Università degli studi di Milano-Bicocca

Decision Models

Final Project

A Deep Q-Learning approach to Dino Run

Authors:
Edoardo Gervasoni - 790544 - e.gervasoni4@campus.unimib.it
Riccardo Maganza - 808053 - r.maganza@campus.unimib.it
Alberto Monaco - 803669 - a.monaco10@campus.unimib.it

June 21, 2019

Contents

1 Introduction **1**

2 Datasets **2**

3 Methodological Approach **3**

3.1 Model Formulation 3

3.2 Model Implementation 6

3.3 Computational Challenges 9

4 Results and Evaluation **10**

5 Discussion **11**

6 Conclusions **12**

Abstract

Dino Run is a simple game embedded in the Google Chrome web browser in which a dinosaur has to jump or duck through obstacles to try and go as far as possible to reach a good score. A deep convolutional neural network is fitted to frames of the game to build a reinforcement learning system so that the dinosaur will teach itself the best strategy for the game. Computational and techological issues will be discussed and the good results achieved, albeit the short available time, will be shown. [1]

1 Introduction

Reinforcement learning has been applied to videogames almost since its inception, as they provide a good environment for teaching an agent how to make good decisions. Many times, models initally used for videogames have been later applied to more complex and non-trivial situations [1]. Therefore, even a simple game can prove itself a challenging ground for a reinforcement learning algorithm.

Dino Run (or *T-Rex Runner*) is a simple and lightweight game embedded in the Google Chrome browser [2], displayed when the browser detects networking issues. In the game, a dinosaur is running through a barren field disseminated with cacti and pterodactyls and it has to avoid these obstacles by either jumping or ducking, or it can just keep on running.

[1]All the code used for the project is available at `https://gitlab.com/xelmagax/dino-run-deep-q-learning`.

[2]The game can be accessed from all web browser from `https://elvisyjlin.github.io/t-rex-runner/`

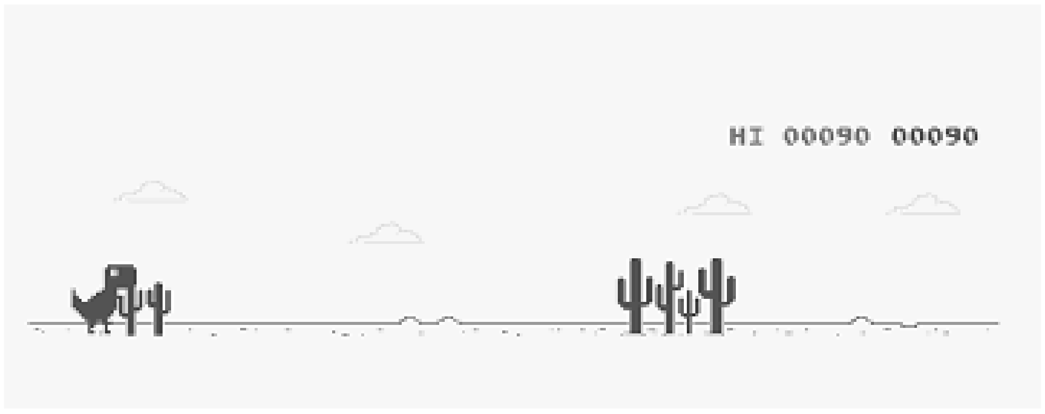

Figure 1: A screnshot from the Dino Run game

The score is higher the further the dinosaur goes, and once it collides with an obstacle the game is over. The speed of the dinosaur also increases with the score.
Standard reinforcement learning techniques such as Q-Learning or SARSA perform badly in this kind of situations, as the environment changes rapidly. In the case of Dino Run, in fact, obstacle positions are randomly generated.

Therefore, the only feasible way to capture game states is by extracting images from the game itself.

A deep convolutional neural network will be constructed taking these images as input as described in [2], where it was originally applied to Atari games, to approximate the so-called state-action value function $Q(s, a)$ as it is impossible to compute it exactly for all possible states.

The complexity of the model posed many computational challenges which will be outlined, and the technological architectures adopted to solve them will be presented.

