Distributed Training for Reinforcement Learning

Christopher Sciavolino
Princeton University
`cds3@princeton.edu`

Abstract

Reinforcement learning (RL) has scaled up immensely over the last few years through the creation of innovative distributed training techniques. This paper discusses a rough timeline of the methods used to push the field forward. I begin by summarizing the problem of reinforcement learning and general solution methods. I then discuss the training environments used to evaluate model performance. I walk through a timeline of breakthroughs in distributed training used to scale up RL models, as well as other innovations in RL training. Finally, I take a look at exciting applications of distributed training processes in complex games like Go, Dota 2, and StarCraft II.

1 Introduction and Problem Space

Reinforcement learning is at the intersection of numerous fields like statistics, machine learning, neuroscience, and robotics. In this section, I provide a broad summary of reinforcement learning and analyze general methods for solving RL problems.

1.1 Reinforcement Learning Background

For the purposes of this survey, I consider a standard reinforcement learning (RL) setting where an agent interacts with an *environment* $\mathcal{E}$ over a set of discrete timesteps. Each timestep t, the agent receives the *state*, denoted $s_t \in \mathcal{S}$, and selects an *action* to take among all possible actions, denoted $a_t \in \mathcal{A}$. The agent selects an action at a particular state using its *policy* π, which maps from the state space to the action space $\pi : \mathcal{S} \rightarrow \mathcal{A}$. After selecting an action, the agent receives the next state s_{t+1} and a scalar *reward* r_t.

This process repeats until the environment's terminal state is reached, called the end of the *episode*. The *return* at any given timestep is defined as the cumulative future reward $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ with some discount factor $\gamma \in [0, 1]$. The goal is for the agent to maximize the expected return from each state s_t.

The *action-value function* $Q^\pi(s, a) = E[R_t | s_t = s, a_t = a]$ is the expected return after selecting action a in state s using policy π. The goal of many RL algorithms is to accurately model the optimal action-value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$, which uses a policy that selects actions maximizing the action-value function at any given state. The *state-value function* $V^\pi(s) = E[R_t | s_t = s]$ is the expected reward a policy π receives from a particular state s. A similar but slightly different goal would be to model the optimal state-value function $V^*(s) = \max_\pi V^\pi(s)$, which maximizes the expected return for any given state s.

The agent either has a model of the environment $\mathcal{E}$, or it chooses not to model the environment $\mathcal{E}$, called *model-free* RL. In general, the field uses model-free reinforcement learning to solve problems because the agent is generalizable past a specific environment and avoids complicated modeling considerations for the environment. In general, most solutions to RL problems fall into one of three categories: value-based methods, policy-based methods, or actor-critic methods.

Value-based model-free RL methods represent the action-value function using a parameterized function approximator with parameters denoted θ. Almost all of the methods I discuss in this paper vary the algorithm used to derive the parameters θ; however, they are all some form of deep neural network.

On the other hand, policy-based model-free methods attempt to directly model the policy $\pi(a|s; \theta)$, typically using gradient-based methods on the expectation of returns $E[G_t]$.

Actor-critic methods consider a learned estimate of the value function called the *baseline* $b_t \approx V^\pi(s)$. Using the baseline, the difference between the reward $Q(a_t, s_t)$ and the baseline b_t is an estimate of the *advantage* of action a_t in state s_t, formally defined as $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$. In this case, the policy π is the actor and baseline b_t is the critic. The goal is for the actor to maximize the value of the advantage.

Reinforcement learning takes two different forms: *on-policy* and *off-policy*. On-policy methods are summarized by David Silver by the phrase "learning while on the job," and approximate a policy π using experience drawn from an agent following policy π. Conversely, off-policy methods are summarized by the phrase "learning by looking over someone's shoulder," and approximate a *target policy* π using experience drawn from an agent following a different *behavior policy* μ.

