Ikhlas Attarwala
06/14/19 – DRL
Final Project

Final paper:

Not having a background in information sciences, I chose to tackle several papers to understand the breadth and history behind Reinforcement Learning (RL). Below you'll find a one-page summary per research paper I've come across that I found fascinating and worthwhile. The progression starts off with certain RL methods, and then it glides into real-world applications. My one regret is not discussing RL in robots. Unfortunately, while fascinated, I was too uninitiated in the world of robotics to understand what was happening in those articles. Jeremy had emailed me and said summarizing 3 papers would be enough for this project; I summarize and dissect 6 papers below, and I also worked on a jupyter notebook deliverable to tackle a fun problem set; that is included in the zip file that contained this document. Enjoy the read!

Papers in order:

1. *An adaptive optimal controller for discrete-time Markov environments*
   by Ian Witten.

2. *Learning from Delayed Rewards*
   by Christopher Watkins

3. *On-line Q-learning using connectionist systems*
   by Rummery and Niranjan

4. *Deep reinforcement learning doesn't work yet*
   by Alex Irpan

5. *Giraffe: Using Deep Reinforcement Learning to Play Chess*
   by Matthew Lai

6. *Playing Atari with Deep Reinforcement Learning*
   by Google DeepMind

Citation #1:

Witten, I.H. (1977) “An adaptive optimal controller for discrete-time Markov environments.” Information and Control, 34, pp. 286-295, August.

Paper summary:

To lay the red carpet for reinforcement learning, we could begin in 1911 with Thorndike’s paper on the psychology of animal learning (I’m a neuroscience major, this would be fun to discuss), but for the sake of the class and assuming you understand the simplest biological definition of “reinforcement learning” as it pertains to change by reward, we’ll start by addressing Markov Chains from the 1930s. Markov Chains are stochastic mathematical systems that transition from one state to another, where each future state can only be one of the potential possibilities stemming only from the present state, no matter how we arrived at the present state. Therefore, the probability of transitioning to ANY state is solely dependent on the current state, and the lack of history of previous states makes this memory-less.

Now, while the idea of temporal difference popped up in Samuel’s paper from 1959, Ian Witten’s 1977 paper introduces the learning rule for RL. He discusses the theoretical significance of a general-purpose controller that relies solely on the indication of whether the environment is currently a desirable state or not. This method evaluates the worth of its actions and transforms the calculation to a direction of future adaptation. The environment itself provides instantaneous reward information, and instantaneous state indication. The interesting thing about this controller is that it is not greedy, and instead evaluates long-term reward scoring. The controller is “encouraged …to traverse a valley in the reward space if this leads to a sufficiently high peak”, but it’s important to note that a discount factor also exists that weighs immediate future states in the environment as being less costly than more distant, future states.

I discussed the Markov Chain earlier because the environment Whitten discusses obeys the Markov property, where the choice of control action only depends on the current state, and not its precedings. The introduction of temporal-difference learning here is what we call approximating the current estimate based on the previously learned estimate (now called TD(0)), where the model dynamics don’t need to be known in advance. It’s important to note that while I won’t be discussing the paper, Sutton’s 1988 paper formalizes TD Learning, calling it prediction-learning by means of the difference between temporally successive predictions.

Citation #2:

Christopher John Cornish Hellaby Watkins. 1989. Learning from Delayed Rewards. Ph.D. thesis,King's College, Cambridge, UK.

Paper summary:

This sets us up for the next paper to introduce from 1989 as it relates to Dynamic Programming. A very well-known paper by Christopher Watkins, "Learning from Delayed Rewards" discusses an iterative method for solving the maximum value of an environment with a large number of states by understanding the transition matrix as well as the rewards. In order to discuss Dynamic Programming, we need to first talk about the Markov Decision Process (MDP).

We already discussed the Markov Property earlier, but now we'll dive deeper. The Markov Process is a sequence of random states with the Markov Property, essentially a tuple of the state space and the transition probability function. The Markov Reward Process is a Markov Process with an accumulation of value/reward through the sequence sampled; this is a tuple of the Markov process, reward function and discount factor. Finally, the MDP is an environment in which all states are Markov (Markov Reward Process with decisions); this is a tuple of the Markov Reward Process alongside finite actions.

While Christopher Watkins' thesis paper is over 200+ pages, the idea is to control an MDP via incremental dynamic programming steps. In other words, this thesis is a collection of numerous algorithms (including Q-learning) that calculates optimal strategies given the ideal environment as an MDP. This giant, gleaming problem here is that an assumption is made that the environment follows a perfect model. Additionally, computational cost is tremendous, and the model is assumed to be known, unlike traditional RL practices.

