Capstone Project

Machine Learning Engineer Nanodegree

Ben Thayer
June 4th, 2018

Definition

Project Overview

Reinforcement learning and control theory have been working for many years to solve the problem of controlling robots. The general problem is mainly concerned with an actor in an environment that has the ability to choose actions to change the state of the environment in order to receive a reward. This is a field that has been researched for many years and is still being improved on today. There is a framework called Gym made by OpenAI that aims to present these problems in a consistent manner. OpenAI has also started and stopped development of a platform known as Universe, that is meant to facilitate researchers to use reinforcement learning within videogames, although OpenAI has since stopped working on the platform, so its usefulness remains limited and there are no good alternatives. Classical control, classical reinforcement learning techniques and modern reinforcement learning techniques can be all used to solve the problems offered within Gym "environments."

Classical control use rigorous mathematics and is useful when the state of the thing being controlled is known and there is a well-defined goal state. Classical reinforcement learning is used when the state is known, but there is not a well-defined goal state and all that can be provided is a reward signal to determine which actions are the best. There are many problems were we also do not know what the underlying state is and we only get to see a transformation of the state, such as an image. In these problems, we don't know how the visible variables relate to the underlying state and the transition model that changes the state from one instant to the next. This is what gives rise to modern "model-free" reinforcement learning.

One of the advancements that has been made recently in model-free reinforcement learning is deep Q-learning, developed significantly by Deepmind. It is a technique that allows for use of visual input and other types of more complicated inputs in trying to choose actions. One well-researched use case of this is in Atari games. Q-learning is able to take in the screen as input and the score as a reward signal and outputs actions which then lead the model to score higher than any human. In this project, I attempted to use deep Q-Learning to make an agent that drifts around the ball in the videogame *Rocket League*. I developed my own interface to use the screen as input and a simulated keyboard as output to control the car within the videogame.

Problem Statement

Rocket League is a video game in which a player controls a rocket-powered car and tries to score a ball into a goal much like soccer. One of the things that can be done in the game is drifting. I would like to create a model that learns how to drift. The game can be played with the camera focused on the ball. Additionally, most arenas have concentric circles around the center of the field, where the ball starts, so for the purpose of this project, I think it is best to have the car drift in circles around the ball while it is in the center of the arena and use the circles on the ground to gauge the performance of the model's drifting.

Metrics

The metric that I used to evaluate the performance of each agent/model is how long it "drifted." When the game starts, the ball is in the center of the field. The center of the field features two concentric circles painted on the ground. I used these to specifically define my evaluation metric. I defined that the car was "drifting" when the wheels were within the concentric circles.

Reinforcement learning additionally requires a reward function. Sometimes this reward function is based on completion of a task and other times, it is based on failure of a task. In this situation, I gave the agent a reward of +1 for every timestep that it was still "drifting" and a final reward of 0 when it stopped drifting. The object of the agent then is to maximize total reward, which can be done by keeping the trial going for as long as possible, which is the goal.

Analysis

Data exploration/Visualization

My primary input for this project was the screen. The screen features the car, the ball as well as other background objects. Below is a full image of the screen.

Figure 1: Full screen

Instead of using the whole screen in my training, I decided that it would be best to crop the screen into the areas of interest, the ball and the car. Below, I show two of the crop styles that I tried. They are all crops of the same frame as Figure 1.

Figure 2: Crop style 0

Figure 3: Crop style 1

I decided to use crop style 1 because it shows more useful information such as the field lines. Though the images are larger, they did not make the training take an unreasonable amount of time longer.

Below is an example of how I defined the boundaries of when I would end a drift.

Figure 4
Left: Car just out of bounds due to top wheel
Middle: Car in bounds
Right: Car just out of bounds due to bottom wheel

There are several irregularities in the data.

- Different lighting
    - In Figure 4, we can see that the image on the left is darker than the image on the right.
- Dirt kicked up
    - In Figure 4, we can also see some of the dirt that has been kicked up while drifting
- Parts of the ball light up at regular intervals.
    - Figure 5 shows two subsequent frames, one with the ball lit, the other without.
- Boost pads
    - Figure 6 shows one of the four boost pads encountered within the concentric circles.

Figure 5: Comparison of subsequent frames when ball lights up

Figure 6: A boost pad

Algorithms/Techniques

For this project, I used reinforcement learning. Reinforcement learning is specifically used when input is used to control what is known as an actor. The actor's choices affect the environment. When using reinforcement learning, all that is specified is a reward signal and the reward signal is used to train a model to pick the action that maximizes the rewards. There are two types of actions that can be used in reinforcement learning, continuous and discrete. Although rocket league does accept continuous actions in the form of joystick movement, it also accepts discrete actions in the form of keyboard key presses. In reinforcement learning, continuous control is considered more difficult than discrete control and inputting actions in the from of key presses is trivial, making using a discrete action space the easy choice.

