CS464

Spring 2020

Project Report

Group - 22

Muhammad Saboor - 21701034

Mian Usman Naeem Kakakhel - 21701015

Abdul Moeed - 21701653

Waqar Ahmed - 21503753

Ufuk Türker - 21502336

Table of Contents

Introduction

In this project, we aim to use reinforcement learning to train an agent to play the Trex Runner game. We will use two different approaches to this problem. The first one is using Deep Q Learning Network (DQN) with hand made features supplied as input to the network. Whereas the second one uses a CNN incorporated in the agent to extract features from raw images of the game. We will use these methods in two environments, custom environment and chrome environment, to compare and analyze the model performance in each setting.

Problem Description

The T-Rex runner game is a 2D endless survivor game in which a T-Rex faces several types of obstacles. This game can be played in the Chrome browser usually when the internet is not available [1]. As the game progresses, the difficulty of the game also increases as the speed of the game is increased. There are 2 types of obstacles that T-Rex has to avoid: cactus and birds. Furthermore, a cacti can be of 3 different widths. These enemies are generated randomly. During the start of the game, only cactus enemies are generated. The bird enemies are only generated once a player reaches 500 score. The score is increased with a constant rate as the game progresses. The score is shown in the top right part of the screen. One has to avoid the maximum number of obstacles in this endless runner. When T-Rex collides with an obstacle, the game ends and the player can restart the game by pressing the restart button.

A Player can do three types of action in the game: jump, duck, or do nothing (in this case, the T-Rex keeps running). One important mechanic to note is that as the T-Rex is jumping and a player sends a command to duck, the T-Rex starts to descend with a much faster speed than usual.

The goal of this project is to use reinforcement learning techniques to train a model that can play the T-Rex runner game by itself. In order to achieve this goal, experience replay is used which is a technique in which an agent learns from its past experiences. So, the agent plays the game by itself and in each game play scores some points and the game is then added to its memory which influences its next game.

Methods

Q-Learning is a reinforcement learning algorithm which is used to learn an optimal policy to come up with an action to be taken by the agent when given a set of states [2]. The algorithm uses the Q function to compute the reward. The function takes the state and old Q-values as input and returns the reward as output which is used by the agent to decide which action to take by considering how it can maximize the reward. We have chosen Q-learning as a reinforcement learning because it is guaranteed to converge given enough time for exploration [2]. Furthermore, it is intuitive to use Q-learning because of how the concept of reward works in the game.
3 different reward configurations were used in this project.

1. If the T-Rex dies, reward is -1 and 0.1 otherwise.

2. If the T-Rex dies, reward is -1 and 0.0001 otherwise.
3. If the T-Rex dies, reward is -1. If the T-Rex makes a useless jump, the reward is 0 and 0.0001 otherwise.

We have used the Deep Q-Learning variant specifically for the models because for Q-Learning, memory complexity of the model becomes huge. This problem can be solved by using a Deep neural network as an approximator.

Environments

Custom Environment

The T-Rex game was implemented in python using the original implementation of the Chrome T-Rex game as a reference [1]. A custom environment was designed to connect this game with the agents. In this environment, the following features were selected to be in the observation space.

1. Global Speed:

   Global speed of the game was selected to be in observation space because this changes what actions a player (or an agent) takes. Since this speed changes over time, it was included in the observation space.
2. The Position of T-Rex:

   The Y-coordinate of T-Rex was selected to be in observation space because this value can be used to determine when to do a duck action to prepare for avoiding the next obstacle. The X-coordinate of T-Rex is not important because it does not change in the game and adding it to the observation space will introduce noise in the data.
3. The Position of Nearest Obstacle:

   Both X and Y coordinates of the next obstacle (within a range of a few pixels) were selected to be in the observation space because this is an important feature which is taken under consideration when deciding the next action. These values are -1 when no obstacle was found in the specified range.
4. The type of Nearest Obstacle:

   The type of the next obstacle (within a range of a few pixels) was selected to be in the observation space because it can be used to decide upon the action differently in case of different enemy types. For birds, this value is 0; for cactus, this value is 1. This value is 2 when no enemy was found in the range.

The size of the observation space becomes 5 when we consider above features to be present in the set of states. The size of the action space is 3 because there are only 3 actions which a player (or an agent) can do during the game: jump, duck, or do nothing.

Chrome Environment

A second environment was created by connecting the agent to the original T-Rex runner game in the Chrome browser by using selenium library. The game can be found by opening the following URL in the Chrome browser:

chrome://dino

Using the selenium library, the state of the game (T-Rex being dead, alive, score, high score, and the screenshot) was found. Furthermore, action taken by sent was also sent to the Chrome browser by the selenium library [3].
The screenshots of the game were taken using the selenium library and these screenshots were processed as follows. First of all, an image was resized to 84x84 px using the open-cv library [4]. Then the HUD was removed by making it black because it does not provide any useful information to the network. Finally, 4 such images were combined and then provided to CNN as an observation state.