Of course, this doesn't make dynamic programming obsolete, as there are methods that can, say, learn a model from samples, and then use the model in the dynamic learning process, which can be a hybrid use of RL and dynamic programming.

Citation #3:

G. A. Rummery and M. Niranjan. 1994. On-line Q-learning using connectionist systems. Technical report.

Paper summary:

Originally I was only going to look into Q-learning as it still pertains to Christopher Watkins' paper, but we covered it well enough from class already. Instead, I'll jump a little bit and discuss SARSA, which stands for $State_t$-$Action_t$-Reward-$State_{t+1}$-$Action_{t+1}$. SARSA is an algorithm developed by Rummery and Niranjan in 1994 as they examined the use of backpropagation in storing information learnt by the Q-learning algorithm. The Q-learning learns the Q-function, that predicts the output of every possible action in every state. It updates and attempts to maximize the expected return of the total return over all successive steps from the current state (essentially it learns a policy to tell an agent what action to take under what circumstances).

Rummery and Niranjan discuss Lin's work in 1992 to build a "Connectionist Q-Learning" algorithm, where the original tabular form of each Q-function is replaced by a neural network. Each action is made up of its own neural network, and for each iteration, the inputs to the neural network is the current state of the system. Interestingly, the weights are only updated for the individual network where the action was performed. I'm curious why there wasn't any weight change for the other networks depending on the action-state combination that yielded the most reward. The authors go on to discuss a modified version of the Connectionist Q-Learning (this modified version was later coined SARSA by Sutton). This version looks at a combination of Q-learning and TD-learning (both of which we've discussed now) proposed by Watkins' earlier paper, with a heavier emphasis on TD-learning. You see originally, in the Connectionist Q-Learning algorithm, each change in weight had the discount function be multiplied by the maximum value of Q_{t+1} with respect to the action taken. In the modified version, aka SARSA, they avoid the greediness that can often be an overestimation, and instead just multiple the discount function by Q_{t+1} for the selected action, irrespective of its returned value. It's subtle, but now if a greedy action is taken on, it's essentially just normal Q-learning.

The reason it is called SARSA is because the Q-value depends on the current state, action chosen for that state, the reward for choosing the action, and the next state and next action to lead to a new state. It's important to know that Q-learning is referred to as an "off-policy" algorithm because it learns the value of the optimal policy independently of the actions taken, but SARSA is "on-policy", as it learns the value of the MDP policy, as well as exploration steps.

Citation #4:

Alex Irpan. 2018. Deep reinforcement learning doesn't work yet.https://www.alexirpan.com/2018/02/14/rl-hard.html.

Paper summary:

Instead of simply discussing Deep Reinforcement Learning (DRL), I'll do it by addressing a paper that talks about the **limitations** of DRL. Alexander Irpan recently wrote a blog post about the critical limitations of DRL, but first, we define it.

Let's first say that many RL approaches lack scalability, and for the most part, have trouble with high-dimensional tasks due to memory complexity, computational complexity, or even sample complexity. Using function approximation (generalizing an estimation of values at states to reduce state space size) and representation learning (learning representations of data to make information extraction to build classifiers/predictors easier), deep neural networks have powerfully handled these problems. The best part is that we can still rely on low-dimensional representations with deep learning because that's the crux of machine learning: it compacts higher-dimensional data for us to use with ease (Jeremy said this in class, proof I was paying attention). DRL is the combination of deep learning algorithms with RL, allowing RL to scale to levels where it can deal with high-dimensional states and actions.

So, what's the problem? Alex suggests the following (to name a few):

1. DRL is ridiculously sample inefficient (ie. it can take thousands of CPU hours to show us impressive results)
2. Sometimes hardcoding will save a world of trouble than exploiting problem-specific information via learning
3. If your reward functions are 100% perfect, it'll throw off the RL algorithm
    a. Additionally, even if the reward function is great, incorrect exploitation of a system can lead to results you don't want
4. DRL is more a research topic than a productionized system

There are more concerns, but for the most part, Alex argues that there's too many problems to consider, and often different environments and cannot take the same approach. I did have a few thoughts:

1. I found it interesting that 30% failure is considered working, so that would mean the the model may be perfectly fine, but it just sometimes doesn't work
2. In the beginning, he compared training to human standards, saying that simple tasks may take hundreds to thousands of hours, but we've been exposed to very similar stimuli and experiences our entire life to allow us to come this far in the first place, so we've already had our "training"
3. Alex talked about exploration-exploitation of the system as a flaw, but sometimes this can show us what we can fix or that which we never considered.

