David Baugher
October 13th, 2016

Capstone Project

Machine Learning Engineer Nanodegree

Definition

Project Overview

For my capstone project I decided to utilize OpenAI's Gym suite to compare several approaches to reinforcement learning. Reinforcement learning is usually defined as an area of research in artificial intelligence separate from both supervised learning and unsupervised learning. In reinforcement learning the goal is usually to program an agent that learns to do some desired task by experimenting, getting feedback, and improving its performance based on that feedback. OpenAI's Gym is a platform showcasing many environments for reinforcement learning, from classical environments like Cartpole to more modern but now widely used environments like the Atari games. Even though these environments are all "toy problems" the application areas of reinforcement learning include decision optimization such as stock selection, mechanical control such as robotic movement, natural language processing, and any other area where a learning agent can be given feedback to improve its performance. Toy problems are useful for testing new approaches because they serve as accessible benchmarks for the community.

OpenAI's Cartpole page: https://gym.openai.com/envs/CartPole-v0

Problem Statement

The goal of any given environment will be specific to that environment but OpenAI provides some useful definitions that can be used as benchmarks. The environment I've chosen to work with is a classical environment called Cartpole. In this environment the agent needs to learn how to balance a hinged pole upright in moving cart, not too unlike balancing a pen upright in one's hand. The state-space of this environment is continuous and infinite; the input given to the agent at any step contains the current velocity and acceleration of the cart and the pole. The possible actions are applying a force from the left or from the right, which is similar to pushing the cart to the right or to the left. We will be showcasing a variety of agents that learn with varying degrees of success when to

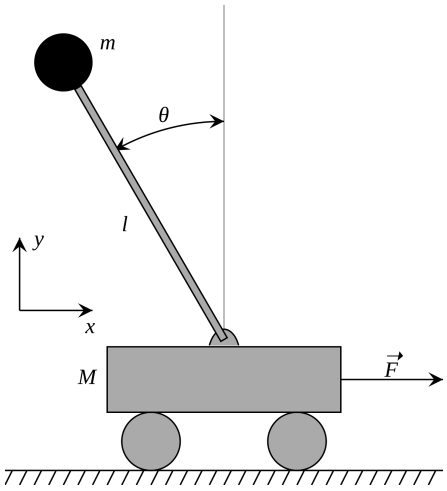
apply a force to the left or to the right of our cart so as to the pole vertical for as long as possible in any given episode.

Metrics

OpenAI defines “solving” this environment as having achieved an average score of 195.0 or more over 100 consecutive trials; we will refer to this average score for the most recent 100 episodes as our “running average” henceforth. The other metric important to our discussion is the number of episodes it takes to solve the environment if it can be solved by the agent; this metric corresponds to our speed of learning. While these are the crucial metrics for comparing our agents ability to solve the environment, we will be discussing other aspects of the agent implementation that are german from a computational perspective such as the amount of data stored and the amount of computation needed by that agent to perform well in the environment.

Analysis

As mentioned on OpenAI’s Cartpole page, the inspiration for this environment comes from the 1983 paper *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems* by Barto, Sutton, and Anderson. It is a type of inverted pendulum problem, and the diagram below was provided by the Inverted Pendulum page on Wikipedia: https://en.wikipedia.org/wiki/Inverted_pendulum



The four inputs provided to our agent by the environment at any given time are as follows:

- ***x*** : position of the cart on the track
- : angle of the pole with the vertical
- ***x*** : cart velocity
- : rate of change of the angle

The following are examples of data provided from the environment to our agent:

[0.0313006 , -0.21150274, 0.02043664, 0.28235688]

[0.02707055, -0.01667816, 0.02608378, -0.00381101]

[0.02673699, -0.21216429, 0.02600756, 0.2969862]

[0.0224937 , -0.01742256, 0.03194728, 0.01261774]

[0.02214525, 0.17722701, 0.03219964, -0.26981671]

Here is a link to the original paper by Barto, Sutton, & Anderson:
https://webdocs.cs.ualberta.ca/~sutton/papers/barto-sutton-anderson-83.pdf

Agents

General Characteristics

There are a variety of desirable traits for reinforcement agents; some of these traits include learning quickly, learning with little input/data, learning without needing a large amount of computation, learning consistently, and having the ability to learn in a variety of environments. Naturally, it would be nice if our agent possessed all of these traits, but as we will see there are trade offs between them, and selecting which ones you are willing to sacrifice is a decision best made with a mind toward the needs of the environment.

