CM50270 Reinforcement Learning
Group Project Report

Group 16: Andreas Polydorou, Giorgos Panteli, Spyros Lontos
Department of Computer Science
University of Bath

Abstract

This project presents an implementation of a deep learning model to successfully teach an agent the control policies of the Dinosaur game directly from sensory inputs of high-dimensionality using reinforcement learning. The model that is used in combination with the algorithm is a convolutional neural network which is trained and updated through Deep Q-Learning. The results obtained are evidence regarding the successful implementation of the algorithm and the high performance of the agent.

1 Introduction

Creating an algorithm which learns to control an agent directly from multi-dimensional sensory inputs like vision has been one of the big challenges faced in the field of Reinforcement Learning (RL). Recent advancements in Artificial Intelligence (AI) and Deep Reinforcement Learning have resulted in several cases where algorithms perform better than humans.

A historical walkthrough would begin in 1996 when the International Business Machine Corporation's (IBM) supercomputer called Deep Blue moved on to defeat the world chess champion, Garry Kasparov which was the first time a computer managed to prevail in traditional tournaments (Tomayko, 2003). In 2013, DeepMind created a Deep RL program which has successfully beat Atari and Consequently led in Google buying the company in 2014. In 2016 AlphaGo was created. The program learnt to play 'Go' from scratch by playing thousands of matches against amateur and professional players. The program took part in several championships, reaching the highest grade of the game before defeating a Go world champion and arguably becoming the strongest Go player in history (Silver et al., 2016). In 2017 AlphaZero was created as a generalised version of AlphaGo. It was able to master the game of chess after only 9 hours of training, shogi after 2 hours of training and Go after 34 hours of training and has been able to beat AlphaGo (Silver et al., 2018). There has been a transition to AlphaStar from AlphaZero in 2019 where the objective was to beat a real-time strategy game called Starcraft. The program has proceeded to achieve a ranking in the top 0.2% of human players and was rated at a Grandmaster level (Risi and Preuss, 2020).

Most successful deep learning applications require a great amount of labelled data for their training phases. The difference of Deep Reinforcement Learning, and what made it so successful is that it does not require such data to train as it is able to learn from scalar reward signals through trial and error. Furthermore, while deep learning methods seem to assume fixed data distributions and data independency, in deep RL there are encounters of highly correlated states and the data distributions will continue to change as the program moves to learn new behaviours and adapt new strategies (Mnih et al., 2013).

This paper demonstrates how convolutional neural networks can be used to control an agent directly from multi-dimensional sensory inputs as the control policies will be optimised from raw image data in a complex RL environment. The neural network is optimised through the Q-Learning algorithms (Watkins and Dayan, 1992), as the weights will be updated after calculating the loss between the Q-values of the input and target variables. To resolve the problems that occur from correlated data and altering distributions, an experience replay mechanism introduced by (Lin, 1993) will be utilised which will randomly take a batch of previous experiences of the agent and consequently smoothing out the distribution during training.

The aforementioned approach will be applied to the famous game known as Dinosaur Game, illustrated in figure 1 below. The goal is to create a single convolutional neural network (CNN) agent that will be able to learn how to play the game and be able to beat the average score of a human player. The network is not provided with any information related to the game and it is set to learn from a batch of screenshots that are fed to it. The code that was used as reference to built upon can be found at (Munde, 2018).

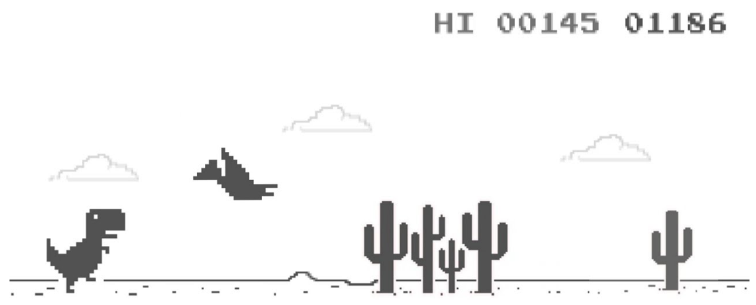


Figure 1: Dinosaur Game

The Dinosaur game was created in 2014 as an easter egg for someone that cannot access the internet. It is an in-built browser game which can be found when accessing the Google Chrome web browser. The game received extensive acknowledgment after its launch as 270 million Dinosaur games are played each month, and in 2018 Google introduced a feature to save the player's scores (Keach and Keach, 2019).