The scores obtained through the training process will be analyzed, and a clear improvement will be shown.

2 Datasets

No data was given as input to the model, as to simulate the absence of knowledge of the agent at the beginning of the training process.

The game is directly played in the web browser by using the Python package *Selenium*, which allows the program to input key presses directly to the browser as well as to inject Javascript code directly in it to control the game behavior and to extract features of interest, such as the *screenshots*.

The game states are stored in a queue of fixed length that makes up the *replay memory*, which is then used in the training process.

The output files are two CSV files:

- A file containing the scores for all terminated games, to use for further analyses;
- A file containg all the actions made by the agent, used only for testing purposes to detect possible issues in the training process.

3 Methodological Approach

3.1 Model Formulation

A reinforcement learning agent selects an action a in a state s if the Q function reaches the maximum value at state s when taking action a. More formally, by defining:

- S as the set of all possible states;
- $\pi^*(s)$ as the optimal policy at state $s \in S$;
- $r(s, a)$ as the immediate reward obtained by performing action a in state s;
- γ as the discount factor;
- $P(s'|s, a)$ as the probability of reaching state s' after taking action a in state s.

then:

$$\pi^*(s) = \arg\max_a r(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) Q^{\pi^*}(s', a') \tag{1}$$

where:

$$Q^{\pi^*}(s, a) = r(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) Q^{\pi^*}(s', a') \tag{2}$$

where actions a' are chosen according to policy π^*.
The latter is one of the well-known Bellman equations, which express the state-action value function Q as a recursive function. Essentialy, this allows to write π^* as:

$$\pi^*(s) = \arg\max_a Q(s, a),\ \forall s \in S \tag{3}$$

The *Q-learning* algorithm is a standard reinforcement learning technique introduced in 1989 which works by iteratively updating of the Q function based on a *learning rate parameter* α as to eventually obtain the optimal policy π^*.

After defining $Q_t(s, a)$ as the Q function for state s and action a at time t then the updates are performed as follows:

$$Q_{t+1}(s, a) = (1 - \alpha)Q_t(s, a) + \alpha \left(r(s, a) + \gamma \max_{a'} Q(s', a') - Q_t(s, a) \right) \tag{4}$$

This *value iteration* process updates the state-action value function for all states by considering all the possible discounted future reward values and picking the highest one at each time step.

The algorithm is guaranteed to converge to the optimal state-action value function Q^{π^*} when $t \to \infty$.

The issues arising when applying this algorithm in the context of video games, and in the specific case of Dino Run are mainly:

- How is a *state* defined?
- How many possible states are there, therefore?
- Given the high correlation between consecutive states, is a state-by-state learning algorithm the best choice?

In defining states as images from the game itself before, a simplification was made. In fact, a single frame does not convey all the information needed to define the state.

For example, in this case, if the dinosaur is mid-air one must know when it jumped to correctly determine the state. Therefore, a state can be defined as *an ordered array of F frames* for an arbitrary number F, defined as a sufficient timeframe to convey the needed information.

It directly follows that the number of states $|S|$ is very high and the states themselves cannot be known beforehand, since obstacle positions are random.

Starting from these issues, Mnih *et al.* proposed in [2] a novel technique, known as Deep Q-Learning, which tries to compute an *estimate* of the Q function by using a convolutional neural network, training it through a process known as *experience replay.*
The algorithm works as follows:

Algorithm 1: Deep Q-Learning with Experience Replay

```
1  Initialize replay memory $D$ to capacity $N$;
2  Initialize state-action value function $Q$ with random weights $\theta_1$;
3  for episode 1 to $M$ do
4      Initialize sequence $s_1 = \{x_1\}$;
5      for $t = 1$ to $T$ do
6          With probability $\epsilon$ select a random action $a_t$;
7          otherwise select $a_t = \max_a Q(s_t, a, \theta_t)$;
8          Execute action $a_t$ and observe reward $r_t$ and image $x_{t+1}$;
9          Set $s_{t+1} = s_t, a_t, x_{t+1}$;
10         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$;
11         Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from
             $D$;
12         if $s_j$ is terminal then
13             $y_j = r_j$
14         else
15             $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a', \theta_t)$
16         end
17         Perform a gradient descent step on $(y_j - Q(s_j, a_j, \theta_t))^2$ to
             update $\theta_t$ to $\theta_{t+1}$
18     end
19 end
```

Essentially, the algorithm tries to learn the best policy from a sequence of actions and their outputs just as Q-Learning, but it goes further by fitting a neural network with weights θ_t with the inputs being the raw values of the frames (RGB or grayscale images).