The environment $\mathcal{E}$ is generally considered a Markov Decision Process (MDP), which is a quintuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$. $\mathcal{S}$ represents the state space, $\mathcal{A}$ represents the action space, $\mathcal{R}$ represents the reward function given a state-action pair, $\mathcal{P}$ represents the transition probability given a state-action-successor triple, and γ is the discount factor.

1.2 Baseline Deep RL Model: DQN

Next, I consider an important baseline model heralded as the first successful deep learning method applied to reinforcement learning problems: the Deep Q-Network (DQN) (Mnih et al., 2013).

DQN builds off of Q-learning, which iteratively applies updates based on the Bellman optimality equation:

$$Q_{t+1}(s, a) = E[r + \gamma \max_{a' \in \mathcal{A}} Q_i(s', a')|s, a]$$

where s' is the successor state, a' is an action taken at the successor state, and $Q_i(s, a)$ is the action-value function at iteration i. These value iteration algorithms are guaranteed to converge to the optimal action-value function as $i \to \infty$. A neural network representing a function approximator for Q with parameters θ, called a Q-network, trains by minimizing the loss function:

$$L_t(\theta_i) = E_{s,a \sim \mu(\cdot)}[(y_i - Q(s, a; \theta_i))^2]$$

where y_i is the target at iteration i defined as $y_i = E_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$, $\mu(s, a)$ is the alternate, behavior distribution, and θ_{i-1} are the fixed parameters from the previous iteration. Instead of computing full expectations, DQN performs stochastic gradient descent on the gradient of the loss function above. Updating the weights at each timestep and using single samples from the behavior distribution μ yields the famed Q-learning algorithm. Note that Q-learning is model-free, since it doesn't explicitly model the environment, and off-policy, since it samples experience from a behavior policy μ instead of the target policy π.

In previous work, most RL practitioners used a linear combination of features to approximate the Q function because neural networks have trouble converging to the optimal function. Instead of using an online network, DQN stabilizes convergence using an *experience replay*, which stores all of the agent's experiences within a given environment for each timestep. Each interaction is denoted as $e_t = (s_t, a_t, r_t, s_{t+1})$ and the replay memory is a dataset of interactions $\mathcal{D} = \{e_1, \ldots, e_N\}$ over many episodes. Each training iteration, DQN samples experiences from the replay and performs a gradient update using an ϵ-greedy action selection strategy. This strategy selects the greedy action with probability $(1 - \epsilon)$ and selects a uniformly random action with probability ϵ.

The DQN has several advantages compared to standard Q-learning. In standard Q-learning, each experience is only sampled once and subsequently forgotten. DQN samples each episode multiple times in order to learn as much as possible from each sample; therefore, DQN is significantly more data efficient. Standard Q-learning trains on sequential experiences which are highly correlated. By sampling randomly from the experience replay, the model breaks this correlation because it can select non-sequential experiences. Learning off-policy with the fixed previous parameters also helps stabilize the training process to converge to a local optimum.

2 Environments and Tasks

In order to properly learn, reinforcement learning agents need to interact with real environments and perform defined tasks. In this section, I describe a few of the tasks used to evaluate the effectiveness of trained agents.

2.1 Arcade Learning Environment (ALE)

DQN popularized the Arcade Learning Environment (Bellemare et al., 2013) (also called the Atari environment) as a robust testbed for reinforcement learning models. While DQN only evaluates on seven popular games (Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders), more recent systems evaluate using 57 different Atari games.

Most systems evaluate the effectiveness of the RL model on any particular game by comparing the agent's score to a human's score. This can be done in quite a few ways. One way would be to directly compare the human-level raw score to the agent's raw score; however, different games have different scoring mechanisms. For example, some games have scores in the thousands while others have top scores in the 10s. Additionally, some games incrementally provide rewards while others have large, infrequent reward structures.

Later methods use a *human-normalized score*, which describes performance on a percentage level. For example, if the median human obtains a score of 100 and the agent obtains a score of 80, then the human-normalized score would be 80%. In addition to the human score, the agent's score can be compared to the score from a random policy and the score from the optimal policy, which provide a good baseline and upper bound respectively.

