Introduction to Machine Learning

CS464 - Fall 2020

Yaman Yücel
Arda Can Aras
Yusuf Umut Çiftci
Berkan Özdamar
TA: Doruk Çakmakçı
Instructor: Ayşegül Dündar

1. Introduction

Many classical control mechanisms are based on accurate modeling or domain knowledge of the underlying dynamics and high dimensional input. However, systems with unknown dynamics and supreme amount of input cannot be generalized with these methods.

In this project we propose to use Deep Convolutional Q-Learning to control the game agent in the Google Dino game. The main problem of the classical game which is embedded in Chrome offline mode is, it uses high-dimensional image input. Therefore, we need to be clever while feeding our Q-Learning algorithm data. The second problem we need to address is the adjustment of the Training Parameters of the Q-Learning algorithm. Since Dino can not control itself and take any action when it already jumps into the sky, even a small epsilon will make the agent die very fast. The other problem of the game is the high speed of the game. This problem will also be addressed in the following section.

Due to lack of time and GPU capacity, the experiment obtained a highest score 194 with training on CPU, which can be easily outperformed by human players. However, the results are intuitive and we can conclude that with sufficient time and GPU our control mechanism of the game agent can outperform human players.

Figure 1: Google Dino Game Screenshot

2) Problem description: Detailed description of the problem. What question are you trying to address?

The first problem that needs to be addressed is the hyperparameter tuning. In traditional exploit-exploration strategies for Deep Q-learning, epsilon value decreases over time to guarantee that the agent explores sufficient stages and final policy converges to optimal policy. However, as it was mentioned above, Google Dino agent does not have constant speed. Therefore, it is crucial to deal with several speed modes. This problem is well addressed and identified by the KeZhaoWei report. In this report, the following graph illustrates how the game is divided into three different stages [1].

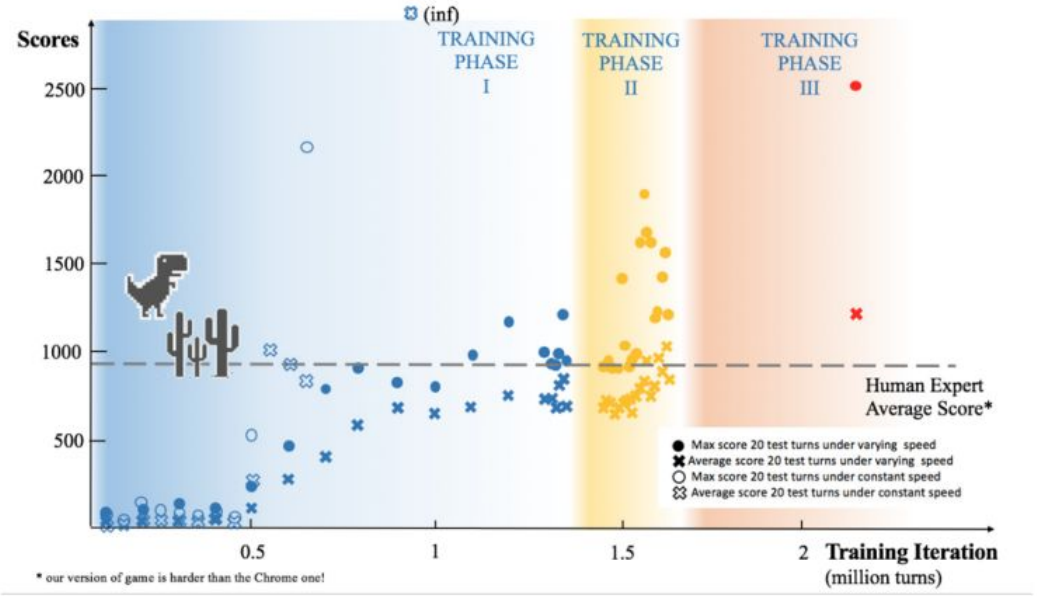

Figure 2: Training curve of Deep Q-learning with different acceleration in different phases [1].

