I N S T I N C T

(It doesN't STand for anythINg and it Can'T anymore)

Alexander Lamson
University of Massachusetts Amherst
Amherst, MA
alamson@umass.edu

Christopher Raff
University of Massachusetts Amherst
Amherst, MA
craff@umass.edu

Evan Rourke
University of Massachusetts, Amherst
Amherst, MA
erourke@umass.edu

Chandler Tayek
University of Massachusetts Amherst
Amherst, MA
ctayek@umass.edu

Abstract

What doesn't kill you makes you stronger, but you can't learn if you actually die. Instincts can be thought of as priors which prevent death to allow for the possibility of experiential learning. On the other hand, traditional RL techniques focus on learning from experience, but typically require the "deaths" of many virtual agents in order to converge to a good policy. Our project attempts to combine these two ideas in a 2D self-driving car environment by combining a genetic algorithm model and a traditional RL model. The goal is to train a car that drives reasonably in as little time as possible. We present two variations of our model and test against two different baseline models. All four models were tested on the same 1,000 automatically generated tracks. Out of all of the models, the INSTINCT model performed the best obtaining an average two-lap progress percentage of 54.39%. Watch our agent: youtu.be/jTTgcN2eEs4

Introduction

One of the biggest issues with traditional reinforcement learning techniques is that when they are deployed in real environments, it's possible for the agent to explore states that cause damage to itself, others, or the environment. This means that developers have to design complicated systems that try to explicitly catch and avoid these situations. Our proposed model seeks to create an agent that safely interacts with the environment, allowing it to explore without destroying itself. The primary advantage to our approach is that the agent will be able to explore a new environment as much as possible without dying (in reinforcement learning terms, extending the episode for as long as possible).

We predicted that the INSTINCT approach will make training time shorter than traditional RL methods. After a population of agents has run a number of trial episodes, the next population is created by sampling from the previous one. Successful agents are proportionally more likely to contribute to the next generation. We expected that our approach should give the agent the ability to avoid situations that will result in an extremely negative reward while still learning the specifics of an environment.

An example of a real application for this algorithm is in the area of exploratory robots, specifically in the case of a fully autonomous Mars rover. This agent could explore using traditional RL techniques, but if it fell down a cliff and broke, it wouldn't be able to continue exploring at all. This means that the agent must already have the appropriate priors so it will never explore situations that could result in the destruction of the robot, while still allowing it to become familiar with its environment.

Related Work

The closest model that resembles our proposed INSTINCT model would be the NEAT algorithm. The NEAT algorithm looks at a neural network architecture design as something which could be solved with a genetic algorithm, with the genes representing weights and neurons.[5-7] This method proved to be quite effective but differs from our method in that it relies on the concept of "complexifying". NEAT learns its architecture by adding weights and neurons. We differ in that the number of weights in our model remains fixed, and instead, we separate the learning process between the genetic algorithm and the experience component. In other words, the genetic algorithm will not be used to improve the model architecture, but rather to augment its results.

Experimental Environment

The environment that the agent learned to navigate is a 2D race track. The quicker the agent makes it through the track, the better. If the agent drives into the edge of the track, the episode is over and the agent is not allowed to make any more progress. The episode also ends if the agent doesn't make a certain amount of progress along the track within a given time threshold. If the agent navigates the course while following the previous two constraints successfully, the episode will end after one lap. These constraints were chosen to encourage the agent to drive quickly on any of the arbitrary tracks. This allowed us to quickly train many agents on several different tracks to prevent overfitting. A successful agent will never touch the edge of the course.

As an agent drives through the track, it is aware of how much lap progress it has made since it took its most recent action. This lap progress is used as a reward for the agent so that it learns to maximize reward and move forward on the track. A gamma factor is used to decay the reward that the agent receives as time goes on to incentivize the agent to navigate the track quickly. The agent must reach checkpoints in the correct order to complete laps. If the agent were to drive the track backward, they would be making negative track progress and thus be receiving a negative reward, so they won't be able to exploit checkpoints for laps or reward.