2.2 DeepMindLab-30 (DMLab-30)

More recently, DeepMind released a set of 30 unique environments to test the effectiveness and generalizability of reinforcement learning agents (Beattie et al., 2016). Each level has different visual elements, some being bright and colorful while others are dark and gloomy. Each level has different tasks. Some levels require the agent to bring objects to a detector and others require the agent to learn to throw objects towards buttons. Some require the agent to act quickly while others require significant long-term planning.

Another interesting difference between tasks is the reward structure. Some tasks withhold rewards for long strings of actions, so agents must explore long sequences of actions before encountering any rewards.

3 Distributed Training for RL

While the DQN baseline demonstrates that deep reinforcement learning is possible, it is far from optimal and requires significant compute power to train. In this section, I present a timeline of systems aimed at scaling deep reinforcement learning to larger and more complex tasks through distributed training.

3.1 GORILA

Nair et al. (2015) propose the General Reinforcement Learning Architecture (GORILA) as one of the first distributed system architectures for deep reinforcement learning. GORILA is based on DistBelief (Dean et al., 2012), a distributed machine learning framework developed by Google prior to TensorFlow (Abadi et al., 2016). The GORILA system architecture is segmented into four parts: the actors, an experience replay memory, the learners, and the parameter servers.

The actors are servers tasked with interacting with a given environment $\mathcal{E}$. In the case of GORILA, there are N_{act} different processes all acting on N_{act} instances of the same environment. Each actor generates a set of experience trajectories $(s_1^i, a_1^i, r_1^i, s_2^i, \ldots, s_T^i, a_T^i, r_T^i)$ with T timesteps for actor i. Each actor has a replica of the central Q-network, which is periodically synchronized with the parameter servers.

The experience replay memory has two forms. A *local* replay memory stores experience for each actor on the actor's own machine. In this case, suppose the local memory is limited and can only hold M experience trajectories. A *global* replay memory aggregates experience from all actors onto a distributed database, which scales to meet memory needs at the cost of more communication overhead.

The learners are in charge of calculating gradients based on minibatches of experience samples from either the local or global experience replay memories. Each of these learners applies an off-policy RL algorithm (like DQN) from the parameter server to calculate a gradient update vector g_i. The learners communicate these gradient updates to the parameter server.

The parameter servers maintains a distributed representation of the current Q-network parameters. The parameter servers periodically receive gradients from the learners and update the central version of the Q-network using an asynchronous stochastic gradient descent algorithm. Every so often, the parameter servers push the updated model to both the actors and the learners in the system.

When using GORILA with the DQN algorithm, the system is able to outperform DQN in the ALE with half the training time. The improved performance of GORILA DQN is attributed to the increased number of states visited by N_{act} parallel actors compared to just a single actor in vanilla DQN.

3.2 A3C

Like GORILA, Mnih et al. (2016) propose an asynchronous RL framework, but with a few key differences. First, rather than having separate machines for actors, learners, and parameter servers, the authors instead use multiple CPU threads on a single machine. Second, they remove the experience replay mechanism from the framework, which allows on-policy training algorithms. Third, instead of each actor interacting with the environment using the same policy, each actor uses a different exploration policy. The use of multiple exploration policies replaces the experience replay's role in stabilizing training.

Rather than just using Q-learning, the authors investigate one-step Q-learning, one-step Sarsa, n-step Q-learning, and advantage actor-critic in their asynchronous RL framework proposed above. Since the advantage actor-critic algorithm performs best, I focus on this particular algorithm.

Asynchronous advantage actor-critic (A3C) maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V^{\pi}(s_t;\theta_v)$. The actor-critic operates using a mix of n-step returns to update both the value function and the policy. Every t_{max} actions (or terminal state), the algorithm performs an update:

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')A(s_t,a_t;\theta,\theta_v)$$

where $A(\cdot)$ is an estimate of the advantage function,

$$\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta_v)$$

For their implementation, the authors add entropy to the policy π in order to encourage further exploration instead of early convergence at poor local optima.