In this model, Training Phase 1 refers to the game with constant speed. The learning curve flattens after 1.3 million turns of training, where the average of 20 test turns is about 750 and max score of 20 rounds is 1000. The error analysis shows under most cases the game agent dies to random action. This is because it cannot control itself once it jumps. It is likely for the game agent to run into adjacent obstacles when it has high velocity. Therefore, in that paper they introduced Training Phase 2 [1]. In that phase epsilon becomes zeros, meaning no exploration. Then they found that sticking into the same pattern instead of exploration yields a better result [1]. They also want to make sure that dino also explores to high speed mode for a sufficient time. Therefore, Training Phase 3 was introduced [1]. Since this problem requires a more complex solution it is directly eliminated by setting the acceleration parameter of the game to zero.

Another problem that needs to be examined is the size of the images as it was mentioned in the previous section. Since we are creating our dataset with the screen shots of the game, its size is very large and images need to be processed to feed into Q-learning method to decrease computational cost. Therefore, a similar convolutional layer pattern is used in the project of KeZhaoWei.

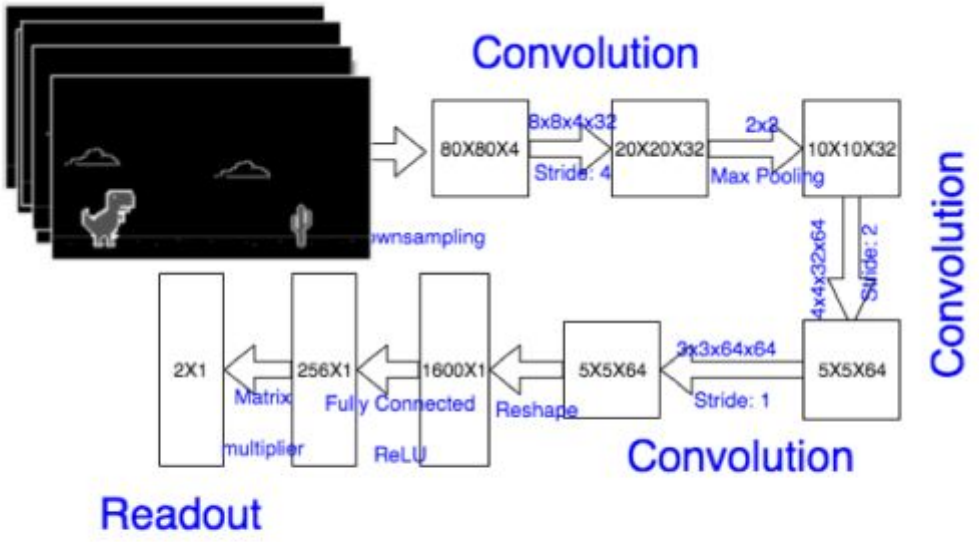

Figure 3: CNN architecture [1].

The methods that are followed to solve all these problems are described in detail in the following section. Above figure was not used directly in our code. We create our own structure which will be mentioned in the following section.

3) Methods: Description of methods and datasets used.

The method that is used to control the game agent in the Google Dino is Deep Convolutional Q-Learning. This method includes several structures and they are built top on each other. Q-Learning is an extension of Reinforcement Learning(RL) which is one of the three primary machine learning paradigms along with supervised learning and unsupervised learning.

Main goal of RL is to train a machine to understand its environment so it can take actions that will maximize cumulative rewards. The strategy of learning is simple. It needs to find a balance between exploring the environment and exploiting what is already learned. However, this strategy needs to be well defined in the algorithm since different cases need to be treated separately as it was mentioned in the problem definition. The trade off between exploration and exploitation is a classic

RL problem that arises in the context. This is called the Multi-armed bandit problem which is named after a gambler at a row of slot machines who is trying to figure out which machine to play to maximize the reward. After understanding the foundations of the RL the next thing that needs to be understood is Q-Learning(QL).