The *exploitation* of the search space is obtained as outlined above, while the *exploration* is guaranteed by the ϵ-greedy starting policy.

Moreover, the *experience replay* mimicks a stochastic gradient descent algorithm and it solves of the correlation of sequential frames by randomized extraction.

As described, the presented algorithm solves all the 3 issues outlined above and has therefore been used for the examined game.

3.2 Model Implementation

First of all, the action space of the dinousaur has been limited to 2 actions: jumping or keeping running.

Based on personal experience of the authors, there is no obstacle for which it is mandatory to duck as they all have some way to be jumped over. This reduction will help expediting the training process.

To help the model and ease the training phase, a couple of preliminary steps are applied.

First, the images for each frame are pre-processed to delete noise, in this order, as outlined in figure 2:

1. They are cropped to a square size (as convolutions work better with square inputs);
2. They are converted to grayscale;
3. An edge detection algorithm is applied to keep only the relevant features such as the position of the dinousaur and of the obstacles.

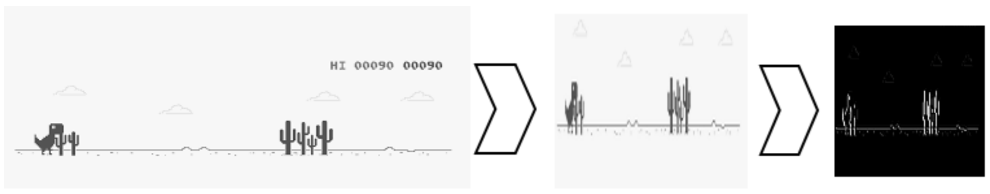

Figure 2: Outline of the image processing steps

Moreover, before the ϵ-greedy training phase begins, the agent performs random actions for a certain amount of frames, as to build up a starting *replay memory* to base the training on.

Finally, the increase in speed of the dinousaur proportional to the score has been disabled and the speed set constant so that there would not be issues where the model fails to output an action before the game progressed to another frame, which would be useless.

The presented model has then been implemented in the Python programming language through the use of the *Keras* library: a high-level interface which allows easy creation of neural network architectures.

Its architecture is described as following, in the clear Keras syntax:

Model Architecture outline in Keras syntax

```
model = Sequential()

model.add(Conv2D(32, (8, 8), padding='same',
                 strides=(4, 4), input_shape=(80, 80, 4)))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Activation('relu'))
model.add(Conv2D(64, (4, 4), strides=(2, 2), padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3), strides=(1, 1), padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(512))

model.add(Activation('relu'))
model.add(Dense(2))
adam = Adam(lr=LEARNING_RATE)
model.compile(loss='mse', optimizer=adam)
```

Basically, the network is made up of 3 convolutional layers and two dense layers. The hyperparameters such as the number of layers and the size of the convolutions have been borrowed from [2], with the notable addition of Max Pooling layers to avoid overfitting.

Probably the most important parameter set is the number of frames F to stack as to compose a single *state*: as suggested by [2], it is set to 4.

Each layer after the first one is preceded by the application of a Rectifier Linear Unit (ReLU) to the outputs of the previous layer as to not allow the propagation of negative values through the network.

