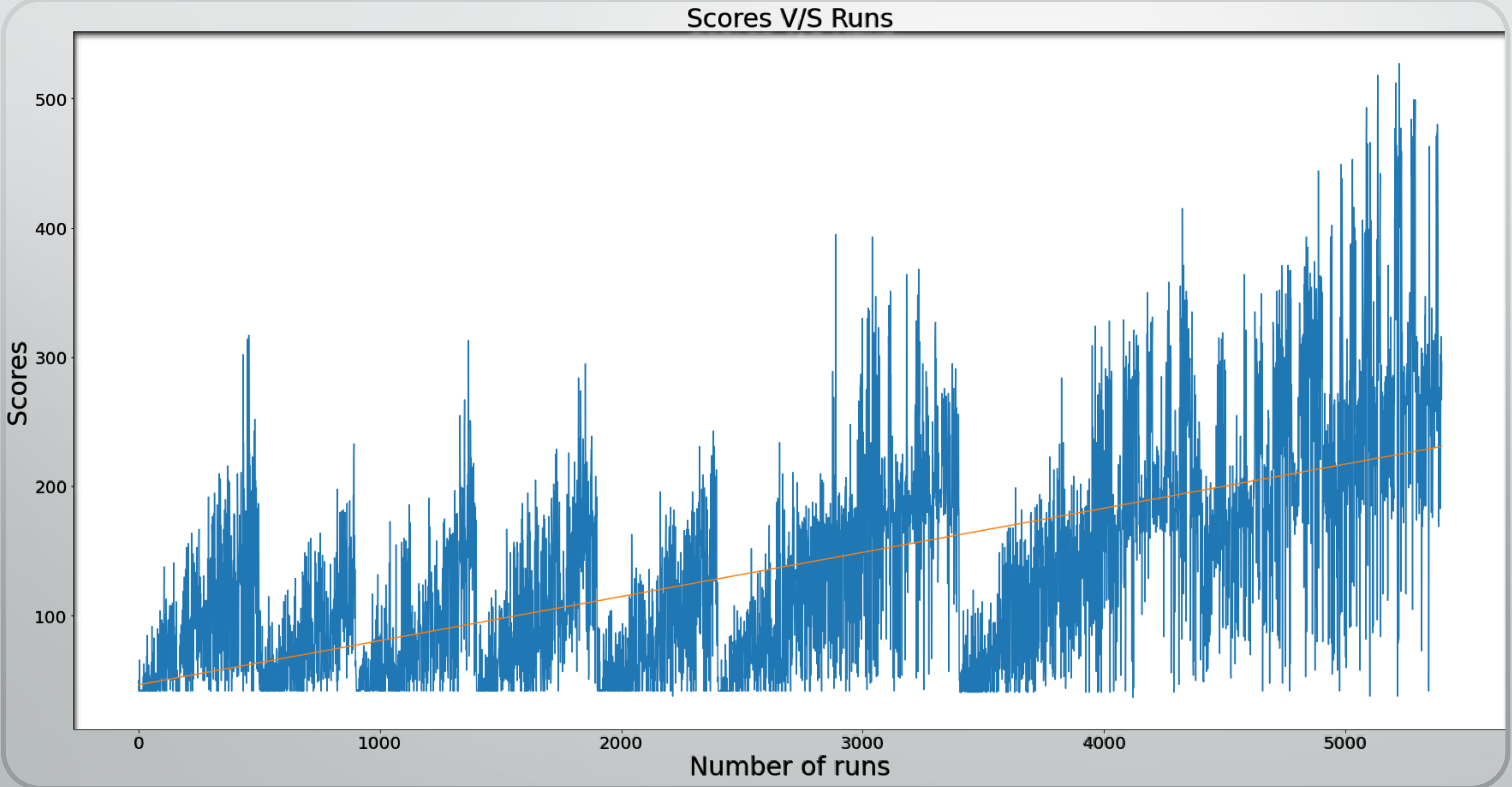


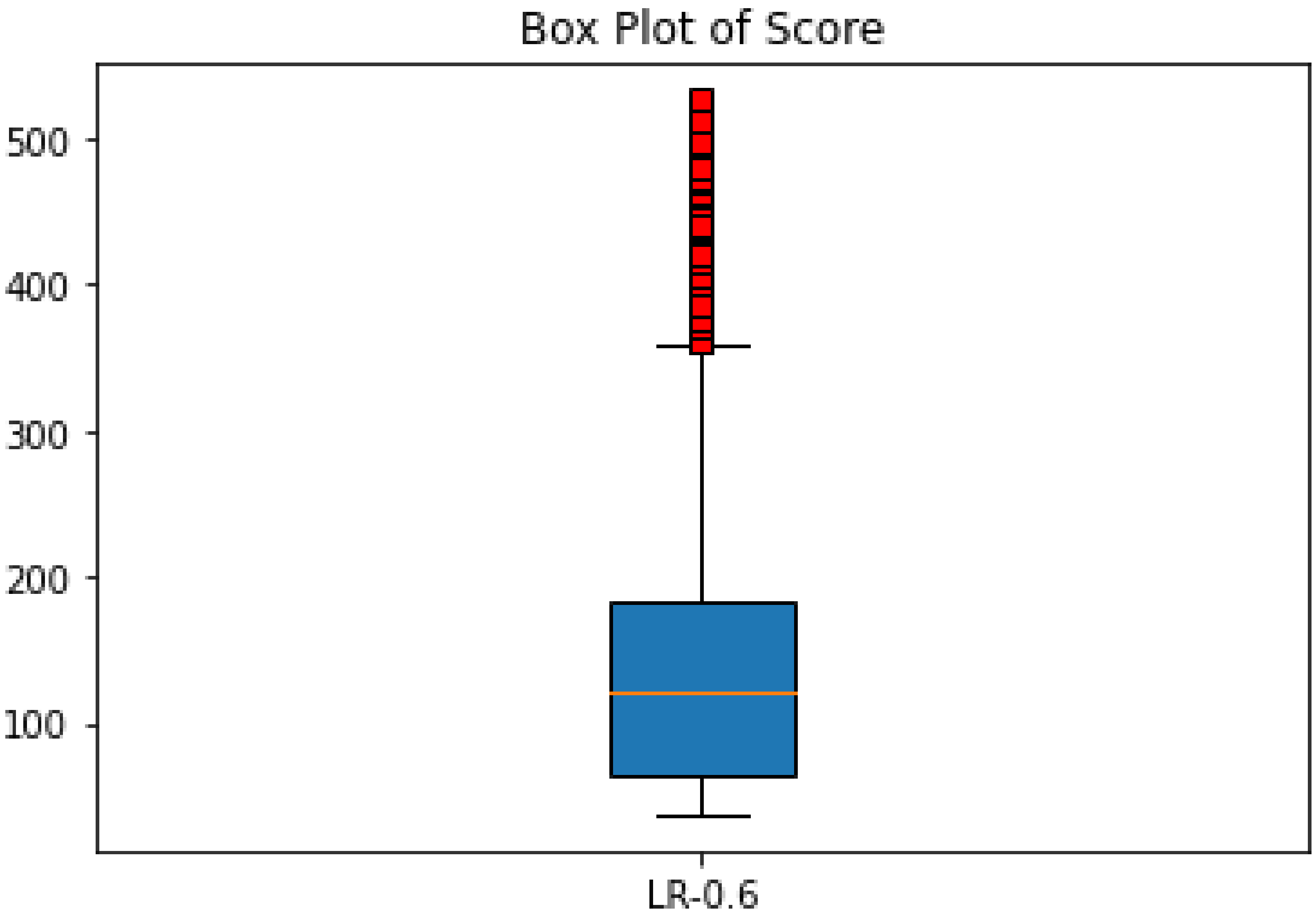


Five-number summary

- First_quartile: 59.0
- Max: 819
- Median: 113.0
- Min: 37
- Third_quartile: 219.0

6th Parameter (Learning Rate - 0.6)