If an agent hits the edge of the course, the episode will end and the agent will receive a negative reward. This negative reward will be interpreted in the conventional reinforcement learning way for the baseline agent. For our agent, the negative reward signal from crashing will contribute to the "fitness" of the INSTINCT component of the model. If the agent fails to reach the next checkpoint within a certain time, the episode will also end and the agent will be given a negative reward. This is to discourage agents from coming to a stop or turning around on the track. That negative reward will be treated the same way as if the agent drove into the edge of the track. The weight of the negative reward given at death is a hyperparameter.

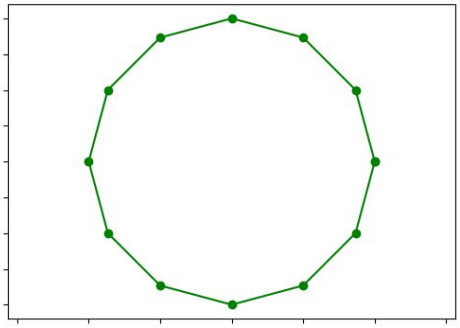
The agent observes the track through five distance sensors spanning the front of the car from left to right. The agent will also observe its own speed. The agent interacts with the environment by pressing the brake or gas and picking a steering angle. Formally, the agent's action set is defined as $\{Brake, Gas\} \times \{Left, Center, Right\}$

Since the state space isn't discrete, function approximation with a linear and Fourier basis was used.[3] The number of dimensions is too large to simply discretize the states, so function approximation must be used.

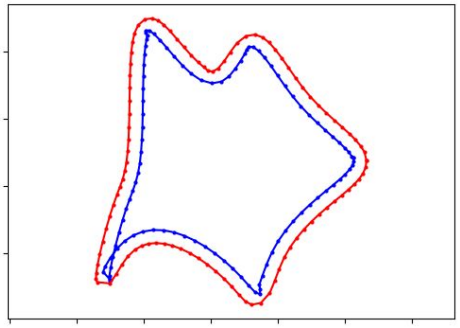
Track Generation Algorithm

In order for the agent to generalize to many different tracks, we needed a large selection of diverse tracks for the agent to train and test on. This required random track generation in order to remain tractable. Below is a list of the steps taken to produce our training and testing data:

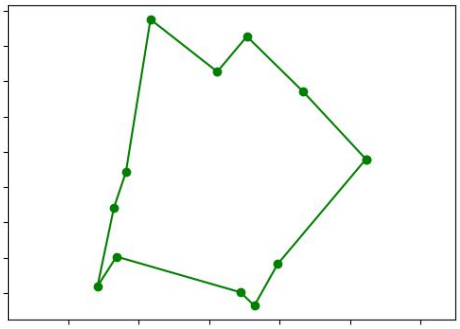
1. Make a circle of control points
2. Push the points in and out away from the midpoint of the circle
3. Uniformly sample a circle, and add the resulting delta vector to each control point
4. Fit a b-spline through all the control points
5. If there are any self-intersections in the b-spline curve, reject that track and try again
6. Form the inner and outer loops of the track by creating curves that are left and right perpendicular to the existing b-spline curve
7. If there are self-intersections on the edge tracks, take all the points that belong to the loop formed by the intersection and set them all to be the intersection point
8. Create equally-spaced checkpoints along the track
9. Create a start point on the track
10. If the car starts on a turn, move the start point forward until it starts on a straight or a slight curve
11. Accept the track
12. Create another track from the completed track by flipping the entire track from left to right (so now the car must drive clockwise instead of counterclockwise)