2 Theory

The task involved in this project consists of an agent which interacts with an environment ϵ, which in this case would be the Dinosaur web-page, in a sequence of taking an action and returning the next observations and rewards. At each time-step of the game the agent is set to choose an action a, from a set of available actions, $A = (\alpha_1, \alpha_2, ..., \alpha_n)$, that are allowed by the game. The chosen action is fed to the environment where it will result in the agent moving on to a new state. Instead of the agent observing the internal state of the environment, it observes a stack of screenshots taken at each timestep.

The pictures are represented by vectors of raw pixels which therefore will represent the current screen. The states are represented by a stack of 4 consecutive screenshots that represent the environment of the agent. Frame stacking is implemented so that the agent's actions will depend on the prior sequence of the game frames. Each screenshot is processed accordingly for more efficient state representation to the agent. Each picture taken is being resized to an 80x80 image. Furthermore, each image taken is converted from an RGB representation to grey scale region whereas after they are converted to a binary representation of black and white pixels according to a predefined threshold, which will effectively reduce most of the noise from the state representation.

As the agent will only be able to learn from screenshots fed to it from the game screen, the task of the game will only be partially realised, according to the frame rate of the game. Therefore, many of the environment's states will be distorted so the agent will need to consider a sequence of actions and observations from a batch of experiences, from where it will learn to play the game and adapt to its own strategies. The batch of experiences will be random and only exist for a limited number of time-steps as they will keep refreshing. This set of sequences will consequently result in a large but finite Markov Decision Process where each set of experiences will represent a different state, allowing the use of reinforcement learning methods that use the experiences as a state representation at each time-step., standard reward structure.

The reward the agent will receive will depend on the game score at the current time-step where the greater score, the greater the reward the agent will receive. The ultimate task of the agent is to adapt a strategy of interacting with the environment that will maximise its future reward. The optimal action-value function follows the Bellman equation which follows the assumption that if the optimal value $Q^* = (s', a')$, of the next state, s' was known for all the next possible actions a', then the best strategy would be to select an action a', which would maximise the expected return of $r + \gamma * maxQ^*(s', a')$,

$$Q^*(s', a') = \mathbb{E}[r + \gamma * maxQ^*(s', a') \mid s, a]$$

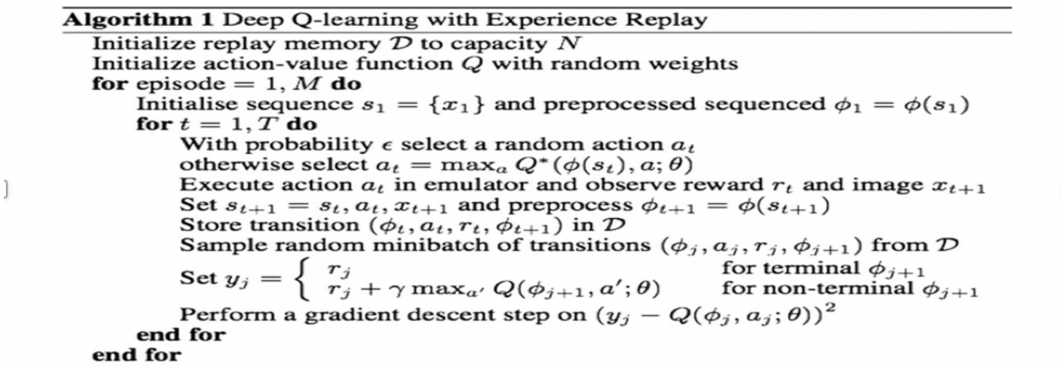
The concept of RL algorithms is to estimate the action-value function by using Bellman's equation as an iterative update of the Q-values, $Q_{i+1}(s', a') = \mathbb{E}[r+\gamma*maxQ^*(s', a') \mid s, a]$. This is an example of a value iteration algorithm which in theory will converge to the optimal action-value function where $Q_i \Rightarrow Q^*$ as $i \Rightarrow \infty$ (Sutton, 1992). When applying this approach though, it will be revealed that actually it is very impractical since the action-value function will be estimated separately for every set of experiences and will not use any generalisation. This is where function approximation is used where it aims to estimate the action-value function instead, $Q(s, a; \theta) \approx Q^*(s, a)$.The function approximator can be either linear or non-linear and is typically a neural network with altering weights θ, based on the loss found between the inputs of the network and the outputs. The neural network is optimised by minimising the loss found at each time-step, $L_i(\theta_i) = \mathbb{E}[y_i - Q(s, a; \theta_i)]$, where $y_i = \mathbb{E}[r + \gamma * maxQ(s', a'; \theta_{i-1}) \mid s, a]$, where the weights of the previous iterations, θ_{i-1}, will be the same when minimising the loss function, $L_i(\theta_i)$

