**Snake With Deep Reinforcement Learning and GYM**

By Samuel J. Bouiss

May 13, 2019

**Introduction**

Generally a reinforcement-learning (RL) problem is a sub-category of machine learning problems where the goal is to map situations to actions in order to maximize a numerical reward (Barto). GYM is a package from OpenAi that aims to standardize how people approach RL problems (OpenAi). In GYM a specific RL problem will correspond to an environment, which consists of an agent, a state space, an action space and a reward. The agent is the entity acting on the environment; the action the agent selects causes the change in state. Each state is associated with a reward; the agent’s goal is to choose the sequence of actions that maximizes its total reward.[[1]](#footnote-1) Generally RL problems come in the context of games, for example the introductory environment from OpenAi is the cart pole problem. In cart pole there is a pole balanced on a cart, the goal is to keep the cart on the screen for about 200 time steps. However there have been some recent practical applications, DeepMind (Google’s machine learning research division) applied deep RL to reduce the cooling costs of one of their data centers by 15% while maintaining the current output (Yuanlong Li).

My goal was to solve the classic game snake, with deep RL. In snake the world is a set of lattice points. The snake spawns at a fixed point, and the apple spawns at a random point (each time it is eaten). The snake’s goal is to grow as large as possible. This is accomplished by eating apples, every apple the snake consumes his body grows a fixed amount, that is the number of lattice points occupied by the snake increases. In my formulation of snake in GYM, I constrained the world to be a 32 by 32 pixel matrix, the snake’s head is represented as the number 10, and any other pieces of its body are represented as a 1, the apple is represented as a -10. The state space will be discussed at length in the result section. At each time step the snake has 3 actions, go right, go left, and do nothing, these actions compose the actions space, . The reward function is as of now a contentious point, which will be discussed further in results.

**Reinforcement Learning**

In our definition of reinforcement learning above, we have an agent who acts on the environment to get a reward. Because the environment at the next time step depends only on the action we take in the current time step, this definition can be reimagined to be a Markov Process. Making this change allows us to develop the tools needed for deep reinforcement learning. From (Barto) we define,

(2.1)

(2.2)

Where is the reward space, is the state space, is the action space,

(2.3)

(2.4)

This framework gives the basis for the lost functions in both of the methods applied, namely deep Q learning, and deep proximal policy optimization.

**Deep Q Learning**

First we define a policy , which is the probability that we take action a given that we are in state s. We define,

(3.1)

We then define a function, to be a function which given the current state, maps it onto the sum of its total expected rewarded given some policy ,

(3.2)

Therefore,

(3.3)

Now we define ,

(3.4)

In light of equation 3.4, equation 3.3 reduces to,

(3.5)

Now we define an optimal policy, (Barto). For an optimal policy, we always select the action that maximizes our expected total return given a current state. In an optimal policy we will never select an action which does not maximize our total expected returns, because that would be suboptimal and thus becomes the Kronecker delta for the optimal action. If we replace with then 3.3 becomes,

(3.6)

where, is the proceeding state and a’ is the proceeding optimal action. This means in a fixed state action pairing, the expected value of our optimal policy is the return at the current time step, followed by the expectation of the optimal policy at the next time step. Now we define the temporal difference error to be the difference between those two for our function (Barto),

(3.7)

For proof of convergence, see (CHRISTOPHER J.C.H. WATKINS).

Now, we have the framework to throw some deep learning at the problem. We take advantage of the fact that neural networks are universal function approximators, and try to learn the function . For training we use experience replay memory with a buffer (Ruishan Liu). Experience replay is a training scheme that alternates learning and playing. First the agent plays under some random policy for some number of steps. After each step an observation is stored in memory, which is typically a tuple, which contains starting state, action, resulting state, and resulting reward. After enough observations are accumulated to form a mini-batch of ones desired size, each k steps of playing is followed by a batch stochastic gradient descent.

Because is a function of state and action the typical convention is to have network take the state as an input and output a vector in , indexing the vector at i should correspond to . Here is the ith element in the set of actions. Notice that this is in the case of a discrete action space, continuous action spaces are possible, but not pertinent to snake, so they will be ignored. Generally vanilla deep q has fallen by the wayside in favor of fancier models. This is consistent with the results we saw which will be discussed in the results section.

**Actor-Critic Models (A2C)**

The remainder of models used will be some variation of actor-critic models. The actor-critic model has two parts, a policy function and a value function (Volodymyr Mnih). These are the same and from deep q. In the model introduced by (Volodymyr Mnih) the first part of the network is fully connected, then the second half is two disjoint parts the policy is a linear layer which has the same number of outputs as the size of . is just a linear layer which has a single output. Other itterations of the actor-critic have the actor and critic each as their own network with no shared parts (John Schulman).

The actor-critic generally tracks the loss of the actor and the critic. In the orignal A2C (Volodymyr Mnih), the loss for the actor is given by,

(4.1)

Where, is an estimate of the advantage function given by,

(4.2)

The term,

is sometimes referred to as the return and is an approximation of 3.1. The loss for the critic is,

(4.2)

Finally the entropy of our policy is added to encourage exploration. Generally the entropy is given a small coefficient because maximizing exploration isn’t our main goal.

The idea of the loss function is similar to that of a control variable in Monte Carlo simulation, by adding more functions meant to approximate the same thing we can get lower variance estimates overall and hopefully converge to a reasonable answer more quickly. Apparently this works well, all of the more modern techniques use this idea in some way.

