Experiments with Reinforcement Learning for a Neural Network to Play Dino
Modality: 2

Henrique Coutinho Layber
Vitória, Espírito Santo, Brazil

Abstract

Dino is a simple game where the player controls a dinosaur that has to dodge obstacles. The game is simple enough that a neural network can learn to play it, but it is also complex enough that it is not trivial to play. In this work, we present a neural network that plays the Dino game using Reinforcement Learning. The neural network is trained using Simulated Annealing. The neural network is evaluated by the score it achieves in the game. The results show that it is able to learn to play the game, and that it is able to achieve a higher score than a simple heuristic. It is also able to achieve a higher score than a neural network trained by the professor.

Keywords: Machine learning, neural network, Reinforcement Learning

1. Introduction

Dino[1] is a runner game featuring a dinosaur as the player, the goal is to score as high as possible dodging obstacles. The game is very simple, thus a good example of a game that can be played by a neural network. The game is simple enough that a neural network can learn to play it, but it is also complex enough that it is not trivial to play.

One of the ways a neural network can learn to play a game is through Reinforcement Learning. Reinforcement Learning is a type of Machine Learning

[1] Officially available on `chrome://dino`, for Chromium browsers only

where an agent learns to behave in an environment, by performing actions and seeing the results of those actions. A common implementation of Reinforcement Learning is with a search heuristic and a classifier. The search heuristic is used to explore the environment (usually neural network's parameters) and the classifier is used to predict the best action to take.

All code for this experiment is available at `https://github.com/Henriquelay/AI-classes/tree/main/Trab2/app`

2. Classifier Description

For this experiment, the classifier is a neural network specialized in only seeing the immediately next obstacle. It is implemented as a feedforward network with 3 layers: the input layer, the hidden layer, and the output layer. The input layer has 3 neurons: distance to the next obstacle with a predicted physics step built-in, height of the next obstacle, speed of the game. The hidden layer has 2 neurons activated with ReLU and the output layer has 1 neuron activated with sigmoid. If the output is greater than 0.55, the player jumps, otherwise it ducks.

To figure out how to design the neural network, several trials were made.

2.1. Input manipulation

By far the most effective way to impact the classifier was to change the input. As the saying goes, "*garbage in, garbage out*". By diminishing the amount of *garbage in*, we can improve the classifier massively with simple changes.

2.1.1. `obType`, `obHeight`

The only scenario where knowing the type of the object would be useful is when facing a high object you *maybe* could duck under. You can duck under a high enough bird, but have to jump over a cactus regardless of its height. Since there is no other scenario where knowing the type of the object is useful, unless it can be used to differentiate between a high bird (which you can duck under) and a similarly high cactus (which you can't duck under), it is useless.

$\exists c \exists b[iscactus(c) \wedge isbird(b) \wedge height(c) > height(b) \wedge isduckable(b)] \leftrightarrow obType$ is useful

However, this scenario is impossible to happen in the implemented game, since birds can only have a height of $\{38, 83, 123\}$, and cactuses can only have a height of $\{38, 58\}$, and the player can only duck under birds of height 83 or more, there is no cactus higher than a bird that can be ducked under, so differentiating duck-able obstacles is unneeded, since you can always duck under obstacles over 58 high. Thus, `obType` is useless.

Also because of this, `obHeight` can be simplified further with an arbitrary value of 100 when the players should jump.

```
obHeight = 100 if obHeight < 83 else 0
```

This makes learning process much quicker, and the height difference only matters with extremely high level play, where the falling player's hitbox can dodge a low object.

2.1.2. `distance`

`distance` is actually the distance to the first rendered obstacle. This means that it can assume negative values, which is not ideal for the neural network, since the obstacle is already beaten when `distance` is negative. To fix this, when `distance` is negative, it is set to the distance of the next obstacle. Importantly, the `obHeight` follows.