QL involves model-free environments. The agent is not seeking to learn about an underlying mathematical model or probability distribution(as with Thompson Sampling). Instead, the AI agent attempts to construct an optimal policy directly by interacting with the environment. QL uses a trial-and-error based approach to determine the best policy for the environment. It continuously updated its policy as it learned more about the environment. A Q-value indicates the quality of the particular action a in a given state s which can be denoted as Q(s,a). Q-values are the current estimates of the sum of future rewards. That is, Q-values estimate how much additional reward we can accumulate through all remaining steps in the current episode if the game agent is in state s and takes action a. Therefore, Q-values increase as the game agent gets closer and closer to the highest reward.

Q-values are stored in a Q-table, which has one row for each possible state and one column for each possible action. An optimal Q-table needs to contain values that allow the game agent to take the best action in any possible state with the optimal path to the highest reward. To calculate Q-values Temporal differences(TDs) need to be determined as follows:

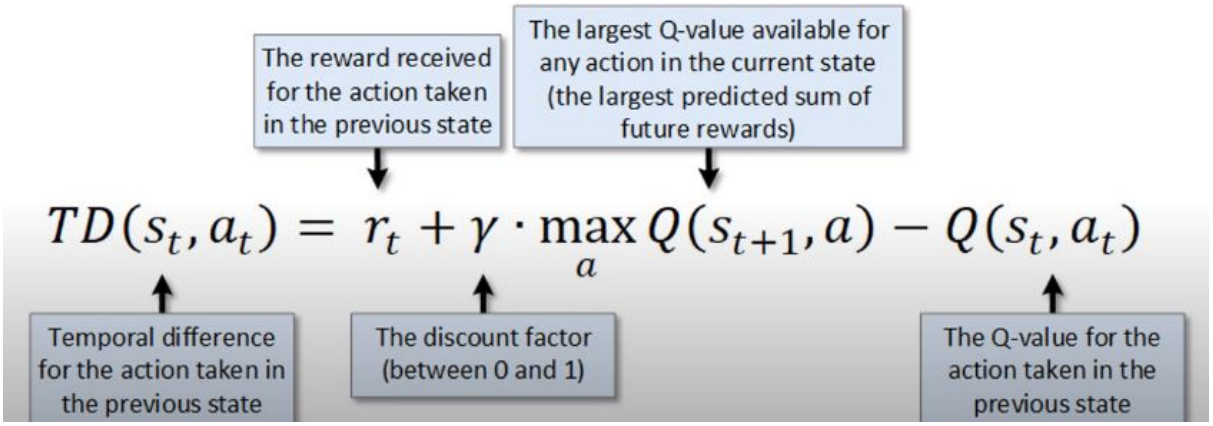

Figure 4: Temporal Difference Formula [2]

This equation means that if the previous action provides us relatively good reward then the Q value for the previous action will be increased. Then the AI agent will rely on this information and make decisions during the next episode. Then using the temporal differences Bellman Equation will be used to determine the next action.