A State-Space Discretizing Agent: *nn_archetypes.py*

This agent with help elucidate some of the challenges of continuous and infinite state spaces. A simple reinforcement agent would just explore a given state-space enough to know what works in which states, but if the state-space is infinite this would require an infinite amount of both memory and exploring. There are several ways can we possibly address this, but we will start by building an agent with the goal of discretizing the state-space so we don't have to remember a vast, possibly infinite, amount of data to make a good decision. Discretizing means that we consider any states in a certain range to be equivalent, so we use a measure of distance to compare, in our case the Euclidean distance, and a hyperparameter to dictate how close states can be before they are "the same"; in the program I refer to these ranges as archetypes.

For this agent we only store the best recorded action for each archetype, state-space range, which we define as the action performed in the past that lead to the most rewards following that action. This implementation not only saves computer memory by discretizing but also on compute time through this simple lookup method.

The important hyperparameters are:

- **epsilon**
- **epsilon_decay**
- **time_before_exploit**

- **archetype_distance**

The **time_before_exploit** parameter simply specifies how much memory which should accumulate before using our knowledge; it relies on the iteration parameter to keep track of time. The **archetype_distance** parameter dictates the size of the range for determining our archetypes. As the last two parameters suggest, we are using a decaying e-greedy implementation for our exploration.

The results of this agent are not terribly promising for our goal of a score averaging 195.0 or more for the last 100 episodes. It is easy to see that our agent is learning, but the highest score we can manage for our running average is usually between 100 and 120, and is accomplished sometime after episode 400. Our agent does manage to achieve scores of 200.0 in infrequent episodes, but this not often enough to balance the more frequent lower scores.

There are two main problems plaguing this implementation. As you can see from the session output, it continues to accrue archetypes regardless of how long it is run, which means that is continues to find itself in states where its experience is not very helpful. The other issue is that the values given to state-action pairs are only probabilistically likely to correspond to a correct action in a given state; an action not terminating an episode in this environment does not guarantee it as the optimal action, only likelier to be so. We will revisit both of these issues with the remaining agents.

Various parameter tweaks, and results are shown below:

Tweaking **archetype_distance** **epsilon_decay** = 0.995	Distance = 0.11	Distance = 0.17	Distance = 0.23
Average Score at 500 Episodes	98.3	102.8	79.8

Tweaking **epsilon_decay** (**e**) **archetype_distance** = 0.17	**e** = 0.75	**e** = 0.80	**e** = 0.85	**e** = 0.90	**e** = 0.95
Average Score at 500 Episodes	109.2	115.6	126.2	131.8	116.0

A Nearest Neighbors Policy Update Agent: *nn_selective_mem.py*

Our second agent uses one of the topics discussed in the Nanodegree, nearest neighbors. Here our agent also performs random actions with decreasing frequency using a decaying e-greedy implementation and stores the state-action pairs from episodes where it did well. These memories are then consulted on moves when it is exploiting instead of exploring by pooling the nearest neighbor states and picking the majority vote of their actions.

The hyperparameters used were:

- **epsilon**
- **epsilon_decay**
- **number_of_neighbors**
- **highest_episode_rewards**
- **did_we_do_well_threshold**
- **iteration**
- **time_before_exploit**
- **max_memory**

In the beginning our agent uses only random actions to initialize its state action memory; the **time_before_exploit** hyperparameter dictates how long to wait and iteration keeps track of how long we've been waiting. The **number_of_neighbors** parameter is the number of closest neighbors that get to vote on our action choice if we are exploiting. The **highest_episode_rewards** parameter keeps track of the highest score we've obtained in any given episode; it gives our agent context for judging if it did well in an episode, and the **did_we_do_well_threshold** dictates what memories to selectively keep based on the percentage of the highest score achieved during the episode. Capitalizing as much as possible on our new standards for what the agent considers having done well, it will also prune its memory every so often of past memories that no longer make

the cut of episodes where it considers itself to have done well. The **max_memory** parameter dictates how many state-action pairs we are allowed to store in memory; larger memories will take up more resources, but be more effective in general. The e-greedy implementation uses the parameters epsilon and **epsilon_decay**; after we begin exploiting our exploration rate is given by epsilon and decayed by **epislon_decay** after every episode.