Also, a simple change to the `distance` is to make it a prediction of the future, by subtracting the speed of the game from it. This way, the neural network can take actions earlier (like jumping early). This impacts the learning process at higher scores.

```
if distance <= 0:  # if already past, look at next obstacle
    distance = nextObDistance
    obHeight = nextObHeight
distance -= speed
```

3. Metaheuristic Description

Simulated Annealing is a metaheuristic that is used to find the global optima of a function. It is inspired by the annealing process of metals, where the metal is heated and then slowly cooled down to remove impurities. The algorithm works by starting at a random point and then moving to a new point based on a probability distribution. The probability distribution is based on the difference between the current point and the new point, and the temperature of the system. The temperature is slowly decreased over time, which causes the algorithm to narrow it's search space.

For this implementation, $energy = score$ was used. To avoid noise, the agent plays the game 15 times and the $avg - std$ of all scores is used as the energy.

3.1. Temperature x Cooling Rate

There are multiple options for how to cool our system. The temperature can be decreased by a constant amount, by a percentage, or by a function. The cooling rate is a metaparameter that determines how fast the temperature decreases. Some well-established functions were experimented with:

$$boltzmann(e) = \frac{t_0}{\log(e+1)} \tag{1}$$

$$exponential(e) = t_0(1 - C_r)^e \tag{2}$$

$$geometric(e) = \frac{t_0}{1 + C_r \cdot e} \tag{3}$$

where: e = epoch

t_0 = initial temperature

C_r = cooling rate

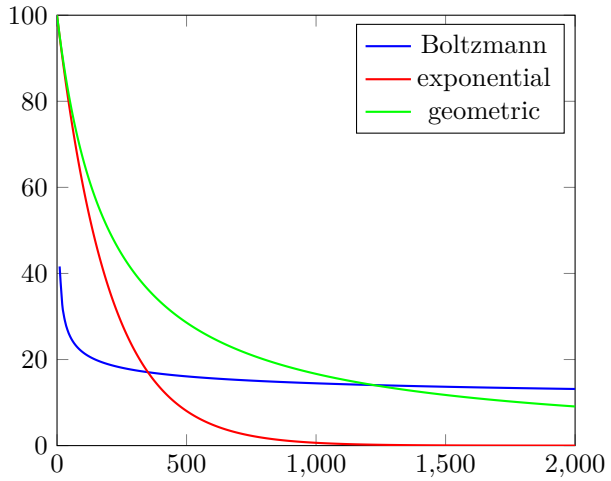

Figure 1: Temperature functions

For shorter training sessions, it is found that a temperature of 100 and a cooling rate of 0.005 with *Boltzmann* works well. For longer training sessions, a starting temperature of 200 and a cooling rate of 0.003 *geometric* was used to great effect, although more experimentation is needed for longer training sessions. *exponential* was found to be too aggressive, and the temperature would drop too quickly, narrowing the search space too fast.

3.2. Perturbation

The perturbation is the function that generates a new state based on the current state. For this implementation, the perturbation is a random number generated between 0 and 1, affected by a damping factor, which takes current temperature into account. The perturbation is then multiplied by temperature summed to the original value. This way, we can scan a huge area initially with big perturbations and decrease the perturbation as the temperature decreases, which makes the algorithm narrows its search space as it goes on.

```
def perturb(weight: float) -> float:
    def perturbation() -> float:
        damper = self.temperature / (self.temperature + 10)
        return np.random.uniform(-damper, damper)
```

```
    p = perturbation()
    # print(f"{p=}")
    result = weight + self.temperature * p
    return result
```

4. Results

Here follows results for 30 rounds of Dino after a long (around 8 hours) training session[2].