When evaluated on the ALE, A3C achieves significantly better results compared to GORILA. Additionally, A3C trains much faster than GORILA and DQN, even though it uses CPUs instead of multiple machines or GPUs. The authors attribute A3C's success to multiple actor-learners updating a shared model, which has a stabilizing effect during the learning process. By removing the experience replay mechanism, their asynchronous RL framework allows for on-policy algorithms to be used, which is a significant contribution in the field.

3.3 IMPALA

Importance Weighted Actor-Learner Architecture (IMPALA) (Espeholt et al., 2018) aims to tackle training a single RL agent to perform well across a large collection of tasks. While the A3C framework is very successful at training a single agent to solve a small set of tasks, the agent typically needs to digest millions or billions of frames over multiple days to master a single domain. This approach is not scalable to produce a single agent that generalizes to a large collection of environments.

IMPALA adopts the actor-critic setup to learn a policy π and a baseline value function approximation V^{π}. IMPALA decouples the actors and the learners. The actors keep track of a local policy μ and interact with their own instance of the environment. The learners continuously update their policy π on batches of trajectories collected from many actors. One notable difference is the learners are updating parameters based on trajectories, not pre-computed gradients (as in GORILA). At the start of each trajectory, the actor updates its local policy μ to match the learner's policy π. Since the learner's policy is likely several updates ahead of the experiences drawn from μ, there is a *policy lag* between the actors and learners. IMPALA introduces an off-policy correction algorithm called V-trace to fix this discrepancy.

V-trace uses truncated importance sampling weighting to approximate the state-value function $V(x_s)$. Formally, the n-step V-trace target is defined as:

$$v_s = V(x_s) + \sum_{s+n-1}^{t=s} \gamma^{t-s}(\prod_{i=s}^{t-1} c_i)\delta_t V$$

where $\delta_t V = \rho_t(r_t + \gamma V(x_{t+1}) - V(x_t))$ is the temporal difference for V, and $\rho_t = \min(\bar{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$ and $c_i = \min(\bar{c}, \frac{\pi(a_i|x_i)}{\mu(a_i|x_i)})$ are truncated importance sampling weights. One useful property is that for an on-policy case (where $\pi = \mu$), the n-step V-trace update reduces to the n-step Bellman target. Thus, the authors can use V-trace for both on-policy and off-policy learning.

Using IMPALA, the frame processing throughput increases from 50k (A3C) to 250k (IMPALA). The increased throughput allows IMPALA to be trained on multiple tasks in a more reasonable amount of time. When evaluated on the DMLab-30 environment, IMPALA using a deep neural network and population-based training (Jaderberg et al., 2017) achieves a human-normalized average score of 49.4%, which significantly outperforms A3C (23.8%).

3.4 APE-X

APE-X (Horgan et al., 2018) extends the idea of prioritized experience replay (Schaul et al., 2015) to a distributed setting. The APE-X framework again decouples actors from learners and uses an experience replay prioritized using a scalar value for each experience.

The actors, which can be distributed across multiple machines, each have their own policy and instance of the environment. The actors interact with the environment and generate experience trajectories. Rather than arbitrarily setting the priority for a trajectory in the experience replay, APE-X requires the actor to come up with an appropriate priority for each trajectory. Both the priority and trajectory are communicated to the experience replay.

The learner continuously samples experiences from the experience replay, selecting higher priority experiences first. Once sampled, the learner updates the priority of each experience using the obtained gradient update. Typically, when an experience is sampled, the priority decreases because the relative usefulness of experience decays the more it is used. Every so often, the learner communicates updated network parameters to the actors. Note that in theory, the learners can be distributed across multiple machines; however, the authors choose to only distribute the actors, keeping both the replay and the learner centralized.

When evaluated on the Arcade Learning Environment, APE-X DQN and APE-X DPG (Silver et al., 2014) (two instances of APE-X using different learning algorithms) both outperform previous methods and decrease training time. The training time of APE-X also decreases as the number of actors increases, likely due to more exploration and better quality experiences. The authors hypothesize the success is due to important experiences being sampled more often than extraneous experiences.