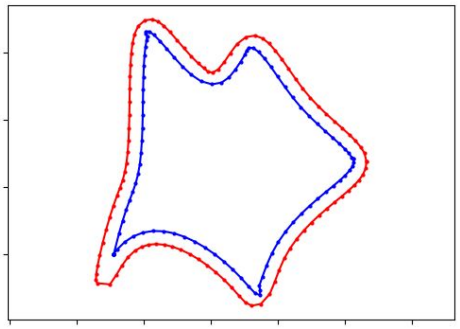
Step 1 - A circle is used to start



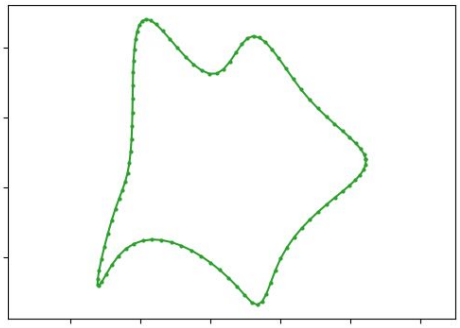
Step 6 - Walls are added to the tracks



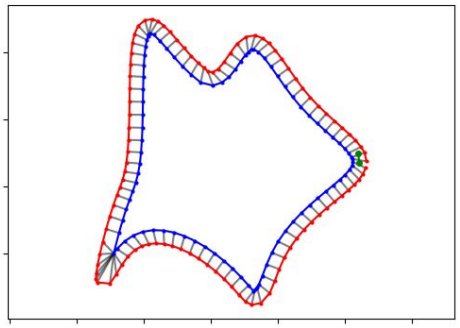
Step 2 and 3 - Alter the points by pushing them away from the center & add random deltas



Step 7 - Walls that intersect are trimmed



Steps 4 and 5 - Make a continuous curve through the control points



Steps 8 through 12 - Checkpoints are added to track progress

Models

We experimented with four different models, where we expected each one to provide different potential advantages. Two of them used conventional reinforcement learning techniques with basis functions using q-learning. The other two were our attempts to improve upon these existing models by adding a genetic algorithm component to the learning process. We hoped that our additions would allow the models to train faster or with fewer attempts by encoding instincts that allow the agent to avoid death. One of the proposed architectures is to have an instinct component that is in parallel with an experience component. The idea of having the two different components in parallel is to allow the model to get input from two different sources, one from its experiences and the other from previous "generations" of the model. A parallel architecture for the model is preferred over having the two components in series, with the idea that it allows for the hereditary knowledge to be passed directly to the output. Using this parallel architecture, we implemented two different variations of the model, which are shown below.

Baseline model: Q-learning with Basis Function

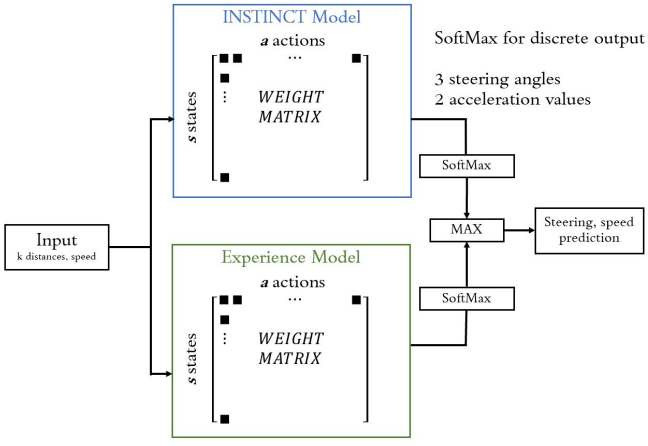
Function approximation uses a basis function – such as the polynomial or Fourier basis – to get additional features from the state space. As mentioned above, we do this because we have a continuous state space that we need to discretize. The weight matrix will be used to approximate how good each action will be.

$$v_w(s) = w^T \Phi(s)$$

Model 1: Parallel weights with function approximation (INSTINCT)

In our model, we have two weight matrices to evaluate the quality of actions. The experience component $w_{experience}$ is updated through the agent's interactions with the environment during the episode, and the INSTINCT component $w_{instinct}$ updates through evolution (a genetic algorithm). The w matrices for RL agents are initialized to a matrix of zeros, so the experience component $w_{experience}$ is initialized this way for each new population. The $w_{instinct}$ is initialized with random values. The training procedure is explained in more detail in the training methodology section.