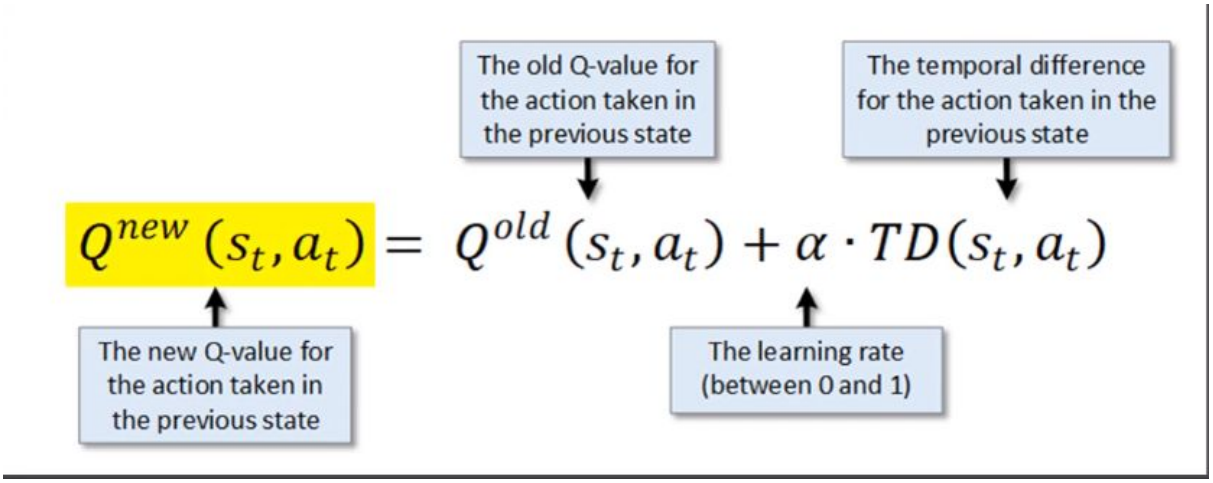

Figure 5: Bellman equation formula [2]

The Bellman Equation exactly states what new value to use as the Q-value for the action taken in the previous state. If the action taken is more promising considering the temporal difference and the learning rate the new Q value will be increased. After understanding the equations the whole process can be described as follows.

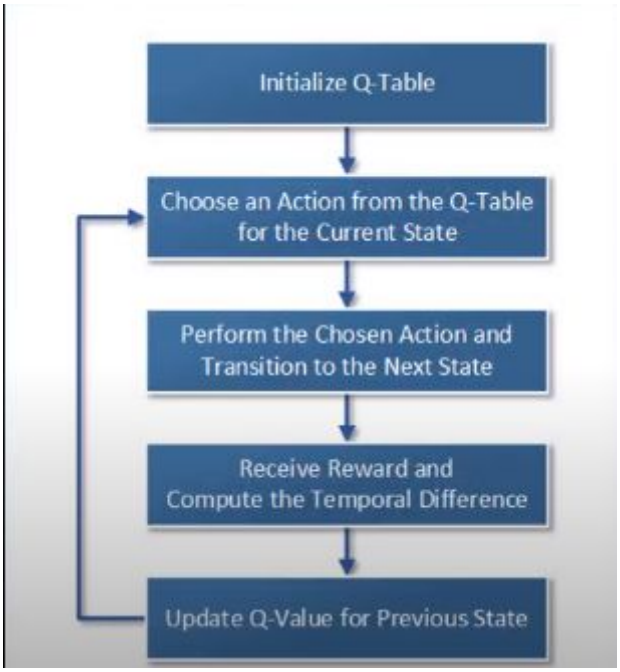

Figure 6: Q value chart [2]

This structure is implemented in our code as follows with several changes. First classical linear decay epsilon greedy algorithm is used to create a balance between exploration and exploitation.

```
if  random.random() > epsilon:
    q = model.predict(s_t)
    max_Q = np.argmax(q)
    action_index = max_Q
    a_t[action_index] = 1

else:
    print("Random Action")
    if random.random() <= cons:
        a_t[1] = 1
    else:
        action_index = random.randrange(ACTIONS)
        a_t[action_index] = 1

if epsilon > FINAL_EPSILON and t > OBSERVE:
    epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE
```

Figure 7: Implementation of Linear Decay Epsilon Greedy Algorithm

Next to updating Q-value, experience replay method is used instead of classical Q-learning method. Algorithm simply takes a single current state, current action, reward, next state, and terminal which determines whether the game is continuing or not. Total number of 16 iterations are used instead of learning from a single state.

Advantage of the experience replay is it is more efficient when using the previous experience by learning with it multiple times. Also Q-values are incremental and do not converge fast enough. Therefore, multiple passes with the same data is crucial.

```
for i in range(np.shape(minibatch)[0]):
    state_t, action_t, reward_t, state_t1, terminal  = minibatch[i]

    inputs[i] = state_t
    targets[i] = model.predict(state_t)
    Q_sa = model.predict(state_t1)

    if not terminal:
        targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)
    else:
        targets[i, action_t] = reward_t
```

Figure 8: Implementation of Q-learning Algorithm

We store the transition in an array which is called experience replay. Transition consists of the current state, action taken, reward,next state, game_over. In the training part, we sample a minibatch of size 16 from the array that stores the

transitions. Using each sample, we update the Q table. If the game is over, reward is directly equal to the value. If the game is not over, we use the bellman equation to update the targets. Bellman equation is given below:

```
if terminal:
    targets[i, action_t] = reward_t
else:
    targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)
```

Figure 9: Bellman Equation Implementation

Then we train our model using the targets and the states of each sample. We store the loss, and Adam optimizer updates the weights.