3.5 R2D3

Recurrent Replay Distributed DQN from Demonstrations (R2D3) (Paine et al., 2019) attempts to tackle three difficult problems in RL: (1) sparse rewards that require long sequences of actions before receiving a reward; (2) partial observability, where the agent only observes a part of the environment at each timestep; and (3) highly variable initial conditions, where different initializations can drastically affect model performance. In contrast to previous methods, R2D3 attempts to learn from *demonstrations*, or experience provided ahead of time to the system (like recordings of human gameplay).

The R2D3 system runs several actor processes, each with a separate agent policy and instance of the environment. The actors record trajectories and initial priorities into an *agent* replay buffer, just like in the APE-X framework. R2D3 also has a prioritized *demo* buffer, which contains a prioritized list of demonstrations the system can also replay. A central learner process samples batches of experiences from both the agent and demo replay buffers proportional to a hyperparameter ρ. For a batch of size B, the learner samples ρB experiences from the demo replay and $(1-\rho)B$ experiences from the agent replay. The learner uses n-step double Q-learning (Hasselt, 2010) and a dueling architecture (Wang et al., 2016). The learner periodically pushes updated model parameters to each of the actor processes.

The authors propose a test suite of 8 tasks (dubbed the "Hard-Eight") specifically designed to have sparse rewards, partial observability, and highly variable initial conditions. Previous models evaluated on this test suite are unable to obtain any reward, failing the tasks outright.

When evaluating R2D3 on this suite, the agent is able to learn and conquer six of the eight tasks. To do this, the agent effectively learns from the expert demonstrations provided and, in some tasks, learns a series of actions better than the human experts.

4 Other Training Regimes

In this section, I describe other important systems and training algorithms that are not necessarily distributed in nature, but still improve the state-of-the-art reinforcement learning methods.

4.1 Rainbow

Rainbow (Hessel et al., 2017) examines 6 different extensions to the DQN algorithm and analyzes their relative importance. The six extensions included in this study are Double DQN (van Hasselt et al., 2015), prioritized experience replay (Schaul et al., 2015), the dueling network architecture (Wang et al., 2016), multi-step learning (Sutton and Barto, 1998), distributional Q-learning (Bellemare et al., 2017), and noisy nets (Fortunato et al., 2017). The authors combine each of these improvements into a single model dubbed Rainbow and perform an extensive ablation study.

The Double DQN (DDQN) addresses overestimation bias in Q-learning by decoupling selection and evaluation of the bootstrapped action. Prioritized experience replay improves data efficiency by replaying useful transitions more often. The dueling network architecture helps generalize the model's understanding across similar states. Multi-step learning propagates rewards to earlier states faster and provides a way to shift the bias-variance trade-off. Distributional Q-learning learns a distribution of returns rather than estimating a single scalar value. Noisy DQN (noisy nets) uses stochastic network layers to support more robust exploration.

By combining each of these improvements into a single model, Rainbow significantly outperforms each individual model on the Arcade Learning Environment. By ablating each of the features in Rainbow, the authors are able to identify the relative importance of each. The authors identify that prioritized experience replay and multi-step learning are the two most crucial components in Rainbow, leading to significant performance drops when removed. The next most important component is distributional Q-learning, which lags behind Rainbow as training continues past 50M frames when removed. For noisy nets, the model performance dips slightly, but does not drop nearly as much as features like prioritized replay. The two least important features in Rainbow are DDQN and the dueling network architecture, which do not lead to significant performance differences when ablated.

4.2 Population-based Training

Population-based Training (PBT) (Jaderberg et al., 2017) is an asynchronous optimization algorithm aimed at achieving optimal model performance in addition to well-tuned hyperparameters. Notably, PBT can discover a *schedule* for hyperparameters, rather than simply finding optimal values.

