AI Runner Game Bot

Arvind Kumar, Pavithran Ramachandran, Vaibhav Rangarajan
Northeastern University, Boston, Massachusetts
kumar.kal@husky.neu.edu, ramachandran.p@husky.neu.edu, rangarajan.v@husky.neu.edu

Abstract

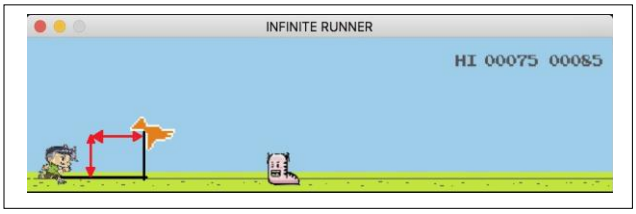
The objective of this project is to simulate a two dimensional object avoidance bot using *reinforcement learning* on a game and comparing the results by implementing the game using *genetic algorithm*. The game designed for the comparison is an infinite runner game designed using pygame. The motive of this project is to introduce obstacles in an infinite runner game and observe how an artificial intelligent agent learns to avoid the obstacles.

1 Introduction

This project aims to study the behavior of an Artificial Intelligence bot that tries to address the object avoidance problem found in autonomous cars, robots etc. It is a miniature representation of a real time object avoidance problem which is constrained in a two dimensional environment with lesser degree of freedom. We will be introducing many obstacles on the way of an infinite runner bot in a game and will be analyzing how the bot learns to avoid the obstacles using Q-learning algorithm.

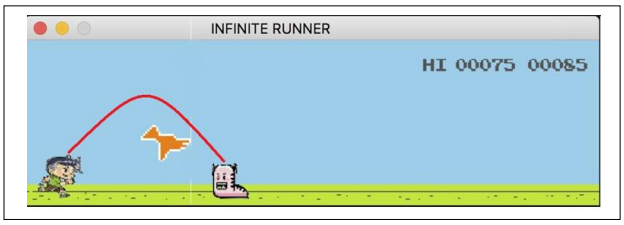
1.1 Issues and Challenges

1.1.1 State and Action Representation



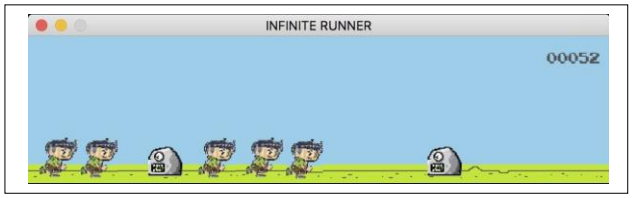
The various actions the runner bot can perform are run, jump or slide. There are two different types of obstacles, worm obstacles on the ground and bird obstacles in air. There are a number of combinations of state action pair and hence the exploration might take a lot of time to explore all the states and learn the optimal action to take in each state which is one of the challenges.

1.1.2 Guaranteed Failure



The pictorial representation above shows a guaranteed failure. In such cases the runner bot always loses irrespective of the action it takes. Hence it might assume that when it comes across a bird obstacle it shouldn't jump and update its training data accordingly. Hence despite lot of training and exploration, the runner bot will again fail thinking that jump is not the right action against the bird obstacle.

1.1.3 Multi-agent training



Introducing multiple agents or bot runner in this case, speeds up the training process and exploration of state space. But, the issue here is how we are going to represent the training data of each of the runner bots. Aggregating the training data to find the best possible action and memory for storing the training data should also be considered.

1.4 Approach Overview

The infinite runner bot game was developed in python language using pygame package. We have implemented two algorithms,

- Reinforcement Learning (Q-Learning)
- Search heuristic (Genetic algorithm)

The various issues and challenges stated before are overcome as follows:

- For training the runner bot of such large state space, we have introduced manual intervention through which the human can train the runner bot by taking control and playing the game. By this way the exploration of state space would largely be reduced.
- In case of a guaranteed failure, we have set the reward in such a way that, the scenario does not affect the training data or q value table.
- For multi agent training one approach was implemented such that there is only one q table for all runner bots and the q-value is averaged for every runner bot's experience.
  The other approach that we tried out was having one q table for each runner bot and maintaining one master table. Once all bots die in iteration, max of all q-values are populated in the master table. In the next iteration, the runner bots will use the master copy of training data and again when all bots die, the master table is updated.