[2]The agents did not play the same game simultaneously, because the professor only provided the results of his classifier. So they played different rounds

	Ours	**Professor**
	3311.00	1214.00
	3289.25	759.50
	3622.25	1164.25
	3237.25	977.25
	3705.75	1201.00
	3155.00	930.00
	3519.25	1427.75
	3728.50	799.50
	3716.25	1006.25
	3205.75	783.50
	3327.75	728.50
	3539.75	419.25
	3509.50	1389.50
	3680.50	730.00
	3270.50	1306.25
	3147.75	675.50
	3607.50	1359.50
	3632.75	1000.25
	3318.00	1284.50
	3282.25	1350.00
	2805.50	751.00
	3224.50	1418.75
	3576.50	1276.50
	3177.00	1645.75
	2827.75	860.00
	3266.75	745.50
	3069.50	1426.25
	3525.50	783.50
	3555.00	1149.75
	3625.25	1482.25
Mean	3381.99	1068.18
Std	246.54	304.04
t-test	$2.00e-38$	
Wilcoxon	$1.86e-09$	

Table 1: Score comparison

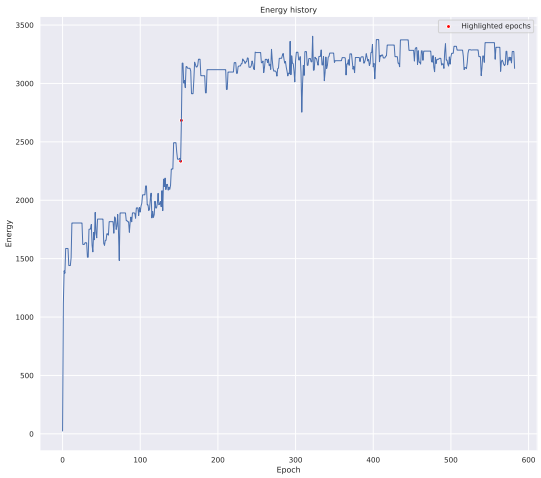

Figure 2: Energy (score) over epochs

Here we include a third agent (*simplest*), which is included with the game's implementation. The agent plays the same round as our agent, but it is not a neural network, it is a simple heuristic that jumps when the obstacle is close.

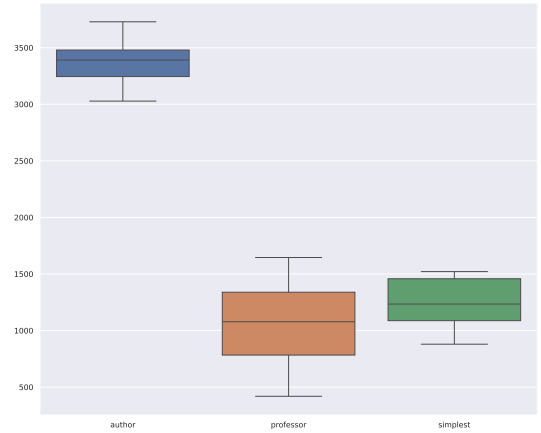

Figure 3: Score over epochs

5. Conclusions

5.1. General Result Analysis

Aside from the initial burst to ≈ 1200 when the agent learns to basic jump and duck, we can see that from epoch 152 to 154 there is another huge jump in score. This is when the agent learns to jump earlier at high speeds. Jumping late at high speeds is a common way to lose the game, since it's likely that the player can't jump the following obstacle due to landing on it, or not having enough time to get high enough.

We can see that our agent is statistically different from the professor's agent, with a p-value of $2.00e-38$ for the t-test and $1.86e-09$ for the Wilcoxon test.

5.2. Contribution

The main contribution of this work is the implementation of a neural network to play the Dino game, using Reinforcement Learning.

The single biggest improvement was the input manipulation, which made the learning process much quicker. In special, the `obHeight` manipulation was the most impactful, since it made the learning process much quicker and the network design simpler.

The second-biggest improvement were the cooling functions experiments, which made the learning process more stable and resilient to noise.

5.3. Future Work

The main limitation of this work is the lack of a proper evaluation of the neural network. The neural network was only evaluated by the score it achieved, but it would be interesting to evaluate it in other ways, such as the number of jumps it made, the number of obstacles it dodged, and the number of times it died. It would also be interesting to evaluate the neural network on other games, to see how well it generalizes.