First, the authors define a function `eval` that evaluates an objective function on a model with parameters θ. The optimal set of parameters would thus be:

$$\theta^* = \arg\max_{\theta \in \Theta} \texttt{eval}(\theta)$$

The authors also define an iterative step function `step` to update the parameters of the model conditioned on some hyperparameters $h \in \mathcal{H}$,

$$\theta_{t+1} \leftarrow \texttt{step}(\theta_t | h_t)$$

In total, the optimization function is

$$\texttt{step}(\texttt{step}(\texttt{...step}(\theta|h_1)\ldots|h_{T-1})|h_T)$$

The above only evaluates the parameters with respect to a specific set of hyperparameters. The authors identify well-tuned hyperparameters by optimizing many models using different sets of hyperparameters. The authors define a *population* $\mathcal{P}$ containing N models $\{\theta^i\}_{i=1}^N$ using different hyperparameters $\{h^i\}_{i=1}^N$. The goal is to find the optimal parameters among the models in the population.

Each worker in the population optimizes its parameters given its hyperparameters using the `step` function asynchronously until the worker is deemed "ready." At this point, the worker can call one of two methods: `exploit` gives the worker the option to copy the parameters and hyperparameters from the best-performing worker in the population; `explore` randomly perturbs the hyperparameters with some noise. Note that the specific implementation of `exploit` and `explore` can be replaced with an application-specific definition. Similarly, the frequency a worker stochastically calling `exploit` or `explore` can also be set by the application.

For the purposes of this study, one implementation of `exploit` randomly samples another worker from the population and copies the higher performing parameters. Another potential implementation could be to stochastically copy the current best-performing parameters in the population or do nothing. One common implementation of `explore` used in the study multiplies each hyperparameter by a random number in the range $[0.8, 1.2]$ to alter the value slightly.

When applied to real domains like DeepMind Lab, Atari, and StarCraft II, PBT increases final agent performance. In most applications, PBT identifies hyperparameter schedules that are significantly better than human hand-tuned schedules. An interesting point the authors make is that PBT occasionally learns complicated schedules that match well-studied, hand-crafted hyperparameter schedules.

5 Applications of Distributed Training

The algorithms described in Sections 3 and 4 may seem abstract; however, they have led to significant advances in real reinforcement learning applications. When evaluated on many different games, reinforcement learning models achieve superhuman performance across a variety of tasks. In this section, I discuss some of the applications RL agents have conquered to date.

5.1 AlphaGo

The game of Go has long been described as one of the most complex and difficult games for AI to tackle. Go has an enormous search space on the order of 10^{170} and requires being very long-term goal-oriented. AlphaGo (Silver et al., 2016) is a reinforcement learning agent that approaches Go using a value network to evaluate board positions and a policy network to select moves.

The authors use a combination of supervised learning and self-play to teach AlphaGo how to play. Initially, the agent has no idea how to effectively play the game, so self-play would require an enormous amount of training to reach even beginner level. Instead, AlphaGo looks through previous games played by experts and attempts to mimic their moves using supervised learning. Using this approach, AlphaGo is able to reach a reasonable starting point before attempting to learn on its own.

AlphaGo then learns to update its policy and value function by playing against itself millions of times. During training, AlphaGo uses Monte-Carlo tree search (MCTS) to identify actions leading to boards with positive outcomes. At inference time, AlphaGo fuses the policy and value networks with a Monte-Carlo tree search algorithm where each edge stores an action value, a visit count, and a prior probability. The tree is traversed using simulation starting from the current root state. At each timestep, the agent selects an action from a state that maximizes the sum of the action-value function and a special bonus inversely proportional to how frequently the successor state is seen (to encourage robust exploration).

Performing this traversal requires significant computation overhead, so AlphaGo uses asynchronous multi-threaded search where simulations are executed on CPUs and neural network evaluation is executed on GPUs. The authors also implement a distributed version of AlphaGo, which exploits significantly more compute power across many machines.