One modification to A2C is A3C, asynchronous actor-critic (Volodymyr Mnih). A3C uses multithreaded agents, each acting in their own environment. The experiences each of the agents accumulates over some number of steps forms a mini-batch which is used to preform gradient descent. This procedure is shown to drastically speed up the training process in (Volodymyr Mnih).

**Trust Region Methods**

The two main trust region methods are trust region policy optimizations (TRPO) and proximal policy optimization (PPO) (John Schulman). TRPO and PPO use the same network structure as A2C; that is a policy and value function estimate. They also add a new element, a target network, which keeps an older version of the parameterized policy network, and periodically updates it.

In TRPO the loss function becomes,

subject to,

(5.1)

Interpreting this, the ratio will be closest when the probabilities of our old and new models agree. So if the advantage of the action we took is negative and our models agree we will correct for this slightly and try to decrease the probability of taking that action. If the models disagree then the ratio will either tend to 0 or infinity, the largest correction would occur if the old probability was near 0 then new probability was near 1 and the advantage is positive. The inclusion of the constraint is to stop the model from making overly optimistic steps.

Practically working with constrained optimization is difficult and computationally more expensive than deep learning already is. Generally a constrained problem would require computing a hessian so we can find the saddle point for the constraint. A solution to this is having and adaptive KL penalty (John Schulman). Under this setup the loss function takes the form,

(5.2)

, is a dynamic coefficient, which is reliant of the choice of a target for the KL divergence similar to the original constraint. Every time the KL divergence is greater than the target by a multiplicative factor of 1.5, is doubled; similarly every time it is less than the target by a multiplicative factor of , is halved. This is meant to hopefully approximate a reasonable for the constraint, and obviously since the old and new policy change it stands to reason should too.

In PPO, we are trying to achieve the same end but through different means. The value of the ratio and the advantage functions become clipped to the ranges and respectively, this is meant to achieve the same thing as in TRPO, but without the use of an adaptive penalty. The loss in PPO is,

(5.3)

Basically, we will follow the gradient for but only to a certain point and anything beyond that will get clipped.

**Results**

There aren’t really any pretty graphs to show the training results. The standard metric seems to be mean return achieved over some number of iterations with the given policy. For computational reasons doing a large number of iterations of training with all the models is not really feasible.

The initial state space considered was just the matrix that represents the world. Each entry in the world could be a snake body, a snakehead, an apple, or none of the above. Meaning the state space for this representation of the game for a board of size is , this is a massive number, which makes the function much more difficult to learn. True to this idea, all models trained regardless of procedure fail to converge in a reasonable number of steps with this state space. (Bowei Ma) propose a different state space, 3 indicator functions which represent if you will die by using actions 1,2 or 3, and two functions which give the relative position of the apple and the snakes tail with respect to the snakes head. Under (Bowei Ma)’s reduced state space the size of is or . The reward scheme proposed by (Bowei Ma) is, +500 each time the snake eats an apple, -100 each time the snake dies, and -10 else.

We use a modified version of this state space and reward scheme, for the state space. For the state space we use the same three indicator functions for if you will die by going forward, to the left, or to the right, and indictor functions for if the apple and mean of the body is to the left, ahead or to the right. The idea is that the combination of indicator functions might be more useful than just knowing my tail is predominantly to the left. This does raise the size of the state space marginally but seems like a reasonable change. Additionally, we use the (Bowei Ma) penalty without the for idling condition.

A simple policy was constructed as a baseline, each action starts with a score of 0. Then if you would die from taking any of the actions is added to its score, if it would bring you closer to the apple 10 is added to its score, and if your tail is in that direction -1 is subtracted from its score. The policy is the soft max of the three actions scores. So first we care about not dying, next we care about going to the apple and final if all things are the same we prefer not going in the direction of our tail. The mean reward over ten episodes for this policy is about 7000, under this policy the snake eats about 14 apples on average.

Vanilla Q learning with multithreading and with replay experience was never able to achieve meaningful results. The policy from A3C was able to achieve 25000 mean reward over ten episodes with 100000 episodes of training. The policy from PPO was able to achieve similar results with about 100000 episodes but using 2 to 3 epochs per episode. Ultimately the training for A3C was quicker but less stable, meaning it would shoot down after finding a good solution where as PPO had variance in how good it’s policies were but the variance was less extreme. Under these constraints it seems hard to determine an optimal policy because you are seriously simplifying the problem. Maybe with a slightly more complicated state space but one that is still simpler than the entire board better results could ultimately be achieved.

# Works Cited

Barto, Richard S. Sutton and Andrew G. Reinforcement Learning: An Introduction. 2nd Edition. The MIT Press, November 5, 2017.

Bowei Ma, Meng Tang, Jun Zhang. "Exploration of Reinforcement Learning to SNAKE." Academic. n.d.

CHRISTOPHER J.C.H. WATKINS, PETER DAYAN. "Q,-Learning." (1992).

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. "Proximal Policy Optimization Algorithms." n.d.

OpenAi. Gym Documentation. <https://gym.openai.com/docs/>.

Ruishan Liu, and James Zou. "The Effects of Memory Replay in Reinforcement Learning." Academic. n.d.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning." Academic. Google DeepMind & Montreal Institute for Learning Algorithms (MILA), University of Montreal, 16 Jun 2016.

Yuanlong Li, Yonggang Wen, Kyle Guan, and Dacheng Tao. "Transforming Cooling Optimization for Green Data Center via Deep Reinforcement Learning." (2018).

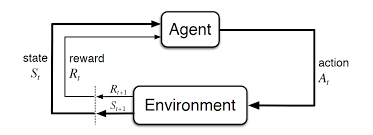
1. 

   Diagram of agent-environment interaction (Barto) [↑](#footnote-ref-1)