Project 4 - Reinforcement Learning

Bruno Carlos Luís Ferreira , **Vitalina Holubenko**
uc2015240050@student.uc.pt , uc2017255810@student.uc.pt
Universidade de Coimbra, Departamento de Engenharia Informática
Artificial Intelligence 2020/2021

Abstract

In this article we study, applying Reinforcement Learning, to design a automated agent to play the T-Rex Game, also known as the Dinosaur Game. In this tutorial, we will implement this paper using Keras. We'll start with the basics of Reinforcement Learning and Deep Q Learning and then dive into the code for a hands-on understanding of our solution for this problem.

1 Introduction

Making use of Artificial Intelligence (AI) and Machine Learning (ML) algorithms to teach autonomous agents how to play computer games has been widely discussed and investigated, owing to the fact that valuable observations and advancements can be made on the Machine Learning play pattern vs that of a human player, and such observations provide knowledge on how to further improve the algorithms. Dino AI provides a framework to play the in-built browser game of the Google Chrome web browser, and we are interested in using ML techniques to teach agents to play this game and reach high scores.
Reinforcement Learning (RL) is one widely-studied and promising ML method for implementing agents that can simulate the behavior of a player. In this project, we study how to construct an RL Dino controller agent, which can learn from the game environment. One of the difficulties of using RL is how to define state, action, and reward. We use the Deep Q-Learning algorithm to evolve the decision strategy that aims to maximize the reward.
The report contains a brief part on the theoretical knowledge needed about RL, Section 2 and 3, followed with our work and implementation of the agent specifically to this environment, Section 4. Section 5 provides evaluation results and some possible future work and improvements to be done.

2 Reinforcement Learning Base Knowledge

Reinforcement Learning (RL) is one of the most hot topics in machine learning right now, where an agent interacts with an environment by following a policy (π). In each state of the environment it makes the action based on the policy, and as a output receives a reward and transitions to a new state. The main goal of reinforcement learning is to learn an optimal policy in order to maximize the reward.
There are several RL algorithms which use the received rewards as the main approach to approximate the best policy, although the teaching process can be challenging (e.g a game when the reward is given when the game is won or lost or a self-driving car). To minimize this problem, we can manually make rewards functions which provide the agent with more frequent rewards[3].
RL can be divided in two main groups, the Model-Based Reinforcement Learning, that uses experience to construct an internal model of transitions and immediate outcome in the environment, and Model-Free Reinforcement Learning, these methods on the other hand, use experience to learn directly the state or/and the policy which can achieve the same optimal behaviour.
Statistically, the Model-Free methods are less efficient than the Model-Based methods due to the information of the environment being combined with the Model-Free methods, possibly containing erroneous estimates or beliefs about state values.

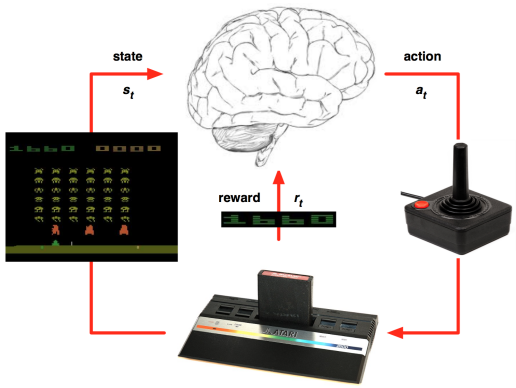

Figure 1: Reinforcement Learning Diagram

2.1 Model-Free RL

Model-Free Reinforcement Learning are methods used to solve MDPs without the knowledge of the reward function. These methods rely on MDPs to sample various sizes of experiences from the environment depending on the algorithm

used. These experiences are then used by an agent to directly make decisions and take actions in the environment.
Two main approaches to represent agents with model-free RL is Policy Optimization and Q-Learning.

Policy optimization
In policy optimization methods the agent learns directly the policy function that maps state to action. The policy is determined without using a value function.
Policy Gradient is a method were there are the policy π that has a parameter θ. This π outputs a probability distribution of actions.

$$\pi_\theta(\boldsymbol{a}|\boldsymbol{s}) = \boldsymbol{P}[\boldsymbol{a}|\boldsymbol{s}]$$

Figure 2: Probability of taking action a given state s with parameters theta

Then it's trying to find the best θ to maximize(optimize) a score function *J(θ)*, given the discount factor γ and the reward *r*.

$$\boldsymbol{J}(\theta) = E_{\pi\theta}[\sum]\gamma r]$$

Figure 3: Policy Score Function

Summing up, the main steps of Policy Gradient are:

- Measure the quality of a policy with the policy score function
- Use policy gradient ascent to find the best parameter that improves the policy

Q-learning
Q-learning is an off policy RL algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward. The **Q** stands for quality, representing how useful a given action is in gaining some future reward.