The following algorithm will be model-free, as the task will be approached using pictures at each framerate of the environment, therefore it will not construct an estimate of the whole environment. The algorithm will also be off-policy as it will ensure that sufficient exploration takes place. The behaviour of the agent will be defined by an ϵ-greedy policy which will select the best action with a probability of $1 - \epsilon$, and will select a random action with a probability of ϵ.

3 Deep Reinforcement Learning

Successful implementations of deep reinforcement learning have evidently shown that training networks with raw inputs and updating the weights with stochastic gradient descent is one of the most effective approaches, while feeding enough data into the neural network can result into much better results rather than feeding the network with crafted inputs (Krizhevsky, Sutskever and Hinton, 2017). These past successes have provided the motivation for the approach of this project where the goal was to create a Deep Q-Learning algorithm which would learn to play the T-Rex Dino Run game through a deep convolutional neural network, operating directly on input images from the game screen and efficiently processing the training data by using stochastic gradient updates.

For the training part, a technique called experience replay (Lin, 1993) will be applied. The agent's experiences over each time-step, $e_t = (s_t, \alpha_t, r_t, s_{t+1}, t)$, will be stored in a variable, $D = (e_1, e_2, .., e_n)$, where it will be assembled over a finite number of episodes and act as the replay memory of the agent. When inside the training loop, a minibatch will be created which will consist of 32 random samples taken from variable 'D'. Through this minibatch of experiences, Q-learning updates are occurring where the agent is set to select and execute actions according to an ϵ-greedy policy. This approach of reinforcement learning is called Deep Q-Learning and has many advantages over the known Q-Learning algorithm presented by (Sutton and Barto, 1998). Since each experience sampled can be used multiple times to update the weights of the neural network, a more efficient use of data is observed. Furthermore, the variance of the weight updates is reduced. This is because there are strong correlations between the different samples and learning directly from sequential samples is very inefficient. Instead, the samples used for learning are randomised which will break these strong correlations. Moreover, the use of experience replay will allow the behaviour distribution of the agent to be averaged over various states that have been previously visited. This consequently smoothens out the learning curve of the agent while avoiding unwanted oscillations of the parameters which will cause them to get stuck in a local minimum. For memory efficiency, the algorithm will only store a predefined number of experience tuples in the replay memory and will randomly sample from variable 'D' when its time to perform weight updates. The pseudocode for the Deep Q-Learning algorithm is presented in Algorithm 1 below.



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

3.1 Convolutional Neural Network Architecture

The input of the neural network consists a stack of 4 images, 80x80x4. Progressing through the network there are several hidden layers which are used for processing the features of the input. The first hidden layer is used to convolute 32 8x8 filters with a stride of 4. The second hidden layer is used to convolute 64 4x4 filters with a stride of 2 while the third hidden layer is used to convolute 64 3x3 filters with a stride of 1. After each of those hidden layers the input passes through pooling with a size of 2x2 and then through a Rectified Linear Unit (ReLU) activation function which applies rectifier non-linearity. Finally, the features are flattened out before progressing to the output layer which would give the expected q values for each action.