Next thing needs to be identified is how convolutional neural networks(CNN) works than the exact method Deep Convolutional Q-Learning is just a combination of the prior knowledge.

CNN is a type of artificial neural network that is designed for use with data that have spatial structure. It means that not only the data itself but also the order of the data is important. In our case, CNN is crucial to construct our dataset. In this project, we use screenshots of the game as an input of our learning algorithm. However, raw image captured has a resolution around 600x150 with 3 color channels (RGB). We intend to use four consecutive screenshots as a single input to the model. Which makes the size of the data as 600x150x3x4 that is computationally expensive and not all the pixels are useful for playing the game. Therefore, the OpenCV library is used to resize,crop and process the image. A CNN works by as follows:

1) Generating a set of feature maps for each input case (creating data for convolutional layer)
2) Using pooling to simplify each feature map
3) Flattening the pooled feature maps
4) Connecting the flattened polled feature maps to traditional fully connected layers.

When an image is input to CNN, the first step in processing the image is to apply one or more filters to the image(like feature detectors or kernels). The purpose of a filter is to determine whether a particular part of an image contains a specific feature. The following image is the illustration of CNN.

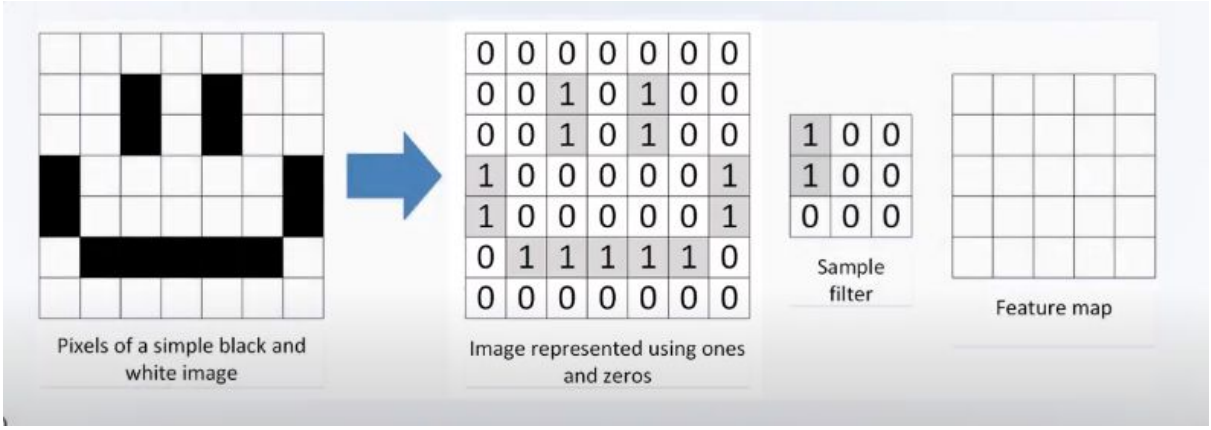

Figure 10: Simple Feature Map Construction [2]

In the figure above, the aim of the filter is to determine a specific shape all across the image. We can construct many different filters and obtain many feature maps which are called convolution layers. The next thing that needs to be done is max pooling to reduce the size of each feature map. Reducing size of feature maps simultaneously reduces the complexity of the model. Lastly we apply flattening to reduce dimension of the pooled feature. The final step is to connect our vector to the other fully connected layers in our deep neural network. The vector of values representing the input image simply becomes the input into a deep neural network and we accomplish our whole model. The output of the CNN network is the estimated Q values for each action. We choose the action with the highest Q value and create the next state.

To summarize, to create a Deep Convolutional Q-learning network, we simply need to use what we have learned about CNN as the front end to feed information into a deep Q-learning network. This allows deep Q-learning networks to learn to perform virtually any task that relies on visual input.