The last layers outputs two values, corresponding the Q-values for each of the two actions at the given state:

1. Do nothing;
2. Jump.

The remaining of the hyperparameters have been set as follows, again borrowing from [2] and through a trial-and-error process:

- The learning rate parameter α of the gradient descent algorithm has been set to 10^{-4};
- The number of observation O to initialize the replay memory has been set to 10000;
- The replay memory fixed size $|D|$ has been set to 50000;
- The batch size for each gradient descent step has been set to 16.
- The discount rate γ has been set to 0.99. The goal of the agent is in fact of continuing the game for as long as possible. It has not been set to 1 to help the agent learn about short-time dependency (i.e. it cannot jump while mid-air as it has just jumped.) by giving a bit more importance to actions close to each others;
- The hyperparameter ϵ for the ϵ-greedy policy has been set to linearly anneal from a starting value of 0.1 to a final value of 10^{-4} through 100000 epochs, after which it stays fixed to 10^{-4} to allow for exploitation of the learned strategy.

Most of these parameters have not been adequately optimized due to the short time available. Therefore there is surely room for improvement.

3.3 Computational Challenges

The network, as described in the previous section, has an astounding number of 112,290 parameters to estimate, hence the name *Deep* Q-Learning.

It is thus highly infeasible to train it on a local, regular, workstation. In fact, when the model was first launched locally, it freezed one of the author's workstation by skyrocketing CPU usage to 100%.

After examining possible solutions, the decision was made to use the *Google Colaboratory* environment to train the model.

Google Colaboratory (or *Colab*) is a cloud environment for research purposes in the field of machine learning gently offered by Google, which comes bundled with a 16GB GPU unit on a Linux VM and all the most used Python packages for machine learning (such as Keras) as well as an interactive notebook environment similar to Jupyter. [3]

This setting allowed the model to be trained with relatively high speed and to achieve close-to-no latency when communicating with the web browser.

It also came with a few disadvantages. Namely:

- The logic had to be implemented twice: once on the Colab notebook and once on the local side to keep the code relatively tidy and organized;
- Google Colab is a volatile environment: each time the runtime is restarted, all files previously saved to the cloud VM are lost. Therefore, a solution was needed for the persistence of the model data as not lose progress in case the runtime halted. The easiest found way out was to connect the Google Colab notebook to a personal Google Drive instance, which interfaces quite easily with the computing environment.
- Google Colab has no GUI capabilities aside from the interactive notebooks, therefore it is not possible to visualize the training process and see the dinousaur actually jump obstacles. This forced the *Selenium* package to communicate with the web browser in *headless mode*, i.e. without using a graphical interface.

Anyway, the pros outweighed the cons by far and Google Colab was picked as the definitive development platform for the project.

4 Results and Evaluation

The training process lasted for about 72 hours without stopping, in which about 800,000 frames were recorded, leading to 3,150 games being recorded.

Unfortunately, no video could be produced of the actual dinosaur playing the game all by itself for the reason outlined in section 3.3 arising from the complexity of the model.

Therefore only the time series of the scores obtained in each match will be analyzed. The data is very dense and has therefore been aggregated for easier display by producing two plots:

1. A plot of the max scores achieved every 10 games (Figure 3);
2. A plot of the mean scores achieved every 10 games (Figure 4).