4 Experiments

In the experiments that were carried out the mini batches that were used for the experience replay were fixed at a size of 32, while the replay memory size of the agent was fixed for the previous 15,000 steps so that it would not get overloaded and interfere with the frame rate of the environment. Moreover, aiming to improve the learning performance of the agent, every 5 terminal states, the weights from the main network are used to update the weights of the target network. The policy of the agent was an ϵ-greedy policy where the initial epsilon value was initiated at a value of 0.2 and was set to gradually decay at a rate of 0.9935 up to a minimum value of 0.0001 for the first 1000 episodes of training, where after it was fixed to that value, while a discount value of 0.99 was used. Initially, the rewards of the agent were set to be fixed as the game developed, but it was realised that the cumulative reward graph would remain a straight line which caused the agent to develop a strategy to immediately lose without performing any action, in order to maximise its reward. This was when it was decided to scale the rewards according to the score it achieved in the game. Figure 2 shows how the rewards for each action of the agent were offered.

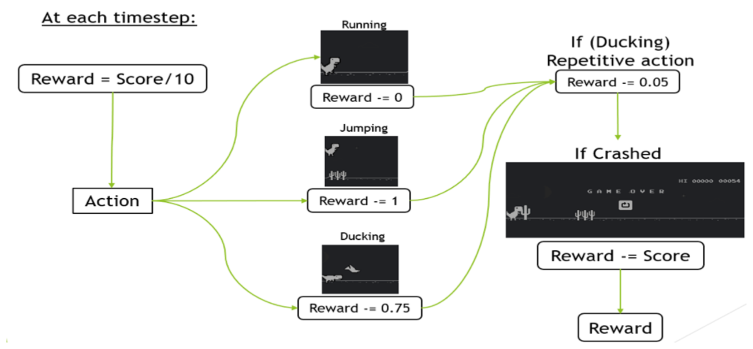


Figure 2: Agent's Rewards

4.1 Training of the agent

In contrast to other machine learning methods where the performance of an algorithm can be easily evaluated on training and validation sets, this is not the case in reinforcement learning. The performance of the algorithm is evaluated by observing the cumulative reward the agent will collect in an episode. The bigger the total reward, the better the performance of the algorithm, but this set of evaluation will tend to be very noisy, as small changes on the weights in the neural network can result to large changes in the distribution of the states that the policy will visit. Additionally, another method to evaluate the algorithm is to project how the Q-values that have been estimated by the action-value function change, which illustrates an estimate of the future reward the agent will receive by following its policy at each state.

For the efficient training of the agent several different challenging aspects of the game had to be taken into consideration. First of all, during the training phase, the agent's velocity is showing a constant increase as the game includes acceleration. The agent will need to adapt to that acceleration up until it reaches its peak where it will continue running at a high constant speed. Furthermore, the game will constantly switch to white and black graphics which represent day and night which will add environmental noise, on top of the other elements in the environment which will contribute to this noise, including the clouds and stars of the sky. Moreover, the obstacles come in different dimensions and different cardinal coordinates which the agent needs to avoid, where in the case of a bat, the agent needs to learn to duck continuously until it surpasses it, instead of selecting an instant action. The long training times have also contributed to the difficulties of the game as in order to see if any changes made to the algorithm have had a positive impact or not, a long training time had to be carried.

After running the game for a total of 2,300 episodes, the necessary parameters have been plotted that indicate the successful implementation of the Deep Q-Learning algorithm. Figure 3 illustrates the decay of epsilon which follows a consistent negative logarithmic line, alongside the gradual increase of the maximum Q-values that are used to choose an action at each time-step. Figure 4 demonstrates how the reward has been scaled up with the game score. It can clearly be seen that as the episodes pass the average score increases showing the successful learning of the agent, with the max score recorded being more than 4,000 points.