Steps of the Q-Learning method:

- Create a Q-table
- Q-Learning and making updates

Appliance of Q-Learning Algorithm in Python

```
import numpy as np
# Initialize q-table values to 0
Q = np.zeros((state_size, action_size))

import random
# Set the percent you want to explore
epsilon = 0.2
if random.uniform(0, 1) < epsilon:
```

Action

State	0	1	2	3	4	5
0	−1	−1	−1	−1	0	−1
1	−1	−1	−1	0	−1	100
2	−1	−1	−1	0	−1	−1
3	−1	0	0	−1	0	−1
4	0	−1	−1	0	−1	100
5	−1	0	−1	−1	0	100

Figure 4: A sample Q-table

```
    """
    Explore: select a random action
    """
else:
    """
    Exploit: select the action with max
    value (future reward)
    """
# Update q values
Q[state,action]=Q[state,action]+lr*
(reward+gamma*np.max(Q[new_state, :])
-Q[state, action])
```

Learning Rate (lr) can be defined as how much you accept the new value vs the old value.
Gamma *(γ)* is a discount factor. It's used to balance immediate and future reward. Typically this value can range between 0.8 and 0.99.
Reward *(reward)* is the value received after completing a certain action at a given state.
Max *(np.max())* uses the numpy library and is taking the maximum of the future reward and applying it to the reward for the current state, impacting the current action by the possible future reward.

3 DQN Learning

One of the problems that comes with Q-Learning is in instances when we are faced with a very complex environment. If we want the input to the RL agent to be as close as the input to a human, we will choose the input to be the array representation of the field which matches the visual representation of the environment. In this case, the environment would be complex, and if we were to use a Q-Learning the Q-Table would be tremendously big, making it computationally impossible to store each and every possible state.

So, due to this limitation, we have taken advantage of the Deep Neural Networks to solve this problem through regression and choose an action with highest predicted Q-value. A DQN, or Deep Q-Network, approximates a state-value function in a Q-Learning framework with a neural network. In the Atari Games and other similar games like Chrome's Dino take in several frames of the game as an input and output state values for each action as an output. In Deep Q Network a deep learning model is built to find the actions an agent can take at each state. The algorithm follows the next formula:

Algorithm 1 Deep Q-learning with experience replay

s

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q
for *episode* = 1 to M **do**
  Initialize sequence s_1 f = x_1 and preprocessed sequence $\phi_1 = \phi(s_1)$
  for t = 1 to T **do**
    With probability ϵ select a random action a_t
    Otherwise select a_t argmax$_a$ Q ϕ (s_t, a; θ)
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set s_{t+1} and preprocess $\phi_{t+1} = \phi\ \phi$ (s_{t+1})
    Store transition (ϕ_t, a_t, r_t, ϕ_{t+1} in D
    Sample random minibatch of transitions (ϕ_j, a_j, r_j, ϕ_{j+1} from D
    Set $y_j \leftarrow r_j$ if episode terminates at step j + 1
    Otherwise set $y_j \leftarrow r + \gamma$ max$_{a'}$ $\hat{Q}$ (ϕ_{j+1}, a'; θ^-)
    Perform a gradient descent step on (y_j - $Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q}$ = Q
  end for
end for

3.1 Technical Definitions

The basic terminologies of reinforcement learning include but are not limited to the following concepts:

1. Current state (*s*)
2. State at the next step (*s*')
3. Action (*a*)
4. Policy (*p*)
5. Reward (*r*)

From the state-action-value function (*Q(s, a)*) we get the expected total reward for an agent starting based on the current state and the output of it is known as the Q-value.

Our main goal is to choose a certain action (*a*) at a state (*s*) in order to maximize the reward (Q-value). DQN is a combination of deep learning and reinforcement learning. The model target is to approximate *Q(s, a)*, and is updated through back propagation. Assuming the approximation of *Q(s, a)* is *y(hat)* and the loss function is L, we have:

1. Prediction
$$y(hat) = f(s, \theta) \tag{1}$$
2. Loss
$$L(y, y(hat)) = L(Q(s, a), f(s, \theta)) \tag{2}$$

In the back propagation process, we take the partial derivative of the loss function to θ to find a value of θ that minimizes the loss. With the Bellman Equation we are able to find the ground-truth Q(s, a):

$$Q(s, a) = max(r + Q(s', a)) \tag{3}$$

where

1.
$$Q(s', a) = f(s', \theta) \tag{4}$$
if s is not the terminal state (state at the last step)