Architectures

DQN

The DQN architecture consists of a Neural Network, and an agent which performs the training for the neural network. The agent stores data regarding the game, generates required results for the backpropagation of the neural network using the bellman equation [5]. Finally, the agent trains the neural network on the stored data.

The neural network in our DQN, consists of an input layer, an output layer and several hidden layers. The number of hidden layers, the number of nodes in the hidden layers and the learning rates are left as variables in the Neural network to see what works best. On each of the hidden layers, the ReLU activation function is used. ReLU was the best choice amongst Sigmoid and ReLU as ReLU is computationally faster than Sigmoid. As for LEAKY ReLU, we did not require leaky ReLU as our learning rate is not very high as to present the dying ReLU problem. Also, LEAKY ReLU is used to factor in any negative features that may be present in the dataset. Since we did not have any negative features thus leaving behind ReLU as the only choice. The size of the input layer and the output layer depend on the states given by the game environment and the output layer depends on the actions required by the environment. In our case, as explained in the environment, we have a 6x1 state input while we have 3 actions to output. This 3x1 vector output by the Neural Network is known as the Q value. Before giving the states as input to our NN, we normalize them to make sure that the NN does not give one of the states higher weightage due to any large value that it may have.

Before training the model, there is an exploration phase where data has to be collected in order to be used for training later on. In this phase, instead of asking the NN to suggest a good action, a random action is performed and its result is recorded in the form of what action led the game from one state to another, the reward of that state and whether that state led to the Dino dying or not.

After exploration, the training stage is reached where a fixed size sample of the recorder data is used to train the NN. This technique is known as mini batch training and is used instead of stochastic training or Batch training as Batch tends to avoid global minima and stochastic takes a lot longer than mini batch. The training process includes giving the states in the mini batch to the NN and then getting the required q values from the output. To backpropagate the desired q values, the equation below is used.

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

This equation calculates the desired q values from the q values gotten from the next state [5]. The reward is added into the q value for the best action by a factor of gamma which is selected as 0.99. Gamma simply controls the influence of the next q value on the current one. Thus if the current action tends to lead to a state where the Dino may die, that action has a decreased q value. Thus in a trained model, the action with the highest Q value outputted is selected as the current action.

CNN-based DQN Using the Custom Environment

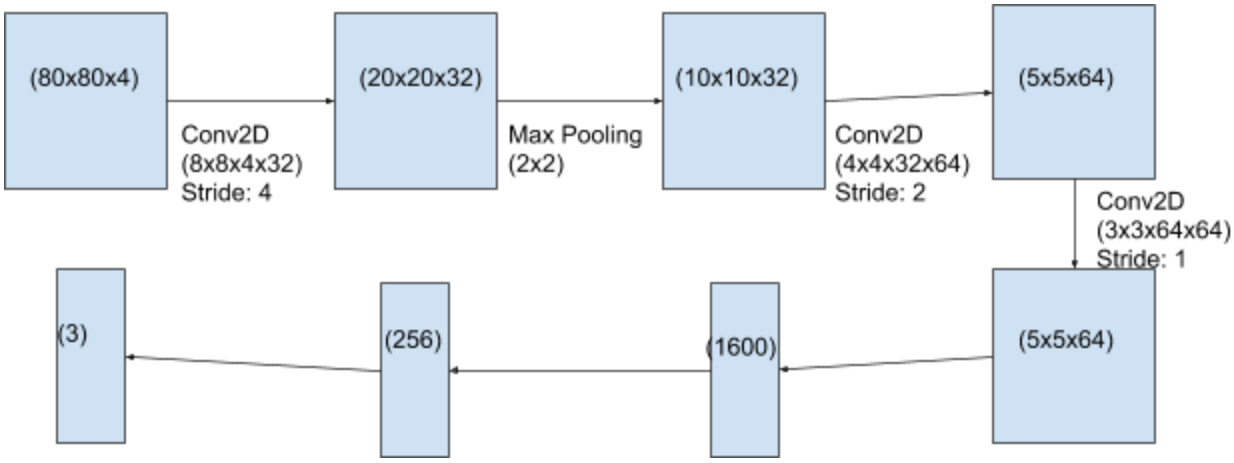
The modified version of the network uses the image matrices generated by the environment, which are shown to the player as part of the default GUI, as the observation space instead of the previously described features. Due to the increased complexity of the input features and the success of CNNs in visual recognition tasks, convolutional layers were added to the neural network, with the new model consisting of 1 to 3 convolutional layers with ReLU activation, which is followed by 2 fully connected layers in a similar manner to the previous network.