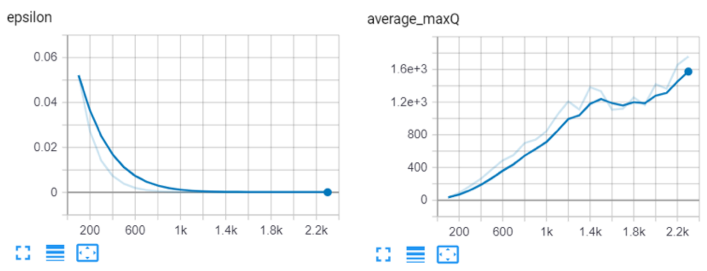


Figure 3: Epsilon and Q-Value trends

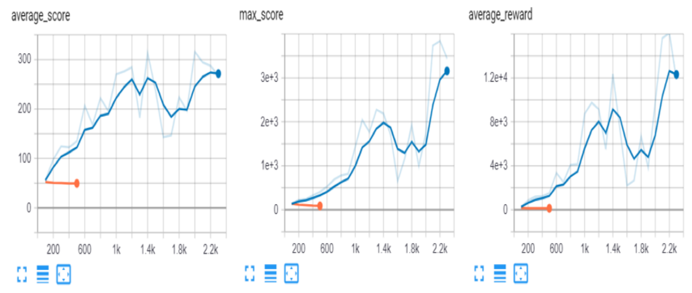


Figure 4: Score and Reward trends

5 Conclusion

This project has presented a successful implementation of Deep Reinforcement Learning in combination with a convolutional neural network and demonstrated the ability to optimise the challenging policies of the Dinosaur game. The use of raw pixels from a batch of screenshots as input for the neural network proved to be very efficient, as Deep Q-learning has combined the mini batch updates and replay memory to speed up weight optimisation of the network.

6 Future work

Even though the implementation of the algorithm has been successful, showing results that proved the agent was learning to play the game, there are several other approaches that could be found to perform more efficient.

If more time were to be allocated for the project, instead of using the Deep Q-Learning algorithm for approximating the Q-values, the Duelling DQN algorithm would be implemented which has previously showed to yield to better performances of the agent (alanwong626/rl-dino-run, n.d.). Furthermore, instead of using environment screenshots as state representations, an alternative approach that could be employed would be to use the value representation of variables that should be kept in track. Some of these could be the runner's current speed, the cardinal coordinates of each obstacle relative to the runner's position, and the dimensions of each obstacle. This approach will reduce all the environmental noise present since the input will be of a much size. Additionally, this could be a benefit in adapting a much smaller network, resulting in less trainable parameters therefore less training time to achieve the same performance.

Finally, another improvement that can be made is to better define the primitive actions of the agent. Theoretically the agent will be able to understand the state representation much better regarding on what action has given a positive or negative reward. This will consequently cause the agent to learn much faster resulting in faster training times.

References

[1] GitHub. n.d. alanwong626/rl-dino-run. [online] Available at: <https://github.com/alanwong626/rl-dino-run>

[2] Keach, S. and Keach, S., 2019. Google Chrome has a dinosaur game that appears when you have no internet. [online] The Sun. Available at: <https://www.thesun.co.uk/tech/10009523/google-chrome-secret-dinosaur-game/> [Accessed 29 April 2021].

[3] Krizhevsky, A., Sutskever, I. and Hinton, G., 2017. ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6), pp.84-90.

[4] Lin, L., 1993. Reinforcement Learning for Robots Using Neural Networks.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning.

[6] Munde, R., 2018. How I build an AI to play Dino Run. [online] Medium.

[7] Risi, S. and Preuss, M., 2020. Behind DeepMind's AlphaStar AI that Reached Grandmaster Level in StarCraft II. KI - Künstliche Intelligenz, 34(1), pp.85-86.

[8] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), pp.484-489.

[9] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. and Hassabis, D., 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), pp.1140-1144.

[10] Sutton, R. and Barto, A., 1998. Reinforcement Learning: An Introduction. IEEE Transactions on Neural Networks, 9(5), pp.1054-1054.

[11] Sutton, R., 1992. Introduction: The challenge of reinforcement learning. Machine Learning, 8(3-4), pp.225-227.

[12] Tomayko, J., 2003. Behind Deep Blue: Building the Computer that Defeated the World Chess Champion (review). Technology and Culture, 44(3), pp.634-635.

[13] Watkins, C. and Dayan, P., 1992. Q-learning. Machine Learning, 8(3-4), pp.279-292.