2 Related Work

2.1 Deep Q-Network (DQN)

In the tabular representation of state space, the search space increases exponentially with increase in the parameters to consider. To combat this issue, the paper "*Playing Atari with Deep Reinforcement Learning*", utilises high dimensional states like images represented as Neural Network to learn Q-values. For example, a simple tic-tac-toe game has approximately 27,000 possible game combinations. Which when represented as an in-memory table may not scalable. The infinite runner has multiple parameter dependencies that need to be referred to get the optimal action from the state-action table which can increase exponentially. The paper discusses using a gradient descent to train neural networks and learning rate (alpha) is included in the gradient decent algorithm we can drop it from Bellman's equation.

2.1.1 Preprocessing

In pre-processing stage, noisy parts from the images are eliminated. Also multiple images are collected into a single state which adds a kind of memory. This highly accelerates the Q-learning.

2.1.2 Learning

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample$$

Where *Q(s, a)* denotes that the Q-function that is built out of a neural net with parameters θ. Minimizing L(θ) w.r.t. θ is used to optimize the neural network parameters.

While this approach works in theory, in practice during the optimization the neural net tends to oscillate or diverge.

- Experience Replay
  Instead of using only a single SARSA observation in the error function L(θ) we use a batch of observations. This breaks correlations in data, reduces the variance of the gradient estimator, and allows the network to learn from a more varied array of past experiences.
- Target Network
  In order to avoid oscillations, we will use a different network, called the target network, which has fixed parameters. The loss is now calculated as,

$$\mathcal{L}(\theta) = \left(Q_\theta(s,a) - (r + \gamma\, max_{a'} Q_{\theta^*}(s',a'))\right)^2,$$

Where $Q_\theta(s,a)$ is the main network and $Q_{\theta^*}(s',a')$ is the target network.
Importantly, both networks have the same architecture but can have different values for the parameters. Periodically we need to update the target parameters with the newest value, i.e. $\theta^* \leftarrow \theta$. By using this separate network to compute the targets more stable training procedure with reduced number of constantly shifting values in the loss function is obtained.

3 Implementation Approach

3.1 Game State Space Representation

The state has following parameters which change at each tick of game play. There are,

- Horizontal distance to the closest obstacle
- Vertical distance to closest obstacle
- Type of obstacle
- Speed of game simulation

The possible actions of the runner bot are,

- Run
- Jump
- Duck

3.2 Q-Learning

Q-learning is a type of reinforcement learning. Reinforcement learning involves an agent, a set of states *s*, and a set of actions per state *a*. By performing an action *a*, the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score). The goal of the agent is to maximize its total (future) reward. It does this by learning which action is optimal for each state. The action that is optimal for each state is the action that has the highest long-term reward. This reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state.
The algorithm has a function that calculates Quality of a state-action combination.

Q: S * A → R

In our game, the state action pair is stored in the q-table and the reward is calculated based on the output, i.e. alive or dead.
The exploration probability is modeled similar to simulated annealing technique by employing a non-zero constantly non-increasing value with the runner bot scoring more than 200. This discourages the bot to take random exploration actions which may lead to death after reaching a threshold score.
In our implementation of Q-Learning, for every game played the runner bot observes the state it had been in, the actions it had taken from the set of available actions and the reward it received for landing on a particular state. With respect to the action it had taken, the bot is punished (-100) or rewarded (+1) based on the state it lands on. After the runner has played the game 'n' times, the bot should avoid obstacles whilst achieving a decent score.
The formula for computing the Q value of the runner bot is as follows,