The general structure:

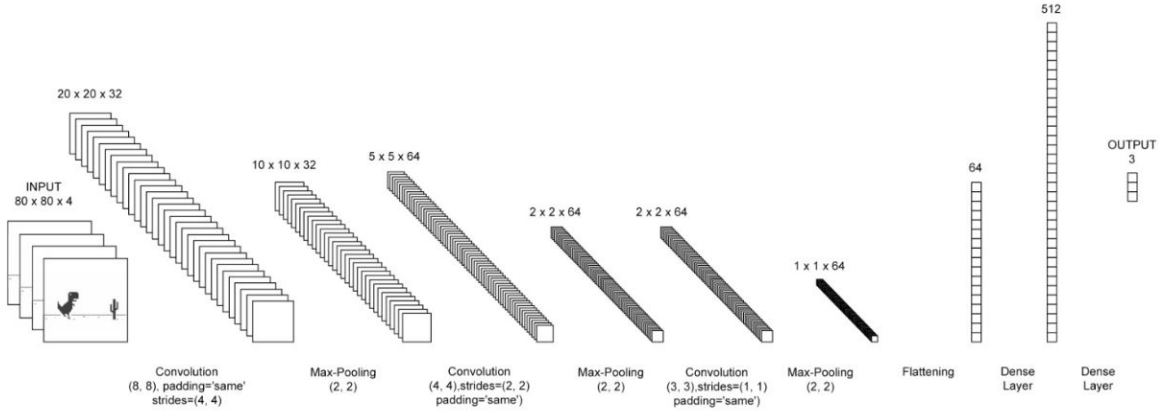

Figure 11: Visualization of CNN architecture

The current state image is appended to stacks of images such that the last four frames become the input of the network. Then a convolution applied to that stacked images with 32 filters which each of the filters have kernel size of (8,8). From the convolution, we receive a feature map of size (20, 20, 32). Afterwards, Max-Pooling applied to the filters to reduce the dimensionality to (10,10,32). Finally, ReLU function is applied to normalize the data and get rid of the negative values.

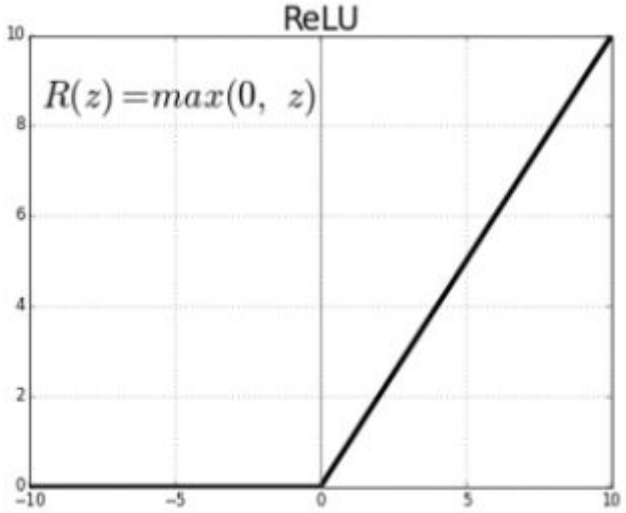

Figure 12: ReLU Function [3]

Then the same process is applied 2 more times consecutively. After the 3rd max pooling, we have the data size of (1, 1, 64). Then, this feature is flattened to have a vector of size (64) to be fed into the neural network. The output of the neural network gives us the Q(s,a) for each action. The maximum of the Q(s,a) pairs is selected to be the action of the corresponding state.

The training is done with mini-batches with size 16, and the loss function of the model is Mean Squared Error. The MSE loss for the current Q(s, a) with the current weights θ_i is given below as:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$$

Figure 13: Loss Function (Mean Square Error) [4]

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$$

Figure 14: Loss Function (Mean Square Error) [4]

Finally, as our optimizer for the model, Adam is used which is an adaptive learning rate optimizer. Adam uses the benefits of both RMSprop and momentum. When updating weights, Adam uses the squared gradients to scale the learning rate like RMSprop, also stores the momentum which is the moving average of the gradient [5].

The game environment was not designed by us. We use trex-game.skipser in our game to run the dino agent. With the aid of selenium library we run java-script code in Python to work across different platforms.