2.
$$Q(s', a) = 0 \tag{5}$$
if s is the terminal state

3.2 Train and Target Network

To get a consistent results we will train two models, the first model "train model" will be fit after every step made by the agent, on the other hand, the second model "target_model" loads the weights of "model" every n steps. We do this because in the beginning, everything is random (because the epsilon value is high), from the initial weights of the "train model" to the actions performed by the agent. This randomness makes it harder on the model to perform good actions, but when we have another model that uses the knowledge gained by the first model every n steps, we have some degree of consistency. The solution of DQN Learning is to create a target network that is basically a copy of the training model at certain time steps so the target model updates every predefined number of iterations.

3.3 Loss Function

In order to train the model to take the appropriate course of action at any given state, we need to express it as a formula that we can optimize on. The loss is just a value that indicates how far our prediction is from the actual target. For example, the prediction of the model could indicate that it sees more value in pushing the jump button when in fact it can gain more reward by doing nothing at that moment. We want to decrease this gap between the prediction and the target (loss). We will define our loss function as follows:

$$loss = (r + \gamma max(Q(s, a') - Q(s, a))) \tag{6}$$

We first carry out an action *a*, and observe the reward *r* and resulting new state *s*'. Based on the result, we then calculate the maximum target *Q* and then discount it so that the future reward is worth less than immediate reward (take the example of concept as interest rate for money. Immediate payment always worth more for same amount of money). In the end, we add the current reward to the discounted future reward to get the target value. Subtracting our current prediction from the target gives the loss.

Another issue with the model is over fitting. When the model is updated after the end of each game, we have already potentially played hundreds of steps, so we are essentially doing batch gradient descent. Because each batch always contains steps from one full game, the model might not learn well from it. To solve this, we create an experience replay buffer (memory) that stores the (s, s', a, r) values of several hundreds of games and randomly select a batch (of 32, in our case) from it to train the model at the end of each episode.

4 Dino Controller Design

4.1 Environment Module

For this work we used a open source library, *Gym* [1], to provide the environment, leaving to us just the responsibility of implementing the algorithm for the agent. For the initialization we simply use the *ChormeDino* environment, which is provided by the *Gym* open source library.

```
env = gym.make('ChromeDino-v0')
state = env.reset()
```

Using the *Tensorflow* [2] and *Keras* [3] libraries we can develop and train ML models using their high level APIs as explained in the next sections.

4.2 Hyper-parameters

There are some parameters that have to be assigned to a Reinforcement Learning agent.

1. Episodes (10000) - Number of episodes that the agent will play.
2. Gamma (0.95) - Discount rate (aka decay), to calculate the future discounted rewards.
3. Epsilon (1.0, initial value)- The exploration rate, in which the agent chooses to explore (take random actions) rater than exploit.
4. Epsilon Decay (0.99) - The epsilon value has to decrease over time in order to increase the rate of exploitation.
5. Epsilon Min (0.01) - Minimum value that the epsilon value can take. We want the agent to explore at least this amount.
6. Learning rate (1.0e-2) - Learning rate of the neural network, which controls how quickly the model is adapted to the problem.
7. Update Rate (500) - Rate of which the target network is updated with the train network's weights.

4.3 Image Processing

AI models don't need to calculate Q values every frame, that would lead to a great computational cost and it wouldn't be very efficient. So the skipping frames technique is based on performing the calculations of Q values every 4 frames and use past 4 frames as inputs (seen in the blend_images() function). As a result, this practice reduces computational cost and gathers more experiences. On top of that we perform downsizing of the frames to more or less half of the initial size, from 150x600 to 65x80 per frame, followed by grey scaling and consequently converting the image to black and white to improve its contrast.

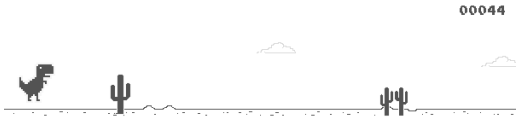

Figure 5: Original observation space (150x600)

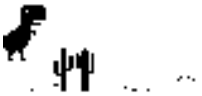

Figure 6: Processed observation space (65x80)

```
def process_frame(obs):
    # crop and downsize
    img = obs[20::2, 20:180:2]
    # to greyscale
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # to black and white
    img = cv2.threshold(img,
                        127,
                        255,
                        cv2.THRESH_BINARY)
    return img.reshape(65, 80, 1)

def blend_images(images, blend):
    avg_image = np.expand_dims(np.zeros(
                               (65, 80),
                np.float64), axis=0)
    for image in images:
        avg_image += image
    if len(images) < blend:
        return avg_image / len(images)
    else:
        return avg_image / blend
```

4.4 States

The environment used has already some predefined states that can be defined as:

- Done: True or False if agent is "alive" or not, being a direct consequence of the agent hitting or not an obstacle.
- Current state of the environment represented by an array.