To limit the computation and memory requirements, the raw input frames with dimensions 3x1200x3 were initially converted to single channel images with smaller dimensions as a preprocessing step, ranging from 30x15 to 60x30, through local mean downscaling. This method of downscaling was chosen to maximize efficiency, due to the two dimensional and color invariant nature of the images and the simple shapes of the sprites used when generating the environment, while the image dimensions were chosen based on visualization of the downscaled data, in a manner that would not distort the available information (differing shapes of the obstacles, visibility of obstacles, etc.). Efficiency of the network also presented itself as a target for optimization while constructing the initial model, due to the relatively high time cost of the convolution operations limiting the performance of the scripts on the available machines, which resulted in the usage of smaller layers (larger kernels, larger strides, less output channels) and partially in the usage of smaller inputs, as described previously.

Another main issue to be considered for the new feature space is the unavailability of a way to infer the constantly increasing global speed as a function of a single frame. Therefore, sequences of frames leading to the current one (2 to 4 frames, including the current frame), were concatenated into a single matrix to be used as input to the network, allowing for global speed to be a parameter that is inferable from the input.
For the hyperparameter optimization stage, the models were trained with varying learning rates, convolutional layer configurations and reward functions, as allowed by the available time, details of which are further discussed in the results.

CNN using Chrome Environment

The same architecture as in DQN is used with the exception that the neural network is different as it inputs images rather than 6 hand made features. This time, the input is of size (4,84,84) this means that 4 images of size (84,84) are put together and input to the NN. The Neural Network consists of 3 convolutional layers followed by ReLU and a max pooling layer. The output is flattened and passed through 2 connected layers.



Results

Following are the results obtained using different parameters. The result has the score corresponding to an episode and mean score up till an episode.

MLP-Based DQN (GYM AI Mountain Car)

In this environment, the goal is to get the car to the top of a mountain [6]. The car engine is not so strong as to reach the top in one go so a swinging motion has to be created to get it to the top. The number of hidden layers used in this model is 2 and the size is 512. After 300 episodes of training, the loss converges and the model gets the car to the top almost every time.

MLP-Based DQN (Custom Environment)

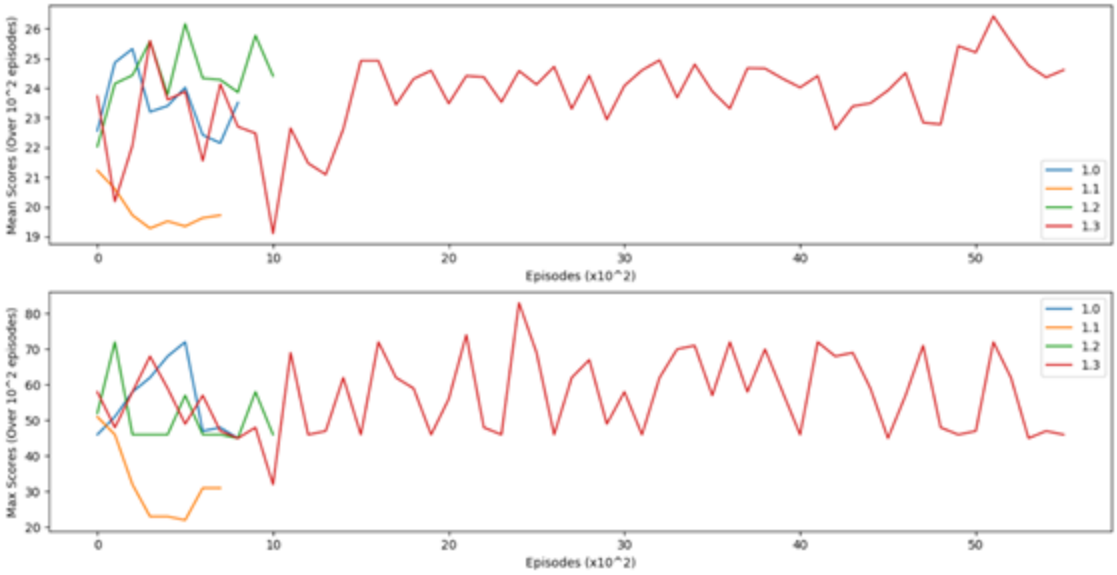


Figure 1. The plotted stats for the MLP-based DQN models showing mean score over 100 episodes (top) and max score over 100 episodes (bottom) with different learning rates.