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

Learn the Q(s, a) values of the runner bot for every trial,

- Receive the sample (s,a,s´,r)
- Consider the old estimate : Q(s,a)
- Consider your new sample estimate:

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

- Incorporate the new estimate into running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

3.2 Multi-agent Reinforcement Learning

Although most work on reinforcement learning has focused exclusively on single agents we can extend reinforcement learning straightforwardly to multiple agents if they are all independent. They together will outperform any single agent due to the fact that they have more resources and a better chance of receiving rewards.
There are three ways of cooperation;

- Agents can communicate instantaneous information
  such as sensation actions or rewards.
- Agents can communicate episodes that are
  sequences of sensation action reward triples experienced by agents
- Agents can communicate learned decision policies.

In our AI runner bot game we implemented two approaches of above mentioned multi-agent reinforcement learning techniques. One approach was implemented such that there is only one q table for all runner bots and the q-value is averaged for every runner bot's experience.
The other approach that we tried out was having one q table for each runner bot and maintaining one master table. Once all bots die in iteration, max of all q-values are populated in the master table. In the next iteration, the runner bots will use the master copy of training data and again when all bots die, the master table is updated.

3.3 Active Reinforcement Learning

Active learning is a special case of semi-supervised machine learning in which a learning algorithm is able to interactively query the user (or some other information source) to obtain the desired outputs at new data points.
Instead of collecting all the states for all the data at once, Active Learning prioritizes which data the model is most confused about and requests states for just those. The model then trains a little bit on that small amount of state data, and then again asks for some more states for the most confusing data.
By prioritizing the most confusing examples, the model can focus the experts on providing the most useful information. This helps the model learn faster, and lets the experts skip labeling data that wouldn't be very helpful to the model. The result is that in some cases we can greatly reduce the number of states we need to collect from experts and still get a great model.
For training the runner bot of such large state space, we have introduced manual intervention through which the human can train the runner bot by taking control and playing the game. By this way, the users can train the runner bot to maneuver at difficult and crucial stages in the game.

3.4 Genetic Algorithm

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution.
The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- Selection rules select the individuals, called parents that contribute to the population at the next generation.
- Crossover rules combine two parents to form children for the next generation.
- Mutation rules apply random changes to individual parents to form children.

In our runner game bot project, genetic algorithm is implemented as follows:

- **Initialise Population**: Create 10 bots, each bot at random location in the game.
- **Evaluation**: Evaluate Bots at each stage using the fitness function as total distance/score.
- Sort all bots of current population in decreasing order of their fitness ranking.
- **Selection**: Select 4 best bots that are passed to the next population.
- **Crossover**:
    - 1 offspring is made by a crossover/average of two best winners
    - 3 offspring are made by a crossover/average of two random winners
    - 2 offspring are direct copy of two random winners
- **Mutation**: Add some variations; apply random mutations on each offspring.

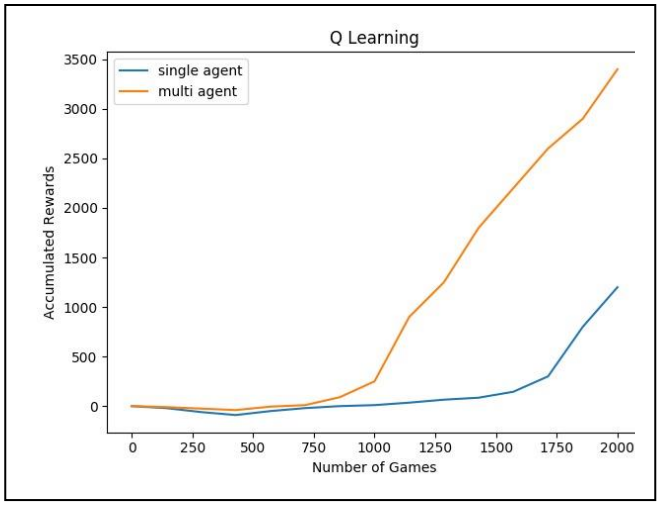
In this approach, we are having one q table for each runner bot and maintaining one master table. Once all bots die in iteration, based on selection, crossover and mutation, training data are populated in the master table. In the next iteration, the runner bots will use the updated master copy of training data and again when all bots die, the master table is updated.