$$v_{w_{instinct}, w_{experience}}(s) = max((w_{experience}{}^T)\Phi(s) + w_{instinct}{}^T \bullet s)$$



Combining the outputs

We originally intended to use addition or multiplication to combine the outputs of the experience component and INSTINCT component. We chose not to use addition because two semi-confident action probabilities could outweigh a single very confident action probability from the INSTINCT component. Multiplication is also

undesirable because one component's low confidence in an action could cause the agent to not take the action that the instinct network was very confident was good. Performing an element-wise max over the two action probability arrays avoids these problems by allowing the most confident probability to be the only one considered.

The INSTINCT component is an $S \times A$ matrix where S is the number of state variables plus one for bias, and A is the number of actions. Recall that the state is defined as a vector of the five distance sensors and the agent's current speed. This matrix is multiplied by the state vector to get a value for each action, and the resulting action score vector is softmaxed. This action distribution is element-wise maxed with the experience component's action distribution. We then take the argmax of this vector to determine the next action for the agent to take.

Model 2: Evolved Experience Initialization Parameters

We also built a model that uses a genetic algorithm to learn the initial weights for the Fourier basis. This does not mean that an experienced agent passes what it learned in the experience component to the next generation. Rather, a separate "experience initialization" matrix is kept for when the successful agents are used to produce the next population. The idea was that this model could perform better than baseline because it was seeded with some of the knowledge from previous generations rather than starting from scratch.

The evolved initialization model didn't perform well because of the large number of parameters of the initialization weight matrix. It didn't have enough time to learn, which caused it to underfit the training data. In order for this model to work it would have needed to run more generations with a larger population size. This model is obviously inferior to the basic INSTINCT model because the INSTINCT model doesn't need as much computational power, nor the time. As you can see in the table above it did only slightly better than the baseline Fourier model and slightly worse than the linear model.

Intractable Models We Explored

Not every model that we proposed proved feasible. The proposed neural network model ended up not showing promising results while requiring much more computational power that we didn't have access to.

Neural Network

Training the neural network for the same amount of time doesn't result in large enough gains in return to be tractable. When learning the tracks and evolving, a low-degree Fourier basis controller proved to be sufficient for the INSTINCT and experience models. We had proposed one sub-model with a continuous output, and one with a discrete output. Both of these sub-models are intractable due to the extremely long training time they would require.

Methodology

Training methodology

Our agent was trained by evolving as it interacts with different environments. Every agent in the population drove a number of trial courses from the dataset of training courses. The reason every agent runs multiple trial courses is to get a more accurate fitness score. Here, fitness is the average RL return over the trial courses. The most successful agents were then selected to produce a new population. Each member of the new population has an INSTINCT component that is a result of being bred from the previous generation, but the experience component was re-initialized with every generation. All models were re-initialized by completely wiping out the experience, except in the case of the "evolved experience initialization" model.

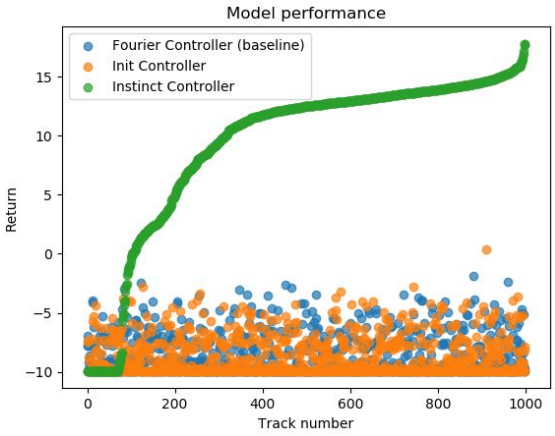
Return is defined as the time-decayed sum of rewards. Death is a large negative reward, which means that dying significantly reduces an agent's chance of passing its genes along. This is especially true in the case of the agent dying early because of time decay.