This table includes the parameters of each model on the graphs.

	gamma	epsilon	alpha(Learning rate)	Batch	h1-h5 size	Output size	High Score	episodes
1.0	0.995	1	0.0001	24	699	3	72	1001
1.1	0.995	1	0.1	24	20	3	58	861
1.2	0.995	1	0.001	24	20	3	73	1161
1.3	0.995	1	0.00005	24	20	3	86	5671

CNN-Based DQN (Custom Environment)

For the training and optimization stages of the CNN-based network, 5 different base models were used, with a varying number of submodels based on small hyperparameter adjustments. The full table of

configurations and the resulting high scores are given in Table 1. It should be noted that the high score itself does not prove to be a reliable measure for model success alone, therefore more in-depth analysis of the results will be made based on the plotted stats for each of the base model categories in the following section.

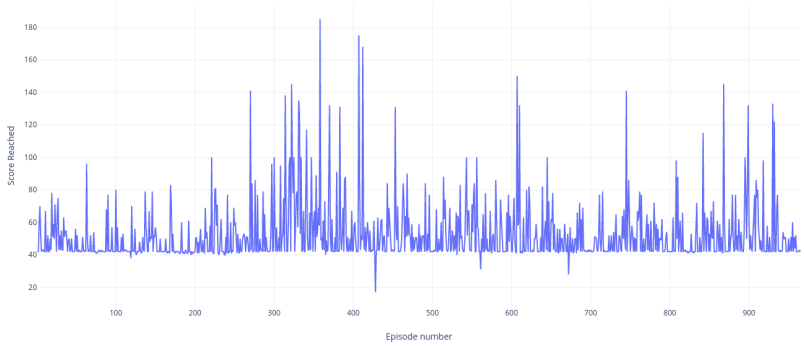
Models	Input Size	Convolutional Layer 1		Convolutional Layer 2		Convolutional Layer 3		Fully Connected Layers		Initial ε	Learning Rate	Instantaneous Reward Function	Frames Per State	High Score
		Kernel Size	Stride	Kernel Size	Stride	Kernel Size	Stride	First Layer Size	Second Layer Size					
1.0	60x30	8x8	2x2	4x4	2x2	2x2	1x1	2816	512	1	0.0001	(0.01)S	1	631
1.1	60x30	8x8	2x2	4x4	2x2	2x2	1x1	2816	512	1	0.0001	(0.001)S	1	56
2.0	60x15	6x6	3x3	-	-	-	-	608	256	1	0.01	(0.001)S	2	104
2.1	60x15	6x6	3x3	-	-	-	-	608	256	1	0.0001	(0.001)S	2	116
3.0	60x15	4x4	2x2	2x2	1x1	-	-	1120	512	1	0.01	(0.001)S	2	70
4.0	60x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	1	0.001	(0.001)S	2	84
4.1	60x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.001	(0.001)S	2	433
4.2	60x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.001	0.001	2	71
4.3	60x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.01	0.01	2	92
4.4	80x20	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.000001	0.01	2	70
4.5	60x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.000001	(0.01)S	2	82
5.0	120x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.001	0.001	4	81
5.1	120x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.001	(0.001)S	4	67

5.2	120x15	4x4	2x2	2x2	1x1	2x2	1x1	3456	512	0.1	0.0001	0.1	4	114

Table 1. The tuned parameters and resulting high scores, for each of the tested CNN-based models (‘S’ in reward functions representing instantaneous score at the given time).

CNN-Based DQN (Chrome Selenium Environment)

In this model, the selenium plugin is used to run chrome and run the offline game through the model. Thus allowing us to run the original chrome game and extract the current screen shot of the game from chrome. The structure of the CNN remains the same as explained in the architecture heading of methods. The results are given by the graph as given below. As it can be seen, the loss has not converged even after a 1000 episodes.



Discussion

MLP-Based DQN (Custom Environment)

We tried running our DQN model with different parameters such as different layer size, learning rates, etc. The graphs in the results sections represent 4 different MLP-based DQN models with different parameters. The DQN model was trained with different learning rates spanning from extremely small to significantly large learning rates for at least a 1000 episodes but the results were almost the same that the high score would be reached in the first few hundred episodes, and after that, the results wouldn’t improve and the mean score would remain consistent. The highest score achieved with one of the models was 86 at around the 2000th episode, the model’s training continued for around 3000 more episodes but the results weren’t promising. The Deep Reinforcement Learning model for the Dino game was trained for over 2 million episodes, and even then the mean score for the first million episodes was around 25 [7]. This shows that other DQN models with the Dino game haven’t been able to achieve the desired results either. The paper also mentioned that the low scores can be credited to but not limited to training on CPU only systems for which some game features were lost. Since our DQN model was also trained on CPUs, this might be one of the factors why we didn’t achieve the desired results.

