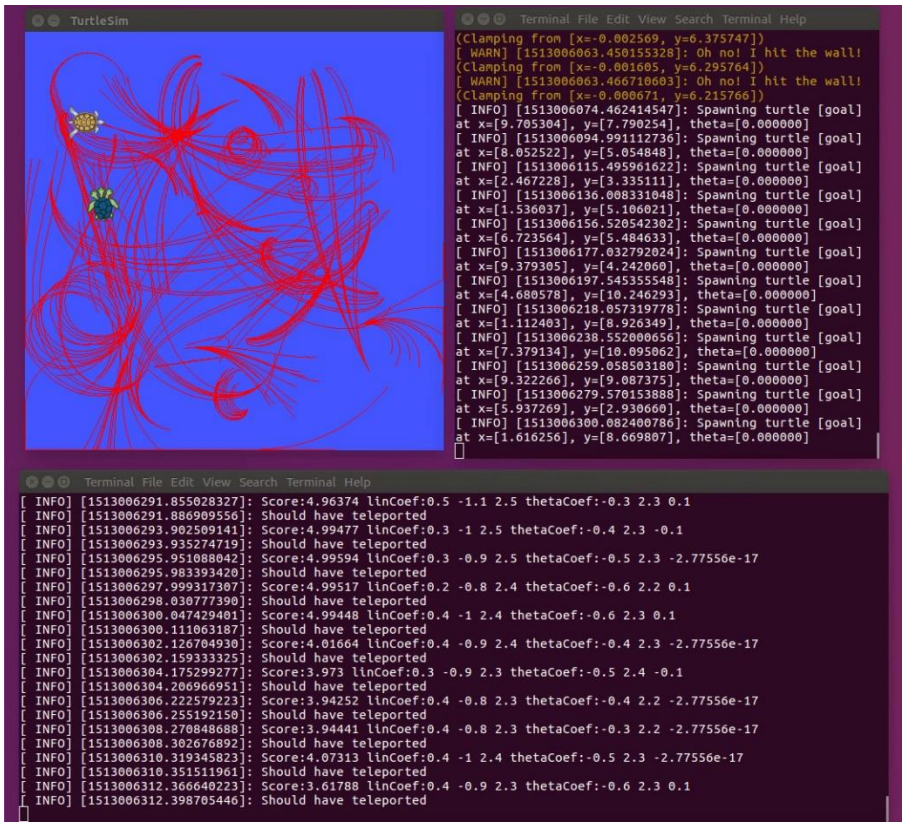
CIS 579 Project: Self-taught AI Control



DEARBORN

COLLEGE OF ENGINEERING & COMPUTER SCIENCE

By Kenneth Yesh (kyesh@umich.edu|05502176)

For Dr. Ortiz, Fall 2017

Abstract

The project was to create an AI that can learn how to navigate a to a specified x,y position given a starting x,y position. This will be done using Turtle Sim as a simulator which reports back the turtle's x,y and theta and can take in linear and angular velocities as inputs. The plan was to use re-enforcement learning to train the controls.

1. Introduction

After reading up on reinforcement learning there did not seem to be a lot of information available on how to deal with a continuous state and action space which is what my simulator worked in. From the material I looked over a policy search sounded like the best method as it only required new polices to be generated and the end result to be evaluated.

2. Background

The ROS framework was chosen for this project as I already had pre-existing experience with it and it will make the results of my project easier to integrate into future projects. ROS is an effort to standardize communication between different applications and sub-modules that are used in robotics applications. The goal is to create a plugin and play approach for robotics so people can easily plugin and test/compare new algorithms on the same platform.

ROS two main ways for nodes (or sub-modules) to communicate with each other. Messages are used to send data streams to one or many nodes who are listening for it. Services are used for less frequently needed communications which typically entail some type of configuration action to be done.

TurtleSim which was used as the simulator for this experiment was built as a basic ROS experimentation and training application. It provides a turtlesim/Pose message containing x, y, theta, linear velocity, and angular velocity of a turtle. It also accepts a geometry_msgs/Twist message taking a linear and angular velocity for the turtle to follow. I used the turtlesim/Pose message to gather state information about the turtle and the geometry_msgs/Twist message to issue actions to the agent.