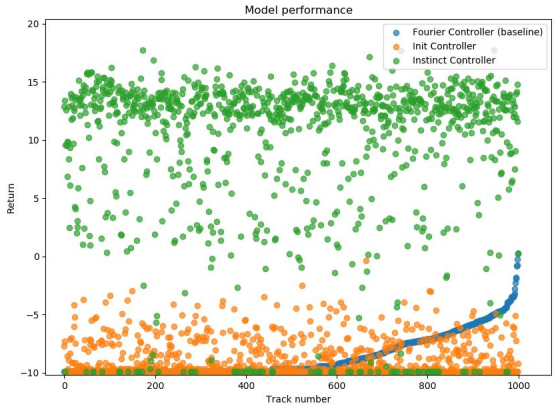
The two reinforcement learning agents were trained on a random subset of 2,000 out of the 10,000 training tracks. The INSTINCT weights and evolved initialization weights were trained over 10 generations with 8 tracks per generation for a total of 80 training tracks each.

Testing methodology

A test set of 1,000 test tracks were generated for each agent to properly evaluate if we created an agent that performs better in new environments. During test time, our agent was allowed to update its experience component, but did not evolve the INSTINCT component from one test course to the next. To make the comparison fair, the baseline agent was allowed to update as it drove the course. We initially expected our agent would be more likely to survive on the test course, and furthermore would be able to perform better than the baseline after a short amount of experience in the new environment.

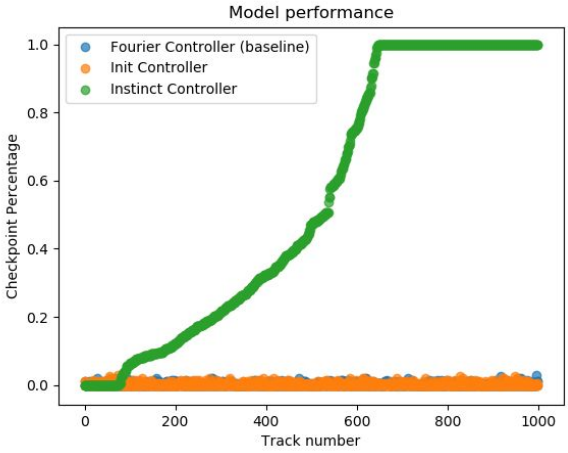
Results





To create the above graphs, we took each track from the 1,000 test tracks, ran each model on it and recorded each model's return at the end of the episode. Then we reset the experience weights for the INSTINCT agent, set the experience weights for the Initialization agent to it's learned initialization matrix, and allowed the Fourier agent to retain the weights it updated from the previous tracks.

After accumulating the returns for each agent on each track, we plot them. The tracks are sorted by the returns from one agent to rank them by the approximate difficulty to make the graph easier to read. The plot on the left is sorted by the INSTINCT controller returns, and the plot on the right is sorted by the Fourier controller returns. Interestingly, there is no apparent correlation in the ease of a track across any two of the controllers shown here.