CNN-Based DQN (Custom Environment)

As a way to visualize and inspect the performance of the networks, a pair of max score and mean score plots were used, with both values represented over 100 episodes per x-axis step (i.e. the mean score is taken as the mean over 100 episodes and the max score is taken as the highest score over 100 episodes, per time step). These were chosen as a way to visualize the consistency and instantaneous success of the networks respectively, while accounting for the randomness introduced by both the epsilon parameter and the randomized mini-batch sampling by taking them over 100 episodes.

For the 1.x variants, a single frame was used as the input to the network, alongside an initial epsilon of 1, learning rate of 0.0001 and varying score-dependent instantaneous rewards, with 3 sequential convolutional layers. Due to the previously mentioned requirement of multiple frames per input to infer global speed, these networks did not provide consistent results, despite the presence of a significantly large high score in the 1.0 model. The mean and max score plots for the submodels can be seen in Figure 2.

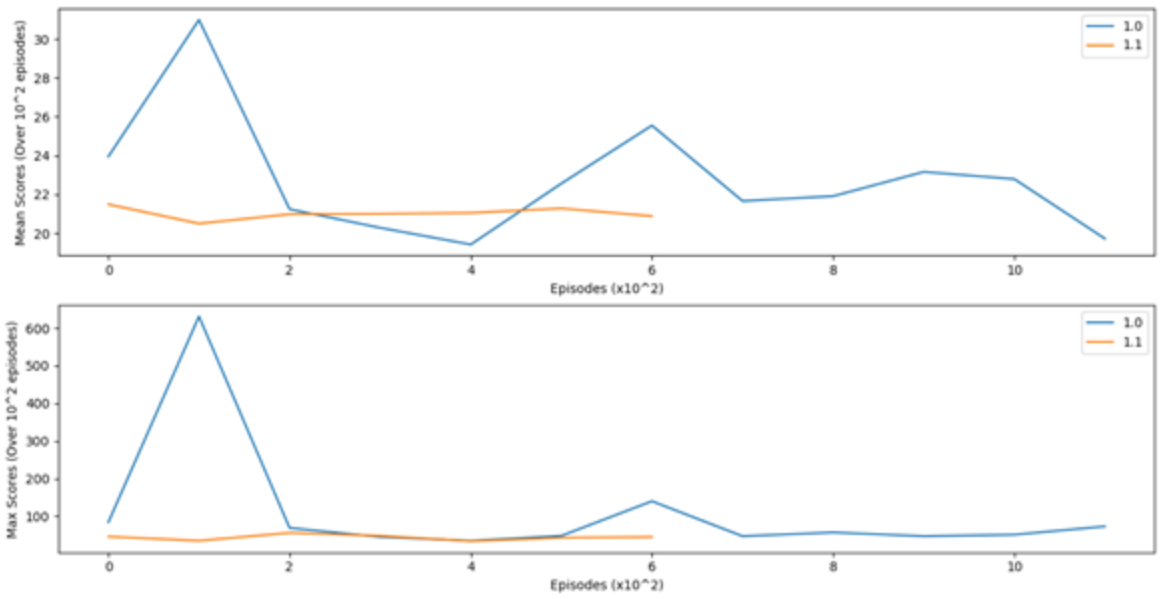


Figure 2. The plotted stats for the 1.x variants of the CNN-based DQN models showing mean score over 100 episodes (top) and max score over 100 episodes (bottom).

It can be deduced from the given stats that the lowered instantaneous rewards resulted in both curves rising more consistently, possibly due to the prevention of overtraining following higher scoring runs, since the final Q values were kept lower. It can also be seen that while the high score from the 1.0 model is significantly higher (at 653), the dip in both plots following the peak value and the fluctuations in the curves suggests significant randomness during model training (as expected of the high epsilon value) and

a lack of consistency, suggesting that the acquisition of such a score does not have a significant effect on the performance of the final model.

For the 2.x variants, 2 sequential frames were used as the input to the network, alongside an initial epsilon of 1, an instantaneous reward function of (0.001)S (with S representing the score at the given instant) and learning rates varying between 0.01 and 0.0001, with a single convolutional layer. The mean and max scores of the submodels can be seen in Figure 3.