Evaluation expression for Genetic Algorithm:

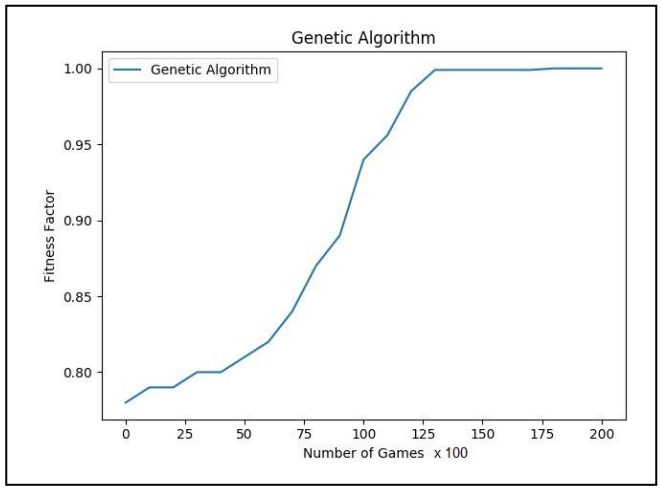
sig (score) = $1/ 1+ e^{-t}$
where 1 is added to denominator to avoid divide by zero exception.

4 Evaluation

We observed the maximum rewards obtained from Bellman's equation to evaluate Q-learning for single and multi-agent and found that multi-agent learns quicker compared to a single agent.



To evaluate performance of genetic algorithm, we plotted the fitness function with respect to the number of games played. We observed that the fitness function converged to an ideal value 1. We also found that genetic algorithm in this case converges quickly compared to Q-learning.



To attain a score of 1000, the Q-learning bot needed 6 hours whilst genetic algorithm took 8 hours. Multi-agent Q-learning outperforms genetic algorithm in this case.

5 Acknowledgements and Contribution

We would like to thank our Professor Stacy Marsella for his constant guidance and support throughout the project reviews. The contribution of our team members are listed below;

- Aravind Kumar – Genetic Algorithm, Multi agent reinforcement learning, Game GUI
- Pavithran Ramachandran – Q-Learning, Active Reinforcement learning, Game GUI
- Vaibhav Rangarajan – Q-Learning, Genetic Algorithm, Game GUI

Programming Language used – Python
Packages used – pygame, JSON

6 References

[1] Mnih, Volodymyr et al. "Playing Atari with Deep Reinforcement Learning." *CoRR*abs/1312.5602 (2013)

[2] Mnih, Volodymyr et al. "Human-level control through deep reinforcement learning." *Nature* 518 7540 (2015): 529-33.

[3] Tan, Ming En B. "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents." (1993).

[4] Reinforcement Learning with Multiple Shared Rewards

Anderson, Eike. (2003). Playing Smart -Artificial Intel ligence in Computer Games

[5] "Genetic Algorithm with python"- Clinton

Sheppard - CreateSpace Independent Publishing Platform (April 29, 2016).

[6] Riechmann, Thomas. “Genetic algorithm learning and evolutionary games.” (2001).

[7] Busoniu, Lucian et al. “Multi-agent reinforcement learning: An overview.” (2010).

[8] Egorov, Maxim. “Multi-Agent Deep Reinforcement Learning.” (2016).

7 Appendices

Attached code:

AI-RUNNER-BOT

- main.py - Single player Q-learning game
- bot.py - Single agent bot
- q_values.json - Q table

Genetic

- main.py - Multi player game
- bot.py - Multi agent bot
- bot_MA.py - Genetic algorithm bot
- q_values{1-10}.json - Q table