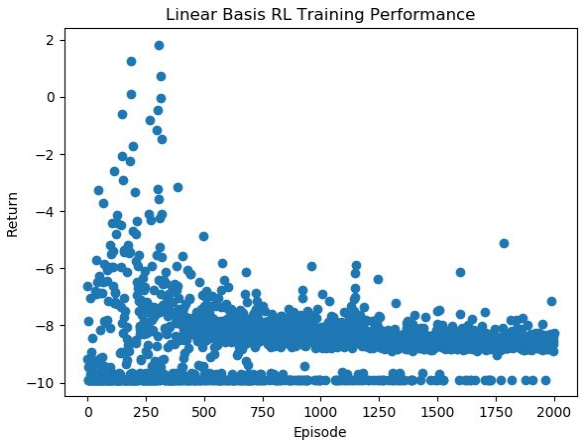


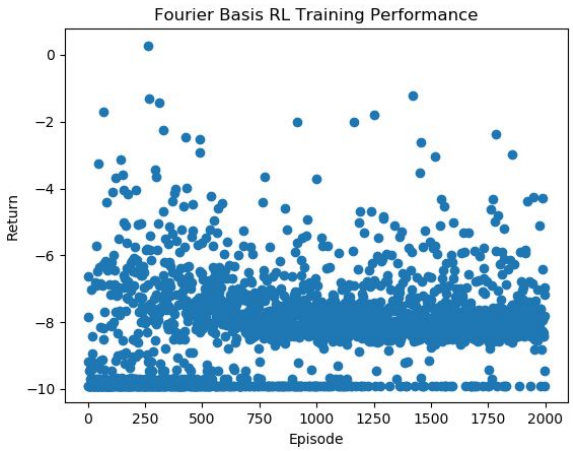
The episode terminates if the agent successfully completes two laps on a track. This is done to prevent the simulations from running indefinitely for easier tracks. Between two agents which both complete two laps, the one that did it faster gets a higher reward.

The plot on the left is generated in the same way, but instead plotting the returns, we plot how far the agents made it through the two laps. The function that we used to score the agents is:

$$f = [\textit{num checkpoints passed before death}] \,/\, (2 * [\textit{total number of checkpoints in the track}])$$

Simply put, it is the percentage of checkpoints that the agent was able to pass out of the total that it could possibly pass.





The above plots were generated by training traditional reinforcement agents with q-learning. We used a linear basis for the left plot and a Fourier basis with degree 3 for the right plot. At around 250 episodes, the agent seems to perform well. This is the point in training at which the agent has learned a bit about how to avoid walls, but the epsilon is still high and many random actions are being taken. The combination of emerging abilities and lucky random actions can sometimes yield a decent outcome, although it should be noted that for each relative success at that time, there are dozens of total failures. Ultimately, the agent converges to poor performance on average. When observing how the agent drives the course, the RL agents tend to learn to turn one way but not the other.

The table below shows our scores for each model. We used the same percentage-completion-metric as above.

Agent	Average track completion of 2 laps
INSTINCT	**54.39%**
Linear	0.43%
Init	0.26%
Fourier	0.23%

Conclusion

During this exploration, we learned many things. We found that creating an algorithm to automatically generate driveable tracks was quite difficult. After we generated the tracks, getting the reinforcement learning agents to perform reasonably was also very difficult. Because the RL method didn't have high confidence, the INSTINCT component took over for the vast majority of actions taken by the INSTINCT agent. The INSTINCT model actually did make training time shorter, as we predicted, but did so by killing more agents than baseline. A common failing point for most of the models is when the track turned extremely sharply (>120°). Only the INSTINCT model was ever able to survive these turns.

Future Work

Improvements that could be made in the future include: tuning hyperparameters, and trying more traditional RL approaches such as TD-lambda. Additionally, the model could be changed so that it takes into account both components at the same time instead of only using the most confident one.

Delegation of Tasks

Make environment - Chris
Track generator - Alex & Evan
Track validation - Alex & Evan
Implement baseline model - Chris
Implement model 1 - Alex & Chris & Evan
Implement model 2 - Chandler & Alex
Implement model 3 - Evan
Tune hyperparameters - Everyone will contribute CPU hours
See also: Github commit log

Code

All our code can be found here: https://github.com/chrisraff/INSTINCT

References

1. "Experiments on the Mechanization of Game-Learning Part I. Characterization of the Model and its parameters" (1963, Donald Michie)
2. "Learning from Delayed Rewards" (1989, Christopher Watkins)
3. "Basis Expansion Explanation" (~2019, Justin Domke)
4. "Artificial Neural Nets and Genetic Algorithms" [Book] (1993, R.F. Albrecht, C.R. Reeves, and N.C. Steele)
5. "Evolving Neural Networks through Augmenting Topologies" (2006, Kenneth O. Stanley and Risto Miikkulainen)
6. "Efficient Reinforcement Learning through Evolving Neural Network Topologies"
7. "Skip Connections and Multiple Matrices in Recurrent Neural Networks" (NA, Mihir Mongia)
8. "Neural Architecture Search with Reinforcement Learning" (2016, Zoph & Le)