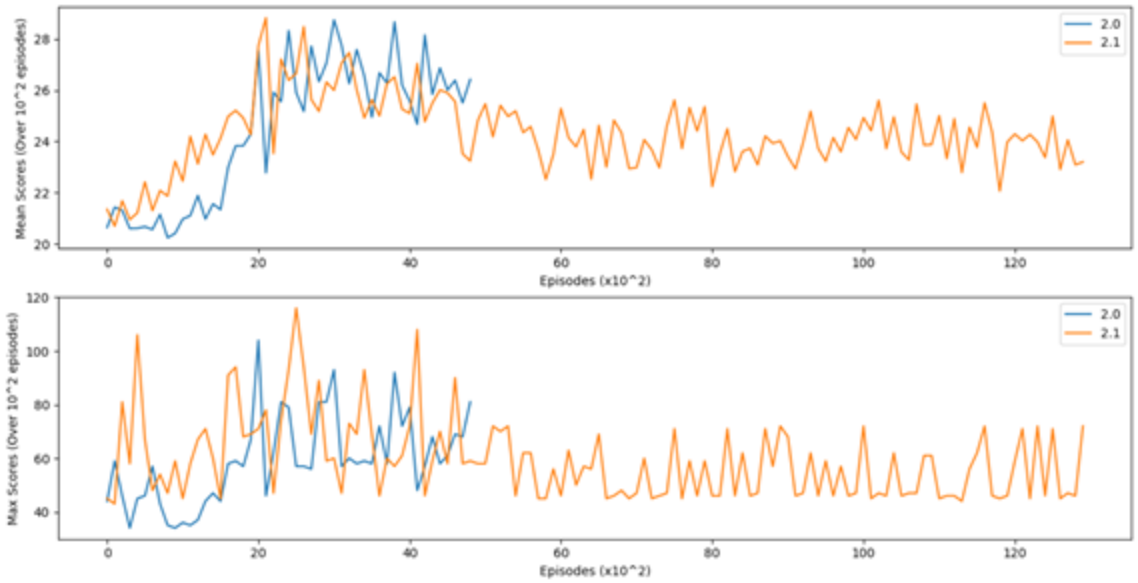


Figure 3. The plotted stats for the 2.x variants of the CNN-based DQN models showing mean score over 100 episodes (top) and max score over 100 episodes (bottom).

It can be seen from the given stats that the networks performed similarly for the varied learning rates, with the randomness from the large initial epsilon being a more prevalent factor in the fluctuations of the curves.

For the single 3.x variant, the network was set similarly to the 2.0 submodel, with the main difference being the presence of 2 sequential convolutional layers instead of a single one. This variant was made mostly to test the performance of a deeper network with the changes to how memory was handled (matrix format and buffer shape changes), but the results are included nonetheless for consistency, and are given in Figure 4.

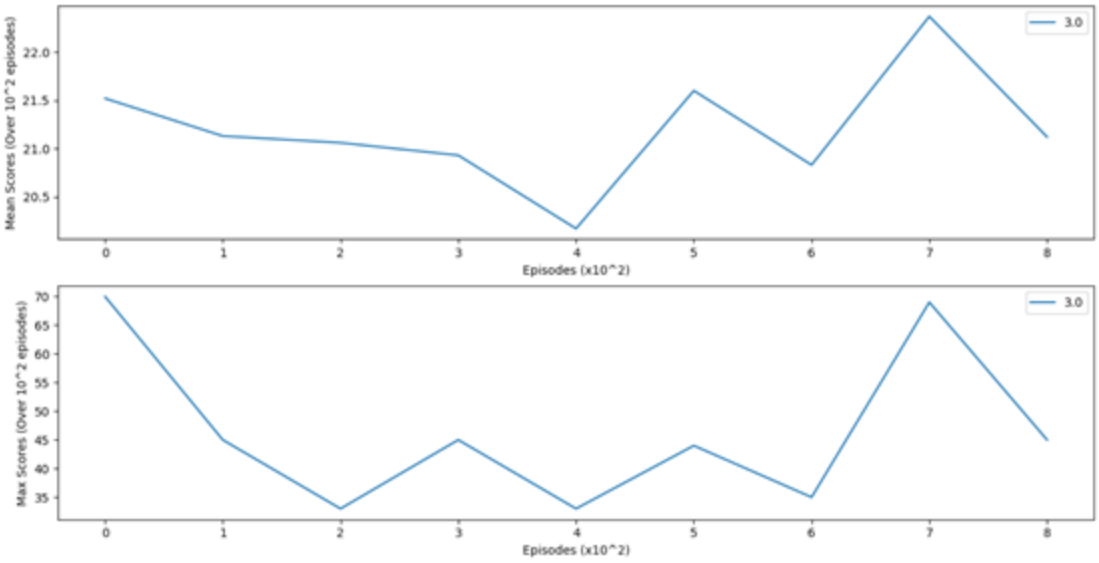


Figure 4. The plotted stats for the 3.x variants of the CNN-based DQN models showing mean score over 100 episodes (top) and max score over 100 episodes (bottom).