Reinforcement learning can be very complex and uses techniques beyond just choosing a good model. The way a reinforcement learning model is trained is just as important as the kind of model used. There are two main types of things that can be trained for. There are deterministic Q-learning models and there are stochastic policy gradient models. I chose to use Q-learning because I found it easier to implement and debug. It also allowed me to use the Deep Q Network (DQN) paper from Deepmind as a reference as I worked on this project.

There are several major decisions that I copied from the the DQN paper. The first is using Temporal Difference (TD) learning over Monte Carlo (MC) learning. Both techniques predict the expected future reward. MC learning uses the actual rewards of a trial while TD uses the reward of one action and the predicted reward for the next action. There are many different types of TD learning, and I chose the simplest type that only includes the predictions for the next step. There are also different ways to store and use data for training. The DQN paper uses a replay buffer that stores a set amount of trials and uses them for training. Lastly, another issue that the DQN paper addresses is that using the same network to generate the values that it trains itself on can be unstable and produce unusable results. The solution to this is by holding the training targets constant. To do this, before I a training epoch, I make all of the predictions and do not change them until the next epoch.

Benchmark

I used two different benchmark actors to compare against. The first is a random actor which I would use to tell if my model has indeed learned something. The second benchmark actor is a human actor, myself. If the model I train is able to do better than me, I will know that it is very useful. Using both benchmarks, I will be able to determine the fitness of each of my final model.

Methodology

Data Preprocessing

I do three things to preprocess the data. First I crop the data to allow the network to focus on the most important areas of the screen. Next, I convert to grayscale because all relevant features can be determined without use of color (See Figure 7 below). Finally, I normalize the data so that the mean is 0 and the standard deviation is 1 for each pixel. Additionally it is worth noting the problem of frame skipping. Although my agent is receiving input at 8 frames per second, the game is actually rendering 144 frames per second, so any frames skipped only leave the current frame off by a negligible amount compared to 8 frames per second. I also have a system in place to ensure that the processing of each frame takes less than 1/8 of a second, giving the algorithm enough time to process everything during each frame.

Figure 7: Color and Grayscale versions of the same image.

Implementation

The main components of my implementation are the classes AgentRunner, Agent, Environment, Preprocessor and Trainer. The AgentRunner is what loads and runs everything else. It takes as input all of the parameters, including which model to use, reward discount, whether or not to use grayscale, what crop dimensions to use and many others. The AgentRunner either loads a model from disk if that model has already been initialized or generates a new model if there is no saved version and constructs an Agent to act using that model. The AgentRunner also constructs an Environment, which acts as the interface between the Agent and the rocket league game. The main function used in the Environment is the `step(action)` function which takes in an action and outputs the images for the model to use. When constructed, the Environment creates a Preprocessor that does everything that is needed to preprocess the data based on the crop style and other factors. After a specified number of frames, the AgentRunner constructs a Trainer and runs the trainer in the background during data collection. The Trainer maintains the data and targets and provides a new model for the Actor to act on when done training.

During the implementation process, I ran into several challenges. Most notably was communicating with the game. There is no library that is already built to be able to interact directly with games, so I effectively had to make my own. I used Python Image Library (PIL) to capture images from the screen, however I found that it was not fast enough to capture images of both the car and the ball. After some experimentation, I found that it was faster to capture one large frame than two smaller ones, so I just captured a screenshot of the entire frame then delegated the cropping to the preprocessor. Additionally, I found that since Rocket League is a videogame, it does not accept keyboard input in the same way that a typical program would, so the game ignores key presses from certain python modules. I found that this problem can simply be avoided by using PyWin32 to press virtual keys, which the game then reads as normal keyboard input, allowing us to control the game from within python. I also found that some things that I tried simply did not end up working. Of these, the thing that took the most time was using a classifier to determine the end of the trial. Although I was able to attain 95% accuracy, I still found it to be too noisy to be used effectively to train the DQN.