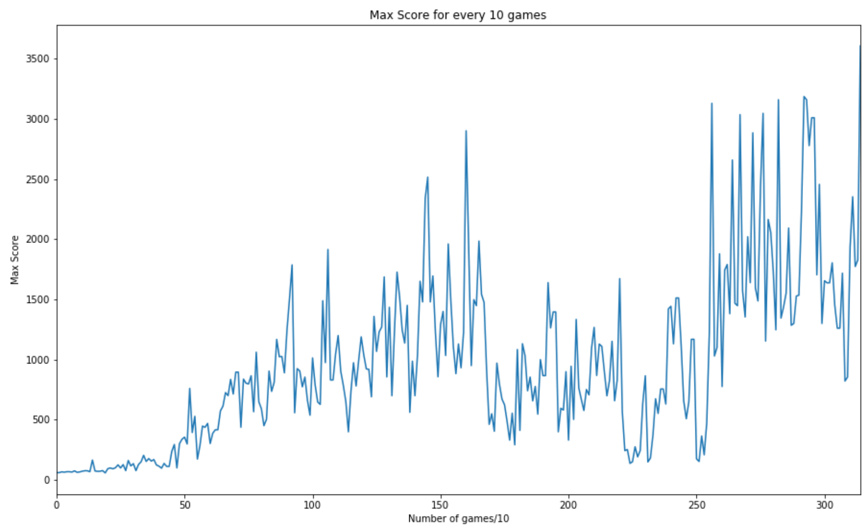

Figure 3: Scores of games played through Selenium - Max every 10 games

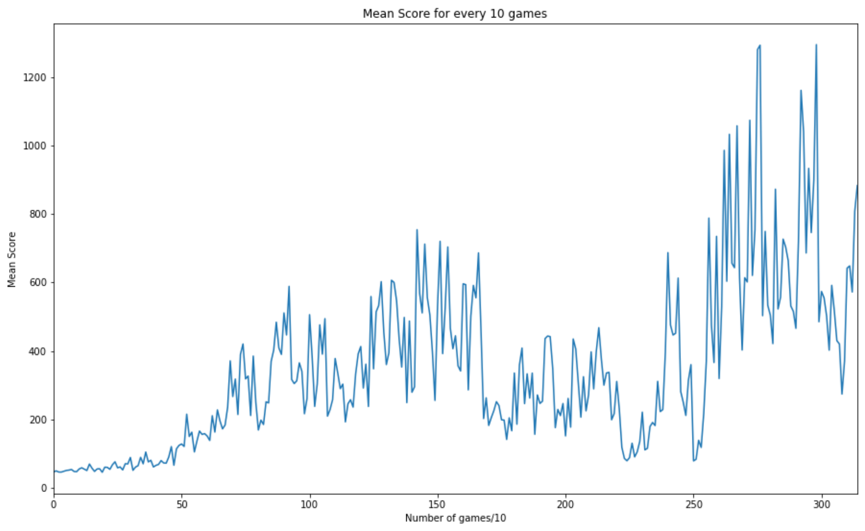

Figure 4: Scores of games played through Selenium - Mean every 10 games

The nonparametric Mann-Kendall test ([4]) for the detection of an upwards trend was applied to the full, non-aggregated, scores series.

The corresponding p-value is 0 and therefore the null hypothesis of absence of an upwards trend is rejected at all standard significance levels.

5 Discussion

As said in section 3.3 the time available was too short to try and explore possible improvements.

Anyway, despite the short training time and the lack of adequate hyperparameter optimization, the agent reached good scores and clear improvements through time, as the Mann-Kendall test confirms.

Thus, there is a lot of room for further development.

First of all, allowing more training time to the model, the ducking action could also be added to it.

In addition, the implementation presents many hyperparameters which have been selected a-priori without in-depth analysis. Their optimization would definitely be of paramount importance, as it could greatly affect model performance, albeit already sufficiently good.

But, more than anything, a trade-off between model performance and complexity should be found as, at the moment, the computational burden is way too high for the model to be utilized effectively on common hardware.

6 Conclusions

This paper showed the application of a deep convolutional neural network for Reinforcement Learning on Chrome's Dino Run game, using arrays of 4 subsequent frames as input.

Several computational problems and challenges have been encountered, which were solved referring to the existing literature and by using the *Google Colaboratory* environment. Finally, after 72 hours of training, the developed model has exhibited improvements and achieved good results, as shown in the graphs presented above.

Leaving aside time limits, further enhancements to the model would be possible, such as the adding of the ducking action and an in-depth analysis of hyperparameters.

References

[1] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PloS one*, vol. 12, no. 4, p. e0172395, 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] T. Carneiro, R. V. M. Da Nóbrega, T. Nepomuceno, G.-B. Bian, V. H. C. De Albuquerque, and P. P. Reboucas Filho, "Performance analysis of google colaboratory as a tool for accelerating deep learning applications," *IEEE Access*, vol. 6, pp. 61 677–61 685, 2018.

[4] H. B. Mann, "Nonparametric tests against trend," *Econometrica*, vol. 13, no. 3, pp. 245–259, 1945. [Online]. Available: http://www.jstor.org/stable/1907187