4) Results

The whole set of actions the Dino is taking until the game is over is called an 'episode'. An episode consists of timesteps which get updated whenever the Dino takes an action in the current time state. Using CNN, the policy values for the current state-action pair and also, the next state and the best action is predicted by considering the max of the current Q values. Since it is not efficient to store the Q values in tabular representation since the state-action pairs are infinitely large, a single feed forward pass is used to compute Q values for all actions from the current state for efficiency.

When the Dino dies, the 'episode'' is over and the game score is recorded. Then the mean square error between the current policy value and optimal policy value is calculated.

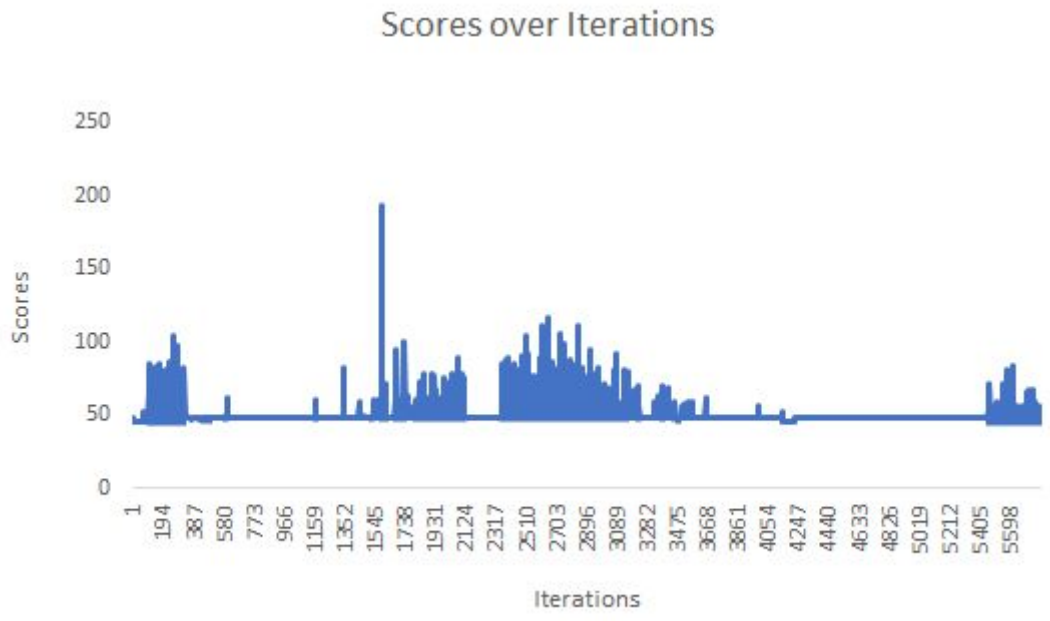

Figure 15: Scores over Iteration Results

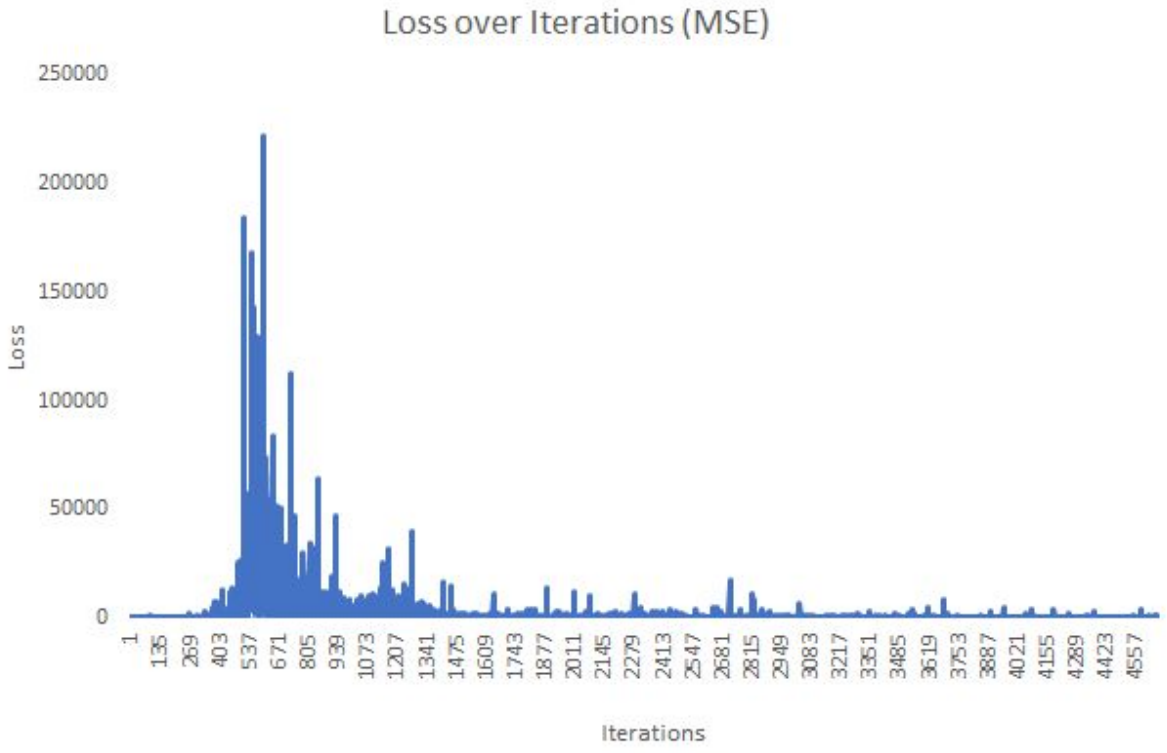

Figure 16: Loss over Iterations Results

5) Discussions

Even, we did not get the desired results, they are still meaningful and generalizable. The reason we could not get the desired results is the dependency of time for the deep Q-learning algorithm to train and the computational complexity. Due to lack of GPU capacity, we trained the code with CPU for 18 hours and the best score we could achieve is 194. Since the score graphic severely deviates from the peak score, we cannot conclude that the game agent learned exactly to score 194.

6) Conclusions

We did not reach the aim of the project which is outperforming the average Google Dino human player. This is because of the lack of GPU capacity. Similar projects train over 2 million times to achieve a maximum score of 4000 points which becomes highly computationally expensive for CPU to train our model. One of the team members' CPU is used to train our code which is able to train a single iteration with an average time of 2 seconds. This will yield to 46 days in total. However, several cross platforms can be used to train the game agent in GPU or a computer that allows this process is also a possible solution.