The 4.x variants were made following the information compiled from the results of the previous models, having 3 sequential convolutional layers similar to the initial model, an initial epsilon of 0.1 instead of 1 (with the exception of 4.0) and varying instantaneous reward functions and learning rates. The mean and max scores of the models can be seen in Figure 5.

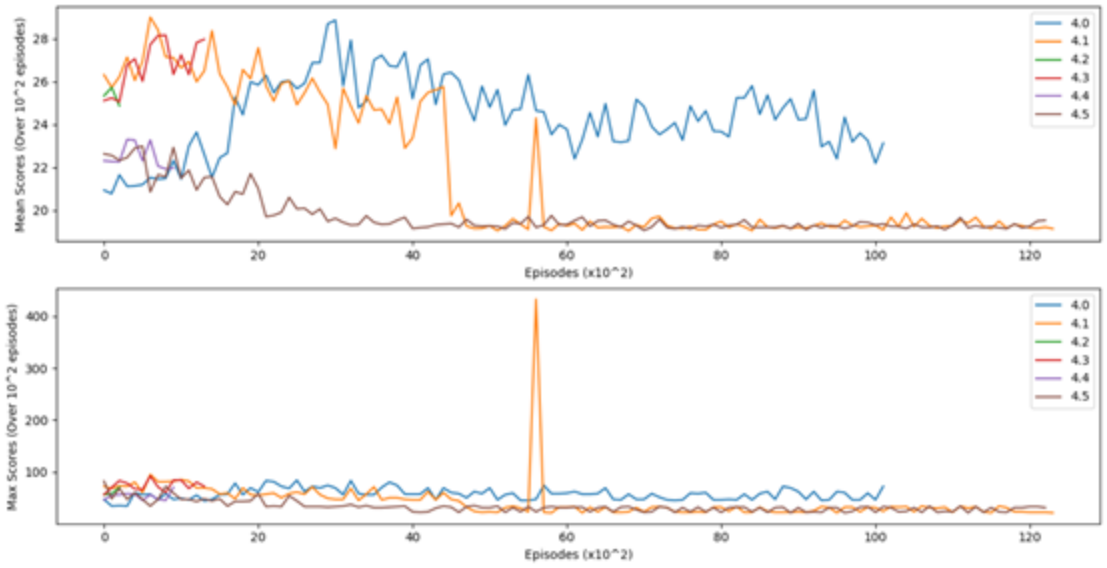


Figure 5. The plotted stats for the 4.x variants of the CNN-based DQN models showing mean score over 100 episodes (top) and max score over 100 episodes (bottom).

It can be seen that while various configurations were run, the models with the smaller initial epsilon get stagnant and the scores dip at a given point based on learning rates and reward values, converging to an unusable model. While one of the models achieved a high score of 433 after the point of stagnation, it did not have a significant effect on the final model, and is therefore considered irrelevant.

For the 5.x variants, the 0.1 initial epsilon value was kept despite the results seen in the previous models, after observing that the initial randomness of the system was apparently enough for successful actions to be observed and the fact that the larger initial epsilon value resulted in stats that were not enough to measure the model performance given the limited training time present with the available devices and time. To counter the negative effects, the system was made so that the relevant observations could be inferred more directly (by increasing the input frames from 2 to 4, resulting in the ability for the network to observe smaller increments of global speed directly from the downscaled frames) and that the more successful attempts would affect the weights more (inst. reward function constant increased), while still keeping the reward function constant so as to prevent overtraining on randomly successful runs (the relation to current score removed from inst. reward function). The mean and max scores of the models can be seen in Figure 6.