This agent attempts to address the previous problems of an infinite state-space and correct value assignment to actions by grouping the nearest neighbors from memory and having them vote on the action to take. This helps with the infinite state-space because having several similar states to draw on makes your experience more analogous; it helps with the correct action because we're now voting on several actions all of which came from high-scoring episodes, so we are more likely to make an optimal choice. While these choices help, they are still not function approximation; a powerful machine learning technique that predicts the right move based on learned features that we will see later.

The agent performs quite well compared to our previous agent, but the sheer amount of computation required to process the large memory in order to get the best action could be prohibitive depending on the application. Unfortunately, even the large increase in sophistication isn't enough to accomplish our goal of a running average greater than 195.0, so this agent does not technically solve our environment. The running average usually levels off at around 185.0 sometime before episode 400, and continuing to run thereafter causes it to oscillate around this value. In dozens of runs I have witnessed it achieve a running average of 192.0, but never greater than 195.

Various parameter tweaks and results are shown below:

Tweaking **epsilon_decay** (**e_d**) **epsilon** = 0.87 **number_of_neighbors** = 7 **did_we_do_well_threshold** = 0.7	**e_d** = 0.78	**e_d** = 0.88	**e_d** = 0.98	**e_d** = 0.999
Average Score at 300 Episodes	146.2	159.8	167.0	36.7

Tweaking **number_of_neighbors** (**nn**) **epsilon_decay** = 0.98 **epsilon** = 0.87 **did_we_do_well_threshold** = 0.7	**nn** = 7	**nn** = 17	**nn** = 47

Average Score at 300 Episodes	167.0	112.6	35.7

Tweaking **did_we_do_well_threshold** (**w_t**) **epsilon_decay** = 0.98 **epsilon** = 0.87 **number_of_neighbors** = 7	**w_t** = 0.6	**w_t** = 0.7	**w_t** = 0.8	**w_t** = 0.9
Average Score at 300 Episodes	154.3	167.0	172.6	159.7

A Parameter Search Agent: *parameter_search.py*

A parameter search agent is an agent that utilizes some function of the agent's inputs that includes tweakable parameters; it is one kind of function approximating agent. For example, if we had an environment with 3 inputs(say **a**, **b**, and **c**) and the function was

Output = a + b + c

we would not be able to use parameter search because our agent would have no parameters to learn. If, on the other hand our function took the form

Output = a*x + **b***y + **c***z

where x, y, and z are parameters that can be tweaked, then we can use an agent that, by some method, tweaks them until they yield desirable results. This is in fact the exact function form we use for our parameter search agent; each input is multiplied by its own variable. In the case of our implementation, the parameter search is done randomly until an episode is found in which the agent scores above a desired threshold. The parameters leading to this score are then stored for later use. One drawback to this kind of agent is the need to guess the appropriate function form. In our case the function is a linear function, but this does not necessarily coincide with the function form needed to solve the environment, not to mention the form that allows for the most efficient learning. We mitigate this problem by storing parameters that have worked in the past and continuing

to explore for new useful parameters. In this way even if the best function form includes variables with differing exponents we can approximate a solution by using good parameters found in the past for the specific state we find ourselves in currently. This is similar to approximating a curve with straight lines.

As outlined above this is a naive agent in the sense that it does not use what it has learned in the past to further its future learning, which makes it more like a fact gathering agent than an inference making agent. Once it has gathered sufficient facts to do well in the environment, the environment is solved. In an attempt to optimize the time in which the environment is solved we've implemented an e-greedy strategy for exploration with a constant epsilon that starts at 0.6 and drops to 0 only if we have possibly solved the environment. We only keeping track of the scores from episodes when we're exploiting to know if the environment has possibly been solved. If we find that the last 50 exploiting episodes had an average score above 195.0 we enter countdown mode to test if we have truly solved the environment.