4.5 Actions

The agent can perform 2 different actions that are 0: do nothing, 1: jump. The action that the agent chooses to take is either based on the current state of the environment, or is chosen randomly, following the ϵ greedy policy. Our agent will randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is done because at first, it is better for the agent to try all kinds of paths before it starts to extract the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action with the highest reward.

```
def get_action(self, state):
    # Random action
    if np.random.rand() <= epsilon:
        action = np.random.choice(2)
        return action
    # Exploit
```

```
    action_values = train_model.
                    predict(state)
    # Max value is the action
    action = np.argmax(action_values[0])
    return action
```

4.6 Rewards

The reward obtained is predefined with the *Gym* environment, where a positive value of 0.01 is given when the agent is alive throughout each step and a negative value of -1.0 is given when the agent collides with an obstacle, which might be a cactus or a bird.

4.7 DQN Model

There are several possible ways of parameterizing Q using a neural network. Because Q maps history–action pairs to scalar estimates of their Q-value, the history and the action have been used as inputs to the neural network by some previous approaches. We use a series of three Convolution layers before flattening them to dense layers and output layer. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. The input to the neural network consists of an 110x150x4 image produced by the pre-processing of the frames. The first hidden layer convolves 16 filters of 4x4 with stride 4 with the input image and applies a rectifier non linearity. The second hidden layer convolves 32 filters of 4x4 with stride 2. The third layer convolves 32 filters of 3x3 with stride of 1, followed by a layer that flattens the resuting array. The output layer is a fully-connected linear layer with a single output for each valid action.

```
def _build_model(self):

    model = Sequential()
    # Convolutional layers
    model.add(Conv2D(16, (8, 8),
              strides=4,
              padding='same',
              input_shape=self.state_shape))
    model.add(Activation('relu'))
    model.add(Conv2D(32, (4, 4),
              strides=2,
              padding='same'))
    model.add(Activation('relu'))
    model.add(Conv2D(32, (3, 3),
              strides=1,
              padding='same'))
    model.add(Activation('relu'))
    model.add(Flatten())
    # FC Layers
    model.add(Dense(self.action_size,
        activation='linear'))
    model.compile(loss='mse',
                  optimizer=Adam(1.0e-2))
    return model
```

Our output layers consists of two neurons, each one representing the maximum predicted reward for each action. We then choose the action with maximum reward (Q-value)

4.8 Memorizing

In order to keep an updated memory with the state-action pairs with the corresponding reward we implemented the following function.

```
def add_experience(state,
                   action,
                   reward,
                   next_state,
                   done):

    memory.append((state,
                   action,
                   reward,
                   next_state,
                   done))
```

In which memory is an array with a maximum length of 5000 experiences that stores various tuples of (current state, current action, reward, next state, done).

4.9 Training

The method that will train the neural net with experiences in the memory is train(). First, we sample some experiences from the memory and call them mini-batch. We set the batch size as 32 for this example. To make the agent perform well in long-term, we need to take into account not only the immediate rewards but also the future rewards we are going to get. In order to do this, we are going to have a 'discount rate' or 'gamma'. This way the agent will learn to maximize the discounted future reward based on the given state.

```
def train(self, batch_size):
    minibatch = random.sample(memory,
                              batch_size)
    for state, action, reward,
        next_state, done in minibatch:
        target = reward
        if not done:
            target = (reward + self.gamma *
                np.amax(train_model.
                predict(next_state)[0]))
        target_f = train_model.predict(
                                    state)
        target_f[0][action] = target
        train_model.fit(state,
                        target_f,
                        epochs=1,
                        verbose=0)
```

The full algorithm for training deep Q-networks is presented in Algorithm 1. The agent selects and executes actions according to an e-greedy policy based on Q. Because using histories of arbitrary length as inputs to a neural network can be difficult, our Q-function instead works on a fixed length representation of histories produced by the function described above.

First, we use a technique known as experience replay in which we save the agent's experiences at each time-step, throughout many episodes (where the end of an episode occurs when a terminal state is reached) into a replay memory. During the inner loop of the algorithm, we apply Q-learning updates, or mini-batch updates, to samples of experience, drawn at random from the pool of stored experiences. This approach has several advantages over standard online Q-learning. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency.