When compared to other Go programs like Crazy Stone (Coulom, 2006), Zen, Fuego (Enzenberger et al., 2010), and Pachi (Baudiš and Gailly, 2012), AlphaGo performs extraordinarily well, winning 494/495 games (99.8% win rate). Even after handicapping AlphaGo by giving the opponent 4 free moves, AlphaGo still wins 77%, 86%, and 99% of the time against Crazy Stone, Zen, and Pachi respectively. The distributed version of AlphaGo is even stronger, defeating single computer AlphaGo in 77% of games.

In 2015, AlphaGo competed against Fan Hui, the European Go champion, and won 5 to 0 in a five game match.

5.2 AlphaZero

While AlphaGo achieved superhuman performance, it was later succeeded by a more robust AlphaGo Zero (Silver et al., 2017b) which achieved better performance without human demonstrations by playing against the original AlphaGo. However, both AlphaGo and AlphaGo Zero are agents solely trained for Go.

The goal of AlphaZero (Silver et al., 2017a) is to achieve superhuman performance across many *different* domains. Specifically, AlphaZero achieves superhuman performance on chess, shogi (Japanese chess), and Go, all with significantly less training compared to previous programs.

AlphaZero uses deep neural networks to model a reinforcement learning algorithm with no priors (commonly referred to as "tabula rasa"). Similar to AlphaGo Zero, the AlphaZero network models a policy function to select actions and a state-value function to estimate the favorability of being in a particular position. In contrast to AlphaGo, AlphaZero learns its policy and value function purely from self-play as opposed to supervised learning. AlphaZero uses Monte-Carlo tree search (MCTS) to consider multiple trajectories of play. The training process updates the network parameters in order to minimize the error between the predicted outcome at any given state and the actual game outcome. AlphaZero performs gradient descent over a loss function that sums the mean-squared error and cross-entropy loss for the value function and policy respectively.

While AlphaGo Zero estimates the probability of winning as a binary win/loss, AlphaZero accounts for other potential outcomes, like draws in the game of chess. AlphaGo Zero also augments its training data using 8 symmetric forms of the same board (since boards can be translated in Go). Instead, AlphaZero uses the raw board and does not transform or augment the board positions during MCTS.

Training for AlphaZero happens in a tournament-like fashion. A challenger plays against the best player for a series of rounds. If the challenger wins more than 55% of games against the best player, the challenger becomes the best player. This process continues until a player can no longer be bested (ideally close to the optimal solution). One note is that AlphaZero trains on an *enormous* amount of compute power: 5,000 TPUs for 700,000 steps with minibatches of size 4,096!

That said, AlphaZero is extraordinarily successful, surpassing AlphaGo Zero (Go), Stockfish (chess), and Elmo (shogi) with comparably less training. The authors also show that AlphaZero is robust across initializations, besting Stockfish using multiple different popular chess openings.

5.3 OpenAI Five

While chess and Go are extraordinarily complex games, they are nothing compared to the video games played across the world today. In that spirit, OpenAI decided to take AI to the next level by creating OpenAI Five (OpenAI et al., 2019), a reinforcement learning agent for Dota 2, a complex, popular online video game. Video games of this level are particularly challenging due to extremely high-dimensional observation and action spaces, partial observability, and incredibly long time horizons (each game lasts around 45 minutes).

OpenAI Five supports a subset of playable "heroes" within the Dota 2 game. Instead of taking in the pixels on the screen, the agent uses an imperfect approximation and takes in a set of data arrays describing the current game state. From these observations, a core 4096-unit LSTM is able to model policy outputs, namely the actions and a value function.

The agent is trained using Proximal Policy Optimization (Schulman et al., 2017) with Generalized Advantage Estimation (GAE) (Schulman et al., 2015). The policy training uses collected self-play experience to learn from experienced actions. The agent with the latest policy plays against itself 80% of the time and against previous versions 20% of the time.