Refinement

To start, I collected training data based on my own actions and used it to create an agent based on the expected rewards of my own actions. I found that it generally did not make consistent choices for its actions and decided to use it as a pretraining for reinforcement learning models. Initially, I was running at 15 frames per second, but I determined that that was too fast for the actor to be making decisions and lead to the actor making random movements and being unable to correctly predict the rewards for each given action, so I started training each of the Q-networks from scratch at 8 frames per second and found that the performance was more consistent. I spent most of my refinement time working to bring my implementation closer to the DQN algorithm. On my first iteration, I started out by training “online” by just

training on each trial as they happened. One of the most important additions in DQN is a replay buffer, so I added that while still computing training targets on the fly. Holding targets constant was the next step, so I implemented a system to do that. Right away after implementing frozen targets, I found a great improvement in the consistency of behavior of the model and became able to distinguish the agent's actions from random actions. Finally, I decoupled the model that was being acted on to be separate from the model being trained, so that each batch of training data was more consistent, although I did not notice much improvement of this addition. At each stage of my implementation of DQN, I tried using different model architectures, but found the model architecture used to be secondary in importance to the DQN algorithm. Additionally, I found that increasing the discount factor generally lead to better performance.

Results

Evaluation/Validation

In the refinement section, I talk about the specifics of my implementation. In this section, I focusing on the most important hyperparameter in reinforcement learning. This hyperparameter is known as the discount factor, commonly represented as the greek letter gamma, and it represents the weight put on future rewards. A low discount factor means that an action is given a higher value for obtaining more immediate rewards while a high discount factor means that an action is given a higher value for rewards received in the future.

To compare each of the different discount factors to the benchmarks, I ran a sequence of 50 trials for three different values of gamma, 0.9, 0.95 and 0.975, all using the same underlying model architecture with different weight initializations. The data from these three configurations and from the two benchmarks is visible in the histograms and the table on the next page. The data from the human trial is cut off in the histogram so that the scale of the x-axis remains the same as the other trials.