Secondly, learning directly from consecutive samples is extremely inefficient, owing to the strong correlations between the samples, randomizing the samples will break these correlations and therefore reduces the variance of the updates.

Thirdly, when learning on policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to jump then the training samples will be dominated by samples from the jumping side action (1).

It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. By using experience replay the behaviour distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.

In practice, our algorithm only stores the last pre-defined N experience tuples in the replay memory, and samples uniformly at random from when performing updates. This approach is in some respects limited because the memory buffer does not differentiate important transitions and always overwrites with recent transitions owing to the finite memory of the experience buffer. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. A more optimized and sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping.

5 Analysis

5.1 Evaluation Results

The results obtained in the end did not achieve the initial expectations that were created when this environment and agent have been chosen. The initial expectation was that at the end the agent evolved could reach some high scores, like 500 or even 1000. But as we can analyse, the results obtained were not that exciting.
In the Figure 7 we can see that the agent as a behaviour extremely random even after hundreds of episodes. Some strong variables contribute to that fact but never the less, the resulting behaviour of the agent has not met our expectations.

Analysing the Figure 8, a graph that represents the epsilon (exploration rate) over the episodes, we can assume that the epsilon follows the behaviour of an exponential fall function.

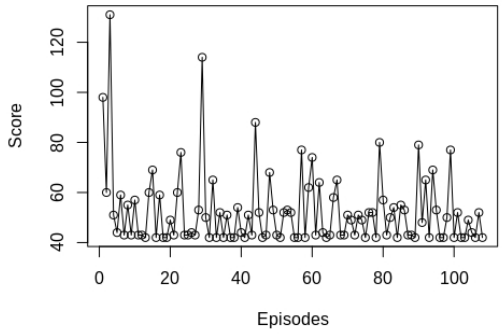

Figure 7: Evaluating Score

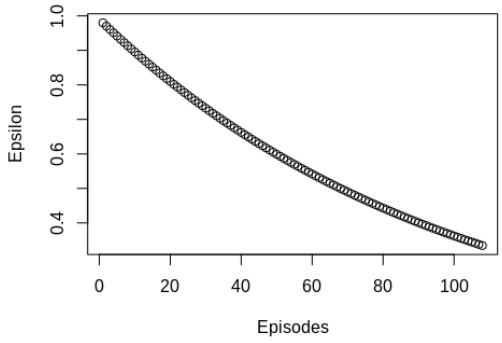

Figure 8: Epsilon over Episodes

5.2 Difficulties and Further Work

The results obtained were strongly connected with some important variables that our experiment was dependent of:

- We used our personal laptops that do not have dedicated GPUs and have an low memory RAM (for ML use cases) causing our training to be much slower and limiting us of using better parameters because of physical limitations of our components.
- The environment is not constant. Every new episode contains a different state of environment, like the positions of the cactus and the length of cactus, making the training of an agent much more difficult compared with an constant environment. To obtain better results with an non constant environment we needed to not be dependent on the previous difficulty.

Some further work that can be done in short time is to attempt to have access to better test machines and optimize parameters. Some parameters that have enormous importance in the training of an agent are:

- Number of Episodes

- Max and Min Experience
- Epsilon Value and its decay over time
- The neural network model itself

6 Conclusion

This paper introduced our version of new deep learning model for reinforcement learning that was based in previous related works in the field, and demonstrated how we could approach a problem such as this, using only raw pixels as input.

In this project, we designed an automated agent using Reinforcement Learning to play the Dino game.

The learning algorithm we presented is a variant of Deep Q-learning that makes use of stochastic mini-batch updates with experience replay memory to ease the training of deep networks for Reinforcement Learning. Our approach still has to reach state-of-the-art results in the game it was tested on, with some adjustment needed to be made for the architecture, perhaps a more simple architecture could be applied instead of the three convolutional layers, or hyper-parameters. A different algorithm altogether could be applied, like Expected Sarsa or a Double DQN.

The results obtained were not optimal due to some variables that were present in our tests as explained more specifically in section 5. But nevertheless, our results were not optimal as we like them to be.

We believe that our work provides a good introduction to this problem and will be of use for the people with initial interests in using reinforcement learning to play computer minigames, being clear in the implementation used and not hiding the problems and bad results that we were faced with.

7 Appendix

- Instructions: README.md
- Code: Dino-RL.zip
- Git Repository: https://github.com/vitalinarh/Dino-RL

References

[1] https://gym.openai.com/

[2] https://www.tensorflow.org/

[3] https://keras.io/