7) Appendix

Arda and Berkan were responsible for the design of Q-learning, epsilon greedy, and experience replay algorithms. They are also responsible for the whole design of the final report except the CNN structure. Umut and Yaman build the CNN model and integrate it with the Q learning algorithm such that raw image features can be used as inputs at the end of the architecture. They process and filter images accordingly to satisfy our criterias and expectations. As a whole team, we have built an interface between the website and code using the selenium library.

References

[1] KeZhaoWei, " AI for Chrome Offline Dinosaur Game," Department of Computer Science, Stanford University, Stanford, Sep. 22,2017.

[2] Dr. Daniel Soper, " Foundations of Q-learning," *Youtube,* 22 April, 2020. [Video file].Available: https://www.youtube.com/watch?v=__t2XRxXGxI&t=609s&ab_channel=Dr.DanielSoper. [Accessed:Dec 15, 2020]

[3] S. Sharma, "Activation Functions in Neural Networks," *Medium*, 14-Feb-2019. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed: 16-Dec-2020].

[4] Fei-Fei Li & Justin Johnson & Serena Yeung, "Reinforcement Learning", Department of Computer Science, Stanford University, Stanford, May 23, 2017

[5] V. Bushaev, "Adam - latest trends in deep learning optimization.," *Medium*, 24-Oct-2018. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c. [Accessed: 19-Dec-2020].

[6] R. Munde, "Build an AI to play Dino Run," *Paperspace Blog*, 25-May-2018. [Online]. Available: https://blog.paperspace.com/dino-run/. [Accessed: 01-Dec-2020].