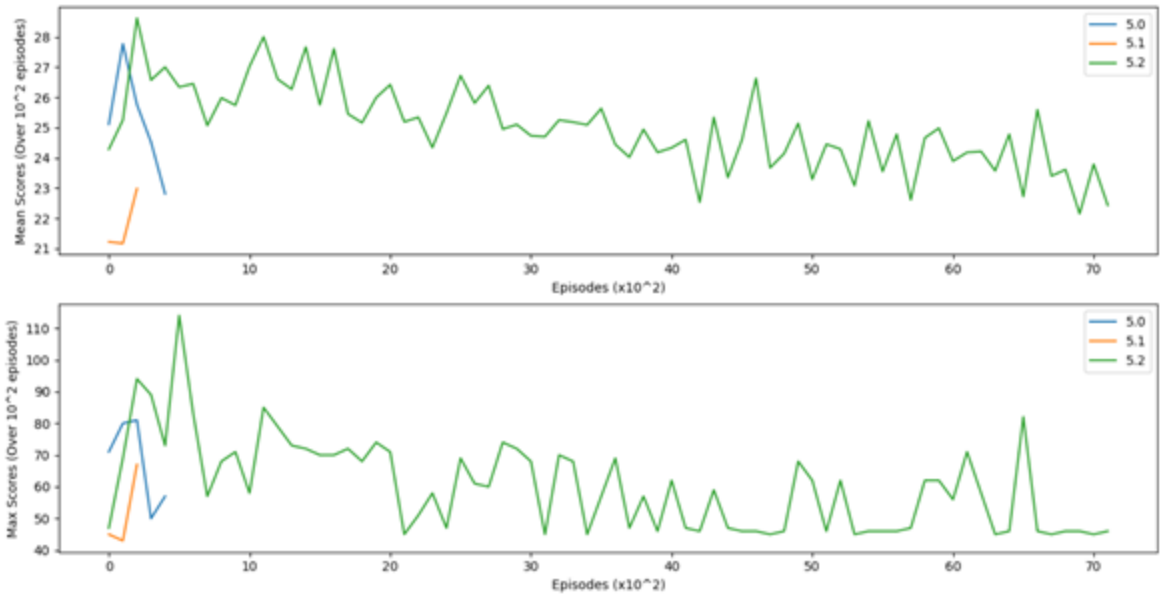


Figure 6. The plotted stats for the 5.x variants of the CNN-based DQN models showing mean score over 100 episodes (top) and max score over 100 episodes (bottom).

It can be seen that, while training was limited for the 5.0 and 5.1 submodels due to the previously mentioned constraints, the 5.2 submodel was able to maintain a significantly higher mean score compared to the previous 0.1 initial epsilon models, while also avoiding the stagnation present in the previous models.

Conclusions

MLP-based DQN

Our MLP-based DQN model was tried on both the custom environment and the GYM AI Mountain car. The model achieved the desired results with the GYM AI Mountain car but not with our custom environment. The undesired results with the custom environment can be attributed to lack of resources such as time, GPU, etc. and since the custom environment was more complex, it required more time to train which wasn't available. This model can be improved by training for a longer time on a GPU with which we can accommodate some more game features.

CNN-based DQN

During the CNN-based model's optimization process, most noticeable challenges were due to the inability to train with more complex models/larger feature sets on the available device and due to the relatively complex nature of error analysis in a reinforcement learning model (i.e. compared to a supervised model). As a result, the stage was educational in a sense in the topics of feature analysis, hyperparameter optimization, and the balancing of efficiency and model performance metrics.

Since the current best-performing model from the ones in the CNN-based DQNs used for this specific task still did not converge properly and had a relatively low mean score and since the training times were minimized to around 1000 episodes only for some of the models, the major further step for this stage would be to train with a larger variety of hyperparameters and over a larger count of episodes and further optimize the network.

References:

[1] *cs.chromium.org*. [Online]. Available: https://cs.chromium.org/chromium/src/components/neterror/resources/offline.js. [Accessed: 10-May-2020].
[2] Melo, Francisco S. "Convergence of Q-learning: a simple proof" (PDF), http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf. [Accessed: 10-May-2020].
[3] “Selenium with Python¶,” *Selenium with Python - Selenium Python Bindings 2 documentation*. [Online]. Available: https://selenium-python.readthedocs.io/. [Accessed: 10-May-2020].
[4] “Introduction to OpenCV-Python Tutorials¶,” *OpenCV*. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html. [Accessed: 10-May-2020].
[5] Dixit, Avinash K. (1990). *Optimization in Economic Theory* (2nd ed.). Oxford University Press. p. 164.
[6] OpenAI, “A toolkit for developing and comparing reinforcement learning algorithms,” *Gym*. [Online]. Available: https://gym.openai.com/envs/MountainCar-v0/. [Accessed: 10-May-2020].
[7] Munde, Ravi. “How I Build an AI to Play Dino Run.” *Medium*, Acing AI, 26 Feb. 2020, medium.com/acing-ai/how-i-build-an-ai-to-play-dino-run-e37f37bdf153. [Accessed: 10-May-2020].

Appendix-A

Division of work among teammates:

Mian Usman Naeem Kakakhel and Muhammad Saboor:

Dino Game, Custom Dino Environment

DQN with Mountain Car GYM

Chrome Dino Environment

CNN Training with Chrome Dino Environment

Problem Description and Models (In report)

Abdul Moeed and Waqar Ahmed:

Optimization of parameters and training of DQN.

Introduction, Discussion of Results and Conclusion: MLP-based DQN (Report)

Ufuk Türker:

CNN-based DQN (Modified from initial agent and environment)

CNN-based DQN Training and Optimization with Custom Environment

Conclusions: CNN-based DQN (Report subsection)