The hyperparameters for this agent are:

- **epsilon** : exploration rate
- **have_parameters** : a bool to signal if we have found any parameters yet
- **pruning_interval** : an interval dictating how often to prune low scoring parameters
- **pruning_threshold** : the average score below which we will prune parameters

As one can see, this agent not only gathers new promising parameters, it also forgets the ones that turn out to be less promising than initially suspected, Per the last two parameters above, **pruning_interval** and **pruning_threshold**, the program will check the average score of stored parameters every so often and get rid of lower scoring ones.

This agent solves cartpole very quickly, usually close to episode 200. Aside from actually being able to solve the environment the great aspects of this agent is that it does not require storage of a large amount of data or need to perform arduous computation to get its optimal action. The only downside to this kind of agent is that because it requires the programmer to specify some kind of initial function form it may not be a good fit for more complex environments.

Tweaking **epsilon**	**e** = 0.5	**e** = 0.6	**e** = 0.7

Average Episodes to Solve	236.6	220.8	231.4

A Policy Gradient Neural Network Agent: *tf_poly_grad.py*

Our final agent utilizes a neural network to approximate a function that solves the cartpole environment given the inputs. This is achieved through a policy gradient implementation, which is a very popular form for solving Atari environments currently. Policy gradients work well in binary environments like ours where the actions are either left or right because when the agent performs well in an episode the implementation updates the weights of the network by assuming every action taken in every state was correct and, often, vice versa; so, not too unlike the supervised learning we covered during this nanodegree, the agent is being feed live training data and then using the updated weights when it is exploiting instead of exploring.

As mentioned previously, our neural net is built using Tensorflow. We use a net with 4 inputs to match our cartpole environment, two hidden layers of 40 neurons each, and an output of size two. The output utilizes a one-hot encoding were [1, 0] represents action 0 and [0, 1] represents action 1. All of our weights are initialized with random data from the normal distribution with standard deviation of 0.1, and all of our biases are initialized to 0.1. Many of the constructs used were informed by doing tutorials on the Tensorflow website, and the reasons for their implementation are beyond the scope of this project, but all neural nets use some kind of loss function, one of a variety of nonlinearities for the neurons, and some form of gradient descent; our are respectively, cross-entropy loss, the rectified linear unit function, and the Adam Optimizer algorithm.

Our reinforcement learning segment utilizes one of two different decay strategies for our e-greedy implementation; one of them converges consistently every time it has been used but it can take a long time to converge(averaging ~6000 episodes), and the other strategy converges in about one-tenth of the average time, very quickly, but fails to converge about one-eighth of the time when run. The code as submitted is utilizing the faster strategy, because if it does not converge after 1000 episodes the first time it can be restarted and will likely do so the second time; the average convergence time is ~700 episodes in 7 out of 8 episodes.

Why such a vast difference in times for our e-greedy strategies? To understand this better we now have to explain what our implementation does in more detail. To start out our agent selects actions completely randomly until it has enough in memory to start training the net. The “memories” are state-action pairs, what action we choose in what state, and they are chosen to be added to our finite memory based on the score of that

episode. There are two parameters that exist to decide if we did well in an episode, **high_score** and **did_well_threshold**; **high_score** keeps track of the highest score we've achieved in any episode thus far, and **did_well_threshold** dictates how close we need to be to that high score in order to qualify any episode as one in which we did well. As the high score increases, so does our standard for having "done well".

It is well known that some noise is often helpful in training neural nets and this is where our epsilon strategies differ. We have a finite memory of 10,000 state-action pairs, but this memory is used differently by the different epsilon strategies. The faster converging strategy decays epsilon based on the current size of our memory and stops adding new memories once it is full, while the slower converging strategy decays epsilon based on the highest episode score achieved so far and continues adding to the memory indefinitely by purging the oldest memories to make room for new ones. What this difference means is that one strategy continues to purge older lower scoring memories, while the other keeps these "bad" memories and continues training with them randomly included in the training batches. Counterintuitively, the faster converging strategy is the one keeping and continuing to train on the lower scoring memories; what they are doing is providing noise. The decay is set up such that most of the memories are ones in which the agent did well, but that a sizable percentage are from episodes where it performed poorly. The trade-off is that the memory is not always initialized well, which it what causes it to fail converging sometimes.