TurtleSim has several services that were used to setup each experiment. turtlesim/TeleportAbsolute is used for placing the turtle in its starting position for the experiment. turtlesim/SetPen was used to pick up the pen when the turtle teleported removing undesired traces from the visualization. turtlesim/Spawn and turtlesim/Kill are used to add and remove a turtle at the goal position to make it clear where the turtle acting as the primary agent is trying to go.

3. Algorithm

The algorithm was developed iteratively originally starting with a linear combination of two derived features extracted from a combination of low level features.

Low Level Feature	Description
xt	Turtle's x position from turtlesim/Pose message
yt	Turtle's y position from turtlesim/Pose message
theta	Turtle's theta position from turtlesim/Pose message
xg	Randomly generated x coordinate of goal
yg	Randomly generate y coordinate of goal

Derived Feature	Equation	Description
thetag	atan2(yg-yt, xg-xt)	Angle of the vector going from the current position to the turtle to the goal position
thetad	angleDiff(thetat,thetag)	The difference between the turtle's theta and the theta it needs to be pointing in to be driving towards the goal.
d	$\sqrt{(xg-xt)^2+(yg-yt)^2}$	The distance from the turtles current position to the goal

The initial policy was of the from the from:

$$LinearVelocity = A \times thetad + B \times d$$

$$AngularVelocity = C \times thetad + D \times d$$

A, B, C and D were all varied randomly by 0.1 up, down or no change. The new policy would be compared to the previous policy and the one with the higher score was kept.

Scoring EQ:

(5.0 - distance(state.x, state.y, XG, YG));

Learning was achieved using the *rosSpinFor* function I wrote. This function essentially let the policy run for the number of seconds given as a parameter. At the end of that time the turtle was given a stop message and it's score was computed using the turtles current position and the goal it was trying to reach. If it scored as good or better than the previous attempt it would save the coefficients that were used. If it preformed worse it reset to the previous coefficients.

This first variation of the policy was ran both with unbounded actions and bounded actions. My initial bounding range of 0 to 1 linear velocity and -1 to 1 angular velocity was too restrictive for the turtle to learn anything in the 1 second I gave it get to the goal. The bounds I ended up using were 0 to 5 and -5 to 5 on the linear and angular velocities respectively.

Next was to get the algorithm to generalize to for any set of starting and goal states. This was achieved by running each randomly generated pair of starting and goal states for ten runs using the original algorithm described by starting with the best coefficients from the previous one and resetting

the initial score. The form of the policy also needed to be updated to account for the thetad component in the linear velocity which should really be a magnitude of the error instead of the direction as well as adding a generic bias.

New Policy Form:

$$LinearVelocity = A \times abs(thetad) + B \times d \ + \mathrm{E}$$

$$AngularVelocity = C \times thetad + D \times d + \ \mathrm{F}$$

4. Results

The initial policy form worked well for a single starting-goal state but did not generalize well to any set of starting-goal states. This was mostly likely due to the thetad contribution to Linear Velocity being negative when the goal is to the left and positive when the goal is to the right. This was corrected for by taking the absolute value of thetad.

Initializing the linear coefficients to a slightly positive value helped with convergence. Otherwise you risk the linear component becoming negative which truncates to 0 linear velocity. Then changes in policy to do not result in any change of score and the turtle just spins in circles.

When the algorithm worked it took about 20 minutes to converge to a reasonable set of results. A video can be viewed at the following link.
https://drive.google.com/a/umich.edu/file/d/1SyuT4YpFo8GdVvJU0j1-5_rBBi8rjjFe/view?usp=drivesdk

Plots of the score over 10 runs can be found here.
https://github.com/kyesh/ReinformentLearningWithTurtleSim/blob/master/catkin_ws/Plot%20Runs.pdf

I think evaluation between runs would be more consistence/clear if there were a set of scoring goal start point pairs to evaluate progress against. It is hard to tell if the score is from turtle performing well or just starting near/far from the goal.

5. Future Work

I think it would be worthwhile to continue investigating this problem space by using the time it reaches the goal as part of the reward function. That way the algorithm will find the fastest way to get to the goal instead of the way to be as close to the goal as possible after two seconds.

It would also be interesting to see how adding more complicated policies forms could result in potentially faster paths to the goal and what they cost in additional time to converge.

It would also be interesting to see how a one parameter at a time update would compare to the random parameter update I used.