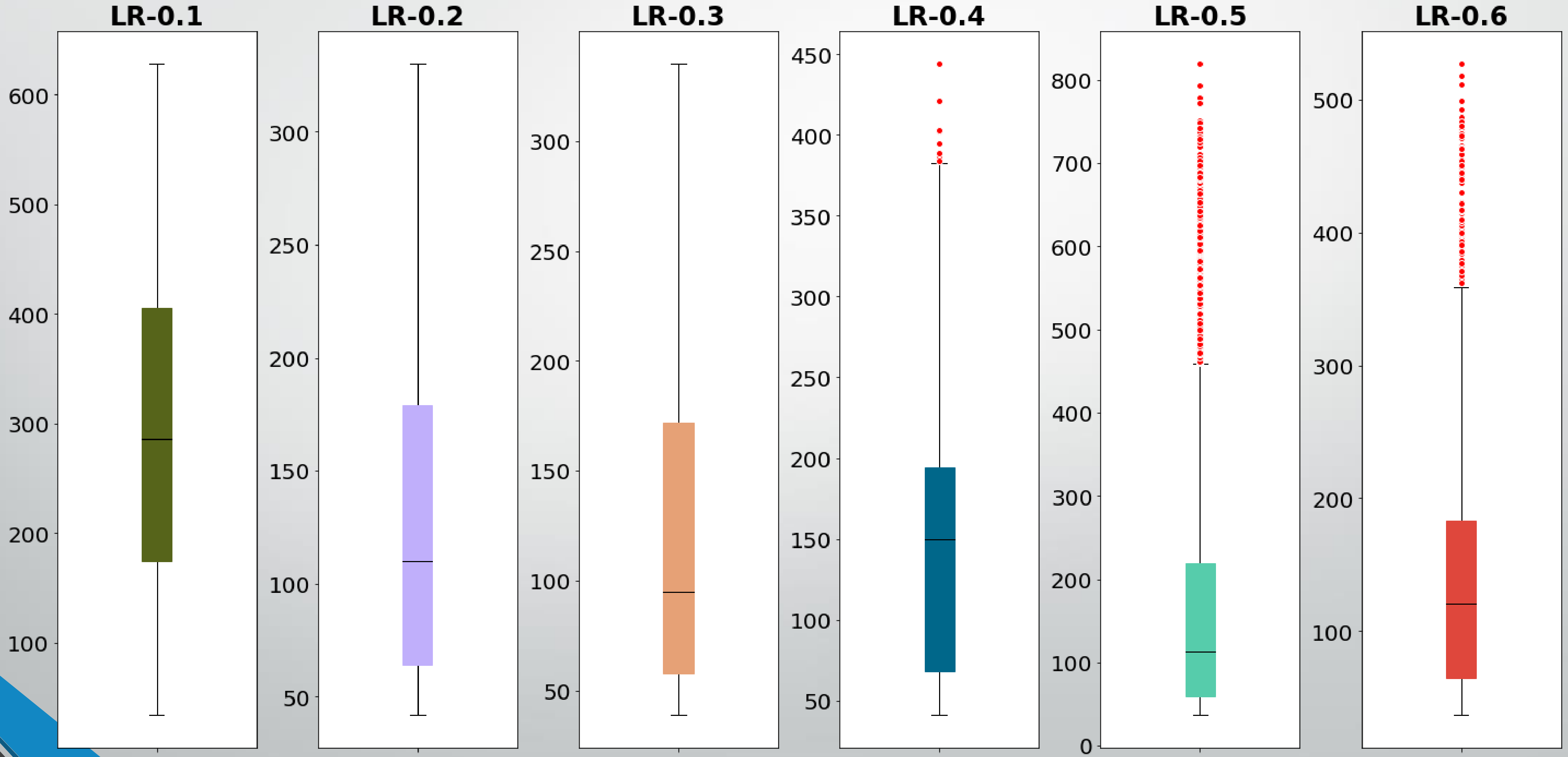




Five-number summary

- First_quartile: 65.0
- Max: 527
- Median: 121.0
- Min: 37
- Third_quartile: 183.0

Combined Box Plot



Summarization

Learning Rate	0.1	0.2	0.3	0.4	0.5	0.6
No. of runs	4090	2621	2425	2060	6910	5400
Max Score of all runs	628	330	335	444	819	527
Min Score of all runs	34	42	39	41	37	37
Mean	295.46	127.83	116.27	146.4	170.48	138.89
Median	286.0	110.0	95.0	150.0	113.0	121.0
Standard Deviation	155.29	70.26	64.91	81.97	149.87	87.01

Interpretation

- 0.1 came out to be the best learning rate which was kind of expected, as learning rate 0.1 means that we are giving higher weightage to the current state rather than the future states.
- The problem of not getting all the states in one single run is visible through graphs, as there is a sharp decrease in the score in the subsequent runs.
- Learning for the higher number of runs results in better scores.
- Even though the highest score of learning rate 0.5 is more than that of learning rate 0.1, and that of 0.6 is comparable to that of learning rate 0.1 but that does not mean 0.5 & 0.6 learning rates are better:
  - They are executed for more number of runs.
  - Mean and Median of scores of 0.5 and 0.6 learning rate are much lower than that of 0.1, which means 0.5 and 0.6 are performing good only for a few number of runs, whereas 0.1 is performing good for most number of runs.

Challenges faced during the training phase?

Huge amount of states and the combinations of states present.

Required a lot of time to initialize Q-Table with the new states.

It took around 14 hours to run over 1000 episodes in one go and get most of the states for one particular parameter.

We still cannot be sure that we have seen all the states and stored it in the Q-Table.

References

- https://medium.com/acing-ai/how-i-build-an-ai-to-play-dino-run-e37f37bdf153
- https://youtu.be/LzaWrmKL1Z4
- https://youtu.be/DhdUlDIAG7Y
- https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56
- https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c
- https://towardsdatascience.com/math-behind-reinforcement-learning-the-easy-way-1b7ed0c030f4
- https://towardsdatascience.com/math-of-q-learning-python-code-5dcbdc49b6f6

Thank you