One of the most interesting parts about using RL agents on games like Dota 2 is the game constantly evolves through updates, patches, and bug fixes. Thus, every two weeks or so, the game environment is altered, potentially significantly. In order to use agents trained on previous environments, the authors introduce the idea of *surgery*. The goal of surgery is to transfer as much of the knowledge from the old agent into a new agent compatible with the updated environment. Over the ten month lifetime of OpenAI Five, the authors perform over twenty surgeries (successful and unsuccessful) to avoid unnecessarily re-training a new agent from scratch.

In April 2019, OpenAI Five competed against Team OG, the reigning Dota 2 world champions, and defeated them 2-0 in best-of-three match. Afterwards, OpenAI ran OpenAI Five Arena, where the public was able to play against OpenAI Five for three days. In total, OpenAI Five dominated, winning 99.4% of the thousands of games played.

5.4 AlphaStar

Concurrent to OpenAI Five, the team at Google DeepMind tackled the game of StarCraft II using deep reinforcement learning. The proposed agent AlphaStar (Vinyals et al., 2019) hoped to achieve superhuman performance without using hand-crafted subsystems, game simplifications, or unfair advantages. For example, StarCraft II players have a minimum reaction time of a few hundred milliseconds. Instead of making instantaneous moves, AlphaStar uses network latencies, computation time, and maximum actions-per-minute (APM) limitations to mimic input delay.

AlphaStar uses a combination of supervised learning and reinforcement learning during training. Each agent is initialized using supervised learning from a sampled set of human replays where the agent learns to play like the human. After being exposed to various human strategies, the agent uses self-play reinforcement learning to build on the baseline's performance. The algorithm itself is based on advantage actor-critic and uses an off-policy approach to learn from experience generated by previous policies.

During training, an agent plays against either a version of itself (main agents), an agent that is trained to exploit weaknesses in main agents (main exploiter agents), or an agent trained to exploit weaknesses in agents across the league (league exploiter agents). Each agent trains over the course of 44 days using 32 third-generation TPUs to reach grandmaster level.

In December 2018 and January 2019, AlphaStar defeated two top professional StarCraft II players, winning each match 5-0.

6 Conclusion

In this paper, I discuss many techniques to tackle distributed training for reinforcement learning. I describe a timeline of advancements built on top of one another allowing current RL methods to flourish. I finally take the knowledge from theory and discuss real-world agents built using the advancements in distributed RL to defeat humans in complex games like Go and StarCraft II.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

Petr Baudiš and Jean-Loup Gailly. 2012. Pachi: State of the art open source go program. volume 7168.

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. 2016. Deepmind lab.

Marc G. Bellemare, Will Dabney, and Rémi Munos. 2017. A distributional perspective on reinforcement learning.

Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Int. Res.*, 47(1):253–279.

Rémi Coulom. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, page 72–83, Berlin, Heidelberg. Springer-Verlag.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *NIPS*.

M. Enzenberger, M. Müller, B. Arneson, and R. Segal. 2010. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. 2017. Noisy networks for exploration.

Hado van Hasselt, Arthur Guez, and David Silver. 2015. Deep reinforcement learning with double q-learning.

Hado V. Hasselt. 2010. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2017. Rainbow: Combining improvements in deep reinforcement learning.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed prioritized experience replay. In *International Conference on Learning Representations*.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population based training of neural networks.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning.

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. 2015. Massively parallel methods for deep reinforcement learning.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with large scale deep reinforcement learning.

Tom Le Paine, Caglar Gulcehre, Bobak Shahriari, Misha Denil, Matt Hoffman, Hubert Soyer, Richard Tanburn, Steven Kapturowski, Neil Rabinowitz, Duncan Williams, Gabriel Barth-Maron, Ziyu Wang, Nando de Freitas, and Worlds Team. 2019. Making efficient use of demonstrations to solve hard exploration problems.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, 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. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017b. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.

Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning*, 1st edition. MIT Press, Cambridge, MA, USA.

Oriol Vinyals, Igor Babuschkin, Wojciech Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John Agapiou, Max Jaderberg, and David Silver. 2019. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA. PMLR.