Citation #5:

Lai, Matthew. "Giraffe: Using Deep Reinforcement Learning to Play Chess." ArXiv.org, 14 Sept. 2015, arxiv.org/abs/1509.01549.

Paper summary:

I could have talked about AlphaGo, a well-documented project developed by DeepMind for its incredible accomplishment of defeating a Go world champion, but it's a more commonly known story than, say, Giraffe, a project I was unfamiliar with and wanted to read more about.

Now, while a lot of recent RL attention has been focused on video games, board games like Chess are still of crucial interest. John Tromp has calculated an upper bound of $7.7*10^{45}$ potential states that can exist in the game of Chess. Whether or not that's true, the fact remains that there is a ridiculous complexity behind the game of Chess. Giraffe was a unique thesis project introduced in 2015 by an employee of Google's DeepMind team. It has been discontinued since 2016.

AlphaGo (and I believe AlphaZero too) used an artificial neural network to evaluate positions with a combination of the Monte-Carlo Tree Search algorithm. Giraffe did something similar but didn't use the Monte-Carlo Tree Search algorithm. It used a nega-max algorithm (variant of minimax that relies on the zero-sum property of 2-player games), to replace the regular evaluation function with a neural network. This was replicated from a previous work called TDLeaf from the KnightCap engine. Additionally, it used a machine learning model to order the nodes within its tree search algorithm. Unfortunately, Giraffe was far too slow due to its usage of a neural network.

A unique note about Giraffe was that the author, Matthew Lai, trained and tested neural networks using 5 million random positions from high-quality games (his method of feature engineering, and it's kind of like introducing expert opinion into a game). Giraffe learned by self-play and evaluating these 'expert' positions. The author used a bootstrapping technique to have it play against itself with the goal of improving its own evaluation functions. It did not need to anticipate far in advance to perform well, but complicated positional concepts seemed to be its largest issue.

Given the time this paper was published, the results were a bit disappointing considering there was no serious novelty, and DRL has shown to outperform human capability. Nevertheless, it was an interesting read, along with related discussion threads I read online about it, to gauge the bar on performance by year.

Citation #6:

Volodymyr Mnih, Koray Kavukcuoglu, David Silver,Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning.arXiv e-prints, pagearXiv:1312.5602.

Paper summary:

Finally, we're going to going to jump backwards just a little bit and discuss Google DeepMind's famous paper, which was referenced by Alex Irpan in his blogpost that I discussed above. DeepMind took a glance at the Atari 2600 gaming system that ran on 4kb of RAM. While traditional methods at the time took the approach of taking advantage of the abstract representations and objects on the screen in the emulator, the authors of this paper learn several Atari games by studying the raw 210x160 video and score state.

In order to do so, the screen was convoluted from 210x160 to 84x84x4, and then fed as 28,224 input pixels through 2 hidden layers of a CNN, and 1 non-convoluted hidden layer that consisted of 256 nodes where each led to a discrete output. A variant of Q-learning was utilized to train the weights in the neural network. It seems they use a **sequence** of a state rather than an individual state; this is also called a sequence of consecutive frames. The idea behind using a sequence is that individual frames cannot show you that a person is walking, for they could just be holding a odd-standing position, but a window of states can show movement and progression. Each sequence then is an individual state (I believe consisting of 4 frames), and this state still obeys the MDP. Like traditional Q-learning, the approach is model-free (not explicit learning of rules or physics of the game), and off-policy. This variant of Q-learning was coined as a Deep Q-Network, or DQN.

Their DQN scored higher than all previous RL approaches in 6/7 games and beat the median human score after 2 hours of gameplay in 3/7 games. As expected, they did not adjust the architecture nor hyperparameters. Succeeding this achievement, DeepMind went on to publish a paper in 2015 that combined Deep Neural Nets with Reinforcement Learning to master 49 Atari 2600 games, again using raw pixels and score as input. This, they called the first demonstration of a general-purpose agent that continually adapted its behavior in a decentralized fashion.

Later work into Atari games such as Xiaoxiao Guo et al.'s paper on Offline Monte-Carlo Tree Search Planning was also an interesting read. It essentially used 3 methods to evaluate the combination of RL and DL offline Monte-Carlo Tree Search to train a deep-learned classifier, and attempt to outperform DeepMind' DQN. They had comparatively better results in 2 of their 3 methods.