While the neural net does solve the environment, it does so significantly slower than the parameter search agent. The main benefit the neural net has over parameter search is that we do not have to provide a function form to the agent, it learns one all its own. Thus, the same neural net agent can be trained to solve a variety of environments, which is very unlikely to be the case for a parameter search agent. The memory requirements of the neural net are only those needed to store training data and not nearly as excessive as the nearest neighbors agent. Furthermore, the agent only uses a small subset of its memory for training in any given episode and thus computational costs are also not nearly as taxing as nearest neighbors.

The main tweaked parameters for this agent were the **did_well_threshold** and a parameter in the training section of the neural net called **keep_prob**. The **keep_prob** parameter turns out to be a very important parameter for modern neural networks; what it does is dictate the probability that any given weight in the network is active, where the inactive weights do not contribute to the output of the network. The concept behind this idea is that by dropping some of the weights you force redundancy into the way the function is learned. Our settled upon **keep_prob** was 0.75; for contrast 0.85 resulted in more than doubling the time to convergence as well as the amount of convergence failures for the fast e-greedy strategy, and while 0.65 resulted similar speed of convergence as 0.75, it fails to converge less than half of the time, the lowest convergence rate. Optimizing our **did_well_threshold** parameter is much more intuitive;

a higher value means we're pickier about what memories we keep and a lower value means we're not very picky. Because our fast converging e-greedy strategy is based on how much we have in memory this will directly impact our time to converge since if we're pickier, it will take longer to gather memory. The final setting for this parameter is 0.77; for contrast 0.67 always fails to converge, using our faster e-greedy strategy, likely because there are too many low scoring memories, while 0.87 converges but takes much longer to gather sufficient data because it is being pickier, averaging ~1400 episodes or twice average the number needed for 0.77.

Conclusion

We have seen a variety of reinforcement learning techniques attempting to solve cartpole; two are successful, and two were unsuccessful. We have also discussed the pros and cons of different agents including memory requirements, computation costs, learning speed, and learning generalizability. I hope you have enjoyed this tour of various reinforcement learning techniques. Below are sections on possible future improvements, and general commentary.

There were several aspects of this journey that I found to be interesting, fun, or surprising. I had never seen discretizing used for a continuous space and I initially had the idea as a means of implementing Q-learning, because I needed state-ranges for state-to-state transitions. However, because of the nature of the state space this proved to be very ineffective despite at least half of a dozen different implementation attempts. This is when I decided to use discretizing as an example of the difficulties of such a state space. I was very pleased with how well nearest neighbors worked initially, especially since it was one of the topics we covered; it was my first successful agent implementation even if it didn't perform well enough to solve the environment. I learned a good amount of code just by going creating a dozen or more agent programs based on the idea of nearest neighbors. The parameter search agent was quite frustrating to get right because even though it was very clear that such an agent could approximate solutions in specific ranges of the state space, it was not immediately clear how to tie this together to saturate the space; as they say, hindsight is 20/20, and when I thought of what became to be the final solution it was not only elegant but painfully obvious. The funnest part about creating a neural net for this project was just creating a neural net; the technology seems to be advancing at the speed of light, it's a very exciting field right now. The biggest surprise, and discovery, in creating the neural net implementation was how noise as data, or "bad data", seemed to speed convergence when there was enough good training data to balance it.

There are a variety of more ambitious approaches that would be worth exploring in the future, such as parallel neural nets. Parallel neural net implementations can be used in a variety of ways; one of the more common is training several nets in parallel and then taking the average or majority vote of their output. Doing things in parallel is a trend for more sophisticated projects, with nets they can be played against each other or used to explore the same environment in different ways while combining experience(this is one of the ways Tesla's autopilot feature improves). One can take the approach of using several agents and combining experience with all of the agents explored in this report, this would simply require more computing power and some implementation tweaking.

Below is a graph of the running average vs episode number comparing our various agents: