Machine Learning and Games

Ben Kompa
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC, 27514

Abstract

We survey current techniques of machine learning and how they can be applied to solving games. The techniques include Deep Reinforcement Learning and Monte Carlo Tree Search. We will also mention the open problems of these fields. Finally, we will cover games recently solved or advanced by machine learning including Go and poker.

1 Introduction

Since the early days of artificial intelligence, games have been used as a yardstick of success[1]. Games have encouraged computer scientists to advance algorithms to conquer them. Solving games has more than an academic purpose though. If computer scientists can create algorithms that can generalize to a large number of tasks, compete in spaces of imperfect information, and plan for the long term, this could have enormous impacts. The same algorithms could be applied to finance, security, healthcare, and perhaps politics to improve our world[2].

Specifically, in order to solve a game, one derives a Nash Equilibirum for a game[3]. A Nash equilibrium is essentially a perfect strategy – no agent can increase its score by deviating from it[3]. However, the techniques of game theory are quite poor at developing Nash equilibirums for games of imperfect information. This is where machine learning researchers can substitute in technqiues like neural networks to approximate the Nash equilibirum.

While Nash equilibrium are optimal, researchers also create techniques to simply outperform humans. There is a wide variety of machine learning techniques that are utilized to play games and outperform humans. These include deep reinforcement learning Monte Carlo Tree Search. Each technique can be applied to the problem of games to try to overcome some of the challenges of playing games with AI. These challenges include transfer learning, long term strategy planning, and understanding imperfect information situations.

In the follow sections, we will describe the basics of each technique, the new developments in the field, and the open problems facing researchers today. Following that, we will survey how two games have been affected by the development of machine learning: poker and Go. These games were previously all thought an extreme challenge for computers, but by using machine learning techniques, researchers were able to reach super human levels of performance. This survey will demonstrate the rapid pace of advancement in machine learning. It is staggering to consider the possibilities of what researchers may accomplish in the next few years.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

2 Deep Reinforcement Learning

2.1 Introduction

Deep reinforcement learning is a group of algorithms that implement the idea of reinforcement learning[4]. Reinforcement learning trains an agent(s) that make decisions in an environment based on the current state of the environment in order to maximize reward. This appeals to researchers on a psychological level as this is a model of organismal behavior. More formally, we can consider the environment as a Markov decision process in which the Markov property

$$p(R_{t+1}, S_{t+1}|S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t) = p(R_{t+1}, S_{t+1}|S_t, A_t)$$

where R_t, S_t, A_t are the reward, state, and actions of the agent at time t[4]. In reinforcement learning, we seek to estimate the optimal action-value function $q_*(s, a)$:

$$q_*(s, a) = \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma \max_{a'} q_*(s', a')]$$

where r is a reward function and γ is the discount rate so future rewards are downweighted[4]. One can see how this might imitate a Nash equilibirum. With an optimal action-value function, an agent will just query based on all possible actions given the current state and the agent will have its next action. Deep reinforcement learning focuses on using neural networks to learn the policy functions and action-value functions. Of current interest is Q learning, which uses neural networks to learn the action-value functions[4].

2.2 New Developments

In fact, the deep neural network implementation of Q learning is a rather recent development. Although Q learning was first introdcued in 1989 by Watkins, in early 2015 Google Deepmind release a paper outlining their implementation of Q learning using deep neural networks. Mnih et al were able to resolve the divergence of reinforcement learning algorithms when using neural networks[2]. First, experience replay was used. Observations were randomly presented to the agent to prevent correlation in input sequences. Second, they updated the action-values to target values only periodically. Mnih et al updated their neural networks with the loss function:

$$L_i(\theta_i) = \mathop{\mathbb{E}}_{(s,a,r,s') \sim \mathrm{U}(D)} [(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2]$$

Mnih et al were able to show that their Deep Q Network (DQN) could use the same architecture and hyperparameters and exceed human performance on 49 Atari video games[2].

Since the 2015 paper, many others have leveraged and improved deep Q learning in their own research. Van Hasselt et al corrected an issue with overestimation of action values in Q networks under certain conditions. They were able to establish that these overestimations are common, harm performance, and can be corrected. Van Hasselt et al developed Double Q Networks, which correct for overestimation by using two sets of θ weights, one to create targets for the DQN and one to evaluate actions[5]. In this way, the DQN does not use the same weights to both select and evaluate the same actions. In this 2015, Van Hasselt et al introduced Double Deep Q Networks, which take advantage of these decoupled weights[5]. Using Double DQNs, they achieved state of the art results on the Atari tests that Mnih et al used. Mnih et al had a mean normalized performance of 122.0% on the 49 games while Double DQN had a mean normalized performance of 273.1%[5].

While Van Hasselt tweaked deep Q learning to correct the overestimations, other groups are changing fundamental aspects of deep Q learning. In 2015, Nair et al pioneered parallel methods for deep Q learning[6]. They created a framework where parameters of the Q network are stored on a server and agents training on multiple machines can send gradient updates to those parameters[6]. In this way, the training of the Q network is massively sped up, in many cases taking only one day of training to beat a single machine DQN network. Mnih et al expanded on the idea of speeding up training in a 2016 paper where they developed asychronous methods for training deep reinforcement algorithms[7]. By training multiple agents in parallel on different CPU threads, Minh et al were able to dramatically reduce the computational power needed for Q learning. Now, instead of training on distributed GPUs like in Nair et al, a single CPU machine was used to

achieve massive improvements in Atari scores[7]. By training in parallel on the same environment, Mnih et al removed experience replay from their DQN algorithm[7]. Multiple agents exploring different areas of the environment provide the stability that experience replay did. Removing experience replay allows many other reinforcement learning algorithms, ones that don't focus on state-action functions but instead value functions, to be trained with this method. With an actor-critic algorithm and asynchronous training, Mnih et al were able to score 623% on the Atari benchmark[7].

Wang et al put another twist on the DQN architecture structure. Instead of having a Q network that simply outputs a reward for each action, they developed an architecture that outputed the value and advantage functions. The advantage function is defined at $Q - V$[8]. This architecture is known as a deuling architecture. Estimates of Q are recovered by implementing:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha)$$

This separation of value and advantage has considerable improvements over standard DQN algorithms. Deuling networks score 591% on the Atari framework[8].

[9] In other developments, researchers have started to combine other reinforcement learning frameworks of Q learning. In a 2017 paper, O'Donoghue et al combined policy gradient and Q-learning[9]. Policy gradient involves explicitly modelling the policy and updating its parameter in the direction of the gradient of performance. O'Donoghue et al were able to derive a correspondence between policy gradient and Q-learning[9]. This allows them to develop Policy Gradient Q Learning, in which a policy is updated not only by its gradient, but also by a Q network that estimates action-value. This dual update strategy proved very succesful with an average Atari score of 877.2%[9].

In a related 2017 paper, Schulman et al developed trust region policy optimization, which monotonically increases performacnes of policies[10]. This optimization works by first collecting a set of state-action pairs with Monte Carlo estimates of their Q-values, then averaging over the samples to construct an objective and constraint, then solving this constrained optimizaiton. The authors prove that solving this constrained optimization is guaranteed to monotonically increase policy performance[10]. The policy's parameters θ are then updated and the whole cycle can be repeated. Trust region policy optimization was applied to three computational simulations of movement. The authors did not have enough time to perform a full Atari benchmark[10].

Another example of modified reinforcement learning is from Barreto et al in a 2016 paper focusing on transfer learning[11]. They develoepd a way to transfer learning where reward functions vary but the environment remains constant. It involves creating successor features to capture the time discounted reward functions at all state action pairs. The authors formally define sucessfor features by:

$$\psi^{\pi}(s, a) = \mathrm{E}^{\pi}[\sum_{i=t}^{\infty} \gamma^{i-t} \phi_{i+1} | S_t = s, A_t = a]$$

where ψ is the successor features and ϕ_t is defined from a reward function $r(s, a) = \phi(s, a)^T w$[11]. These successfor features represent how the dyanmic of an environment are changed by a specific policy. By learning weights w and ψ, the authors are able to demonstrate transfer learning[11].

Finally, Pérolat et al develoepd techniques to learn Nash equilibrium of Markov games using some of the techniques of Q learning[3]. We see here a more direct theoretical connection between games and reinforcement learning. Markov games are very related to the Markov decision processes that the above papers generally focused on. Instead of one agent exploring an environment, N agents explore an environment simultaneously and also earn a reward base don the joint actions of all agents. We can consider a policy (strategy) π to be a Nash equilibirum if $\forall i \in \{1, ..., N\}, v_{\pi}^{i} = v_{\pi^{-i}}^{*i}$[3]. This means for each player i, the player cannot improve his own value by changing his action unless other players change there actions. Pérolat et al go on to train Q networks to learn the state-action functions and policies of a Markov game and show that the learnt strategies are in ϵ-Nash equilibirum, a slightly weaker version of Nash equilibrium[3].

In summary, we see that there has been many theoretical advancement in reinforcement learning

in the past few years. Researchers are identifying issues with current reinforcement learning algorithms and by correcting these issues, find out new fundamental features of the networks.

2.3 Open Questions and Future Directions

Despite all these advancements, there are still many avenues of open research in reinforcement learning.

2.3.1 The Value of the Value fuction

A few researchers have previously considered how to better construct reward functions for learners[12][13][14][15]. The standard value function is not necessarily optimal. Given that we consider future rewards, if the rewards are very sparse, this can lead to poor training. One might consider trying to "inspire" the learners with intrinsic motivation. Not to anthropormorphize, but if one can cause the learners to move throught the environment in efficient ways of their own accords, not necessarily with rewards, this could lead to better training. This would involve modifying the standard value function .Already there is some early work on this subject. Still, getting agents to efficiently explore the environment is a hard problem.

2.3.2 Biological Bridges

Another area to explore is the increasing connections between machine learning and neuroscience. Some scientists that straddle biology and machine learning think the brain may implement cost functions in learning[16]. Mnih et al suggest that the hippocampus may provide experience replay for the brain while we rest[2]. Others have investigated how successor representations are intertwined in human reinforcement learning[17]. These successor representations were generalized by Barreto et al with their successor features.

3 Monte Carlo Tree Search

3.1 Introduction

Monte Carlo tree search is a popular method in machine learning to play games[18]. It focuses on creating a tree of game states and exploring which games states have the highest value. One can think of Monte Carlo tree search as having four main parts[18]:

- Selection
- Expansion
- Simulation
- Backpropogation

Selection involves deciding which node to expand. Expansion is just that, the most urgent node is selected and an action is added to the tree as a leaf node. Together, selection and expansion form the *tree policy*. How one decides which nodes to select and expand form a major part of Monte Carlo search tree theory[18].

In simulation, a new node is evaluated by looking ahead and seeing the result of taking the action that produced that node. That result is then backpropogated through the tree to inform the network about how beneficial taking that particular action was. Together, simulation and backprogation form the *default policy*[18]. The tree and default policies and then iterated to explore the state-action space.

One of the main issues in Monte Carlo tree search is balancing exploration of new nodes and exploitation of exisiting nodes. Most versions of the Monte Carlo tree search algorithm today use a variation fo UCT, or Upper Confidence bound for Trees[18]. It can be written as:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\log t}{n_i}}$$

where w is the number of wins after a node i, n is the number of times that node has been visited, $t = \sum_i n_i$, and c is a constant. One tree policy is to choose the node with the highest UCT value. For unexplored nodes, $n_i = 0$, guaranteeing all children are expanded for a layer before we consider new layers. It was shown that Monte Carlo search trees converge to minimax trees and are optimal[18].

3.2 New Developments

In a 2014 paper, Lanclot et al were able to show that by maintaining a bit of extra information in the tree, one could increase tree performance[19]. The key insight was to maintain a minimax heuristic value derived from the evaluation of subtrees below a node. Then Lanclot et al modified the tree policy rule to use:

$$Q^{IM}(s,a) = (1-\alpha)\frac{r^{\tau}_{s,a}}{n_{s,a}} + \alpha v^{\tau}_{s,a}$$

where r is the reward (wins) and v is the heuristic function[19]. This was shown to improve performance in the games of Kalah, Breakthrough, and Lines of Action[19].

Another interesting modification to the UCT tree policy rule was suggested in a 2016 paper by Devlin et al[20]. Their goal was to mimic human play in the card game Spades, which is a trick taking card game. A popular app used Monte Carlo tree search to create an AI for players to play against, but players complained that the AI, though strong, was not very human like in its plays. Devlin et al sought to bias the Monte Carlo tree search towards choosing human-like moves[20]. They modified the UCT equation by writing:

$$UCT_i = \frac{w_i}{n_i} + .7\sqrt{\frac{\log n}{n_i}} + C_{BT}\sqrt{\frac{DR}{n+DR}}P(m_i)$$

where $P(m_i)$ is the probability a human would choose that move according to an exponential distribution derived from gameplay data[20]. This resulted in a decrease of the KL divergence from human moves from 0.1142 to 0.0291. The agents kept a win rate of approximately 50%[20]. Others are turning to parallel computing to try to improve Monte Carlo search trees. In 2016, Mirsoleimani et al focused on this issue[21]. Their idea was to use pipelining to reduce the three overheads that are common problems in parallel Monte Carlo search trees:

- Communication overhead between processors
- Synchronization overhead waiting for processes to finish across machines
- Search overhead for searching the same space on different machines

The key idea of this work is taken from computer architecture: pipelining. Pipelining is the practice of trying to utilize all parts of an idle machine at the same time to speed up a job. Now, one machine could be selecting the next node, while another machine is expanding the previous node, while another machine(s) is simulating from two nodes ago, and yet another machine is backpropogating the rewards from three nodes ago. In this way, little time is wasted from waiting for stages to complete. There is however, the issue of balancing the amount of time spent in each stage. The authors resolved this by suggesting using two processors for the simulation stage[21].

Others are focusing on applying Monte Carlo tree search directly to games. A NIPS 2014 paper from Guo et al combined the look ahead of Monte Carlo tree search with neural networks to achieve start of the art results on the Atari Learning Environment[22]. They merged convolutional neural networks on the UCT algorithm of Monte Carlo tree search. They had three agents that trained in the Atari environment[22]:

- UCTtoRegression
- UCTtoClassification
- UCTtoClassification-Interleaved

For the first two agents, UCT was used to evaluate game states and then a neural network was trained via regression or classification respectively to select the best action. In the Interleaved agent, the UCT agent ran for 1/4 of the training runs, then a neural network suggested states to explore. In

this way, the network was able to train on data it was likely to see. Guo et al achieved state of the art results on the 7 Atari games it was tested on[22]. However, there are major drawbacks to this work. While DQNs can run orders of magnitude faster than real time, UCT agents are orders of magnitude slower than real time. Additionally, the UCT agents had access to the game state, while this is not something a human player or a DQN agent can do[22].

3.3 Open Questions and Future Directions

Although there has been substantial progress on Monte Carlo tree search since it was introduced in 2006, there is still a zoo of issues to consider. One of the main problems is the speed of UCT algorithms. Oftentimes the simulation is a bottleneck in performance. Although pipelining may be one answer as we saw above, perhaps there are other ways to accelerate UCT training. Other issues include:

- Understanding the zoo of Monte Carlo tree search
- Optimal exploration algorithms
- Estimating node reliability
- Tree Shape Analysis

In a 2012 review, there was 50+ different ways of doing Monte Carlo tree search[18]. Understanding this many variations is a challenge in meta-anylsis. There is work to be done in understanding what parts of each variation are useful and what should be investigated more. All of the differences in algorithms concern exploration. There is sitll no optimal way to balance exploration and exploitation[18]. Oftentimes researchers still hand tune the c constant that balances the two factors in the UCT equation[18]. Additionally, it's not known how large n_i should be before a node is trusted to give a reliable estimate of reward[18]. Finally, Williams et al did interesting analysis in 2010 about how the shape of the tree produced by Monte Carlo tree search corresponds to human interest in the game[23]. This could be used to identify the best candidates for new games from mahcine generated games[23].

4 Go

The game of Go was previously thought intractable because of the size of the its state space[1]. Go is played on a 19x19 board where the objective is to capture territory by placing stones on the intersections of the lines on the board. In games of smaller state spaces, such as chess and checkers, it is possible to expand the game state tree sufficiently and use value functions to approximate who will win the game.

Researchers at Google DeepMind were able to build a Go AI that exceeded human capibilities for the first time[1]. AlphaGo, the AI, is a novel combination of reinforcement learning and Monte Carlo tree search. The figure below is a visual representation of the components of AlphaGo[1].

The first network was trained a fast rollout policy. This policy is an extremely fast policy that is used to score games. It takes only 2μs to evaluate a move in this policy. The rollout policy is a linear softmax policy that is fed Go specific features about the current board state. It can execute approximately 1,000 simulations per second per CPU thread. The rollout policy can afford to be more inaccruate because of the Monte Carlo search tree is another part of AlphaGo[1].

The next network was a Supervised Learning policy network, or SL policy network. The SL policy network was trained on 30 million expert human games for an online Go site. The goal of the SL policy network is to select which moves the experts play given a board position. The network was able to recapitulate this 57.0% of the time, besting the previous state of the art of 44.4%[1].

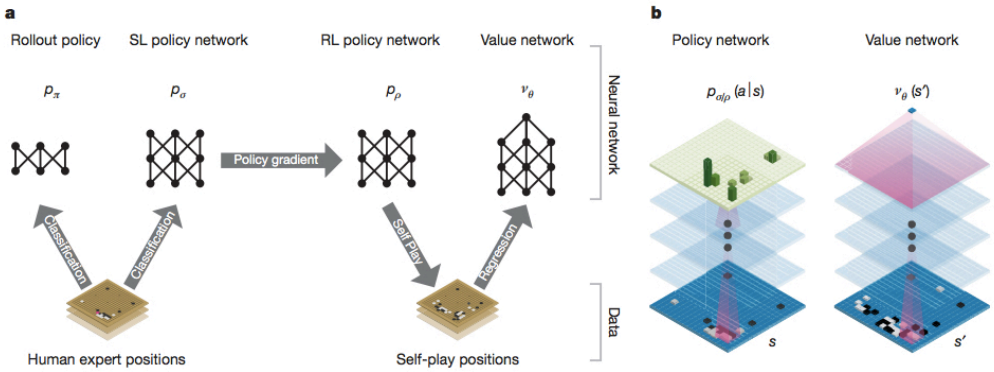

Figure 1: Visualizaiton of the networks of AlphaGo[1]

The researchers then took the trained SL network and used reinforcement learning to train further. This became the RL policy network. The RL policy network was used to trained value network, which approximated the optimal values for each position[1].

All these networks were used in Alpha Go's Monte Carlo tree search. When selecting an action to consider, AlphaGo uses:

$$a_t = \text{argmax}_a(Q(s_t, a) + u(s_t, a))$$

where

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

The bonus u is proportional to the probability that action was selected by the SL policy. Once a node is selected, that leaf is evaluated using two of the trained networks: the fast rollout network, which returns a reward z_L and the value network, v_θ. The value of leaf s_L is a combination of two:

$$V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L$$

This $V(s_L)$ is used to update the Q value of a node and then AlphaGo selects the most traversed node as its next move[1].

AlphaGo was measured in virtual tournaments against the best Go AI's available. It won 99.8% of those matches. A distributed version of AlphaGo that used parallel computing won 100% of its matches. However, it was the match against Fan Hui, a professional Go player and current European champion of Go, which garnered AlphaGo worldwide fame. AlphaGo won 5-0 against Fan Hui. This represents a major achievement in AI and machine learning research[1].

5 Poker

In early 2017, more AI milestones were surpassed. This time, a team from the University of Alberta were able to create a AI that beats professional poker players at Heads Up No Limit Texas Holdem[24]. Texas Holdem is a game of incomplete information. Players have two private cards, in addition to a pot in which players bet and community cards. Bowling et al developed DeepStack, which uses counter factual regret minimization and neural networks to strongly play poker[24].

Deep stack consists of three components[24]:

- Local strategy computation for the public state
- Limited lookahead with a value function
- A set of lookahead actions

This mimics the structure of a Monte Carlo search tree in spirit. Describing the details of counter factual regret minimization is outside of the scope of this review, but in short it recursively calculates

how to minimize negative rewards of a particular strategy[24].

Once DeepStack looks far enough ahead with counterfactual regret minimization, it uses a neural network to estimate the value of different actions. It consisted of 7 fully connected hidden layers with 500 nodes and ReLU outputs. DeepStack was evaluated against 33 professional poker players from around the world and won 49.2 big blinds per 100 hands, which is an enormous win rate[24].

6 Conclusion

There has been substantial progress in the field of machine game play. Researchers have made substantial impacts to diverse fields of machine learning. Notably, reinforcement learning has had a renaissance of sorts with its new combination with deep neural networks. Monte Carlo tree search has bloomed since 2006 to become a dominant force in machine gaming. Several games in Go and poker that were previously thought to beyond our collective computational ability for several years to decades have been recently solved. That begs the question, where do researchers go from here? Bridge, a card game, remains and open question. AlpahGo could be improved. Other variants of poker could be considered. Every Atari game could be excelled at. The field of machine game play is wide open for exploration.

7 References

References

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, den D. van, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489N, Jan 28 2016. Copyright - Copyright Nature Publishing Group Jan 28, 2016; Document feature - ; Equations; Diagrams; Graphs; Tables; Last updated - 2016-02-08; CODEN - NATUAS.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533H, Feb 26 2015. Copyright - Copyright Nature Publishing Group Feb 26, 2015; Document feature - ; Graphs; Equations; Diagrams; Tables; Last updated - 2015-03-16; CODEN - NATUAS.

[3] Julien Pérolat, Florian Strub, Bilal Piot, and Olivier Pietquin. Learning nash equilibrium for general-sum markov games from batch data. *CoRR*, abs/1606.08718, 2016.

[4] V. Jojic. Basics of reinforcement learning.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[6] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[8] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[9] Brendan O'Donoghue, Rémi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. PGQ: combining policy gradient and q-learning. *CoRR*, abs/1611.01626, 2016.

[10] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[11] André Barreto, Rémi Munos, Tom Schaul, and David Silver. Successor features for transfer in reinforcement learning. *CoRR*, abs/1606.05312, 2016.

[12] Jürgen Schmidhuber and J Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990-2010). *IEEE transactions on autonomous mental development*, 2(3):230–247, 09 2010.

[13] Bradly C. Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *CoRR*, abs/1507.00814, 2015.

[14] Satinder Singh, S Singh, RL Lewis, AG Barto, and J Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE transactions on autonomous mental development*, 2(2):70–82, 06 2010.

[15] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016.

[16] Adam Henry Marblestone, Greg Wayne, and Konrad P Kording. Towards an integration of deep learning and neuroscience. *bioRxiv*, 2016.

[17] Ida Momennejad, Evan M. Russek, Jin H. Cheong, Matthew M. Botvinick, Nathaniel Daw, and Samuel J. Gershman. The successor representation in human reinforcement learning. *bioRxiv*, 2016.

[18] Cameron B. Browne, CB Browne, E Powley, D Whitehouse, and SM Lucas. A survey of monte carlo tree search methods. *IEEE transactions on computational intelligence and AI in games.*, 4(1):1–43, 03 2012.

[19] Marc Lanctot, Mark H. M. Winands, Tom Pepels, and Nathan R. Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. *CoRR*, abs/1406.0486, 2014.

[20] Sam Devlin, Anastasija Anspoka, Nick Sephton, Peter I Cowling, and Jeff Rollason. Combining gameplay data with monte carlo tree search to emulate human play. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[21] S. Ali Mirsoleimani, Aske Plaat, H. Jaap van den Herik, and Jos Vermaseren. A new method for parallel monte carlo tree search. *CoRR*, abs/1605.04447, 2016.

[22] Xiaoxiao Guo, Satinder Singh, Honglak Lee, and Richard Lewis. Deep learning for real-time atari game play using offline monte-carlo tree search planning. pages 3338–3346, 2014.

[23] Gareth MJ Williams. *Determining game quality through UCT tree shape analysis*. PhD thesis, MS thesis, Imperial Coll., London, 2010.

[24] Matej Moravcík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael H. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017.

[25] Yujing Hu, Yang Gao, and Bo An. Online counterfactual regret minimization in repeated imperfect information extensive games. *Jisuanji Yanjiu Yu Fazhan*, 51(10):2160–2170, 01 10.

[26] Daniel Dewey. *Learning What to Value*, pages 309–314. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.