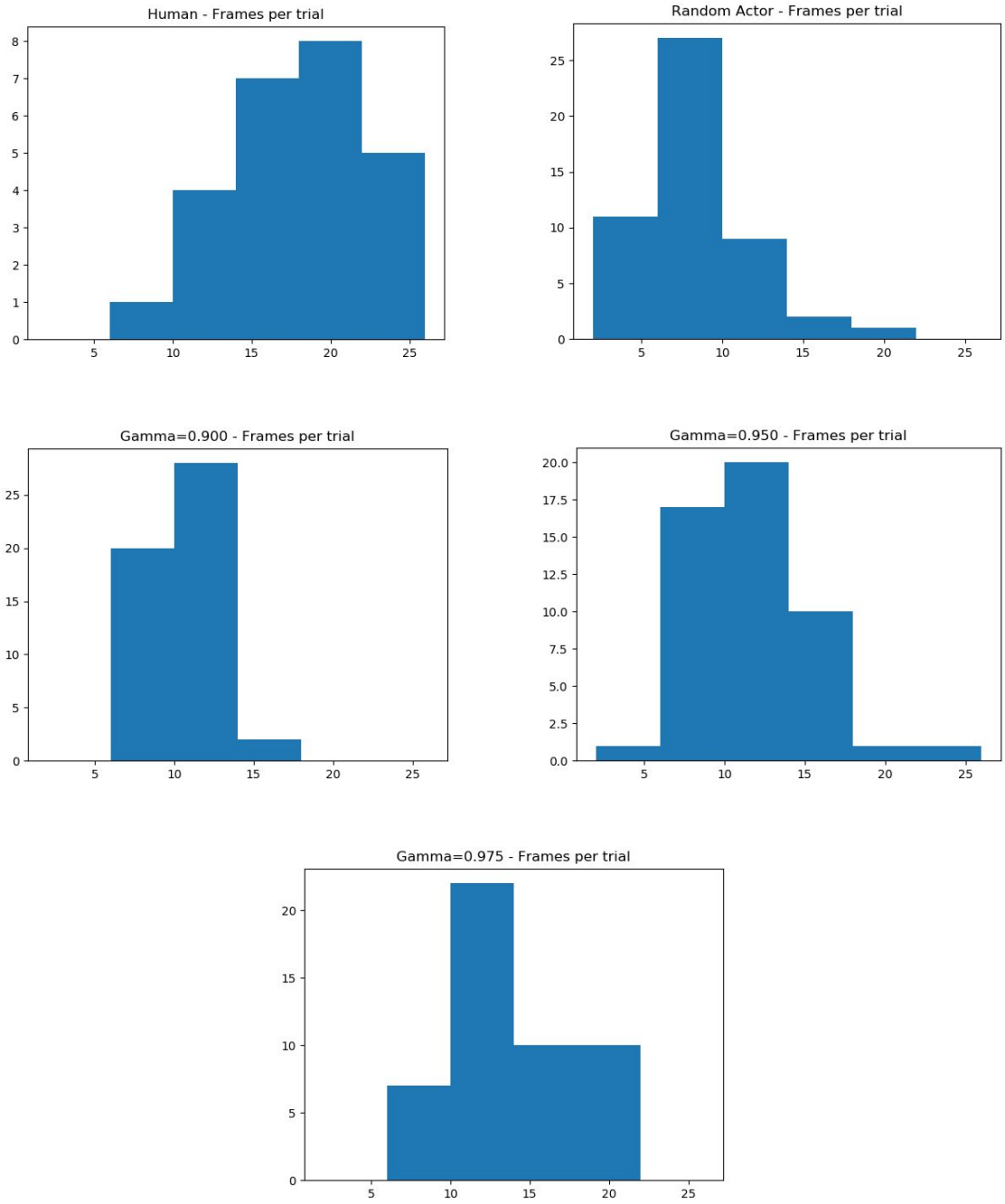

Figure 8: Histograms for trial lengths in number of frames

Actor	Max	Mean	Median	StdDev
Human	186	44.24	26.5	43.95

Random	19	8.04	7	3.21
Gamma=0.900	15	9.68	10	2.26
Gamma=0.950	26	11.48	11.5	3.86
Gamma=0.975	21	13.34	13	4.31

Figure 9: Statistics of different runs

The first thing that is obvious is that none of the algorithms come close to the human benchmark, however they do perform significantly better than random making this project a partial success. The most significant result is that a higher discount factor performs better. When watching the model run, I noticed that all of the actors were able to steer away from the boundaries much of the time, however a higher discount factor means that the actor is more likely to plan for what happens after it steers away, so it veers out of control less often. There are additionally some abnormalities in the results. Although, there is a clear ordering when looking at the means and medians, the maxes did not follow the same order. The random actor is able to achieve a relatively good maximum trial length. This is likely because when in the right position, moving straight is the ideal action and on average, the random actor is moving straight. Additionally, the agent with gamma=0.95 has a maximum higher than the agent with gamma=0.975. This could mean that a larger discount factor would lead to the agent being less worried about crashing in the short term.

Conclusion

Free-Form Visualization

One of the biggest limitations is that of human error. In this project, I was the one doing all of the labeling. This leads to a variance in what the model should be approximating, making the solutions less reliable.

Figure 10: Difference in two of the end frames

In the screenshots above are of the last frame in two different trials. Although they look fairly similar, there are some subtle differences resulting from human error that would make it challenging for the Q-net to approximate the correct value function. The easiest difference to tell is that the bottom boundary line is in a different place. In the left screenshot, much of it is visible, while on the right, it is barely visible. Similarly, when looking closely at the wheels closest to the top of the frames, it can be seen that the one on the left is further down then the one on the right. This can also be seen in the antenna. This is only a subtle difference, but this is the kind of difference that will take place over the course of a frame or two. This inconsistency is noise to the network and can cause it to approximate an incorrect function.

Reflection

I based my solution off of the DQN architecture from Deepmind. I found that it was able to perform better than random, though not nearly as good as a human. Deepmind's paper touts that the DQN architecture is able to perform significantly better than humans in nearly all cases. There are several issues that I think lead to an underperformance of the DQN on this task. The largest issue was inconsistency of when the trial ended due to human error. The games that the DQN trains off of in the paper were all digitally supervised and no human intervention was used to start and end the trials. This means that the network had a much more consistent set of training data unlike this project which relied on me to end the trials, inevitably producing noise. Additionally, human supervision lead to the issue of slow training. Although I was able to write the code so that the training was happening while the data was being gathered, the data gathering was limited by my own capacity to start and end trials, significantly slowing down the process and reducing the number of models I was able to test and how long those models were able to train.

Improvement

To further improve the model, the best thing to do would be to automate the process of starting, stopping and resetting the trials. This in itself would be another challenging problem, but automated training would allow for more accurate training and the testing of more models and sets of hyperparameters such as buffer length, training interval, and discount factor. It would be worthwhile to test many different model architectures, however this is unfeasible without automated training. Each of the models takes several hours to train because of the need to manually label each trial. After automating trials, I would try using LSTMs, batch normalization and other types of layers in the model to see if any of them could lead to better performance.

References:

"OpenAI Gym" https://gym.openai.com/

"OpenAI Universe" https://github.com/openai/universe

"Human-level control through deep reinforcement learning" Mnih, Kavukcuoglu, Silver (Deepmind DQN) - https://deepmind.com/research/dqn/